IBM eserver

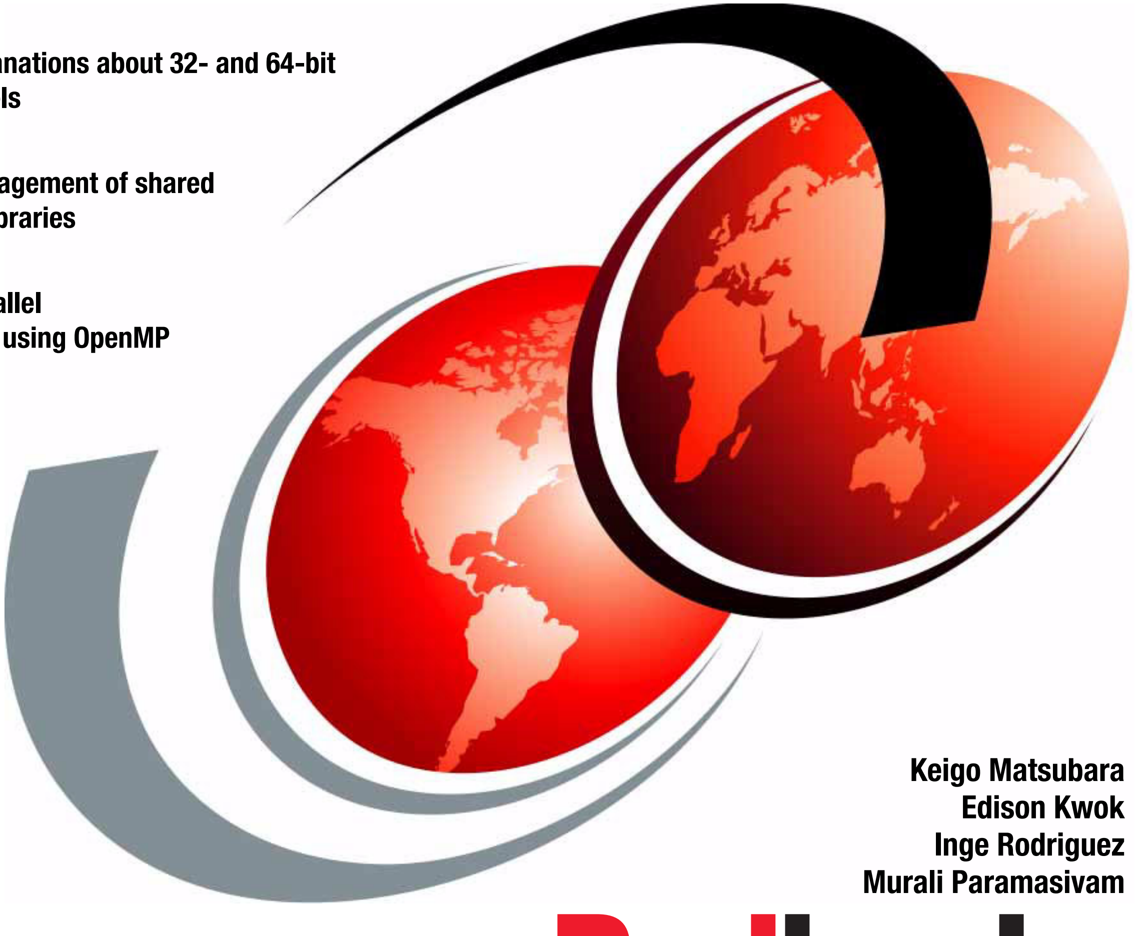
IBM

# Developing and Porting C and C++ Applications on AIX

Detailed explanations about 32- and 64-bit process models

Effective management of shared objects and libraries

Exploring parallel programming using OpenMP



Keigo Matsubara
Edison Kwok
Inge Rodriguez
Murali Paramasivam

Redbooks

ibm.com/redbooks

International Technical Support Organization

# Developing and Porting C and C++ Applications on AIX

June 2003

SG24-5674-01

**Note:** Before using this information and the product it supports, read the information in "Notices" on page xvii.

**Second Edition (June 2003)**

This edition applies to C for AIX (program number 5765-F57) and VisualAge C++ for AIX Version 6.0 (product number 5765-F56) installed on AIX 5L Version 5.2 (product number 5765-E62).

# Contents

# Figures

# Tables

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
*IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.*

*The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law*: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

# Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- AIX 5L™
- AIX®
- C Set ++®
- @server ™
- @server™
- IBM eServer™
- IBM®
- Open Class®
- OS/2®
- PartnerWorld®
- POWER4™
- PowerPC®
- pSeries™
- Redbooks (logo)™
- Redbooks™
- RISC System/6000®
- VisualAge®
- zSeries™

The following terms are trademarks of other companies:

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product and service names may be trademarks or service marks of others.

# Preface

This IBM Redbook will help experienced UNIX application developers who are new to the AIX operating system. The book explains the many concepts in detail, including the following:

- Enhancements and new features provided by the latest C and C++ compilers for AIX
- Compiling and linking tasks required to effectively use and manage shared libraries and run-time linking
- Use of process heap and shared memory in the 32- and 64-bit user process models
- A new programming paradigm in a partitioned environment where resources can be dynamically changed
- Parallel programming using POSIX threads and OpenMP

The following chapters are also useful for system administrators who are responsible for the software problem determination and application software release level management on AIX systems:

- Chapter 3, “Understanding user process models” on page 105
- Chapter 7, “Debugging your applications” on page 249
- Chapter 12, “Packaging your applications” on page 405

This publication expands on the information found in the *AIX 5L Porting Guide*, SG24-6034.

## The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

**Keigo Matsubara** is an advisory IT specialist at the International Technical Support Organization (ITSO), Austin Center. Before joining the ITSO, he worked in the System and Web Solution Center in Japan as a Field Technical Support Specialist (FTSS) for pSeries™. He has been working for IBM® for 11 years.

**Edison Kwok** is a senior Software Developer in the IBM Toronto Software Laboratory in Canada. He has eight years of experience in C and C++ compiler development on the zSeries™ and pSeries platforms. He holds a degree in electrical engineering from the University of Victoria. His area of expertise include compiler construction, C language standard, and C and C++ programming on various UNIX operating systems and the mainframe.

**Inge Rodriguez** is an IT specialist from IBM Germany. She has 20 years of experience in UNIX application development. She has been working for IBM for three years. Her main responsibility is support for ISVs regarding application development and porting. She holds a MSc in Medical Computer Science of University Heidelberg.

**Murali Paramasivam** is a Software Engineer from IBM India. He has nearly three years of application development experience in C and C++ on various UNIX operating systems. He holds an engineering degree in Material Science and Metallurgy. His areas of expertise include shared libraries, multi-threaded programming, and C/C++ compilation and linking concepts in AIX®.

Thanks to the following people for their contributions to this project:

**International Technical Support Organization, Austin Center**
Scott Vetter and Wade Wallace

**IBM Austin**
Alfredo Mendoza, Ann Wigginton, Betty Riggle, Donald Stence, David Hepkin, Gary Hook, Joel H Schopp, Julie Craft, Kedron J Touvell, Kenji Kindo, Kevin W Monroe, Luke Browning, Mark Rogers, Michael Mall, Nathan Fontenot, Randy Swanberg, Richard Cutler, Sara D Epsztein, Steven Molis

**IBM Japan**
Hajime Mita and Tomoyuki Niijima

**IBM Toronto**
Steven E. Hikida, Wang Chen, Sean Perry, Roger E. Pett.

## Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

**ibm.com**/redbooks/residencies.html

## Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  **ibm.com**/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. JN9B Building 003 Internal Zip 2834
  11400 Burnet Road
  Austin, Texas 78758-3493

# Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-5674-01
for Developing and Porting C and C++ Applications on AIX
as created or updated on March 25, 2009.

## June 2003, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

### New information

The following chapters are new:

- Chapter 2, “Compiling and linking” on page 37
- Chapter 3, “Understanding user process models” on page 105
- Chapter 4, “Managing the memory heap” on page 165
- Chapter 5, “Creating DLPAR-aware applications” on page 207
- Chapter 6, “Programming hints and tips” on page 219
- Chapter 7, “Debugging your applications” on page 249
- Chapter 9, “Program parallelization using OpenMP” on page 335
- Chapter 12, “Packaging your applications” on page 405

### Changed information

The following chapters were rewritten in order to cover new features and enhancements provided by the latest products:

- Chapter 1, “C and C++ compilers” on page 1
- Chapter 8, “Introduction to POSIX threads” on page 275

### Unchanged information

The following chapters and appendixes are unchanged but reviewed again:

- Chapter 10, “Dealing with C++ templates” on page 377
- Chapter 11, “Creating shared objects from C++ source codes” on page 393
- Appendix A, “Previous versions of C and C++ compiler products” on page 429
- Appendix E, “Supported IBM SMP directives” on page 483

## September 2000, First Edition

The first version of this book, *C and C++ Application Development on AIX*, SG24-5674 was written by the following authors:

Richard Cutler, Francois Armingaud, Eduardo Conejo, Kumaravel Nagarajan

The following list shows contributors for the first version of this book, *C and C++ Application Development on AIX*, SG24-5674:

**IBM Toronto**
Derek Truong, Mark Changfoot, Paul Pacholski, Rene Matteau

# 1

# C and C++ compilers

This chapter focuses on the latest versions of the IBM C and C++ compiler products for AIX: C for AIX Version 6.0 and VisualAge C++ for AIX Version 6.0. The latest compiler products offer enhanced support in optimizations, POWER4™ architecture exploitation, the latest ISO C and C++ Standard conformance, as well as compatibility features targeted to GNU C/C++ portability, explained in the first two sections:

- Section 1.1, “C for AIX Version 6.0” on page 2
- Section 1.2, “VisualAge C++ for AIX Version 6.0” on page 16

The other sections provide a comprehensive guide to installing and configuring the compiler products on your AIX systems:

- Section 1.3, “Installing the compilers” on page 19
- Section 1.4, “Activating the compilers” on page 23
- Section 1.5, “Activating the LUM server” on page 26
- Section 1.6, “Enrolling a product license” on page 27
- Section 1.7, “Invoking the compilers” on page 29
- Section 1.8, “Where to find help” on page 31

For a description of the previous versions of IBM C and C++ compiler products, please refer to Appendix A, “Previous versions of C and C++ compiler products” on page 429 for details.

## 1.1 C for AIX Version 6.0

The C for AIX Version 6.0 compiler is the latest IBM C compiler product available on AIX. It offers several new enhancements over the previous versions, particularly in the area of optimization features and new PowerPC® architecture support. This compiler is supported on AIX Version 4.3.3 or later.

C programs written using Version 4 or 5 of IBM C for AIX are source compatible with IBM C for AIX Version 6.0. C programs written using either Version 2 or 3 of IBM Set ++ for AIX or the XL C compiler component of AIX Version 3.2 are source compatible with IBM C for AIX Version 6.0 with exceptions to detect invalid programs or areas where results are undefined. Source compatibility, however, does not guarantee a program will perform in an identical manner; new option defaults can sometimes influence how a program behaves. Always consult the official documentation when migrating to a new version of the product.

If installed, the compiler is installed under /usr/vac by default and uses the /etc/vac.cfg configuration file. To install to an alternate directory, or to retain the installation of a previous version of C for AIX compiler, refer to 1.3.2, “Retaining a previous version of the compiler” on page 22.

The C for AIX Version 6.0 compiler uses the LUM licensing system, which is explained in the following sections, to control usage of the product.

- Section 1.4, “Activating the compilers” on page 23
- Section 1.5, “Activating the LUM server” on page 26
- Section 1.6, “Enrolling a product license” on page 27

### 1.1.1 New or improved optimization features

A number of optimization features have been introduced or improved in C for AIX Version 6.

#### Interprocedural analysis

Interprocedural analysis, or IPA, is an optimization performed across function boundaries. In a traditional compilation, only intraprocedural analysis is done, where each function is optimized individually within a single compilation unit. IPA takes optimization one step further by analyzing all functions in the entire application.

In addition to the usual optimizations performed by the optimizer, IPA also performs many optimizations interprocedurally, including:

- Inlining across compilation units
- Program partitioning
- Global variables coalescing
- Code straightening
- Dead code elimination
- Constant propagation
- Copy propagation

Keep in mind that because the compiler is performing extra processing with -qipa, compilation time is expected to increase. However, with C for AIX Version 6.0, a new suboption, -qipa=threads, has been introduced to take advantage of multi-threaded interprocedural analysis. You can also specify -qipa=threads=*N*, where $N$ is the number of threads used by the compiler for IPA analysis and code generation. Please refer to *C for AIX Compiler Reference*, SC09-4960 for more details.

### Profile-directed feedback

With profile-directed feedback, or PDF, special instrumentation code is inserted in the executable to gather information about the program's execution pattern. Information gathered from the execution is then fed back to the compiler for a second compilation, and the compiler performs optimization based on the code execution frequency and conditional branch pattern.

In order to gain the most using this feature, make sure the program execution is performed as close to the intended conditions as possible. That is, choose input parameters and a data set that are representative and meaningful.

Only use PDF towards the very end of a development cycle, where the program is fully functional at a high optimization level.

### New options and pragmas

C for AIX Version 6.0 introduces several new performance related options and pragmas:

- -qarch=pwr4
- -qtune=pwr4
- -qhot
- -qlargepage

- -qsmallstack
- -qunwind
- -qtocmerge
- -qreport
- -qipa=threads=N
- #pragma execution_frequency
- #pragma pack
- #pragma snapshot

**Note:** The -qarch=pwr4 and -qtune=pwr4 options are used to generate executable files optimized and tuned for the POWER4 processor.

For a detailed description of these options, please refer to the *C for AIX Compiler Reference*, SC09-4960. For detailed explanation about the tuning considerations on the POWER4 processor, please refer to *The POWER4 Processor Introduction and Tuning Guide*, SG24-7041.

### New built-in functions

Compiler built-in functions are often provided to allow programmers direct access to features or machine code instructions on the hardware architecture. They are directly mapped to hardware instructions, hence any overhead associated with function calls (for example, parameter passing, stack allocation and adjustment, and so on) are completely eliminated. Please refer to *C for AIX Compiler Reference*, SC09-4960 for a list of built-in functions supported in C for AIX Version 6.0.

## 1.1.2 ISO C Standard conformance

The IBM C for AIX Version 6.0 compiler supports the latest ISO/IEC 9899:1999 International Standard, commonly referred to as C99. C99 includes many new features and enhancements to the original ISO/IEC 9899:1990 International Standard (C89), which extends the capability of the C language. We will discuss some useful features defined in this standard.

### _Bool type

Similar to the C++ bool type, C99 supports _Bool type in addition to the long list of type specifiers already in the original C89 standard. Applications no longer need to define their own macros, as the system header file stdbool.h already defines the macros *true* and *false*.

## long long data type

Unlike the -qlonglong option in the previous versions of the C for AIX compilers, the addition of the long long data type in C99 changes the semantics for integer constants. In C89, the type of an unsuffixed integer constant is either int, long int, or unsigned long int, whichever is large enough to represent the constant. However in C99, unsuffixed integer constants have type int, long int, or long long int instead. To illustrate this difference, consider the following example:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    printf("sizeof(2147483648) = %d\n", sizeof(2147483648));
    exit(0);
}
```

In the above example, the constant 2147483648 is one greater than LONG_MAX. When compiled with -qlanglvl=stdc89, the execution of this example will print the value 4, since the type selected for the constant is unsigned long int. With -qlanglvl=stdc99, on the other hand, the result will be 8, and the constant will have long long int type.

## Complex data types

C99 introduces native complex data types to the C language. There are three complex types: float _Complex, double _Complex, and long double _Complex, as well as three pure imaginary types: float _Imaginary, double _Imaginary, and long double _Imaginary. Collectively they are called complex floating types. Each complex type is logically the same as an array of two elements of the corresponding real floating type, where the first element is the real part of the complex number, and the second element the imaginary part. Therefore, the size of a complex type is double the size of its corresponding floating type.

The following shows how to declare a complex variable c, and initialize it to {1.0, 2.0i}:

```
double _Complex c = 1.0 + 2.0 * __I;
```

Basic arithmetic operators are supported natively in the language. There are also mathematical functions provided by the run-time library, by including the system header file complex.h. For more information on the semantics of the complex data types, please refer to *C for AIX C/C++ Language Reference*, SC09-4958.

## inline function specifier

Function inlining reduces function call overhead, as well as allowing the optimizer to perform better optimizations at or near the function call site. The compiler already does inline optimization with the -O option, so the use of this feature may not provide any further performance benefits.

### restrict qualifier

If an object is modified through a restrict qualified pointer, than all access to that object must be based on, directly or indirectly, the same pointer, that is, no other pointers will access the object. This allows the compiler to perform better optimization. Please refer to the *C for AIX C/C++ Language Reference*, SC09-4958 for more details.

### static keyword in array declaration

In a function declaration, array parameters are generated as pointers to the array element type. For example:

```
void func(int arr[])
{
    ...
}
```

and

```
void func(int *arr)
{
    ...
}
```

are equivalent declarations. In C99, you can use the storage class specifier *static* in an array parameter declaration, to indicate to the compiler that the argument in the function call is guaranteed to be non-NULL, and contains at least the specified number of elements. For example:

```
void vector_add(int a[static 10], const int b[static 10])
{
    int i;
    for (i = 0; i < 10; i++)
        a[i] += b[i];
}
```

With this extra information, the compiler will be able to apply better optimization analysis and generate faster performing code.

### Universal character name

Universal characters support, already available since C for AIX Version 5 with the -qlanglvl=ucs option, is now part of the C99 standard. It is used to write characters that are not in the basic character set. You can have universal characters in identifiers, string literals, and comments.

## __func__

Similar to the C for AIX compiler predefined macro __FUNCTION__, __func__ is a compiler generated internal variable that has the following declaration:

```
static const char __func__[] = “function_name”;
```

where *function_name* is the name of the current function where __func__ is referenced. This is useful in writing debug code. See “Function-like macros with variable arguments” on page 9 for an example usage.

## Hexadecimal floating point constant

Just as you can use hexadecimal integer constants to represent exactly the binary format of an integer, C99 allows you to have floating point constant specified in hexadecimal format. For example:

```
double d = 0x123.abcp+10;
```

## Variable length arrays

The size of local automatic objects is determined at compile time, and the duration and scope of these objects end when you leave the function body where the object is declared. If the size requirement of an object is unknown at compile time, for example, an array of unknown number of elements, the programmer is responsible for dynamically allocating storage at run time, and freeing the storage before exiting the current scope. C99 introduces variable length arrays, where its usage removes the burden from the programming for allocating and remembering to free local automatic storage.

In the following example, the local array, *new*, is a variable length array:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reverse(const char *str, int n)
{
    int i;
    char new[n];
    for (i = 0; i < n; i++)
        new[i] = str[n-i-1];
    printf("%.*s\n", n, new);
}

int main(int argc, char *argv[])
{
    reverse("Hello World", strlen("Hello World"));
    exit(0);
}
```

Without variable length array support, storage for the local array new would have to be dynamically allocated as follows:

```
char *new = (char *)malloc(n);
```

and freed explicitly at the end of function reverse:

```
free(new);
```

## Compound literals

A compound literal is an unnamed object of type specified in parentheses. The value of the object is given in a braced initializer list. It is mainly used in situations where a temporary object would otherwise be required. In the following example, the emphasized line shows an example usage of compound literals:

```
#include <stdio.h>

typedef struct {
    short serial;
    char *name;
} Record;

void show(Record rec)
{
    printf("Employee serial: %d\n", rec.serial);
    printf("Employee name:   %s\n", rec.name);
}

int main(int argc, char *argv[])
{
    show((Record){ 12345, "Elizabeth" });
    exit(0);
}
```

Without a compound literal, a temporary object of type *Record* would be needed to be used in the function call:

```
Record tmp_rec = { 12345, “Elizabeth” };
show(tmp_rec);
```

## Designated initialization

In C89, initializers must be specified in the order and sequence of the elements or members to be initialized. Although for static storage duration objects, where they are implicitly initialized to zero already, if you need to initialize only specific members or elements of the object, you have to supply enough initializers. For example, for the following structure declaration:

```
typedef struct {
    short serial;
    char *name;
    int   salary;
    char  addr[40];
    char  city[20];
    char  state[2];
    char  zip[5];
    short location;
} Record;
```

If all members are to be initialized with the default value of zero, except the last member, *location*, all initializers must still be supplied:

```
Record emp1 = { 0, 0, 0, "", "", "", "", 649 };
```

With a designated initializer, on the other hand, initializers are only needed for members that are required to be explicitly initialized:

```
Record emp2 = { .location = 649 };
```

This greatly reduces the risk of errors that are proven to be hard to debug.

## Non-constant initializers for automatic aggregates

With C99, you can now initialize automatic storage duration aggregate members with non-constant initializers. For example:

```
#include <stdlib.h>

void func()
{
    struct {
        short serial;
        char *name;
    } rec = { .name = (char*)malloc(30) };
}
```

## Function-like macros with variable arguments

Functions with variable arguments, for example:

```
extern int printf(const char *, ...);
```

eliminate the need for many versions of the same function that accepts different numbers of arguments. C99 extends the concept further and allows variable arguments in function-like macros. As shown in the following example, debug code can now be handled more elegantly:

```
#include <stdio.h>

#if !defined(DEBUG)
    #define DBGMSG(fmt, ...) ((void)0)
#else
```

```
    #define DBGMSG(fmt, ...) ( \
        fprintf(stderr, "In %s: ", __func__), \
        fprintf(stderr, fmt, __VA_ARGS__) )
#endif

int main(int argc, char *argv[])
{
    int rc = 55;
    DBGMSG("return code = %d\n", rc);
    return rc;
}
```

### Pragma operator

The _Pragma operator allows you to code pragma directives in macros. In the following example, the declaration of struct S and the definition of its instance s are surrounded by two macros, PACK_1 and PACK_POP:

```
#define PACK_1   _Pragma("pack(1)")
#define PACK_POP _Pragma("pack(pop)")

PACK_1 struct S {
    char ch;
    int  i;
} s; PACK_POP
```

### Mixed declarations and code

C99 allows declarations mixed with code similar to C++. For example:

```
void func()
{
    int i;
    i = 10;
    int j;
    j = 20;
    ...
}
```

## 1.1.3 GNU C compatibility

There are plenty of programs developed using the GNU C compiler (also known as gcc). The proliferation of the open source concept, together with the far reaching nature of the Web, spawns a whole new group of developers who collaborate across physical boundaries, and the choice of compiler for this group is the gcc compiler. This does not necessarily mean that gcc is superior; on the contrary, the IBM compiler optimization technologies are among the best in the industry. The main reason for the gcc compiler's wide acceptance has to do with its many useful extensions, and the fact that it is freely available. Also, the GNU C and C++ compilers are available on various operating systems running on the many different types of hardware, providing cross platform development capabilities that are rivaled by no other.

The C for AIX Version 6 compiler supports many of the gcc extensions, and they allow you to port programs written for gcc to AIX more easily. The availability of each extension is indicated by a compiler predefined macro of the form __IBM_*feature*, where *feature* is the gcc feature name.

For example, the following code fragment written for gcc:

```
#if defined(__GNUC__)
    extern char *func(const char *__s, int __c) __attribute__((__pure__));
#endif
```

will be successfully compiled with C for AIX Version 6, if the code is modified as follows:

```
#if defined(__IBM_ATTRIBUTES) || defined(__GNUC__)
    extern char *func(const char *__s, int __c) __attribute__((__pure__));
#endif
```

To compile this code with previous versions of C for AIX compilers, it must be modified as follows:

```
#if defined(__IBM_ATTRIBUTES) || defined(__GNUC__)
    extern char *func(const char *__s, int __c) __attribute__((__pure__));
#else
    extern char *func(const char *__s, int __c);
    #pragma isolated_call(func)
#endif
```

The identifier, __pure__, is one of function attributes supported by C for AIX Version 6 (see “Function attributes” on page 13).

Please refer to *C for AIX Compiler Reference*, SC09-4960 for the full list of supported GNU C compatibility.

## Local labels

Ordinary labels has function scope, that is, they can only be defined once within a function body. This prevents the use of labels and goto statements inside macros, when the macro is expected to be expanded more than once in a function. As shown in the following example, a local label, on the other hand, is visible only within the block where it is declared, as well as in all nested blocks. A local label also hides the function scope label of the same name:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define FIND(str, val, len)                    \
{                                              \
    __label__ done;                            \
    int i;                                     \
    for (i = 0; i < len; i++)                  \
        if (str[i] == val) goto done;          \
    printf("%c not found in %s\n", val, str); \
    done:                                      \
    printf("found %c in [%d]\n", val, i);     \
}

int main(int argc, char *argv[])
{
    FIND("hello", 'l', 5);
    FIND("world", 'l', 5);
    exit(0);
}
```

## __typeof__ operator

Similar to the sizeof operator, the __typeof__ operator takes an expression or a type as an operand, but returns the type of the operand instead of the size. It can be used anywhere a typedef name is used. This operator is particularly useful in marcos, where the type of the macro argument is not known before hand. For example, instead of writing several versions of a SWAP_*type* macro for the different integral types:

```
#define SWAP_char(a,b) { char temp = a; a = b; b = temp; }
#define SWAP_short(a,b) { short temp = a; a = b; b = temp; }
...
```

you can use one SWAP macro that handles all types:

```
#define SWAP(a,b) { __typeof__(a) temp = a; a = b; b = temp; }
```

## __alignof__ operator

Use the __alignof__ operator to find out the alignment of a type or an object. Due to the different alignment rules and packing supported by the C for AIX compiler, the alignment of an object may change depending on options or pragmas used, as shown in the following example:

```
#include <stdio.h>
#include <stdlib.h>

#pragma pack(2)
struct {
    int i;
    double d;
} s;
```

```
int main(int argc, char *argv[])
{
    printf("alignment of double is %d\n", __alignof__(double));
    printf("alignment of s.d is %d\n", __alignof__(s.d));
    exit(0);
}
```

will yield:

```
alignment of double is 8
alignment of s.d is 2
```

## Function attributes

Function attributes are used to help the compiler apply better optimization on function calls. The C for AIX Version 6.0 compiler supports three function attributes: *noreturn*, *const*, and *pure*.

The noreturn attribute indicates to the compiler that the function does not return control to the statement following the function call. The C library already has several functions, such as abort and exit, that behave as if the function is declared with the noreturn attribute; the compiler is already aware of these functions and is able to optimize the function calls accordingly (when the -qlibansi option is in effect). User defined functions that do not return control to the calling side can be declared with this attribute for better performance. This attribute is functionally equivalent to #pragma leaves.[1]

The const and pure attributes are equivalent, and are used to indicate to the compiler that the function does not have or rely on any side effects. The return value only depends on the parameters, and pointer arguments are not examined in the function body. The function does not call any non-const function, nor access any global or external storage. This attribute is functionally equivalent to #pragma isolated_call.[2]

To use the pure function attribute, see the example in 1.1.3, “GNU C compatibility” on page 10.

## Variable attributes

Variable attributes (*aligned*, *mode*, or *packed*) are used to modify variable declarations.

[1] The #pragma leaves directive specifies that a function never returns.

[2] The #pragma isolated_call directive specifies that a function does not have or rely on side effects.

The aligned attribute causes the complier to use a different alignment for variable or structure members. It specifies the minimum number of bytes to use for aligning the declaration, as shown in the following example:

```
#include <stdio.h>
#include <stdlib.h>

struct {
    int i;
    double __attribute__((__aligned__(16))) d;
} s;

int main(int argc, char *argv[])
{
    printf("alignment of double is %d\n", __alignof__(double));
    printf("alignment of s.d is %d\n", __alignof__(s.d));
    exit(0);
}
```

will yield the result:

```
alignment of double is 8
alignment of s.d is 16
```

To change the packing of aggregate or structure members, or to use the smallest possible alignment, use the packed attribute, as shown in the following example:

```
#include <stdio.h>
#include <stdlib.h>

struct {
    int i;
    double __attribute__((__packed__)) d;
} s;

void main()
{
    printf("alignment of double is %d\n", __alignof__(double));
    printf("alignment of s.d is %d\n", __alignof__(s.d));
}
```

will yield the result:

```
alignment of double is 8
alignment of s.d is 1
```

The mode attribute lets you select an integer based on width, as shown in the following example (the supported widths are byte, word, and pointer):

```
#include <stdio.h>
#include <stdlib.h>
```

```
int __attribute__((__mode__(byte))) x;

void main()
{
    printf("sizeof(int) is %d\n", sizeof(int));
    printf("sizeof(x)   is %d\n", sizeof(x));
}
```

will yield the result:

```
sizeof(int) is 4
sizeof(x)   is 1
```

### 1.1.4 Enhanced language level support

Not only does the C for AIX Version 6.0 compiler support the latest ISO C language standard (C99), it also has several language extension modes that make the use of the C language ever more powerful. These extensions add flexibility and allow the programmer to achieve a programming task more easily.

Language levels are supported by the -qlanglvl compiler option. The two core levels, -qlanglvl=stdc89 and -qlanglvl=stdc99, strictly enforces the standards, and are used mainly for compiling standard conforming programs. To use extensions to the standard language levels, -qlanglvl=extc89 and -qlanglvl=extc99 add to the core levels orthogonal features that do not interfere with the core standard features (see Figure 1-1), that is, a standard conforming program will be compiled just as successfully with -qlanglvl=extc89 or -qlanglvl=extc99 as it will be with -qlanglvl=stdc89 or -qlanglvl=stdc99. Therefore, it is recommended that you use the extension language levels to compile programs that are written using other compilers such as gcc.

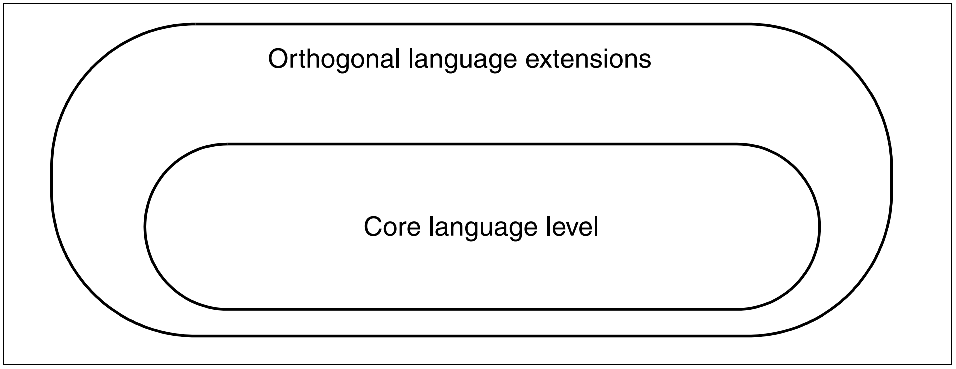


*Figure 1-1 Core and orthogonal extensions*

# 1.2 VisualAge C++ for AIX Version 6.0

VisualAge C++ for AIX Version 6.0 offers a command-line compiler that supports the ISO/IEC 14882:1998 C++ International Standard. This latest version of the C++ compiler features a conforming C++ Standard Library, including the Standard Template Library (STL), enhanced optimization options that exploit the POWER4 architecture, OpenMP support, and more. It is supported on AIX Version 4.3.3 or later.

**Note:** The Standard Template Library is discussed in more detail in 10.6, “Standard C++ Library and STL” on page 388.

C++ programs written using Version 4 or 5 of IBM VisualAge C++ Professional for AIX are source compatible with VisualAge C++ for AIX Version 6.0. However, programs written using IBM C and C++ Compilers for AIX Version 3.6 and earlier are not source compatible, because the former compilers were based on the ISO C++ Draft. Also, the IBM Open Class® library has been removed and is no longer supported.

If installed, the compiler is installed under /usr/vacpp by default and uses the /etc/vacpp.cfg configuration file. To install to an alternate directory, or to retain the installation of a previous version of VisualAge C++ for AIX compiler, refer to 1.3.2, “Retaining a previous version of the compiler” on page 22.

The VisualAge C++ for AIX Version 6.0 compiler uses the LUM licensing system to control usage of the product. Refer to the following sections:

- Section 1.4, “Activating the compilers” on page 23
- Section 1.5, “Activating the LUM server” on page 26
- Section 1.6, “Enrolling a product license” on page 27

## 1.2.1 New or improved optimization features

A number of optimization features have been introduced in VisualAge C++ for AIX Version 6.

### Interprocedural Analysis

Interprocedural Analysis, or IPA, is now available in VisualAge C++ for AIX Version 6.0. IPA is an optimization performed on the entire application across function boundaries. See “Interprocedural analysis” on page 2 for more details.

### New options and pragmas

In addition to the “New options and pragmas” on page 3 supported by C for AIX Version 6.0, VisualAge C++ for AIX Version 6.0 now supports the following options:

- -qsmp (previously supported by C for AIX only)
- -O4 and -O5 (previously supported by C for AIX only)
- -qipa (previously supported by C for AIX only)
- -qcache (previously supported by C for AIX only)
- -qkeepinlines
- -qtemplateregistry
- -qtemplaterecompile
- -qalign=bit_packed
- -qoldpassbyvalue
- -qlanglvl=ansiinit
- #pragma unroll (previously supported by C for AIX only)

For a detailed description of these options, please refer to the *VisualAge C++ for AIX Compiler Reference*, SC09-4959.

## 1.2.2 OpenMP support

New in VisualAge C++ for AIX Version 6.0, the compiler now supports the OpenMP Version 1.0 specification for shared memory parallel programming. OpenMP is an Application Program Interface specification that provides a simple and flexible interface, including a number of pragma directives, data scope attributes, library functions, and environment variables, for parallel application development. Applications that conform to the OpenMP specification is easily portable to other platforms that support the specification.

OpenMP support is enabled in the compiler by the -qsmp option, which was previously available with the C for AIX compiler only. To ensure strict compliance to the OpenMP specification, use the -qsmp=omp option. Bear in mind that -qsmp=omp disables automatic parallelization performed by the compiler. See 1.2.3, “Automatic parallelization” on page 18. In either case, you must use one of the thread-safe compiler driver with the -qsmp option (see Table 1-4 on page 30).

For further information about OpenMP support, see Chapter 9, “Program parallelization using OpenMP” on page 335.

### 1.2.3 Automatic parallelization

In addition to strict OpenMP specification support, the -qsmp option enables automatic parallelization for program loops. Each parallel portion of the program is executed in its own thread, perhaps in its own processor of a multi-processor machine. To use the -qsmp option, you must use one of the thread-safe compiler drivers, as described in Table 1-4 on page 30.

### 1.2.4 Improved template handling

VisualAge C++ for AIX Version 6.0 introduces two new options, -qtemplateregistry and -qtemplaterecompile, which completely eliminate the need to structure your template code for reduced compilation time.

The -qtemplateregistry option works by storing template instantiation information in a registry as compilation occurs. The registry is read for each compilation, and a check is done when an instantiation is encountered. If the instantiation has already been seen, nothing will happen; otherwise, the template will be instantiated in the object file. In either case, a record is added to the registry to keep track of the information about use and instantiation for each compilation unit.

Should a change to a compilation unit remove a template instantiation, recompiling the compilation will result in undefined symbols at link time, since other files now may require the missing template instantiation. In this case, use the -qtemplaterecompile option to cause a recompilation of all source files that rely on the template.

The -qtemplateregistry and -qtemplaterecompile options are discussed in more detail in 10.5, “Template registry: The preferred method” on page 387.

### 1.2.5 C99 features

The following C99 features are available in VisualAge C++ for AIX Version 6.0 with the -qlanglvl=extended option:

- “restrict qualifier” on page 6
- “Function-like macros with variable arguments” on page 9
- “Pragma operator” on page 10

### 1.2.6 GNU G++ compatibility

The VisualAge C++ for AIX Version 6.0 compiler supports the same set of GNU C extension features described in 1.1.3, “GNU C compatibility” on page 10.

# 1.3 Installing the compilers

The installation of the C for AIX Version 6.0 and VisualAge C++ for AIX Version 6.0 compilers is a very simple task. There are a number of steps that need to be performed to end up with correctly installed and working compilers.

## 1.3.1 Install compiler filesets

The first step in the installation process is to install the compiler product filesets onto the system. The filesets to be installed will vary, depending on the compiler product and the desired configuration.

### Selecting the required filesets

The compiler products are either delivered on CD-ROM media, or downloaded from the official IBM Web site, and are accompanied by a license certificate for the number of licenses purchased. The CD-ROM media or download includes the compiler filesets along with a number of other filesets, some of which are optionally installable, and some of which are co-requisites of the compiler filesets and are installed automatically. Table 1-1 lists the main packages in the C for AIX Version 6.0 product, and Table 1-2 lists the main packages in the VisualAge C++ for AIX Version 6.0 product.

*Table 1-1 C for AIX Version 6.0 packages*

| Group | Description |
|---|---|
| IMNSearch | Search engine for HTML documentation |
| idebug | Debugger with graphical user interface |
| memdbg | Memory debugging toolkit |
| vac | C for AIX compiler |
| xlopt | Optimization library and run time |
| xlsmp | Parallelization run-time component |

*Table 1-2 VisualAge C++ for AIX Version 6.0 packages*

| Group | Description |
|---|---|
| IMNSearch | Search engine for HTML documentation |
| idebug | Debugger with graphical user interface |
| memdbg | Memory debugging toolkit |
| vac | C for AIX compiler |

| Group | Description |
|---|---|
| vacpp | VisualAge C++ for AIX compiler[a] |
| vatools | VisualAge® Tools Help |
| xlC | C++ Application Development Toolkit |
| xlopt | Optimization library and run time |
| xlsmp | Parallelization run-time component |

[a] The vacpp group contains installp packages for part of the C++ compiler only; it requires packages from the vac group.

**Note:** A single package can contain multiple filesets and a fileset can be included in only one package. For further information about AIX software packaging terminology, see Chapter 12, "Packaging your applications" on page 405.

In all cases, the target AIX system should already have the bos.adt.include fileset installed, which contains the system provided header files. The other filesets in the bos.adt package contain useful tools and utilities often used during application development, so it is a good idea to install the entire package. If your system does not have the filesets installed, you will need to locate your AIX installation media and install them prior to installing the compilers, since these filesets are AIX version specific and are not supplied with the compiler product.

When installing the C for AIX Version 6.0 product, installing the vac.C fileset will automatically install the minimum of additional required filesets. Installing the vacpp.cmp fileset will automatically include the minimum required filesets for VisualAge C++ for AIX. The additional filesets you may wish to install are the documentation filesets. Ensure that the vac.lic and vacpp.lic filesets are installed, as they contain the license files required when activating the compiler.

Regardless of the product or required configuration, the filesets can be installed using one of two methods, as discussed in the following sections.

### Install using the Web-based System Manager

If your system has a graphical user interface, the filesets can be installed using the **wsm** command. The procedure is as follows:

1. Log in as the root user.
2. Insert the product CD-ROM media in the CD or DVD device.
3. Start the software installation taskguide with the following command:

```
# wsm install
```

4. From the Software drop-down menu, select **New Software (Install/Update)** → **Install Additional Software** → **Advanced Method**.
5. In the Install Software dialog, select the CD-ROM device as the software source. Then, select to install the specific software available from the software source.
6. Select the **Browse** button to generate a list of software on the media.
7. Select the desired filesets from the dialog. Press and hold down the control key while dragging the mouse cursor to select one or more additional objects.
8. Select the **OK** button once you have selected the desired filesets to return to the Software Install dialog.
9. Select the **OK** button to start the install.
10. Select the **YES** button to continue with the install. A pop-up window will appear and show the output of the installation process.
11. Select the **Close** button once the installation has completed.

## Install using SMIT

If your system does not have a graphical user interface, or you do not wish to use a Web-based System Manger, you can install the required filesets using the `smit` command as follows:

1. Log in as the root user.
2. Insert the product CD-ROM media in the CD or DVD device.
3. Start the SMIT dialog with the following command:

   ```
   # smit install_latest
   ```

4. Press the F4 key to generate a list of possible input devices.
5. Select the CD-ROM device.[3]
6. Press the F4 key to generate a list of available filesets.
7. Select the required filesets by highlighting them and then pressing the F7 key.
8. Press the Enter key once the required filesets have been selected.
9. Press the Enter key to start the install.
10. Press the Enter key to continue the install.
11. Press the F10 key to exit once the installation has completed.

[3] A DVD-RAM or DVD-ROM device is also shown as CD-ROM device in the selection panel.

> **Note:** The compiler products cannot be used immediately after installation. Prior to invoking the compiler, a product licence must be enrolled with the License Use Management (LUM) system. See 1.4, “Activating the compilers” on page 23.

### 1.3.2 Retaining a previous version of the compiler

Before installing the compiler, it is recommended that you uninstall any previous version of C for AIX and VisualAge C++ for AIX compilers already installed. However, you can install the compilers in another directory, and retain the previous installation. This is done by using the supplied Perl scripts, vacndi and vacppndi.

Ensure the Perl run-time environment fileset, perl.rte, is installed on your system. Install the vac.ndi fileset from the C for AIX CD-ROM media, or the vacpp.ndi fileset from the VisualAge C++ for AIX CD-ROM media, or both. You can then use the /usr/vac/bin/vacndi and /usr/vacpp/bin/vacppndi scripts to install just the compiler, or the compiler with help documentation and samples, to a location of your choice. Do not move or rename the directory or any of its components after installation; you must reinstall to a new location if you want to change the installed directory.

To install C for AIX compiler with help documentation and samples, run:

```
# perl /usr/vac/bin/vacndi -d source_path -b target_directory
```

where *source_path* is where the C for AIX product filesets are located, and *target_directory* is the installation directory. If the -b option is omitted, the default installation directory is used (that is, /usr/vac).

To install C for AIX compiler only without help documentation and samples, run:

```
# perl /usr/vac/bin/vacndi -d source_path -b target_directory -m
```

To install the VisualAge C++ for AIX compiler product, simply replace /usr/vac/bin/vacndi in the above commands with /usr/vacpp/bin/vacppndi.

After installation, check the ./vacndi.log or ./vacppndi.log log file and make sure the installation is performed successfully. To remove the installation, delete the *target_directory* directory as specified in the installation step.

## 1.4 Activating the compilers

Once you have installed the desired compiler filesets onto the system, the next step in the process is to enroll a license for the product into the LUM system. This section describes the process of configuring a LUM server and enrolling a product license. If you already have a LUM environment enabled, you may go directly to 1.6, “Enrolling a product license” on page 27.

### 1.4.1 What is LUM

IBM License Use Management Runtime, referred to hereafter as License Use Management (LUM), contains the tools needed in an end-user environment to manage product licenses and to get up-to-date information about license usage.

LUM is the replacement for the iFOR/LS and Net/LS systems that were used in previous versions of AIX and with previous versions of the IBM compilers.

The LUM run time is included with AIX Version 4.3 and higher and is automatically installed. A comprehensive description of the functionality of LUM can be found in the LUM online documentation supplied on the AIX Version 4.3 and higher product media in the ifor_ls.html.en_US.base.cli fileset, which is not automatically installed when installing AIX.

### 1.4.2 Configuring LUM

Normally, one or more LUM license servers need to be configured. However, no license server needs to be configured if the licensed product supplies a simple nodelock license certificate. Both the C for AIX Version 6.0 and VisualAge C++ for AIX Version 6.0 compiler products supply a simple nodelock license certificate.

The simplest method of licensing the latest compiler products is to use the simple nodelock license certificate. When this is done, there is no need to configure a LUM server; however, the installation of the certificate in large numbers of machines can be cumbersome.

If you wish to use the simple nodelock certificate, you can skip directly to 1.6, “Enrolling a product license” on page 27. If you wish to use the additional functionality available when using a license server, then the first step is to decide which server type is best suited for your environment.

There are two types of license servers:

- Concurrent nodelock license server
- Concurrent network license server

A *concurrent nodelock license server* supports concurrent nodelock product licenses. A concurrent nodelock license is local to the node where the LUM enabled product has been installed. It allows a limited number of simultaneous users to invoke the enabled licensed product on the local system.

A *concurrent network license server supports* concurrent network product licenses. A concurrent network license is a network license that can temporarily grant a user on a client system the authority to run a LUM enabled product.

Either or both of the above license servers may be configured on a single system. The number of concurrent users for the product is specified during the enrollment of the product license certificate, as described in 1.6, "Enrolling a product license" on page 27.

The advantage of using a concurrent nodelock license server is that the server is installed on the same machine as the compiler, and, therefore, users can obtain compiler licenses even if the machine is temporarily disconnected from the network. The disadvantage, however, is that installation of licenses is cumbersome in environments with a large number of client machines.

The main advantage of using a central network license server is that the administration of product licenses is very simple. The disadvantage is that client machines must be able to contact the license server in order to use the licensed products.

Configuring LUM requires answering several questions on how you would like to set up the LUM environment. It is recommended that users read the LUM documentation supplied with the AIX product media prior to configuring LUM.

A LUM server can be configured in several different ways. You can issue commands on the command line with appropriate arguments to configure the LUM server. You can issue a command that starts a dialog and asks a number of questions to determine the appropriate configuration, or you can configure the server using a graphical user interface.

### Configuring a nodelock server

For small numbers of client machines (typically 10 or less), using a nodelock license server on each machine is the simplest method of configuring LUM.

Log in as the root user and perform the following commands to configure a machine as a nodelock license server:

```
# /usr/opt/ifor/ls/os/aix/bin/i4cfg -a n -S a
# /usr/opt/ifor/ls/os/aix/bin/i4cfg -start
```

The first command configures the local machine as a nodelock license server and sets the option that the LUM daemons should be started automatically when the system boots. The second command starts the LUM daemons.

## Using the interactive configuration tool

As an alternative to using the above commands, you can use the /usr/opt/ifor/ls/os/aix/bin/i4config interactive configuration script to perform the same actions.

1. Log in as user ID root on the system where the license server will be installed.
2. Invoke the LUM configuration tool by entering the `/usr/opt/ifor/ls/os/aix/bin/i4config` command. This is the command line version of the LUM configuration tool.
3. Answer the LUM configuration questions as appropriate. The answers to the configuration questions are dependent on the LUM environment you wish to create.

The following are typical answers to the configuration questions of LUM in order to configure both concurrent nodelock and concurrent network license servers on a single system. You may change the various answers accordingly to suit your preferred system environment. For details on configuring LUM, please read the documentation that comes with LUM.

1. Select 4 “Central Registry (and/or Network and/or Nodelock) License Server” on the first panel.
2. Answer y to “Do you want this system be a Network License Server too?”
3. Answer y to “Do you want this system be a Nodelock License Server too?”
4. Answer n to “Do you want to disable remote administration of this Network License Server?”
5. Answer n to “Do you want to disable remote administration of this Nodelock License Server?”
6. Select 2 “Direct Binding only” as the mechanism to locate a license server.
7. Answer n to “Do you want to change the Network License Server ip port number?”
8. Answer n to “Do you want to change the Central Registry License Server ip port number?”
9. Answer n to “Do you want to change the Nodelock License Server ip port number?”
10. Select 1 “Default” as the desired server(s) logging level.
11. Enter blank to accept the default path for the default log file(s).

12. Answer y to “Do you want to modify the list of remote Nodelock and/or Network License Servers this system can connect to in direct binding mode (both for administration purposes and for working as Network License Client)?”
13. Select 3 “Create a new list” to the direct binding list menu.
14. Enter the host name, without the domain, of the system you are configuring LUM when prompted for the “Server network name(s).”
15. Answer n to “Do you want to change the default ip port number?”
16. Select 3 “Create a new list” to the direct binding list menu.
17. Enter the host name, without the domain, of the system you are configuring LUM when prompted for the “Server network name(s).”
18. Answer n to “Do you want to change the default ip port number?”
19. Answer y to “Do you want the License Server(s) automatically start on this system at boot time?”
20. Answer y to continue the configuration setup and write the updates to the i4ls.ini file.
21. Answer y to “Do you want the License Server(s) start now?”

Both concurrent nodelock and concurrent network license servers should now be configured on your system.

For more information on configuring and using LUM, refer to the LUM online documentation supplied with AIX. As an alternative, the LUM manual, *Using License Use Management Runtime for AIX*, can be viewed online in PDF format at the following URL:

ftp://ftp.software.ibm.com/software/lum/aix/doc/V4.6.0/lumusg.pdf

## 1.5 Activating the LUM server

After configuring and starting the LUM server, you can enroll product licenses. Before attempting to enroll a license, you must first ensure that the LUM daemons are active. This can be done with the following command:

```
# /usr/opt/ifor/ls/os/aix/bin/i4cfg -list
```

Depending on the type of LUM server configured, the output will be similar to the following:

```
i4cfg Version 4.6.6 AIX -- LUM Configuration Tool
(c) Copyright 1995-2002, IBM Corporation, All Rights Reserved
US Government Users Restricted Rights - Use, duplication or disclosure
```

```
restricted by GSA ADP Schedule Contract with IBM Corp.

Subsystem         Group            PID         Status
 i4llmd           iforls           24006       active
```

If no subsystem is listed as active, then start them with the following command:

```
# /usr/opt/ifor/ls/os/aix/bin/i4cfg -start
```

The only daemon that must be active is the Nodelock License Server Subsystem (i4llmd) daemon. The other daemons that may be active, depending on your configuration, are as follows:

- License Sever Subsystem (i4lmd)
- Central Registry Subsystem (i4gdb)
- Global Location Broker Data Cleaner Subsystem (i4glbcd)

## 1.6 Enrolling a product license

After LUM has been configured on your system, the product license certificates can be enrolled with the LUM license server. Three LUM product license certificates are provided with each of the compiler products:

1. Concurrent nodelock license certificate
2. Concurrent network license certificate
3. Simple nodelock license certificate

You should enroll the appropriate license certificate for the type of LUM environment you have configured.

The default locations of the license certificates for the compiler products are detailed in Table 1-3.

*Table 1-3 License certificate locations*

| Compiler | License certificate type | Location |
|---|---|---|
| C for AIX Version 6.0 | Concurrent Network | /usr/vac/cforaix_c.lic |
| | Concurrent Nodelock | /usr/vac/cforaix_cn.lic |
| | Simple Nodelock | /usr/vac/cforaix_n.lic |
| VisualAge C++ for AIX Version 6.0 | Concurrent Network | /usr/vacpp/vacpp_c.lic |
| | Concurrent Nodelock | /usr/vacpp/vacpp_cn.lic |
| | Simple Nodelock | /usr/vacpp/vacpp_n.lic |

### 1.6.1 Enrolling a concurrent license

To enroll a Concurrent Network or Concurrent Nodelock license certificate, perform the following steps:

1. Log in as root on the system where the license server is installed.
2. Invoke the LUM configuration tool by entering the LUM Basic License Tool command as follows:

   ```
   /usr/opt/ifor/ls/os/aix/bin/i4blt
   ```

The i4blt tool contains both a graphical user interface and a command line interface. Note that the LUM daemons must be running before starting the i4blt tool. Refer to 1.5, “Activating the LUM server” on page 26 for information on how to check the status of the LUM daemons.

If the X11 run time (X11.base.rte fileset) has been installed on your system, the GUI version of the tool will be invoked. Otherwise, the command line version will be invoked, and an error will occur since the appropriate command line parameters were not specified.

#### Enrolling using the graphical user interface

When the GUI version of i4blt tool is available, follow these steps:

1. Select the **Products** pull-down and click on the **Enroll Product** item.
2. Click on the **Import** button. The Import panel should be displayed.
3. In the Filter entry prompt, enter `/usr/vacpp/*.lic` if you are enrolling a license for VisualAge C++ for AIX or `/usr/vac/*.lic` if you are enrolling a license for C for AIX, and press Enter. This will show the various product license files in the Files panel. The three license files for the product, as detailed in Table 1-3 on page 27, should be displayed.
4. Select either the *prod*_c.lic or *prod*_cn.lic (where *prod* is either vacpp or cforaix) license by clicking on the entry.
5. Click **OK**. The Enroll Product panel should be re-displayed with information regarding the product indicated.
6. Click on the **OK** button of the Enroll Product panel. The Enroll Licenses panel should be displayed.
7. Fill in the information on the Administrator Information portion of the panel (optional.)
8. Fill in the number of valid purchased licenses of the product under Enrolled Licenses in the Product information portion of the panel. (mandatory.)
9. Click on the **OK** button of the Enroll Licenses panel. The product should be successfully enrolled. You may terminate the i4blt tool.

### Enrolling using the command line

When you use the command line interface of i4blt tool, follow these steps:

1. From the required product license file, as detailed in Table 1-3 on page 27, extract the **`i4blt`** command from the top of the file.
2. Replace number_of_lics from the command with the number of valid purchased licenses of the product (mandatory).
3. Replace admin_name with the name of the administrator (optional).
4. Invoke this command as root from /var/ifor. The product should be successfully enrolled.

## 1.6.2 Enrolling a simple nodelock license

Read the instructions at the top of the simple nodelock license certificate file. In general, this type of license will be installed when no LUM system has been configured. This means enrolling the license is simply a case of placing the indicated license information line into the LUM nodelock file, /var/ifor/nodelock.

# 1.7 Invoking the compilers

Once a compiler product license has been enrolled, you are now ready to use the compilers. However, the compiler drivers are not installed in a directory that is searched with the default PATH environment variable. There are a number of methods for resolving this issue:

- Create symbolic links of the compiler drivers to /usr/bin using the **`ln`** command.
- Add the directory containing the compiler drivers to the default PATH environment variable set in the /etc/environment configuration file.
- Add the directory containing the compiler drivers to the PATH environment variable in each user's .profile shell configuration file.
- Change the Makefiles used in your development environment to configure the compiler macro to use the absolute path. For example:

  ```
  CC=/usr/vac/bin/cc
  ```

**Note:** Creating symbolic links is the preferred option since it resolves the problem for all users after a simple action by the root user.

## 1.7.1 Default compiler drivers

The Version 6.0 compiler products include a number of default compiler configurations in the /etc/vac.cfg compiler configuration file. The default C++ command line driver is /usr/vacpp/bin/xlC. The three main C compiler command line drivers are as follows:

| | |
|---|---|
| **/usr/vac/bin/cc** | Extended mode C compiler. |
| **/usr/vac/bin/xlc** | ANSI C compiler, using UNIX header files. |
| **/usr/vac/bin/c89** | ANSI C compiler, using ANSI C header files. |

There are a number of additional command line drivers available, each one based on the basic cc, xlc, and xlC drivers described above, as described in Table 1-4.

*Table 1-4 Compiler driver extensions*

| Command extension | Meaning |
|---|---|
| _r | Use the UNIX98 threads libraries. |
| _r7 | Use the POSIX Draft 7 threads libraries. |
| _r4[a] | Use the POSIX Draft 4 (DCE)[b] threads libraries. |
| 128 | Enable 128 bit double precision floating point values and use appropriate libraries. |
| 128_r | Enable 128 bit double precision floating point values and use the UNIX98 threads libraries. |
| 128_r7 | Enable 128 bit double precision floating point values and use the POSIX Draft 7 threads libraries. |
| 128_r4 | Enable 128 bit double precision floating point values and use the POSIX Draft 4 (DCE) threads libraries. |

[a] Compiler drivers with extension _r4 are not supported on AIX 5L Version 5.2 and later. AIX 5L Version 5.1 supports those compiler drivers only if DCE is installed on the system.

[b] DCE stands for Distributed Computing Environment.

For example, to compile an ANSI C program using Draft 7 of the POSIX threads standard, use the xlc_r7 compiler driver. To compile a C++ program that uses 128-bit floating point values, use the xlC128 compiler driver.

**Note:** The use of compiler drivers with extensions _r4 and _r7 is discouraged when developing new applications.

# 1.8 Where to find help

The Version 6.0 compilers provide documentation in both HTML and PDF format. The default configuration makes it very easy to view the online documentation on the machine on which it is installed. There is also information available on the Web that provides useful information for developers using the AIX platform.

## 1.8.1 Online documentations

The Version 6.0 compiler documentation is written in HTML format. The HTML files are stored in a single file in ZIP format. The files are viewed using an HTML browser, which uses a cgi-bin script to extract and view the required files. There is no need to manually unpack the ZIP file.

If not already installed, the online help documentation can be found in the vac.html.en_US and vacpp.html.en_US filesets. Once installed, you can access the online help with the `/usr/vac/bin/cforaixhelp` and `/usr/vacpp/bin/vacpphelp` commands. The commands start the default Netscape browser (which is supplied on the AIX Bonus Pack media) with the correct URL.

If you are using the AIX CDE interface, the C for AIX compiler documentation can also be started by selecting **Application Manager** → **C for AIX** → **Help Homepage**, or **Application Manager** → **VisualAge C++ Professional** → **Help Homepage for VisualAge C++ for AIX**.

The official compiler documentations are also available in PDF format in /usr/vac/pdf and /usr/vacpp/pdf.

## 1.8.2 Viewing online documentation remotely

By default, it is not possible to view the online documentation from a remote machine. It can be done in a simple way by logging in to the machine that has the documentation installed, set the DISPLAY environment variable to use a remote X11 display, then view the documentation by invoking the same command used to view locally.

A better solution, particularly in larger environments or where remote clients do not have the capability to display X applications, is to configure the machine to allow remote viewing of the documentation.

### Configuring the HTTP server

Suppose the machine that has the documentation filesets installed has a fully qualified domain name of docs.ibm.com. The following example demonstrates the steps performed on that machine to allow remote clients to view the compiler documentation using their Web browser:

1. Log in as the root user.
2. Perform the following command:

   ```
   # cp /etc/IMNSearch/httpdlite/httpdlite.conf \
       /etc/IMNSearch/httpdlite/vacpp.conf
   ```

3. Edit /etc/IMNSearch/httpdlite/vacpp.conf, and make the following changes:

   a. Change the HostName line from:

   ```
   HostName localhost
   ```

   to:

   ```
   HostName docs.ibm.com
   ```

   If the HostName line is not present, or has a comment symbol (#) at the start of the line, then simply add the following line to the file:

   ```
   HostName docs.ibm.com
   ```

   b. Change the Port line from:

   ```
   Port 49213
   ```

   to:

   ```
   Port 49214
   ```

   c. Add one or more Allow lines to specify which hosts are permitted to access the Web server. The Allow statement has the following syntax:

   ```
   Allow network-ip network-mask
   ```

   A client is only granted access if the following rule is met (& is a bitwise AND operation):

   ```
   client-ip & network-mask == network-ip & network-mask
   ```

   For example, if you wanted machines within a network address, such as 9.x.x.x, to be able to access the help server, you would add the following statement to vacpp.conf:

   ```
   Allow 9.0.0.0 255.0.0.0
   ```

   d. Save the file and exit the editor.

4. Edit the file /etc/inittab. There is a line that executes the **httpdlite** command with a configuration filename argument. The line is as follows:

   ```
   httpdlite:23456789:once:/usr/IMNSearch/httpdlite/httpdlite -r
   /etc/IMNSearch/httpdlite/httpdlite.conf >/dev/console 2>&1
   ```

   Make a copy of this line immediately below the original line. In the new line:

   a. Change the first field from httpdlite to httpdlite2.

b. Change the part of the line that reads httpdlite.conf to vacpp.conf

   The result should be as follows:

   ```
   httpdlite2:23456789:once:/usr/IMNSearch/httpdlite/httpdlite -r
   /etc/IMNSearch/httpdlite/vacpp.conf >/dev/console 2>&1
   ```

   Save the file and exit from the editor.

5. Reboot the system or run the following command to start the second instance of the httpdlite process:

   ```
   /usr/IMNSearch/httpdlite/httpdlite -r /etc/IMNSearch/httpdlite/vacpp.conf
   >/dev/console 2>&1
   ```

The steps described above configure an instance of an HTTP server to respond on a specific port number to requests to access compiler documentation.

The following sections detail the additional steps required to configure the documentation for each compiler product to be served by the HTTP server.

## Configuring the C++ documentation

The following steps are required to enable the online documentation for the VisualAge C++ for AIX Version 6 compiler to be served by the HTTP server:

1. Log in as the root user.
2. Change the directory to /var/vacpp/en_US.
3. Edit the hgssrch.htm file, and change the line:

   ```
   <form action=”http://localhost:49213/cgi-bin/vacsrch.exe” method=”POST”
   target=”content”>
   ```

   to:

   ```
   <form action=”http://docs.ibm.com:49214/cgi-bin/vacsrch.exe” method=”POST”
   target=”content”>
   ```

   Then, save the file and exit the editor.

4. Issue the following command:

   ```
   /usr/IMNSearch/cli/imndomap -u "VAC6ENUS"
   "http://docs.ibm.com:49214/cgi-bin/vahwebx.exe/en_US/vacpp6/Extract/0/"
   "VisualAge C++"
   ```

5. Users can point their browser at the following URL to browse and search the documentation:

```
http://docs.ibm.com:49214/cgi-bin/vahwebx.exe/en_US/vacpp/Extract/0/index.htm
```

### Configuring the C compiler documentation

The following steps are required to enable the online documentation for the C for AIX Version 6 compiler to be served by the HTTP server:

1. Log in as the root user.
2. Change the directory to /var/vac/en_US.
3. Edit the hgssrch.htm file, and change the line:

   ```
   <form action=”http://localhost:49213/cgi-bin/caixsrch.exe” method=”POST”
   target=”content”>
   ```

   to:

   ```
   <form action=”http://docs.ibm.com:49214/cgi-bin/caixsrch.exe” method=”POST”
   target=”content”>
   ```

   Then, save the file and exit the editor.
4. Issue the following command:

   ```
   /usr/IMNSearch/cli/imndomap -u "CAIXENUS"
   "http://docs.ibm.com:49214/cgi-bin/vahwebx.exe/en_US/cforaix/Extract/0/" "C
   for AIX"
   ```

5. Users can point their browser at the following URL to browse and search the documentation:

   ```
   http://docs.ibm.com:49214/cgi-bin/vahwebx.exe/en_US/cforaix/Extract/0/index
   .htm
   ```

## 1.8.3 Where to find help on the Web

IBM maintains many Web sites that provide useful information for developers using the AIX platform. The most important ones are described in the following sections.

### AIX operating system documentation

The online documentation for the AIX operating system can be viewed at the following URL:

http://www.ibm.com/servers/aix/library/

The site contains up-to-date versions of the HTML documentation supplied with the AIX product media.

As new releases of the AIX operating system become available, they generally add new functionality. As a developer, you might wish to use some of the new functionality, but the decision to do so may also be based on the minimum level of AIX required to use a particular feature. The IBM Redbook, *AIX 5L Differences Guide Version 5.2 Edition,* SG24-5765, is updated with each new release of AIX, and contains information on when particular features were introduced.

### Compiler product information

The latest compiler products both have support Web sites that contain useful hints, tips, frequently asked questions, and links to other useful Web sites. The support page for the VisualAge C++ for AIX Version 6.0 compiler is:

http://www.ibm.com/software/ad/vacpp/support.html

The support page for the C for AIX Version 6.0 compiler is:

http://www.ibm.com/software/ad/caix/support.html

Information on the availability of IBM products for the AIX operating system, along with details of when support for products will be withdrawn, is available on the following Web site:

http://www.ibm.com/servers/aix/products/ibmsw/list/

### PartnerWorld® for developers

PartnerWorld for Developers is a worldwide program supporting developers who build solutions using IBM technologies. The program covers all IBM platforms, not just AIX. Its Web site contains a lot of useful information for the AIX developer, including white papers, sample code, and technology articles. It can be located on the Web at the following URL:

http://www.ibm.com/partnerworld/developer

## 1.8.4 Applying fixes and service updates

From time to time, IBM issues fixes and corrective service updates to its products that are still being supported. You can download these updates for C for AIX under the “Support downloads” section of the support Web site at:

http://www.ibm.com/software/ad/caix/support.html

You will also find service updates for VisualAge C++ for AIX at:

http://www.ibm.com/software/ad/vacpp/support.html

You can also access the Fix Delivery Center for AIX 5L™ at:

http://techsupport.services.ibm.com/server/aix.fdc where you can search for available fixes and updates based on fileset name or APAR/PTF number.

Once you have downloaded the service update to the AIX system, which you are going to apply the service update, follow these steps to apply the update:

1. Log on as the root user.
2. If the downloaded files are in compressed tar format (with the .tar.Z suffix), uncompress and untar with the following commands:

   ```
   # uncompress filename.tar.Z
   # tar -xvf filename
   ```

3. Start the SMIT dialog with the following command:

   ```
   # smit install_latest
   ```

4. Type the directory where the downloaded files reside as the **INPUT device / directory for software** and press Enter.
5. Press the F4 key to generate a list of available filesets, or press Enter to install the full product update.
6. If F4 is selected on the previous step, select the desired filesets by highlighting them and then pressing the F7 key.
7. Press the Enter key once the required filesets have been selected.
8. Press the Enter key to start the update.
9. Press the Enter key to continue the update.
10. Press the F10 key to exit once the update has completed.

If you installed the compiler to a non-default directory as described in 1.3.2, “Retaining a previous version of the compiler” on page 22, follow these instructions to apply the update:

1. Create a text file listing the filesets you want to update, one fileset per line.
2. Execute the vacndi or vacppndi script with the following command:

   ```
   # perl /usr/vac/bin/vacndi -d source_path -b target_directory \
       -u update_file
   ```

   where *source_path* is where the downloaded filesets are located, and *target_directory* is the directory that contains the installation. *update_file* is the file created in step 1 above. To update VisualAge C++ for AIX, replace /usr/vac/bin/vacndi with /usr/vacpp/bin/vacppndi in the command in step 2.

# 2

# Compiling and linking

Developers porting code to the AIX operating system from other UNIX operating systems might, at first, have difficulties with the compile and linking tasks on AIX. This chapter helps the developers with the tasks on AIX by providing the following sections:

- Section 2.1, “32- and 64-bit development environments” on page 38
- Section 2.2, “Compiling and linking: A quick overview” on page 43
- Section 2.3, “Resolving symbols at link-time” on page 53
- Section 2.4, “Supported link methods on AIX” on page 63
- Section 2.5, “Run-time linking” on page 68
- Section 2.6, “Dynamic loading” on page 82
- Section 2.7, “Commands when manipulating objects and libraries” on page 85
- Section 2.8, “Creating shared objects” on page 92
- Section 2.9, “Shared libraries in a development environment” on page 99

For further information about the compile and linking tasks on AIX and how to manage shared libraries, please refer to *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* and the `ld` command section in *AIX 5L Version 5.2 Reference Documentation: Commands Reference*.

> **Note:** The definitions shown in Table B-1 on page 443 are very useful in understanding the technical details of linking and loading process on AIX.

## 2.1 32- and 64-bit development environments

AIX, together with the C and C++ compilers, offer two different programming models:

- ILP32
- LP64

ILP32, which stands for integer/long/pointer 32, is the native 32-bit programming environment for AIX. It provides a 32-bit address space, with a theoretical memory limit of 4 GB.

LP64, or long/pointer 64, is the 64-bit programming environment for AIX. It can address memory beyond the 4 GB limit by providing a 64-bit address space. In general, except for the data type size and alignment difference, LP64 supports the same programming features as the ILP32 model, and backward compatibility with the most widely used int data type.

According to the C and C++ language standards, int and short should be at least 16 bits, and long should be at least as long as int, but not smaller than 32 bits. This relationship among the integral data types still holds true in the LP64 model:

```
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
```

The LP64 data model is the *de facto* standard on 64-bit UNIX-based systems provided by all major system vendors. Applications that transition to the LP64 data model are therefore highly portable to other LP64 vendor platforms.

Table 2-1 on page 39 lists the basic C and C++ data types on AIX and their corresponding sizes in bits for both the ILP32 and LP64 programming models.

*Table 2-1 C and C++ data type sizes in bits*

| Data type | ILP32 | LP64 |
|---|---|---|
| char | 8 | 8 |
| short | 16 | 16 |
| int | 32 | 32 |
| long | 32 | 64 |
| long long | 64 | 64 |
| pointer | 32 | 64 |
| float | 32 | 32 |
| double | 64 | 64 |
| long double[a] | 64 or 128 | 64 or 128 |

a. The size of long double is controlled by the -qlongdouble option, or when you invoke the compiler with the 128 suffix (see Table 1-4 on page 30).

### 2.1.1 The 64-bit advantage

The primary objective for developing in 64-bit is to take advantage of the newer, faster 64-bit hardware[1] and operating systems[2], and to make complex, memory demanding applications (for example, database and scientific computational applications) perform more efficiently. It offers the following benefits not found in 32-bit systems:

- Full 64-bit addressing that expands the address space available to applications beyond the 4 GB limit
- Large process data space mapped in a large virtual address space
- Support for large data structures and executable files
- Large file support using standard system library calls
- Large file caches on systems with large physical memory
- 64-bit data elements with instructions for performing efficient arithmetic and logical computations as operations, using full-register widths, the full-register set, and new instructions
- Greater scalability of system derived data types, for example, time_t and dev_t

[1] Refer to 3.1.1, “How to determine hardware bit mode” on page 107.

[2] Refer to 3.1.2, “How to determine kernel bit mode” on page 107.

In addition, by keeping data in memory rather than writing out to disk, I/O bound applications can realize improved performance, since disk I/O is usually more time-consuming than memory access.

## 2.1.2 Compiler support

The compiler drivers (see 1.7.1, "Default compiler drivers" on page 30) by default invoke the compiler and linker in 32-bit mode. The following features are provided to enable 64-bit development:

- Predefined __64BIT__ macro when invoked for 64-bit compilations
- OBJECT_MODE environment variable
- The -q64 option
- -qarch support for 64-bit suboption

The compilers can be invoked for 64-bit or 32-bit mode by setting an environment variable, or by using a command line option to set the compilation mode of the compiler. Inconsistent options for compilation mode are resolved in the following order:

1. OBJECT_MODE environment variable
2. Configuration file
3. Command line options

Please refer to the *VisualAge C++ for AIX Compiler Reference*, SC09-4959 for more details on the options supported.

### __64BIT__ preprocessor macro

Like many of the features supported by the compilers, the preprocessor macro __64BIT__ is defined when compiling in 64-bit mode. The purpose of the macro is to allow the programmer to select different data structures or lines of code for 32-bit and 64-bit execution in the same source file.

The macro can be tested using conditional directives. For example:

```
#if defined(__64BIT__)
    /* 64-bit specific data structures or code */
#else
    /* 32-bit mode */
#endif
```

This ability to choose execution mode (of the final executable) at compile time using the __64BIT__ macro implies that there is no need to test execution mode at run time. It also eliminates the need of maintaining different variations of common header files. All the library header files are already coded in such a way that they can be compiled regardless of the mode.

## Command line options

The compilers can be invoked for 64-bit or 32-bit mode by setting an 64-bit mode support in the C and C++ compilers is mainly provided by the two compiler options, -q64 and -q32, respectively.

**Note:** Unlike the -X option for the utility commands, the -q32 or -q64 compiler options do not allow a white space after -q; -q64 is valid whereas -q 64 is invalid.

When used in conjunction with the -qarch option, they determine the mode and instruction set for the target architecture. The -q32 and -q64 options take precedence over the setting of the -qarch option. Conflicts between the -q32 and -q64 options are resolved by the last option wins rule. Setting -qarch=com will ensure future compatibility for applications, whereas specific settings will be more hardware architecture dependent.

Please refer to the “Acceptable Compiler Mode and Processor Architecture Combinations” section of the *VisualAge C++ for AIX Compiler Reference*, SC09-4959 for valid combinations of the -q32, -q64, -qarch, and -qtune compiler options.

**Note:** In 64-bit mode, -qarch=com is treated the same as -qarch=ppc.

## OBJECT_MODE environment variable

Having to specify an option every time you compile can become cumbersome and prone to mistakes (for example, inadvertently mixing 32-bit and 64-bit objects in the link edit step, which is not supported). If you are always compiling in one mode during a development session, you can set the OBJECT_MODE environment variable to change the default compilation mode. Permissible values for the OBJECT_MODE environment variable are:

| | |
|---|---|
| **(unset)** | Generate and/or use 32-bit objects. |
| **32** | Generate and/or use 32-bit objects. |
| **64** | Generate and/or use 64-bit objects. |
| **32_64** | Accept both 32- and 64-bit objects. |

> **Note:** The compiler and linker do not support OBJECT_MODE=32_64, and using this choice will generate the error message:
>
> ```
> 1501-254 OBJECT_MODE=32_64 is not a valid setting for the compiler.
> ```

The benefit of using OBJECT_MODE to control the development environment is that other utilities that are often used during development are also sensitive to this environment variable.

> **Note:** The use of OBJECT_MODE to determine the default mode can cause serious problems if a user is unaware of the current setting. For example, the user may not be aware that OBJECT_MODE has been set to 64 and may unexpectedly obtain 64-bit object files from a compiling source file that is not designed for 64-bit. We strongly urge users to be aware of the setting of OBJECT_MODE at the time and to set OBJECT_MODE to ensure that the compiler is invoked for the correct mode.

### Linker command line options

The linker, `ld`, also supports 64-bit link editing with the -b64 option as well as the default 32-bit link editing with -b32. Since the compiler drivers automatically sets the linker option properly based on the -q64 and -q32 options, it is not necessary to specify these linker specific options.

## 2.1.3 Utility commands support

The following utility commands that deal with object files, by default, assumes the object file is in 32-bit XCOFF[3] format:

**`ar`** Maintains the indexed libraries used by the linkage editor.

**`dump`** Dumps selected parts of an object file.

**`lorder`** Finds the best order for member files in an object library.

**`nm`** Displays information about symbols in object files, executable files, and object-file libraries.

**`ranlib`** Converts archive libraries to random libraries.

**`size`** Displays the section sizes of the XCOFF object files.

**`strip`** Reduces the size of an XCOFF object file by removing information used by the binder and symbolic debug program.

[3] The eXtended Common Object File Format, XCOFF, is the object format for AIX. For a detailed description of the XCOFF format, see the *AIX 5L Version 5.2 Files Reference*.

To support the 64-bit XCOFF object format, these utility commands have been enhanced with the -X option. The -X option specifies the type of object file the utility should examine, and accepts one of the following values:

| | |
|---|---|
| **32** | Processes only 32-bit object files |
| **64** | Processes only 64-bit object files |
| **32_64** | Processes both 32-bit and 64-bit object files |

**Note:** These utility commands also refer to the OBJECT_MODE environment value; however, the -X option overrides the environment value setting.

Before developing your 64-bit application, you must confirm whether the libraries that your application depends on are provided in 64-bit. Most C and C++ libraries provided by AIX are hybrid mode archives (both 32- and 64-bit objects are included). However, this may not be the case with third party vendor libraries. See “32- and 64-bit objects” on page 48 and “Hybrid mode library archives” on page 48 for further information about this topic.

## 2.2 Compiling and linking: A quick overview

This section provides a quick overview for developers about compiling and linking on AIX.

### 2.2.1 Building C and C++ programs with system libraries

It is quite straightforward to build (compiling and linking) C and C++ programs with system libraries on AIX. For example, to compile and link the very simple C program shown in Example 2-1, do the following:

```
$ cc helloworld.c
```

*Example 2-1 helloworld.c*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World\n");
    exit(0);
}
```

Then, the compiler driver[4] will generate an executable file in the current directory (if the executable file name is not specified with the -o option, the compiler driver uses the default executable file name a.out):

```
$ ls -l a.out
-rwx------   1 k5       k5            4429 Apr 09 16:18 a.out
```

If you execute the generated executable file, it prints the following output as expected:

```
$ ./a.out
Hello World
```

Internally, the compiler driver **cc** does the following processes in this example:

1. Pick up the default compiler and linker options listed in /etc/vac.cfg (see Example 2-2), then invokes several internal programs.
2. Generate an object file (in this example, the file name would be helloworld.o).
3. Invoke the linker, **ld**, and it links the object file with default libraries listed in /vac/vac.cfg, in order to generate the executable file, a.out.
4. Remove the object file helloworld.o and temporary files.

*Example 2-2 The cc compiler driver stanza in /etc/vac.cfg*

```
* C compiler, extended mode
cc:     use         = DEFLT
    crt        = /lib/crt0.o
    mcrt       = /lib/mcrt0.o
    gcrt       = /lib/gcrt0.o
    libraries  = -L/usr/lpp/xlopt,-lxlopt,-lc
    proflibs   = -L/lib/profiled,-L/usr/lib/profiled
    options    = -qlanglvl=extended,-qnoro,-qnoroconst
```

These internal processes are shown if the -v compiler option is specified, as shown in the following example:

```
$ cc -v helloworld.c
exec:
/usr/vac/exe/xlcentry(/usr/vac/exe/xlcentry,-D_AIX,-D_AIX32,-D_AIX41,-D_AIX43,-
D_AIX50,-D_AIX51,-D_AIX52,-D_IBMR2,-D_POWER,-qlanglvl=extended,-qnoro,-qnorocon
st,-ohelloworld.o,helloworld.c,/tmp/xlcW0m.58Ea,/tmp/xlcW1m.58Eb,/dev/null,hell
oworld.lst,/dev/null,/tmp/xlcW2m.58Ec,NULL)
exec:
/usr/vac/exe/xlCcode(/usr/vac/exe/xlCcode,-qlanglvl=extended,-qnoro,-qnoroconst
,/tmp/xlcW0m.58Ea,/tmp/xlcW1m.58Eb,helloworld.o,helloworld.lst,/tmp/xlcW2m.58Ec
,NULL)
```

[4] Select the appropriate compiler driver as explained in 1.7.1, “Default compiler drivers” on page 30.

```
exec:
/bin/ld(/bin/ld,-b32,/lib/crt0.o,-bpT:0x10000000,-bpD:0x20000000,helloworld.o,-
L/usr/lpp/xlopt,-lxlopt,-lc,NULL)
unlink: helloworld.o
unlink: /tmp/xlcW0m.58Ea
unlink: /tmp/xlcW1m.58Eb
unlink: /tmp/xlcW2m.58Ec
```

### Specifying a library name with the -l linker option

If your program needs to be linked with system libraries other than libc.a, specify the library name using the -l linker option.

**Note:** The standard C library, libc.a, is defined in the compiler driver stanza as an automatically linked library in /etc/vac.cfg, as highlighted in Example 2-2 on page 44. Therefore, you do not have to specify the -lc option in most cases.

For example, the following example shows you how to link a user program, foo.c, with the system provided mathematical library, /usr/lib/libm.a:

```
$ cc foo.c -lm
```

If the -l linker option is used, the linker treats it as a part of a library name. In this example, the linker automatically adds “lib” in front of “m” and adds “.a” after “m”, in order to complete the library file name. This is a generic behavior of linker on most UNIX operating systems.

When searching libraries to resolve symbols, the linker always looks into two directories, /usr/lib and /lib[5], where most system libraries are installed.

This option can be specified more than once in the same command line. In the following example, libabc.a and libdef.a will be searched from the /usr/lib or /lib directories to resolve symbols:

```
$ cc foo.c -labc -ldef
```

To find the appropriate system library name, which contains functions or variables you are going to use in your application, please consult the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions*.

## 2.2.2 Objects and libraries

Compilers normally generate an object file from an input program source file. The term *object file* is a generic term for a file containing executable code, data, relocation information, a symbol table, and other information. Objects files are

[5] The /lib directory is actually a symbolic link to the /usr/lib directory on AIX.

defined by XCOFF[6] (eXtended Common Object File Format) on AIX. Multiple object files can be archived into a single *library archive file*, which is sometimes simply called *library*. The merit of creating a library is that it is easy to handle a fewer number of library files than many object files. Once a library is created from several object files, you can erase those object files, as long as the program source files are kept.

For example, assuming that there are three C program source files, foo1.c, foo2.c, and foo3.c, in the current directory, to generate object files from these source files, do the following:

```
$ cc -c foo1.c foo2.c foo3.c
```

The **cc** compiler driver would generate three object files as follows:

```
$ ls *.o
foo1.o  foo2.o  foo3.o
```

To archive them into a library, named libfoo.a, do the following:

```
$ ar -vq lifoo.a foo1.o foo2.o foo3.o
ar: Creating an archive file lifoo.a.
q - foo1.o
q - foo2.o
q - foo3.o
```

**Note:** If the libfoo.a library does not exist, this command creates it and enters copies of the files foo1.o, foo2.o, and foo3.o into it. If the libfoo.a library does exist, then this command adds the new members to the end of the library archive without checking for duplicate members. The -v option sets verbose mode, in which the **ar** command displays progress reports as it proceeds.

Figure 2-2 on page 50 illustrates these processes to create a library.

[6] For the complete definition of XCOFF, refer to the *AIX 5L Version 5.2 Files Reference*.

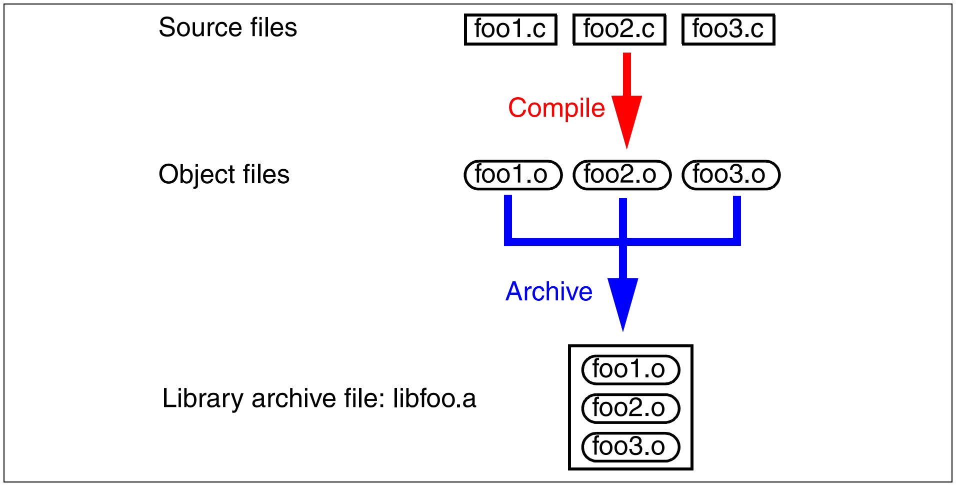


*Figure 2-1 Object files and a library archive*

Once the library is created, it can be maintained using the **ar** command as follows:

- To list the table of contents of a library, enter:

  ```
  ar -v -t libfoo.a
  ```

  This command lists the table of contents of the libfoo.a library, displaying a long listing similar to the output of the **ls -l** command. To list only the member file names, omit the -v option.

- To replace or add new members to a library, enter:

  ```
  ar -v -r libfoo.a foo1.o foo4.o
  ```

  This command replaces the foo1.o member and adds foo4.o to the end of the library.

- To update a member that has been changed, enter:

  ```
  ar -v -r -u libfoo.a foo2.o
  ```

  This command replaces the existing foo2.o member, but only if the foo2.o file has been modified since it was last added to the library.

- To extract library members, enter:

  ```
  ar -v -x libfoo.a foo1.o foo3.o
  ```

  This command copies the foo1.o and foo3.o members into individual files named foo1.o and foo3.o, respectively.

- To delete a member, enter:

  ```
  ar -v -d libfoo.a foo2.o
  ```

This command deletes the foo2.o member from the libfoo.a library.

**Note:** The object files and libraries explained in this section are static (non-shared). Shared object files and libraries must be created and treated by specific ways, as explained in 2.8, “Creating shared objects” on page 92.

For further information about the usage of the **ar** command, please consult with *AIX 5L Version 5.2 Reference Documentation: Commands Reference*.

### 32- and 64-bit objects

As explained in 2.1, “32- and 64-bit development environments” on page 38, AIX provides two different development environments: 32- and 64-bit. Therefore, object files on AIX have two different bit modes: 32-bit object files and 64-bit object files. To build executable files, all the participated object files and archive members must be in the same bit mode, either 32 or 64.

To distinguish the bit mode of object files, use the **file** command as follows:

```
$ file *.o
foo2.o:        executable (RISC System/6000) or object module not stripped
foo3.o:        executable (RISC System/6000) or object module not stripped
foo4.o:        64-bit XCOFF executable or object module not stripped
```

In this example, foo4.o is a 64-bit object, whereas foo2.o and foo3.o are 32-bit.

### Hybrid mode library archives

A library archive can contain both the 32- and 64-bit object modules as its members on AIX. This is called a hybrid mode library archive. In fact, most system libraries provided by AIX are hybrid mode.

As explained in 2.1.3, “Utility commands support” on page 42, utility commands that deal with object files, such as **ar**, **dump**, and **nm**, have been enhanced with the -X option in order to support the 64-bit XCOFF format object format.

For example, to list only 64-bit object modules contained in the standard C library, libc.a, do the following:

```
$ ar -X 64 -t /usr/lib/libc.a
frexp_64.o
itrunc_64.o
ldexp_64.o
modf_64.o
logb_64.o
scalb_64.o
finite_64.o
... rest of output is omitted on purpose ...
```

If you omit the -X 64 option, or specify -X 32, the command lists only 32-bit object modules contained in libc.a. If -X 32_64 is specified, the command lists both 32- and 64-bit object modules.

To determine if the functions you require are provided in 64-bit, use the **nm** command. The mathematical library (libm.a) provided by AIX supports both a 32-bit and a 64-bit version of the acos() sub-routine in the following example output[7]:

```
$ nm -X 32 -g /usr/lib/libm.a | head -5
/usr/lib/libm.a[acos.o]:
._Errno              U           -
.acos                T           0
acos                 D         704      12
guesses              U           -
$ nm -X 64 -g /usr/lib/libm.a | head -7
/usr/lib/libm.a[acos_64.o]:
._Errno              U           -
._restf27            U           -
._savef26            U           -
.acos                T           0
acos                 D         744      24
guesses              U           -
```

## 2.2.3 Difference between shared object and library on AIX

If you are already familiar with other UNIX operating systems, there is nothing to be explained in the simple example used in 2.2.1, “Building C and C++ programs with system libraries” on page 43. However, it is worth mentioning that the program is linked with system provided several libraries, as shown in the following **ldd**[8] command output:

```
$ ldd ./a.out
./a.out needs:
         /usr/lib/libc.a(shr.o)
         /unix
         /usr/lib/libcrypt.a(shr.o)
```

The two system libraries, libc.a and libcrypt.a, are needed by the executable file and they both contain a shared object, named shr.o; in other words, these two objects are *dependent modules* of the executable file. Although both libraries coincidently contain the same name object file in this case, the two shared objects with the same name shr.o are different modules. Most system libraries provided by AIX contain one or more shared objects.

[7] The -g option instructs the **nm** command to handle all archive members contained in the specified library archive.

[8] See 2.7.3, “ldd” on page 90 for further information about the **ldd** command.

Figure 2-2 illustrates the compile and link processes of our example program and references to shared objects contained in system libraries.

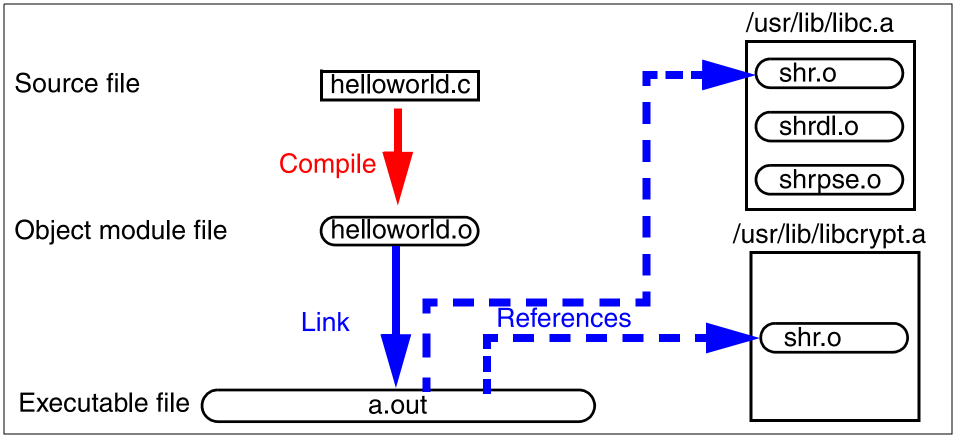


*Figure 2-2 Compiling and linking*

As shown in the following **genkld**[9] command output, there are three shared object modules loaded into the system global memory from the libc.a system library on our test system:

```
$ genkld | grep 'libc.a'
        d2023070            1df /usr/lib/libc.a/dl.o
        d012cd7a            1da /usr/lib/libc.a/pse.o
        d01cfbe0         1e6257 /usr/lib/libc.a/shr.o
```

Although, our example program shown in Example 2-1 on page 43 needs only shr.o out of libc.a, the other two have been loaded since other programs needed them.

In fact, this is the marked difference between AIX and other UNIX operating systems. On other UNIX operating systems, shared libraries (also often referred to as *dynamic link libraries* or DLL) are actually shared objects, whereas multiple shared objects can be contained in a single library on AIX.[10]

[9] The **genkld** command is used to list already loaded shared objects into the system memory. See 2.7.2, “genkld” on page 88 for further information about this command.

[10] Technically, a shared object contained in a shared library archive is referred to as an archive member that has been archived into the library from a separate object file with SHROBJ.

The terms, shared library and shared object, are generally used interchangeably on other UNIX operating systems, whereas there is a distinct difference between the two terms on AIX:

**Shared object** A shared object is a single object file that has the SHROBJ flag in the XCOFF header (see Example 2-3 on page 52). A shared object normally has a name of the form *name*.o on AIX. This is the default file name extension generated by compilers.

**Shared library** A shared library refers to an ar format archive library file[11], where one or more of the archive members is a shared object. Note that the library can also contain regular, non-shared object files, which are handled in the normal way by the linker. A shared library normally has a name of the form lib*name*.a on AIX.

The magic number of the file is used by the linker to determine whether the file is valid object file or not. Therefore, it is possible not to use .o as the file name extension for shared objects, though it could be misleading. As for library name naming convention (lib*name*.a), it is strongly recommended to use it; otherwise the linker cannot find the library location if the -l option is used.

### 2.2.4 Difference between shared and static objects on AIX

The AIX linking and loading mechanism uses an unique file name convention for the shared and static object. On many UNIX operating systems, a shared object file normally has a file name extension ".so" (stands for *shared object*) and a static object file has normally a file name extension ".o" (stands for object).

However, an object normally has a file name extension ".o" regardless of shared or static on AIX. Therefore, you cannot determine whether an object is shared or regular static object from the file name extension on AIX.

> **Note:** AIX also supports shared objects with a file name extension ".so" like other UNIX operating systems do, which are used for run-time linking (see 2.5, "Run-time linking" on page 68).

To determine whether the object file is shared or static, use the **dump** command. As highlighted in Example 2-3 on page 52, if the SHROBJ keyword is shown in the Flags line, then the object file is shared; otherwise, it is static.

[11] An ar format archive file is a file that is created by the **ar** command.

*Example 2-3 SHROBJ flag in the XCOFF header of a shared object*

```
$ dump -ov shr.o

shr.o:

                        ***Object Module Header***
# Sections      Symbol Ptr      # Symbols      Opt Hdr Len     Flags
         5      0x00251764          26925                72     0x3002
Flags=( EXEC DYNLOAD SHROBJ )
Timestamp = "Feb 03 08:59:14 2003"
Magic = 0x1df  (32-bit XCOFF)

                        ***Optional Header***
Tsize        Dsize       Bsize       Tstart      Dstart
0x00171bc0  0x00045ae0  0x00045bc8  0x00000000  0x00000000

SNloader     SNentry     SNtext      SNtoc       SNdata
0x0004      0x0000      0x0001      0x0002      0x0002

TXTalign     DATAalign   TOC         vstamp      entry
0x0005      0x0003      0x00043ad0  0x0001      0xffffffff

maxSTACK     maxDATA     SNbss       magic       modtype
0x00000000  0x00000000  0x0003      0x010b        RE
```

To determine whether archive members in a library are shared or static, use the **dump** command with the -g option.[12] For example, the frexp.o archive member is static, whereas shr.o is shared in the /usr/lib/libc.a library, as shown in Example 2-4.

*Example 2-4 SHROBJ flag in the XCOFF header of archive members in a library*

```
$ dump -gov /usr/lib/libc.a

/usr/lib/libc.a[frexp.o]:

                        ***Object Module Header***
# Sections      Symbol Ptr      # Symbols      Opt Hdr Len     Flags
         3      0x0000030c             34                28     0x0000
Flags=( )
Timestamp = "Sep 15 16:12:35 2002"
Magic = 0x1df  (32-bit XCOFF)

                        ***Optional Header***
Tsize        Dsize       Bsize       Tstart      Dstart
0x00000108  0x00000080  0x00000000  0x00000000  0x00000108
```

[12] The -g option instructs the **dump** command to examine archive members in the specified library.

```
... many output lines are omitted on purpose ...
/usr/lib/libc.a[shr.o]:

                        ***Object Module Header***
# Sections      Symbol Ptr      # Symbols       Opt Hdr Len     Flags
         5      0x00250c0c          26913                72     0x3002
Flags=( EXEC DYNLOAD SHROBJ )
Timestamp = "Sep 19 00:14:43 2002"
Magic = 0x1df  (32-bit XCOFF)

                        ***Optional Header***
Tsize        Dsize       Bsize       Tstart      Dstart
0x00171360  0x00045a08  0x00045bb8  0x00000000  0x00000000

SNloader     SNentry     SNtext      SNtoc       SNdata
0x0004      0x0000      0x0001      0x0002      0x0002

TXTalign     DATAalign   TOC         vstamp      entry
0x0005      0x0003      0x00043a00  0x0001      0xffffffff

maxSTACK     maxDATA     SNbss       magic       modtype
0x00000000  0x00000000  0x0003      0x010b        RE
... rest of output is omitted on purpose ...
```

## 2.3 Resolving symbols at link-time

On AIX, symbol resolution is performed at link-time and cannot be rebound at the program load-time, except for the two exceptions explained in the following sections:

- 2.5, “Run-time linking” on page 68
- 2.6, “Dynamic loading” on page 82

This means that once an executable file was generated, dependent shared objects and archive members in libraries must be referenced using the same path name all the time; otherwise, the executable cannot run.

However, this does not necessary mean that dependent shared objects and archive members in libraries must be in the same directory all the time. If no path information is associated those objects and members, the system loader will look for them in several directories specified by the -L linker option (see 2.3.1, “The -L linker option” on page 55) and the LIBPATH environment variable (see 2.3.3, “LIBPATH environment variable” on page 58), in addition to the default library search directories, /usr/lib and /lib.

This mechanism simplifies the work of the system loader when a module is loaded, and thus results in better execution performance, though it could be seen rigid and tedious from the application programmers' view.

To demonstrate this behavior, we have copied libc.a to /tmp and build an executable file from the program shown in Example 2-1 on page 43 as follows:

```
$ cp /usr/lib/libc.a /tmp
$ ls -l /tmp/libc.a
-r-x------   1 k5       k5          6793964 Apr 18 15:35 /tmp/libc.a
$ chmod a+r /tmp/libc.a
$ ls -l /tmp/libc.a
-r-xr--r--   1 k5       k5          6793964 Apr 18 15:35 /tmp/libc.a
$ cc helloworld.c /tmp/libc.a
```

**Note:** Shared objects or libraries must be readable from other users; otherwise they are treated as private shared objects (see 2.9.2, "Private shared objects" on page 101 for more detail).

The executable file cannot run if /tmp/libc.a is removed as follows:

```
$ ./a.out
Hello World
$ rm -i /tmp/libc.a
rm: Remove /tmp/libc.a? y
$ ./a.out
exec(): 0509-036 Cannot load program ./a.out because of the following errors:
        0509-150   Dependent module /tmp/libc.a(shr.o) could not be loaded.
        0509-022 Cannot load module /tmp/libc.a(shr.o).
        0509-026 System error: A file or directory in the path name does not
exist.
```

The reason is that the reference information to the removed /tmp/libc.a file was statically stored in the XCOFF header of the executable file, as emphasized in Example 2-5 on page 55.

The directory path names, except for the first (index 0) section, shown in the PATH column are called *optional path components* for dependent modules. In Example 2-5 on page 55, libc.a has the optional path component /tmp.

*Example 2-5 dump -H output with the full directory path name*

```
$ dump -H a.out

a.out:

                        ***Loader Section***
                      Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000007       0x00000010       0x00000031

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000002       0x00000188       0x0000002a       0x000001b9


***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lpp/xlopt:/usr/lib:/lib
1      /tmp                          libc.a              shr.o
```

> **Note:** It is always recommended to avoid having any optional path components when creating shared objects and libraries and building executable files, in order to avoid unnecessary dependency to the file path names of the dependent modules.

If the same program is built as follows:

```
$ cc helloworld.c
```

the XCOFF header will not contain any path name information for libc.a as highlighted in the following (the compiler driver automatically add -lc before invoking the linker in this case):

```
***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lpp/xlopt:/usr/lib:/lib
1                                    libc.a              shr.o
```

Since libc.a has no path information, the system loader will look for this library from the directories (/usr/lpp/xlopt:/usr/lib/:/lib) listed in the first (index 0) loader header section of the XCOFF header of the generated executable file.

## 2.3.1 The -L linker option

If the program is referencing libraries, then use the -L and -l linker options rather than specifying library file path names directly.

For example, if the referenced library, libabc.a, is placed in the /project/test/lib directory, specify the options as follows:

```
$ cc -o a.out main.c -labc -L/project/test/lib
```

The linker adds the /project/test/lib directory in front of the default directory search path in order to look for referenced shared objects and libraries.

In this case, the directory name, /project/test/lib, will be added to in front of the first entry for the first (index 0) loader header section of the XCOFF header of the generated executable file, as shown in Example 2-6.

*Example 2-6 dump -H with the PATH information*

```
$ dump -H a.out

a.out:

                        ***Loader Section***
                      Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000007       0x00000010       0x00000037

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000002       0x00000188       0x0000002a       0x000001bf


                        ***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /project/test/lib:/usr/lpp/xlopt:/usr/lib:/lib
1                                    libabc.a              shr.o
```

When executing the command, if libabc.a is found in the directories shown in the following directories, the command can run:

```
/project/test/lib:/usr/lpp/xlopt:/usr/lib:/lib
```

**Note:** The shared objects and libraries must not have any optional path component information in the XCOFF loader header section to be searched from the directories listed in the first (index 0) loader header section of the executable file.

## 2.3.2 Searching objects and libraries at link-time

The linker can handle two types of files as input: object file or library. Those object files and libraries can be specified in the linker command line explained in this section.

**Note:** The order of libraries and objects specified on the linker command line is not important unless run-time linking (see 2.5, “Run-time linking” on page 68) is used.

## Object files

- Specify the absolute path name for the object file. For example:

  ```
  $ cc -o a.out main.c /prod/obj/shr1.o
  ```

- Specify the relative path name for the object file. For example:

  ```
  $ cc -o a.out main.c ../../prod/obj/shr1.o
  ```

- Specify the file name for the object file, if it resides in the current directory. For example:

  ```
  $ ls main.c shr.o
  main.c    shr.o
  $ cc -o a.out main.c shr1.o
  ```

Except for the last method, the generated executable file would have an optional path component for its dependent shared object shr.o in its XCOFF header, if shr.o is a shared object.

## Libraries

- Specify the absolute path name for the library. For example:

  ```
  $ cc -o a.out main.c /prod/lib/libabc.a
  ```

- Specify the relative path name for the library. For example:

  ```
  $ cc -o a.out main.c ../../prod/lib/libabc.a
  ```

- Specify the file name for the library, if it resides in the current directory. For example:

  ```
  $ ls main.c libabc.a
  libabc.a  main.c
  $ cc -o a.out main.c libabc.a
  ```

- Specify the library installed directory using the **`-L`** and **`-l`** linker options. For example:

  ```
  $ cc -o a.out main.c -L/prod/lib -labc
  ```

Except for the last two methods, the generated executable file would have an optional path component for its dependent shared library libabc.a in its XCOFF header, if libabc.a is a shared library.

**Note:** The linker -bnoipath options instructs the command to not include any file path name information in the resultant module. The default option, -bipath, preserves file path name information.

### 2.3.3 LIBPATH environment variable

If the LIBPATH[13] environment variable is defined, the system loader refers to it in order to search referenced shared objects and libraries that contain shared archive members, when the executable file is invoked. The linker does not refer to LIBPATH in order to look for shared objects and libraries.

The syntax of LIBPATH is:

```
LIBPATH=/path1:/path2:/path3:…
```

If defined, the system loader does the following processes when loading modules:

1. Adds the text string value of LIBPATH in front of the PATH information stored in the first (index 0) loader header section of the XCOFF header of the executable.
2. The system loader will search shared objects and libraries referenced by the executable in the directories in the order of the list created in the previous step.

**Note:** When a non-root user is attempting to run a setuid or setgid executable, only the directories listed in the header section of the executable are searched; the LIBPATH variable is ignored, even if it is set.

For example, if the LIBPATH environment variable is defined as follows when executing the program example used in 2.3.1, “The -L linker option” on page 55:

```
LIBPATH=/project/build/lib
```

then the system loader will search for the referenced shared objects and libraries in the following order:

1. /project/build/lib
2. /project/test/lib
3. /usr/lpp/xlopt
4. /usr/lib
5. /lib

[13] On some other UNIX operating systems the LD_LIBRARY_PATH variable is used for a similar purpose. On AIX, however, LD_LIBRARY_PATH has no meaning.

**Note:** The shared objects and libraries must not have any optional path component information in the XCOFF loader header section to be searched from the directories specified by the LIBPATH environment variable.

For example, assuming the following:

- The shared object shr.o is linked with main.o in order to create the executable a.out.
- Both shr.o and main.o are located in the current directory.

then you can select the following two methods to specify the file name shr.o in the command line to build the executable:

1. `cc -o a.out main.o shr.o`
2. `cc -o a.out main.o ./shr.o`

The executable file generated using the first method would have the loader information for shr.o, as shown in Example 2-7. In this case, the system loader will look for shr.o in the /usr/lpp/xlopt, /usr/lib, and /lib directories. If shr.o is stored in one of these directories, there is no need to set LIBPATH. If shr.o is stored in other than these directories, set the LIBPATH environment variable accordingly.

*Example 2-7   dump -H without dot in PATH*

```
$ dump -H a.out

a.out:

                        ***Loader Section***
                      Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000007       0x00000010       0x00000036

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000003       0x00000188       0x0000002a       0x000001be


                        ***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lpp/xlopt:/usr/lib:/lib
1                                    libc.a              shr.o
2                                    shr.o
```

For example, if shr.o is stored in the current directory, run the executable file after setting the variable as follows:

```
$ LIBPATH=$PWD ./a.out
```

If you have moved shr.o to /project/lib, then do the following:

```
$ LIBPATH=/project/lib ./a.out
```

If the executable is generated using the second method, the last five lines in Example 2-7 on page 59 would be:

```
***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lpp/xlopt:/usr/lib:/lib
1                                    libc.a              shr.o
2      .                             shr.o
```

The dot character in the PATH column for shr.o, which is an optional path component for shr.o, makes a big difference. When executing the executable generated with the second method, the system loader will not refer to LIBPATH nor the directories listed in the first (index 0) loader header section, in order to search shr.o. Therefore, shr.o must be located in the current directory whenever the executable file is invoked.

## Using LIBPATH for modules could not be loaded

If a shared object cannot be found by the system loader when trying to start an executable, an error message similar to the following will be seen:

```
exec(): 0509-036 Cannot load program ex1 because of the following errors:
      0509-022 Cannot load library libone.so.
      0509-026 System error: A file or directory in the path name does not
exist.
```

The missing objects will be listed with 0509-022 error messages. Use the **find** command to search the system for the missing shared objects. If the object is found, try setting the LIBPATH environment variable to include the directory that contains the shared object and restart the application. Also, ensure that the object or library has read permission for the user trying to start the application.

A similar error message is produced when the system loader finds the specified shared objects, but not all of the required symbols can be resolved. This can happen when an incompatible version of a shared object is used with an executable. The error message is similar to the following:

```
exec(): 0509-036 Cannot load program ./example because of the following errors:
      0509-023 Symbol func1 in ex1 is not defined.
      0509-026 System error: Cannot run a file that does not have a valid
format.
```

The unresolved symbols are listed in the 0509-023 message lines. Write down the name of the unresolved symbol (func1), and use the **dump -Tv** command to determine which shared object the executable expects to resolve the symbol from. For example:

```
# dump -Tv example | grep func1
[4]     0x00000000    undef      IMP     DS EXTref libone.a(shr1.o) func1
```

This indicates that the executable is expecting to resolve the symbol func1 from the shared object shr1.o that is an archive member of libone.a. This information can help you start the problem determination process.

**Note:** To solve typical link-time errors, see 6.3, “Diagnosing link-time errors” on page 239.

## 2.3.4 Link-time and load-time

Shared objects and libraries are used in two stages when creating and executing an executable on AIX:

1. At link-time, the link editor (the **ld** command) searches the specified shared objects and libraries to resolve all undefined symbols that are referenced in the generating executable file. If a shared object file or library contains the referenced symbols, the loader section of the XCOFF header of the created executable file should contain a reference to that shared object or library.

   In other words, symbols are *exported* from those shared objects or libraries and *imported* from the executable file.

2. At the program load-time, the system loader (the kernel component that starts new processes) reads the XCOFF header information of the executable and attempts to locate any referenced shared libraries. Assuming all the referenced shared objects and libraries are found, the executable can be started. Then, the system loader attempts to load the sections in the executable file into the appropriate segments in the process address space, as explained in Table 2-2 on page 62. The program text in shared objects and libraries is loaded into the global system memory by the first program that needs it and is shared by all programs that use it.

*Table 2-2 XCOFF headers and loading target segments*

| Program executable components | Corresponding section header name in the XCOFF format | Loading target segment in the process address space |
|---|---|---|
| Program text | .text | Process text segment |
| Program data (initialized and un-initialized) | .data and .bss | Process data segment |
| Referenced shared objects | loader | Shared library text segment (per-process shared library data segment will be also populated with the necessary data for the process) |

For the detailed information about the each segment usage in the process address space, see Chapter 3, "Understanding user process models" on page 105.

Figure 2-3 on page 63 depicts the processes done by the system loader at the program load-time.

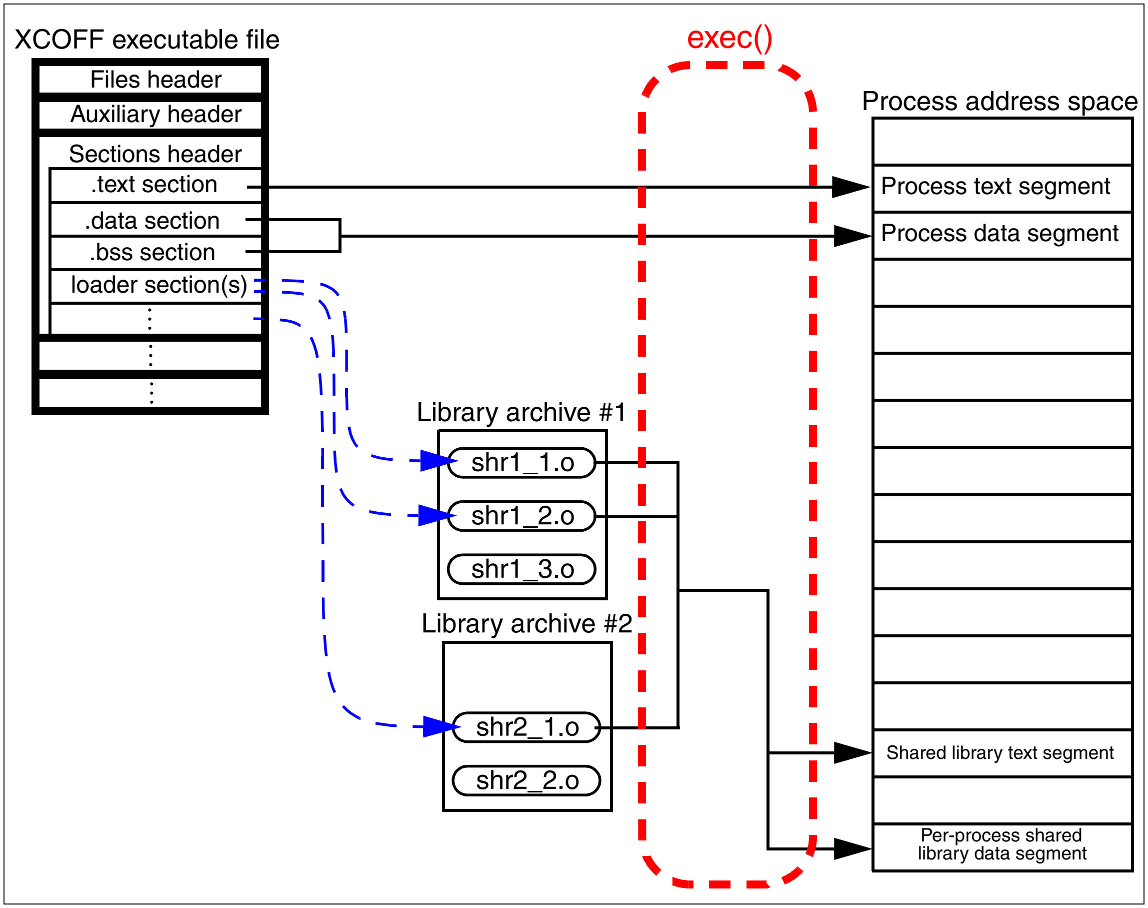


*Figure 2-3 An XCOFF format executable file and exec()*

Technically, each section header in the XCOFF file provides an offset address to the actual section in the file. However, this is not shown in Figure 2-3 to avoid unnecessary complexity.

## 2.4 Supported link methods on AIX

In order to link application program code with objects and libraries, AIX supports several link methods shown in Table 2-3 on page 64 (for the definition of terms used in this table, see Table B-1 on page 443). These methods are not mutually exclusive, except for the combination of lazy loading and run-time link methods. Therefore, an executable can be generated using more than one link methods.

*Table 2-3 Supported link methods*

| Link method | Symbol resolution | Symbol rebound | Module loading | Linker option | Explained section |
|---|---|---|---|---|---|
| Default | link-time | N/A | program load-time | N/A | Section 2.4.1, “AIX default linking” on page 64 |
| Static | link-time | N/A | program load-time[a] | -bstatic | Section 2.4.2, “Static linking” on page 66 |
| Lazy loading[b] | link-time | N/A | run time | -blazy | Section 2.4.3, “Lazy loading” on page 67 |
| Run time | link-time | program load-time | program load-time | -brtl | Section 2.5, “Run-time linking” on page 68 |
| Dynamic loading[c] | run time | | | N/A | Section 2.6, “Dynamic loading” on page 82 |

a. Program text of statically linked objects and archive members are contained in the executable file.
b. The lazy loading is actually a variation of the default link method. Except for the timing when referenced shared modules are loaded, the behavior is quite similar to the default link method.
c. Dynamic loading is a programming scheme provided by a set of sub-routines rather than by linker options or special object file types.

## 2.4.1 AIX default linking

The linker can handle two types of files as input: object file or library. On AIX, object files and archive members in a library can be static or shared, as explained in 2.2.4, “Difference between shared and static objects on AIX” on page 51. Regardless of static or shared, the input file names are specified in the same way in the linker command line in the default link method, as explained in 2.3.2, “Searching objects and libraries at link-time” on page 56.

It is imperative to understand that the default link method is used when generating an executable file, unless the following linker options are explicitly specified:

| | |
|---|---|
| **-bstatic** | Static linking |
| **-blazy** | Lazy loading |
| **-brtl** | Run-time linking |

### Shared and static objects

Static objects are always statically linked and contained in the generated executable file. Shared objects are usually dynamically linked, thus the shared library code is not contained in the generated executable file. However, shared objects can be linked statically (see 2.4.2, “Static linking” on page 66).

Figure 2-4 depicts the difference between static and shared objects or archive members in the generated executable file. In this figure, the following commands are used in order to build the executable file a.out:

- Compile

```
cc -c main.c foo1.c foo2.c
```

- Link

```
cc main1.o foo1.o foo2.o -labc -L/project/lib
```

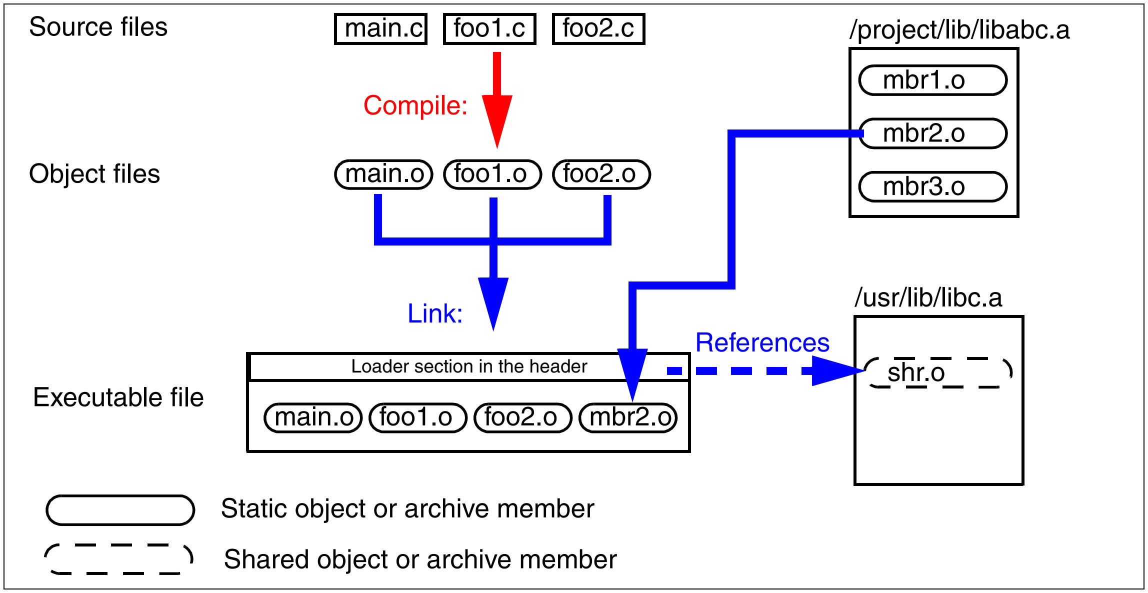


*Figure 2-4 Static and shared text code in the executable file*

If a shared object is linked, its shared program text is loaded into the system shared library segment and shared by all processes that reference it, as depicted in Figure 2-3 on page 63.

## LDR_CNTRL=PREREAD_SHLIB

If not already loaded, a shared object is loaded into memory when the program that depends on the shared object is executed. The loading process is done by the kernel virtual memory manager (VMM) on a demand-page basis, and the actual loading page size depends on the current VMM setting defined by several options, which are set by either the **vmo** command on AIX 5L Version 5.2 or the **vmtune** command on other versions of AIX.

For detailed information about VMM, please refer to the *AIX 5L Version 5.2 Performance Management Guide*.

In some cases, especially if shared libraries are written by C++ and there are many references between these libraries, it may be faster to read the library into memory rather at the program load-time than the default demand-page basis.

In this case, set the following environment variable:

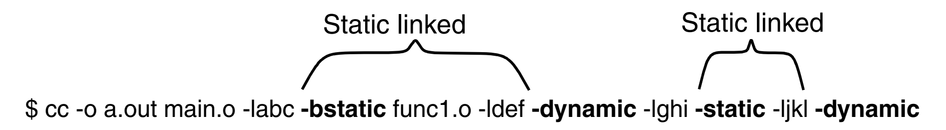
```
LDR_CNTRL=PREREAD_SHLIB
```

## 2.4.2 Static linking

AIX supports the -bdynamic and -bstatic linker options to determine how shared objects and libraries should be treated by the linker.[14] These options are toggles and can be used repeatedly in the same linker command line.

When -bdynamic is in effect, which is the default, shared objects are used in the usual way, whereas when -bstatic is in effect, all referenced objects are linked statically, even if those objects are shared objects.

For example, if a program is built using the command line shown in Figure 2-5, the three command line arguments, func1.o, -ldef, and -ljkl, are treated as static, whereas the other arguments, -labc and -lghi, are treated as shared. Therefore, the object module func1.o and referenced archive members in libraries, libdef.a and libjkl.a, are statically linked with main.o.



```
$ cc -o a.out main.o -labc -bstatic func1.o -ldef -dynamic -lghi -static -ljkl -dynamic
```

*Figure 2-5 The -bdynamic and -bstatic linker options*

**Note:** We are assuming all objects and libraries except for main.o are shared in Figure 2-5. Non-shared objects or archive members are statically linked regardless of the effectiveness of -bdynamic.

If you use the -bstatic option, the -bdynamic option should be specified as the last option on the link line to ensure that the system libraries are treated as shared objects by the linker. Otherwise, all the object members in the system libraries are treated as static, and the executable produced will be larger than normal and may not work on future versions of AIX since it is statically linked with a specific version of system libraries. The -bdynamic added at the end of the command line ensures that the system libraries, such as libc.a, are processed as shared objects.

[14] The -bdynamic and -bstatic linker options has been supported by AIX, starting from Version 4.3.

**Note:** Statically linking shared objects or shared archive members in the 64-bit development environment is not supported.

### 2.4.3 Lazy loading

Lazy loading,[15] which is a variation of the default linking method, is a mechanism for deferring the loading of modules until one of its functions is required to be executed. By default, the system loader automatically loads all of the module's dependants at the same time.

By linking a module with the -blazy linker option, the module is loaded only when a function within it is called for the first time; however, symbol resolution is performed at link-time.

**Note:** Lazy loading only works if the run-time linker (-brtl option) is not specified when building executables. A module is lazy loaded when all references to the module are function calls. If variables in the module are referenced, the module is loaded in the normal way.

Let us assume, for example, that main() calls myfunc1() while myfunc1() is in the libone.so shared module. If myfunc1() calls myfunc2() conditionally in shared module libtwo.so, then libtwo.so is a candidate for lazy loading. If myfunc2() is never called, then libtwo.so is not loaded at all. If myfunc2() *is* called, the lazy loading code executes the load() function to load libtwo.so, patches a function descriptor to make sure subsequent calls simply go directly to the function itself, and then invokes the called function. If for some reason the module cannot be loaded, the lazy loader's error-handler is invoked.

Using lazy loading does not usually change the behavior of a program, but there are the following exceptions:

- Any program that relies on the order that modules are loaded in is going to be affected, because modules can be loaded in a different order, and some modules might not be loaded at all.
- Be careful while comparing function pointers if you are using lazy loading. Usually a function has a unique address in order to compare two function pointers to determine whether they refer to the same function. When using lazy loading to link a module, the address of a function in a lazy loaded module is not the same address computed by other modules. Programs that depend upon the comparison of function pointers should not use lazy loading.

[15] The lazy loading function has been supported by AIX, starting from Version 4.3.

- If any modules are loaded with relative path names and if the program changes working directories, the dependent module might not be found when it needs to be loaded. When you use lazy loading, you should use only absolute path names when referring to dependent modules at link-time.

The decision to enable lazy loading is made at link-time on a module-by-module basis. In a single program, you can mix modules that use lazy loading with modules that do not. When linking a single module, a reference to a variable in a dependent module prevents that module from being loaded lazily. If all references to a module are to function symbols, the dependent module can be loaded lazily.

### Tracing the lazy loading execution

The environment variable LDLAZYDEBUG can be used to trace the lazy loading activity as it takes place. The value of this variable is the sum of one or more of the values shown in Table 2-4.

*Table 2-4 LDLAZYDEBUG environment variable values*

| Value | Description |
|---|---|
| 1 | Show load or look-up errors. If a requested symbol is not available in the loaded referenced module, a message is displayed before the error handler is called. |
| 2 | Write tracing messages to *stderr* instead of *stdout*. |
| 4 | Display the name of the module that is getting loaded. |
| 8 | Display the name of the called function. |

## 2.5 Run-time linking

As explained in 2.4.1, “AIX default linking” on page 64, all referenced symbols must be resolved at link-time when building executable files using the default, static, and lazy loading link methods on AIX. Those pre-resolved symbols cannot be rebound after the executable files are created.

The run-time link method, or run-time linking, enables a program to resolve its referenced symbols at the program load-time rather than link-time. It is the ability to resolve undefined and non-deferred symbols in shared modules after the program execution has already began. It is a mechanism for providing run-time definitions (for example, function definitions that are not available at the program link-time) and symbol rebinding capabilities. For example, if main() calls func1() in libfunc1.so, which then calls func2() in libfunc2.so, assuming that libfunc1.so and libfunc2.so were built to enable run-time linking, then the main application could provide an alternate definition of func2() that would override the one originally found in libfunc2.so.

Please note that it is the main application that has to be built to enable run-time linking. Simply linking a module with the run-time link library is not enough. This structure allows a module to be built to support run-time linking, yet continue to function in an application that has not been so enabled.

**Note:** In AIX, even if the run-time link method is used, all symbols except for the *deferred* symbols (see “Displaying symbol definition with dump -Tv” on page 87 for the definition of deferred symbols) must be resolved at the program load-time. However, in some other UNIX operating systems, resolution of function symbols is deferred until the function is first called (references to variables must be resolved at load-time). This allows the definition for a function to be loaded after the module referring to that symbol is loaded on those operating systems.

In order to use the run-time link method, the following points must be understood:

- The run-time link method is enabled by the run-time link library (/usr/lib/librtl.a) specified by the -brtl option when generating the executable file. The -brtl option is mutually exclusive with the -blazy option.
- Only run-time linking shared objects are run-time linked. Objects or archive members other than run-time shared objects are linked in the default link method.
- Run-time linking shared objects must be created before linking them with the main program, as explained in “Creating run-time linking shared objects” on page 71.
- Run-time linking shared objects should use the file name convention lib*name*.so. If this naming convention is used, it is easy to distinguish run-time linking shared objects from other types of objects and easy to specify the file path name on the command line (see 2.5.5, “Extended search order with the -brtl linker option” on page 81).

An advantage of using run-time linking is that developers do not need to maintain a list of module interdependencies and import/export lists (see 2.8.1, “Import and export files” on page 92). By using the -bexpall linker option, all shared objects can export all symbols, and the run-time linker can be used to resolve the inter-module dependencies.

## 2.5.1 How to use run-time linking

This section explains how to use run-time linking by providing several simple examples.

### Sample program source files

Our example program is composed of four C program source files, main.c (shown in Example 2-8), func1.c (Example 2-9), func2.c, and func3.c (Example 2-10). The source file of func2.c is the same as func3.c, except that the function name is func*2*, not func*3*.

*Example 2-8 Run-time linking example (main.c)*

```
#include <stdio.h>
#include <stdlib.h>

extern void func1(void);

int main(int argc, char *argv[])
{
        func1();
}
```

*Example 2-9 Run-time linking example (func1.c)*

```
#include <stdio.h>
#include <stdlib.h>

void func1(void)
{
        printf("within function %s at line number: %d in %s\n"
                , __FUNCTION__, __LINE__, __FILE__);
        func3();
}
```

*Example 2-10 Run-time linking example (func3.c)*

```
#include <stdio.h>
#include <stdlib.h>

void func3(void)
{
        printf("within function %s at line number: %d in %s\n"
                , __FUNCTION__, __LINE__, __FILE__);
}
```

If this simple application is statically built as follows:

```
$ cc -c main.c func1.c func2.c func3.c
main.c:
func1.c:
func2.c:
func3.c:
$ ls main.o func1.o func2.o func3.o
func1.o  func2.o  func3.o  main.o
$ cc main.o func1.o func2.o func3.o
```

then the generated executable file a.out prints the following output:

```
$ ./a.out
within function func1 at line number: 7 in func1.c
within function func3 at line number: 7 in func3.c
```

Please note the function func2 in func2.c is not called at all in this application. This function is intentionally included in this example in order to show the important aspect of run-time linking in later sections.

## Creating run-time linking shared objects

In order to use run-time linking, run-time linking shared objects must be created before linking them with the main program. To create a run-time linking shared object, do the following:

1. Compile the source file and create an object file.
2. Re-link the object file with the -G linker option and create a run-time linking shared object (.so).

   **Note:** Multiple object files can be combined into a single run-time linking shared object file.

3. Archive the created run-time linking shared object into a library archive (lib*name*.a). This is an optional step.

In fact, these steps are very similar to the steps to create regular (run-time linking disabled) shared objects except for the specified linker options (see 2.8, “Creating shared objects” on page 92).

For example, to create three run-time linking shared objects from func1.c, func2.c, and func3.c, do the following:

```
$ cc -c func1.c func2.c func3.c
func1.c:
func2.c:
func3.c:
```

```
$ ld -G -o func1.so func1.o -bnoentry -bexpall
$ ld -G -o func2.so func2.o -bnoentry -bexpall -lc
$ ld -G -o func3.so func3.o -bnoentry -bexpall -lc
$ ls *.o *.so
func1.o   func1.so  func2.o   func2.so  func3.o   func3.so
```

The following linker options are specified in this example:

**-bnoentry** Indicates that the output file has no entry point. To retain any needed symbols, specify them with the -u flag or with an export file. You can also use the -r flag or the -bnogc or -bgcbtpass options to keep all external symbols in some or all object files. If neither the -bnoentry nor the -bnox option is used and the entry point is not found, a warning is issued.

**-bexpall** Exports all global symbols, except imported symbols, unreferenced symbols defined in archive members, and symbols beginning with an underscore (_). You may export additional symbols by listing them in an export file. This option does not affect symbols exported by the -bautoexp option. This option only applies to AIX Version 4.2 and later.

When you use this option, you may be able to avoid using an export file. On the other hand, using an export file provides explicit control over which symbols are exported, and allows you to use other global symbols within your shared object without worrying about conflicting with names exported from other shared objects. The default is -bnoexpall.

**-lc** Specifies referenced libraries. As for func2.o and func3.o, the standard C library must be specified to resolve the symbol of printf(). In the case of func1.o, there is no need to specify any libraries, since it does not call any functions, except for func3.

**-G** The -G linker option is equivalent to specifying all the options shown in Table 2-5 on page 73.

*Table 2-5 Linker options equivalent to -G*

| Option | Description |
|---|---|
| -berok | Enables creation of the object file, even if there are unresolved references |
| -brtl | Enables run-time linking. All shared objects listed on the command line (those that are not part of an archive member) are listed in the output file. The system loader loads all such shared modules when the program runs, and the symbols exported by these shared objects may be used by the run-time linker. |
| -bsymbolic | Assigns this attribute to most symbols exported without an explicit attribute. |
| -bnortllib | Removes a reference to the run-time linker libraries. This means that the module built with the -G option (which contains the -bnortllib option) will be enabled for run-time linking, but the reference to the run-time linker libraries will be removed. Note that the run-time libraries should be referenced to link the main executable only. |
| -bnoautoexp | Prevent automatic exportation of any symbol. |
| -bM:SRE | Build this module to be shared and reusable. |

For the detailed information about these linker options, please refer to the `ld` command section in the *AIX 5L Version 5.2 Reference Documentation: Commands Reference*.

**Note:** Compiler drivers also have the -G option. Do not use the compiler -G option to create run-time linking shared objects from object files.

## Creating an executable with run-time linking shared objects

In order to create an executable linked with run-time linking shared objects, do the following:

1. Compile the main program source file, which contains main(), and create an object file.
2. Re-link this object file with the -G linker option and create a run-time linking shared object (.so).
3. Link the object file compiled from the main program source file with run-time linking shared objects as well as the -brtl linker option specified.

If the object file created from the main program source file does not have be run-time linking enabled, then the required task is shorten as follows:

Use an appropriate compiler driver, not the linker, to compile the main program source file, which contains main(), then link the object file with run-time linking shared objects by specifying the -brtl option.

As for our program example, we have built the executable a.out as follows using the later simple method:

```
$ ls *.o *.so
func1.o   func1.so  func2.o   func2.so  func3.o   func3.so
$ cc main.c func1.so func2.so func3.so -brtl
$ ls a.out
a.out
```

If executed, the program would print the following error message and fail to execute, since the dependent module func1.so could not be found by the system loader:

```
$ ./a.out
exec(): 0509-036 Cannot load program ./a.out because of the following errors:
        0509-150   Dependent module func1.so could not be loaded.
        0509-022 Cannot load module func1.so.
        0509-026 System error: A file or directory in the path name does not
exist.
```

If LIBPATH is set as follows, it executes as expected:

```
$ LIBPATH=$PWD ./a.out
within function func1 at line number: 7 in func1.c
within function func3 at line number: 7 in func3.c
```

If the linker command (**ld**) was used instead of the compiler driver (**cc**) as follows:

```
$ cc -c main.c
$ ld main.o func1.so func2.so func3.so -brtl
```

then the program would print the following error message, since the generated executable would not contain the necessary start up routine:

```
$ LIBPATH=$PWD ./a.out
exec(): 0509-036 Cannot load program ./a.out because of the following errors:
        0509-151 The program does not have an entry point or
                   the o_snentry field in the auxiliary header is invalid.
        0509-194 Examine file headers with the 'dump -ohv' command.
```

## 2.5.2 Examining the executable and shared objects using dump

By using the **dump** command, you can examine whether the generated executable is linked using run-time linking or not. For example, the **dump** command with the -H option shows the header information for our example application a.out, as shown in Example 2-11.

Example 2-11 includes two important lines (highlighted):

- The run-time linking shared object func2.so is included in the dependent module list, though any symbols in this module are not referenced in the application at all.

  **Note:** If this executable was compiled without -brtl, func2.so would not be included in the dependent module list.

- The run-time linker, the archive member shr.o in librtl.a, is included in the dependent module list.

*Example 2-11 dump -H a.out*

```
$ dump -H a.out

a.out:

                        ***Loader Section***
                      Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000009       0x00000011       0x0000005e

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000006       0x000001c4       0x0000002a       0x00000222


                        ***Import File Strings***
INDEX  PATH                         BASE                MEMBER
0      /usr/lpp/xlopt:/usr/lib:/lib
1                                   func1.so
2                                   func2.so
3                                   func3.so
4                                   libc.a              shr.o
5                                   librtl.a            shr.o
```

**Note:** The order of run-time linking shared objects listed in the executable's header is same with the order of object files specified in the linker command line when generating the executable file:

```
$ cc main.c func1.so func2.so func3.so -brtl
```

The **dump** command with the -Tv option shows the referenced symbols information for our example application a.out, as shown in Example 2-12.

A very important fact in Example 2-12 is that there is no entry for the function func3, which is called from func1 in func1.c. As explained in 2.3, "Resolving symbols at link-time" on page 53, all symbols must be resolved at the program link-time, and all the resolved symbol information will be shown in the XCOFF loader section of the generated executable on AIX by default. However, in the case of run-time linking, symbols to be resolved by the run-time linker will not be shown in the XCOFF loader section of the generated executable, thus func3 is *not* shown in Example 2-12.

*Example 2-12   dump -Tv a.out (1)*

```
$ dump -Tv a.out

a.out:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x20000448    .data              RW SECdef        [noIMid] __rtinit
[1]     0x00000000    undef      IMP     RW EXTref   libc.a(shr.o) errno
[2]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) exit
[3]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) __mod_init
[4]     0x00000000    undef      IMP     BS EXTref   libc.a(shr.o) __crt0v
[5]     0x00000000    undef      IMP     BS EXTref   libc.a(shr.o) __malloc_user_defined_name
[6]     0x00000000    undef      IMP     DS EXTref        func1.so func1
[7]     0x00000000    undef      IMP     DS EXTref librtl.a(shr.o) __rtld
[8]     0x20000458    .data    ENTpt     DS SECdef        [noIMid] __start
```

Example 2-13 on page 77 shows the header information for the dependent shared object module func1.so. The double dot characters (..) in the last line indicate that at least one run-time linking shared objects are required in order to resolve unresolved symbols in this module.

*Example 2-13 dump -H func1.so*

```
$ dump -H func1.so

func1.so:

                        ***Loader Section***
                      Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000003       0x00000006       0x00000015

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000002       0x000000b0       0x00000000       0x00000000


                        ***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lib:/lib
1                                    ..
```

Example 2-14 shows the loader section information for the dependent shared object module func1.so. There are two imported symbols, printf and func3, in this module.[16] The double dot characters (..) in the IMPid column indicate that the symbol will be resolved by the run-time linker at the program load-time.

*Example 2-14 dump -Tv func1.so*

```
$ dump -Tv func1.so

func1.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x00000050    .data      EXP     DS SECdef        [noIMid] func1
[1]     0x00000000    undef      IMP     DS EXTref              .. printf
[2]     0x00000000    undef      IMP     DS EXTref              .. func3
```

Figure 2-6 on page 78 depicts the function calling relationship for our application built in this section. The run-time linking shared object func2.so is referenced by the executable file (a.out) and will be loaded into the system memory by the system loader at the program execution time, even if a symbol in this object is not referenced by the executable.

[16] See "Displaying symbol definition with dump -Tv" on page 87 how to interpret columns shown in the **`dump -Tv`** output.

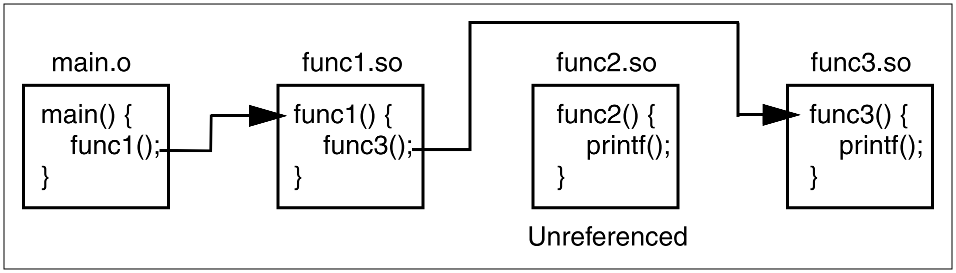


*Figure 2-6 Function calling relationship*

### 2.5.3 Enabling the main program object as run-time linking

If you closely examine Example 2-12 on page 76, you notice that func1.so was not linked as a run-time linking shared object. The IMPid column of the func1 symbol shows the file name of func1.so, thus it is treated as a regular shared object. The reason is that the object file created from the main program source file, which includes main(), had not been run-time linking enabled, therefore all dependent modules that were required to resolve symbols referenced by main.o had to be loaded at the program load-time by the system loader.

In order to enable the main program object as a run-time linking shared object, we could have done the following (highlighted lines and command options are different):

```
$ cc -c main.c func1.c func2.c func3.c
main.c:
func1.c:
func2.c:
func3.c:
$ ld -G -o main.so main.o -bexpall
ld: 0711-327 WARNING: Entry point not found: __start
$ ld -G -o func1.so func1.o -bnoentry -bexpall
$ ld -G -o func2.so func2.o -bnoentry -bexpall -lc
$ ld -G -o func3.so func3.o -bnoentry -bexpall -lc
$ cc -o a.out main.so func1.so func2.so func3.so -brtl
```

After specifying LIBPATH, the program prints the following output as expected:

```
$ LIBPATH=$PWD ./a.out
within function func1 at line number: 7 in func1.c
within function func3 at line number: 7 in func3.c
```

Example 2-15 on page 79 shows the referenced symbol information for a.out generated by the compiler driver. Please note that Example 2-15 on page 79 does not contain the symbol information for either func1 or func3.

*Example 2-15 dump -Tv a.out (2)*

```
$ dump -Tv a.out

a.out:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x200003e8    .data              RW SECdef        [noIMid] __rtinit
[1]     0x00000000    undef      IMP     RW EXTref   libc.a(shr.o) errno
[2]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) exit
[3]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) __mod_init
[4]     0x00000000    undef      IMP     BS EXTref   libc.a(shr.o) __crt0v
[5]     0x00000000    undef      IMP     BS EXTref   libc.a(shr.o) __malloc_user_defined_name
[6]     0x00000000    undef      IMP     DS EXTref         main.so main
[7]     0x00000000    undef      IMP     DS EXTref librtl.a(shr.o) __rtld
[8]     0x200003f8    .data    ENTpt     DS SECdef        [noIMid] __start
```

Although, it is technically possible to link all dependent shared objects in an application using run-time linking, it should be avoided in order for the application performance. In general, to take advantage of the AIX architecture, the shared modules should be as self contained as possible. Run-time linking should be used only when necessary. Application program code ported from other UNIX operating systems often does not have this sort of organization and can therefore require extra effort to enable it on AIX. It is important to emphasize the fact that the performance and efficiency of AIX is best explained by a well organized application structure with a well defined interface between modules.

## 2.5.4 Rebinding symbols at the program load-time

Example 2-16 on page 80 and Example 2-17 on page 80 respectively show the symbol information for func2.so and func3.so used in 2.5.1, “How to use run-time linking” on page 70. Although these objects themselves are run-time linking enabled, they have no dependent run-time linking shared object.

*Example 2-16 dump -Tv func2.so*

```
$dump -Tv func2.so

func2.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]     0x00000050    .data      EXP     DS SECdef        [noIMid] func2
```

*Example 2-17 dump -Tv func3.so*

```
$ dump -Tv func3.so

func3.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]     0x00000050    .data      EXP     DS SECdef        [noIMid] func3
```

If func2.so and func3.so are swapped as follows, the new func2.so will contain the function func3 and the new func3.so will contain func2:

```
$ mv func3.so temp.so
$ mv func2.so func3.so
$ mv temp.so func2.so
```

After func2.so and func3.so swapped, the run-time linker looks for the symbol definition for func3 and found in the new func2.so in the order of func1.so, func2.so, and func3.so. The referenced symbol func3 is rebound at the program load-time without re-linking the application, thus the executable prints the following output:

```
$LIBPATH=$PWD ./a.out
within function func1 at line number: 7 in func1.c
within function func3 at line number: 7 in func3.c
```

Figure 2-7 on page 81 depicts the function calling relationship for our application after swapping func2.so and fun3.so. Now, new func2.so, which was originally func3.so, contains func3, and new func3.so, which was originally func2.so, contains func2.

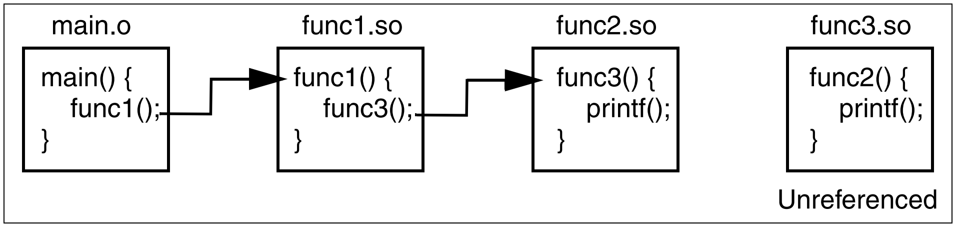


*Figure 2-7 Function calling relationship after rebinding symbols*

The run-time linking shared object func3.so is required by the executable file (a.out) and will be loaded into the system memory by the system loader at the program execution time, even if no symbol in this object is not referenced by the executable.

## 2.5.5 Extended search order with the -brtl linker option

When an executable file is linked in the run-time link method by specifying the -brtl linker option, the linker uses the extended search order to look for the shared objects and libraries to resolve symbols. In order to exploit this extended search order, run-time linking shared objects must follow the file name convention lib*name*.so.

If the -brtl linker option is specified, the linker looks for libname.so, as well as libname.a, in the directories specified by the -L linker option in addition to the default directory search path (/usr/lib and /lib). The -l*name* linker option can be also used to specify libname.so as well as libname.a.

For example, if a run-time linking shared object named libfunc1.so, which is stored in the /project/lib directory, is linked when generating the executable file a.out, there are several methods to specify the object:

- `cc main.c /project/lib/libfunc1.so -brtl`

  This method almost invalidates the merit of the run-time link method. Although libfunc1.so is loaded at the program load-time, it must reside in the /project/lib directory when the executable is invoked.

- `cp /project/lib/libfunc1.so .; cc main.c libfunc1.so -brtl`

  This method is useful if all the run-time linking shared objects are located in the current directory. However, once a stable version is developed, those objects are most likely deployed in a specific directory. Therefore, it is less useful compared to the third method.

- `cc main.c -lfunc1 -L/project/lib -brtl`

  This is the recommended method to generate an executable file using the run-time link method. The linker looks for the specified run-time shared objects in the order explained in 2.3.1, “The -L linker option” on page 55.

  > **Note:** This method can be used only if the run-time shared object’s file name follows the lib*name*.so naming convention.

To follow the libname.so naming convention, either rename the run-time linking shared object files or archive them into a library.

## 2.6 Dynamic loading

Dynamic loading is the process in which one can attach a shared library to the address space of the process during execution, look up the address of a function in the library, call that function, and then detach the shared library when it is no longer needed. It is implemented as an interface to the services of the dynamic linker. This gives programmers extra control in managing the memory allocated to the shared library segment in an effective manner.

The dlopen() family of subroutines is supported on the AIX operating system. The functions include:

- dlopen()
- dlclose()
- dlsym()
- dlerror()

### dlopen()

The dlopen() subroutine is used to open a shared object, and dynamically map it into the running programs address space. The syntax of the subroutine is as follows:

```
#include <dlfcn.h>

void *dlopen (FilePath, Flags);
const char *FilePath;
int Flags;
```

The FilePath parameter is the full path to a shared object, for example, shrobj.o, or libname.so. It can also be a pathname to an archive library that includes the required shared object member name in parenthesis, for example, /lib/libc.a(shr1.o).

The Flags parameter specifies how the named shared object should be loaded. The Flags parameter must be set to RTLD_NOW or RTLD_LAZY. If the object is a member of an archive library, the Flags parameter must be OR'ed with RTLD_MEMBER.

The subroutine returns a handle to the shared library that gets loaded. This handle is then used to with the dlsym subroutine to reference the symbols in the shared object. On failure, the subroutine returns NULL. If this is the case, the dlerror() subroutine can be used to print an error message.

> **Note:** On AIX, the handle from dlopen() is specific to an instance of dlopen(). The dlclose() call only closes the instance of the object to which the handle refers.

## dlsym()

The dlopen() subroutine is used to load the library. If successful, it returns a handle for use with the dlsym() routine to search for symbols in the loaded shared object. Once the handle is available, the symbols (including functions and variables) in the shared object can be found easily. For example:

```
lib_func = dlsym(lib_handle, "locatefn");
error = dlerror();
if (error) {
    fprintf(stderr, "Error:%s \n",error);
    exit(1);
}
```

The dlsym() subroutine accepts two parameters. The first is the handle to the shared object returned from the dlopen() subroutine. The other is a string representing the symbol to be searched for.

If successful, the dlsym() subroutine returns a pointer that holds the address of the symbol that is referenced. On failure, the dlsym subroutine returns NULL. This, again, can be used with the dlerror subroutine to print an error message as shown above.

## dlclose()

The dlclose() subroutine is used to remove access to a shared object that was loaded into the processes' address space with the dlopen subroutine. The subroutine takes as its argument the handle returned by dlopen().

### dlerror()

The dlerror() subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, dlopen, dlsym, or dlclose). The returned value is a pointer to a null-terminated string without a final newline. Once a call is made to this subroutine, subsequent calls without any intervening dynamic loading errors will return NULL.

Applications can avoid calling the dlerror() subroutine, in many cases, by examining errno after a failed call to a dynamic loading routine. If errno is ENOEXEC, the dlerror() subroutine will return additional information. In all other cases, it will return the string corresponding to the value of errno.

**Note:** The dlerror() subroutine is not thread-safe.

## Using dynamic loading subroutines

In order to use the dynamic loading subroutines, an application must be linked with the libdl.a library. The shared objects used with the dynamic loading subroutines can be traditional AIX shared objects or shared objects that have been enabled for run-time linking with the -G linker option.

When the dlopen() subroutine is used to open a shared object, any initialization routines specified with the -binitfini option, as described in 2.8.4, “Initialization and termination routines” on page 99, will be called before dlopen returns. Similarly, any termination routines will be called by the dlclose subroutine.

## Advantages of dynamic loading

Use of dynamic loading allows several benefits for application developers:

- The ability to share commonly-used code across many applications, leading to disk and memory savings.
- It allows the implementation of services to be hidden from applications.
- It allows the re-implementation of services, for example, to permit bug and performance fixes or to allow multiple implementations selectable at run time.

The following example program is used to demonstrate the use of dynamic loading subroutines:

```
/* File: main.c */

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(int argc, char *argv[])
{
    void *handle;
    void (*fct)();

    printf("hello from main()\n");

    if ((handle = dlopen("libfoo.so", RTLD_NOW)) == NULL) {
        perror("dlopen");
        exit(1);
    }
    if ((fct = (void (*)())dlsym(handle, "foo_")) == NULL) {
        perror("dlsym");
        exit(1);
    }
    (*fct)();
    printf("exit from main()\n");
    dlclose(handle);
}

/* File : foo.c - Shared library libfoo.so */

#include <stdio.h>
#include <stdlib.h>

void foo_(void)
{
    printf("hello from foo_()\n");
}
```

Compile the program as follows:

```
$ cc -c foo.c
$ ld -o libfoo.so foo.o -bM:SRE -bnoentry -bexpall -lc
$ cc -o main main.c
```

The program will print the following output if libfoo.so is successfully found and loaded by the dynamic linker:

```
hello from main()
hello from foo_()
exit from main()
```

## 2.7 Commands when manipulating objects and libraries

This section explains how to use the following commands, which are necessary to manipulate objects and libraries on AIX by providing the following subsections:

- Section 2.7.1, “dump” on page 86
- Section 2.7.2, “genkld” on page 88

- Section 2.7.3, “ldd” on page 90
- Section 2.7.4, “nm” on page 90
- Section 2.7.5, “rtl_enable” on page 91
- Section 2.7.6, “slibclean” on page 91

## 2.7.1 dump

The **dump** command is used to examine the header information of executable files and shared objects. The main options that are useful when working with shared libraries are the -H option and the -Tv options.

### Displaying header information with dump -H

Use the **dump -H** command to determine which shared objects an executable or shared object depends on for symbol resolution at run time. The interesting information is in the last section of output, as shown under the `***Import File Strings***` header in Example 2-18.

*Example 2-18 Sample dump -H output*

```
$ dump -H func1.so

func1.so:

                        ***Loader Section***
                      Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x00000003       0x00000006       0x00000023

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000003       0x000000b0       0x00000000       0x00000000


                        ***Import File Strings***
INDEX  PATH                          BASE                MEMBER
0      /usr/lib:/lib
1                                    libc.a              shr.o
2                                    ..
```

The number of INDEX entries will depend on how many shared objects the target depends on for symbol resolution.

The INDEX 0 entry is a colon separated list of directories shown as /usr/lib:/lib in Example 2-18. If the LIBPATH environment variable is not set when the executable is started, the directories listed in the first (index 0) entry will be used in order to look for the shared objects listed in subsequent entries. If LIBPATH is defined, the directories listed in it will be used before the directories in the first (index 0) entry.

The format of the interesting columns is as follows:

**PATH** Optional path name component of the shared object or library. A pathname will be present if a path name was specified on the linker command line. It is recommended to avoid having this optional path name. See 2.3.1, "The -L linker option" on page 55 and 2.3.3, "LIBPATH environment variable" on page 58 for further detailed information how this optional path name component will be selected depending on the linker command line file name selections.

**BASE** The name of the archive library containing the shared archive member, or the name of the shared object itself.

**MEMBER** The archive member name of the shared library. In the case of non-archived shared objects, this column will be blank.

> **Note:** If there is a .. entry in the BASE column, the shared object is enabled for the run-time linking (see 2.5, "Run-time linking" on page 68).

### Displaying symbol definition with dump -Tv

Use the **`dump -Tv`** command to examine the symbol information of a shared object or executable. It displays information on the symbols the object is exporting. It also lists the symbols the object or executable will try and import at load time and, if known, the name of the shared object that contains those symbols. The interesting columns are IMEX, IMPid, and Name, as highlighted in Example 2-19.

*Example 2-19 Sample dump -Tv output*

```
$ dump -Tv func1.so

func1.so:

                        ***Loader Section***

                        ***Loader Symbol Table Information***
[Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

[0]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
[1]     0x00000050    .data      EXP     DS SECdef        [noIMid] func1
[2]     0x00000000    undef      IMP     DS EXTref              .. func3
```

The values in these columns have the following meaning:

**Name** The symbol name.

**IMEX** Determines whether a symbol is exported from the object file or archive member (EXP) or imported from the other object files or archive members (IMP).

**IMPid** For exported symbols, the value is not used and is always [noIMid]. For imported symbols, there are three patterns for the value, as shown in Table 2-6.

*Table 2-6 IMPid values for imported symbols*

| Value | Description |
|---|---|
| File name | If the file name is in the form of lib*abc*.a(*def*.o), then the symbol is imported from the def.o member of the libabc.a library. If the file name is without parentheses, then the symbol is imported from that shared object file. In both cases, the dependent shared object or archive member will be loaded by the system loader upon the program execution time. |
| .. | The symbol will be resolved by the run-time linker. After resolving the symbol, the run-time linker tells the system loader to load an appropriate shared object or archive member. |
| [noIMid] | The symbol is a *deferred* symbol and will not be resolved by the run-time linker. It is the application's responsibility to resolve deferred symbols in order to avoid the application abend due to the unresolved symbols. In order for the programatic symbol resolution, use the dynamic loading (see 2.6, "Dynamic loading" on page 82). |

## 2.7.2 genkld

The **genkld** command is used to list the shared objects that are loaded in the system shared library segment. The output is quite lengthy and has three columns, the virtual address of the object within the system segment, the size of the object, and the name of the file that was loaded, as shown in Example 2-20.

*Example 2-20 genkld output example (1)*

```
$ genkld | pg
Virtual Address            Size File


        d1febce0           19907 /usr/lib/libcurses.a/shr.o
... rest of the output is omitted on purpose ...
```

If the file name does not have a slash character at its end, then it is a shared archive member of a library archive. In Example 2-20, shr.o is a shared archive member of /usr/lib/libcurses.a.

If the file name has a slash character at its end, it is a shared object file, as shown in Example 2-21.

*Example 2-21 genkld output example (2)*

```
$ genkld | head -1; genkld | grep NIS
Virtual Address             Size File
        d00e7000            2237 /usr/lib/security/NIS/
$ ls -l /usr/lib/security/NIS
-r-xr-xr-x   1 root     system         8760 Sep 15 2002  /usr/lib/security/NIS
$ file /usr/lib/security/NIS
/usr/lib/security/NIS:  executable (RISC System/6000) or object module
```

The virtual address of a 32-bit shared object starts from 0xD, whereas the address of a 64-bit shared object typically starts from 0x90000000, as shown in Example 2-22.

*Example 2-22 genkld output example (3)*

```
$ genkld | head -1; genkld | grep 'libc.a'
Virtual Address             Size File
        d01bb3b8             1df /usr/lib/libc.a/dl.o
        d014d0c2             1da /usr/lib/libc.a/pse.o
        d01d0be0          1e5907 /usr/lib/libc.a/shr.o
 90000000022aca0          20d75e /usr/lib/libc.a/shr_64.o
```

The 0xD segment is the system-wide 32-bit shared text segment, which is shared by all the 32-bit processes on the system. As for 64-bit processes, shared objects can be loaded into the system-wide 64-bit shared text segments, which can be located from 0x9000_0000_0000_0000 to 0x9FFF_FFFF_FFFF_FFFF (576 - 640 PB); however, they are most likely loaded into the first segment in this address range.

For further information about the system-wide shared text segments, see 3.2, “The 32-bit user process model” on page 109 and 3.3, “The 64-bit user process model” on page 130.

## 2.7.3 ldd

The **ldd**[17] command lists the shared objects and archive members that will be loaded to start the executable. The following example shows what shared object modules and shared libraries (*dependencies*) are required to run the executable file helloworld:

```
# ldd helloworld

helloworld needs:
         ./libone.so
         ./libtwo.so
         /usr/lib/libc.a(shr.o)
         /usr/lib/librtl.a(shr.o)
```

The command reports dependencies by traversing valid XCOFF header information of the specified executable file.

## 2.7.4 nm

The **nm** command displays information about symbols in the specified file, which can be an object file, an executable file, or shared object. For example:

```
$ nm libone.so

.myfunc1             T          144
.myfunc1             t          144       40
.myfunc2             T            0
.myfunc2             T          184
.myfunc2             t          184       40
.printf              T          104
.printf              t          104       40
TOC                  d           52
_$STATIC             d            0       40
_$STATIC             d           52        4
glink.s              f            -
glink.s              f            -
glink.s              f            -
myfunc1              U            -
myfunc1              d           60        4
myfunc2              D           40       12
myfunc2              d           64        4
printf               U            -
printf               d           56        4
source1.c            f            -
```

[17] The **ldd** command is supported on AIX 5L Version 5.2 and later.

If the file contains no symbol information, the **nm** command reports the fact, but does not interpret it as an error condition. The **nm** command reports numerical values in decimal notation by default. Unlike the **dump** command, **nm** does not display the shared object or archive member name that is expected to supply the symbol.

### 2.7.5 rtl_enable

The **rtl_enable** command is used to convert a shared library that is not enabled for run-time linking into a run-time linking enabled one.

### 2.7.6 slibclean

The **slibclean** command can be used by the root user to unload all shared objects with a use count value of zero from the system shared library segment. This command is useful in an environment when shared libraries are under development. You can run the **slibclean** command followed by the **genkld** command to ensure that the shared objects under development are not loaded in the system shared library segment. This means that any application started after this will automatically use the latest version of the shared objects since the system loader will search for and load them. It also prevents multiple versions of the same objects existing in the system segment.

During the development of shared objects, you may sometimes see an error message similar to the following when creating a new version of an existing shared object:

```
# make libone.so
        cc -O -c source1.c
        cc -berok -G -o libone.so source1.o
ld: 0711-851 SEVERE ERROR: Output file: libone.so
        The file is in use and cannot be overwritten.
make: 1254-004 The error code from the last command is 12.
```

The error message means that the target shared object file is in use and it has been loaded into the system shared library segment. Once loaded, the file is marked as in use, even if the use count is already zero. Running the **slibclean** command will unload all of the unused shared objects from the system. An alternative (and simpler) method of avoiding this problem is to use the **rm -f** command to remove the shared object before creating it.

However, frequent **slibclean** invocation on production systems should be avoided, because it may affect the system performance by unloading frequently used, but unused when the command is issued, shared objects and libraries on the system. It is recommended to issue **slibclean** on production systems that are in a software maintenance phase, especially before the deinstallation of no longer required applications or the updating of installed applications.

> **Note:** The root authority is required to use the `slibclean` command.

# 2.8 Creating shared objects

This section explains how to create shared objects on AIX. Once shared objects are created, you can archive them into a library using the `ar` command (see 2.2.2, "Objects and libraries" on page 45).

In order to create a shared object, the following tasks are required:

1. Compile the source code files and create object files. Use the appropriate compiler driver to do so, for example, `cc -c foo.c`.
2. Create an export file to explicitly control exported symbols from the shared object, which you are going to create.
3. Re-link the created object file(s) to a shared object file. It is possible to combine multiple object files into a single shared object file. The following linker options are used in this step:

   **-bE:*export_file*** Specifies the export file name created in the previous step.

   **-bM:SRE** Marks the resultant object file as a re-entrant shared object module.

   **-bnoentry** Indicates that the output object file has no entry point.
4. Create an import file to explicitly control how exported symbols in the shared object will be imported by other modules. This is an optional step.

> **Note:** Steps 2 and 4 can be skipped if the -bexpall linker option is used. However, if this option is used, you cannot precisely control the symbol exporting (see "The -bexpall linker option" on page 94).

## 2.8.1 Import and export files

An export file is a text file containing a list of symbols. It is used to control which symbols are visible outside the shared object. The symbols not specified in the export file are only visible to other routines within the shared object. The use of export files allows a developer to create a shared object that has a well defined interface. Only the symbols listed in the export file can be referenced by executables and other shared objects that are linked with the object. Export files are normally identified by the file name extension .exp.

An import file is a text file that lists the names of symbols that the shared object may reference. It allows the object to be created *without* the source of those symbols being available. Once an export file is created, an import file can be created from the export file by adding several directive lines starting from "#!". Table 2-7 shows the supported directives for import files.

*Table 2-7 Directive lines for import files*

| Directive | Description |
|---|---|
| #! | (Nothing after the #!) Use null path, null file, and null number. This is treated as a deferred import by the system loader. |
| #!() | Use -bipath, -bifile, and -bimember. This line can be used if the import file is specified as an *InputFile* parameter on the command line. The file must begin with #! in this case. This line can also be used to restore the default name if it was changed by another #! line. |
| #!path/file(member) | Use the specified path, file, and archive member. |
| #!path/file | Use the specified path and file, and a null member. |
| #!file | Use a null path, the specified file, and a null member. At run time, a list of directories is searched to find the shared object. |
| #!(member) | Use -bipath, -bifile, and the specified member. At run time, a list of directories is searched to find the shared object. |
| #!file (member) | Use a null path and the specified file and member. At run time, a list of directories is searched to find the shared object. |
| #!.[1] | (A single dot) This name refers to the main executable. Use this file name when you are creating a shared object that imports symbols from multiple main programs with different names. The main program must export symbols imported by other modules, or loading will fail. This import file name can be used with or without the run-time linker. |
| #!..[1] | (Two dots) Use this name to list symbols that will be resolved by the run-time linker. Use this file name to create shared objects that will be used by programs making use of the run-time linker. If you use a module that imports symbols from .. in a program that was not linked with the -brtllib option, symbols will be unresolved, and references to such symbols will result in undefined behavior. |
| 1. These directive lines have been supported on AIX since Version 4.2. | |

The creation and use of import files are not mandatory except for the following situation:

- Limiting accessible symbols exported from shared objects so that only specified symbols listed in the import file will be imported.
- Creating multiple shared objects that have dependencies on each other (see 2.8.3, “Interdependent shared objects” on page 96).
- Accessing special symbols such as symbols in the main program. In this case, use the #!. directive line in the import file before the symbols that will be imported from the main program.

If run-time linking (explained in 2.5, “Run-time linking” on page 68) or dynamic loading (explained in 2.6, “Dynamic loading” on page 82) is not used, all referenced external symbols must be accessible and resolvable when the module is linked.

If symbols are listed after the #! directive in an import file, the linker considers those symbols to be created declared differently, so they must be programatically resolved and handled by a set of library calls provided by dynamic loading. If symbols are listed after the #!.. directive in an import file, the linker considers those symbols to be declared to be handled by the run-time linker.

### The -bexpall linker option

If you are creating a shared object and want all symbols to be exported, then you do not need to create an export file. You can use the -bexpall linker option, which will automatically export all global symbols (except imported symbols, unreferenced symbols defined in archive members, and symbols beginning with an underscore). Additional symbols may be exported by listing them in an export list. Any symbol with a leading underscore will not be exported by this option. These symbols must be listed in an exports list to be exported.

### The -bexpfull option

The new -bexpfull[18] link option operates like -bexpall, except that it actually exports all symbols and does not skip symbols that begin with an underscore.

### Creating an export list using CreateExportList

You can use the /usr/vac/bin/CreateExportList shell script supplied with the C for AIX Version 6 compiler to automatically generate the symbols that should be included in an export list. It can save a considerable amount of time if you need to create shared objects.

[18] The -bexpfull linker option is supported on AIX 5L Version 5.2 and later and on AIX 5L Version 5.1 with 5100-02 Recommended Maintenance Level and later.

To use this command, do the following:

1. Compile all of the source files that will be included in the shared object.
2. Create a single file that lists the names of all of the object files that will be included in the shared object. For example, create a file called object.list that contains the following lines:

   ```
   source1.o
   source2.o
   ```

3. Invoke **CreateExportList** as follows:

   ```
   /usr/vac/bin/CreateExportList file.exp -f object.list
   ```

   where file.exp is the name of the export file you want to create, and object.list is the file that contains the list of object file names.
4. Edit the resulting export file to remove the symbol names you wish to keep private within the shared object.
5. Edit the export file to include the #!pathname(member) lines at the appropriate positions in order to use the file as an import file.

### 2.8.2 A self-contained shared object

The scenario described in this section for creating a self-contained shared object uses the following source code files:

The file source1.c is as follows:

```
/* source1.c : First shared library source */
void private(void)
{
    printf(“private\n”);
}
int addtot(int a, int b)
{
    int c;
    c = a + b;
    return c;
}
```

The file source2.c is as follows:

```
/* source2.c : Second shared library source */
#include <stdio.h>
int disptot(int a)
{
    printf(“The total is : %d \n”,a);
}
```

To create a self-contained shared object, do the following:

1. Create the object files that will be combined together to create the shared object. This is achieved by using the -c compiler option. For example:

   ```
   cc -c source1.c source2.c
   ```

2. Create an export file that lists the symbol names that should be visible outside the shared object. In this example, the symbols addtotal and displaytotal are the names of the functions that will be called by the main application. The symbol names can also include variable names in addition to function names. Create the libadd.exp export file as follows:

   ```
   addtot
   disptot
   ```

3. Create the shared object shrobj.so with the following command:

   ```
   cc -o shrobj.o source1.o source2.o -bE:libadd.exp -bM:SRE -bnoentry
   ```

   Use the **dump** command to examine the exported symbols from this shared object, as shown in the following example:

   ```
   $ dump -Tv shrobj.o

   shrobj.o:

                           ***Loader Section***

                           ***Loader Symbol Table Information***
   [Index]      Value      Scn     IMEX Sclass   Type           IMPid Name

   [0]     0x00000000    undef      IMP     DS EXTref   libc.a(shr.o) printf
   [1]     0x2000020c    .data      EXP     DS SECdef        [noIMid] addtot
   [2]     0x20000218    .data      EXP     DS SECdef        [noIMid] disptot
   ```

Two symbols, addtot and disptot, are exported from the created shared object shrobj.o as we expected (see “Displaying symbol definition with dump -Tv” on page 87 how to interpret the command output).

**Note:** If you do not have to precisely control which symbols will be exported, you can skip the step 2 and use the following command line in step 3:

```
$ cc -o shrobj.o source1.o source2.o -bexpall -bM:SRE -bnoentry
```

### 2.8.3 Interdependent shared objects

The process for creating interdependent shared objects is similar to the process of creating a self-contained shared object but requires the use of an import file. Suppose there are two shared objects, shr1.o and shr2.o, and each references symbols in the other. When creating the first shared object (shr1.o), the second shared object may not exist. This means that when the command to create the first shared object is executed, there will be unresolved symbols since, at this point, the second shared object does not exist. To avoid this paradox, create and use import files for these object files.

Consider the following files for use in this example scenario:

The file source1.c is as follows:

```
/* source1.c : First shared library source */
int function1(int a)
{
    int c;
    c = a + function2(a);
    return c;
}

int function3(int a)
{
    int c;
    c = a / 2;
    return c;
}
```

The file source2.c is as follows:

```
/* source2.c : Second shared library source */
int function2(int a)
{
    int c;
    c = function3(a + 5);
    return c;
}
```

In this example, each source file needs to be made into a separate, shared object. Note that the resulting shared objects are interdependent, since:

- Subroutine function1 in source1.c calls function2 in source2.c.
- Subroutine function2 in source2.c calls function3 in source1.c.

Create the export file libone.exp to define exporting symbols of the shared object shr1.o, which we are going to create from source1.c:

```
function1
function3
```

then insert the following directive at the beginning of libone.exp in order to use the file as an import file:

```
#!libone.a(shr1.o)
```

Create the export file libtwo.exp to define exporting symbols of the shared object shr2.o, which we are going to create from source2.c:

```
function2
```

then insert the following directive at the beginning of libtwo.exp in order to use the file as an import file:

```
#!libtwo.a(shr2.o)
```

To create two libraries, libone.a and libtwo.a, which contain only one archive member, shr1.o and shr2.o respectively, do the following:

```
cc -c source1.c source2.c
cc -o shr1.o source1.o -bE:libone.exp -bI:libtwo.exp -bM:SRE -bnoentry
ar rv libone.a shr1.o
cc -o shr2.o source2.o -bE:libtwo.exp -bI:libone.exp -bM:SRE -bnoentry
ar rv libtwo.a shr2.o
```

Note the use of the file libone.exp as an export file when creating the first shared object and as an import file when creating the second. If the file was not used when creating the second shared library, the creation of the second shared object would have failed with an error message complaining of unresolved symbols as follows:

```
cc -o shr2.o source2.o -bE:libtwo.exp -bM:SRE -bnoentry
ld: 0711-317 ERROR: Undefined symbol: .function3
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

Figure 2-8 depicts the function calling relationship between these two shared objects.

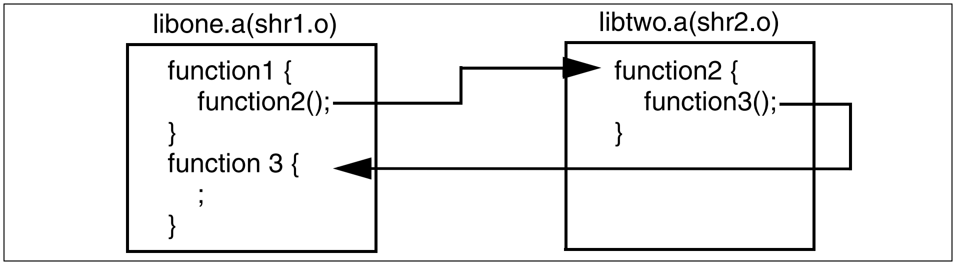


*Figure 2-8 Function calling relationship for an interdependent shared object*

A single import file can be used to list all symbols that are imported from different modules. In this case, the import file is just a concatenation of the individual export files for each of the shared objects. For example, Example 2-23 on page 99 could have been used for the combined import file when linking two shared objects in the previous example.

*Example 2-23 Combined import file*

```
#!libone.a(shr1.o)
function1
function3
* a comment line starts with an asterix character. blank lines are ignored.

#!libtwo.a(shr2.o)
function2
```

**Note:** It is recommended to create a single self-contained shared object instead of multiple interdependent shared objects whenever possible, in order to avoid the complexity and the possible application performance degrade.

### 2.8.4 Initialization and termination routines

Optional shared object initialization and termination routines can be specified when creating the shared object. You can use one or the other, or both. The routines may be useful for initializing dynamic data structures or reading configuration information. The initialization routines are called by the program startup code and are performed before the application main routine is started. Termination routines are called when the program makes a graceful exit. They will not be called if the program exits due to receipt of a signal.

The -binitfini linker option is used to specify the names of the routines along with a priority number. The priority is used to indicate the order that the routines should be called in when multiple shared objects with initialization or termination routines are used.

## 2.9 Shared libraries in a development environment

When an application is started, the system loader reads the loader section of the header of the executable file. It reads the dependency information for any shared objects the executable requires and attempts to load the code for those shared objects into the system shared library segment if they are not already loaded. Shared objects that are loaded into the system shared library segment have an attribute called the use count. Each time an application program that uses the shared object is started, the use count is incremented. When an application terminates, the use count for any shared objects it was using is decreased by one. When the use count for a shared object in the system shared library segment reaches zero, the shared object is not unloaded, but instead remains in memory. This reduces the overhead of starting any applications that use the shared object since they will not have to load the object into the system shared segment again.

To relinquish the unused shared objects or library archive members residing in the system memory, you need to either explicitly call the **slibclean** command (requires root authority on the system) or remove the shared objects or libraries using the **rm** command.

However, this default behavior of shared objects and libraries on AIX can be difficult to control, especially in the development environment where shared objects or libraries can be constantly changed and altered so that new versions should be used. Furthermore, multiple application developers might be working with their own version of a single shared library in a large development environment.

In order to effectively manage shared objects and libraries in the development environment, there are several methods, including the following:

- Use the -L and -l linker options and the LIBPATH environment variable.
- Use private shared objects and libraries.

**Note:** To solve typical link-time and run-time errors, see 6.3, “Diagnosing link-time errors” on page 239 and 6.4, “Diagnosing run-time errors” on page 244.

## 2.9.1 Production and development environments

If your application development directory structure does not match the directory structure used when your application is installed in a production environment, then, potentially, you need to adjust the arguments used with the linker to ensure that the resulting executables have the desired library search path.

When the application is installed in a production environment, for example, after being installed on the production system, the directory structure may be different. The method to use when compiling the executables will depend on the degree of freedom the customer is permitted when installing the application. For example, some applications expect that the executables and libraries must be installed in a specific directory, such as /opt/*productname*. Some applications allow the binaries and libraries to be installed in any directory structure.

- If the libraries for the application need to be installed in a specific directory, then you can either:
  - Create the shared libraries and then copy them to the same directory structure to be used when the product is installed in a production environment. In this case, use the -L option to specify the directory where the linker looks for shared objects and libraries. For example:

    ```
    cc -o ../bin/app1 main.c -llibone -L/product/lib
    ```

  - Create the shared libraries, but leave them in the development directory structure. When compiling the applications, use absolute pathnames to specify the shared libraries along with the -bnoipath linker option to prevent the pathname being included in the header section of the final executable. At the same time, use the -L option to specify the directory where the libraries will exist on a production system. For example:

    ```
    cc -o ../bin/app1 main.c -bnoipath ../lib/libone.a -L/product/lib
    ```

  > **Note:** In either case, all dependent shared objects and libraries should be installed in /product/lib (or /usr/lib, /lib) on the production system.

- If your application allows the executables and libraries to be installed in any directory structure, then the LIBPATH environment variable must be used accordingly on the production system to look for the dependent shared objects and libraries.

> **Note:** It is always recommended to avoid having any optional path components (see “Displaying header information with dump -H” on page 86) when creating shared objects and libraries and building executable files, in order to avoid unnecessary dependency to the file path names of the dependent modules.

## 2.9.2 Private shared objects

When used under normal circumstances, a shared object is loaded into the system global shared object segment. Subsequent executables that use the shared object benefit from the fact that it is already loaded.

In a development environment, particularly on a system with multiple developers, it may be preferable to use a private copy of a shared object. This may be useful when developing and testing new functionality in a shared object that is specific to a particular version of the application that a single developer is working on.

If the shared object or library has the access permissions modified as detailed below, then when the system loader starts an application that uses this shared object, the shared object text will be loaded into the process private segment rather than the system shared object segment. The shared object data will also be loaded into the process private segment instead of its normal location of the process shared object data segment. This means every application that uses private shared objects and libraries will have its own private copy of the shared object text and data while sacrificing the smaller size of a process private data segment.

When the program exits, all private shared objects required by the program will be released from the memory.

> **Note:** Private shared objects are not shown in the **genkld** command output, even if they are loaded into memory.

To use a private copy of the shared text and data, modify the access permissions as follows:

- Remove the read permission for other of the shared object as follows:

  ```
  chmod o-r foo.o
  ```

- Remove the read permission for other of the shared library as follows:

  ```
  chmod o-r libfoo.a
  ```

> **Note:** If the read permission for other of a library is removed, all the shared archive members in the library are considered *private*.

## 2.9.3 NFS consideration

Often, an application runs on NFS clients, whereas the actual executable file for the application and referenced shared objects or libraries stored on the NFS server. AIX tries to maintain the reliability of the running process by copying modules to the NFS client's paging space.

Although the likelihood of a shared module changing from a running process is minimal, this behavior avoids the potential of a process abend due to an NFS client and server losing synchronization in their respective understandings of the contents of a particular file. If a shared object referenced by the application is updated on the NFS server, and the inode does not change, then the kernel believes that the existing copy is current, and therefore will not pick up the new file. In this case, the **slibclean** command must be executed on the NFS client to pick up the new module on the NFS server. This is only effective if the module in question is no longer in use by any running process on the client.

## 2.9.4 Sufficient free disk space on the target directory and /tmp

The linker[19] maps the output file into its shared memory segment using shmat() (see "Mapping files with shmat()" on page 144 for detailed information about the shmat() services). The pages are flushed to disk when the linker exits. If the output file is on a remote file system, or on a file system that does not support mapping, a temporary file is created locally, which is then copied to the remote

[19] The linker, ld, is a 32-bit program.

file system. The temporary file will be created in the directory defined by the TMPDIR environment value, or in /tmp, if TMPDIR is not defined. Therefore, it is recommended to have enough free disk space on the target directory and /tmp in a development environment for this action.

# 3

# Understanding user process models

This chapter explains several memory models available on AIX and how these models affect the heap and shared memory usage in your applications by providing the following sections:

- Section 3.1, “User process models on AIX” on page 106
- Section 3.2, “The 32-bit user process model” on page 109
- Section 3.3, “The 64-bit user process model” on page 130
- Section 3.4, “Introduction to shared memory” on page 140
- Section 3.5, “Shared memory segments allocation order” on page 149
- Section 3.6, “Large page support” on page 157

For further information about the topics explained in this chapter, please refer to the following sections in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*:

- “Shared Libraries and Shared Memory”
- “System Memory Allocation Using the malloc Subsystem”

## 3.1 User process models on AIX

Starting with AIX Version 4.3, AIX supports two user process models, 32- and 64-bit, which we explain in the following sections:

- Section 3.2, “The 32-bit user process model” on page 109
- Section 3.3, “The 64-bit user process model” on page 130

Before explaining these models in detail, we provide a short introduction in this section. As shown in Figure 3-1, there are four combinations of kernel and hardware support of 32- and 64-bit support.

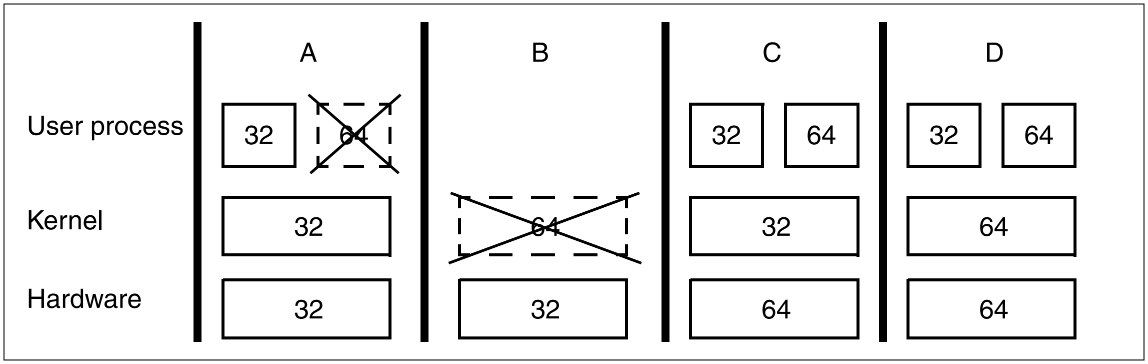


*Figure 3-1 Hardware, kernel, and user process relationships*

If the hardware bit mode is 32-bit, then 64-bit user processes cannot run on the 32-bit kernel (represented as A in Figure 3-1), also the 64-bit kernel cannot run on the 32-bit hardware (B).

If the hardware bit mode is 64-bit, then both 32- and 64-bit user processes can run regardless of the kernel type, though 64-bit user processes performance can be slightly degraded on the 32-bit kernel (C).

**Note:** AIX has been supporting the 64-bit user process model starting with Version 4.3. 64-bit application programs developed on AIX Version 4.3 are source-compatible with AIX 5L Version 5.1 and later; however, they are not binary-compatible. Therefore, those programs must be recompiled on AIX 5L Version 5.1 and later before execution.

### 3.1.1 How to determine hardware bit mode

To determine the hardware bit mode on your system, run either of the following commands, depending on the operating system versions:

- AIX 5L Version 5.2:

  ```
  getconf HARDWARE_BITMODE
  ```

- AIX 5L Version 5.1 and AIX Version 4.3[1]:

  ```
  bootinfo -y
  ```

  The command requires the root user authority.

These commands return either 32 (32-bit hardware) or 64 (64-bit hardware). The following example shows the command output executed on an AIX 5L Version 5.2 partition on the pSeries 690:

```
# oslevel; getconf HARDWARE_BITMODE
5.2.0.0
64
```

If you need to know the hardware bit mode where your application is running, you can use the sysconf() routine with the _SC_AIX_HARDWARE_BITMODE parameter, as shown in the following code fragment:

```
#include <unistd.h>
long bit_mode;
bit_mode = sysconf(_SC_AIX_HARDWARE_BITMODE);
```

If the application is running on the 32-bit hardware, it returns 32. If running on the 64-bit hardware, it returns 64.

> **Note:** In general, you do not have to know the hardware bit mode within your application source code, since the difference of hardware bit-modes does not affect user processes.

### 3.1.2 How to determine kernel bit mode

To determine the kernel bit mode on your system, run either of the following commands, depending on the operating system versions:

- AIX 5L Version 5.2:

  ```
  getconf KERNEL_BITMODE
  ```

[1] Prior to Version 4.3, AIX did not support 64-bit hardware models.

- AIX 5L Version 5.1[2]:

  ```
  bootinfo -K
  ```

  The command requires the root user authority.

These commands return either 32 (32-bit kernel) or 64 (64-bit kernel). The following example shows the command output executed on an AIX 5L Version 5.2 partition with the 64-bit kernel on the pSeries 690:

```
# oslevel; getconf KERNEL_BITMODE
5.2.0.0
64
```

If you need to know the kernel bit mode, where your application is running, you can use the sysconf() routine with the _SC_AIX_KERNEL_BITMODE parameter as shown in the following code fragment:

```
#include <unistd.h>
long bit_mode;
bit_mode = sysconf(_SC_AIX_KERNEL_BITMODE);
```

If the application is running on the 32-bit kernel, it returns 32. If running on the 64-bit kernel, it returns 64.

> **Note:** In general, except for developing kernel extensions, you do not have to know the kernel bit mode within your application source code, since the difference of kernel bit-modes does not affect user processes.

### 3.1.3 How to determine user process bit mode

To determine the user process bit mode, run the **`file`** command with the executable file of the corresponding user process.

If the executable file is 64-bit, then the command prints `64-bit XCOFF executable`, as shown in the following example:

```
$ cc -o helloworld32 helloworld.c
$ cc -q64 -o helloworld64 helloworld.c
$ file helloworld*
helloworld.c:   C program text
helloworld32:   executable (RISC System/6000) or object module not stripped
helloworld64:   64-bit XCOFF executable or object module not stripped
```

[2] Prior to Version 5.1, AIX supported 32-bit kernels (UP or MP) only.

**Note:** The -q64 option instructs the compiler to generate a 64-bit executable file (see 2.1.2, "Compiler support" on page 40). Although you can generate 64-bit executable files on 32-bit hardware systems, you cannot run them on 32-bit hardware systems.

## 3.2 The 32-bit user process model

This section provides a brief explanation about 32-bit user process models. Every 32-bit user process has its own *address space*, whose maximum size is 4 GB. An address space is prepared by the kernel as part of the process initialization tasks as a subset of virtual memory address on the system. A 32-bit address space is composed of up to sixteen 256 MB memory chunks called *segments*. Each segment has a specific purpose illustrated in several figures in the following sections. A segment is divided into smaller size memory chunks called *pages*. A page is an atomic unit, with which the virtual memory manager (VMM) in the kernel uses to satisfy many different memory requests.

On AIX, the default page size is 4 KB, unless the large page support[3] is used. If the default page size, 4 KB, is used, a segment contains 256 MB / 4 KB = 256 / 4 x 1024 = 64 K = 65,536 pages.

In the 32-bit user process model, there are several variations, called *large memory model* and *very large memory model*, depending on the required heap size for the process. We also explain these models in the following sections:

- Section 3.2.2, "Large memory model" on page 116
- Section 3.2.3, "Very large memory model" on page 117

### 3.2.1 Default memory model

Figure 3-3 on page 111 illustrates the segment usage of the default 32-bit memory model. This is the default memory model, unless you explicitly set a linker option (-b:maxdata) when generating executables or set an environment value (LDR_CNTRL) when executing 32-bit executables.

[3] See 3.6, "Large page support" on page 157 for more detail about the large page support.

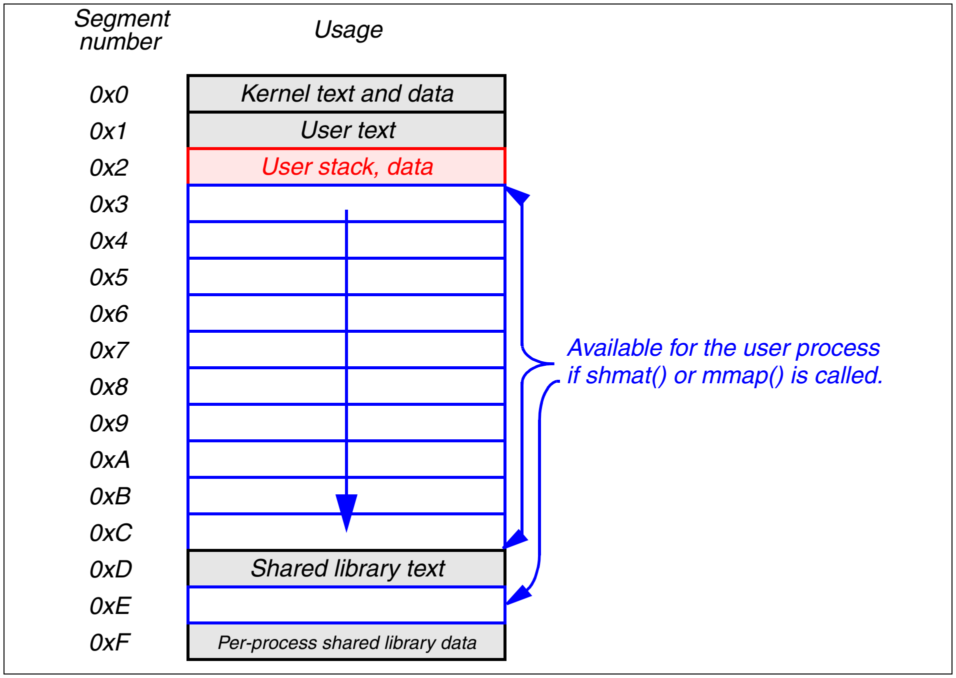


Figure 3-2 Default memory model (segment usage)[4]

The usage of each segment is briefly described in the following:

- The first segment, 0x0, is reserved by system for the kernel text and data. Therefore, access from the user process to the segment 0x0 is prohibited.[5]
- The 0x1 segment contains user process text. If another process shares the same executable file, there will be only one instance of the text pages for that executable file on the system.
- The 0x2 segment contains user data, heap, and stack. We provide detailed information about this segment in “Process private segment (0x2)” on page 111.
- The 0xD segment contains shared library text. This segment is shared by all 32-bit user processes. The virtual addresses of loaded shared text objects can be examined using the **genkld** command. For further information about the usage of **genkld**, see 2.7.2, “genkld” on page 88.
- The 0xF segment contains per-process shared library data.

[4] The access to the 0xE segment from a user process has been supported on AIX Version 4.2.1 and later.

[5] User processes can access to kernel data only through the system calls with specific ways.

- Segments 0x3 - 0xC and 0xE are attached to the address space if the process called shmat() or mmap() routines to allocate shared memory segments. These segments are allocated in order, starting from 0x3 towards 0xE excluding 0xD, unless DSA (Dynamic Segment Allocation) is used. Further detailed explanation is provided in 3.5.1, “Order in the 32-bit default memory model” on page 153 and 3.5.2, “Order in the 32-bit very large memory model with DSA” on page 154.

### Process private segment (0x2)

The process private segment, 0x2, is the most interesting segment for application developers on AIX. In this segment, many memory components that are mandatory for a user process are allocated (see Figure 3-3). In this figure, white rectangles represent un-allocated virtual address pages; if a process touches these address ranges, it receives the SIGSEGV[6] signal and dumps a core file. Highlighted rectangles represent allocated virtual address pages. You may notice that a 32-bit address space is quite vacant in most cases.

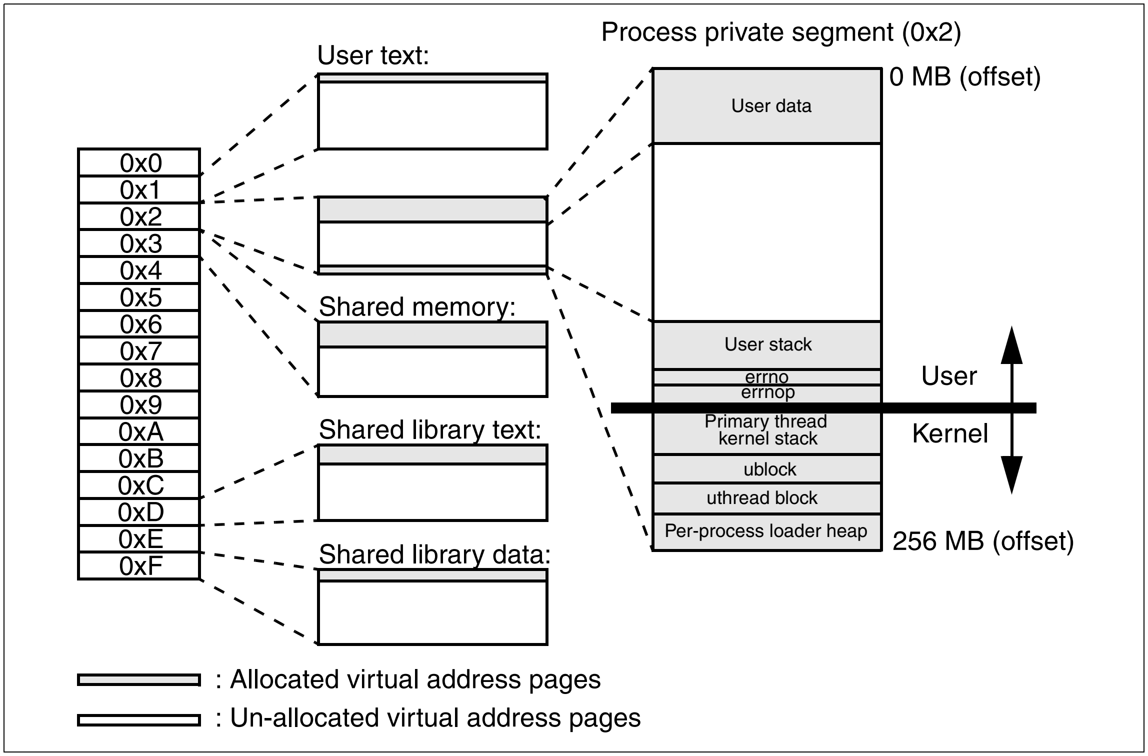


*Figure 3-3 Default memory model (detail)*

[6] Signal for segmentation violation

First of all, some pages within this segment are prepared by the kernel as a part of the process initialization tasks. The access to those pages from the user process is prohibited (represented as several highlighted rectangles under the *Kernel* in the right most rectangle). The information contained in this area are only accessible from the user process through system calls.

The rest of the pages within this segment can be accessed by the process, though there is a big *hole* between the process heap (represented as “User data” in the figure) and stack. If the process touches pages in this hole, it receives the SIGSEGV signal and dumps a core file.

User data contains three different memory components: initialized data, un-initialized data,[7] and heap. To demonstrate how they are mapped into virtual pages, we have prepared a simple program shown in Example 3-2 on page 113.

When compiled and executed, it prints addresses of several predefined symbols in the XCOFF format, as shown in Example 3-1. Those symbols, starting from the underscore character, are well explained in “XCOFF Object (a.out) File Format”, *AIX 5L Version 5.2 Files Reference*. We only explain _data and _edata here; the _data symbol defines the starting address of the initialized data in the executable file, while the _edata symbol defines the first location address after the initialized data in the executable file.

The **size** command tells you the size of the .text, .data, and .bss segments of the specified executable file.[8]

*Example 3-1 Addresses of several predefined symbols*

```
$ size -fx a.out
a.out: 428(.text) + 234(.data) + 8(.bss) + 2db(.loader) = 0x93f
$ a.out
_text =                 0x10000128
_etext =                0x10000550
_data =                 0x20000550
_edata =                0x20000784
argv[] =                0x2ff22a14
environ[] =             0x2ff22ff4
errnop =                0x2ff22ffc
errno =                 0x2ff22ff8
i_initialized =         0x200006f0
l_uninitialized =       0x20000788
str_buf =               0x2ff219b0
heap_mem =              0x20000798
```

[7] The un-initialized data is generally referred to as BSS (block started by symbol) in computer science terminology.

[8] The -x option instructs the command to display data in the hexadecimal format.

As for the addresses of _data and _edata, we can easily subtract 0x20000550 from 0x20000784 using the **bc** command, as shown in the following example:

```
$ bc
ibase=F
obase=F
20000784 - 20000550
234
^D
```

To quit the command, type Control-D. The ibase=F and obase=F keywords instruct the command to treat both input and output as hexadecimal numbers. The answer is 0x234, which is equivalent to the .data segment size in the executable file reported by the **size** command.

> **Note:** You need to use upper-case characters (A - F) to represent hexadecimal 0xA - 0xF with the **bc** command.

After the initialized data, un-initialized data (BSS) is loaded into the memory as shown by the address of l_uninitilized. Then heap starts as shown by the address of heap_mem. It grows toward the last address of this segment (0x2FFFFFFF).

On the other hand, the stack grows from a relatively higher address toward the first address of this segment (0x20000000). The user process stack contains several predefined symbols, such as argv, environ, errno, and so on. The user process stack grows, if the process calls functions or allocates auto storage class symbols.

> **Note:** Thread stacks for Pthreads, except the initial thread within a multi-threaded processes, are allocated in the process heap. See 8.3.4, “Thread stack” on page 292 for further information about the thread stack for multi-threaded processes.

*Example 3-2 underscore_symbols_32.c*[9]

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

extern int errnop;
extern int errno;
extern _text;
extern _etext;
extern _data;
extern _edata;
```

[9] Extern symbols start with underscore are run-time symbols prepared by the system loader.

```
extern char *environ[];

int     i_initialized = 1;      /* initialized global variable. */
long    l_uninitialized;        /* uninitialized global variable. */

int
main(int argc, char *argv[])
{
    char    str_buf[BUFSIZ];    /* auto storage class variable. */
    char    *heap_mem;          /* heap memory. */

    if ((heap_mem = malloc(BUFSIZ)) == (void *)NULL) {
        sprintf(str_buf, "malloc() failed with errno = %d", errno);
        perror(str_buf);
    }

    printf("_text =\t\t\t0x%08p\n", &_text);
    printf("_etext =\t\t0x%08p\n", &_etext);
    printf("_data =\t\t\t0x%08p\n", &_data);
    printf("_edata =\t\t0x%08p\n", &_edata);

    printf("argv[] =\t\t0x%08p\n", argv);
    printf("environ[] =\t\t0x%08p\n", environ);
    printf("errnop =\t\t0x%08p\n", &errnop);
    printf("errno =\t\t\t0x%08p\n", &errno);

    printf("i_initialized = \t0x%08p\n", &i_initialized);
    printf("l_uninitialized = \t0x%08p\n", &l_uninitialized);
    printf("str_buf = \t\t0x%08p\n", str_buf);
    printf("heap_mem = \t\t0x%08p\n", heap_mem);

    exit(0);
}
```

Because the process heap and stack share the same segment, the heap size must be smaller than 256 MB in the default memory model. This is the reason why the default definition of user soft data limit is 131,072 KB = 128 MB (see Example 3-6 on page 125).

We provide detailed information about how heap and stack grows in this segment in 3.2.6, “Resource limits in 32-bit model” on page 125.

## Environment variables

Environment variables are accessible from a process using either of the following, regardless of user process models, including 64-bit:

- The environ extern symbol (see Example 3-2 on page 113).
- The third parameter of the main() routine. This parameter is commonly represented as envp, as shown in the following example:

  ```
  int main(int argc, char *argv[], char *envp[]);
  ```
- The getenv() sub-routine call.

On AIX, environment variables are copied into the process stack by the system loader as a part of the process initialization tasks. The maximum size of environment variables per process are defined by a system-wide configuration value, *ncargs*, as shown in the following example:[10]

```
# lsattr -El sys0 -a ncargs
ncargs 6 ARG/ENV list size in 4K byte blocks True
```

The default value 6 means 6 pages x 4 [KB/page] = 24 KB. If a process is given environment variables that require more than ncargs x 4 KB, those environment values that exceeded the value will not be copied into virtual pages and cannot be accessed by the process.

If you need to increase ncargs, do either of the following as the root user:

- Use the **chattr** command:

  ```
  # chattr -El sys0 -a ncargs=N
  ```

  where *N* is the number of pages that can be used for environment variables per-process basis.
- Use SMIT

  Select **System Environments** → **Change / Show Characteristics of Operating System**, then specify an appropriate number in the “ARG/ENV list size in 4K byte block” field, and then press Enter.

In both ways, the minimum value is 6, and the maximum value is 128 (512 KB). This change takes affect immediately and is preserved over boot.

> **Note:** The default value ncargs = 6 is sufficient for most processes. Change it only when your applications absolutely require a larger size of environment variables. If there are many processes that require a large size of environment variables, those processes may consume a significant amount of memory.

[10] The ncargs system parameter has been supported on AIX since Version 5L Version 5.1.

## 3.2.2 Large memory model

Although many 32-bit applications do not require a heap size more than 256 MB, there has been a huge demand to run 32-bit user processes with a larger size of initialized data, process heap, or process stack. To address this demand, AIX has been supporting another memory model called *large memory model*, as a variation of the 32-bit user process model.

The concept of the large memory model is simple. By sacrificing segments available for shared memory segments, the user process running in the large memory model can place its heap on those segments (see Figure 3-4).

In Figure 3-4, four segments (0x3 - 0x6) are reserved for heap, so seven segments (0x7 - 0xC and 0xE) are available for shared memory segments.

Because the process heap and initialized and un-initialized data are moved from the 0x2 segment to 0x3, the user stack can enjoy more room in the 0x2 segment in this model.

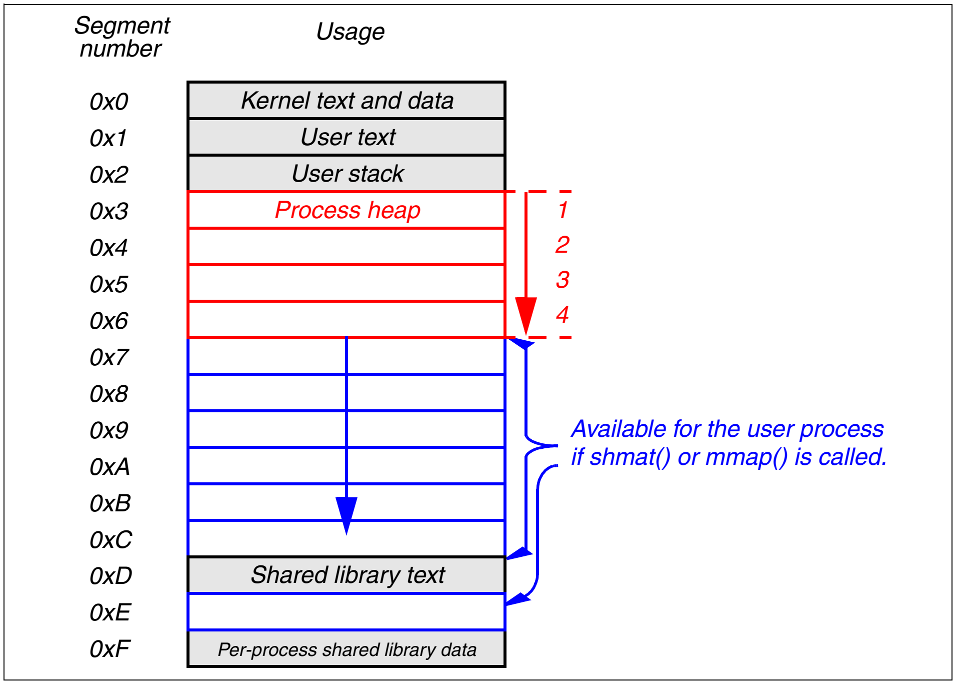


*Figure 3-4 Large memory model (segment usage)*

To apply the large memory model to your application programs, see 3.2.4, “Using the large and very large memory model” on page 121.

**Note:** In the large memory model, the soft data limit of a process has a slightly different effect (see 3.2.6, “Resource limits in 32-bit model” on page 125).

## 3.2.3 Very large memory model

In addition to the large memory model, AIX 5L Version 5.2 supports another model, called *very large memory model*, as a variation of the 32-bit user process model. AIX 5L Version 5.1 also supports some of the features provided by the very large memory model; it does support up to 2 GB heap memory, but not up to 3.25 GB.

Although the concept of the very large memory model remains the same, there are three different allocation mechanisms, depending on the maxdata value setting explained in 3.2.6, “Resource limits in 32-bit model” on page 125:

- “Heap size less than 2.5 GB” on page 117
- “Heap size greater than 2.5 GB” on page 118
- “Heap size less than 256 MB” on page 119

To apply the very large memory model to your application programs, see 3.2.4, “Using the large and very large memory model” on page 121.

**Note:** In the very large memory model, the soft data limit of a process has a slightly different effect (see 3.2.6, “Resource limits in 32-bit model” on page 125).

### Heap size less than 2.5 GB

The big difference between the very large memory model and the large memory model is that segments can be dynamically allocated. In the large memory model, the allocation of segments are rigid, and there is no way to alter this segment allocation while executing the application program. The very large memory model relaxes this limitation by providing a new function, called *dynamic segment allocation* (DSA).

With DSA, segments are dynamically allocated to either the process heap or shared memory segments in the address space. Once a segment is allocated to either the process heap or shared memory segments, it cannot be returned to the free pool of segments, even if all virtual pages in the segment are relinquished.

In Figure 3-5 on page 118, we are assuming that a process is running with maxdata=0xA0000000 and the DSA option specified, which means the process can acquire heap memory up to 10 segments (2.5 GB).

Another big difference between the very large and large memory models is the direction of how shared memory segments are allocated. In this figure, shared memory segments will be allocated in the segments, starting from 0xE, 0xC, 0xB, and on towards 0x3, if the target segments are not already allocated to the process heap. To understand how these segments are allocated, see 3.5.2, “Order in the 32-bit very large memory model with DSA” on page 154.

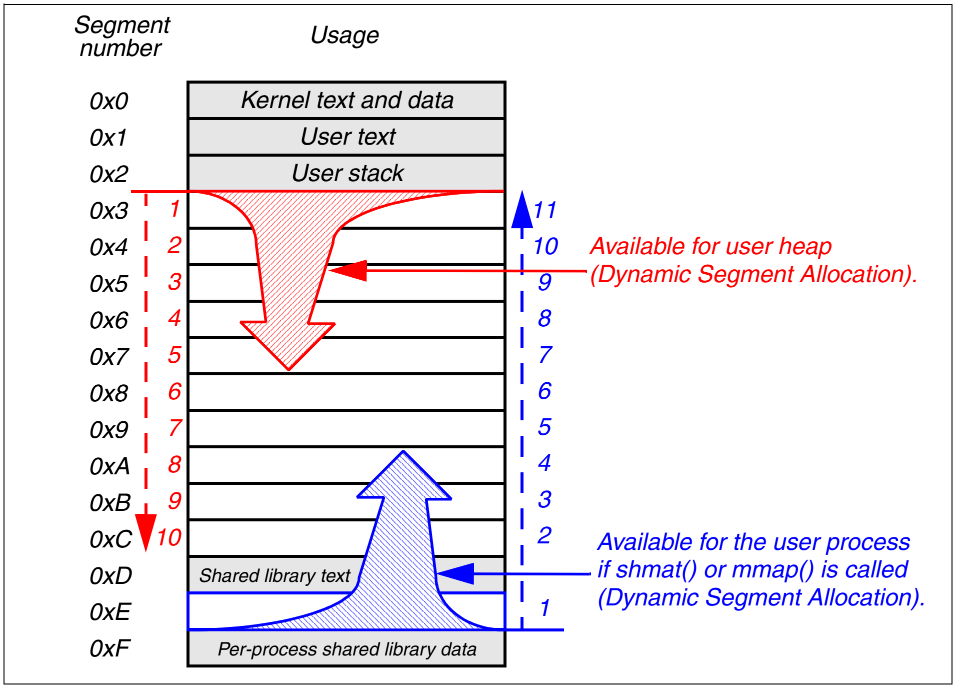


*Figure 3-5 Very large memory model (0 < maxdata < 0xB0000000)*

Because of the process heap, initialized and un-initialized data are moved from the 0x2 segment to 0x3, and the user stack can enjoy more room in the 0x2 segment in this model.

**Note:** AIX 5L Version 5.1 supports this model only when maxdata is less than or equal to 0x80000000.

## Heap size greater than 2.5 GB

If a process is running with maxdata=0xB0000000 and greater with DSA, then the address space has a significant change; there are no shared text segments available, as shown in Figure 3-6 on page 119. Therefore, all the shared text and data that are referenced by the process will be loaded into the 0x2 segment, as private shared objects, even if the other processes has already loaded shared objects into the system memory. The process performance might be affected by this change.

However, by sacrificing segments 0xD - 0xF, the process can acquire heap memory up to 13 segments (3.25 GB). Also, shared memory segments will be allocated in the segments, starting from 0xF, 0xE, 0xD, and on towards 0x3, if the target segments are not already allocated to the process heap. To understand how these segments are allocated, see 3.5.2, “Order in the 32-bit very large memory model with DSA” on page 154.

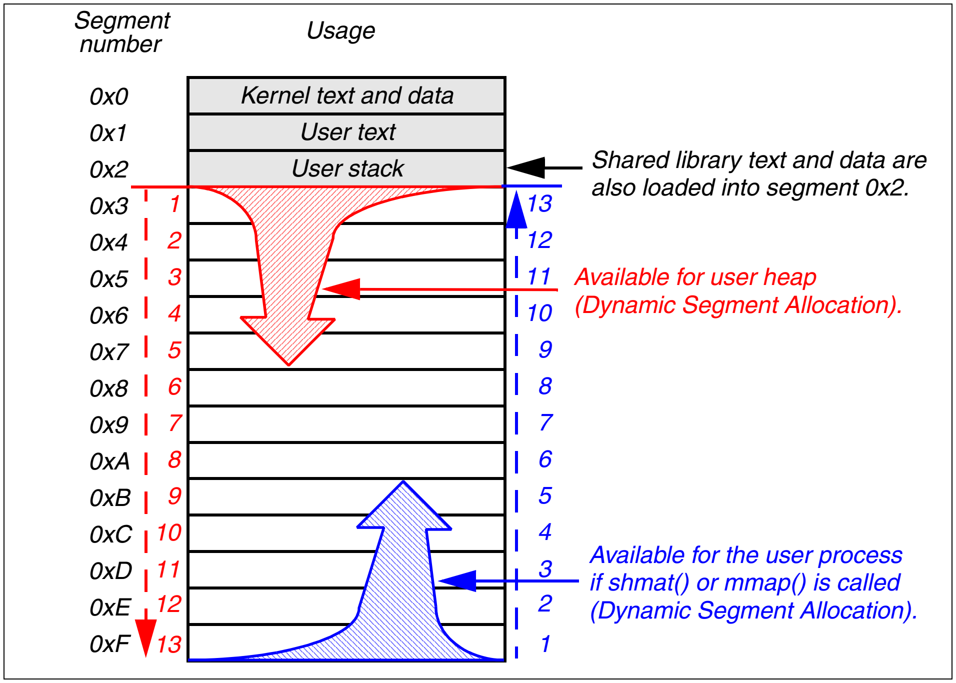

Segment number
Usage
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xA
0xB
0xC
0xD
0xE
0xF
Kernel text and data
User text
User stack
1
2
3
4
5
6
7
8
9
10
11
12
13
13
12
11
10
9
8
7
6
5
4
3
2
1
Shared library text and data are also loaded into segment 0x2.
Available for user heap (Dynamic Segment Allocation).
Available for the user process if shmat() or mmap() is called (Dynamic Segment Allocation).

*Figure 3-6 Very large memory model (0xB0000000 =< maxdata < 0xD0000000)*

## Heap size less than 256 MB

If a process is running with maxdata=0 with DSA, then all the following memory components are packed into the 0x2 segment:

- Initialized data
- Uninitialized data
- Process heap
- Process stack
- Shared library text
- Shared library data

Apparently, this does not provide much room for the process heap; in fact, it is smaller than the default memory model. However, it ensures that all 13 segments (3.25 GB) are available for shared memory segments in the address space. Some application programs that manage their own memory requests on shared memory segments can exploit this memory model. To understand how these segments are allocated, see 3.5.2, “Order in the 32-bit very large memory model with DSA” on page 154.

**Note:** If you decide to apply this memory model to your application, it is your responsibility to ensure that the process heap and stack usage are well managed. You must be aware that the 0x2 segment can be easily corrupted in this model.

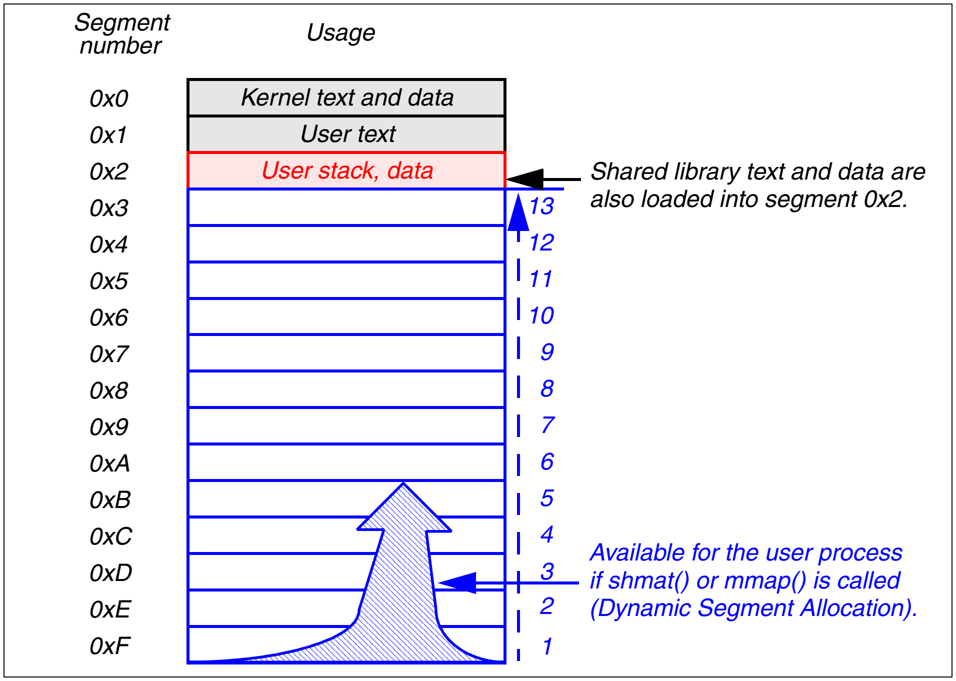

Segment number
Usage
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xA
0xB
0xC
0xD
0xE
0xF
Kernel text and data
User text
User stack, data
13
12
11
10
9
8
7
6
5
4
3
2
1
Shared library text and data are also loaded into segment 0x2.
Available for the user process if shmat() or mmap() is called (Dynamic Segment Allocation).

*Figure 3-7 Very large memory model (maxdata = 0)*

**Note:**

- On AIX 5L Version 5.1, shared library text and data will be loaded into segments 0xD and 0xF in this model (maxdata=0 with DSA).
- This model does not support large pages (see 3.6, “Large page support” on page 157).

### 3.2.4 Using the large and very large memory model

To use the large memory model, any one of the following methods are supported:

- Specify the -bmaxdata:0xN0000000 linker option when linking your source codes (where N is hexadecimal from 0 to 8).
- After compiling the source code, binary-edit the XCOFF header file of the generated executable file using the **/usr/ccs/bin/ldedit** command. For example:

  ```
  $ /usr/ccs/bin/ldedit -bmaxdata:0x60000000 a.out
  ```

- Specify the LDR_CNTRL environment value when running the executable. For example:

  ```
  $ LDR_CNTRL=MAXDATA=0x80000000 a.out
  ```

To use the very large memory model, the following methods are supported:

- Specify the -bmaxdata:0xN00000000/dsa compiler option when compiling your source codes (where N is hexadecimal from 0 to D).
- After compiling the source code, binary-edit the XCOFF header file of the generated executable file using the **/usr/ccs/bin/ldedit** command. For example:

  ```
  $ /usr/ccs/bin/ldedit -bmaxdata:0x80000000/dsa a.out
  ```

- Specify the LDR_CNTRL environment value when running the executable. For example:

  ```
  $ LDR_CNTRL=MAXDATA=0xC0000000@DSA a.out
  ```

**Note:** When using the LDR_CNTRL environment variable, the keyword *DSA* must be upper case character, while *dsa* must be lower case characters in the other two methods.

To demonstrate the usage, we have prepared the short example program shown in Example 3-3. When executed, it takes an integer as a command line parameter, which will be used as an index to allocate memory with a size of multiplies of 256 MB from the process heap.

*Example 3-3 grabheap_32.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

#define ONE_SEG (256 * 1024 * 1024)
```

```
int main(int argc, char *argv[])
{
    char    *p;
    char    buf[BUFSIZ];
    int     rc;
    size_t  sz, sz_7;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s digit-number\n", argv[0]);
        exit(1);
    }
    sz = atoi(argv[1]);

    if (sz > 7) {
        sz_7 = sz - 7;

        if ((p = malloc(7 * ONE_SEG)) == (void *)NULL) {
            sprintf(buf, "1st malloc() with 7 x 256 MB failed with errno = %d"
                    , errno);
            perror(buf);
        } else {
            printf("1: starting address of 7 x 256 MB memory heap is 0x%08p\n"
                , p);
            printf("1: ending address of 7 x 256 MB memory heap is 0x%08p\n"
                , p + (7 * ONE_SEG));
        }
        if ((p = malloc(sz_7 * ONE_SEG)) == (void *)NULL) {
            sprintf(buf, "2nd malloc() with %d x 256 MB failed with errno = %d"
                    , sz_7, errno);
            perror(buf);
        } else {
            printf("2: starting address of %d x 256 MB memory heap is 0x%08p\n"
                , sz_7, p);
            printf("2: ending address of %d x 256 MB memory heap is 0x%08p\n"
                , sz_7, p + (sz_7 * ONE_SEG) - 1);
        }
    } else {
        if ((p = malloc(sz * ONE_SEG)) == (void *)NULL) {
            sprintf(buf, "malloc() with %d x 256 MB failed with errno = %d"
                    , sz, errno);
            perror(buf);
        } else {
            printf("starting address of %d x 256 MB memory heap is 0x%08p\n"
                , sz, p);
            printf("ending address of %d x 256 MB memory heap is 0x%08p\n"
                , sz, p + (sz * ONE_SEG) - 1);
        }
    }
    exit(0);
}
```

The following example shows the output when we run Example 3-3 on page 121 using the very large memory model on AIX 5L Version 5.1:

```
$ oslevel -r
5100-03
$ cc grabheap_32.c
$ LDR_CNTRL=MAXDATA=0xD0000000@DSA ./a.out 7
starting address of 7 x 256 MB memory heap is 0x30000968
ending address of 7 x 256 MB memory heap is 0xa0000967
$ LDR_CNTRL=MAXDATA=0xD0000000@DSA ./a.out 8
1: starting address of 7 x 256 MB memory heap is 0x30000968
1: ending address of 7 x 256 MB memory heap is 0xa0000967
2nd malloc() with 1 x 256 MB failed with errno = 12: Not enough space
```

The second call of malloc() failed with errno = 12 (ENOMEM), because it requested another 256 MB in addition to the already acquired 7 x 256 MB (1.75 GB) heap memory. Although AIX 5L Version 5.1 supports the heap size up to 2 GB, there is always a very small amount of overhead memory (initialized and un-initialized data) required in the beginning of segment 0x3 in the large and very large memory model regardless of operating system versions.

The following example shows the output when we run Example 3-3 on page 121 using the very large memory model on AIX 5L Version 5.2:

```
$ oslevel
5.2.0.0
$ cc grabheap_32.c
$ LDR_CNTRL=MAXDATA=0xD0000000@DSA ./a.out 12
1: starting address of 7 x 256 MB memory heap is 0x30000988
1: ending address of 7 x 256 MB memory heap is 0xa0000987
2: starting address of 5 x 256 MB memory heap is 0a0000998
2: ending address of 5 x 256 MB memory heap is 0xf0000997
```

The first call of malloc() acquired memory from the segment 0x3 to 0xA and the second call acquired memory from the segment 0xA to 0xF; the outcome is clear. This process successfully acquired 12 x 256 MB = 3 GB memory in total. In fact, the program could have requested more memory, roughly 256 MB; therefore, it could have acquired roughly 3.25 GB memory in total.

If your application needs a heap size larger than 2 GB in the very large memory model, then it must call system heap allocation routines, such as malloc() and calloc(), more than once, like we did in Example 3-3 on page 121. The reason for this limitation is that the sbrk() system call, which is internally called from system heap allocation routines, takes a signed integer value as a parameter. Therefore, it cannot extend the break value more than 2 GB within a single call.

For further information about the use of the system heap, see Chapter 4, “Managing the memory heap” on page 165.

## 3.2.5 Checking large memory model executables

After compiling your source code with the -bmaxdata:0xN0000000 option, use the **file** command with the modified /etc/magic file to check if the o_maxdata field in the XCOFF header of executable is correctly set. On AIX, instead of the /etc/magic file, the **file** command consults with a locale message file (depending on the current locale setting) by default. Therefore, to have the command refer to the modified /etc/magic file, do the following:

1. Log in to the system as the root user.
2. Create a backup of the original /etc/magic:

   ```
   # cp -p /etc/magic /etc/magic.orig
   ```

3. Edit the file and add highlighted lines in Example 3-4 using a text editor, then save the file and exit.
4. Run the **file** command against the executable file with the -m /etc/magic option.

*Example 3-4 Excerpt of modified /etc/magic*

```
0   short       0x01df      executable (RISC System/6000) or object module
>12 long        >0      not stripped
>18 byte        0x14        LP_TEXT
>18 byte        0x18        LP_DATA
>19 byte        >0x3F       /DSA
>76 long        0x00000000  (maxdata=0)
>76 long        0x10000000  (maxdata=1)
>76 long        0x20000000  (maxdata=2)
>76 long        0x30000000  (maxdata=3)
>76 long        0x40000000  (maxdata=4)
>76 long        0x50000000  (maxdata=5)
>76 long        0x60000000  (maxdata=6)
>76 long        0x70000000  (maxdata=7)
>76 long        0x80000000  (maxdata=8)
>76 long        0x90000000  (maxdata=9)
>76 long        0xA0000000  (maxdata=A)
>76 long        0xB0000000  (maxdata=B)
>76 long        0xC0000000  (maxdata=C)
>76 long        0xD0000000  (maxdata=D)
0   short       0x01f7      64-bit XCOFF executable or object module
>18 byte        0x14        LP_TEXT
>18 byte        0x18        LP_DATA
>19 byte        >0x3F       /DSA
```

```
>20 long        >0      not stripped
```

Example 3-5 shows the sample output from the executable file compiled with the -bmaxdata:0x80000000/dsa option.

*Example 3-5 Checking large memory executable with modified /etc/magic*

```
$ cc -bmaxdata:0x80000000 helloworld.c
$ file a.out
a.out:          executable (RISC System/6000) or object module not stripped
$ file -m /etc/magic a.out
a.out:          executable (RISC System/6000) or object module not stripped /DSA (maxdata=8)
```

For further information about the /etc/magic file and XCOFF format, please refer to the *AIX 5L Version 5.2 Files Reference*.

## 3.2.6 Resource limits in 32-bit model

The resource limits are used to regulate several resources consumed by a process on UNIX operating systems. Although the resource limits are set on a per-user basis, they are applied on a per-process basis. Therefore, if a user is executing hundreds of processes, the user may consume huge amounts of resources, even if the resource setting values for the user are relatively small numbers.

To display the current user's resource limits, use the **ulimit** command (see Example 3-6).

*Example 3-6 Default resource limits setting*

```
$ ulimit -Ha
time(seconds)        unlimited
file(blocks)         2097151
data(kbytes)         unlimited
stack(kbytes)        unlimited
memory(kbytes)       unlimited
coredump(blocks)     unlimited
nofiles(descriptors) unlimited
$ ulimit -Sa
time(seconds)        unlimited
file(blocks)         2097151
data(kbytes)         131072
stack(kbytes)        32768
memory(kbytes)       32768
coredump(blocks)     2097151
nofiles(descriptors) 2000
```

The -H option instructs the command to display *hard* resource limits, while the -S option instructs the command to display *soft* resource limits.[11] The hard resource limit values are set by the root user using the **chuser** command for each user. The soft resource limit values can be relaxed by the individual user using the **ulimit** command, as long as the values are smaller than or equal to the hard resource limit values.

Although there are seven types of resource limits displayed in Example 3-6 on page 125, we only discuss the data and stack resource limits, which regulate process heap and stack usage. For further information about resource limits, please refer to the command reference of **ulimit**, found in the *AIX 5L Version 5.2 Reference Documentation: Commands Reference*.

### Resource limits in the default memory model

In the default memory model, soft data and stack resource limits are strictly enforced. During the process initialization tasks, the kernel sets soft data and stack resource limits in the process address space (see Figure 3-8 on page 126). While the process is running, the heap and stack can grow up to the soft data limits from opposite directions, as represented by arrows in the figure.

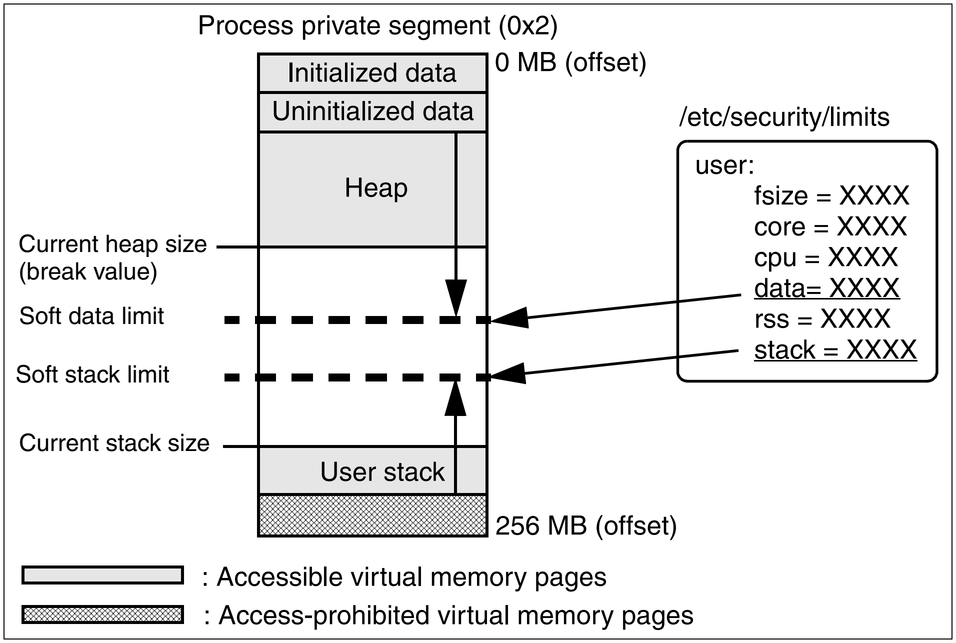


*Figure 3-8 Data and stack resource limits (default 32-bit process model)*

[11] Resource limits are defined in the /etc/security/limits file on a per-user basis.

If either the data or stack soft limit is set to unlimited before the program execution, or if the setrlimit() routine is called to set these limit values to unlimited while the process is running, it is your responsibility to prevent the process stack from being overwritten by the over-grown process heap.

## Resource limits in the very large and large memory model

When you execute a program that uses the large memory model, the operating system attempts to modify the soft limit on the data size to match the maxdata value. If the maxdata value is *greater* than the current hard limit on the data size, either the program will not execute, if the environment variable XPB_SUS_ENV has the value set to ON, or the soft limit will be set to the current hard limit. If the maxdata value is *smaller* than the size of the program's static data, the program will not execute.

To demonstrate this behavior, we have prepared a simple program, shown in Example 3-10 on page 128. When compiled and executed, it prints the soft and hard resource limit values by calling the getrlimit() sub-routine.

When we have executed this program in the default memory model, it prints the values shown in Example 3-7 on page 127.

*Example 3-7 Verifying resource limits in default memory model*

```
$ a.out
Resource name                   Soft                    Hard
RLIMIT_CORE               1073741312              2147483647
RLIMIT_CPU                2147483647              2147483647
RLIMIT_DATA                134217728              2147483647
RLIMIT_FSIZE              1073741312              1073741312
RLIMIT_NOFILE                   2000              2147483647
RLIMIT_STACK                33554432              2147483647
RLIMIT_RSS                  33554432              2147483647
```

You will find that this output resembles Example 3-6 on page 125.

On AIX, all hard resource limit values are set to *unlimited* by default, except for the file resource limit. The value unlimited is defined as 0x7FFFFFFF in the /usr/include/sys/resource.h include file when the program is running in the 32-bit user process model (see Example 3-8).

*Example 3-8 RLIM_INFINITY definition*

```
#if defined(__64BIT__) && !defined(__64BIT_KERNEL)
#define RLIM_INFINITY   0x7fffffffffffffffL
#else
#define RLIM_INFINITY   0x7FFFFFFF
```

```
#endif /* __64BIT__ */
```

When we have executed this program in the large memory model, it prints the values shown in Example 3-10 on page 128. As shown in this example, the soft data limit value is set to RLIM_INFINITY without using either the **ulimit** command or the setrlimit() sub-routine.

*Example 3-9 Verifying resource limits in large memory model without DSA*

```
$ LDR_CNTRL=MAXDATA=0x80000000 a.out
Resource name                   Soft                    Hard
RLIMIT_CORE               1073741312              2147483647
RLIMIT_CPU                2147483647              2147483647
RLIMIT_DATA               2147483645              2147483647
RLIMIT_FSIZE              1073741312              1073741312
RLIMIT_NOFILE                   2000              2147483647
RLIMIT_STACK                33554432              2147483647
RLIMIT_RSS                  33554432              2147483647
```

In fact, when we executed the program shown in Example 3-3 on page 121 using the large and very large memory models, we did not increase the soft data limit value.

Example 3-10 is a program that calls the getrlimit() routine to print current soft and hard resource limit values.

*Example 3-10 printlimits.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/resource.h>

#define pr_limit(XXX)   print_limit(#XXX, XXX)

void
print_limit(const char *name, int resource)
{
    int rc;
    char    buf[BUFSIZ];
    struct  rlimit  rlimit;

    if ((rc = getrlimit(resource, &rlimit)) < 0) {
        sprintf(buf, "getrlimit() failed with errno = %d, at %d in %s"
            , errno, __LINE__, __FILE__);
        perror(buf);
    } else {
        printf("%-14s\t%20ld\t%20ld\n"
            , name, rlimit.rlim_cur, rlimit.rlim_max);
    }
}

int
main(int argc, char *argv[])
{
    printf("%-14s\t%20s\t%20s\n", "Resource name", "Soft", "Hard");
    pr_limit(RLIMIT_CORE);
    pr_limit(RLIMIT_CPU);
    pr_limit(RLIMIT_DATA);
    pr_limit(RLIMIT_FSIZE);
    pr_limit(RLIMIT_NOFILE);
    pr_limit(RLIMIT_STACK);
    pr_limit(RLIMIT_RSS);
}
```

### 3.2.7 Large file support in a 32-bit model

In the computer industry, the term *large file* is generally considered to refer to a file larger than 2 GB. In the 32-bit user process model, the data type off_t is defined as signed long type. Therefore, by default, 32-bit processes can handle files up to 2 GB -1 byte, which is addressable by the off_t type. If a 32-bit process attempts to seek a file pointer more than 2 GB, then the system call fails with errno EOVERFLOW. This limitation exists on all UNIX operating systems that conform to the POSIX standards.

Beginning with AIX Version 4.2.1, there are two methods to work around the addressability of large files in the 32-bit user process model:

- If the -D_LARGE_FILE option is specified when compiling the program, the off_t type is redefined as long long (64-bit signed integer). Also, most system calls and library routines are redefined to support large files.
- In some circumstances, your existing 32-bit user programs might be corrupted by defining the _LARGE_FILE macro, especially if off_t is interchangeably used with the int or long type within your source codes. In this case, you can modify your source codes to explicitly call 64-bit versions of file I/O related system calls and library routines, such as fopen64(), lseek64(), and so on. Also, variables defined as the off_t type must be carefully converted to off64_t.

Using one of the above methods, 32-bit processes can handle large files as long as the following conditions are met:

- The file hard and soft limits have been relaxed. The default value of file limit is 2,097,151 disk blocks = 1 GB (see Example 3-6 on page 125).
- JFS2 or large file enabled JFS is used.

For further information about large file support of the 32-bit user process, please refer to the “Writing Programs That Access Large Files” section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 3.3 The 64-bit user process model

This section provides a brief explanation about the 64-bit user process model. The 64-bit user process model shares the same concept of segments and pages with the 32-bit user process model. The difference is the number of available segments in the address space. In the 64-bit user process model, an address space is composed of $2^{32}$ segments, while it is composed of $2^4$ = 16 segments in the 32-bit user process model.

Therefore, the 64-bit user process can address up to 1 EB (exabytes), which is easily returned by the following simple calculation:

$$2^{32} \text{ segments x } 256 \text{ [MB/segment]} = 2^{32} \text{ x } 2^{8} \text{ x } 2^{20} \text{ bytes} = 2^{32+8+20} = 2^{60} = 1 \text{ EB}$$

To address this tremendous huge space, the pointer type is defined as 64-bit in the 64-bit user process model. The program shown in Example 3-11 prints 8 bytes in the 64-bit user process model, while it prints 4 bytes in the 32-bit user process model:

```
$ cc -q64 sizeofpointer.c
$ a.out
size of pointer data type is 8 byte.
$ cc sizeofpointer.c
$ a.out
size of pointer data type is 4 byte.
```

*Example 3-11 sizeofpointer.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    char *p;

    printf("size of pointer data type is %d byte.\n", sizeof(p));
    exit(0);
}
```

Figure 3-9 on page 131 illustrates the segment usage in the 64-bit user process model. The address space is divided into the areas explained in the following sections:

- Section 3.3.1, “The first 16 segments (0 - 4 GB)” on page 132 (see Figure 3-10 on page 132)
- Section 3.3.2, “Application text, data, and heap (4 GB - 448 PB)” on page 133
- Section 3.3.3, “Default shared memory segments (448 - 512 PB)” on page 135
- Section 3.3.4, “Privately loaded objects (512 - 576 PB)” on page 135
- Section 3.3.5, “Shared text and data (576 - 640 PB)” on page 135
- Section 3.3.6, “System reserved (640 - 960 PB)” on page 136
- Section 3.3.7, “User process stack (960 PB - 1 EB)” on page 136

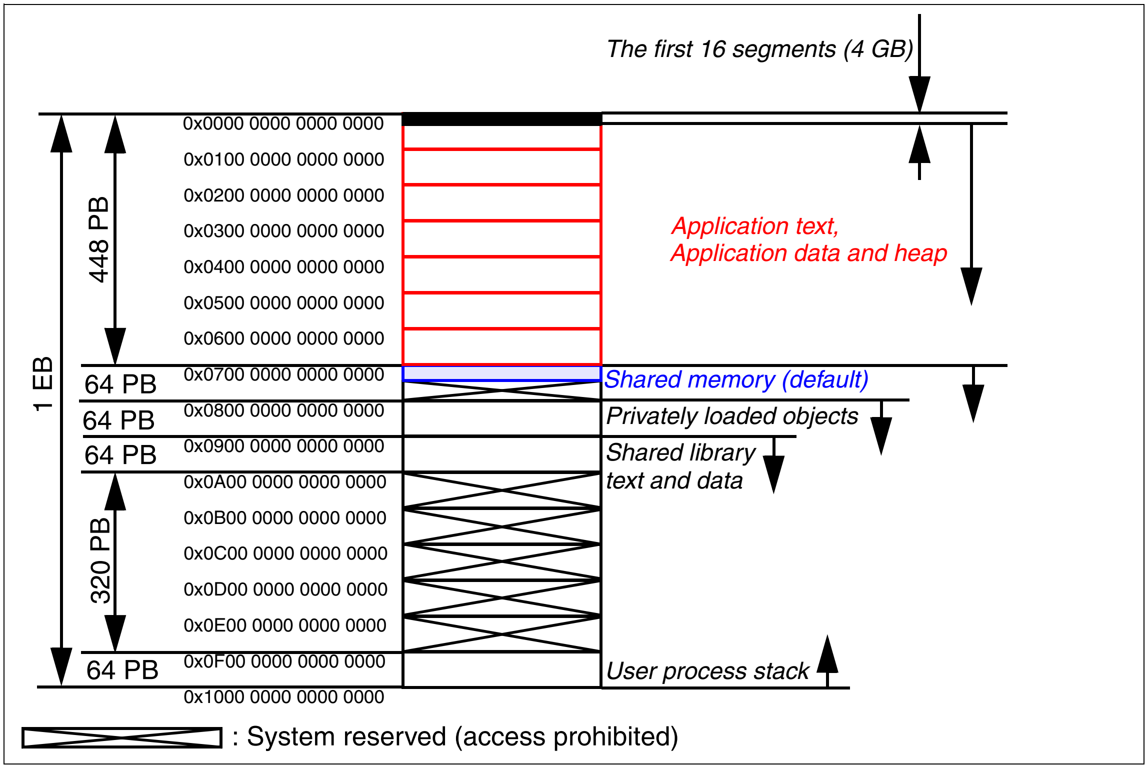


*Figure 3-9 The 64-bit memory model (1EB)*

### 3.3.1 The first 16 segments (0 - 4 GB)

These segments are exempt from general use in order to keep the compatibility with the 32-bit user process model. Therefore, access to the segments 0x0, 0x1, 0xD, and 0xE are prohibited; these segments are reserved by the system. The 0x2 segment contains a few pages that are set up by the system loader upon the exec() time. Necessary data, such as command line argument values, environment variables, and errno, are stored in these pages (see Example 3-12 on page 134). The access to the rest of the 0x2 segment is prohibited.

Segments 0x3 - 0xC and 0xE are only accessible if you specify the attaching memory address with the shmat() routine. You may use these segments to share shared memory segments between 32-bit and 64-bit processes.

In general, hardcoding the attaching memory address with the shmat() routine is bad programming. Unless it is absolutely required, your 64-bit application should not specify the attaching memory address with the shmat().

Figure 3-10 illustrates the usage of first 16 segments (4 GB) in the 64-bit user process model.

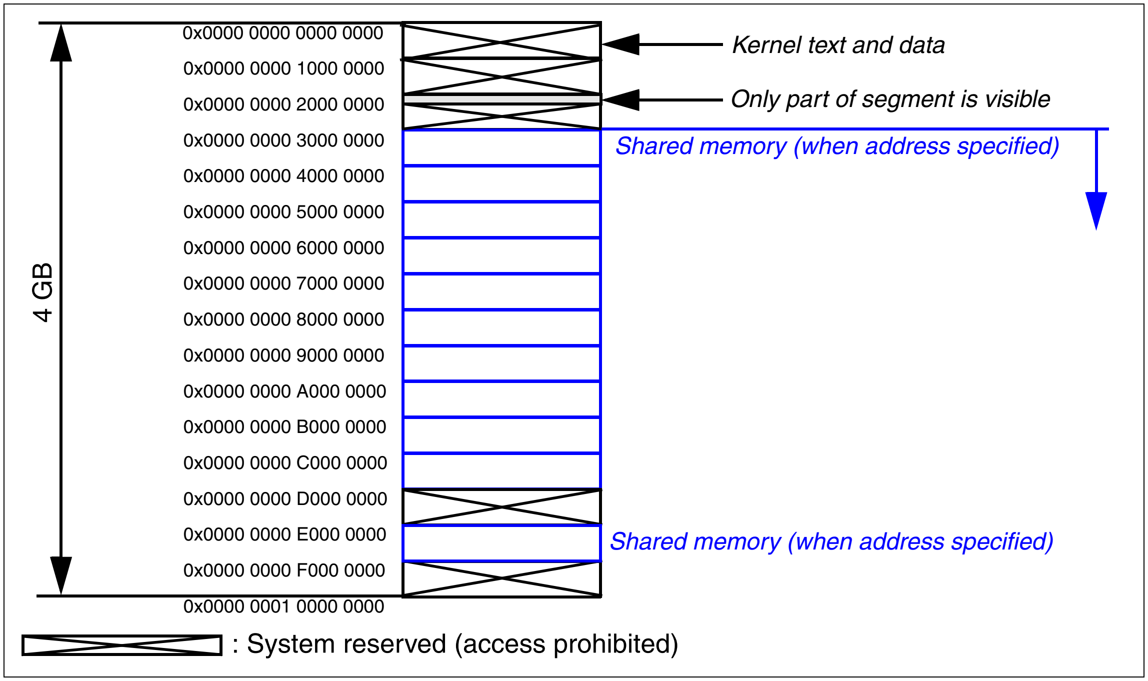


*Figure 3-10 The 64-bit memory model (4 GB, the first 16 segments)*

### 3.3.2 Application text, data, and heap (4 GB - 448 PB)

When a 64-bit program is executed, the user text is mapped into the first segment in this area. Also, user data is mapped into another segment in this area.

In both cases, if a segment is not sufficient to contain text or data, another segment will be contiguously attached to the process address space.

To demonstrate this behavior, we prepared a program listed in Example 3-13 on page 134. We compiled and ran this program, as shown in Example 3-12 on page 134. In this example, the user text is loaded into the first segment (0x0000000100000000) in this area. The user data is loaded into the second segment (0x0000000110000000) in this area. The heap memory acquired by malloc() is also allocated in this segment.

*Example 3-12 Segment mapping in the 64-bit user process model*

```
$ cc -q64 underscore_symbols_64.c
$ file a.out
a.out:          64-bit XCOFF executable or object module not stripped
$ a.out
_text =         0x00000001000001f8
_etext =        0x00000001000005e8
_data =         0x00000001100005e8
_edata =        0x0000000110000838
argv[] =        0x00000000200fe8d0
environ[] =     0x00000000200fefd8
errnop =        0x00000000200fefe8
errno =         0x00000000200fefe0
&heap_mem =     0x0fffffffffffff70
heap_mem =      0x0000000110000850
```

Example 3-13 is a program simplified from the one listed in Example 3-2 on page 113 to demonstrate the segment mapping in the 64-bit user process model.

*Example 3-13 underscore_symbols_64.c*[12]

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

extern int errnop;
extern _text;
extern _etext;
extern _data;
extern _edata;
extern char *environ[];

int
main(int argc, char *argv[])
{
    char    buf[BUFSIZ];
    char    *heap_mem;          /* heap memory. */

    if ((heap_mem = malloc(BUFSIZ)) == (void *)NULL) {
        sprintf(buf, "malloc() failed with errno = %d", errno);
        perror(buf);
    }

    printf("_text =\t\t0x%016p\n", &_text);
    printf("_etext =\t0x%016p\n", &_etext);
    printf("_data =\t\t0x%016p\n", &_data);
```

[12] Extern symbols that start with underscore are run-time symbols prepared by the system loader.

```
    printf("_edata =\t0x%016p\n", &_edata);

    printf("argv[] =\t0x%016p\n", argv);
    printf("environ[] =\t0x%016p\n", environ);
    printf("errnop =\t0x%016p\n", &errnop);
    printf("errno =\t\t0x%016p\n", &errno);

    printf("&heap_mem =\t0x%016p\n", &heap_mem);
    printf("heap_mem =\t0x%016p\n", heap_mem);

    exit(0);
}
```

As we demonstrate in the later section, the 64-bit user process model gives you a very flat memory model that is easy to use, as long as there are enough physical pages to be allocated on the system (see 3.3.8, “Resource limits in 64-bit mode” on page 136).

### 3.3.3 Default shared memory segments (448 - 512 PB)

If a 64-bit process calls shmat() or mmap() routines without specifying the attaching memory address, segments in this area will be attached to the address space contiguously.

See 3.4.4, “Shared memory limits” on page 147 for the IPC limitation in the 64-bit user process model.

### 3.3.4 Privately loaded objects (512 - 576 PB)

If objects are loaded into the address space in the following cases, those objects will be loaded into the segments in this area:

- Objects are explicitly loaded by load() and dlopen().
- The file permission modes of shared objects are r-xr-x--- (no read and execute permission bits are set for others).
- All the global shared text and data segments are full (very unlikely in the 64-bit user process model).

For further information about the private shared objects, see 2.9.2, “Private shared objects” on page 101.

### 3.3.5 Shared text and data (576 - 640 PB)

On the first load of a shared library object, the shared library text is loaded into a segment in this area. This segment will be shared by all 64-bit user processes on the system. Also, shared library data will be created in another segment in this area per process basis at the same time.

In both cases, if there is a segment that has enough free space to contain shared text or shared library data, that segment will be used. Otherwise, another segment will be attached to the process address space.

The virtual addresses of loaded shared text objects can be examined using the **genkld** command. For further information about the usage of **genkld**, see 2.7.2, “genkld” on page 88.

### 3.3.6 System reserved (640 - 960 PB)

All the segments in this area are reserved by the system and prohibited from the user process access.

### 3.3.7 User process stack (960 PB - 1 EB)

By default, a 64-bit user process uses the last segment in this area for the user process stack. The stack grows from the last address, 0x0FFF_FFFF_FFFF_FFFF, toward the first address in this area. The address of the heap_mem pointer depicts this behavior (see Example 3-12 on page 134).

To use more than one segment for user process stack, you need to specify the -bmaxstacksize linker option. For example, to specify two segments for the user process stack, you need to compile your program as follows:

```
$ cc -q64 -bmaxstacksize:0x20000000 hello.c
```

**Note:** Thread stacks for Pthreads, except the initial thread within a multi-threaded processes, are allocated in the process heap. See 8.3.4, “Thread stack” on page 292 for further information about the thread stack in multi-threaded processes.

### 3.3.8 Resource limits in 64-bit mode

Unlike the 32-bit user process model, the data resource limit value always defines the actual upper limit of allocatable heap memory in the 64-bit user process memory model.

With the default soft data limit of 128 MB (see Example 3-6 on page 125), your application can only allocate memory from a process heap up to 128 MB. To demonstrate this, we have simplified grabheap_32.c (Example 3-3 on page 121) to get rid of the complexity, because the 64-bit user process memory model is flat

(Example 3-14). If we compile and run this program under the default soft data limit, it fails to acquire a 256 MB heap:

```
$ cc grabheap_64.c
$ file a.out
a.out:          64-bit XCOFF executable or object module not stripped
$ a.out 1
malloc() with 1 x 256 MB failed with errno = 12: Not enough space
```

*Example 3-14   grabheap_64.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

#define ONE_SEG (256 * 1024 * 1024)

int main(int argc, char *argv[])
{
    char    *p;
    char    buf[BUFSIZ];
    size_t  sz;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s digit-number\n", argv[0]);
        exit(1);
    }
    sz = atoi(argv[1]);

    if ((p = malloc(sz * ONE_SEG)) == (void *)NULL) {
        sprintf(buf, "malloc() with %d x 256 MB failed with errno = %d"
                , sz, errno);
        perror(buf);
    } else {
        printf("starting address of %d x 256 MB memory heap is 0x%016p\n"
            , sz, p);
        printf("ending address of %d x 256 MB memory heap is 0x%016p\n"
            , sz, p + (sz * ONE_SEG) - 1);
    }

    exit(0);
}
```

Once the soft data limit is relaxed, the program can acquire a 256 MB heap:

```
$ ulimit -Sd unlimited
$ ulimit -Sa
time(seconds)        unlimited
file(blocks)         2097151
```

```
data(kbytes)         unlimited
stack(kbytes)        32768
memory(kbytes)       32768
coredump(blocks)     2097151
nofiles(descriptors) 2000
$ file a.out
a.out:          64-bit XCOFF executable or object module not stripped
$ a.out 1
starting address of 1 x 256 MB memory heap is 0x0000000110000850
ending address of 1 x 256 MB memory heap is 0x000000012000084f
```

If the soft data limit is set to unlimited, it is set to 9,223,372,036,854,775,807 byte = 0x7FFF_FFFF_FFFF_FFFF byte in the 64-bit user process model.[13] Example 3-15 shows the soft resource limit value in the 64-bit user process model (the source code is the same one listed in Example 3-10 on page 128).

*Example 3-15 Soft data limit set to RLIM_INIFINITY in 64-bit mode*[14]

```
$ cc -q64 printlimits.c
$ file a.out
a.out:          64-bit XCOFF executable or object module not stripped
$ a.out
Resource name                  Soft                   Hard
RLIMIT_CORE              1073741312    9223372036854775807
RLIMIT_DATA               134217728    9223372036854775807
RLIMIT_DATA               134217728    9223372036854775807
RLIMIT_FSIZE             1073741312             1073741312
RLIMIT_NOFILE                  2000    9223372036854775807
RLIMIT_STACK               33554432    9223372036854775807
RLIMIT_RSS                 33554432    9223372036854775807
$ ulimit -Sd unlimited
$ a.out | egrep '^(Resource|RLIMIT_DATA)'
Resource name                  Soft                   Hard
RLIMIT_DATA     9223372036854775807    9223372036854775807
```

Therefore, a 64-bit user process can acquire as much heap memory as it requests once the soft data limit is set to unlimited. In the following example, we specified 409600 as the command line parameter and the program successfully acquired 409,600 x 256 MB = 100 TB:

```
$ a.out 409600
starting address of 409600 x 256 MB memory heap is 0x0000000110000850
ending address of 409600 x 256 MB memory heap is 0x0000640110000850
```

[13] The value 0x7FFF_FFFF_FFFF_FFFF is defined as RLIM_INIFINITY for the 64-bit user process model in /usr/include/sys/resource.h (see Example 3-8 on page 127).

[14] The actual numerical limits might be changed in the future release of AIX. These values are applicable on AIX starting from Version 4.3 to 5.2.

Although the malloc() routine does not actually allocate virtual pages (virtual pages are allocated when the program touches them the first time), requesting this huge amount of memory puts unnecessary stress on the system.

The 64-bit user process model gives you a very flat memory model that is easy to use, but it is your responsibility to request the proper size of heap memory in your application. You may consider selecting one of the following methods to place a safety mechanism on your 64-bit programs:

- Specify the -bmaxdata:0xNNNNNNNNNNNNNNNN linker option when compiling your source codes.
- After compiling the source code, binary-edit the XCOFF header file of the generated executable file using the **/usr/ccs/bin/ldedit** command. For example:

  ```
  $ /usr/ccs/bin/ldedit -bmaxdata:0xNNNNNNNNNNNNNNNN a.out
  ```
- Specify the LDR_CNTRL environment value when running the executable. For example:

  ```
  $ LDR_CNTRL=MAXDATA=0xNNNNNNNNNNNNNNNN a.out
  ```
- Call the setrlimit() sub-routine in your application to explicitly set the soft data limit.

The appropriate value of 0xNNNNNNNNNNNNNNNN varies, depending on your application's needs and the physical memory size; however, the following numbers can be good starting points:

| | |
|---|---|
| **0x0000000040000000** | 1 GB |
| **0x0000001000000000** | 64 GB |
| **0x0000002000000000** | 128 GB |

## 3.3.9 Large file support in 64-bit model

In the 64-bit user process model, the data type off_t is always defined as long long. Therefore, 64-bit processes can handle large files as long as the following conditions are met:

- The file's hard and soft limits have been relaxed. The default value of file limit is 2,097,151 disk blocks = 1 GB (see Example 3-6 on page 125).
- JFS2 or large file enabled JFS is used.

## 3.4 Introduction to shared memory

The Inter-Process Communication (IPC) facilities are used by processes to communicate with each other and to synchronize their activities.[15] IPC facilities, semaphores, message queues, and shared memory are quite common services in the modern UNIX operating systems, including AIX. In this section, we focus on the shared memory on AIX.

The shared memory is usually used for the following purposes:

- Sharing memory segments between processes

  Mapped shared memory segments can serve as a large pool for exchanging data among processes. The mechanism does not provide locks or access control among the processes by default. Therefore, processes using shared memory areas must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another is using. Shared memory segments can be most beneficial when the amount of data to be exchanged between processes is too large to transfer with message queues or when many processes have to share common data.

- Mapping files into memory segments

  Memory mapped files provide a mechanism for a process to access files by directly incorporating file data into the process address space. The use of mapped files can significantly reduce I/O data movement because the file data does not have to be copied into process data buffers, as is done by the read and write subroutines. When more than one process maps the same file, its contents are shared among them, therefore providing a low-overhead mechanism by which processes can synchronize and communicate.

However, once shared memory segments are attached to the address space of your application process, those segments do not have to be shared with other processes. Therefore, some applications exploit shared memory segments as if they are an *extended* memory heap (see 4.4, “Heap management using MEMDBG” on page 199 for detail).

AIX supports the following two well-known services for memory mapping:

- Section 3.4.1, “The shmat services” on page 141
- Section 3.4.2, “The mmap services” on page 145

Both services address the same type of memory usage, but shmat is normally used to create and use shared memory segments within a program, and the

[15] Signals are also commonly used for IPC in the UNIX operating systems. However, we do not discuss signals in this redbook, because the signal management on AIX conforms very well to many UNIX standards, such as POSIX.

mmap is mostly used for mapping files into the process address space, although it can be used for creating shared memory segments as well as mapping files.

The term *shared memory segments* is widely used to refer to the memory chunks allocated by these two services on UNIX operating systems, including AIX. Although a shared memory segment smaller than 256 MB usually consumes a 256 MB *segment* on AIX, if the EXTSHM=ON environment variable is not defined, the term is not the same concept referenced by *segments* (explained in 3.2, "The 32-bit user process model" on page 109).

For further information about the topics explained in this section, please refer to the "Shared Libraries and Shared Memory" section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 3.4.1 The shmat services

The shmat services supported by AIX include the following routines:

| | |
|---|---|
| **shmat()** | Attaches a shared memory segment to a process. |
| **shmctl()** | Controls shared memory operations. |
| **shmget()** | Gets or creates a shared memory segment. |
| **shmdt()** | Detaches a shared memory segment from a process. |
| **ftok()** | Provides the key that the shmget() subroutine uses to create the shared segment. |

These subroutine references are provided in Appendix C, "Subroutine references for shmat and mmap services" on page 447.

### Terms used in the shmat services

In the shmat services, the following terms are frequently used:

**key** The unique identifier of a particular shared segment. It is associated with the shared segment as long as the shared segment exists. In this respect, it is similar to the file name of a file.

**shmid** The identifier assigned to the shared segment for use within a particular process. It is similar in use to a file descriptor for a file.

**attach** Specifies that a process must attach a shared segment in order to use it. Attaching a shared segment is similar to opening a file.

**detach** Specifies that a process must detach a shared segment once it is finished using it. Detaching a shared segment is similar to closing a file.

There are some well-established programming disciplines about the use of these routines. We introduce a few of them in this section.

### Mapping memory with shmat()

The following pseudo-code shows how to use shared memory segments with shmat services:

1. Create a key to uniquely identify the shared segment. Use the ftok() subroutine to create the key. For example, to create the mykey key using a variable, proj (integer type), and a file name of /path/some_file, insert the following statement:

   ```
   mykey = ftok(“/path/some_file”, proj);
   ```

2. Either:

   a. Create a shared memory segment with the shmget() subroutine. For example, to create a shared segment that contains 4096 bytes and assign its ID to an integer variable, mem_id, insert the following statement:

      ```
      mem_id = shmget(mykey, 4096, IPC_CREAT | permission_modes);
      ```

      You must define appropriate permission_modes ORing with S_ macros (see Appendix C, “Subroutine references for shmat and mmap services” on page 447).

   b. Get a previously created shared segment with the shmget() subroutine. For example, to get a shared segment that is already associated with the mykey key and assign the ID to an integer variable, mem_id, insert the following statement:

      ```
      mem_id = shmget(mykey, 0, IPC_ACCESS);
      ```

3. Attach the shared segment to the process with the shmat() subroutine. For example, to attach a previously created segment, insert the following statement:

   ```
   ptr = shmat(mem_id, 0, SHM_MAP);
   ```

   If you specify address 0 as the second parameter of shmat(), then the operating system will select the attaching target address. This is the recommended programming manner.

   The variable ptr is a pointer to a structure that defines the fields in the shared segment. Use this structure to store and retrieve data in the shared segment. The definition of the structure should be the same for all processes using the segment.

4. Work with the data in the segment through ptr.
5. Detach from the segment using the shmdt() subroutine:

   ```
   shmdt(ptr);
   ```

6. If the shared segment is no longer needed, remove it from the system with the shmctl() subroutine:

   ```
   shmctl(mem_id, IPC_RMID, ptr);
   ```

If a shared memory segment is created accordingly, multiple processes can attach it into their address space, as illustrated in Figure 3-11. In this figure, two processes, A and B, are sharing the sheared memory segment, which is attached to 0x3 in the process A's address space and attached to 0x4 in the process B's address space. If the process A writes data in this memory area, the process B can instantly read the data value from it.

Although it is a very efficient way to transfer data between processes, a data synchronization algorithm must be used between processes. Semaphores are commonly used to implement this data synchronization algorithm.

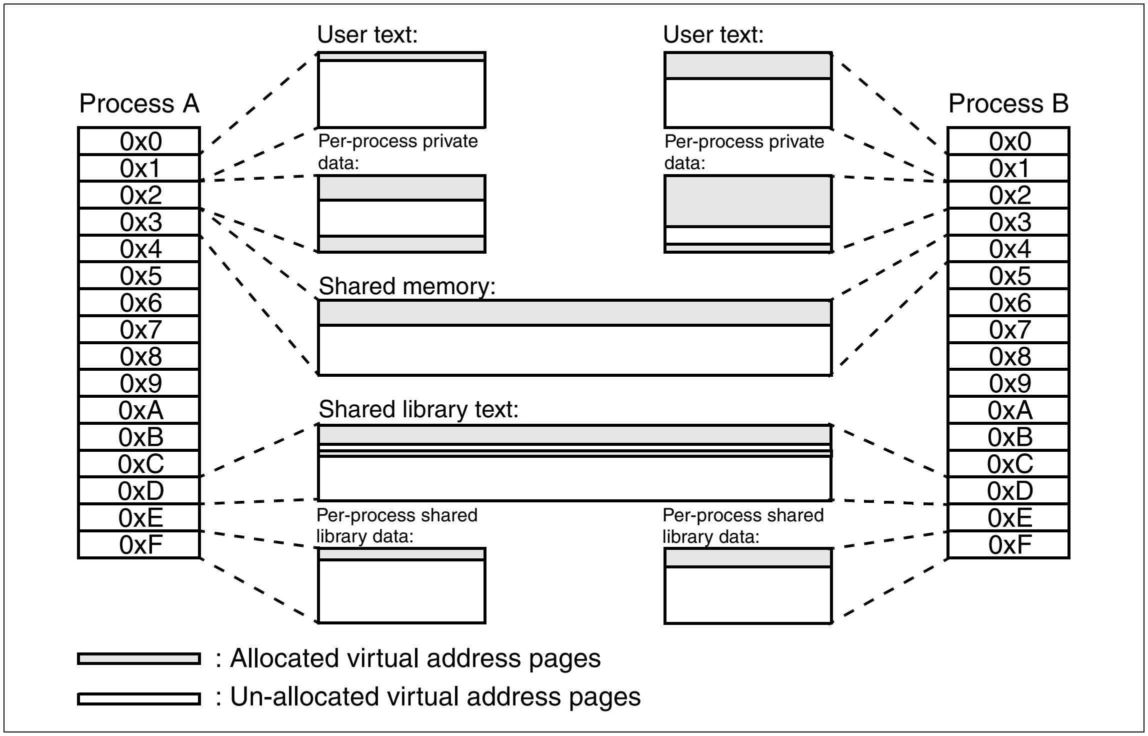

Process A
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xA
0xB
0xC
0xD
0xE
0xF
User text:
User text:
Per-process private data:
Per-process private data:
Shared memory:
Shared library text:
Per-process shared library data:
Per-process shared library data:
Process B
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xA
0xB
0xC
0xD
0xE
0xF
▨ : Allocated virtual address pages
□ : Un-allocated virtual address pages

*Figure 3-11 Shared memory segments between two processes*

The shared memory segments created by the following methods can be shared between processes:

- The MAP_SHARED flag is specified with mmap().
- Appropriate permission modes are specified with shmget().
- After creating a shared memory segment with shmget(), its permission modes are altered accordingly with the IPC_SET command flag with shmctl().

## Mapping files with shmat()

The following pseudo-code shows how to map files with shmat services. To do so, you need to map the file first, then set the end of the mapped file using lseek(). The shmat services do not provide any mechanisms to detect the end of mapped file.

1. To create the mapped data file:
   a. Open (or create) the file and save the file descriptor:

   ```
   if ((fildes = open(filename, O_RDWR)) < 0) {
       printf("cannot open file\n");
       exit(1);
   }
   ```

   b. Map the file to a segment with the shmat subroutine:

   ```
   file_ptr = shmat(fildes, 0, SHM_MAP);
   ```

   The SHM_MAP constant is defined in the /usr/include/sys/shm.h file. This constant indicates that the file is a mapped file. Include this file and the other shared memory header files in a program with the following directives:

   ```
   #include <sys/shm.h>
   ```

2. To detect the end of the mapped file:
   a. Use the lseek() subroutine to go to the end of file:

   ```
   eof = file_ptr + lseek(fildes, 0, SEEK_END);
   ```

   This example sets the value of eof to an address that is 1 byte beyond the end of file. Use this value as the end-of-file marker in the program.

   b. Use file_ptr as a pointer to the start of the data file, and access the data as if it were in memory:

   ```
   while (file_ptr < eof) {
       .
       .
       .
       (references to file using file_ptr)
   }
   ```

**Note:**

- A file system object should not be simultaneously mapped using both the shmat and mmap services.
- Unexpected results may occur when references are made beyond the end of the object.[a] In the current implementation, mapping files with the shmat services allocate only enough 256 MB segments to hold the existing file. If the process attempts to access the memory area beyond the allocated shared memory segments using a pointer, the SIGSEGV signal will be delivered to the process.

[a] When a process maps a file using shmat(), only enough 256 MB segments to hold the file will be allocated in the current AIX implementation. If the memory area beyond the allocated shared memory segments is referenced by a pointer, the SIGSEGV signal will be delivered to the process.

c. Close the file when the program is finished working with it:

```
close(fildes);
```

## 3.4.2 The mmap services

The mmap services supported by AIX include the following routines:

| | |
|---|---|
| **mmap()** | Maps an object file into virtual memory. |
| **madvise()** | Advises the system of a process' expected paging behavior. |
| **mincore()** | Determines residency of memory pages. |
| **mprotect()** | Modifies the access protections of memory mapping. |
| **msync()** | Synchronizes a mapped file with its underlying storage device. |
| **munmap()** | Unmaps a mapped memory segment. |

These subroutine references are provided in Appendix C, “Subroutine references for shmat and mmap services” on page 447.

There are some well-established programming disciplines for the use of these routines. We introduce a few of them in this section.

### Mapping files with mmap()

The following pseudo-code shows how to map files with the mmap services:

1. Create a file descriptor, fd, for a file system object using the open() routine:

   ```
   fd = open(pathname, permissions);
   ```

2. Determine the file length by using the lseek() routine. For example:

   ```
   len = lseek(fd, 0, SEEK_END)
   ```

3. Map the file into the process address space with the mmap subroutine. For example, to map the file for the file descriptor fd, starting at address addr, using len bytes of size, with the access permissions defined by prot and 0 bytes of offset, use the statement:

   ```
   ptr = mmap(addr, len, prot, MAP_FILE, fd, 0)
   ```

   This specifies the creation of a new mapped file segment by mapping the file associated with the fd file descriptor. The mapped segment can extend beyond the end of the file, both at the time when the mmap subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the mmap subroutine, or if a file was later truncated.

   If you specify address 0 as the first parameter of mmap(), then the operating system will select the attaching target address. This is the recommended programming manner.

4. Work with the data in the segment.
5. The file descriptor can be closed by using:

   ```
   close(fd);
   ```

6. Detach from the segment using the munmap subroutine:

   ```
   munmap(addr, len);
   ```

**Note:** A file system object should not be simultaneously mapped using both the shmat and mmap services.

## Mapping memory with mmap()

The following pseudo-code shows how to map memory segments with mmap services:

1. If you specify MAP_ANONYMOUS instead of MAP_FILE for the mmap()'s fourth parameter, then the file descriptor parameter must be specified as -1. This is called an *anonymous memory segment*, because there is no file system object associated. All pages in the anonymous memory segment are 0-filled:

   ```
   ptr = mmap(addr, len, prot, MAP_ANONYMOUS, -1, 0)
   ```

   This memory segment pointed to by ptr can be shared only with the descendants of the current process.

2. Work with the data in the segment.
3. Detach the segment from the address space using the munmap() subroutine:

   ```
   munmap(addr, len);
   ```

## 3.4.3 Difference between shmat and mmap services

Although both the shmat and mmap services can be used for the purposes explained in 3.4, "Introduction to shared memory" on page 140, there are significant differences between these two services:

- The address for a shared memory segment mapped by shmat() must be SHMLBA (256 MB) aligned if the EXTSHM=ON environment variable is not set, while mmap() works on a page-size alignment (4 KB).
- The mmap() memory mappings are a process resource and thus are cleaned up at process exit-time. The shmget() shared memory segments are a system-wide resource and are not cleaned up at process exit-time.

Therefore, use the shmat services under the following circumstances:

- When mapping shared memory regions that need to be shared among unrelated processes (no parent-child relationship).
- When mapping entire files.

Use the mmap services under the following circumstances:

- Portability of the application is a concern.
- Many files are mapped simultaneously.
- Only a portion of a file needs to be mapped.
- Page-level protection needs to be set on the mapping.
- Private mapping (MAP_PRIVATE) is required.

> **Note:** Section 3.4.4, "Shared memory limits" on page 147 and 3.5, "Shared memory segments allocation order" on page 149 are only applicable to the shared memory segments created by shmat().

## 3.4.4 Shared memory limits

Shared memory limits vary, depending on AIX versions, as shown in Table 3-1 on page 148.

*Table 3-1 Shared memory limits*

| Description | AIX V4.3.0 | 4.3.1 | 4.3.2, 4.3.3 | 5.1 | 5.2 |
|---|---|---|---|---|---|
| Maximum segment size (32-bit) | 256 MB | 2 GB | 2 GB | 2 GB | 2 GB |
| Maximum segment size (64-bit) | 256 MB | 2 GB | 2 GB | 64 GB | 1 TB |
| Minimum segment size | 1 | 1 | 1 | 1 | 1 |
| Maximum number of shared memory IDs | 4096 | 4096 | 131072 | 131072 | 131072 |
| Maximum number of segments per process (32-bit) | 11 | 11 | 11 | 11 | 11 |
| Maximum number of segments per process (64-bit) | 268435456 | 268435456 | 268435456 | 268435456 | 268435456 |

To confirm system constants regarding shared memory, it is recommended to call the vmgetinfo()[16] routine instead of hardcoding numerical values (see the program example shown in Example 3-17 on page 149).

Example 3-16 shows the output from this program. Please compare the values in this example with values for AIX 5L Version 5.2 in the Table 3-1.

*Example 3-16 Output from vmgetinfo() routine*

```
$ a.out
/* Shared memory limits */
max # of shared memory id's =                            131072
64bit proc max shm segment size =                         1099511627776
32bit proc max shm segment size =                         2147483648
min shared memory segment size =                          1
max # of shm segs per 64bit proc =                       268435456
max # of shm segs per 32bit proc (without EXTSHM=ON) =  11
```

Example 3-17 on page 149 shows an example program to call the vmgetinfo() routine.

[16] The vmgetinfo() routine is supported on AIX 5L Version 5.2 and later.

*Example 3-17 vmgetinfo.c*[17]

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/vminfo.h>

int main(int argc, char *argv[])
{
    int                 rc;
    char                buf[BUFSIZ];
    struct ipc_limits   ipc_lim;

    if ((rc = vmgetinfo(&ipc_lim, IPC_LIMITS, sizeof(struct ipc_limits))) != 0)
    {
        sprintf(buf, "vmgetinfo() at %d in %s failed with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
        exit(1);
    }

    printf("/* Shared memory limits */\n");
    printf("max # of shared memory id's =\t\t\t\t%lld\n", ipc_lim.shmmni);
    printf("64bit proc max shm segment size =\t\t\t%lld\n", ipc_lim.shmmax64);
    printf("32bit proc max shm segment size =\t\t\t%u\n", ipc_lim.shmmax32);
    printf("min shared memory segment size =\t\t\t\t%u\n", ipc_lim.shmmin);
    printf("max # of shm segs per 64bit proc =\t\t\t%lld\n", ipc_lim.shmseg64);
    printf("max # of shm segs per 32bit proc (without EXTSHM=ON) =\t%u\n"
        , ipc_lim.shmseg32);

    exit(0);
}
```

## 3.5 Shared memory segments allocation order

This section explains how shared memory segments are allocated in the process address space. To demonstrate the allocation order, we have prepared the program shown in Example 3-19 on page 151. To compile this program, type the following command:

```
$ cc -DCALL_SHMDT grabshm.c
```

If -DCALL_SHMDT is not specified, then the shmdt() and shmctl() routines will not be called in the program; it then leaves shared memory segments after the

[17] The structure type ipc_limits is defined in the /usr/include/sys/vminfo.h header file.

execution. If the program is executed by the ausres01 user, it then leaves the shared memory segments highlighted in Example 3-18.

*Example 3-18 ipcs -m*

```
$ ipcs -m
IPC status from /dev/mem as of Sun Feb  9 18:21:48 CST 2003
T        ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m         0 0xe4663d62 --rw-rw-rw-   imnadm   imnadm
m         1 0x9308e451 --rw-rw-rw-   imnadm   imnadm
m         2 0x52e74b4f --rw-rw-rw-   imnadm   imnadm
m         3 0xc76283cc --rw-rw-rw-   imnadm   imnadm
m         4 0x298ee665 --rw-rw-rw-   imnadm   imnadm
m         5 0xffffffff --rw-rw----     root   system
m         6 0xffffffff --rw-rw----     root   system
m     48159 0xffffffff --rw------- ausres01  itsores
m     48160 0xffffffff --rw------- ausres01  itsores
m     48161 0xffffffff --rw------- ausres01  itsores
m     48162 0xffffffff --rw------- ausres01  itsores
```

To remove stale shared memory segments, type **ipcrm -m shared_memory_ID**. The shared_memory_IDs are shown in the second column in the example. If you need to remove many shared memory segments, do the following:

```
$ ipcs -m | grep user_name | awk '{print $2}' | while read id
> do
> ipcrm -m $id
> done
```

Where *user_name* is the user name of the application process that acquired the shared memory segments.

Example 3-19 on page 151 is a sample program to allocate shared memory segments. When executed, it requires two arguments:

```
Usage: a.out [-m|-s] digit-number
```

If you specify the -m option, then it tries to allocate multiplies of a 1 MB shared memory segment. If -s specified, it tries to allocate multiplies of a 256 MB shared memory segment. The argument digit-number specifies how many shared memory segments should be allocated.

In this program, the created shared memory segments cannot be shared with other processes, because the IPC_PRIVATE flag is specified with the shmget() routine. Also, please note that the attaching target address is specified as (void *)0 with the shmat() routine, so that we instruct the system to select the attaching target address of shared memory segments.

*Example 3-19 grabshm.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <strings.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MAX_SHM_SEGMENTS 131072
#define ONE_MB (1024 * 1024)
#define ONE_SEG (256 * 1024 * 1024)

char *size_of(int i)
{
    if (i == ONE_MB) {
        return("1 MB");
    } else if (i == ONE_SEG) {
        return("256 MB");
    } else {
        exit(1);
    }
}

int main(int argc, char *argv[])
{
    char    *shmptr[MAX_SHM_SEGMENTS];
    char    buf[BUFSIZ];
    int     shmid[MAX_SHM_SEGMENTS];
    int     size_flag;
    int     cnt, max, rc;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s [-m|-s] digit-number\n", argv[0]);
        exit(1);
    } else {
        if (!strcmp(argv[1], "-m")) {
            size_flag = ONE_MB;
        } else if (!strcmp(argv[1], "-s")) {
            size_flag = ONE_SEG;
        } else {
            fprintf(stderr, "Usage: %s [-m|-s] digit-number\n", argv[0]);
            exit(1);
        }
        max = atoi(argv[2]);
    }

    for (cnt = 0; cnt < max; cnt++) {
        if ((shmid[cnt] = shmget(IPC_PRIVATE, size_flag
                            , IPC_CREAT | S_IRUSR | S_IWUSR)) < 0) {
            sprintf(buf
                    , "[%2d] shmget(\"%s\") at %d in %s failed with errno = %d"
                    , cnt, size_of(size_flag), __LINE__, __FILE__, errno);
            perror(buf);
        }
        if ((shmptr[cnt] = shmat(shmid[cnt], (void *)0, 0)) == (void *) -1) {
            sprintf(buf, "[%2d] shmat() at %d in %s failed with errno = %d"
                , cnt, __LINE__, __FILE__, errno);
            perror(buf);
        } else {
            printf(
            "[%2d] beginning address of %s shared memory segment is 0x%016p\n"
            , cnt, size_of(size_flag), shmptr[cnt]);
            printf(
            "[%2d] ending address of %s shared memory segment is 0x%016p\n"
            , cnt, size_of(size_flag), shmptr[cnt] + (size_flag - 1));
        }
    }

    /* if you comment out the following code block,
     * you need to explicitly remove the allocated shared memory segment
     * by calling ipcrm -m ID.
     */
#if defined(CALL_SHMDT)
    for (cnt = 0; cnt < max; cnt++) {
        if ((rc = shmdt(shmptr[cnt])) == -1) {
            sprintf(buf, "[%2d] shmdt() at %d in %s failed with errno = %d"
                , cnt, __LINE__, __FILE__, errno);
            perror(buf);
        }

        if (shmctl(shmid[cnt], IPC_RMID, 0) < 0) {
            sprintf(buf, "[%2d] shmctl() at %d in %s failed with errno = %d"
                , cnt, __LINE__, __FILE__, errno);
            perror(buf);
        }
    }
#endif

    exit(0);
}
```

Please note that this program is written in order to demonstrate the order of shared memory segments allocation. If you need to allocate a larger size of shared memory segment, you can get it with the one shmget() call.

### 3.5.1 Order in the 32-bit default memory model

If the program shown in Example 3-19 on page 151 is executed in the 32-bit default memory model, then segments 0x3 - 0xE will be sequentially used, excluding 0xD.

Example 3-20 shows the output from this program, when we have attempted to allocate twelve 256 MB shared memory segments. Because the 32-bit user process model supports up to 11 shared memory segments by default, if the EXTSHM=ON environment variable is not specified, the last attempt to allocate the 12th shared memory segment would fail.

*Example 3-20 Acquiring twelve 256 MB shared memory segments*[18]

```
$ a.out -s 12
[ 0] beginning address of 256 MB shared memory segment is 0x0000000030000000
[ 0] ending address of 256 MB shared memory segment is 0x000000003fffffff
[ 1] beginning address of 256 MB shared memory segment is 0x0000000040000000
[ 1] ending address of 256 MB shared memory segment is 0x000000004fffffff
...
[ 9] beginning address of 256 MB shared memory segment is 0x00000000c0000000
[ 9] ending address of 256 MB shared memory segment is 0x00000000cfffffff
[10] beginning address of 256 MB shared memory segment is 0x00000000e0000000
[10] ending address of 256 MB shared memory segment is 0x00000000efffffff
[11] shmat() at 58 in grabshm.c failed with errno = 24: Too many open files
[11] shmdt() at 76 in grabshm.c failed with errno = 22: Invalid argument
```

Example 3-21 shows the output from this program, when we have attempted to allocate twelve 1 MB shared memory segments. Because the 32-bit user process model supports up to 11 shared memory segments, by default, if the EXTSHM=ON environment variable is not specified, the last attempt to allocate the 12th shared memory segment would fail, even if the total shared memory size is just 11 MB.

*Example 3-21 Acquiring twelve 1 MB shared memory segments*

```
$ a.out -m 12
[ 0] beginning address of 1 MB shared memory segment is 0x0000000030000000
[ 0] ending address of 1 MB shared memory segment is 0x00000000300fffff
[ 1] beginning address of 1 MB shared memory segment is 0x0000000040000000
[ 1] ending address of 1 MB shared memory segment is 0x00000000400fffff
...
[ 9] beginning address of 1 MB shared memory segment is 0x00000000c0000000
[ 9] ending address of 1 MB shared memory segment is 0x00000000c00fffff
[10] beginning address of 1 MB shared memory segment is 0x00000000e0000000
[10] ending address of 1 MB shared memory segment is 0x00000000e00fffff
```

[18] The value 24 of errno means EMFILE (too many open files) and the value 22 of errno means EINVAL (invalid argument).

```
[11] shmat() at 58 in grabshm.c failed with errno = 24: Too many open files
```

You may notice the system selected different segments for allocating just 1MB size shared memory segments. For example, the 0x300FFFFF - 0x3FFFFFFF address cannot be used to allocate another shared memory segment.

### 3.5.2 Order in the 32-bit very large memory model with DSA

If the program shown in Example 3-19 on page 151 is executed in the 32-bit very large memory model with DSA, then segments 0xF - 0x3 would be sequentially used; however, higher address segments cannot be used, depending on the maxdata variable.

Example 3-22 shows the output from this program, when we have attempted to allocate eleven 1 MB shared memory segments with LDR_CNTRL=MAXDATA=0x80000000@DSA. Because the 0x3 segment is always reserved for the process heap in the very large memory model, the 11th attempt to allocate another shared memory segment failed.

**Note:** If segments 0x3 - 0xE were attached to the process address heap for the process heap before allocating shared memory segments, this program could have failed earlier.

*Example 3-22 LDR_CNTRL=MAXDATA=0x80000000@DSA*

```
$ LDR_CNTRL=MAXDATA=0x80000000@DSA a.out -m 11
[ 0] beginning address of 1 MB shared memory segment is 0x00000000e0000000
[ 0] ending address of 1 MB shared memory segment is 0x00000000e00fffff
[ 1] beginning address of 1 MB shared memory segment is 0x00000000c0000000
[ 1] ending address of 1 MB shared memory segment is 0x00000000c00fffff
...
[ 8] beginning address of 1 MB shared memory segment is 0x0000000050000000
[ 8] ending address of 1 MB shared memory segment is 0x00000000500fffff
[ 9] beginning address of 1 MB shared memory segment is 0x0000000040000000
[ 9] ending address of 1 MB shared memory segment is 0x00000000400fffff
[10] shmat() at 58 in grabshm.c failed with errno = 24: Too many open files
[10] shmdt() at 76 in grabshm.c failed with errno = 22: Invalid argument
```

Example 3-23 on page 155 shows the output from this program, when we have attempted to allocate fourteen 1 MB shared memory segments with LDR_CNTRL=MAXDATA=0@DSA. Because the program ran out of all the available 13 segments, the 14th attempt to allocate another shared memory segment failed.

*Example 3-23 LDR_CNTRL=MAXDATA=0@DSA*

```
$ LDR_CNTRL=MAXDATA=0@DSA a.out -m 14
[ 0] beginning address of 1 MB shared memory segment is 0x00000000f0000000
[ 0] ending address of 1 MB shared memory segment is 0x00000000f00fffff
[ 1] beginning address of 1 MB shared memory segment is 0x00000000e0000000
[ 1] ending address of 1 MB shared memory segment is 0x00000000e00fffff
...
[11] beginning address of 1 MB shared memory segment is 0x0000000040000000
[11] ending address of 1 MB shared memory segment is 0x00000000400fffff
[12] beginning address of 1 MB shared memory segment is 0x0000000030000000
[12] ending address of 1 MB shared memory segment is 0x00000000300fffff
[13] shmat() at 58 in grabshm.c failed with errno = 24: Too many open files
```

## 3.5.3 Extended mode shared memory segments

As explained in the previous sections, up to 11 shared memory segments are available in the 32-bit default memory model and up to 13 shared memory segments are available in the 32-bit very large memory model.

Although most 32-bit applications are satisfied with this limitation, some applications require more shared memory segments. To address this requirement, AIX supports another type of shared memory segments in the 32-bit user process called *extended mode* shared memory segments (EXTSHM).[19]

The extended mode shared memory segment capability is a process basis dynamic feature. To use the capability, simply define the following environment variable before executing your applications:

```
EXTSHM=ON
```

**Note:**

- The keyword must be upper case characters. EXTSHM=on is invalid.
- Do not insert this line into the /etc/environment file on your system. Some 32-bit applications do not support this capability. Please consult with the publications shipped with the software products installed on your system.

Once defined, a 32-bit process can have shared memory segments up to 131,072 on AIX Version 4.3.2 and later.

Example 3-24 on page 156 shows the output from the program, when we have attempted to allocate 100 1 MB shared memory segments with EXTSHM=ON.

[19] The extended mode shared memory segment capability has been supported since AIX Version 4.2.1.

*Example 3-24 Acquiring 100 shared memory segments with EXTSHM=ON*

```
$ EXTSHM=ON a.out -m 100
[ 0] beginning address of 1 MB shared memory segment is 0x0000000030000000
[ 0] ending address of 1 MB shared memory segment is 0x00000000300fffff
[ 1] beginning address of 1 MB shared memory segment is 0x0000000030100000
[ 1] ending address of 1 MB shared memory segment is 0x00000000301fffff
...
[98] beginning address of 1 MB shared memory segment is 0x0000000036200000
[98] ending address of 1 MB shared memory segment is 0x00000000362fffff
[99] beginning address of 1 MB shared memory segment is 0x0000000036300000
[99] ending address of 1 MB shared memory segment is 0x00000000363fffff
```

### Restrictions of extended shared memory segments

Although they seem very convenient, the extended shared memory segments have some restrictions, which are described in the following:

- The EXTSHM environment variable will be ignored and has no effect in the 64-bit user process model.
- When attaching a segment larger than 268,431,360 bytes (256 MB - 4 KB), the EXTSHM environment variable will be ignored, and the process attaches the segment with a granularity of 256 MB.
- When calling the mmap() routine, the EXTSHM environment variable will be ignored, and the process attaches the segment with a granularity of 256 MB.
- The SHM_SIZE parameter of shmctl() is not supported for segments created with EXTSHM=ON.
- No raw I/O is allowed for segments created with EXTSHM=ON.

## 3.5.4 Order in the 64-bit memory model

Example 3-25 shows the output from this program, when we have attempted to allocate twelve 1 MB shared memory segments in the 64-bit user process model. Each 1 MB shared memory segment is allocated into separate 256 MB segments, starting from 0x0700_0000_0000_0000.

*Example 3-25 Acquiring 12 shared memory segments in 64-bit mode*

```
$ cc -q64 -DCALL_SHMDT grabshm.c
$ file a.out
a.out:          64-bit XCOFF executable or object module not stripped
$ a.out -m 12
[ 0] beginning address of 1 MB shared memory segment is 0x0700000000000000
[ 0] ending address of 1 MB shared memory segment is 0x07000000000fffff
[ 1] beginning address of 1 MB shared memory segment is 0x0700000010000000
[ 1] ending address of 1 MB shared memory segment is 0x07000000100fffff
```

```
...
[10] beginning address of 1 MB shared memory segment is 0x07000000a0000000
[10] ending address of 1 MB shared memory segment is 0x07000000a00fffff
[11] beginning address of 1 MB shared memory segment is 0x07000000b0000000
[11] ending address of 1 MB shared memory segment is 0x07000000b00fffff
```

Although it is technically possible that a 64-bit process can request shared memory segments up to 131,072 on AIX Version 4.3.2 or later, it is your responsibility to assure that your application will not consume all the available system memory.

**Note:** The EXTSHM environment variable will be ignored and has no effect in the 64-bit user process model.

## 3.6 Large page support

Historically, AIX supported 4 KB page size only. Starting with AIX 5L Version 5.1 plus 5100-02 Recommended Maintenance Level, AIX supports alternate page size (called *large pages*), in addition to the traditional 4 KB page size. The large page size on systems using the POWER4 processor is 16 MB, but the large page size may be different size on future architectures. It is recommended to call the sysconf() routine with the _SC_LARGE_PAGESIZE parameter in your applications, in order to determine the supported large page size.

To verify you are using POWER4 processor systems, run the **lscfg -vpl procX** command, where *X* is the instance number of processors. The following example shows the output of this command executed on an AIX 5L Version 5.2 partition on the pSeries 690:

```
# lsdev -Cc processor
proc0      Available 00-00        Processor
proc1      Available 00-01        Processor
# lscfg -vpl proc0
  proc0            U1.18-P1-C1  Processor

        Device Specific.(YL)........U1.18-P1-C1

PLATFORM SPECIFIC

  Name:  PowerPC,POWER4
    Node:  PowerPC,POWER4@0
    Device Type:  cpu
    Physical Location: U1.18-P1-C1
```

The following sections are excerpted from the technical white paper, *AIX Support for Large Pages*, found at:

http://www.ibm.com/servers/aix/whitepapers/large_page.html

### 3.6.1 Large page support overview

Large page usage is primarily intended to provide performance improvements to high performance computing (HPC) applications. Memory access intensive applications that use large amounts of virtual memory may obtain performance improvements by using large pages. The large page performance improvements are attributable to reduced translation look-aside buffer (TLB) misses due to the TLB being able to map a larger virtual memory range. Large pages also improve memory prefetching by eliminating the need to restart prefetch operations on 4 KB boundaries.

The POWER4 large page architecture requires that all virtual pages in a 256 MB segment be the same size. AIX uses this architecture to support a *mixed mode* process model. Some segments in a process are backed with 4 KB pages and 16 MB pages back other segments. Applications may request that their heap segments be backed with large pages. Applications may also request that shared memory segments be backed with large pages. Other segments in a process are backed with 4 KB pages.

AIX supports large page usage with both 32- and 64-bit applications. Both the 32- and 64-bit versions of the AIX kernel support large pages.

AIX maintains separate 4 KB and 16 MB size physical memory pools. The customer specifies the amount of physical memory in the 16 MB memory pool using the **vmo** command on AIX 5L Version 5.2.[20] This amount of physical memory is allocated to the 16 MB memory pool at boot time. The remaining physical memory is used to back 4 KB virtual pages. The size of the 16 MB pool is fixed at boot time and cannot be changed without rebooting the system.

On AIX Versions of 5.1 and 5.2, large pages are not paged and treated as pinned memory. Therefore, an application’s data backed by large pages remains in physical memory until the application completes.[21] A security access control prevents unauthorized applications from using large pages. This prevents unauthorized applications from using large page physical memory and preventing authorized users from using large pages for their applications.

[20] On AIX 5L Version 5.1, use the **vmtune** command instead of **vmo**.

[21] The implementation of large pages may be changed in the future versions of AIX. Do not depend on large pages being pinned when developing your applications.

## 3.6.2 Large page application usage

Applications may use large pages in two ways. An application may request that large pages back its data and heap segments. An application may also request shared memory segments be backed by large pages.

### Large page data/heap segments

An application may request that its initialized program data, uninitialized program data (BSS), and heap segments be backed with large pages. There are two ways to request large pages back an application's data/heap segments:

- The executable file can be marked to request large pages.
- An environment variable can be set to request large pages.

A program's large page data/heap use is established when the program is exec()ed. A program cannot switch modes after it has begun executing. Large page use is inherited by children processes on fork().

#### *Marking an executable for large page use*

The XCOFF header in an executable file contains a new flag to indicate that the program wants to use large pages to back its data and heap segments. This flag can be set when the application is linked by specifying the -blpdata option on the **`ld`** command. The flag can also be set or cleared using the **`ldedit`** command. The **`ldedit –blpdata`** *filename* command sets the large page data/heap flag in the specified file. The **`ldedit –bnolpdata`** *filename* clears the large page flag. The **`ldedit`** command may also be used to set an executable's maxdata value. To check if the flags are correctly set, see 3.2.5, "Checking large memory model executables" on page 124.

#### *Environment variables for large page use*

An environment variable is provided to allow users to indicate they want an application to use large pages for an application's data and heap segments. The environment variable takes precedence over the executable large page flag. Large page usage is provided as options on the LDR_CNTRL environment variable.

**LDR_CNTRL=LARGE_PAGE_DATA=Y**

Specifies that the exec()ed program should use large pages for its data and heap segments. This is the same as marking the executable to use large pages.

**LDR_CNTRL=LARGE_PAGE_DATA=N**

Specifies that the exec()ed program should not use large pages for its data and heap segments. This overrides the setting in a executable marked to use large pages.

**LDR_CNTRL=LARGE_PAGE_DATA=M**

Specifies that the exec()ed program should use large pages in a mandatory mode for its data and heap segments.

You can separate multiple options on the LDR_CNTRL environment variables by using an '@' character. For example, the following LDR_CNTRL environment variable setting requests large page usage along with the maxdata option:

```
LDR_CNTRL=MAXDATA=0x80000000@LARGE_PAGE_DATA=Y
```

Users are advised to be cautious in their use of the environment variable to specify large page usage. Performance tests have shown there can be a significant performance loss in environments where a number of shell scripts or small, short running applications are invoked. One example saw a shell script's execution time increase over 10 times when the large page environment variable was specified. Customers are advised to only set the large page environment variable around specific applications that can benefit from large page usage.

### *Advisory and mandatory modes*

An application can indicate that it wants to use large pages for data/heap segments in either *advisory* or *mandatory* mode. In advisory mode, the application will use large pages if possible. The conditions needed to use large pages are:

- The user ID is authorized to use large pages.
- The system is running on a machine that has the POWER4 large page architecture feature.
- The customer defined a large page memory pool.
- There are enough pages in the large page memory pool to back the entire segment with large pages.

If all of these conditions are met, the application's data/heap segments will be backed with large pages. Otherwise, the application's data/heap segments will be backed with 4 KB pages.

In advisory mode, an application may have some of its heap segments backed by large pages and some of them backed by 4 KB pages. 4 KB pages are used to back segments when there are not enough large pages available to back the segment. Executable programs marked to use large pages use large pages in advisory mode.

In mandatory mode, the brk() or sbrk() system calls, which are internally called from the malloc() subroutine, will fail if the application requests a heap segment and there are not enough large pages to satisfy the request. Customers that use the mandatory mode must monitor the size of the large page pool and ensure it does not run out of large pages. Otherwise, their mandatory large page mode applications may fail.

### *Large page data/heap segments fully backed*

The POWER4 architecture requires all pages in a segment (256 MB) be backed with the same size physical pages. AIX backs the entire 256 MB segment with large pages when an application requests a large page heap segment. Even if only a few bytes are needed in the new heap segment, the entire 256 MB segment is backed. AIX does this to avoid terminating applications when they want to grow a heap segment (such as when using malloc() or sbrk()) and there are no large pages available to back the new space. This supports the *advisory* mode of large page usage. It also eliminates the need for installations to closely monitor the size of their large page physical memory pools.

### Using large pages to back shared memory segments

AIX uses the POWER4 large page architecture feature to provide large page backing for shared memory segments. Applications can request their shared memory segments be backed with large pages by specifying both the SHM_LGPAGE and SHM_PIN flags on the shmget() function.

The request to use large pages to back a shared segment is advisory. Large pages will back a shared memory segment under the same conditions as advisory mode large page data/heap usage. A shared segment is silently backed with 4 KB pages if large pages are not available.

The physical memory to back large page shared memory and large page data/heap segments comes from the large page physical memory pool. Customers must size their large page physical memory pool to contain enough large pages for both shared memory and data/heap large page usage.

## 3.6.3 Large page usage security capability

AIX provides a security mechanism to control the use of large page physical memory by non-root users. The large page physical memory pool is a fixed size, pinned memory system resource. The security mechanism prevents unauthorized users from using the large page pool and thus preventing its use by the intended users or applications.

Non-root users must have a CAP_BYPASS_RAC_VMM capability in order to use large pages. A system administrator can grant this capability to a user by using the **chuser** command. The following command grants the ability to use large pages to user lpuserid:

```
chuser capabilities=CAP_BYPASS_RAC_VMM,CAP_PROPAGATE lpuserid
```

Both large page data/heap and large page shared memory segments are controlled by this capability.

## 3.6.4 Configuring system to use large pages

The customer must configure the system to use large pages. The customer must specify the amount of physical memory to be used to back large pages. The default is to not have any memory allocated to the large page physical memory pool.

### AIX 5L Version 5.2

The **vmo** command is used to configure the size of the large page physical memory pool on AIX 5L Version 5.2.[22] The following command will allocate 256 pages x 16 MB = 4 GB to the large page physical memory pool:

```
vmo -r -o lgpg_regions=256 -o lgpg_size=1677216 -o v_pinshm=1
```

Where:

**-r** Updates the /etc/tunable/nextboot file so that the modified tunable values will take effect after the next system reboot.

**-o lgpg_regions=256** Specifies the reserved memory blocks for large pages.

**-o lgpg_size=1677216** Specifies the large page size in bytes. The allowable value is 16777216 (16 MB) on POWER4-based systems.

**-o v_pinshm=1** Allows pinning of shared memory segments.

You must run the **bosboot** command and reboot before the new size large page memory pool takes effect.

### AIX 5L Version 5.1

The **vmtune** command is used to configure the size of the large page physical memory pool on AIX 5L Version 5.1.[23] The following command will allocate 256 pages x 16 MB = 4 GB to the large page physical memory pool:

```
vmtune -g 16777216 –L 256
```

The -g option specifies the large page size in bytes. The allowable value is 16777216 (16 MB) on POWER4-based systems. The -L option is the number of the -g sized blocks that are allocated to the large page physical memory pool.

[22] The **vmo** command is included in the bos.perf.tune fileset.

[23] The **vmtune** command is located in the /usr/samples/kernel directory.

You must run the **bosboot** command and reboot before the new size large page memory pool takes effect.

If you want to use large pages for shared memory in your applications, the application source codes must be modified to use the SHM_PIN shmget() system call flag. The following **vmtune** command makes the necessary changes in the kernel to support the SHM_PIN flag:

```
vmtune –S 1
```

**Note:** The **vmtune** command must be called after every system boot. To place a permanent change into the system, insert the following lines into /etc/inittab:

```
vmtune -g 16777216 –L 256
vmtune –S 1
```

## Considerations when determining large page pool size

Here are some things to consider when determining the size of the large page physical memory pool:

- Memory allocated to the large page physical memory pool is not available to back 4 KB pages. Allocating too much physical memory to large pages will degrade system performance to the point of not having enough memory to back 4 KB pages. During system boot, AIX reserves enough physical memory for 4 KB pages to ensure that the system will boot. However, system failures may occur after booting if there is not enough physical memory to back 4 KB pages.
- The size of the large page physical memory pool is fixed at boot time and remains the same for the entire boot. A reboot is required to change the size of the large page memory pool.
- Large pages are only used for applications that explicitly request them. There is no need for a large page memory pool if your applications do not request them.
- Advisory mode large page applications will use large pages if there are large pages available. If not, advisory mode large page applications will use 4 KB pages. However, the inverse is not true. A 4 KB application will not use large pages if the system runs low on 4 KB pages.
- Mandatory mode large page applications will fail if the application requests a large page and one is not available.

### 3.6.5 Other system changes for large pages

The mprotect() function can not be used against a large page. It returns a -1 return code with an EINVAL errno if called to modify the protection attributes of a large page.

Some debug malloc tools use mprotect() to diagnose memory management problems. These tools will not work properly with large pages. Such applications must use 4 KB pages.

Multi-threaded applications may use large pages for their data/heap segments. However, when large pages are used, the libpthreads library does not place a protected *red zone* page at the bottom of a pthread's stack.

The sysconf(_SC_LARGE_PAGESIZE) function call will return the large page size on systems that have large pages.

The vmgetinfo() function returns information about large page pools size and other large page related information.

### 3.6.6 Large page usage considerations

Large page is a special purpose performance improvement feature. It is not recommended for general use. Large page usage provides performance value to a select set of applications. These are primarily long running memory access intensive applications that use large amounts of virtual memory.

Not all applications benefit by using large pages. Some applications can be severely degraded by the use of large pages. Applications that do a large number of fork()s (such as shell scripts) are especially prone to performance degradation when large pages are used. Tests have shown a tenfold increase in shell script execution time when the LDR_CNTRL environment specifies the large page usage variable. Consider marking specific executable files to use large pages rather than using the LDR_CNTRL environment variable. This limits large page usage to the specific applications that benefit from large page usage.

Consider the overall performance effect that large pages may have on your system. While some specific applications may benefit from large page use, the overall performance of your system may be degraded by large page usage due to having reduced the amount of 4 KB page storage available in the system. Consider using large pages when your system has sufficient physical memory such that reducing the number of 4 KB pages does not significantly impact overall system performance.

# 4

# Managing the memory heap

The term *heap*, or *memory heap*, generally means a free memory pool, from which a process can dynamically allocate chunks of memory. Although the management of the memory heap is the most basic programming task on any operating system environments, the mismanagement of the memory heap is a quite common mistake when developing applications using the C and C++ languages.

AIX provides a useful feature, called *malloc debug*, to diagnose these misuse of the memory heap without recompiling or modifying application source codes.

The first two sections in this chapter provide basic information about the malloc subsystem on AIX, and the third section explains how to use the malloc debug feature. The last section contains the use of library functions to transparently use process heap or shared memory segments to satisfy memory requests.

- Section 4.1, “Malloc subsystem” on page 166
- Section 4.2, “Memory allocators” on page 169
- Section 4.3, “Use of MALLOCDEBUG options” on page 182
- Section 4.4, “Heap management using MEMDBG” on page 199

## 4.1 Malloc subsystem

A process can dynamically allocate chunks of memory from the process heap by calling malloc subsystem subroutines. If the subroutine call succeeds, a process will be given the requested amount of virtual memory pages from the operating system, and those pages will be contiguously mapped in the process address space.

On AIX, the mapping addresses of newly allocated virtual pages are varied, depending on the user process model. For the detailed explanation about the available address range of the process heap, see 3.2, “The 32-bit user process model” on page 109 and 3.3, “The 64-bit user process model” on page 130.

The malloc subsystem performs the following fundamental memory operations:

**Allocation** Allocates the specified size of virtual memory.

**Deallocation** Deallocates (frees) the previous acquired memory space.

**Reallocation** Reallocates (adjusts) the size of the previous acquired memory space.

The malloc subsystem provides the following subroutines grouped by the function categories:

- Allocation
  - malloc()
  - calloc()
  - alloca()
  - valloc()
- Deallocation
  - free()
- Reallocation
  - realloc()
- Other purposes
  - mallopt()
  - mallinfo()
  - mallinfo_heap()
  - disclaim()

Once virtual memory pages are allocated by calling either malloc(), calloc(), or valloc() sub-routines, those routines internally call the system call sbrk() and increase the break value, which defines the maximum address of the process data segment (see Figure 3-8 on page 126).

If the process calls the free() sub-routine with the address of previously allocated memory, the corresponding virtual memory pages are marked *free* and placed in the list,[1] with which the malloc subsystem manages free virtual pages. The reference to free virtual pages should return a segmentation violation (SIGSEGV) to the caller.

If the process again requests to allocate memory through the malloc subsystem after that, the subsystem will try to allocate the requested memory size from the list. If sufficient chunks of memory are available in the list, the subsystem returns it. Otherwise, sbrk() is again internally called by the subsystem to increase the break value.

### 4.1.1 malloc(), calloc(), valloc(), and alloca()

Use the malloc() or calloc() subroutines to request only as much space as you actually need. Never request and then initialize a maximum-sized array when the actual situation uses only a fraction of it.

When a process touches a new page to initialize the array elements, the process effectively forces the VMM to steal a page of real memory from someone. Later, this results in a page fault when the process that owned that page tries to access it again. The difference between the malloc() and calloc() subroutines is not just in the interface.

Because the calloc() subroutine zeroes the allocated storage, it touches every page that is allocated, whereas the malloc() subroutine touches only the first page. If you use the calloc() subroutine to allocate a large area and then use only a small portion at the beginning, you place an unnecessary load on the system. Not only do the pages have to be initialized, but if their real-memory frames are reclaimed, the initialized and never-to-be-used pages must be written out to paging space. This situation wastes both I/O and paging space.

The valloc() subroutine, found in many BSD systems, is supported as a compatibility interface in the Berkeley compatibility library (libbsd.a). The valloc() subroutine calls the malloc() subroutine and automatically page-aligns requests that are greater than one page. The only difference between the valloc() subroutine in the libbsd.a library and the one in the standard C library (described above) is in the value returned when the size parameter is zero. The valloc() subroutine has the same effect as malloc(), except that the allocated memory is aligned to a multiple of the value returned by sysconf(_ SC_PAGESIZE).

[1] Technically, it is an internal data structure, not a linked-list.

The alloca() subroutine allocates the number of bytes of space specified by the Size parameter in the stack frame of the caller. This space is automatically freed when the subroutine that called the alloca subroutine returns to its caller.

If alloca() is used in the code and compiled with the C++ compiler, #pragma alloca would also have to be added before the usage of alloca() in the code. Alternatively, the -ma option would have to be used while compiling the code.

### 4.1.2 mallopt(), mallinfo, and mallinfo_heap()

The mallopt() and mallinfo() subroutines are provided for source-level compatibility with the System V malloc subroutines. Nothing done with the mallopt() subroutine affects how memory is allocated by the system, unless the M_MXFAST option is used.

The mallinfo() subroutine can be used to obtain information about the heap managed by the malloc() subroutine. Refer to the malloc.h file for details of the mallinfo structure.

**Note:** When MALLOCTYPE is set to buckets and the memory request is within the range of block sizes defined for the buckets, the memory request is serviced but the heap statistics that are reported by mallinfo() are not updated. See 4.2.3, “The default memory allocator with the malloc buckets extension” on page 173 for a detailed explanation about malloc buckets.

The mallinfo_heap() subroutine provides information about a specific heap if malloc multiheap is enabled (see 4.2.6, “Malloc multiheap” on page 181). The mallinfo_heap() subroutine returns a structure that details the properties and statistics of the heap specified by the user. Refer to the malloc.h file for details about the mallinfo_heap structure.

**Note:** The mallinfo_heap() subroutine should not be used with the 3.1 memory allocator explained in 4.2.1, “The 3.1 memory allocator” on page 171.

### 4.1.3 disclaim()

If a large structure is used early and then left untouched for the remainder of the process life, it should be released. It is not sufficient to use the free() subroutine to free the space that was allocated with the malloc() or calloc() subroutines on AIX, unless the MALLOCDISCLAIM environment variable is defined. The free() subroutine releases only the address range that the structure occupied from the process address space. To release the real memory and paging space, use the disclaim() subroutine to disclaim the space as well. The call to disclaim() should be before the call to free().

When the disclaim() routine is called with the specific range of memory address, it instructs VMM to relinquish physical in-memory pages, and disk blocks in the paging space. The resultant address-range is logically 0 again, just like it was when it was created, before any accesses were made to it.

This is a very useful optimization technique that some applications can use to get performance gains. For example, assume that an application has just finished using a buffer that is a small portion of a virtual memory address range, and it needs to manipulate the other data in the same address range. In this case, the application can call disclaim() specifying that memory range, which will be replenished with 0-filled pages on subsequent accesses. The call to disclaim() avoids the unnecessary page-in of *staled* data from the paging space.

### MALLOCDISCLAIM

If the MALLOCDISCLAIM environment variable is set as shown in the following example before a process start-up, all calls to the free() subroutine automatically instruct the malloc subsystem to call the disclaim() subroutine internally:

```
MALLOCDISCLAIM=true
```

This is useful in circumstances where a process has a high paging-space usage, but is not actually using the memory.

> **Note:** It is not supported to insert MALLOCDISCLAIM=true in /etc/environment.

For a detailed description of the malloc subsystem subroutines, please refer to the "System Memory Allocation Using the malloc subsystem" section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 4.2 Memory allocators

There are several internal mechanisms, called *memory allocators*, in the malloc subsystem on AIX. Each memory allocator implements a different memory allocation policy. The allocation policy refers to a set of data structures and algorithms to represent the heap and to implement allocation, deallocation, and reallocation.

The malloc subsystem on AIX supports the following memory allocators:

- 3.1 memory allocator
- Default memory allocator
- Default memory allocator with the malloc buckets extension
- Debug memory allocator

It is also supported to implement the user-defined memory allocator on AIX (see 4.2.5, “User-defined malloc replacement” on page 178).

The programming interface to the malloc subsystem is the same regardless of the selected memory allocator. The selection of memory allocators is made on a per-process basis by setting the MALLOCTYPE environment value when a program is executed. There are several environment variables that affect the behavior of the selected memory allocator.

Table 4-1 explains how to specify the MALLOCTYPE environment variable in order to select a memory allocator (related environment variables and sections are also shown).

*Table 4-1 Memory allocators and MALLOCTYPE*

| MALLOCTYPE= | Memory allocator | Related environment variables | Detailed explanation found in |
|---|---|---|---|
| 3.1 | 3.1 memory allocator | ▸ MALLOCDEBUG | Section 4.2.1, “The 3.1 memory allocator” on page 171 |
| (null)[1] | Default memory allocator[2] | ▸ MALLOCDEBUG<br>▸ MALLOCMULTIHEAP | Section 4.2.2, “The default memory allocator” on page 172 |
| buckets | Default memory allocator with the malloc buckets extension | ▸ MALLOCDEBUG<br>▸ MALLOCMULTIHEAP<br>▸ MALLOCBUCKETS | Section 4.2.3, “The default memory allocator with the malloc buckets extension” on page 173 |
| debug | Debug malloc allocator | ▸ MALLOCDEBUG | Section 4.2.4, “The debug malloc allocator” on page 176 |
| user:*archive_name*[2] | User-defined memory allocator[3] | N/A | Section 4.2.5, “User-defined malloc replacement” on page 178 |

1. The default memory allocator is selected by un-setting the MALLOCTYPE environment variable.
2. Where the *archive_name* specifies the library archive name that contains the user-defined malloc subsystem replacement subroutines.
3. A user-defined memory allocator can also be specified in the program code, as explained in 4.2.5, “User-defined malloc replacement” on page 178.

By default, the default memory allocator is always selected unless the MALLOCTYPE environment variable is explicitly set. For example, to specify the default memory allocator with the malloc extension, do the following on the Korn shell command line:

```
$ export MALLOCTYPE=buckets
```

To reset to the default memory allocator, do the following:

```
$ export MALLOCTYPE=
```

**Note:** These memory allocators are mutually exclusive.

### 4.2.1 The 3.1 memory allocator

The 3.1 memory allocator is mainly provided to support applications that were originally developed on AIX Version 3. Some of those applications may depend on the behavior of the memory allocator on AIX Version 3 and may misbehave when used with the other memory allocators. Use this memory allocator only when it is absolutely required by a specific set of applications.

To select the 3.1 memory allocator, set the MALLOCTYPE environment variable before executing programs on the Korn shell prompt:

```
$ export MALLOCTYPE=3.1
```

The 3.1 memory allocator maintains the process heap as a set of 28 hash buckets, each of which points to a linked list. Hashing is a method of transforming a search key into an address for the purpose of storing and retrieving items of data. The method is designed to minimize the average search time. A bucket is one or more fields in which the result of an operation is kept. Each linked list contains blocks of a particular size. The index into the hash buckets indicates the size of the blocks in the linked list. The size of the block is calculated using the following formula:

$$size = 2^{i + 4}$$

where i identifies the bucket. This means that the blocks in the list anchored by bucket zero are $2^{0 + 4}$ = 16 bytes long. Therefore, given that a prefix is 8 bytes in size, these blocks can satisfy requests for blocks between 0 and 8 bytes long.

To use the 3.1 memory allocator, keep the following points in mind:

- The 3.1 memory allocator supports the 32-bit user process environment only. It does not support the 64-bit user process environment.

- The 3.1 memory allocator does not support the following environment variables. If these are set, the 3.1 memory allocator simply ignores them:
  - MALLOCMULTIHEAP
  - MALLOCBUCKETS
- The algorithm can use as much as twice the amount of memory actually requested by the application. An extra page is required for buckets larger than 4096 bytes because objects of larger page are page-aligned. Because the prefix immediately precedes the block, an entire page is required solely for the prefix.

The 3.1 memory allocator supports the MALLOCDEBUG environment variable with the following keywords:

- log
- verbose
- trace

## 4.2.2 The default memory allocator

The default allocation policy maintains the free space in the heap as a binary tree, in which nodes are sorted vertically by length and horizontally by address. The data structure imposes no limitation on the number of block sizes supported by the tree, allowing a wide range of potential block sizes. Tree-reorganization techniques optimize access times for node location, insertion, and deletion, and also protect against fragmentation.

To select the default memory allocator, the MALLOCTYPE environment variable must be unset before executing programs. To confirm if it is unset, do following on the Korn shell prompt:

```
$ echo $MALLOCTYPE

$
```

The **echo** command should return a blank line, as shown in the above example.

The default memory allocator supports the following:

- Both the 32- and 64-bit user process environment
- The MALLOCMULTIHEAP environment variable
- The MALLOCDEBUG environment variable with the following keywords:
  - log
  - verbose
  - arena_check

  - trace

The default memory allocator does not support the MALLOCBUCKETS environment variable. If it is set, the allocator simply ignores it.

## 4.2.3 The default memory allocator with the malloc buckets extension

The default memory allocator with the malloc buckets extension, or simply *malloc buckets*, provides an optional buckets-based extension of the default allocator. It is intended to improve malloc subsystem performance for applications that issue large numbers of small allocation requests. When malloc buckets is enabled, allocation requests that fall within a predefined range of block sizes are processed by malloc buckets. All other requests are processed in the usual manner by the default allocator.

To select malloc buckets, set the MALLOCTYPE environment variable before executing programs on the Korn shell prompt:

```
$ export MALLOCTYPE=buckets
```

Additional user configuration can be done by explicitly setting the MALLOCBUCKETS environment value (see “MALLOCBUCKETS” on page 174).

### Bucket composition and sizing

A bucket consists of a block of memory that is subdivided into a predetermined number of smaller blocks of uniform size, each of which is an allocatable unit of memory. Each bucket is identified using a bucket number. The first bucket is bucket 0, the second bucket is bucket 1, the third bucket is bucket 2, and so on. The first bucket is the smallest, and each succeeding bucket is larger in size than the preceding bucket, using a formula described later in this section. A maximum of 128 buckets is available per heap.

The block size for each bucket is a multiple of a bucket-sizing factor. The bucket-sizing factor equals the block size of the first bucket. Each block in the second bucket is twice this size, each block in the third bucket is three times this size, and so on. Therefore, a given bucket's block size is determined as follows:

block size = (bucket number + 1) * bucket sizing factor

For example, a bucket-sizing factor of 16 would result in a block size of 16 bytes for the first bucket (bucket 0), 32 bytes for the second bucket (bucket 1), 48 bytes for the third bucket (bucket 2), and so on.

The bucket-sizing factor must be a multiple of 8 for 32-bit implementations and a multiple of 16 for 64-bit implementations in order to guarantee that addresses returned from malloc subsystem functions are properly aligned for all data types.

The bucket size for a given bucket is determined as follows:

```
bucket size = number of blocks per bucket *
             (malloc overhead + ((bucket number + 1) * bucket sizing factor))
```

The preceding formula can be used to determine the actual number of bytes required for each bucket. In this formula, malloc overhead refers to the size of an internal malloc construct that is required for each block in the bucket. This internal construct is 8 bytes long for 32-bit applications and 16 bytes long for 64-bit applications. It is not part of the allocatable space available to the user, but is part of the total size of each bucket.

The default memory allocator with the malloc buckets extension supports the following:

- Both the 32- and 64-bit user process environment
- The MALLOCMULTIHEAP environment variable
- The MALLOCDEBUG environment variable with the following keywords:
  - log
  - verbose
  - arena_check
  - trace

## MALLOCBUCKETS

The number of blocks per bucket, number of buckets, and bucket-sizing factor are all set with the MALLOCBUCKETS environment variable. The syntax of the variable is as follows (multiple keywords can be separated by a comma):

```
MALLOCBUCKETS=[[ number_of_buckets:N | bucket_sizing_factor:N |
blocks_per_bucket:N | bucket_statistics:[stdout|stderr|path_name]],...]
```

Where:

**number_of_buckets:N**

This option can be used to specify the number of buckets available per heap, where N is the number of buckets. The value specified for N will apply to all available heaps. The default value for number_of_buckets is 16. The minimum value allowed is 1. The maximum value allowed is 128.

**bucket_sizing_factor:N**

This option can be used to specify the bucket-sizing factor, where N is the bucket-sizing factor in bytes. The value specified for bucket_sizing_factor must be a multiple of 8 for 32-bit implementations and a multiple of 16 for

64-bit implementations.
The default value for bucket_sizing_factor is 32 for 32-bit implementations and 64 for 64-bit implementations.

**blocks_per_bucket:N**

This option can be used to specify the number of blocks initially contained in each bucket, where N is the number of blocks. This value is applied to all of the buckets. The value of N is also used to determine how many blocks to add when a bucket is automatically enlarged because all of its blocks have been allocated. The default value for blocks_per_bucket is 1024.

**bucket_statistics:[stdout|stderr|path_name]**

The bucket_statistics option will cause the malloc subsystem to output a statistical summary for malloc buckets upon normal termination of each process that calls the malloc subsystem while malloc buckets is enabled. This summary shows buckets-configuration information and the number of allocation requests processed for each bucket. If multiple heaps have been enabled by way of malloc multiheap, the number of allocation requests shown for each bucket will be the sum of all allocation requests processed for that bucket for all heaps.

The buckets statistical summary will be written to one of the following output destinations, as specified with the bucket_statistics option.

|  |  |
|---|---|
| **stdout** | Standard output |
| **stderr** | Standard error |
| **path_name** | A user-specified path name |

If a user-specified path name is provided, statistical output will be appended to the existing contents of the file (if any).

Standard output should not be used as the output destination for a process whose output is piped as input into another process.

The buckets_statistics option is disabled by default.

**Note:**

1. One additional allocation request will always be shown in the first bucket for the atexit() subroutine that prints the statistical summary.
2. For multi-threaded processes, additional allocation requests will be shown for some of the buckets due to malloc subsystem calls issued by the Pthread library.

If the MALLOCBUCKETS environment variable is not set, then the default values shown in Table 4-2 are assumed.

*Table 4-2 Default configuration values for malloc buckets*

| Configuration option | Default value (32-bit) | Default value (64-bit) |
|---|---|---|
| Number of buckets per heap | 16 | 16 |
| Bucket sizing factor | 32 bytes | 64 bytes |
| Allocation range | 1 to 512 bytes (inclusive) | 1 to 1024 bytes (inclusive) |
| Number of blocks initially contained in each bucket | 1024 | 1024 |
| Bucket statistical summary | disabled | disabled |

### 4.2.4 The debug malloc allocator

Debugging applications that are mismanaging memory allocated via the malloc subsystem can be difficult and tedious. Most often, the problem is that data is written past the end of an allocated buffer. Since this has no immediate consequence, problems do not become apparent until much later when the space that was overwritten (usually belonging to another allocation) is used and no longer contains the data originally stored there.

The AIX malloc subsystem includes the debug memory allocator[2] to allow users to identify memory overwrites, over-reads, duplicate frees, and reuse of freed memory allocated by malloc(). This memory allocator is sometimes referred to as debug malloc.

Memory problems detected by the debug memory allocator result in an abort() or a segmentation violation (SIGSEGV). In most cases, when an error is detected the application stops immediately and a core file is produced.

[2] The debug memory allocator has been supported on AIX since Version 4.3.3.

To select the debug memory allocator, set the MALLOCTYPE environment variable before executing programs on the Korn shell prompt:

```
$ export MALLOCTYPE=debug
```

Additional user configuration can be done by explicitly setting the MALLOCDEBUG environment value as follows:

```
MALLOCDEBUG=<options...>
```

where *options* is a comma-separated list of one or more predefined configuration options:

- align:N
- validate_ptrs
- postfree_checking
- allow_overreading
- override_signal_handling
- record_allocations
- report_allocations

More than one option can be specified (and in any order) as long as options are comma-separated, as shown in the following example:

```
MALLOCDEBUG=align:0,validate_ptrs,report_allocations
```

Each configuration option should only be specified once when setting MALLOCDEBUG. If a configuration option is specified more than once per setting, only the final instance will apply. For a further detailed explanation about these options, see 4.3.1, “MALLOCDEBUG with the debug memory allocator” on page 182.

**Note:** The debug memory allocator does not support the several MALLOCDEBUG options explained in 4.3.2, “MALLOCDEBUG with memory allocators other than debug” on page 190.

The debug memory allocator is not appropriate for full-time, constant, or system-wide use. Although it is designed for minimal performance impact upon the application being debugged, it may have significant negative impact upon overall system throughput if it is used widely throughout a system. In particular, setting MALLOCTYPE=debug in the /etc/environment file (to enable debug malloc for the entire system) is unsupported, and will likely cause significant system problems, such as excessive use of paging space. The debug memory allocator should only be used to debug single applications or small groups of applications at the same time.

In addition, please note that the debug memory allocator is not appropriate for use in some debugging situations. Because the debug memory allocator places each individual memory allocation on a separate page, programs that issue many small allocation requests will see their memory usage increase dramatically. These programs may encounter new failures as memory allocation requests are denied due to a lack of memory or paging space. These failures are not necessarily errors in the program being debugged, and they are not errors in the debug memory allocator.

### 4.2.5 User-defined malloc replacement

If none of memory allocators provided by AIX did not satisfy the specific memory management requirement for your applications, you can implement your own memory allocator, called a *user-defined* memory allocator, on AIX. This functionality is referred to as a user-defined malloc replacement.

Once the user-defined memory allocator has been prepared, all the calls to the malloc subsystem from user applications, such as malloc(), free(), and so on, are transparent; no modification is necessary.

In the C and C++ program languages linkage mechanism, it is always possible to override system subroutines (external linkage symbols) by user-defined functions (internal linkage symbols). For example, the sample source code shown in Example 4-1 defines a fake version of malloc(), which simply returns the address of global character array (as long as the program requires only one memory area up to size of BUFSIZ, this fake version of malloc() is sufficient for this program). If compiled and executed, the program prints the following output:

```
$ cc fake_malloc.c
$ a.out
p = 1234567890
$ echo $?
0
```

*Example 4-1 fake_malloc.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

char global_buf[BUFSIZ];

/* fake malloc() routine. */
void *malloc(size_t sz)
{
    /* return the address of global character array. */
    return((void *)global_buf);
}
```

```
int
main(int argc, char *argv[])
{
    char *p;

    if ((p = (char *)malloc(BUFSIZ)) == (char *)NULL) {
        perror("malloc() failed.\n");
        exit(1);
    }
    /* write some data to the address pointed by p. */
    strcpy(p, "1234567890");
    /* print the written data. */
    printf("p = %s\n", p);

    exit(0);
}
```

It must be clear that this overriding of external symbol names is different from the user-defined malloc replacement. If programmed appropriately, the user-defined memory allocator will be used for all the malloc subsystem calls, not only from user codes, but also from functions within shared libraries (many subroutines within shared libraries provided by AIX call the malloc subsystem internally).

To implement user-defined memory allocators, the following requirements must be satisfied:

- A user defined memory memory allocator must provide both the 32- and 64-bit object modules. Both modules must be placed in a library archive file and the 32-bit shared object must be named mem32.o and the 64-bit shared object must be named mem64.o.
- A user-defined memory memory allocator should be thread-safe. This is the programmer's responsibility; there are no automatic checks to verify it.
- A user-defined memory memory allocator must implement the functions with those names that start with double underscore characters, as shown in Table 4-3.

*Table 4-3 User-defined replacement subroutines*

| Function name, proto-type declaration | Description |
|---|---|
| `void *__malloc__(size_t)` | A user-defined replacement of malloc(). |
| `void __free__(void *)` | A user-defined replacement of free(). |
| `void *__realloc__(void *, size_t)` | A user-defined replacement of realloc(). |
| `void *__calloc__(size_t, size_t)` | A user-defined replacement of calloc(). |

| Function name, proto-type declaration | Description |
|---|---|
| `int __mallopt__(int, int)` | A user-defined replacement of mallopt(). |
| `struct mallinfo __mallinfo__()` | A user-defined replacement of mallinfo(). |
| `void __malloc_once__()` | Will be called once before any other user-defined malloc entry point is called. |
| `void __malloc_init__(void)` | Called by the Pthread initialization routine to initialize the user-defined memory allocator in the multi-threaded programming environment.[1] |
| `void __malloc_prefork_lock__(void)` | Called by Pthread when the fork() subroutine is called.[1] |
| `void __malloc_postfork_unlock__(void)` | Called by Pthread when the fork() subroutine is called.[1] |
| 1. These functions are mandatory in the multi-threaded environment. | |

- The shared objects (mem32.o for 32-bit and mem64.o for 64-bit) must export the following symbols:
  - `__malloc__`
  - `__free__`
  - `__realloc__`
  - `__calloc__`
  - `__mallinfo__`
  - `__mallopt__`
  - `__malloc_init__`
  - `__malloc_prefork_lock__`
  - `__malloc_postfork_unlock__`

  The shared objects can optionally export `__malloc_once__`.

To select the user-defined memory allocator, set the MALLOCTYPE environment variable before executing programs on the Korn shell prompt:

```
$ export MALLOCTYPE=user:archive_name
```

Where the archive_name specifies the library archive name that contains the user-defined malloc subsystem replacement subroutines.

A user-defined memory allocator can also be specified in the program code by declaring the global symbol _malloc_user_defined_name, as shown in the following example:

```
char *_malloc_user_defined_name=”archive_name”;
```

If both the MALLOCTYPE environment variable and the global symbol are used to specify the archive_name, the name specified by MALLOCTYPE will override the one specified by the global symbol.

**Note:** User-defined memory allocators written in C++ are not supported, because the C++ standard library libC.a depends on the standard malloc subsystem provided by the C standard library libc.a.

### 4.2.6 Malloc multiheap

Historically, the malloc subsystem was designed for the non-threaded programming environment. Therefore, there was an single memory pool, or memory heap, per-process basis. After the evolution of the multi-threaded programming environment, it was realized that the single heap does not satisfy the memory allocation requests from multi-threaded applications, because a single malloc() call from a user thread can lock the entire malloc subsystem and the other user threads would be starving; thus, malloc() calls within a process would be serialized.

By providing multiple heaps, malloc multiheap efficiently supports the memory allocation requests from multi-threaded applications; thus, the malloc multiheap has a finer locking mechanism than the single heap malloc subsystem. The potential performance enhancement is particularly likely for multi-threaded C++ programs, because these may make use of the malloc subsystem whenever a constructor or destructor is called.

Beginning with Version 5.1, the malloc multiheap is enabled by default in the malloc subsystem on AIX. Therefore, it does not require any user settings, though it can be tuned using the MALLOCMULTIHEAP environment variable. To set the MALLOCMULTIHEAP environment variable, use the following syntax:

```
MALLOCMULTIHEAP=[[heaps:N],[considersize],...]
```

Where:

**heaps:N** The heaps:N option can be used to change the maximum number of heaps to any value from 1 through 32, where N is the number of heaps. If n is set to a value outside the given range, the default value of 32 is used.

**considersize** By default, malloc multiheap selects the next available heap. If the considersize option is specified, malloc multiheap will use an alternate heap-selection algorithm that tries to select an available heap that has enough free space to handle the request. This may minimize the working set size of the process by reducing the number of sbrk subroutine calls. However, because of the additional processing required, the considersize heap-selection algorithm is somewhat slower than the default heap selection algorithm.

Multiple keywords are separated by a comma.

> **Note:** The malloc multiheap is enabled internally on AIX beginning with Version 5.1 with the default value of 32. Therefore, **`echo $MALLOCMULTIHEAP`** prints a blank line by default.

# 4.3 Use of MALLOCDEBUG options

This section explains the use of the MALLOCDEBUG environment variable by providing the following sections:

- Section 4.3.1, “MALLOCDEBUG with the debug memory allocator” on page 182
- Section 4.3.2, “MALLOCDEBUG with memory allocators other than debug” on page 190

## 4.3.1 MALLOCDEBUG with the debug memory allocator

The debug memory allocator supports the following options specified by the MALLOCDEBUG environment variable:

- “align:N” on page 183
- “validate_ptrs” on page 185
- “postfree_checking” on page 186
- “allow_overreading” on page 188
- “override_signal_handling” on page 188
- “record_allocations” on page 189
- “report_allocations” on page 189

**Note:** The debug memory allocator does not support the MALLOCDEBUG options explained in 4.3.2, “MALLOCDEBUG with memory allocators other than debug” on page 190.

## align:N

By default, malloc() returns a pointer aligned on a 2-word[3] boundary in the 32-bit and 4-word boundary in the 64-bit user process environment. The align:N option can be used to change the default alignment, where N is the number of bytes to be aligned and can be any power of 2 between 0 and 4096 inclusive (for example, 0, 1, 2, 4, …). The values 0 and 1 are treated as the same, that is, there is no alignment, so any memory accesses outside the allocated area will cause an abort().

The following formula can be used to calculate how many bytes of over-reads or over-writes the debug memory allocator will allow for a given allocation request when MALLOCDEBUG=align:N and size is the number of bytes to be allocated:

((((size / N) + 1) * N) - size) % N

The example program shown in Example 4-2 demonstrates the effect of the align:N option. This program allocates a character string array size of 10 bytes, then prompts user to input some data, which will be stored in the previously allocated array.

*Example 4-2 debug_malloc_align.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define MAX_SIZE 10
#define EXIT_CODE -1

int main(int argc, char *argv[])
{
    char *ptr = (char *)NULL;
    char str[BUFSIZ];

    if ((ptr = (char *)malloc(MAX_SIZE)) == (char *)NULL) {
        perror("malloc() failed.\n");
        exit(EXIT_CODE);
    }
    printf("Enter the value for ptr: ");
    gets(str);
    strcpy(ptr, str);
```

[3] The term word means an implementation dependent unit of memory. On AIX, a word is 32 bits (4 bytes).

```
    printf("ptr points at : %p\n", ptr);
    printf("The value stored in ptr is : %s\n", ptr);
    free(ptr);
}
```

Before executing the program, set the following environment variables to enable the align:N option from the command prompt:

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=align:2
```

Applying the above mentioned formula for align:2, the number of bytes of over-read or over-write allowed is:

**align:2** ((((10/2) + 1) * 2) - 10) % 2 = 0

Therefore, the debug memory allocator will not allow any over-reads or over-writes. If executed, the program would print the following output (the character string `12345678901` is user input):

```
$ a.out
Enter the value for ptr: 12345678901
Segmentation fault(coredump)
```

The program is terminated by a segmentation fault, because the length of ptr is 12 (11 printable characters plus the NULL-termination character ‘0x0’).

If align:4 or align:8 is specified, the allowed over-read or over-write memory byte region size would be 2 or 6 bytes, as shown in the following calculation:

align:4 ((((10/4) + 1) * 4) - 10) % 4 = 2

align:8 ((((10/8) + 1) * 8) - 10) % 8 = 6

For example, if align:4 is specified, the same program prints the following output (no segmentation fault occurs):

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=align:4
$ a.out
Enter the value for ptr: 12345678901
ptr points at : 20001ff4
The value stored in ptr is : 12345678901
```

The following points should be considered while setting the align:N option:

- For allocated space to be word aligned, specify align:N with a value of 4.
- If the align:N option is not explicitly set, it defaults to 8.

## validate_ptrs

By default, free() does not validate its input pointer to ensure that it actually references memory previously allocated by malloc(). If the parameter passed to free() is a NULL value, free() will return to the caller without taking any action. If the parameter is invalid, the results will be undefined. A core dump may or may not occur in this case, depending upon the value of the invalid parameter. Specifying the validate_ptrs option will cause free() to perform extensive validation on its input parameter. If the parameter is found to be invalid (that is, it does not reference memory previously allocated by a call to malloc() or realloc()), debug malloc will print an error message stating why it is invalid. The abort() function is then called to terminate the process and produce a core file.

The example program shown in Example 4-3 demonstrates the effect of the validate_ptrs option. This is slightly modified from Example 4-2 on page 183 and calls the free() subroutine twice at the end of the program. Though the second call of free() is an error, it does not abort the execution of the program in normal situations.

*Example 4-3 debug_malloc_vptr.c*

```
/*This program is a slightly modified version of debug_mallo_align.c. We are
just trying to free the ptr memory even after it is freed by the first free()
call*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define MAX_SIZE 10
#define EXIT_CODE -1

int main(int argc, char*argv[])
{
    char *ptr = NULL;
    char str[BUFSIZ];
    if ((ptr = (char *)malloc(MAX_SIZE)) == (char *)NULL) {
        perror("malloc() failed.\n");
        exit(EXIT_CODE);
    }
    printf("Enter the value for ptr: ");
    gets(str);
    strcpy(ptr, str);
    printf("ptr points at : %p\n", ptr);
    printf("The value stored in ptr is : %s\n", ptr);
    free(ptr);
    free(ptr); /* This is invalid call. ptr is already freed. */
}
```

Before executing the program, set the following environment variables to enable the validate_ptr option from the command prompt:

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=validate_ptrs
```

If executed, the program would print the following output (the character string 1234567890 is user input):

```
$ a.out
Enter the value for ptr: 1234567890
ptr points at : 20001ff0
The value stored in ptr is : 1234567890
Debug Malloc: Buffer (0x20001ff0) has already been free'd.
IOT/Abort trap(coredump)
```

As highlighted in the output, the debug memory allocator has detected the invalid second free() call.

## postfree_checking

By default, the malloc subsystem allows the calling program to access memory that has previously been freed. This should result in an error in the calling program. If the postfree_checking option is specified, any attempt to access memory after it is freed will cause the debug memory allocator to report the error and abort the program; then a core file will be produced.

> **Note:** Specifying the postfree_checking option automatically enables the validate_ptrs option.

If the same program shown in Example 4-3 on page 185 is executed with the postfree_checking by setting the following environment variables:

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=postfree_checking
```

then it will result in a segmentation fault, though the reason for that is not clearly reported in the following output:

```
$ a.out
Enter the value for ptr: 1234567890
ptr points at : 20001ff0
The value stored in ptr is : 1234567890
Segmentation fault(coredump)
```

The postfree_checking option identifies the access (if any) to the memory after it is freed, but the validate_ptrs option does not. The example program shown Example 4-4 on page 187 illustrates the difference between the postfree_checking and validate_ptrs options. This program is trying to access the ptr memory after it is freed by the free() call.

*Example 4-4 debug_malloc_pfc.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errono.h>
#define MAX_SIZE 10
#define EXIT_CODE -1

int main(int argc, char *argv[])
{
    char *ptr = (char *)NULL;
    char str[BUFSIZ];
    if ((ptr = (char *)malloc(MAX_SIZE)) == (char *)NULL) {
        perror("malloc() failed.\n");
        exit(EXIT_CODE);
    }
    printf("Enter the value for ptr: ");
    gets(str);
    strcpy(ptr, str);
    printf("ptr points at : %p\n", ptr);
    printf("The value stored in ptr is : %s\n", ptr);
    free(ptr);
    /* Wrong. trying to access memory after it is freed. */
    printf("The value stored in ptr is : %s\n", ptr);
}
```

Before executing the program, set the following environment variables to enable the validate_ptr option from the command prompt:

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=validate_ptrs
```

If executed, the program would print the following output (the highlighted character string `12345` is a user input):

```
$ a.out
Enter the value for ptr: 12345
ptr points at : 20001ff0
The value stored in ptr is : 12345
```

Apparently the debug memory allocator with the validate_ptr option did not detect the error. The reason is that the validate_ptrs option checks for the validity of ptr only when it is passed to a free() function call.

If the same program is executed with the validate_ptrs option, the program would print the following output (the highlighted character string `12345` is a user input):

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=postfree_checking
$ a.out
Enter the value for ptr: 12345
ptr points at : 20001ff0
The value stored in ptr is : 12345
Segmentation fault(coredump)
```

With the postfree_checking option set, the debug memory allocator identifies the erroneous access to the memory after it is freed.

### allow_overreading

By default, the debug memory allocator will respond with a segmentation violation and if the program attempts to read past the end of allocated memory. The allow_overreading option instructs the debug memory allocator to ignore *over-reads* of this nature so that other types of errors, which may be considered more serious, can be detected first.

### override_signal_handling

The debug memory allocator reports errors in one of two ways:

- Memory access errors (such as trying to read or write past the end of allocated memory) will cause a segmentation violation (SIGSEGV), resulting in a core dump.
- For other types of errors (such as trying to free space that was already freed), the debug memory allocator will print an error message, then call abort(), which will send a SIGIOT signal to terminate the current process.

If the calling program is blocking or catching the SIGSEGV and/or the SIGIOT signals, the debug memory allocator will be prevented from reporting errors. The override_signal_handling option provides a means of addressing this situation without recording and rebuilding the application.

If the override_signal_handling option is specified, the debug memory allocator will perform the following actions upon each call to one of the memory allocation routines (malloc(), free(), realloc(), or calloc()):

1. Disables any existing signal handlers set up by the application.
2. Sets the action for both SIGIOT and SIGSEGV to the default (SIG_DFL).
3. Unblocks both SIGIOT and SIGSEGV.

When using the override_signal_handling option, keep in mind the following:

- If an application signal handler modifies the action for SIGSEGV between memory allocation routine calls and then attempts an invalid memory access, the debug memory allocator will be unable to report the error (the application will not exit and no core file will be produced).
- The override_signal_handling option may be ineffective in a multi-threaded application environment because the debug memory allocator uses sigprocmask() and many multi-threaded processes use pthread_sigmask().
- If a user thread calls sigwait() without including SIGSEGV and SIGIOT in the signal set and the debug memory allocator subsequently detects an error, the user thread will hang because the allocator can only generate SIGSEGV or SIGIOT.

### record_allocations

The record_allocations option instructs the debug memory allocator to create an allocation record for each malloc() request. Each record contains the following information:

- The original address returned to the caller from malloc().
- Up to six function trace backs starting from the call to malloc().

Each allocation record will be retained until the memory associated with it is freed.

### report_allocations

The report_allocations option instructs the debug memory allocator to report all active allocation records at application exit. An active allocation record will be listed for any memory allocation that was not freed prior to application exit.

> **Note:** Specifying the report_allocations option automatically enables the record_allocations option.

To demonstrate how the report_allocations works, we have slightly modified the example program shown in Example 4-3 on page 185 by removing two free() lines (highlighted in the example).

Before executing the program, set the following environment variables to enable the validate_ptr option from the command prompt:

```
$ export MALLOCTYPE=debug
$ export MALLOCDEBUG=report_allocations
```

If executed, the program would print the following output (the highlighted character string 12345678901 is a user input):

```
$ a.out
Enter the value for ptr: 12345678901
ptr points at : 20003ff0
The value stored in ptr is : 12345678901
Current allocation report:
    Allocation #1: 0x20003FF0
        Allocation size: 0xA
        Allocation traceback:
        0x100001B4  __start
        0x1000035C  main
        0xD01D7104  malloc

    Allocation #2: 0x20001FF0
        Allocation size: 0x10
        Allocation traceback:
        0x1000035C  main
        0xD01D7104  malloc
        0xD01D6C28  init_malloc
        0xD01D6120  check_environment
        0xD022BA10  malloc_debug_start
        0xD01E42D4  atexit

Total allocations: 2.
```

The output contains the allocation report (Allocation #1) for the un-freed memory in the program (the memory address printed for Allocation #1 is same as the address location of ptr: 0x20003FF0). Because this memory has not been freed, the debug memory allocator detects it and then prints the allocation report.

**Note:** One allocation record will always be listed for the atexit() handler that prints the allocation records, as shown in the previous output (Allocation #2).

### 4.3.2 MALLOCDEBUG with memory allocators other than debug

The options shown in Table 4-4 on page 191, which can be specified by the MALLOCDEBUG environment value, are supported by 3.1, default, and default with malloc buckets extension memory allocators:

*Table 4-4 MALLOCDEBUG options*

| Option | Feature name | Related section |
|---|---|---|
| verbose | Malloc report | “verbose” on page 191 |
| arena[a] | | “arena” on page 192 |
| trace | Malloc trace | “trace” on page 193 |
| log | Malloc log | “log” on page 196 |

[a] The arena option is not supported by the 3.1 memory allocator.

Multiple options can be specified by separating them using a comma as follows:

```
MALLOCDEBUG=option1,option2,...
```

## verbose

The verbose option instructs memory allocators that information on errors that occurred in the malloc subsystem will be reported and actions can be performed if specified.

The verbose option is not enabled by default, but can be enabled and configured prior to process startup by setting the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=verbose
```

All errors caught in the malloc subsystem are output to standard error, along with detailed information.

The verbose option allows the user to provide a function that the malloc subsystem will call when it encounters an error. Before returning, the malloc subsystem calls the user-provided function, if it is specified.

A global function pointer is available for use by the user. In the code, the following function pointer should be set to the user’s function:

```
extern void (*malloc_err_function)(int, ...)
```

The following user-defined function must be implemented:

```
void malloc_err_function(int, ...)
```

For example, to use the user-defined function abort_sub, the following code must be inserted into the user’s application:

```
malloc_err_function = &abort_sub;
```

A sample program shown in Example 4-5 illustrates the use of the verbose option. When executed, the program prints the following error message:[4]

```
$ MALLOCDEBUG=verbose a.out
Malloc Report: Corruption in the Yorktown Malloc arena has been detected.
IOT/Abort trap(coredump)
```

The default memory allocator with the verbose option has detected the corruption in this output.

*Example 4-5 debug_malloc_verbose.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    char *ptr;
    char buf[BUFSIZ];

    if ((ptr = (char *)malloc(1200)) == (char *)NULL) {
        sprintf(buf, "malloc() failed at %d in %s with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
    }
    free(ptr);
    memset(ptr - 8, (char)-1, 40);
    free(ptr - 4096);

    exit(0);
}
```

### arena

The arena option instructs the malloc subsystem to check the structures that contain the free blocks before every allocation request is processed. This option will ensure that the arena is not corrupted. Also, the arena will also be checked when an error occurs.

The checkarena option checks for NULL pointers in the free tree or pointers that do not fall within a certain range. If an invalid pointer is encountered during the descent of the tree, the program might perform a core dump depending on the value of the invalid address.

[4] *Yorktown* is an internal name used in the AIX development to refer to the default memory allocator.

The arena option is not enabled by default, but can be enabled and configured prior to process startup by setting the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=checkarena
```

A sample program shown in Example 4-6 illustrates the use of the arena option. When executed, the program prints the following error message:

```
$ MALLOCDEBUG=verbose,arena a.out
Malloc Report: The address passed to free, 0x1ffff5f8, is outside the valid
range of addresses allocated by malloc (errno = 0).
IOT/Abort trap(coredump)
```

*Example 4-6   debug_malloc_arena.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    char *ptr;
    char buf[BUFSIZ];

    if ((ptr = (char *)malloc(16)) == (char *)NULL) {
        sprintf(buf, "malloc() failed at %d in %s with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
        exit(1);
    }
    free(ptr-4096);

    exit(0);
}
```

## trace

The trace option instructs the malloc subsystem to use the trace facility. Traces of the malloc(), realloc(), and free() subroutines are recorded for use in problem determination and performance analysis.

The trace option is not enabled by default, but can be enabled and configured prior to process startup by setting the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=trace
```

The trace option supports the following trace hook IDs:

- HKWD_LIB_MALL_COMMON (hook ID: 60a)

  When tracing is enabled for HKWD_LIB_MALL_COMMON, the input parameters, as well as return values for each call to malloc(), realloc(), and free() subroutines, are recorded in the trace subsystem. In addition to providing trace information about the malloc subsystem, the trace option also performs checks of its internal data structures. If these structures have been corrupted, these checks will likely detect the corruption and provide temporal data, which is useful in problem determination.

- HKWD_LIB_MALL_INTERNAL (hook ID: 60b)

  When tracing is enabled for HKWD_LIB_MALL_INTERNAL and corruption is detected, information about the internal data structures are logged through the trace subsystem.

To use the trace option, do the following:

1. Start the trace subsystem before executing the target application. If you use the command line interface, type the following command as the root user:

   ```
   # trace -j’60a,60b’ -a
   ```

   If you use SMIT, run **`smit -C trace`** and select **START Trace**, and then select the hook keywords 60a and 60b in the high-lighted field in Example 4-7.

*Example 4-7 Start trace*

```
                               START Trace

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                      [Entry Fields]
  EVENT GROUPS to trace                             []                 +
  ADDITIONAL event IDs to trace                     []                 +
  Event Groups to EXCLUDE from trace                []                 +
  Event IDs to EXCLUDE from trace                   []                 +
  Trace MODE                                        [alternate]        +
  STOP when log file full?                          [no]               +
  LOG FILE                                          [/var/adm/ras/trcfile]
  SAVE PREVIOUS log file?                           [no]               +
  Omit PS/NM/LOCK HEADER to log file?               [yes]              +
  Omit DATE-SYSTEM HEADER to log file?              [no]               +
  Run in INTERACTIVE mode?                          [no]               +
  Trace BUFFER SIZE in bytes                        [131072]            #
  LOG FILE SIZE in bytes                            [1310720]           #
  Buffer Allocation                                 [automatic]        +

F1=Help             F2=Refresh          F3=Cancel           F4=List
```

```
F5=Reset            F6=Command          F7=Edit             F8=Image
F9=Shell            F10=Exit            Enter=Do
```

2. Run the target application. You should remember the process ID of the application.
3. Stop the trace subsystem. If you use the command line interface, type the following command as the root user:

   ```
   # trcstop
   ```

   If you use SMIT, run **`smit -C trace`** and select **STOP Trace**.
4. Generate a trace report. If you use the command line, type the following command as the root user:

   ```
   # trcrpt -O exec=y -O pid=y -O tid=y -O svc=y -O timestamp=1
   ```

   If you use SMIT, run **`smit -C trace`** and select **Generate a Trace Report**.

*Example 4-8 Trace output of the trace option*

```
Fri Feb 28 14:29:20 2003
System: AIX 5.2 Node: murumuru
Machine: 000C91AD4C00
Internet Address: 0903046A 9.3.4.106
The system contains 2 cpus, of which 2 were traced.
Buffering: Kernel Heap
This is from a 32-bit kernel.
Tracing only these hooks, 60a,60b


/usr/bin/trace -j60a 60b -a


ID  PROCESS NAME   I SYSTEM CALL     ELAPSED   APPL    SYSCALL KERNEL  INTERRUPT

001 --1-                            0.000000                   TRACE ON channel 0
                                                               Fri Feb 28 14:29:20 2003
60A --1-                            5.561069   HKWD_LIBC_MALL_COMMON
                                               function=malloc() [Default Allocator]
                                               size=000A
                                               returnptr=20000728
60A --1-                            5.561261   HKWD_LIBC_MALL_COMMON
                                               function=free() [Default Allocator]
                                               inptr=20000728
60A --1-                           10.131345   HKWD_LIBC_MALL_COMMON
                                               function=malloc() [Default Allocator]
                                               size=0290
                                               returnptr=20005878
```

## log

The log option instructs the malloc subsystem to record information on the number of active allocations of a given size and stack trace back of a user program. This data can be used in problem determination and performance analysis, if the user program is modified accordingly.

### *Data recorded with the log option*

If the log option is enabled, the following data is recorded for each malloc or realloc subroutine invocation:

- The size of the allocation.
- The stack trace back of the invocation. The depth of the trace back that is recorded is a configurable option.
- The number of currently active allocations that match the size and stack trace back.

The data is stored into the following global structure:[5]

```
struct malloc_log *malloc_log_table;
#ifndef MALLOC_LOG_STACKDEPTH
#define MALLOC_LOG_STACKDEPTH 4
#endif
    struct malloc_log {
    size_t size;
    size_t cnt;
    uintptr_t callers [MALLOC_LOG_STACKDEPTH];
}
size_t malloc_log_size;
```

The size of the malloc_log structure can change. If the default call-stack depth is greater than 4, the structure will have a larger size. The current size of the malloc_log structure is stored in the globally exported malloc_log_size variable. A user can define the MALLOC_LOG_STACKDEPTH macro to the stack depth that was configured at process start time.

The malloc_log_table can be accessed in the following ways:

- Using the get_malloc_log() sub-routine as follows:

  ```
  #include <malloc.h>
  size_t get_malloc_log (void *addr,void *buf,size_t bufsize);
  ```

  This function copies the data from malloc_log_table into the provided buffer. The data can then be accessed without modifying the malloc_log_table. The data represents a snapshot of the malloc log data for that moment of time.

[5] This is included in the /usr/include/malloc.h header file.

- Using the get_malloc_log_live() sub-routine as follows:

  ```
  #include <malloc.h>
  struct malloc_log *get_malloc_log_live (void *addr);
  ```

  The advantage of this method is that no data needs to be copied, therefore performance suffers less. Disadvantages of this method are that the data referenced is volatile and the data may not encompass the entire malloc subsystem, depending on which malloc algorithm is being used.

To clear all existing data from the malloc log tables, use the reset_malloc_log() subroutine as follows:

```
#include malloc.h
void reset_malloc_log(void *addr);
```

The sample program shown in Example 4-9 on page 198 demonstrates the use of the get_malloc_log_live() sub-routine. After it is compiled, the program would print the following output, if the log option is enabled:

```
$ cc get_malloc_live.c
$ export MALLOCTYPE=
$ export MALLOCDEBUG=log
$ a.out
i is 1217.
The size of the allocations is 8.
The number of matching allocations is 1.
i is 1241.
The size of the allocations is 16.
The number of matching allocations is 1.
i is 3293.
The size of the allocations is 8.
The number of matching allocations is 1.
```

**Note:** Do not set the MALLOCTYPE=log environment variable before invoking the compiler, because it does not support this environment variable. If set, it prints the following message:

```
cc: 1501-230 Internal compiler error; please contact your Service
Representative.
```

The program has a for loop to dump the content of log_ptr, if there are referenced slots in the data structure referenced by log_ptr.

*Example 4-9 get_malloc_log_live.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    char *ptr1 = (char *)NULL, *ptr2 = (char *)NULL, *ptr3 = (char *)NULL;
    int i;

    struct malloc_log *log_ptr;

    ptr1 = (char *)malloc(8);
    ptr2 = (char *)malloc(16);
    ptr3 = (char *)malloc(8);

    log_ptr = get_malloc_log_live(ptr1);

    for (i = 0; i < DEFAULT_RECS_PER_HEAP; i++) {
        if (log_ptr->cnt > 0) {
            printf("i is %d.\n", i);
            printf("The size of the allocations is %d.\n", log_ptr->size);
            printf("The number of matching allocations is %d.\n"
                , log_ptr->cnt);
        }
        log_ptr++;
    }
    exit(0);
}
```

### *Enabling malloc log*

The log option is not enabled by default, but can be enabled and configured prior to process startup by setting the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=log
```

To enable the trace option with user-specified configuration options, set the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=log:records_per_heap:stack_depth
```

**Note:** The records_per_heap and stack_depth parameters must be specified in order. Leaving a value blank will result in setting that parameter to the default value.

The predefined MALLOCDEBUG configuration options include the following:

**records_per_heap** Used to specify the number of malloc log records that are stored for each heap. This parameter affects the amount of memory that is used by malloc log. The default value is 4096, the maximum value is 65535.

**stack_depth** Used to specify the depth of the function-call stack that is recorded for each allocation. This parameter affects the amount of memory used by malloc log, as well as the size of the malloc_log structure. The default value is 4, the maximum is 32.

### *Limitations*

The performance of all programs can degrade when the trace option is enabled, due to the cost of storing data to memory. Memory usage will also increase.

## 4.4 Heap management using MEMDBG

The memdbg package provided by IBM C and C++ compiler products provides several useful features, including the *user-created heap* (explained in this section), to deal with managing the process heap. The user-created heap can be used in your application source codes in order to replace the malloc subsystem calls, such as malloc() and calloc(). It could be seen as a wrapper to the interface to the malloc subsystem and shmat services.

If applications are modified to use the user-created heap, then they can use not only the default process heap, but also the shared memory segments for memory allocation requests.

The user-defined memory allocation functions are provided by the libhu.a library and defined in the header file umalloc.h. You can either handle user-created heaps or the default process heap with the following functions:

- _ucalloc()
- _umalloc()
- _uheapmin()

As shown in Example 4-10 on page 200, the following filesets must be installed on the AIX 5L Version 5.2 system in order to use these functions:

- memdbg.adt
- memdbg.aix50.adt
- memdbg.msg.en_US

*Example 4-10 /usr/lib/libhm.a and memdgb.* package*

```
# ls -l /usr/lib/libhu.a
lrwxrwxrwx   1 bin      bin              20 Jan 29 19:47 /usr/lib/libhu.a@ ->
/usr/vac/lib/libhu.a*
# lslpp -w /usr/lib/libhu.a
  File                                          Fileset               Type
  ----------------------------------------------------------------------------
  /usr/lib/libhu.a                              memdbg.adt            Symlink
# lslpp -w /usr/vac/lib/libhu.a
  File                                          Fileset               Type
  ----------------------------------------------------------------------------
  /usr/vac/lib/libhu.a                          memdbg.aix50.adt      Symlink
# lslpp -L memdbg.*
  Fileset                      Level  State  Type  Description (Uninstaller)
  ----------------------------------------------------------------------------
  memdbg.adt                 4.4.3.0    C     F    User Heap/Memory Debug Toolkit
  memdbg.aix50.adt           4.4.3.0    C     F    User Heap/Memory Debug Toolkit
                                                   for AIX 5.0
  memdbg.msg.en_US           4.4.3.0    C     F    User Heap/Memory Debug
                                                   Messages--U.S. English
```

**Note:** Although these sub-routines are convenient to use, the following points must be understood before using these sub-routines:

- Application source codes must be modified to call these routines.
- These subroutines are not considered as a user-created malloc replacement (see 4.2.5, “User-defined malloc replacement” on page 178). Therefore, the user processes still require the default process heap for the memory allocation requests made by functions in shared libraries, even if you have specified to use shared memory segments for the user-created heap.

More further information about these subroutines, please refer to Chapter 4, “Using Memory Heaps”, in *VisualAge C++ Professional for AIX Programming Tasks and Library Reference*, SC09-4963.

## 4.4.1 How to handle a user-created heap

To use a user-created heap, there are some basic tasks, as summarized in the following:

1. Allocate memory chunks to be used for the user-created heap, depending on the type of memory source:
   - Call the system malloc() sub-routine to request memory from the process heap.

  - Call the shmget() and shmat() sub-routines to request from the shared memory segments.
2. Call the _ucreate() sub-routine to create a handle to refer to the allocated memory. The created handle will be used to refer to the allocated memory pool in the subsequent _umalloc() subroutine calls.
3. Use the user-created heap.
   - Call the _umalloc() subroutine with the handle instead of the system malloc() subroutine.
   - Call the _ucalloc() subroutine with the handle instead of the system calloc() subroutine.
   - Call the _ufree() subroutine with the handle instead of the system free() subroutine.

   **Note:** For any memory object allocated by _umalloc() or _ucalloc(), you can find out from what heap it was allocated by calling the _mheap() sub-routine.

4. Release the user heap memory before the process exit.
   a. Deallocate the all acquired memory within the user-created heap by the _ufree() function
   b. Destroy the user-created heap with the _udestroy() function with the handle.
   c. Deallocate the memory chunks allocated in the first step depending on the type of memory source:
      - Call the system free() sub-routine to free memory from the process heap.
      - Call the shmdt() and shmctl() sub-routines to remove the allocated shared memory segments from the process address space.

## Managing the user-created heap size

Once a user-created heap is fully utilized, further allocation requests from that heap will fail, unless you have added more memory chunks using the _uaddmem() sub-routine. So make sure to allocate a block of memory large enough to satisfy all the memory requests in your program.

## Changing the default heap source

The use of _umalloc() and _ucalloc() does not conflict with the malloc subsystem subroutines, such as malloc() and calloc(). If malloc() is called within a user program, then the default process heap would be used to satisfy the memory request, even if the program has allocated a user-created heap in the shared memory segments.

However, if the _udefault() sub-routine with a handle parameter that points to the shared memory segments is called within a user program, then subsequent calls to malloc() and calloc() would start use of the user-created heap.

**Note:** The default heap source can be changed only when _udefault() is called.

The default heap source changes only for the Pthread that has called _udefault() within a process. You can use a different default heap for each thread of your multi-threaded application program.

The _udefault() function call returns the current default heap, so that the value can be saved and be used later to restore the default heap source. The default process heap is specified by the _RUNTIME_HEAP[6] macro.

### Gathering statistical data about user-created heaps

The _ustats() sub-routine reports the following statistical data about the specified user-created heap:

- How much memory the heap holds (excluding memory used for overhead)?
- How much memory is currently allocated from the heap?
- What type of memory is in the heap?
- The size of the largest contiguous piece of memory available from the heap?

## 4.4.2 A user-defined heap allocated from shared memory segments

An sample program shown in Example 4-12 on page 203 demonstrates how to use a user-defined heap. The program allocate 512 MB memory in shared memory segments, then calls the following sub-routines in turn:

**_ucreate()** Creates a handle to refer to the user-created heap.

**_umalloc()** Allocates memory size of 100000 bytes from the user-created heap.

**_ustats()** Prints the usage of the user-created heap.

**_ufree()** Un-allocates the previously created memory object using _umalloc() from the user-created heap.

**_udestroy()** Destroys the user-create heap.

[6] The _RUNTIME_HEAP macro is defined in the umalloc.h header file.

To compile this program, do the following:

```
$ cc user_heap.c -lhu
```

When executed, it prints the output shown in Example 4-11. It allocated a memory chunk size of 100016 bytes (16 bytes are used for overhead to maintain the statistics data) out of shared memory segments size of 512 MB.

*Example 4-11 Output from _ustats()*

```
$ a.out
----------------------------------------------------------
Heap information from _ustats():
..heapstat._provided            =  536870384
..heapstat._used                =          0
..heapstat._max_free            =  536870384
----------------------------------------------------------
----------------------------------------------------------
Heap information from _ustats():
..heapstat._provided            =  536870384
..heapstat._used                =     100016
..heapstat._max_free            =  536770368
----------------------------------------------------------
----------------------------------------------------------
Heap information from _ustats():
..heapstat._provided            =  536870384
..heapstat._used                =          0
..heapstat._max_free            =  536870384
----------------------------------------------------------
```

*Example 4-12 user_heap.c*

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <umalloc.h>
#include <sys/shm.h>

#define TWOSEGS 0x20000000

void heapstate(Heap_t usrheap)
{
    _HEAPSTATS heapstat;

    _ustats(usrheap, &heapstat);
    printf("----------------------------------------------------------\n");
    printf("Heap information from _ustats():\n");
    printf("..heapstat._provided\t\t= %10u\n", heapstat._provided);
    printf("..heapstat._used\t\t= %10u\n", heapstat._used);
    printf("..heapstat._max_free\t\t= %10u\n", heapstat._max_free);
    printf("----------------------------------------------------------\n");

    return;
}

int main(void)
{
    int     shmid, rc;
    char    buf[BUFSIZ];
    char    *ptr1, *ptr2, *shmptr;
    Heap_t  myheap;

    if ((shmid = shmget(IPC_PRIVATE, TWOSEGS, IPC_CREAT|S_IRUSR|S_IWUSR))
        < 0) {
        sprintf(buf, "shmget() failed at %d in %s with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
        exit(1);
    }
    if ((shmptr = shmat(shmid, (void *)0, 0)) == (void *) -1) {
        sprintf(buf, "shmat() failed at %d in %s with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
        exit(1);
    }
    /* create a user-created heap. */
    if ((myheap = _ucreate(shmptr, TWOSEGS, _BLOCK_CLEAN
        , _HEAP_SHARED|_HEAP_REGULAR, NULL, NULL)) == (void *)NULL) {
        sprintf(buf, "_ucreate() failed at %d in %s with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
        exit(1);
    }
    heapstate(myheap);
    ptr1 = _umalloc(myheap, 100000);       /*allocate from user heap. */
    heapstate(myheap);
    _ufree(ptr1);
    heapstate(myheap);
    /* destroy user heap. */
    if (_udestroy(myheap, _FORCE)) {
        sprintf(buf, "_destroy() failed at %d in %s with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
        exit(1);
    }

    if ((rc == shmdt(shmptr)) == -1) {
        sprintf(buf, "shmdt() at %d in %s failed with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
    }
    if (shmctl(shmid, IPC_RMID, 0) < 0) {
        sprintf(buf, "shmctl() at %d in %s failed with errno = %d"
            , __LINE__, __FILE__, errno);
        perror(buf);
    }

    exit(0);
}
```

# 5

# Creating DLPAR-aware applications

This chapter provides a brief overview for understanding the new programming paradigm available in a partitioned environment where resources can be dynamically changed/triggered by DLPAR operations. We discuss the following topics:

- Section 5.1, “Dynamic logical partitioning overview” on page 208
- Section 5.2, “The process flow of a DLPAR operation” on page 210
- Section 5.3, “DLPAR-safe and DLPAR-aware applications” on page 214
- Section 5.4, “Integrating a DLPAR operation into the application” on page 217

For further detailed information about the dynamic logical partitioning and the new programming paradigm, please refer the following publications:

- *The Complete Partitioning Guide for IBM @server pSeries Servers,* SG24-7039
- *IBM Hardware Management Console for pSeries Installation and Operations Guide*, SA38-0590

## 5.1 Dynamic logical partitioning overview

DLPAR, or dynamic logical partitioning, supports the following dynamic resource change in a partition without requiring a partition reboot:

- Resource addition
- Resource removal

By achieving the resource changes sequentially in the following order on two partitions in a system, the specified resource can be moved from a partition to another partition:

1. Resource removal from a partition
2. Resource addition to another partition

This resource movement is implemented as single task on the IBM Hardware Management Console for pSeries (hereafter referred to as HMC), although it is actually composed of two separate tasks on two partitions internally.

A resource is either of the following types:

- CPU

  The granularity of a CPU resource of a DLPAR operation is one CPU. More than one CPU can be specified as a resource of a DLPAR operation.

  A partition must be assigned at least the minimum number of processors specified in the partition profile, and it can be assigned up to the maximum number of processors specified in the partition profile.

  Therefore, you can dynamically add or remove processors for that partition within the range of the minimum and maximum values.

- Memory

  The granularity of a memory resource of a DLPAR operation is 256 MB[1]. Multiplies of 256 MB memory can be specified as a resource of a DLPAR operation.

  A partition must be assigned at least the minimum size of memory specified in the partition profile, and it can be assigned up to the maximum size of memory specified in the partition profile.

  Therefore, you can dynamically add or remove memory for that partition within the range of the minimum and maximum values.

[1] This memory chunk is referred to as a logical memory block (LMB).

- I/O resource

  The granularity of an I/O resource of a DLPAR operation is a PCI slot with a PCI adapter. Multiple I/O slots can be specified as a resource of a DLPAR operation. If a PCI adapter has multiple ports, all the ports and devices configured beneath the ports are treated as a resource.

  For example, if a 10/100 4-Port Ethernet adapter (FC 4961) is selected, both Ethernet devices (entX0) and interfaces (enX) configured on this adapter are treated as a single resource.

  A partition must be assigned all the adapters specified as *required* in the partition profile, and it can be assigned adapters specified as *desired* in the partition profile. Therefore, you can dynamically add or remove only adapters specified as desired for that partition.

  You cannot remove I/O slots listed as required; however, you can remove I/O slots listed as desired, or those that were added as a result of a DLPAR operation. In other words, a partition can currently contain an I/O slot that is *not* listed as either desired or required in the active partition profile.

**Note:** A DLPAR operation can perform only one type of resource change. You cannot add and remove memory to and from the same partition in a single DLPAR operation. Also, you cannot move CPU and memory from a partition to another partition in a single DLPAR operation.

Resources removed from a partition are marked free (free resources) and owned by the global firmware of system; you can consider these resources as kept in the "free resource pool." Free resources can be added to any partition in a system as long as the system has enough free resources.

It is imperative to understand that the DLPAR function is not solely provided by AIX 5L Version 5.2, but it is supported by the integration of following components:

- Hardware

  A partitioning-capable pSeries server model is required.

- Firmware

  Depending on the models you have selected, a firmware update might be required.

- HMC

  HMC software Release 3 Version 1 or later is required.

- Operating system

  AIX 5L Version 5.2 or later is required.

If one of these components does not satisfy the requirement to support DLPAR, the function is not available. For example, if a partition is installed with AIX 5L Version 5.1, that partition does not support DLPAR, although the other partitions installed with AIX 5L Version 5.2 support DLPAR.

## 5.2 The process flow of a DLPAR operation

A DLPAR operation initiated on the HMC is transferred to the target partition through Resource Monitoring and Control (RMC). The request produces a DLPAR event on the partition. After the event has completed, regardless of the result from the event, a notification will be returned to the HMC in order to mark the completion of the DLPAR operation. This means that a DLPAR operation is considered as a single transactional unit; thus, only one DLPAR operation is performed at a time.

A DLPAR operation is executed in the process flow, as illustrated in Figure 5-1.

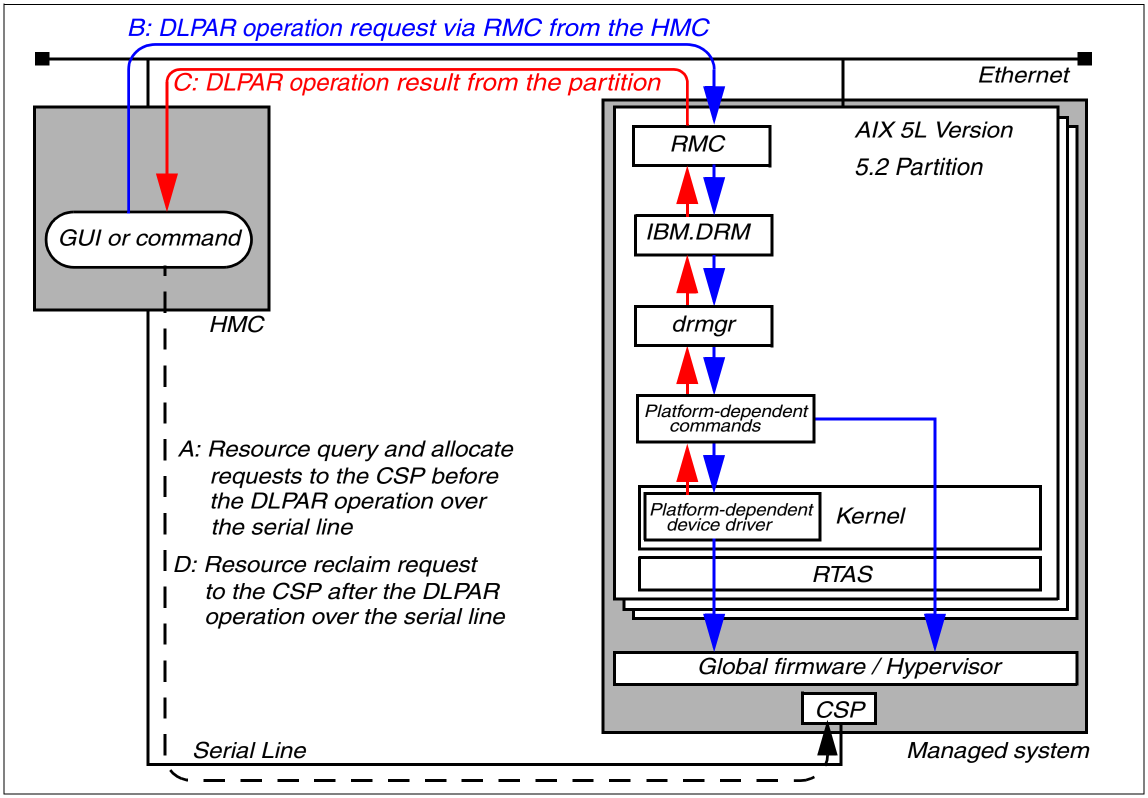

B: DLPAR operation request via RMC from the HMC
C: DLPAR operation result from the partition
Ethernet
RMC
AIX 5L Version
5.2 Partition
GUI or command
IBM.DRM
drmgr
HMC
Platform-dependent
commands
A: Resource query and allocate
requests to the CSP before
the DLPAR operation over
the serial line
Platform-dependent
device driver
Kernel
D: Resource reclaim request
to the CSP after the DLPAR
operation over the serial line
RTAS
Global firmware / Hypervisor
CSP
Serial Line
Managed system

*Figure 5-1 Process flow of a DLPAR operation*[2]

The following steps explain the process flow of a DLPAR operation:

1. The system administrator initiates a DLPAR operation request on the HMC using either the graphical user interface or command line interface.
2. The requested DLPAR operation is verified on the HMC with the current resource assignment to the partition and free resources on the managed system before being transferred to the target partition. In other words, the HMC provides the policy that determines whether or not a DLPAR operation request is actually performed on the managed system.
3. If the request is a resource addition, the HMC communicates with the global firmware in order to allocate free resources to the target partition through the service processor indicated as arrow A in Figure 5-1.

[2] A managed system is a partitioning partitioning-capable pSeries server system managed by the HMC.

If enough free resources exist on the system, the HMC assigns the requested resource to the specified partition and updates the partition’s object to reflect this addition, and then creates associations between the partition and the resource to be added.

4. After the requested DLPAR operation has been verified on the HMC, it will be transferred to the target partition using Resource Monitoring and Controlling (RMC), which is an infrastructure implemented on both the HMC and AIX partitions, as indicated as arrow B in Figure 5-1 on page 211. The RMC is used to provide a secure and reliable connection channel between the HMC and the partitions.

> **Note:** The connection channel established by RMC only exists between the HMC and the partition where the DLPAR operation is targeted to. There are no connection paths required between partitions for DLPAR operation purposes.

5. The request is delivered to the IBM.DRM resource manager running on the partition, which is in charge of the dynamic logical partitioning function in the RMC infrastructure in AIX. As shown in the following example, it is running as the IBM.DRMd daemon process and is included in the devices.chrp.base.rte fileset on AIX 5L Version 5.2 or later:

```
# lssrc -ls IBM.DRM
Subsystem         : IBM.DRM
PID               : 18758
Cluster Name      : IW
Node Number       : 1
Daemon start time : Wed Aug 21 16:44:12 CDT 2002

Information from malloc about memory use:
   Total Space    : 0x003502c0 (3474112)
   Allocated Space: 0x0030b168 (3191144)
   Unused Space   : 0x00043e40 (278080)
   Freeable Space : 0x00000000 (0)

Class Name(Id)    : IBM.DRM(0x2b) Bound
# ps -ef | head -1 ; ps -ef | grep DRMd | grep -v grep
     UID   PID  PPID   C    STIME    TTY  TIME CMD
    root 18758 10444   0   Aug 21      -  0:22 /usr/sbin/rsct/bin/IBM.DRMd
# lslpp -w /usr/sbin/rsct/bin/IBM.DRMd
  File                                          Fileset              Type
  ---------------------------------------------------------------------------
  /usr/sbin/rsct/bin/IBM.DRMd
                                         devices.chrp.base.rte       File
```

> **Note:** The absence of the IBM.DRM resource manager in the **lssrc -a** output does not always mean that the partition has not been configured appropriately for the dynamic logical partitioning. The resource manager will be automatically configured and started by RMC after the first partition reboot if the network configuration is correctly set up on the partition and the HMC.

Resource managers are sub-systems used in the RMC infrastructure. For further information about RMC and its sub-components, please refer to the following publications:

– *A Practical Guide for Resource Monitoring and Control (RMC)*, SG24-6615

– *IBM Reliable Scalable Cluster Technology for AIX 5L: Administration Guide*, SA22-7889

6. The IBM.DRM resource manager invokes the **drmgr** command, which is an platform-independent command designed as the focal point of the dynamic logical partitioning support on AIX.

   As shown in the following example, the **drmgr** command is installed in the /usr/sbin directory provided by the bos.rte.methods fileset:

```
# whence drmgr
/usr/sbin/drmgr
# lslpp -w /usr/sbin/drmgr
  File                                        Fileset               Type

---------------------------------------------------------------------------
  /usr/sbin/drmgr                             bos.rte.methods       File
```

> **Note:** The **drmgr** command should not be invoked by the system administrator in order to directly perform resource changes in a partition. It must be invoked in the context explained here to do so.

7. The **drmgr** command invokes several platform-dependent commands[3] depending on the resource type (CPU, memory, or I/O resource) and request (resource addition or removal) in order to instruct the kernel to process the actual resource change with necessary information.
8. The kernel does many internal tasks.
9. After the DLPAR event has completed, regardless of the result, a notification will be returned to the HMC in order to mark the completion of the DLPAR

[3] On the current partitioning-capable pSeries server models, the platform-dependent commands are included in the bos.chrp.base.rte fileset and installed in the /usr/lib/boot/bin directory.

operation, indicated as arrow C in Figure 5-1 on page 211. The notification also includes the exit code, standard out, and standard error from the **drmgr** command. The system administrator who has initiated the DLPAR operation will see the exit code and outputs on the HMC.

10. If the request is a resource removal, the HMC communicates with the global firmware in order to reclaim resources to the free resource pool from the source partition through the service processor, indicated as arrow D in Figure 5-1 on page 211.

    The HMC unassigns the resource from the partition and updates the partition's object to reflect this removal, and then removes associations between the partition and the resource that was just removed.

A DLPAR operation can take a considerable amount of time, depending on the availability and the capability to configure or deconfigure a specific resource.

## 5.3 DLPAR-safe and DLPAR-aware applications

The dynamic logical partitioning function on AIX 5L Version 5.2 is designed and implemented to not impact the existing applications. In fact, most applications are not affected by any DLPAR operations results. Therefore, those applications are called *DLPAR-safe* applications.

There are two types of application classifications regarding DLPAR operations:

**DLPAR-safe** Applications that do not fail as a result of DLPAR operations. The application's performance may suffer when resources are removed, or it may not scale as resources are added.

**DLPAR-aware** Applications that incorporate DLPAR operations that allow the application to adjust its use of the system resources equal to the actual capacity of the system. DLPAR-aware applications are always DLPAR-safe.

### 5.3.1 DLPAR-safe

Although, most applications are DLPAR-safe without requiring any modification, there are certain instances where programs may not be inherently DLPAR-safe.

There are two cases where DLPAR operations may introduce undesirable effects into the application:

- Programs that are optimized for uni-processor may have problems when a processor is added to the system resources.

- On programs that are indexed by CPU numbers, the increased processor number may cause the code to go down an unexpected code path during its run-time checks.

In addition, applications that use uni-processor serialization techniques may experience unexpected problems. In order to resolve these concerns, system administrators and application developers need to be aware of how their applications get the number of processors.

## 5.3.2 DLPAR-aware

DLPAR-aware applications adapt to system resource changes caused by DLPAR operations. When these operations occur, the application will recognize the resource change and accommodate accordingly.

Two techniques can be used to make applications DLPAR-aware:

- The first method is to consistently poll for system resource changes. Polling is not the recommended way to accommodate for DLPAR operations, but it is valid for systems that do not need to be tightly integrated with DLPAR. Because the resource changes may not be immediately discovered, an application that uses polling may have limited performance. Polling is not suitable for applications that deploy processor bindings, because they represent hard dependencies.
- Applications have other methods to react to the resource change caused by DLPAR operations. See 5.4, "Integrating a DLPAR operation into the application" on page 217.

Several applications should be made DLPAR-aware, because they need to scale with the system resources. These types of applications can increase their performance by becoming DLPAR-aware. Table 5-1 on page 216 lists some examples of applications that should be made DLPAR-aware.

> **Note:** These are only a few types of common applications affected by DLPAR operations. The system administrator and application developer should be sensitive to other types of programs that may need to scale with resource changes.

*Table 5-1 Applications that should be DLPAR-aware*

| Application type | Reason |
|---|---|
| Database applications | The application needs to scale with the system. For example, the number of threads may need to scale with the number of available processors, or the number of large pinned buffers may need to scale with the available system memory. |
| License Managers | Licenses are distributed based on the number of available processors or the memory capacity. |
| Workload Managers | Jobs are scheduled based on system resources, such as available processors and memory. |
| Tools | Certain tools may report processor and memory statistics or rely on available resources. |

Applications should also be made DLPAR-aware, if they cause DLPAR requests to fail.

The following application specific conditions can cause DLPAR remove requests to fail:

- Processor bindings
- Large amounts of pinned memory

Processor bindings and attachments are obstacles to CPU removal, since they represent work units that cannot be rescheduled on other processors. DLPAR-aware programs remove these obstacles by unbinding, re-binding, or terminating before the operating system attempts to remove the resource. AIX provides several interfaces for binding and attaching to processors including bindprocessor(), ra_exec(), ra_fork(), and ra_attachrset(). If the bindprocessor() interface is used, then the target processor is always the Nth online processor. The other interfaces target logical processors, so there is a direct correlation between the processor being removed and the attachment, which is not necessarily the case with bindprocessor(). It should be noted that bindprocessor() bindings may also be resolved through application handlers that may have already been provided for dynamic processor deallocation (for example, CPU Guard).

Applications that pin large amounts of memory may cause DLPAR memory removal failures, since they impede the system's capability to pin new memory pages, which is a critical component of the DLPAR support for memory. AIX has the capability to migrate pinned memory, but it must ensure that the target of the migration is also pinned, so it must be able to pin memory on demand to satisfy DLPAR memory remove requests. DLPAR-aware applications that pin lots of memory should resize their buffers so that portions of them can be unpinned, before the operating system attempts to remove memory. Applications pin memory through the plock() and shmget(SHM_PIN) interfaces.

Applications that have processor attachments and pin memory are DLPAR-safe and they should be made DLPAR-aware.

Applications are not impacted by the dynamic reconfiguration of I/O resources, because access to the device is controlled through the device support layer, which is not manipulated by DLPAR. The system administrator is expected to unload device drivers before initiating DLPAR remove requests and, in a similar vein, he is expected to manually configure them after performing DLPAR add requests to make them available to applications. The DLPAR support for I/O resources is therefore DLPAR-safe. For example, DLPAR I/O remove requests will fail If a device driver that is associated with the I/O resource is loaded. The device driver does not have to be open for the DLPAR remove operation to fail. It simply has to be loaded.

## 5.4 Integrating a DLPAR operation into the application

The DLPAR operation can be integrated into the application using the following two methods:

- Script-based DLPAR event handling

  If the application is externally controlled to use a specific number of threads or to size its buffers, use this method. In order to facilitate this method, a new command, **drmgr**, is provided. The **drmgr** command is the central focal point of the dynamic logical partitioning function of AIX.

- API-based DLPAR event handling

  If the application is directly aware of the system configuration, and the application source code is available, use this method.

Applications can monitor and respond to various DLPAR events, such as a memory addition or processor removal, by utilizing these two methods. Although, at the high-level, both methods share the same DLPAR events flow explained in the following section, several key differences exist between these two methods.

One difference is that the script-based method externally reconfigures the application once a DLPAR event takes place, while the API-based method can be directly integrated into the application by registering a signal handler using the dr_reconfig system call, so that the process can be notified with the SIGRECONFIG signal when the DLPAR event occurs.

**Note:** The DLPAR events of I/O resources are not notified applications in AIX 5L Version 5.2.

# 6

# Programming hints and tips

In developing your application with C or C++, you may encounter problems while compiling, linking, or executing for the first time.

This chapter starts off by offering some general programming tips that would make an application perform better on AIX:

- Section 6.1, “Programming recommendations” on page 220

The next sections continue to describe some of the more common errors that you may encounter during the different phases of the development cycle:

- Section 6.2, “Diagnosing compile-time errors” on page 227
- Section 6.3, “Diagnosing link-time errors” on page 239
- Section 6.4, “Diagnosing run-time errors” on page 244

All the options and #pragma directives referred to in this chapter are explained in more details in the *C for AIX Compiler Reference*, SC09-4960 and *VisualAge C++ for AIX Compiler Reference*, SC09-4959.

## 6.1 Programming recommendations

C and C++ are very powerful high level programming languages. Two applications can be performing the same tasks, but written in completely different ways using the many different features in the language. Programming style is very personal, but there are programming practices that, when followed, will help get the best results from the optimization techniques used by the compiler. This section offers some tips and guidelines that will allow you to write programs that exploit the C for AIX and VisualAge C++ for AIX compilers.

### 6.1.1 Variables and data structures

In C and C++, the default storage duration, scope, and linkage of variables and objects depend on where they are declared. However, storage duration can be explicitly overridden with storage class specifiers. Whenever possible, it is recommended that you use local variables of the automatic storage class.

Several optimizations performed by the compiler rely on data flow analysis. For example, whether a store into a variable is redundant and can be removed, because a later store will invalidate the first store, before the variable is referenced:

```
int func1()
{
    int x;
    x = 1;
    func();
    x = 2;
    return x;
}
```

In this example, the first assignment, x = 1, can be safely removed, since the second assignment will overwrite it, and the call to function func() will not access variable x. Had variable x been declared as global, the compiler would not have made the same assumption, since it is now possible for the function func() to be referencing the global variable x.

Data flow analysis also helps in determining whether branches can be eliminated. For example:

```
void func2()
{
    int x;
    x = 1;
    func();
    if (x == 1)
        printf("true\n");
    else
        printf("false\n");
}
```

Again, since the function call to func() cannot possibly be updating the local variable x, the compiler can safely remove the conditional statement, and generate code for the true part of the statement only.

Using local, automatic storage variables is not always possible. If two functions within the same source file need to share data, it is recommended that you use variables with static storage. Static storage class variables and objects have internal linkage, that is, the variable is accessible within the compilation unit only, and not visible externally. The compiler in this case can use this information to better optimize your program, since static variables appear as local within the compilation unit.

If you must use external variables that are shared between more than one compilation units, try to group the variables into structures to avoid excessive memory access.

### 6.1.2 Functions

Without knowing what a function does, that is, if the function is not defined in the current compilation unit, the compiler has to assume the worse, that calling the function may have side effects. A side effect, according to the standard, is a change in the state of the execution environment. Several operations can produce a side effect:

- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Calling a function that does any of the above

If a function requires input, it is recommended that you pass the input as arguments, rather than having the function access the input from global variables. If your function has no side effects, that is, it does not violate any of the points above, you can use the #pragma isolated_call directive to specify that the compiler may take a more aggressive approach in optimization. This generally improves the execution performance. See *C for AIX Compiler Reference*, SC09-4960 for more details on the directive. Functions that are marked with the #pragma isolated_call directive are allowed to modify storage pointed to by pointer arguments (that is, reference arguments).

If a function is only required in the current compilation unit, declaring it as static will speed up calls to the function.

In C++, use virtual functions only when absolutely necessary. The location of virtual functions are not known until execution time. The compiler generates an indirect call to virtual functions via entries in a virtual function table, which gets populated with the location of the virtual functions at load time. To reduce code bloat, avoid declaring virtual functions inline, which tends to make the virtual function table bigger and causes the virtual function defined in all compilation units that use the class.

### 6.1.3 Pointers

The use of pointers causes uncertainty in data flow analysis, which can sometimes inhibit optimization performed by the compiler. If a pointer is assigned to the address of a variable, both the pointer and the variable can now be used to change or reference the same memory location in an expression. For example:

```
void func()
{
    int i = 55, a, b;
    int *p = &i;

    a = *p;
    i = 66;
    b = *p;
}
```

At first glance, it looks as if the expression *p does not need to be evaluated the second time, and the variable a and b would have the same value. However, since p and the address of i refer to the same memory location, the assignment i = 66 invalidates the first *p evaluation and b = *p must be evaluated again.

When the compiler knows nothing about where a pointer can point to, it will have to make the pessimistic assumption that the pointer can point to any variable.[1] For example:

```
int rc;

void foo(int *p)
{
    rc = 55;
    *p = *p + 1;
```

[1] In the standard conforming language levels, e.g. -qlanglvl=stdc89 and -qlanglvl=stdc99, or when the -qalias=ansi option is in effect, a pointer can only point to variables of the same type. This is referred to as type-based aliasing. The only exception is with character pointers and pointers to void; they can point to any type. The use of -qalias=ansi to correct programming mistakes is not recommended, as it inhibits optimization opportunities and degrades execution performance.

```
    rc = 66;
}
```

In this case, the compiler has to assume that when the function foo is called, pointer p can potentially point to the variable rc. The first assignment, rc = 55, is relevant and cannot be safely removed, since *p on the following line can be referring to rc.

If p is guaranteed not to be pointing to rc, you can use the #pragma disjoint directive to inform the compiler:

```
#pragma disjoint(*p, rc)
```

Marking variables that do not share physical memory storage with the #pragma disjoint directive allows the compiler to explore more optimization opportunities, and the performance gain can be quite substantial with complex applications. However, if the directive is used inappropriately on variables that may share physical storage, the program may yield incorrect results.

### 6.1.4 Arithmetic operations

Multiplication in general is faster than division. If you need to perform many divisions with the same divisor, it is recommended that you assign the reciprocal of the divisor to a temporary variable, and change all your divisions to multiplications with the temporary variable. For example, instead of:

```
double preTax(double total)
{
    return total / 1.0825;
}
```

this will perform faster:

```
double preTax(double total)
{
    return total * (1.0 / 1.0825);
}
```

The division (1.0 / 1.0825) will be evaluated once at compile time only due to constant folding.

### 6.1.5 Selection and iteration statements

When using the if selection statement, order your conditional expressions efficiently by putting the most decisive test first. For example:

```
struct {
    char *name;
    short len;
    _Bool active;
} rec;
...
if (rec.active == true && rec.len == 9 && !memcmp(rec.name, “Elizabeth”, 9))
...
```

Also, order your case statements or if-else if conditions in a way that the most likely occurring conditions appear first. For example:

```
typedef enum { VOWEL, CONSONANT, DIGIT, PUNCTUATION } _Type;
struct {
    char  ch;
    _Type type;
} word;
...
switch (word.type) {
    case VOWEL:
        ...
      break;
    case CONSONANT:
        ...
      break;
    case PUNCTUATION:
        ...
      break;
    case DIGIT:
        ...
      break;
}
```

and:

```
if (!error) {
    /* most likely condition first */
} else {
    /* error condition that does not happen too often */
}
```

When using iteration statements such as for loop, do loop, and while loop, move invariant expressions (expressions that do not change with loop iterations, nor depend on the loop index of a for loop) out of the loop body.

### 6.1.6 Expression

The C and C++ compilers are able to recognize common sub-expressions, when the sub-expression either:

- Appears at the left end of the expression
- Within parentheses

For example, the compiler recognizes the sub-expression a + b in the two assignments:

```
x = a + b + c;
y = d * (a + b);
```

and evaluates the two sub-expressions only once. Essentially, it is logically transforming the two lines into:

```
temp = a + b;
x = temp + c;
y = d * temp;
```

The use of the temp illustrates how the compiler can keep the result of the sub-expression evaluation in a register, and simply reuse the register for both assignments.

## 6.1.7 Memory usage

Avoid heap storage fragmentation with frequent allocations of small, temporary objects using memory allocation functions such as malloc() and calloc().

```
while (list) {
    char *temp = (char*)malloc(list->len+1);
    strcpy(temp, list->element);
    PrettyPrint(temp);
    free(temp);
    list = list->next;
}
```

In the example above, if the size of the longest element is known beforehand, you can replace the call to malloc() and obtain storage from the stack instead by using an array of characters:

```
char temp[MAX_ELEMENT_SIZE];
```

If the size is unknown, you can make use of the C99 feature, “Variable length arrays” on page 7 to allocate storage dynamically for the array:

```
char temp[list->len+1];
```

In either case, the storage is obtained from the stack and not from the heap, and it is freed automatically when the scope of the variable terminates.

## 6.1.8 Built-in functions

For performance reasons, a large number of the library functions are also provided as compiler built-ins. A compiler built-in avoids any overhead associated with function calls (for example, parameter passing, stack allocation and adjustment, and so on) by expanding the function body at the call site. Various hardware instruction built-ins are also provided to allow programmers direct access to hardware instructions.

To use the built-in version of library functions, you need to include the appropriate library header files. Including the proper header files also prevents parameter type mismatch and ensures optimal performance.

### 6.1.9 Virtual functions

In general, when writing C++ code, you should try and avoid the use of virtual functions. They are normally encoded as indirect function calls, which are slower than direct function calls.

Usually, you should not declare virtual functions inline. In most cases, declaring virtual functions inline does not produce any performance advantages. Consider the following code sample:

```
class Base {
public:
    virtual void foo() { /* do something. */ }
};

class Derived: public Base {
public:
    virtual void foo() { /* do something else. */ }
};

int main(int argc, char *argv[])
{
    Base*  b = new Derived();
    b->foo(); // not inlined.
}
```

In this example, b->foo() is not inlined because it is not known until run time which version of foo() must be called. This is by far the most common case.

There are cases, however, where you might actually benefit from inlining virtual functions, for example, if Base::foo() was a really hot function, it would get inlined in the following code:

```
int main(int argc, char *argv[])
{
    Base  b;
    b.foo();
}
```

If there is a non-inline virtual function in the class, the compiler generates the virtual function table in the first file that provides an implementation for a virtual function; however, if all virtual functions in a class are inline, the virtual table and virtual function bodies will be replicated in each compilation unit that uses the class. The disadvantage to this is that the virtual function table and function bodies are created with internal linkage in each compilation unit. This means that even if the -qfuncsect option is used, the linker cannot remove the duplicated table and function bodies from the final executable. This can result in a very bloated executable size.

# 6.2 Diagnosing compile-time errors

During compilation, the compiler emits diagnostic messages to the standard error device file (the terminal by default) whenever it encounters programming errors caused by invalid syntax or incorrect usage of features. There are various compiler diagnostic aids that can help you in determining where the error occurred.

## 6.2.1 Anatomy of a message

A compiler error message has the following format, which provides enough information about the location and reason for the error:

“*file*”, `line` *line.column*: `15`*cc-nnn* (*sev*) *msg*

Where:

**file** The name of the source file where the error occurred.

**line** The line in the source file where the error occurred.

**column** The column on the line where the error occurred.

**cc** A two-digit code identifying the compiler component that issued the diagnostic message:

| | |
|---|---|
| **00** | Optimizer or code generator |
| **01** | Compiler services |
| **05** | C compiler |
| **06** | C compiler |
| **40** | C++ compiler |
| **47** | Munch utility |
| **86** | Interprocedural analysis (IPA) |

**nnn** The message number.

**sev** The severity of the error.

**I** Informational

**W** Warning

**E** Error

**S** Severe error

**U** Unrecoverable error

**msg** Short message describing the error.

## 6.2.2 Useful options and compiler aids

The C and C++ compilers offer several options that help you detect and correct programming errors in you program.

### The -qsrcmsg option

By default, compiler diagnostic messages are issued in the format depicted in 6.2.1, “Anatomy of a message” on page 227. Sometimes it may not be apparent what the problem is with the short explanation in the message text. The C compiler -qsrcmsg option prints the source line and a finger line pointing to where the compiler thinks the error is to the stderr file, giving a more precise explanation by showing the actual source line containing the error. For example:

```
        7 |         char new[n];
            .................a..
a - 1506-195 (S) Integral constant expression with a value greater than zero is
required.
```

The source line shown will be after macro expansion.

### Compiler listing

You can use the -qsource option to ask for a compiler listing. A compiler listing contains several sections that are useful in determining what has gone wrong in a compilation. Diagnostic messages, if present, are written to the compiler listing as well. See Example F-1 on page 490 for a sample compiler listing.

- SOURCE SECTION

  Shows the source code with line numbers. Lines containing macros will have additional lines showing the macro expansion. By default, this section only lists the main source file. Use the -qshowinc option to expand all header files as well.

- OPTIONS SECTION

  Shows any non-default options that were in effect for the compilation. To list all options in effect, specify the -qlistopt option.

- FILE TABLE SECTION

  Lists all the files used in the compilation. Each file is associated with a file number, and it always begin with the main source file with file number 0. It shows, for each file, from which file and line the file was included. If the -qshowinc option is also in effect, each source line in the SOURCE SECTION will have a file number to indicate which file the line came from.

- COMPILATION EPILOGUE SECTION

  Shows a summary of the diagnostic messages by severity level, the number of lines read, and whether the compilation was successful or not.

- ATTRIBUTE AND CROSS REFERENCE SECTION

  The -qattr and -qxref options will cause this section to be produced. Independently, they provide information on all identifiers used in the compilation. This is where you will find pertinent information about variables on their type, storage duration, scope, and where they are defined and referenced.

- OBJECT SECTION

  The -qlist option will cause this section to be produced. It shows the pseudo assembly code generated by the compiler. This section is invaluable for diagnosing execution time problems, if you suspect the program is not performing as expected due to code generation error.

## The -qinfo option

The -qinfo option causes the compiler to produce additional informational messages for possible programming errors. The extra diagnostic messages can help you in debugging your program.

Messages related to the same problem area are grouped together, and each group is controlled via a suboption. For instance, the -qinfo=ini suboption diagnoses possible problems with variables that are not explicitly initialized but should be. For a list of the groups (suboptions) supported, refer to the *VisualAge C++ for AIX Compiler Reference*, SC09-4959.

One new info group introduced in the Version 6 C compiler is the -qinfo=c99 suboption. It diagnoses C code that may have different behavior between the -qlanglvl=stdc89 and -qlanglvl=stdc99 language levels. For example:

```
$ cat test.c
#include <stdio.h>

int main()
{
    printf("sizeof(2147483648) = %d\n", sizeof(2147483648));
    return 0;
}
$ cc -qinfo=c99 -c test.c
"test.c", line 4.48: 1506-786 (I) Integral constant "2147483648" has an implied
type of unsigned long int under the C89 language level. It has an implied type
of long long int under C99.
```

### The -qsuppress option

When you use diagnostic aids like the -qinfo option, you can sometimes get numerous messages that you may not be interested in at the moment, clobbering the terminal or listing file. Use the -qsuppress option to stop those messages from being emitted by the compiler. You can suppress more than one message by listing the message numbers in a colon separated list.

### The -qflag option

In some instances, you may want to ignore all messages below a certain severity level. For example, during a production build, warning messages that are deemed to be harmless by development teams will only clobber log files and cause unnecessary concerns from the build team. You can use the -qflag option to stop diagnostic messages from being emitted to the terminal and the listing file. Similarly, the -w compiler flag suppresses informational and warning messages from being emitted, and is the same as specifying the -qflag=e:e option. See 6.2.1, “Anatomy of a message” on page 227 for a list of single letter severity codes that you can use with the -qflag option.

### The -qhaltonmsg option

On the contrary, with the C++ compiler, if you want to stop the compilation when a certain message is encountered, use the -qhaltonmsg option. The message is issued with a higher severity level and compilation terminates.

### The -qhalt and -qmaxerr options

To stop a compilation in general when a message of a certain severity level or higher is encountered, use the -qhalt option. See 6.2.1, “Anatomy of a message” on page 227 for a list of single letter severity codes that you can use with the -qhalt option.

The -qmaxerr option allows you to stop the compilation when the specified number of times messages of a certain severity level or higher is reached. This option takes precedence over the -qhalt option.

## 6.2.3 Migrating from 32-bit to 64-bit

When migrating a 32-bit program to 64-bit, the data model differences (as described briefly in 2.1, “32- and 64-bit development environments” on page 38) may result in unexpected behavior at execution time. In 64-bit mode, the size of pointers and long data type are now 8 bytes long, and can lead to several conversion or truncation problems. The -qwarn64 option can be used to detect these portability errors.

## int and long types

In 32-bit mode, both int and long data types are 32 bits in size. Because of this similarity, these types may have been used interchangeably. As shown in Table 2-1 on page 39, the data type long is 64 bits in length in 64-bit mode. A general guideline is to review the existing use of long data types throughout the source code. If the values to be held in such variables, fields, and parameters will fit in the range of [-$2^{31}$...$2^{31}$-1] or [0...$2^{32}$-1], then it is probably best to use int or unsigned int instead. Also, review the use of the size_t type (used in many subroutines), since it is typedef as unsigned long.

## long to int truncation

Truncation problems can occur when converting between 64-bit and 32-bit data objects. Since int and long are 32 bits in 32-bit mode, a mixed assignment or conversion between these data types did not represent any problem. It does, however, in 64-bit mode, as long is larger in size than int. When converting from long to int, either implicitly or explicitly through a cast, truncation may now occur:

```
void foo(long l)
{
    int i = l;
}
```

Without an explicit cast, the compiler is unable to determine whether the narrowing assignment is intended. If the value l is always within the range representable by an int, or if the truncation is intended by design, use a cast to silence the -qwarn64 message that you will receive for this code.

## Unexpected result due to conversion to long

Due to the difference in size for int and long in 64-bit mode, conversions to long from other integral types may result in different execution behavior from 32-bit mode in some boundary cases.

When a signed char, signed short, or signed int is converted to unsigned long, sign extension may result in a different unsigned value in 64-bit mode. For example:

```
#include <stdio.h>
void foo(int i)
{
    unsigned long l = i;
    printf("%lu (0x%lx)\n", l, l);
}
void main()
{
    foo(-1);
}
```

This program will yield 4294967295 (0xffffffff) in 32-bit mode but 18446744073709551615 (0xffffffffffffffff) in 64-bit mode due to sign extension.

When an unsigned int variable with values greater than INT_MAX is converted to signed long, you will get different results between 32-bit and 64-bit mode. For example:

```
#include <stdio.h>
#include <limits.h>
void foo(unsigned int i)
{
    long l = i;
    printf("%ld (0x%lx)\n", l, l);
}
void main()
{
    foo(INT_MAX + 1);
}
```

In 32-bit mode, the value INT_MAX+1 will wrap around and yield -2147483648 (0x80000000). The same value can be represented in 64-bit mode by a 8-byte signed long and will result in the correct value of 2147483648 (0x80000000).

When a signed long long variable with values greater than UINT_MAX or less than 0 is converted to unsigned long, truncation will no longer occur in 64-bit mode. For example:

```
#include <stdio.h>
#include <limits.h>
void foo(signed long long ll)
{
    unsigned long l = ll;
    printf("%lu (0x%lx)\n", l, l);
}
void main()
{
    foo(-1);
    foo(UINT_MA X+ 1ll);
}
```

This program will yield:

```
4294967295 (0xffffffff)
0 (0x0)
```

in 32-bit mode and in 64-bit mode:

```
18446744073709551615 (0xffffffffffffffff)
4294967296 (0x100000000)
```

When an unsigned long long variable with values greater than UINT_MAX is converted to unsigned long, truncation will no longer occur in 64-bit mode:

```
#include <stdio.h>
#include <limits.h>
void foo(unsigned long long ll)
{
    unsigned long l = ll;
    printf("%ld (0x%lx)\n", l, l);
}
void main()
{
    foo(UINT_MAX + 1ull);
}
```

The higher order word is truncated and will result in `0 (0x0)` in 32-bit mode, but yield the correct result of `4294967296 (0x100000000)` without truncation in 64-bit mode.

When a signed long long variable with values less than INT_MIN or greater than INT_MAX is converted to signed long, truncation will no longer occur in 64-bit mode. For example:

```
#include <stdio.h>
#include <limits.h>
void foo(signed long long ll)
{
    signed long l = ll;
    printf("%ld (0x%lx)\n", l, l);
}
void main()
{
    foo(INT_MIN - 1ll);
    foo(INT_MAX + 1ll);
}
```

This program will yield (in 32-bit):

```
2147483647 (0x7fffffff)
-2147483648 (0x80000000)
```

and in 64-bit mode:

```
-2147483649 (0xffffffff7fffffff)
2147483648 (0x80000000)
```

And finally, when an unsigned long long variable with values greater than INT_MAX is converted to signed long, truncation will no longer occur in 64-bit mode. For example:

```
#include <stdio.h>
#include <limits.h>
void foo(unsigned long long ll)
{
    signed long l = ll;
    printf("%ld (0x%lx)\n", l, l);
}
void main()
{
    foo(INT_MAX + 1ull);
}
```

In 32-bit mode, the value INT_MAX+1ull will wrap around and yield `-2147483648 (0x80000000)`. The same value can be represented in 64-bit mode by a 8-byte signed long and will result in the correct value of `2147483648 (0x80000000)`.

## Pointer assignment and arithmetic

When migrating a program from 32-bit environment to 64-bit environment, it is crucial to avoid pointer corruption. Some of the possible problems are:

- Assigning an int (32 bits) or a 32-bit hexadecimal constant to a pointer type variable (64 bits) or casting a pointer to an int will yield an invalid address, and will cause errors when the pointer is dereferenced. Also, the comparison of an int to a pointer may cause unexpected results.
- Pointers are converted to int or unsigned int with the expectation that the pointer value will be preserved, as casting a pointer to an int will result in data truncation.
- Without proper function prototypes, functions that return pointers will return truncated return values, as the functions are implicitly declared to return an int that is just 32 bits, instead of the expected 64 bits of a pointer.
- The code assumes that pointers and int are the same size in an arithmetic context, as pointer arithmetic usually is a source of problems in migration. The ISO C standard dictates that incrementing a pointer yields adding the size of the data type to which it points to the pointer value. For example, if the variable p is a pointer to long, then the operation (p+1) increments the value of p by 4 bytes in 32-bit mode but by 8 bytes in 64-bit mode. Therefore, casts

between long* to int* are problematic because of the size differences of pointer objects (32 bits versus 64 bits).

## Incorrect pointer to int and int to pointer conversions

When a pointer is explicitly converted to an int, truncation of the high order word occurs. When an int is explicitly converted to a pointer, the pointer may not be correct and may cause invalid memory access if dereferenced. For example:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int i, *p, *q;

    p = (int*)malloc(sizeof(int));
    i = (int)p;
    q = (int*)i;
    p[0] = 55;

    printf("p = %p q = %p\n", p, q);
    printf("p[0] = %d q[0] = %d\n", p[0], q[0]);
}
```

In 32-bit mode, the pointers p and q are pointing to the same memory location. However, the pointer q is likely pointing to an invalid address in 64-bit mode, and could result in a segmentation fault when q is dereferenced.

## Integer constants

A loss of data can occur in some constant expressions because of lack of precision. These types of problems are very hard to find and may have gone unnoticed so far. You should therefore be very explicit about specifying the type(s) in your constant expressions and use constant suffix {u,U,l,L,ll,LL} to specify exactly its type. You might also use casts to specify the type of a constant expression.

This is especially true when migrating to 64-bit, because integer constants may have different types when compiled in 64-bit mode. The ISO C standard states that the type of an integer constant, depending on its format and suffix, is the first (that is, smallest) type in the corresponding list (see Table 6-1 on page 236) that will hold the value. The quantity of leading zeros does not influence the type selection.

*Table 6-1 ISO C99 integer constant type selection*

| Suffix | Decimal constant | Octal or hexadecimal constant |
|---|---|---|
| unsuffixed | int<br>long<br>long long | int<br>unsigned int<br>long<br>unsigned long<br>long long<br>unsigned long long |
| u or U | unsigned int<br>unsigned long<br>unsigned long long | unsigned int<br>unsigned long<br>unsigned long long |
| l or L | long<br>long long | long<br>unsigned long<br>long long<br>unsigned long long |
| Both u or U and l or L | unsigned long<br>unsigned long long | unsigned long<br>unsigned long long |
| ll or LL | long long | long long<br>unsigned long long |
| Both u or U and ll or LL | unsigned long long | unsigned long long |

For example, a hexadecimal constant that could only be represented by an unsigned long in 32-bit will now fit within a long in 64-bit. The change in type of the constant in an expression may cause unexpected results.

## Data alignment

Modern processor designs usually require data in memory to be aligned to their natural boundaries in order to gain the best possible performance. The compiler in most cases guarantees data objects to be properly aligned by inserting padding bytes immediately before the misaligned data. Although the padding bytes do not affect the integrity of the data, they can cause the layout and hence the size of structures and unions to be different than expected.

Because the size of pointers and long is doubled in 64-bit mode, structures and unions containing them as members will become bigger than in 32-bit mode. For example, consider the structure in Example 6-1 on page 237.

*Example 6-1 Structure alignment*

```
#include <stdio.h>
#include <stddef.h>
void main()
{
    struct T {
        char c;
        int *p;
        short s;
    } t;
    printf("sizeof(t) = %d\n", sizeof(t));
    printf("offsetof(t, c) = %d sizeof(c) = %d\n",
            offsetof(struct T, c), sizeof(t.c));
    printf("offsetof(t, p) = %d sizeof(p) = %d\n",
            offsetof(struct T, p), sizeof(t.p));
    printf("offsetof(t, s) = %d sizeof(s) = %d\n",
            offsetof(struct T, s), sizeof(t.s));
}
```

When Example 6-1 is compiled and executed in 32-bit mode, the following result indicates paddings have been inserted before the member p, and after the member s:

```
sizeof(t) = 12
offsetof(t, c) = 0 sizeof(c) = 1
offsetof(t, p) = 4 sizeof(p) = 4
offsetof(t, s) = 8 sizeof(s) = 2
```

Three padding bytes are inserted before the member p to ensure p is aligned to its natural 4-byte boundary. The alignment of the structure itself is the alignment of its strictest member. In this example, it is 4-byte due to the same member p. Therefore, two padding bytes are inserted at the end of the structure to make the total size of the structure a multiple of 4-byte (see Figure 6-1). This is required so that if you declare an array of this structure, each elements of the array will be aligned properly.

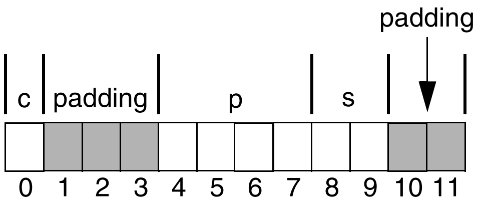


*Figure 6-1 Structure padding in 32-bit mode*

However, when Example 6-1 on page 237 is compiled and executed in 64-bit mode, the size of the structure doubles, which is caused by more paddings required to make the member p to fall on a natural alignment boundary of 8-byte (see Figure 6-2):

```
sizeof(t) = 24
offsetof(t, c) = 0 sizeof(c) = 1
offsetof(t, p) = 8 sizeof(p) = 8
offsetof(t, s) = 16 sizeof(s) = 2
```

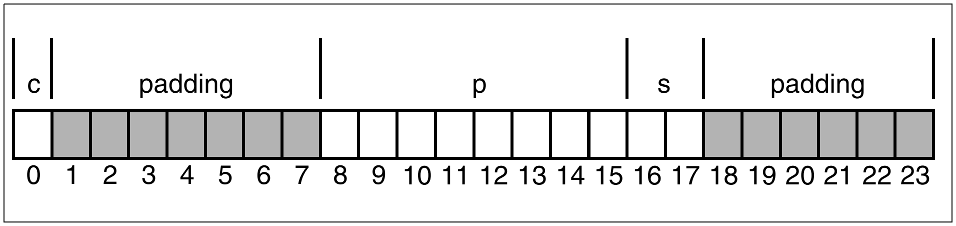


*Figure 6-2 Structure padding in 64-bit mode*

And imagine if this structure is shared or exchanged among 32-bit and 64-bit processes, the data fields (and paddding) of one environment will not match the expectations of the other.

To eliminate the difference, and to allow the structure to be shared, you can reorder the fields in the data structure to get the alignments in both 32-bit and 64-bit environments to match. However, this may not be possible in all cases. It depends a great deal on the data types used in the structure, and the way in which the structure as a whole is used (for example, whether the structure is used as a member of another structure or array).

If you are unable to reorder the members of a structure, or if reordering alone cannot provide correct alignment, you can introduce user-defined paddings to cause the members of the structure to fall on their natural boundaries. Depending on the data types involved, a conditional compile section may be necessary. A conditional compile section will be required when the structure uses data types that have different sizes in the 32-bit and 64-bit environments.

For example, if the structure layout of the Example 6-1 on page 237 is changed to the following:

```
struct T {
    char c;
    short s;
#if !defined(__64BIT__)
    char pad1[4];
#endif
    int *p;
#if !defined(__64BIT__)
    char pad2[4];
#endif
} t;
```

The structure will have the same size and member layout in both 32-bit and 64-bit environments:

```
sizeof(t) = 16
offsetof(t, c) = 0 sizeof(c) = 1
offsetof(t, s) = 2 sizeof(s) = 2
offsetof(t, p) = 8 sizeof(p) = 4
```

Note that this output is from a 32-bit execution. The size of the member p, sizeof(p), will be 8 in 64-bit mode. Figure 6-3 shows the member layout of the structure with user-defined padding.

**Important:** This example is for illustrative purposes only. Sharing pointers between 32-bit and 64-bit processes is *not* recommended and will likely yield incorrect results.

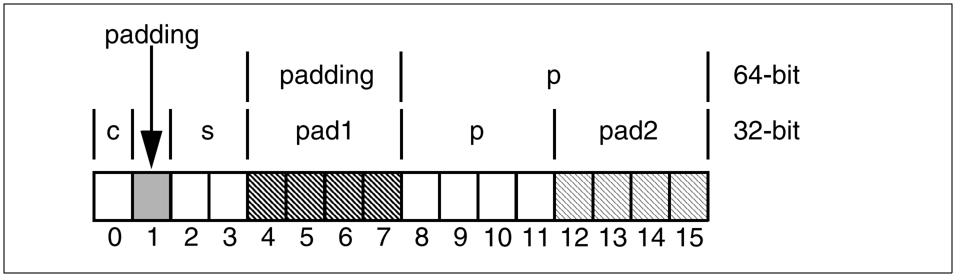


*Figure 6-3 Structure with user-defined paddings in both 32-bit and 64-bit mode*

When inserting paddings to structures, use an array of char to avoid any further alignment requirement on the paddings themselves. The natural alignment of a char is 1-byte, meaning it can reside anywhere in memory.

## 6.3 Diagnosing link-time errors

The C and C++ compiler drivers (see 1.7.1, “Default compiler drivers” on page 30) handle object file linking by invoking the system link editor **ld** with the appropriate options and flags. There is no need to use **ld** directly to link edit your C and C++ object files. This section describes some of the more common link-time errors that you may encounter during development.

### 6.3.1 Unresolved symbols

When linking your application with many libraries, particularly those supplied by a third party product, such as a database, it is not unusual during the development cycle to see a linker error warning about unresolved symbols.

The system link editor accepts input such as object files, shared object files, archive object files, libraries, and import and export files, resolves external symbol references, and creates a single executable object file that can be run. When an external symbol cannot be resolved, the link would fail and an executable is not generated.

Unresolved symbol errors can occur in many situations, the most common being missing input files. For example, if you call a library function that is not in the default C run-time library libc.a, you need to specify the archive library file in which the symbol is found:

```
$ cat test.c
#include <stdio.h>
#include <math.h>
void main()
{
    printf("%f\n", pow(2.0,3.0));
}
$ cc test.c
ld: 0711-317 ERROR: Undefined symbol: .pow
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
$ cc test.c -lm
```

This is especially true when using functions provided by third-party libraries. Aside from searching the product documentation, finding the library file in which an unresolved symbol is defined can be quite a daunting task. You can include all supplied libraries in the **link** command and let the link editor find out where the symbol is, or you may use the **nm** command to try find it yourself.

The linker supports options that can be used to generate linker log files. These log files can then be analyzed to determine the library or object file that references the unresolved symbol. This can help in tracking interdependent or redundant libraries being used in error.

The -bnoquiet option writes each binder sub command and its results to standard output. It gives a better of picture of the unresolved symbol references if any exists and also the list of symbols imported from the specified library modules.

The -bmap:*filename* option is used to generate an address map. Unresolved symbols are listed at the top of the file, followed by imported symbols.

The -bloadmap:*filename* option is used to generate the linker log file. It includes information on all of the arguments passed to the linker along with the shared objects being read and the number of symbols being imported. If an unresolved symbol is found, the log file produced by the -bloadmap option lists the object file or shared object that references the symbol. In the case of using libraries supplied by a third-party product, you can then search the other libraries supplied by the product in an effort to determine which one defines the unresolved symbol. This is particularly useful when dealing with database products that supply many tens of libraries for use in application development.

If errors show up during linking or loading in a version controlled application development environment, then try linking outside the version control system. If errors happen only when linking or loading under version control, then check with your version control system vendor for known linker/loader problems.

### 6.3.2 Duplicate symbols

Duplicate symbol errors usually indicate a programming error. It is invalid to have multiple external function definitions of the same name. The link editor in this case will use the first definition that it encounters, and the result may not be desirable. Change the name of the function or use the static function.

The use of template functions in C++ may generate duplicate symbol errors at link time. This happens when the template is implicitly instantiated in multiple source files. For example:

```
// t.h
template <class T> class A {
public:
    int f();
};

template <class T> int A<T>::f()
{
    return 55;
}

// file1.C
#include "t.h"
int func();
int main()
{
    A<int> a;
    int obj1 = a.f() + func();
    return obj1;
}
```

```
#include "t.h"
int func()
{
    A<int> a;
    int obj2 = a.f();
    return obj2;
}

$ xlC -c file1.C file2.C
file1.C:
file2.C:
$ xlC file1.o file2.o
ld: 0711-224 WARNING: Duplicate symbol:
.A<int>::f()
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

Using **nm**, you will note that the symbol A<int>::f() is defined in both object files:

```
$ nm -g file1.o
.A<int>::f()        T       100
.func()             U         -
.main               T         0
A<int>::f()         D       180      12
main                D       168      12
$ nm -g file2.o
.A<int>::f()        T        68
.func()             T         0
A<int>::f()         D       148      12
func()              D       136      12
```

In this case, since the duplicate symbol A<int>::f() has the same definition, it is safe to suppress the link editor message with -bhalt:5:

```
$ xlC -bhalt:5 file1.C file2.C
file1.C:
file2.C:
```

The -bhalt link editor option specifies the maximum error level for the linking process to continue, and suppresses any error below the maximum. The recommended solution to fixing this error is to use the -qtemplateregistry compiler option described in 10.5, “Template registry: The preferred method” on page 387, and to take advantage of the improved template handling provided by the VisualAge C++ for AIX Version 6 compiler. See the *VisualAge C++ for AIX Compiler Reference*, SC09-4959 for details on the option.

## 6.3.3 Insufficient memory for the linker process

The linker can run out of memory when trying to link very large files. When this happens, linking fails with the following error:

```
ld: 0711-101 fatal error, allocation of bytes fail in routine initsymtab_info.
There is not enough memory.
```

This may be because of low paging space or because of low resource limits for the user invoking the command. The AIX linker offers a great deal more functionality than traditional UNIX linkers, but it does require a reasonable amount of virtual memory, particularly when linking large applications with many libraries.

If this type of error is encountered, check the following and increase the space, if necessary:

- The available paging space on the machine.
- The resource limits for the user invoking the linker using the **ulimit** command (see 3.2.6, "Resource limits in 32-bit model" on page 125 for information on how to use the command).

If the problem is not solved by these activities, you may consider running the linker process in the 32-bit large memory model, as explained in 3.2.4, "Using the large and very large memory model" on page 121.

## 6.3.4 The c++filt utility

The C++ programming language allows function and operator overloading. An overloaded function is a function with the same name as a previously declared function, but with a different parameter list. To distinguish overloaded function names, the compiler uses name mangling to encode the names into unique names so that the link editor will not mistake them as duplicate symbols. The mangled name contains further information about the function and its parameters.

The c++filt utility is provided with the VisualAge C++ for AIX compiler product to convert these mangled names to their original source code names. With the -bloadmap:*filename* link editor option, function names appearing in the loadmap file, filename, are in mangled form. Use the c++filt utility to get the original function names:

```
ld: 0711-228 WARNING: Duplicate symbols were found while resolving symbols.
    The following duplicates were found:
 Symbol                    Source-File(Object) OR Import-File{Shared-object}
 ------------------------- -------------------------------------------------
 .f__1AXTi_Fv              file1.C(file1.o)
```

```
    ** Duplicate **        file2.C(file2.o)
```

The **c++filt** command waits for input from the standard input. If you type in mangled function names, it prints the original functions to the standard output:

```
$ c++filt
f__1AXTi_Fv
A<int>::f()
```

To quit the command, type ^D (Control-D).

# 6.4 Diagnosing run-time errors

Although your program may compile and link successfully, you may still encounter unexpected results during execution. Programming errors that do not violate the syntax of the language are not necessarily flagged by the compiler. This section describes some of the most common errors, how to detect them, and what you can do to rectify the problem.

## 6.4.1 Uninitialized variables

Unlike variables with static storage duration, according to the C and C++ programming language standards, an automatic (that is, stack allocated) storage duration object is not implicitly initialized, and its initial value is indeterminate. In other words, there is no guarantee that if an auto variable is used before it is set, it will yield the same result in every execution of the program. For example, when you first execute the following program, it may seem to be working correctly:

```
$ cat initauto.c
#include <stdio.h>
void func(int *p)
{
    if (p == NULL)
        printf("NULL pointer.\n");
    else
        *p = 0xdeadbeef;
}

void main()
{
    int *ptr;
    func(ptr);
}
$ cc initauto.c
$ a.out
NULL pointer.
```

However, there is still the potential that, given the right conditions, the program will start to fail and yield unexpected results.

The C and C++ compilers provide the -qinitauto option, which causes the compiler to generate extra code to initialize all automatic variables to a specified initial value. Due to the extra code that the option generates, it reduces the run-time performance of your program, and its use is only recommended for debugging only. For example, compile and execute the above program with -qinitauto=FE and you will receive a different result indicating a potential problem:

```
$ cc -qinitauto=FE initauto.c
$ a.out
Segmentation fault(coredump)
```

To find out where auto variables are used before being set, use the -qinfo=gen compiler option:

```
$ cc -qinfo=gen initauto.c
"initauto.c", line 13.10: 1506-438 (I) The value of the variable "ptr" may be
used before being set.
```

## 6.4.2 Run-time checking

The -qcheck option inserts run-time checking code into the program executable. The following checks are supported:

**NULL pointer** Checks if the pointers used in any pointer referencing have addresses greater than 512.

**Array bounds** Checks for array indexing to be within the array bounds, when the size of the array is known at compile time.

**Divide by zero** Checks for integer divide by zero.

Depending on the suboptions specified, if a violation occurs, a run-time SIGTRAP exception is raised. You can provide your own signal handler to handle the exception:

```
$ cat check.c
#include <stdio.h>

#ifdef DEBUG
#include <signal.h>
#define SIGNAL(sig, handler) signal(sig, handler)
void trap_handler(int sig)
{
    printf("SIGTRAP handled\n");
    exit(-1);
}
#else
```

```
#define SIGNAL(sig, handler) ((void)0)
#endif

void func(int *p)
{
    printf("p has address %p\n", p);
    *p = 0xdeadbeef;
}

void main()
{
    SIGNAL(SIGTRAP, trap_handler);
    func(NULL);
}
$ cc -DDEBUG -qcheck check.c
$ a.out
p has address 0
SIGTRAP handled
```

Similar to the -qinitauto option, the -qcheck option reduces the run-time performance of your program. It is recommended that you use it only for debugging.

### 6.4.3 Unsignedness preservation in C

If you use the **cc** command, specify the -qlanglvl=extended option, or the -qupconv option, and the C compiler will use unsignedness preservation in integral promotion. This was the semantic used prior to the c89 standard, also commonly referred to as K&R C.

Unsignedness preservation promotes unsigned integral types smaller than int, that is, unsigned char and unsigned short, to unsigned int. On the contrary, c89 and c99 mandate value preservation, where they are promoted to int. Unsignedness preservation may result in unexpected execution behavior. For example:

```
$ cat upconv.c
#include <stdio.h>
void main()
{
    unsigned char zero = 0;
    if (-1 < zero)
        printf("NOUPCONV: Value-preserving rules in effect \n");
    else
        printf("UPCONV: Unsignedness-preserving rules in effect \n");
}
$ cc upconv.c
$ a.out
```

```
UPCONV: Unsignedness-preserving rules in effect
$ cc -qnoupconv upconv.c
$ a.out
NOUPCONV: Value-preserving rules in effect
```

### 6.4.4 ANSI aliasing

The standard conforming language levels supported by the C and C++ compilers enforce a type-based aliasing rule during optimization. Type-based aliasing, sometimes referred to as ANSI-aliasing, restricts the lvalues[2] that can be safely used to access a data object. In essence, it mandates that a pointer can only point to an object of the same type. In other words, any pointer cast followed by a dereference violates this rule. The only exceptions are:

- The sign and type qualifiers are not subject to type-based aliasing
- A character pointer can point to any type.

For example, the following program gives different results when optimization is in effect:

```
$ cat ansialias.c
#include <stdio.h>

unsigned int rc;
unsigned int function( float *ptr )
{
    rc = 0;
    *ptr = 1;
    return rc;
}

void main()
{
    unsigned int x = function((float *)&rc);
    printf("x = %x rc = %x\n", x, rc);
}
$ c89 ansialias.c
$ a.out
x = 3f800000 rc = 3f800000
$ c89 -O ansialias.c
$ a.out
x = 0 rc = 3f800000
```

When optimization is turned on with -O, the compiler assumes that the pointer ptr cannot be pointing to the external variable rc due to type-based aliasing. Hence, it can return the value 0 to the caller because of the rc = 0 assignment.

[2] An lvalue is an expression with an object type. It is a locator value representing an object.

If you use the cc compiler driver, or specify the -qalias=noansi option, the compiler makes the worst case aliasing assumptions, and assumes that a pointer of a given type can point to an external object or any object whose address is already taken, regardless of type. This will correct the programming error in the above example. However, it also greatly reduces optimization opportunities and degrades run-time performance. It is therefore recommended that you change you program to conform to type-based aliasing rule.

### 6.4.5 #pragma option_override

Unfortunately, it may not be easy to fix all programming errors in a complex application, especially when the error only shows up when optimization is used. In this case it may be worth while to turn off optimization for the function where the programming error is, while the rest of the program still benefits from optimization.

Use the #pragma option_override directive to specify alternate optimization options for specific functions. In the example in 6.4.4, "ANSI aliasing" on page 247, adding the directive:

```
#pragma option_override(function, "opt(level,0)")
```

to the source will correct the programing error:

```
$ c89 -O ansialias.c
$ a.out
x = 3f800000 rc = 3f800000
```

The #pragma option_override directive is also useful in finding out, by a process of elimination, which function is causing the problem. By selectively turning off optimization for each function within the directive until the problem disappears, it will let you identify where the programming error resides.

# 7

# Debugging your applications

This chapter describes several methods for debugging C and C++ applications by providing the following sections:

- Section 7.1, “Working with core files” on page 250
- Section 7.2, “Using the printf()-debug method” on page 258
- Section 7.4, “Using dbx” on page 259
- Section 7.5, “Debugging with the truss command” on page 264
- Section 7.6, “Using the trace facility” on page 267

For further information about the debugging methods on AIX, please refer to *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 7.1 Working with core files

A core file is generated when various errors occur in an application and the process aborts. The reasons can be errors such as memory-address violations, illegal instructions, bus errors, and user-generated quit signals. The core file contains a memory image of the aborted process. For a detailed definition of the contents of a core file, please refer to *AIX 5L Version 5.2 Files Reference*.

Even if the application does not stop, it might be useful to have a core file as an image of the memory state of the process in a specific time. Core files can be used for debugging and examination by yourself on your system or by remote technical support specialists on their system. For the distribution of a core file, there is a special way to collect all needed information.

### 7.1.1 Core file naming

Before AIX 5L Version 5.1, a core file was always stored in a file named *core*. If the same or another application generated another core file before you renamed the previous one, the original content was lost.

Beginning with AIX 5L Version 5.1, you can enable a unique naming of core files, but be aware that the default behavior is to name the files core. You apply the new enhancement by setting the environment variable CORE_NAMING to a non-NULL value, for example:

```
CORE_NAMING=yes
```

After setting CORE_NAMING, you can disable this feature by setting the variable to the NULL value. For example, if you are using the Korn shell, do the following:

```
export CORE_NAMING=
```

After setting CORE_NAMING, all new core will be stored in files of the format core.*pid.ddhhmmss*, where:

| | |
|---|---|
| **pid** | Process ID |
| **dd** | Day of the month |
| **hh** | Hours |
| **mm** | Minutes |
| **ss** | Seconds |

In the following example, two core files are generated by a process identified by PID 30480 at different times:

```
$ ls -l core*
-rw-r--r--   1 ausres01 itsores        8179 Jan 28 2003  core.30480.28232347
-rw-r--r--   1 ausres01 itsores        8179 Jan 28 2003  core.30482.28232349
```

The time stamp used is in GMT[1] and your time zone will not be used.

## 7.1.2 Creating core files with assert()

The assert() macro, provided by the assert.h header file, is a common way to produce a core file, when you are sure where the application logic goes wrong. For example, if you are sure the integer variable cnt should not be greater than 100 in the certain point of your code, you can insert the following line into your source code:

```
#include <assert.h>
int func(void)
{
    ...
    /* You are sure that cnt should not be greater than 100 in here. */
    assert(cnt <= 100);
    ...
    /* Application logic continues. */
}
```

If the variable cnt is greater than 100 in the highlighted line in the above example, the assert() macro calls the abort() routine and generates a core file and the process dies.

## 7.1.3 Creating core files with coredump()

If you have an application behaving unexpectedly, you can have a core file without terminating the application process by adding the coredump() sub-routine in your source code. The generated core file, which contains the memory image of the process, can be used for debugging the problem with **dbx**.

In a multi-threaded process, only one thread at a time should attempt to call coredump(). Subsequent calls to coredump() while a core dump (initiated by another thread) is in progress will fail.

To use coredump(), you must compile your source code with the -bM:UR options; otherwise, the routine will fail with an error code of ENOTSUP.

[1] Greenwich Mean Time

The syntax of coredump() is:

```
#include <core.h>
int coredump(struct coredumpinfo *coredumpinfop)
```

By specifying a file name and its string length in the coredumpinfo structure parameter, you can specify any file name as the generating core file. For example, if we compile the source file shown in Example 7-1 with cc -bM:UR coredump.c, then run a.out, then the core file, mycore, will be generated as shown in the following example:

```
$ cc -bM:UR coredump.c
$ ./a.out; echo $?
0
$ ls -l mycore
-rw-r--r--   1 ausres01 itsores       8183 Feb 05 10:55 mycore
```

*Example 7-1 coredump.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <core.h>

int
main(int argc, char *argv[])
{
    int rc;
    struct coredumpinfo cdinfo;

    cdinfo.name = "mycore";
    cdinfo.length = strlen("mycore");

    if ((rc = coredump(&cdinfo)) == -1) {
        perror("coredump()");
    }

    exit(0);
}
```

For more information about coredump(), please refer to *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions*.

## 7.1.4 Including shared memory information in the core file

On AIX, by default, the detailed information about shared memory segments and thread stacks are not collected when generating core files. If you need to have core files that contain detailed information (called *full-core*), there are two ways to do so.

### System wide full-core setting

Use the **chdev** command to change the full core attribute setting system wide. The default value is not to take full core dump, as shown in the following example:

```
# lsattr -El sys0 -a fullcore
fullcore false Enable full CORE dump True
```

To change the setting, issue the **chdev -l sys0 -a fullcore=true** command as the root user.

### Add a signal handler with the SA_FULLDUMP flag

You can modify your application source code by adding a signal handler with the SA_FULLDUMP flag set for the signal that will cause the core file. To register a signal handler, use the sigaction().

> **Note:** The full-core setting is required to debug multi-threaded applications with most debugging methods.

## 7.1.5 Gathering core files

All associated information of a core file can be packed and archived in a pax file, which can be stored on disk or tape or sent to another system for investigation. At the time of the writing of this redbook, the Distributed Debugger only supports debugging of core files on the machine that created them.

### The snapcore command

Use the **snapcore** command to gather the core file, its program binary executable file, and its dependent library files. The syntax of the command is:

```
snapcore core_filename [program_filename]
```

Specify the full path names for core file and program file name. Without defining the program name, **snapcore** will read the program name out of the core file and search it in the directories defined by PATH.

The command will produce a compressed pax file in the /tmp/snapcore directory by default. Use the -d directory option to specify an alternative directory. In the following example, a core file generated by prog1 and its dependent library files are archived in a compressed pax file, /tmp/snapcore/core.31442.pax.Z:

```
$ snapcore core.31374.29164438 prog1
Core file "core.31374.29164438" created by "prog1"
pass1() in progress ....
                Calculating space required .
                Total space required is 6578 kbytes ..
                Checking for available space ...
                Available space is 119148 kbytes
pass1 complete.
pass2() in progress ....
                Collecting fileset information .
                Collecting error report of CORE_DUMP errors ..
Creating readme file ..
                Creating archive file ...
                Compressing archive file ....
pass2 completed.
Snapcore completed successfully. Archive created in /tmp/snapcore.
$ ls -l /tmp/snapcore
total 5960
-rw-r--r--   1 ausres01 itsores     3049565 Jan 29 10:52 snapcore_31442.pax.Z
```

You can check what files are gathered in the archive file, as shown in the following example:

```
$ uncompress -c /tmp/snapcore/snapcore_31442.pax.Z | pax
core.31374.29164438
README
lslpp.out
errpt.out
prog1
./usr/lib/libc.a
./usr/lib/libcrypt.a
./usr/ccs/lib/libc.a
```

For more information about the snapcore command refer to *AIX 5L Version 5.2 Reference Documentation: Commands Reference*.

### The check_core command

You can determinate the program that caused the core and the dependent libraries for an existing core file by using the **check_core**[2] command.

[2] The **check_core** command is included in the bos.rte.serv_aid fileset.

The following example shows that the core file, core.31374.29164438, was generated by the program file, prog1, and its dependent libraries:

```
$ /usr/lib/ras/check_core core.31374.29164438
/usr/lib/libc.a
/usr/lib/libcrypt.a
prog1
```

### 7.1.6 AIX error log entry

Each core dump creates a new entry in the AIX error log. It can be useful for identifying an application that dumps. Use the following command for examining all entries caused by an core dump:

```
errpt -aJ CORE_DUMP
```

Use the -s *mmddhhmmyy* option to filter error log entries starting after the given time (*mm* = month, *dd* = day, *hh* = hours, *mm* = minutes, *yy* = year).

If you use the -A option instead of -a, you will see a more condensed output. Example 7-2 shows that the program prog1 identified with PID 31242 generated a core file because of the signal 11 (SIGSEGV) delivery.

*Example 7-2 AIX error log: CORE_DUMP*

```
$ errpt -A -J CORE_DUMP -s 0129130003
-----------------------------------------------------------------------
LABEL:          CORE_DUMP
Date/Time:       Wed Jan 29 13:20:35 CST
Type:            PERM
Resource Name:   SYSPROC
Description
SOFTWARE PROGRAM ABNORMALLY TERMINATED
Detail Data
SIGNAL NUMBER
          11
USER'S PROCESS ID:
       31242
FILE SYSTEM SERIAL NUMBER
          13
INODE NUMBER
        2048
PROGRAM NAME
prog1
ADDITIONAL INFORMATION
main 10C
main F8
__start 8C
```

## 7.1.7 Lightweight core file support

Besides the standard core file format, you can use the lightweight core file format, which complies with the Parallel Tools Consortium Lightweight Core File Format. If a multi-threaded program crashes or hangs, it is nearly impossible to find out how far the execution got. Dumping out the memory information of many processors requires time and disk space. The resulting information in a standard core file is of little use for this case. The lightweight core file format is a platform independent format with a snapshot of the current location of an application. The lightweight core file is a high level, symbolic collection of the program state. It is an ASCII file, readable by humans and analysis programs. Running prog2 from Example 7-3 on page 257, we receive the lightweight core file named lwcore:

```
$ more lwcore
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0 Wed Feb 12 08:19:07 2003 Generated by IBM AIX 5.1
#
+++ID Node 0 Process 30820 Thread 1
***FAULT "SIGSEGV - Segmentation violation"
+++STACK
main : 0x0000004c
---STACK
---ID Node 0 Process 30820 Thread 1
---LCB
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0 Wed Feb 12 08:31:09 2003 Generated by IBM AIX 5.1
#
+++ID Node 0 Process 37548 Thread 1
***FAULT "SIGSEGV - Segmentation violation"
+++STACK
main : 0x0000004c
---STACK
---ID Node 0 Process 37548 Thread 1
---LCB
```

### Creating a lightweight core file with install_lwcf_handler()

The install_lwcf_handler() subroutine provides a lightweight core file instead of a standard core file when an application crashes. It is part of the PTools Library (libptools_ptr.a). There are two ways to create the lightweight core file:

- Call the install_lwcf_handler() subroutine directly in your application to register a signal handler (Example 7-3 on page 257).
- Use the linker option -binitfini:install_lwcf_handler, so that the function will be called by starting the program automatically. In this way, you do not have to change your application code.

The default file name for the lightweight core file is *lw_core*. Use the LIGHTWEIGHT_CORE environment variable to change it to your desired file name, or set it to STDERR to redirect the lightweight core file content to the standard error.

*Example 7-3 prog2.c with install_lwcf_handler() subroutine*

```
#include <stdlib.h>
#include <stdio.h>

void install_lwcf_handler (void);

main(int argc, char *argv[])
{
        int     i,j;
        char s[10];

        install_lwcf_handler();
        printf("I will start counting!\n");
        while (i >= j) {
                s[j] = j;
                j--;
                i++;
        }
        printf("My result is %s\n",s);
        exit(0);
}
$ cc prog2.c -o prog2 -lptools_ptr
$ export LIGHTWEIGHT_CORE="lwcore";
$ prog2
I will start counting!
$
```

Please refer to *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions* for more information.

## Creating a lightweight core file with mt_trce()

The mt_trce() subroutine provides a lightweight core file with the trace back information of all threads allocated in the process space. Threads, except for the calling thread, are suspended during the execution of the mt_trce() subroutine.

Refer to the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions* for a complete overview of this topic.

## 7.2 Using the printf()-debug method

The printf()-debug method is easy to understand and is still the most common method used to debug applications. To use the method, define the following macro in your codes:

```
#if !defined(DEBUG)
#define DBGMSG(MSGSTR) ((void)0)
#else
#define DBGMSG(MSGSTR) {\
        fprintf(stderr,"In %s() at line %d in file %s: %s"\
                , __FUNCTION__, __LINE__, __FILE__, (MSGSTR));\
        }
#endif
```

then call this macro where you would like to print messages, for example:

```
int main(int argc, char *argv[])
{
    DBGMSG("Hi there!\n");
    exit(0);
}
```

If you compile the source code with the -DDEBUG compiler option, then it will print a debug message very similar to the following line:

```
In main() at line 16 in file printf-debug.c: Hi there!
```

Although it is a very commonly used method in the early development cycle, it has the following disadvantages:

- You may need to recompile your source code every time you want to insert new printf()-debug lines.
- Debugging multi-threaded applications with the printf()-debug method is not trivial.
- Because it may cause many calls to printf(), the application performance get degraded in most cases.
- If you insert the DBGMSG macro too many times, it can be hard to trace the huge quantity of debug messages.

Function-like macros introduced by C99 are very useful when defining the debug macro (see “Function-like macros with variable arguments” on page 9).

## 7.3 Preparing your application for debugging

To fully exploit any debugger, you need to compile your application with the -g compiler option and without optimization. Otherwise, the debugger cannot resolve referenced symbols in your application, and you cannot refer to variables or functions by their name while debugging.

If you run the `strip` command to strip symbol information from your application program, the debugger  cannot resolve referenced symbols in your application either.

The latest C and C++ compilers on AIX support the following useful options:

**-qfullpath** This puts the full path of the source files into the debug information. This can be very useful if you have a very large source tree, since the debugger does not need to ask for the location of the source file.

**-qlinedebug** This eliminates the variable descriptions from the debug information. This is recommended if the code is optimized, since the variable monitoring in optimized code will generally give incorrect results. This option significantly reduces the size of the debug information, especially in C++ programs.

**-qdbxextra** This include debug information for all symbols in your program, regardless of whether they are referenced. Normally, only debug information for referenced symbols is included. This option will significantly increase the size of the resulting executable.

When debugging information is created in an application, the size of the executable may grow dramatically. A C++ executable may grow in size by a factor of 10 or more when debug information is generated. Intelligent use of -qlinedebug and turning off debug information in uninteresting areas of the code may help in this case.

## 7.4 Using dbx

The symbolic debugger, dbx, is the most commonly used debugger on UNIX operating systems, including AIX. The command provides many useful functions to debug application problems, including:

- Examines object and core files
- Provides a controlled environment for running a program

- Sets break points
- Traces program flow
- Supervises symbolic variables

Although the usage of dbx is well explained in Chapter 3, “Debugging Programs”, in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*, we describe the basic use of dbx and show useful customization examples.

> **Note:** dbx does not support lightweight core file.

## 7.4.1 Starting a dbx session

If an application aborted and generated a core file, run dbx as follows:

```
dbx ObjectFile CoreFile
```

If you do not specify the core file name, then the core file in the current directory is assumed to be the correct file. After starting a dbx session, it prints the sub-command prompt (dbx). The following example shows that the core file, core.30330.30144523, generated by prog1, is core dumped at line number 13 in main() because of the segmentation fault:

```
$ dbx prog1 core.30330.30144523
Type 'help' for help.
reading symbolic information ...
[using memory image in core.30330.30144523]

Segmentation fault in main at line 13
   13                   s[j] = j;
```

To exit from a dbx session, type `quit` on the dbx sub-command prompt:

```
(dbx) quit
```

### Attaching to a running process

If you need to debug a running process, for example, your application seems to run into an endless loop, you can attach a dbx session to the process by specifying the process ID (PID) with the -a option, as shown in the following example:

```
$ dbx -a 25952
Waiting to attach to process 25952 ...
Successfully attached to prog6.
Type 'help' for help.
reading symbolic information ...
stopped in main at line 15
```

```
   15          while (i >= j) {
(dbx) quit
[2] + Killed                   prog6 &
```

**Note:** If you type the quit sub-command on the sub-command prompt, the attached process will be killed as well as the dbx session. To exit from the dbx session without terminating the attached process, use the detach sub-command.

## 7.4.2 Customizing a dbx session

The **dbx** command reads in an initial configuration file, .dbxinit, in the following order:

1. The current directory
2. User's home directory

If the .dbxinit file is found in the current directory, the command does not read the one in the user's home directory. Or, you can explicitly specify the file name using the -c option.

By defining aliases of frequently used sub-commands or macros, you can customize your dbx session.

After starting the dbx session, you can also instruct **dbx** to dynamically read the configuration file by using the source sub-command as follows:

```
(dbx) source /home/ausres01/.dbxinit
```

The following settings are useful to customize your dbx session.

### Displaying more readable output of structure

Using the print sub-command, you can print the value of a variable in a dbx session. If you have to print complex C or C++ structures or unions, it could be difficult to read and interpret the resulting output. Use the following sub-command to display more readable output.

#### *Set $pretty=”on”*

Use the set $pretty=”on” sub-command to make your output more readable. Every value is print on its own line, with its scope value indented, as shown in the following example:

```
(dbx) set $pretty="off"
(dbx) print names[0]
(effort = (low = (0, 0), name1 = "Fabian", name2 = "Julian", name3 = "Manuel",
high = (
(a = (0), b = 0)
(a = (0), b = 0)
)), range = (0))
(dbx) set $pretty="on"
(dbx) print names[0]
{
    effort = {
        low[0] = 0
        low[1] = 0
        name1 = "Fabian"
        name2 = "Julian"
        name3 = "Manuel"
        high[0] = {
            a[0] = 0
            b = 0
        }
        high[1] = {
            a[0] = 0
            b = 0
        }
    }
    range[0] = 0
}
```

### *Set $pretty=”verbose”*

You can also use the set $pretty=”verbose” sub-command for other printing styles, with each value on its own line and with qualified names:

```
(dbx) set $pretty="verbose"
(dbx) print names[0]
effort.low[0] = 0
effort.low[1] = 0
effort.name1 = "Fabian"
effort.name2 = "Julian"
effort.name3 = "Manuel"
effort.high[0].a[0] = 0
effort.high[0].b = 0
effort.high[1].a[0] = 0
effort.high[1].b = 0
range[0] = 0
(dbx)
```

## Command line editor mode

You can set the dbx’s sub-command line editor mode to either vi or emacs by using the sub-command set edit [vi/emacs] or set -o [vi/emacs].

### 7.4.3 Working with breakpoints: The stop subcommand

The stop subcommand halts the application program when certain conditions are fulfilled. The program is stopped when:

- The Condition is true when the if Condition flag is used.
- The Procedure is called if the in Procedure flag is used.
- The Variable is changed if the Variable parameter is specified.
- The SourceLine line number is reached if the at SourceLine flag is used.

### 7.4.4 Redirection of library location in object files with the -p flag

If you examine a core file, **dbx** has to resolve all references for libraries and shared objects loaded by the application when the core file is generated. The information is kept in the loader section of the core file. All file names except the main executable module are treated as absolute path names. The **dbx** command uses this table, not the LIBPATH environment variable.

If you bring the core file to another system to examine it, the libraries can be missing or in another location. Starting with AIX 5L Version 5.2, the -p flag of **dbx** allows you to map the old library names to the new ones. You do not need to alter the core file.

In the following example, the ./libone.so library (listed as entry 5) is used, which is located in the current directory:

```
(dbx) map
Entry 1:
   Object name: prog8
   Text origin:     0x10000000
...
Entry 5:
   Object name: ./libone.so
   Text origin:     0xd00d7000
   Text length:     0x2fd
   Data origin:     0xf0323210
   Data length:     0x34
   File descriptor: 0x8

(dbx)
```

If the library is no longer at its original location, you will get an error upon the dbx session start:

```
$ dbx prog8 core
Type 'help' for help.
```

```
reading symbolic information ...dbx: fatal error: 1283-012 cannot open
./libone.so
```

To solve this problem, use the -p option to map the missing library to its new location, as shown in the following example:

```
$ dbx -p ./libone.so=./lib/libone.so prog8 core
Type 'help' for help.
[using memory image in core]
reading symbolic information ...

Segmentation fault in func8 at 0xd00d7158
0xd00d7158 (func8+0x30) 98640048        stb   r3,0x48(r4)
(dbx) map
Entry 1:
   Object name: prog8
...
Entry 5:
   Object name: ./lib/libone.so
   Text origin:     0xd00d7000
   Text length:     0x2fd
   Data origin:     0xf0323210
   Data length:     0x34
   File descriptor: 0x9
```

You can also put the mapping information in a file and call the -p flag as -p*filename*.

### 7.4.5 Using dbx with gcc

Starting with AIX 5L Version 5.2, **dbx** has been modified to support broad compatibility with the GNU C compiler (gcc), so the command now can be used to debug applications compiled with gcc.

To debug application compiled with **dbx**, you have to specify the -gxcoff option when building an executable file as follows:

```
$ gcc -gxcoff prog.c -o prog
```

## 7.5 Debugging with the truss command

The **truss** command traces library and system calls and signal activity for a process.[3] It allows you to see all the system calls being made by a process, the parameters passed by those calls, and any data or errors returned from those

[3] The **truss** command has been supported on AIX, starting from Version 5.1. Several enhancements have been made in AIX 5L Version 5.2.

calls. You can watch what your application is requesting from AIX and the results of those requests on *live*.

To demonstrate how it works, we present an application, which is composed of the four source files shown in Example 7-4.

*Example 7-4 Source codes of prog_to_truss*

```
/* main.c */
int main(int argc, char *argv[])
{
    func9_1(1);
    func9_3(2);
    exit(0);
}
/* sample9a.c */
void func9_1(int i)
{
    int     offset;

    offset = 10;
    func9_2(i);
    func9_3(i + offset);
}
/* sample9b.c */
void func9_2(int i)
{
    func9_3(i);
}
/* sample9c.c */
#include <stdio.h>
#include <stdlib.h>

void func9_3(int i)
{
    printf("I am here in func9_3 with %d !!\n", i);
}
```

We have built the application, prog_to_truss, as follows:

```
$ cc -G -o libs91.so sample9a.c
$ cc -G -o libs92.so sample9b.c
$ cc -G -o libs93.so sample9c.c
$ cc -brtl prog9.c  -L. libs91.so libs92.so libs93.so -o prog_to_truss
```

The prog_to_truss application contains an object module, main.o, and references to three run-time linking shared objects (see 2.5, “Run-time linking” on page 68).

By default, **truss** does not trace library calls. In the following example, we have specified the -t!__loadx option, which means that any calls of the system call __loadx will be excluded from the output. Because the program calls three functions in run-time linking shared objects, if we do not specify this option, the output becomes very unclear for our demonstration:

```
$ truss -t!__loadx prog_to_truss
execve("./prog_to_truss", 0x2FF229FC, 0x2FF22A04)  argc: 1
sbrk(0x00000000)                                = 0x2000050C
sbrk(0x00000004)                                = 0x2000050C
sbrk(0x00010010)                                = 0x20000510
kioctl(1, 22528, 0x00000000, 0x00000000)        = 0
I am here in func9_3 with 1 !!
kwrite(1, 0xF01B5168, 24)                       = 24
I am here in func9_3 with 11 !!
kwrite(1, 0xF01B5168, 25)                       = 25
I am here in func9_3 with 2 !!
kwrite(1, 0xF01B5168, 24)                       = 24
kfcntl(1, F_GETFL, 0x00000000)                  = 67110914
kfcntl(2, F_GETFL, 0xF01B5168)                  = 67110914
_exit(0)
```

In the following example, we have specified the -u’*’ option to display the tracing output of any library function calls:

```
$ truss -t!__loadx -u'*' prog_to_truss
execve("./prog_to_truss", 0x2FF229FC, 0x2FF22A04)  argc: 1
sbrk(0x00000000)                                = 0x2000050C
sbrk(0x00000004)                                = 0x2000050C
sbrk(0x00010010)                                = 0x20000510
->librtl.a:func9_1(0x1)
->librtl.a:func9_3(0x1)
kioctl(1, 22528, 0x00000000, 0x00000000)        = 0
I am here in func9_3 with 1 !!
kwrite(1, 0xF01B5168, 24)                       = 24
<-librtl.a:func9_3() = 18               0.000000
->librtl.a:func9_3(0xb)
I am here in func9_3 with 11 !!
kwrite(1, 0xF01B5168, 25)                       = 25
<-librtl.a:func9_3() = 19               0.000000
<-librtl.a:func9_1() = 19               0.000000
->librtl.a:func9_3(0x2)
I am here in func9_3 with 2 !!
kwrite(1, 0xF01B5168, 24)                       = 24
<-librtl.a:func9_3() = 18               0.000000
->librtl.a:exit(0x0)
kfcntl(1, F_GETFL, 0x00000000)                  = 67110914
kfcntl(2, F_GETFL, 0xF01B5168)                  = 67110914
_exit(0)
```

As shown in the high-lighted lines, func9_3() called three times with an integer parameter, 1 (0x1), 11 (0xb), and 2 (0x2). Apparently, these values are passed by the parent function, func9_2(), as shown in Example 7-4 on page 265.

For further information about the **truss** command and its enhancements, please refer to the *AIX 5L Differences Guide Version 5.2 Edition,* SG24-5765 and the *AIX 5L Version 5.2 Reference Documentation: Commands Reference*.

# 7.6 Using the trace facility

The trace facility is a flexible system monitoring service that supplies a stream of events. It is up to you what information you extract. Trace provides detailed information about system activity based on a collection of pre-defined system events. You can add new trace hooks into your application in order to examine the application behavior. By utilizing trace hooks, your application may be debugged without having core files.

Please refer to the “Analyzing Performance with the Trace Facility” section of the *AIX 5L Version 5.2 Performance Management Guide* for further information.

## 7.6.1 Introduction to trace

Events are the basics of every trace running. These events are compiled into kernel or application code, but are only traced if tracing is active. You activate tracing with the **trace** command or the trcstart() subroutine. Tracing is stopped with the **trcstop** command or the trcstop() subroutine.

When tracing is active, you suspend or resume it with the **trcoff** and **trcon** commands or with the trcoff() and trcon() subroutines.

During the tracing, the events are stored in the trace log file. The recorded events can be selected and formatted with the **trcrpt** command to be a human-readable report.

**Note:** Tracing may impact system performance, if many events are traced.

### Trace hooks

The events monitored by trace are the trace hooks. Each hook is assigned to a unique number. The trace hooks are defined in the /usr/include/sys/trchkid.h header file.

You can define additional user trace hooks used for tracing your application. The range for user defined hooks is between 0x010 and 0x0FF.

### Trace report

Once the trace events are stored in the log file, use the **trcrpt** command to produce a human-readable report.

## 7.6.2 Tracing an application on the command line

This is the most common and useful way of tracing user defined trace hooks in user applications.

### Defining the trace hook in the application

Before starting trace, you have to modify your application code. The trace interfaces in your application will be only active if the trace daemon is active and is collecting data.

We use the example source code shown in Example 7-5 to demonstrate a simple usage of trace hook in the application source code.

*Example 7-5 Sample source code using trchook (prog11.c)*

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned int d1,d2,d3,d4,d5;
    int a,b;

    trchook(0x02010000,d1,d2,d3,d4,d5); /* trace hook 0x020 */
    trchook(0x02110000,d1,d2,d3,d4,d5); /* trace hook 0x021 */

    a = 13;
    b = 3;
    trchook(0x02220000,(a+b),d2,d3,d4,d5);/* trace hook 0x022 */

    exit(0);
}
```

The trchook() subroutine has the following syntax:

```
void trchook(unsigned int HkWord, unsigned int d1
            , unsigned int d2, unsigned int d3
            , unsigned int d4, unsigned int d5);
```

where HkWord is an unsigned integer consisting of a hook ID (HkID), a hook type (Hk_Type), and two bytes of data from the calling program (HkData). A hook ID is a 12-bit value. For user programs, the hook ID may be a value from 0x010 to 0x0FF. Hook identifiers are defined in the /usr/include/sys/trchkid.h file. Hk_Type is a 4-bit value that identifies the amount of trace data to be recorded (see Figure 7-1).

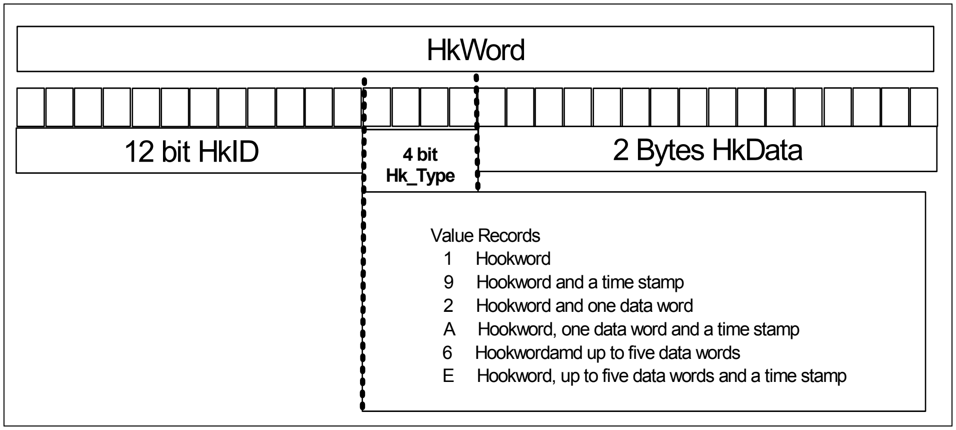

HkWord
12 bit HkID
4 bit
Hk_Type
2 Bytes HkData
Value Records
1 Hookword
9 Hookword and a time stamp
2 Hookword and one data word
A Hookword, one data word and a time stamp
6 Hookwordamd up to five data words
E Hookword, up to five data words and a time stamp

*Figure 7-1 Definition of HkWord*

## Defining trace templates

Prepare the trace report by naming the desired messages in the output by defining your user trace hooks (in our example, hook id 020, 021, and 022).

The trace templates for the **trcrpt** command are stored in the /etc/trcfmt file. Instead of using the default /etc/trcfmt file, we have made our own copy file, mytrcfmt, so that we can refer to this file by using the -t *filename* option.

In our example, we have inserted the following definition entries in mytrcfmt:

```
020 1.0 L=APPL "USER EVENT - HKWD_USER2" 02.0      \n \
               "Program runs over here"
021 1.0 L=INT  "USER EVENT - HKWD_USER2" 02.0      \n \
               "Simulatied Interrupt"
022 1.0 L=APPL "USER EVENT - HKWD_USER2" 02.0      \n \
               "Program runs over here value:" U4
```

Please refer to *AIX 5L Version 5.2 Files Reference* for the complete definition of trace templates.

## Tracing the application

We have compiled the program as follows:

```
$ cc prog11.c -o prog11
```

Then we start to trace it as follows:

```
$ trace
-> trcon
-> ! prog11
-> trcoff
-> q
```

When invoked, the **trace** command gives you the sub-command prompt “->”. The trcon sub-command starts the collection of trace hooks. We started the program, prog11, and stopped the collection with the trcoff sub-command.

### Reporting output of the trace events

The following example shows the trace report of newly defined trace hooks:

```
# trcrpt -d 020,021,022 -t mytrcfmt
Mon Feb  3 17:31:25 2003
System: AIX lpar02 Node: 5
Machine: 0021768A4C00
Internet Address: 09030442 9.3.4.66
The system contains 2 cpus, of which 2 were traced.
Buffering: Kernel Heap
This is from a 32-bit kernel.
Tracing all hooks.

trace


ID     ELAPSED_SEC DELTA_MSEC   APPL SYSCALL KERNEL  INTERRUPT

006    0.864997761 864.997761               TRACEBUFFER WRAPAROUND 0002
006    1.802352538 937.354777               TRACEBUFFER WRAPAROUND 0003
006    2.789985334 987.632796               TRACEBUFFER WRAPAROUND 0004
006    3.687272139 897.286805               TRACEBUFFER WRAPAROUND 0005
006    4.454988847 767.716708               TRACEBUFFER WRAPAROUND 0006
006    4.657498888 202.510041               TRACEBUFFER WRAPAROUND 0007
020    4.669032274*             USER EVENT - HKWD_USER2
                                Program runs over here
021    4.669032274*                                   USER EVENT - HKWD_USER2
                                                      Simulatied Interrupt
022    4.669032274*             USER EVENT - HKWD_USER2
                                Program runs over here value: 16
006    4.749999720 80.967446                 TRACEBUFFER WRAPAROUND 0008
006    5.695142879 945.143159                TRACEBUFFER WRAPAROUND 0009
005    5.695142879*                          LOGFILE WRAPAROUND 0001
006    6.534979135 839.836256                TRACEBUFFER WRAPAROUND 000A
002    6.609241540 74.262405                 TRACE OFF channel 0000 Mon Feb  3
17:31:33 2003
```

## 7.6.3 Tracing an application with subroutine calls

You can include the trace control, starting the trace, enable and resume data collection, as well as trace stopping completely inside the application.

### Defining the trace hook in the application

In the prog10.c sample, we use a predefined macro called TRCHKL1T (see Example 7-6). The TCHKL1T macro found in /usr/include/sys/trcmacros.h is defined to call the utrchook() subroutine, which is similar to the trchook() subroutine used in our previous example, but used for non generic hook events. The utrchook() subroutine is called by the macro with hook ID 0x010 and HType 2.

In Example 7-6, instead of using trcon() and trcoff() subroutines, we use ioctl() calls for the trcon and trcoff functions. trcon() and trcoff() subroutines perform the following tasks:

1. Opens the trace control device (/dev/systrctl).
2. Issues the appropriate ioctl() subroutine.
3. Closes the control device.
4. Returns to the calling program.

So, using the ioctl()-calls directly tunes the trace, because we do not need additional I/O and system calls.

*Example 7-6 Sample source code using TRCHKL1T (prog10.c)*

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/trcctl.h>
#include <sys/trcmacros.h>
#include <sys/trchkid.h>

#include <sys/trchkid.h>
char *ctl_file = "/dev/systrctl";
int ctlfd;
int i;

main(int argc, char *argv[])
{
    printf("Configuring trace collection \n");
    if (trcstart("-ad")) {
        perror("trcstart");
        exit(1);
    }

    printf("Opening the trace device  \n");
    if ((ctlfd = open(ctl_file, 0)) < 0) {
        perror(ctl_file);
        exit(1);
    }

    printf("Turning  trace on \n");
    if (ioctl(ctlfd, TRCON, 0)) {
        perror("TRCON");
        exit(1);
    }

    for (i=1;i <6; i++) {
        TRCHKL1T(HKWD_USER1, i);
    }

    printf("Turning trace off\n");
    if (ioctl(ctlfd, TRCSTOP, 0)) {
        perror("TRCOFF");
        exit(1);
    }

    printf("Stopping the trace daemon \n");
    if (trcstop(0)) {
        perror("trcstop");
        exit(1);
    }

    exit(0);
}
```

We have compiled the application as follows:

```
$ cc prog10.c -o prog10 -lrts
```

## Tracing the application

When the application starts, it automatically starts and stops the trace:

```
$ prog10
Configuring trace collection
Opening the trace device
Turning  trace on
Turning trace off
Stopping the trace daemon
```

## Reporting the output of the trace events

After defining the template for hook ID 0x010, we can generate the trace report, as shown in Example 7-7 on page 273.

*Example 7-7 Trace report for the hook ID 010*

```
$ trcrpt -d 010 -t mytrcfmt

Mon Feb  3 18:20:06 2003
System: AIX lpar02 Node: 5
Machine: 0021768A4C00
Internet Address: 09030442 9.3.4.66
The system contains 2 cpus, of which 2 were traced.
Buffering: Kernel Heap
This is from a 32-bit kernel.
Tracing all hooks.

/usr/sbin/trace -ad


ID     ELAPSED_SEC     DELTA_MSEC   APPL    SYSCALL KERNEL  INTERRUPT

001    0.000000000       0.000000                   TRACE ON channel 0
                                                    Mon Feb  3 18:20:06 2003
010    0.000004057       0.004057   USER EVENT - HKWD_USER1
                                    The # of loop iterations =1
                                    The elapsed time of the last loop =
010    0.000005344       0.001287   USER EVENT - HKWD_USER1
                                    The # of loop iterations =2
                                    The elapsed time of the last loop = [1 usec]
010    0.000006565       0.001221   USER EVENT - HKWD_USER1
                                    The # of loop iterations =3
                                    The elapsed time of the last loop = [1 usec]
010    0.000007785       0.001220   USER EVENT - HKWD_USER1
                                    The # of loop iterations =4
                                    The elapsed time of the last loop = [1 usec]
010    0.000009020       0.001235   USER EVENT - HKWD_USER1
                                    The # of loop iterations =5
                                    The elapsed time of the last loop = [1 usec]
002    0.000043623      0.034603                    TRACE OFF channel 0000 Mon Feb  3 18:20:06
2003
```

For more information about the trace facility, please refer to the “Trace Facility” section of *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs* and the “Analyzing Performance with the Trace Facility” section of *AIX 5L Version 5.2 Performance Management Guide*. See the *AIX 5L Version 5.2 Files Reference* for the trcfmt file format.

# 8

# Introduction to POSIX threads

This chapter gives you an introduction to parallel programming on AIX using POSIX threads by providing several example programs that explain the following concepts:

- Creating and joining POSIX threads
- Mutexes (mutual exclusive locks)
- Condition variables
- Read-write locks
- Thread-specific data
- Thread scheduling
- Environment variables

For further information about POSIX thread programming on AIX, please refer to the following sections in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*:

- “Parallel Programming”
- “Programming on Multiprocessor Systems”
- “Threads Programming Guidelines”

## 8.1 Overview of threads

A thread is a single, sequential flow of control within a process. Within each thread, there is a single point of execution. On most UNIX operating systems, including AIX, a thread possesses the following characteristics:

- It has its own independent flow of control within a process.
- It shares resources with other threads within a process.
- It can voluntary terminate before the process termination, or all the threads within a process terminate when the process terminates.

Most traditional programs execute as a process with a single thread. From a programmer's point of view, a thread can be considered as a procedure that can concurrently and independently run from the main processing flow.

### 8.1.1 Relationship between a process and a user thread

A process is a running program instance, which contains the following resources:

- Address space, including program text, shared library, global data, and so on.
- File descriptors
- Signal handlers
- Environment (variable) and working directory
- Set of identifiers (PID, GID, and so on)
- Inter-process communication (IPC) facilities, such as message queues, pipes, semaphores, and shared memory segments.

Multiple user threads can exist within a process and use these process resources, yet are able to be scheduled by the operating system and run as independent entities within a process. A user thread can posses an independent flow of control and can be scheduled because it maintains its own resources:

- Priority
- Program counter, stack pointer, and other registers
- Stack
- Scheduling priorities
- Signal mask
- Thread ID
- errno[1]

[1] See "Use of errno in multi-threaded programming" on page 283.

Figure 8-1 illustrates the relationship between a process and two user threads.

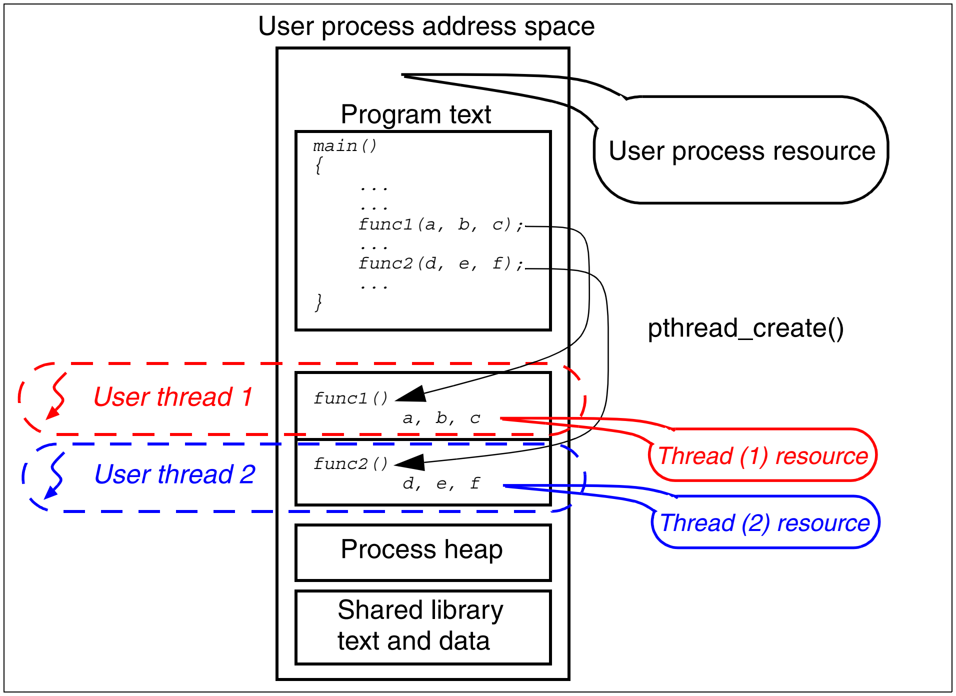


*Figure 8-1 Two user threads in a process*

A process can have multiple user threads, all of which share the resources within a process and all of which execute within the same address space. At any given point in time, there are multiple points of execution within a multi-threaded process.

Because user threads within the same process share resources, changes made by one user thread to a process resource will be transparently seen by all other user threads. Also, because multiple user threads can read and write the same memory locations within a process address space, the synchronization of data must be explicitly taken among these threads.

The cost to create or manage user threads is relatively cheap compared with processes in terms of CPU cycles. Also, the creation of a user thread requires a very small amount of memory compared with a process creation. With careful design and coding, the use of user threads gives programmers the ability to write multi-threaded applications that run on both uni- and multi-processor systems, taking advantage of the additional processors on SMP systems. Additionally, multi-threaded applications can increase performance even in a uniprocessor environment when the application performs operations that are likely to block or cause delays, such as file or socket I/O.

## 8.2 POSIX threads (Pthreads) on AIX

AIX supports the following standards:

- The Single UNIX Specification Version 2

  The specification includes the POSIX thread standard, known as the IEEE POSIX 1003.1c standard.[2] Thread implementations that conform to this standard are referred to as POSIX threads or Pthreads.[3]

- The Open Group UNIX 98 specification[4]

  The specification defines many additional functions on the former UNIX 95 specification, as well as extended threads functions over POSIX threads, based on industry input from major UNIX vendors.

On AIX, POSIX threads are defined as a set of C language programming types and sub-routine calls, implemented with a header file (/usr/include/pthread.h) and the POSIX thread library (/usr/lib/libpthreads.a).

AIX provides binary compatibility for existing multi-threaded applications that were written for the draft of Version 7 of the POSIX threads standard. The compatibility POSIX thread library, /usr/lib/libpthreads_compat.a, is only provided for backward compatibility for those applications. The compatibility POSIX thread library supports 32-bit applications only.

Both libraries are included in the bos.rte.libpthreads fileset, which is installed by default, as shown in the following example:

```
# lslpp -w /usr/lib/libpthreads*.a
  File                                        Fileset               Type
  ----------------------------------------------------------------------------
  /usr/lib/libpthreads.a                      bos.rte.libpthreads   Symlink
  /usr/lib/libpthreads_compat.a               bos.rte.libpthreads   Symlink
```

[2] AIX has been supporting the IEEE POSIX 1003.1c standard since Version 4.3. Later, it was included in the Single UNIX Specification Version 2, which AIX has been supporting since Version 5.1.
[3] Currently, AIX does not support the POSIX thread option *real-time extension*.
[4] AIX has been supporting the UNIX 98 specification since Version 4.3.

## 8.2.1 Advantages of using Pthreads

The following are the advantages of using Pthreads:

- The primary purpose of using Pthreads is to realize potential performance gains. POSIX threads can be created and managed with much less operating system overhead and system resources than creating and managing a process.
- Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication.
- Multi-threaded applications offer potential performance gains over non-threaded applications in several other ways:
  - Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one Pthread is waiting for an I/O system call to complete, CPU intensive work can be performed by other Pthreads.
  - Priority/Real-time scheduling: High priority jobs can be scheduled to supersede or interrupt lower priority jobs.
  - Asynchronous event handling: Jobs that service events of indeterminate frequency and duration can be interleaved. For example, a Web server can both transfer data from previous requests and manage the arrival of new requests.
- POSIX threads provide the infrastructure to parallelize applications using OpenMP, as explained in Chapter 9, “Program parallelization using OpenMP” on page 335.

## 8.2.2 The POSIX threads API

The POSIX threads API is defined in the ANSI/IEEE POSIX 1003.1 standard. All identifiers in the Pthreads library begin with *pthread_*. The POSIX threads API contains over 60 subroutines. Brief descriptions about the POSIX threads sub-routines supported on AIX are provided in Appendix D, “Subroutine references for POSIX threads” on page 473.

### Naming conventions

All the POSIX threads sub-routines are categorized into several groups distinguished by the following function name prefixes:

|  |  |
|---|---|
| **pthread_** | Threads and miscellaneous subroutines (see Table D-1 on page 474 and Table D-6 on page 481) |
| **pthread_attr_** | Thread attributes objects (see Table D-2 on page 476) |
| **pthread_mutex_** | Managing mutexes (see Table D-3 on page 477) |

| | |
|---|---|
| **pthread_mutexattr_** | mutex attribute objects (see Table D-3 on page 477) |
| **pthread_cond_** | Condition variables (see Table D-3 on page 477) |
| **pthread_condattr_** | Condition attribute objects (see Table D-3 on page 477) |
| **pthread_rwlock_** | Read/Write lock objects (see Table D-5 on page 480) |
| **pthread_rwlockattr_** | Read/Write lock attributes (see Table D-5 on page 480) |
| **pthread_key_** | Thread-specific data keys (see Table D-4 on page 479) |

### 8.2.3 Multi- and single-threaded processes

On AIX, the scheduling entity is a kernel thread. A single kernel thread can be mapped to multiple user threads (Pthreads) within a process (detailed information about the relationship between kernel threads and user threads is provided in 8.7.1, "Thread models in AIX" on page 322).

In order to illustrate how multi-threaded applications run on AIX, we have excerpted several lines from the `ps -emo THREAD` command output, as shown in Example 8-1 on page 281. The -o THREAD option instructs the `ps` command to display thread-level information; without this option, the command displays process-level information only.

Each column in the output represents the following:

| | |
|---|---|
| **USER** | The login name of the process owner. |
| **PID** | The process ID of the process. |
| **PPID** | The process ID of the parent process. |
| **TID** | The thread ID of the kernel thread. |
| **S** | The state of the process or kernel thread. |
| **C** | The CPU utilization of the process or kernel thread. |
| **PRI** | The priority of the process or kernel thread. |
| **SC** | The suspend count of the process or kernel thread. |
| **WCHAN** | The wait channel of the process or kernel thread. |
| **FLAG** | The flags of the process or kernel thread. |
| **TTY** | The controlling terminal of the process. |
| **BND** | The CPU to which the process or kernel thread is bound. |
| **CMD** | The command being executed by the process. |

*Example 8-1 ps -emo THREAD*

```
# ps -emo THREAD
    USER   PID  PPID    TID ST  CP PRI SC    WCHAN        F     TT BND COMMAND
    root     1     0      - A    0  60  1        -   200003      -   - /etc/init
       -     -     -    259 S    0  60  1        -   410410      -   - -
... (some lines are omitted) ...
    root 10432 13686      - A    0  60 11 f015ab98   240001      -   - /usr/sbin/rpc.mountd
       -     -     - 19881 S    0  60  1 f015ab98   c10400      -   - -
       -     -     - 23811 Z    0  60  1        -   c00001      -   - -
       -     -     - 49449 Z    0  60  1        -   c00001      -   - -
       -     -     - 53961 Z    0  60  1        -   c00001      -   - -
       -     -     - 54215 Z    0  60  1        -   c00001      -   - -
       -     -     - 55603 S    0  60  1        -   418400      -   - -
       -     -     - 60085 Z    0  60  1        -   c00001      -   - -
       -     -     - 60705 Z    0  60  1        -   c00001      -   - -
       -     -     - 62039 Z    0  60  1        -   c00001      -   - -
       -     -     - 63481 Z    0  60  1        -   c00001      -   - -
       -     -     - 63955 Z    0  60  1        -   c00001      -   - -
... (some lines are omitted) ...
    root 16266 13686      - A    0  60  1 c0042100   240001      -   - /usr/sbin/qdaemon
       -     -     - 22197 S    0  60  1 c0042100    10400      -   - -
... (rest of lines are omitted) ...
```

In this example, the rpc.mountd daemon process (PID 10432) is a multi-threaded process, since it contains 11 lines underneath the highlighted line, while the qdaemon daemon process (PID 16266) is a single-threaded process, since it has only one line that represents the kernel thread for the process's initial thread.

Two kernel threads (TID 19881 and 55603) of PID 10432 shown the status S, which means the kernel thread is in the sleeping status. Other kernel threads did not exist at the time the **ps** command was invoked, though TIDs were displayed.

the process could be verified by inspecting the /proc file system.[5] The /proc file system is a system interface to represent process information. The process information for the process <PID> is shown as several files and sub-directories under the /proc/<PID> directory.

As shown in Example 8-2 on page 282, there were only two sub-directories, 19881 and 55603, under the /proc/10432/lwp directory[6] at the moment the **ps** command was invoked. The sub-directory names, 19881 and 55603, were the same TIDs for the PID 10432 in Example 8-1.

[5] AIX has been supporting the /proc file system since AIX 5L Version 5.1.

[6] LWP stands for light-weight process. In the /proc file system semantic, kernel threads are represented as LWPs.

*Example 8-2 Inspecting the /proc file system*

```
# ls /proc/10432
as       ctl      fd/      map      psinfo   status
cred     cwd@     lwp/     object/  sigact   sysent
# ls /proc/10432/lwp
19881/  55603/
# ls -lR /proc/10432/lwp
total 0
dr-xr-xr-x   1 root     system            0 Feb 21 18:41 19881/
dr-xr-xr-x   1 root     system            0 Feb 21 18:41 55603/
/proc/10432/lwp/19881:
total 0
--w-------   1 root     system            0 Feb 21 18:41 lwpctl
-r--r--r--   1 root     system          120 Feb 21 18:41 lwpsinfo
-r--------   1 root     system         1200 Feb 21 18:41 lwpstatus

/proc/10432/lwp/55603:
total 0
--w-------   1 root     system            0 Feb 21 18:41 lwpctl
-r--r--r--   1 root     system          120 Feb 21 18:41 lwpsinfo
-r--------   1 root     system         1200 Feb 21 18:41 lwpstatus
```

For further information about the /proc file system, please refer to the *AIX 5L Version 5.2 Files Reference*.

Although the rpc.mountd daemon process (PID 10432) was easily determined to be a multi-threaded process in Example 8-1 on page 281, in order to determine if the qdaemon daemon process (PID 16266) is non-threaded, use the **ldd**[7] command against the executable file, as shown in Example 8-3. Because qdaemon does not depend on the POSIX thread library, we are sure that it is not a multi-threaded program.

*Example 8-3 Inspecting an executable file using ldd*

```
# ldd /usr/sbin/qdaemon
/usr/sbin/qdaemon needs:
         /usr/lib/libc.a(shr.o)
         /usr/lib/libqb.a(shr.o)
         /unix
         /usr/lib/libcrypt.a(shr.o)
```

### Initial thread

When a process is created, one user thread is automatically created. This user thread is called the *initial* thread. It ensures the compatibility between the

[7] The **ldd** command is supported on AIX 5L Version 5.2 and later. On earlier versions of AIX, use the **dump -H** command.

non-threaded processes with a unique implicit Pthread and the multi-threaded processes. The initial thread has some special properties, not visible to the programmer, that ensure binary compatibility between the non-threaded processes and the multi-threaded operating system. It is also the initial thread that executes the main function in multi-threaded programs.

## Use of errno in multi-threaded programming

In the multi-process UNIX programming semantic, a system call or sub-routine would set a non-zero value to the global variable *errno* in case of failure. This is still true in multi-threaded programming, though there is a subtle difference.

Within a multi-threaded process, each Pthread has its own errno to avoid being overwritten by the other Pthreads. On AIX, the Pthread-basis errno is implemented as a macro to a function pointer to an internal function, as shown in Example 8-4, which is excerpted from /usr/include/errno.h.[8] In the multi-threaded programming on AIX, the _THREAD_SAFE macro is always defined.

*Example 8-4 Definition of errno*

```
#if defined(_THREAD_SAFE) || defined(_THREAD_SAFE_ERRNO)
/*
 * Per thread errno is provided by the threads provider. Both the extern int
 * and the per thread value must be maintained by the threads library.
 */
extern  int *_Errno( void );
#define errno   (*_Errno())
#else
extern int errno;
#endif  /* _THREAD_SAFE || _THREAD_SAFE_ERRNO */
```

Although the process level global symbol errno is still accessible from Pthreads within a process, references to it is unreliable and useless in the multi-threaded programming.

Therefore, all the multi-threaded applications that reference to errno must have the following directive:

```
#include <errno.h>
```

and must not have the following declaration:

```
extern int errno;
```

[8] In the POSIX thread standard, the Pthread-basis errno is defined as implementation-dependent.

**Note:** Avoid directly referencing the internal function, because this is implementation-dependent and may be subject to change in future versions of AIX. Use the errno macro to assure the portability of your multi-threaded applications.

# 8.3 Pthread management

This section explains some of the basic Pthread management routines used for creating, joining, exiting, and detaching Pthreads.

## 8.3.1 Creating and terminating Pthreads

In order to create new Pthreads (besides the initial thread), the program must call the pthread_create() sub-routine. Example 8-5 is our first multi-threaded sample.

*Example 8-5 create_5threads.c*

```
#include <pthread.h>                                                 /* #A */
#include <stdio.h>
#include <stdlib.h>
#define NUM_OF_THREADS     6
#define EXIT_CODE -1

void *thr_func(void *id)
{
    int j;

    for (j = 0; j < 500000; j++) {
        ;                                                            /* #B */
    }
    printf("Hello world from Pthread %d!\n", (int *)id);
    pthread_exit(NULL);                                              /* #C */
}

int main (int argc, char *argv[])
{
    int rc, i;
    pthread_t tid[NUM_OF_THREADS];

    for (i = 1; i < NUM_OF_THREADS; i++) {
        printf("Creating Pthread: %d\n", i);
        rc = pthread_create(&tid[i], NULL, thr_func, (void *)i);    /* #D */
        if (rc != 0) {
            fprintf(stderr
                , "pthread_create() failed with rc = %d at %d in %s.\n"
```

```
                    , rc, __LINE__, __FILE__);
                exit(EXIT_CODE);
            }
        }
        pthread_exit(NULL);
}
```

This program creates five Pthreads, as illustrated in Figure 8-2.

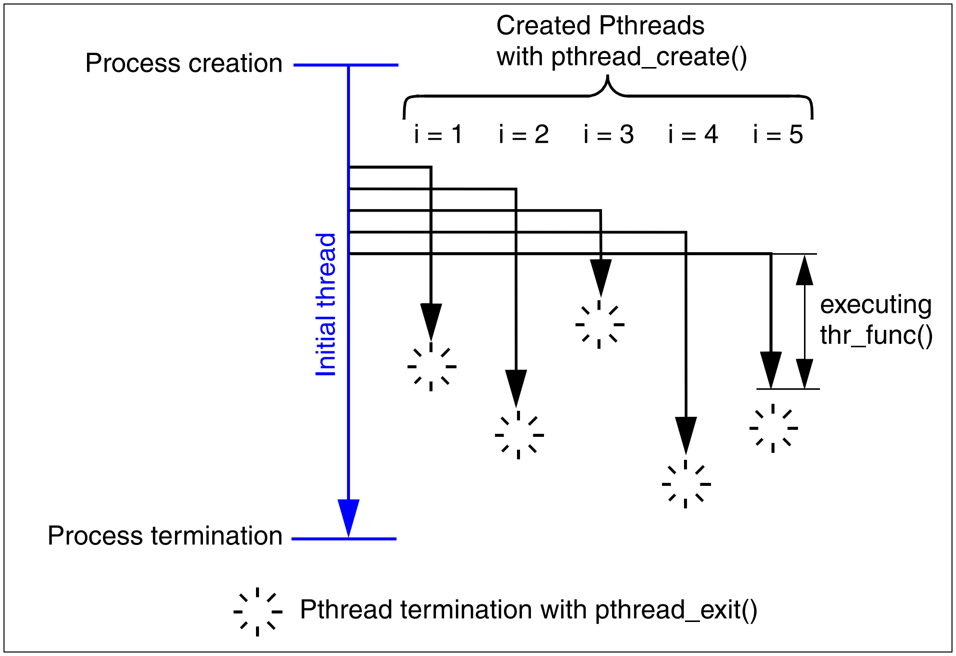


*Figure 8-2 Five Pthreads created by pthread_create()*

The highlighted lines with comments #A, #B, #C, and #D in Example 8-5 on page 284 explain the following important programming manners when developing multi-threaded applications:

- Comment #A

  To use the Pthread subroutines, the pthread.h header file must be included as the *first* header file in the each source file using the Pthreads library. This is because it defines some important macros that affect other system header files. Having the pthread.h header file as the first included file ensures the usage of thread-safe subroutines.

- Comment #B

  A Pthread's life-cycle can be very short; it may terminate just after it is created. In this example, a delay loop commented as #B is necessary to demonstrate that created Pthreads run in parallel. If this loop is removed or the program is compiled with optimizing options, all the created Pthreads could terminate before other Pthreads could be created.

  The programmer must be aware of this volatile nature of Pthreads.

- Comment #C

  If a Pthread calls pthread_exit(), it terminates; this marks the end of Pthread's life-cycle. Thread resources for the Pthread will be freed after the termination. However, other resources created by the terminating Pthread, such as file descriptors and sockets, will not be freed.

- Comment #D

  Most Pthreads library sub-routines return an integer value, which is interpreted as an error code. A value of zero indicates that the call was successful. Other values are passed or retrieved through appropriately typed arguments.

Also, the initial thread executing the main() function could have executed other functions after creating the other Pthreads. Therefore, there were six independent control flows in the process, as depicted in Figure 8-2 on page 285.

## Compiling multi-threaded programs

To compile multi-threaded programs, use one of the compiler drivers with the _r suffix (see Table 1-4 on page 30). To compile the program shown in Example 8-5 on page 284, we used **cc_r**, as shown in the following example:

```
$ cc_r create_5threads.c
```

> **Note:** The POSIX thread library is automatically linked when programs are compiled with _r compiler drivers. Therefore, the -lpthreads linker option is not required.

The created executable file, a.out, is a 32-bit multi-threaded program, as shown in the following:

```
$ file a.out
a.out:          executable (RISC System/6000) or object module not stripped
$ ldd a.out
a.out needs:
         /usr/lib/threads/libc.a(shr.o)
         /usr/lib/libpthreads.a(shr_comm.o)
         /usr/lib/libpthreads.a(shr_xpg5.o)
         /unix
```

```
        /usr/lib/libcrypt.a(shr.o)
```

### Running multi-threaded programs

We ran this application on an AIX 5L Version 5.2 partition on the pSeries 690, which is assigned two processors:

```
# lsdev -Cc processor
proc0 Available 00-00 Processor
proc1 Available 00-01 Processor
```

At the first invocation of this program, it produced the following output:

```
$ a.out
Creating Pthread: 1
Creating Pthread: 2
Creating Pthread: 3
Hello world from Pthread 1!
Hello world from Pthread 2!
Hello world from Pthread 3!
Creating Pthread: 4
Creating Pthread: 5
Hello world from Pthread 4!
Hello world from Pthread 5!
```

At the second invocation of this program, it produced the following output:

```
$ a.out
Creating Pthread: 1
Creating Pthread: 2
Creating Pthread: 3
Creating Pthread: 4
Creating Pthread: 5
Hello world from Pthread 2!
Hello world from Pthread 4!
Hello world from Pthread 5!
Hello world from Pthread 1!
Hello world from Pthread 3!
```

Apparently, the execution order of Pthreads is different between these two outputs. In fact, it could be any order and you cannot predict which Pthread is scheduled first in a multi-threaded process.

## Passing arguments to Pthreads

We discuss several aspects of the pthread_create() sub-routine in the following sections:

### *Syntax*

```
#include <pthread.h>
int pthread_create (thread, attr, start_routine (void), arg)
pthread_t     *thread;
const         pthread_attr_t *attr;
void          **start_routine (void);
void          *arg;
```

### *Parameters*

**thread** Points to where the thread ID will be stored.

**attr** Specifies the thread attributes object to use in creating the thread. If the value is NULL, the default attributes values will be used.

**start_routine** Points to the routine to be executed by the thread.

**arg** Points to the single argument to be passed to the start_routine routine.

### *Return values*

If successful, the pthread_create function returns zero. Otherwise, an error number is returned to indicate the error.

### *Error codes*

The pthread_create function will fail if:

**EAGAIN** If WLM[9] is running, the limit on the number of threads in the class may have been met.

**EINVAL** The value specified by attr is invalid.

**EPERM** The caller does not have the appropriate permission to set the required scheduling parameters or scheduling policy.

To pass multiple arguments the start_routine, create a structure that contains all of the arguments, and then pass a pointer to the structure as the last parameter of the pthread_create() routine.

[9] Work Load Manager

For example, assuming that the following structure is defined:

```
struct thread_data {
   int  thread_id;
   char *msg;
} thread_data_array[NUM_OF_THREADS];
```

then pass the structure to the pthread_create() routine as follows:

```
pthread_create(&tid[t], NULL, thr_func, (void *)&thread_data_array[i]);
```

## 8.3.2 Joining Pthreads

The pthread_join() subroutine will block the calling Pthread until the Pthread you specify has terminated, and then, optionally, store the terminated Pthread's return value. Calling pthread_join() detaches the specified Pthread automatically.

When a Pthread is created, one of its attributes defines whether it is joinable or detached. Detachable means that it cannot be joined. To explicitly create a Pthread as joinable or detached, the second argument in the pthread_create() routine is used. The pthread_detach() routine can be used to explicitly detach a Pthread even though it was created as joinable. The following example demonstrates how to wait for Pthread completions by using the pthread_join() routine. The Pthreads in this example are explicitly created in a joinable state so that they can be joined later:

```
/* File: pthread-join.c*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_OF_THREADS 3
#define EXIT_CODE -1

void *Func_Join(void *t)
{
   printf("The argument is %d\n", t);
   pthread_exit((void *) 0);
}

int main(int argc, char *argv[])
{
    pthread_t thread[NUM_OF_THREADS];
    pthread_attr_t attr;
    int rc, t, status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    /*Creates a thread as Joinable*/
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for (t = 0; t < NUM_OF_THREADS; t++) {
        printf("Creating thread %d\n", (t+1));
        rc = pthread_create(&thread[t], &attr, Func_Join, (void*)t);
        if (rc) {
            fprintf(stderr, "pthread_create() failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
    }

    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for (t = 0; t < NUM_OF_THREADS; t++) {
        rc = pthread_join(thread[t], (void **)&status);
        if (rc) {
            fprintf(stderr, "pthread_join() failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
        printf("Joining the thread %d \n",(t+1));
    }
    pthread_exit(NULL);
}
```

When executed, the above program printed the following output on our system:

```
Creating thread 1
Creating thread 2
The argument is 0
Creating thread 3
The argument is 1
Joining the thread 1
The argument is 2
Joining the thread 2
Joining the thread 3
```

As shown in the example above, to explicitly create a Pthread as joinable, the following steps are followed:

1. Declare a Pthread attribute variable of the pthread_attr_t data type.
2. Initialize the attribute variable with the pthread_attr_init() routine.
3. Set the attribute detached status (`PTHREAD_CREATE_JOINABLE`) with the pthread_attr_setdetachstate() routine (by default, the Pthread is joinable).
4. When done, free the library resources used by the attribute with the pthread_attr_destroy() routine.

The following factors need to be considered when deciding whether a Pthread has to be joined or not:

- If a Pthread requires joining, consider (as shown above) creating it as joinable. This provides portability, as not all implementations may create Pthreads as joinable by default.
- If you know in advance that a Pthread will never need to join with another Pthread, consider creating it in a detached state. Some system resources may be able to be freed.

### 8.3.3 Detaching a Pthread

We have seen how Pthreads can be joined using the pthread_join() function. In fact, Pthreads that are in a *joinable* state must be joined by other Pthreads, or else their memory resources will not be fully cleaned out. This is similar to what happens with processes whose parents did not clean up after them (also called *orphan* or *zombie* processes).

If we have a Pthread that we wish would exit whenever it wants without the need to join it, we should put it in the detached state. This can be done either with an thread attribute object with a PTHREAD_CREATE_DETACHED attached to the pthread_create() function, or by using the pthread_detach() function.

The pthread_detach() function gets one parameter, of type pthread_t, that denotes the Pthread we wish to put in the detached state. For example, we can create a Pthread and immediately detach it with code similar to the following example:

```
pthread_t a_thread;             /* store the thread's structure here.  */
int rc;                         /* return value for pthread functions. */
extern void* thread_loop(void*); /* declare the thread's main function. */

/* create a new thread. if succeeded, detach the newly created thread. */
if ((rc = pthread_create(&a_thread, NULL, thread_loop, NULL)) == 0) {
    rc = pthread_detach(a_thread);
}
```

By default, the Pthread that is created using the pthread_create() routine is joinable (the detach state is set to PTHREAD_CREATE_JOINABLE) and the scope is set to the process level (PTHREAD_SCOPE_PROCESS), which means that the user Pthread is not bound to a particular kernel thread. This is often referred to as the M:N thread model, where M is the number of user threads and N is the number of kernel threads.

## 8.3.4 Thread stack

When a Pthread is created, its own thread stack is also created. All auto storage class variables for the Pthread are allocated in its thread stack. When the Pthread terminates, all data allocated in its thread stack will be unallocated.

On AIX, thread stacks (except for the initial thread's stack) are created in the process heap. The thread stack of the initial thread is created when the process is created.[10]

By default, a thread stack size is 96 KB in the 32-bit user process model while it is 192 KB in 64-bit. The thread stack size can be tuned through either environment variables or through function calls, as explained in "Changing a thread stack size" on page 295. At the end of a thread stack, there is a 4 KB read/write protected page referred to as a *guard page* or *red zone* (see "AIXTHREAD_GUARDPAGES" on page 331 for further information about the guard page).

Figure 8-3 on page 293 illustrates how multiple thread stacks are allocated in the process heap in the 32-bit default memory model. The default thread stack size of 96 KB plus additional memory for the several necessary data structures will be consumed from the process heap every time a new Pthread is created. If a Pthread terminates, then the memory area for the terminated Pthread can be used again if another Pthread is created.

[10] For 32-bit process processes see 3.2.6, "Resource limits in 32-bit model" on page 125; for 64-bit processes, see 3.3.8, "Resource limits in 64-bit mode" on page 136.

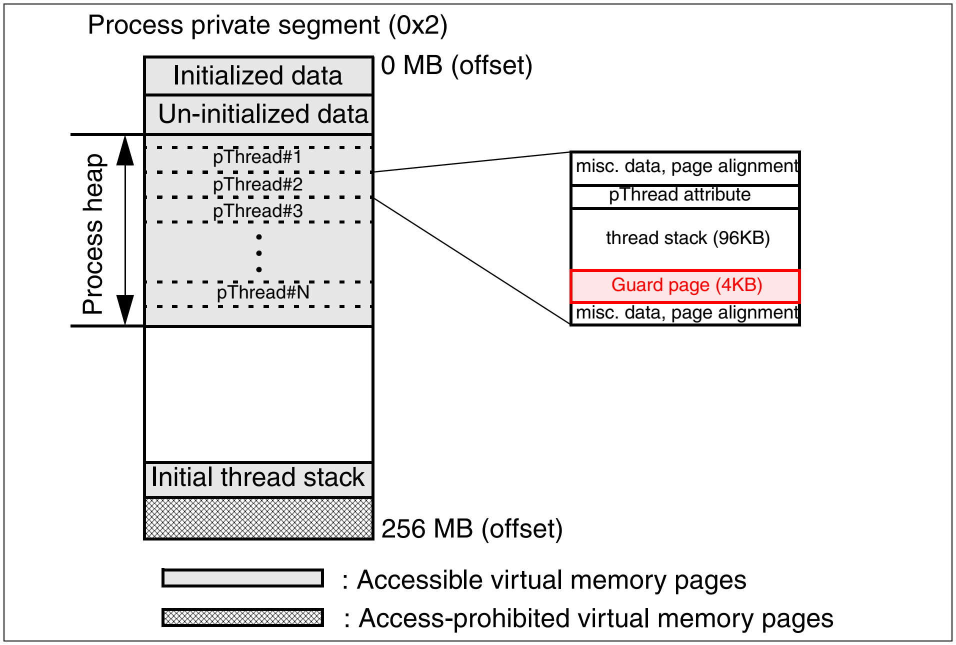


*Figure 8-3 Thread stacks (default 32-bit process model)*

Because the creation of Pthreads consumes the process heap, depending on number of Pthreads in a process, you may need to consider the large or very large memory model,[11] when developing 32-bit multi-threaded applications. When developing 64-bit multi-threaded applications, the data resource limit value only affects the maximum number of Pthreads within a process.[12]

To demonstrate how many Pthreads can be created in the 32-bit user process model, we have prepared a short example program, which is shown in Example 8-6 on page 294.

To compile the program, do the following:

```
$ cc_r -D_LARGE_THREADS max_threads.c
```

If the _LARGE_THREADS macro is defined, the PTHREAD_THREADS_MAX compilation time symbolic constant macro, which defines the maximum number of Pthreads per process, is set to 32767. If not defined, PTHREAD_THREADS_MAX is set to 512, which is a sufficient number for most multi-threaded applications, regardless of 32- or 64-bit applications.

[11] See 3.2.2, “Large memory model” on page 116 and 3.2.3, “Very large memory model” on page 117.
[12] See 3.3.8, “Resource limits in 64-bit mode” on page 136.

When executed, the program would print the number of Pthreads in the angle brackets (highlighted in the example) within the process after failing to create another Pthread, as shown in the following example:[13]

```
$ a.out
[ 1131]: pthread_create() failed with rc=11 at 21 in max_threads.c
$ for i in 1 2 3 4 5 6 7 8
do
echo "LDR_CNTRL=MAXDATA=0x${i}0000000"
LDR_CNTRL=MAXDATA=0x${i}0000000 a.out
done
LDR_CNTRL=MAXDATA=0x10000000
[ 2281]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x20000000
[ 4583]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x30000000
[ 6884]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x40000000
[ 9186]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x50000000
[11268]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x60000000
[13788]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x70000000
[15523]: pthread_create() failed with rc=11 at 21 in max_threads.c
LDR_CNTRL=MAXDATA=0x80000000
[18391]: pthread_create() failed with rc=11 at 21 in max_threads.c
```

**Note:** The numbers shown in this example should be considered *theoretical* maximum values. If a process allocates memory objects from its heap, uses many initialized or un-initialized data, or increases thread stack sizes, these numbers could be much smaller ones.

*Example 8-6 max_threads.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

void *thread_func(void *j)
{
    sleep(60);
    pthread_exit(0);
}

int main()
```

[13] The return code 11 means EAGAIN (resource temporary unavailable).

```
{
    int i, rc;
    pthread_t tid[PTHREAD_THREADS_MAX];

    for (i = 0; i < PTHREAD_THREADS_MAX; i++) {
        if ((rc = pthread_create(&tid[i], NULL, thread_func, (void *)i)) != 0){
            fprintf(stderr
                , "[%5d]: pthread_create() failed with rc=%d at %d in %s\n"
                , i, rc, __LINE__, __FILE__);
            exit(1);
        }
    }

    for (i = 0; i < PTHREAD_THREADS_MAX; i++) {
        pthread_join(tid[i], NULL);
    }
}
```

## Changing a thread stack size

On AIX, the minimum thread stack size for a Pthread is 8 KB and the maximum size is 256 MB. These values are defined by the compilation time symbolic constants PTHREAD_STACK_MIN and PTHREAD_STACK_MAX, respectively.

Although the default stack sizes of 96 KB for 32-bit and 192 KB for 64-bit applications suffice for most multi-threaded applications' demand, the stack size can be changed using the two methods described in the next two sections.

### *Setting the AIXTHREAD_STKSIZE environment value*

If this environment value is set before the execution of a multi-threaded application, then the default thread stack size of the application process will be set to the value specified by the environment value (see "AIXTHREAD_STK" on page 332). This is a process-basis setting.

### *Using the pthread_attr_setstacksize() sub-routine*

By creating a thread attribute with the specified thread size before the creation of a Pthread, the new Pthread will have the specified thread size. The following pseudo-code explains how to use this sub-routine:

1. Create a thread attribute using pthread_attr_init().
2. Set the stack size in the thread attribute using pthread_attr_setstacksize().
3. Create a new Pthread using pthread_create() with the thread attribute.

This is a Pthread-basis setting and overrides the effect of the AIXTHREAD_STKSIZE environment value.

## 8.4 Data synchronization between Pthreads

Synchronization is a programming method that allows multiple Pthreads to coordinate their data accesses, therefore avoiding the situation where one Pthread can change a piece of data at the same time another one is reading or writing the same piece of data. This situation is commonly called a race condition.

Consider, for example, a single counter, X, that is incremented by two Pthreads, A and B. If X is originally 1, then by the time Pthreads A and B increment the counter, X should be 3. Both Pthreads are independent entities and have no synchronization between them. If both Pthreads are executed concurrently on two CPUs, or if the scheduling makes the Pthreads alternatively execute on each instruction, the following steps may occur:

- Pthread A executes the first instruction and puts X, which is 1, into the Pthread A register. Then, Pthread B executes and puts X, which is 1, into the Pthread B register.
- Next, Pthread A executes the second instruction and increments the content of its register to 2. Then, Pthread B increments its register to 2. Nothing is moved to memory X, so memory X stays the same.
- Last, Pthread A moves the content of its register, which is now 2, into memory X. Then, Pthread B moves the content of its register, which is also 2, into memory X, overwriting Pthread A's value.

Note that, in most cases, Pthread A and Pthread B will execute the three instructions one after the other, and the result would be 3, as expected. Race conditions are usually difficult to discover because they occur intermittently.

To avoid this race condition, each Pthread should lock the data before accessing the counter and updating memory X. For example, if Pthread A takes a lock and updates the counter, it leaves memory X with a value of 2. Once Pthread A releases the lock, Pthread B takes the lock and updates the counter, taking 2 as its initial value for X and incrementing it to 3, the expected result.

To write a program of any complexity using Pthreads, you will need to share data between Pthreads, or cause various actions to be performed in some coherent order across multiple Pthreads. To do this, you need to synchronize the activity of the Pthreads. Data synchronization among Pthreads can be done using any of the three types of locking primitive: mutexs, condition variables, and read-write locks, as explained in the following sections:

- Section 8.4.1, “Synchronizing Pthreads with mutexes” on page 297
- Section 8.4.2, “Synchronizing Pthreads with condition variables” on page 303
- Section 8.4.3, “Synchronizing Pthreads with read-write locks” on page 310

Although these sections provide enough information to understand the concept of these locking primitives, data synchronization in the multi-threaded programming environment can be a challenging task. For further information about it, please refer to the following publications:

- *Programming with POSIX Threads*, by Lewine
- *POSIX Programmer's Guide: Writing Portable UNIX Programs*, by Butenhof

### 8.4.1 Synchronizing Pthreads with mutexes

One of the basic problems when running several Pthreads that use the same memory space is making sure that they do *not* interfere with each other. By this, we refer to the problem of using a data structure from two different Pthreads.

The mutual exclusion lock (mutex) is the simplest synchronization primitive provided by the Pthread library, and many of the other synchronization primitives are built upon it.

It is based on the concept of a resource that only one person can use, in a period of time, for example, a chair or a pencil. If one person sits in a chair, no one can sit on it until the first person stands up. This kind of primitive is quite useful for creating critical sections. A critical section is a portion of code that must run atomically because they normally are handling resources, such as file descriptors, I/O devices, or shared data. A critical section is a portion of code delimited by the instructions that lock and unlock a mutex variable. Ensuring that all Pthreads acting on the same resource or shared data obey this rule is a very good practice to avoid trouble when programming with Pthreads. A mutex is a lock that guarantees three things:

**Atomicity** Locking a mutex is an atomic operation, meaning that the operating system (or Pthreads library) assures you that if you locked a mutex, no other Pthread succeeded in locking this mutex at the same time.

**Singularity** If a Pthread managed to lock a mutex, it is assured that no other Pthread will be able to lock the Pthread until the original Pthread releases the lock.

**Non-busy wait** If a Pthread attempts to lock a Pthread that was locked by a second Pthread, the first Pthread will be suspended (and will not consume any CPU resources) until the lock is freed by the second Pthread. At this time, the first Pthread will wake up and continue execution, having the mutex locked by it.

Very often, the action performed by a Pthread owning a mutex is the updating of global variables. This is a safe way to ensure that when several Pthreads update the same variable, the final value is the same as what it would be if only one Pthread performed the update. The variables being updated belong to a critical section. A typical sequence of using mutex is as follows:

1. Create and initialize the mutex variable.
2. Several Pthreads attempt to lock the mutex.
3. Only one succeeds and that Pthread owns the mutex.
4. The owner Pthread performs some set of actions.
5. The owner unlocks the mutex.
6. Another Pthread acquires the mutex and repeats the process.
7. Finally, the mutex is destroyed.

## Creating and initializing a mutex

In order to create a mutex, we first need to declare a variable of type pthread_mutex_t, and then initialize it. The simplest way is by assigning it the PTHREAD_MUTEX_INITIALIZER constant. For example, statically, it is declared as follows:

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
```

A mutex variable can also be dynamically initialized with the pthread_mutex_init() routine. This method permits the setting of mutex object attributes.

The pthread_mutexattr_init() and pthread_mutexattr_destroy() routines are used to create and destroy mutex attribute objects respectively.

## Locking and unlocking a mutex

In order to lock a mutex, we may use the pthread_mutex_lock() function. This function attempts to lock the mutex, or block the Pthread if the mutex is already locked by another Pthread. In this case, when the mutex is unlocked by the first Pthread, the function will return with the mutex locked by the other Pthreads. Here is how to lock a mutex (assuming it was initialized earlier):

```
int rc = pthread_mutex_lock(&a_mutex);
if (rc) {
    /* an error has occurred */
    fprintf(stderr, "pthread_mutex_lock() failed with rc = %d.\n", rc);
    pthread_exit(NULL);
}
/* mutex is now locked - do your stuff. */
...
```

After the Pthread did what it had to (change variables or data structures, handle file, or whatever it intended to do), it should free the mutex, using the pthread_mutex_unlock() function, as follows:

```
rc = pthread_mutex_unlock(&a_mutex);
if (rc) {
    fprintf(stderr, "pthread_mutex_unlock() failed with rc = %d.\n", rc);
    pthread_exit(NULL);
}
```

### Destroying a mutex

After we finished using a mutex, we should destroy it. Finished using it means no Pthread needs it at all. If only one Pthread is finished with the mutex, it should leave it alive for the other Pthreads that might need to use it. Once all Pthreads finished using it, the last one can destroy it using the pthread_mutex_destroy() function:

```
rc = pthread_mutex_destroy(&a_mutex);
```

The pthread_mutex_destroy function destroys the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. An implementation may cause pthread_mutex_destroy() to set the object referenced by mutex to an invalid value. A destroyed mutex object can be re-initialized using pthread_mutex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

### An example of using mutexs

The example program shown in Example 8-7 on page 300 illustrates the use of mutex variables in a multi-threaded program that increments a counter variable declared globally. Each Pthread works on the same data. The initial thread waits for all the other Pthreads to complete their executions, and then it prints the resulting sum.

This program demonstrates how to use mutexes to synchronize the operation of Pthreads. In this program, the main Pthread creates two Pthreads. Each Pthread tries to access a global variable *counter*, increment its value, and print it. A mutex is used to allow only a single Pthread to access the counter at any point of time.

*Example 8-7 pthread_mutex.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define EXIT_CODE -1

pthread_t tid1, tid2;    /* thread IDs. */
pthread_attr_t t_attr;   /* thread attribute structure. */
pthread_mutex_t mut1;    /* mutex. */
pthread_mutexattr_t attr;/* mutex attribute structure. */
int counter = 0;         /* global data that will be accessed by two threads.*/

/* first thead executes this function. */
void* mythread1(void* arg)
{
    int rc;
    while (counter < 10) {
        /* block until the mutex lock is obtained, lock when you get it. */
        if ((rc = pthread_mutex_lock(&mut1)) != 0) {
            fprintf(stderr
                , "pthread_mutex_lock() failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
        if (counter < 10) {
            counter++;
        }
        fprintf(stderr,"Thread 1 = %02d\n ", counter);
        /* release or unlock the mutex */
        if ((rc = pthread_mutex_unlock(&mut1)) != 0) {
            fprintf(stderr
                , "pthread_mutex_unlock() failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
        sleep(1);
    }
}

/* second thread executes this function */
void* mythread2(void* arg)
{
    int rc;
    while (counter < 10) {
        /* block untill the mutex lock is obtained, lock when you get it. */
        if ((rc = pthread_mutex_lock(&mut1)) != 0) {
            fprintf(stderr
                , "pthread_mutex_lock() failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
        if (counter < 10) {
            counter++;
        }
        printf("Thread 2 = %02d\n", counter);
        /* release or unlock the mutex */
        if ((rc = pthread_mutex_unlock(&mut1)) != 0) {
            fprintf(stderr
                , "pthread_mutex_unlock() failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
        sleep(1);
    }
}

/* main thread of execution. */
int main(int argc, char* argv[])
{
    int rc, status;
    /* create the mutex attribute structure. */
    if ((rc = pthread_mutexattr_init(&attr)) != 0) {
        fprintf(stderr, "pthread_mutexattr_init() failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    /* initialize the mutex to be used. */
    if ((rc = pthread_mutex_init(&mut1, &attr)) != 0) {
        fprintf(stderr, "pthread_mutex_init() failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    /* initialize the attribute structure for the threads to be created. */
    if ((rc = pthread_attr_init(&t_attr)) != 0) {
        fprintf(stderr, "pthread_attr_init() failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    /* create the first thread. */
    if ((rc = pthread_create(&tid1, &t_attr, mythread1, 0)) != 0) {
        fprintf(stderr, "pthread_create(#1) failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    /* create the second thread */
    if ((rc = pthread_create(&tid2, &t_attr, mythread2, 0)) != 0) {
        fprintf(stderr, "pthread_create(#2) failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    pthread_attr_destroy(&t_attr);
    pthread_join(tid1, (void **)&status);
    pthread_join(tid2, (void **)&status);

    printf("\nSum is = %d\n", counter);
    pthread_mutex_destroy(&mut1);
    pthread_exit(NULL);
}
```

The output of the above program is:

```
Thread 1 = 01
Thread 2 = 02
Thread 1 = 03
Thread 2 = 04
Thread 1 = 05
Thread 2 = 06
Thread 1 = 07
Thread 2 = 08
Thread 1 = 09
Thread 2 = 10

Sum is = 10
```

Please note that the program examines the value of the counter again after it examines the value in the while condition (highlighted in Example 8-7 on page 300). If it does not, the program results in a potential race condition, so that the value of the counter could be 11 instead of 10.

## Starvation and deadlock situations

We should remember that pthread_mutex_lock() might block for a undetermined duration if the mutex is already locked. If it remains locked forever, it is said that our Pthread is *starved*, that is, it tried to acquire a resource, but never got it. It is up to the programmer to ensure that such starvation will not occur. The Pthread library does not help us with that.

The Pthread library might, however, figure out a *deadlock*. A deadlock is a situation in which a set of Pthreads are all waiting for resources taken by other Pthreads, all in the same set. Naturally, if all Pthreads are blocked waiting for a mutex, none of them will ever come back to life again. The Pthread library keeps track of such situations, and thus would fail the last Pthread trying to call pthread_mutex_lock() with an error of type EDEADLK. The programmer should check for such a value, and take steps to solve the deadlock somehow.

The following code fragment will result in a deadlock:

```
pthread_mutex_t mutex_A;
pthread_mutex_t mutex_B;

int counter_A = 0;
int counter_B = 0;
```

```
/* func_1() will be executed by the Pthread #1. */
void func_1()
{
   pthread_mutex_lock(&mutex_A);
   counter_A++;

   pthread_mutex_lock(&mutex_B);
   counter_B++;
   pthread_mutex_unlock(&mutex_B);

   pthread_mutex_unlock(&mutex_A);
}

/* func_2() will be executed by the Pthread #2. */
void func_2()
{
   pthread_mutex_lock(&mutex_B);
   counter_B++;

   pthread_mutex_lock(&mutex_A);
   counter_A++;
   pthread_mutex_unlock(&mutex_A);

   pthread_mutex_unlock(&mutex_B);
}
```

In the above example, while mutex_A is already locked by Pthread #1 executing func_1(), multex_B can be also locked by Pthread #2 executing func_B(). Both Pthreads wait for a mutex until the other Pthread releases it. To avoid this situation, both Pthreads must try to acquire to lock in the same order.

## 8.4.2 Synchronizing Pthreads with condition variables

Condition variables provide yet another way for Pthreads to synchronize. While mutexes implement synchronization by controlling Pthread access to data, condition variables allow Pthreads to synchronize based upon the actual value of data. Without condition variables, the programmer would need to have Pthreads continually polling (possibly in a critical section) to check if the condition is met. This can be very resource consuming since the Pthread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

### What is a condition variable

A condition variable is a mechanism that allows Pthreads to wait (without wasting CPU cycles) for some event to occur. Several Pthreads may wait on a condition variable, until some other Pthread signals this condition variable (thus sending a notification). At this time, one of the Pthreads waiting on this condition variable wakes up, and can act on the event. It is possible to also wake up all Pthreads waiting on this condition variable by using a broadcast method on this variable.

Note that a condition variable itself does not provide any kind of locking. Thus, a mutex is used along with the condition variable to provide the necessary locking when accessing this condition variable.

### Creating and initializing a condition variable

Condition variables must be declared with pthread_cond_t, and must be initialized before they can be used. There are two ways to initialize a condition variable.

- Statically, when it is declared. For example,

  ```
  pthread_cond_t a_cond_var = PTHREAD_COND_INITIALIZER;
  ```

- To initialize the condition variable at run time, use the pthread_cond_init() routine. This method permits setting condition variable object attributes. The pthread_condattr_init() and pthread_condattr_destroy() routines are used to create and destroy condition variable attribute objects.

### Signalling a condition variable

In order to signal a condition variable, one should either use the pthread_cond_signal() function (to wake up only one Pthread waiting on this variable), or the pthread_cond_broadcast() function (to wake up all Pthreads waiting on this variable). Here is an example of using a signal, assuming *a_cond_var* is a properly initialized condition variable:

```
int rc = pthread_cond_signal(&a_cond_var);
```

Or by using the broadcast function:

```
int rc = pthread_cond_broadcast(&a_cond_var);
```

When either function returns, the return code rc is set to 0 on success, and to a non-zero value on failure. In case of failure, the return code denotes the error that occurred; EINVAL denotes that the given parameter is not a condition variable and ENOMEM denotes that the system has run out of memory.

The success of a signaling operation does not mean any Pthread was awakened. It might be that no Pthread was waiting on the condition variable, and thus the signaling does nothing (that is, the signal is lost).

> **Note:** The term signal, as used in this section, is a different concept from the one used in the UNIX signal mechanism.

## Waiting on a condition variable

If one Pthread signals the condition variable, other Pthreads would probably want to wait for this signal. They may do so using one of two functions, pthread_cond_wait() or pthread_cond_timedwait(). Each of these functions takes a condition variable and a mutex (which should be locked before calling the wait function), unlocks the mutex, and waits until the condition variable is signaled, suspending the Pthread's execution. If this signaling causes the Pthread to awake (see the discussion about pthread_cond_signal() in “Signalling a condition variable” on page 304), the mutex is automatically locked again by the wait function, and the wait function returns.

The only difference between these two functions is that pthread_cond_timedwait() allows the programmer to specify a timeout for the waiting, after which the function always returns with a proper error value (ETIMEDOUT) to notify that condition variable was *not* signaled before the timeout passed. The pthread_cond_wait() would wait indefinitely if it was never signaled.

The code fragment shown in Example 8-8 depicts how to use pthread_cond_wait(). We make the assumption that *a_cond_var* is a properly initialized condition variable, and that request_mutex is a properly initialized mutex.

*Example 8-8 An example of using pthread_cond_wait()*

```
/* first, lock the mutex. */
int rc;
if ((rc = pthread_mutex_lock(&a_mutex)) != 0) {
    /* an error has occurred. */
    fprintf(stderr, "pthread_mutex_lock() failed with rc = %d.\n", rc);
    pthread_exit(NULL);
}
/* mutex is now locked - wait on the condition variable.             */
/* During the execution of pthread_cond_wait, the mutex is unlocked. */
if ((rc = pthread_cond_wait(&a_cond_var, &a_mutex)) == 0) {
    /*
     * we were awakened due to the condition variable, which had been signaled.
     * The mutex is now locked again by pthread_cond_wait().
     * do your stuff....
     */
    ...
}
/* finally, unlock the mutex. */
pthread_mutex_unlock(&a_mutex);
```

The code fragment shown in Example 8-9 on page 306 shows how to use pthread_cond_timedwait().

*Example 8-9 An example using pthread_cond_timedwait()*

```
#include <sys/time.h>           /* struct timeval definition.          */
#include <unistd.h>             /* declaration of gettimeofday().      */

struct timeval  now;            /* time when we started waiting.       */
struct timespec timeout;        /* timeout value for the wait function.*/
int             done;           /* are we done waiting?                */

/* first, lock the mutex. */
int rc = pthread_mutex_lock(&a_mutex);
if (rc) {
    /* an error has occurred. */
    fprintf(stderr, "pthread_mutex_lock() failed with rc = %d.\n", rc);
    pthread_exit(NULL);
}
/* mutex is now locked. */

/* get current time. */
gettimeofday(&now);
/* prepare timeout value. */
/* t_sec member is represented in seconds, while tv_usec in microseconds. */
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;

/* wait on the condition variable. */
/* we use a loop, since a UNIX signal might stop the wait before the timeout.*/
done = 0;
while (!done) {
    /* remember that pthread_cond_timedwait() unlocks the mutex on entrance. */
    rc = pthread_cond_timedwait(&a_cond_var, &a_mutex, &timeout);
    switch(rc) {
    case 0:
        /*
         * we were awakened due to the condition variable
         * , which had been signaled.
         * The mutex is now locked again by pthread_cond_wait().
         * do your stuff....
         */
        ...
        done = 0;
        break;
    case ETIMEDOUT: /* our time is up. */
        done = 0;
        break;
    default:        /* some error occurred (e.g. we got a UNIX signal). */
        break;      /* break this switch, but re-do the while loop.     */
    }
}
```

```
/* finally, unlock the mutex. */
pthread_mutex_unlock(&a_mutex);
```

It might be that a condition variable that has two or more Pthreads waiting on it is signaled many times, and yet one of the Pthreads waiting on it never awakens. This is because we are not guaranteed which of the waiting Pthreads is awakened when the variable is signaled. It might be that the awakened Pthread quickly comes back to waiting on the condition variables, and gets awakened again when the variable is signaled again, and so on. The situation for the un-awakened Pthread is called *starvation*. It is up to the programmer to make sure that this situation does not occur if it implies bad behavior.

When the mutex is being broadcast using pthread_cond_broadcast(), this does not mean all Pthreads are running together. Each of them tries to lock the mutex again before returning from their wait function, and thus they will start running one by one, each one locking the mutex, doing their work, and freeing the mutex before the next Pthread gets its chance to run.

### Destroying a condition variable

After we are done using a condition variable, we should destroy it to free any system resources it might be using. This can be done using pthread_cond_destroy(). In order for this to work, there should be no Pthreads waiting on this condition variable. Here is how to use this function, again assuming *a_cond_var* is a pre-initialized condition variable:

```
int rc;

if ((rc = pthread_cond_destroy(&a_cond_var)) == EBUSY) {
    /* some Pthread is still waiting on this condition variable. */
    /* handle this case here... */
}
```

What if some Pthread is still waiting on this variable? Depending on the case, it might imply some flaw in the usage of this variable, or just lack of proper Pthread cleanup code.

### Using a condition variable

The example program shown in Example 8-10 on page 308 demonstrates the use of several Pthread condition variable routines. The main routine creates three Pthreads. Two Pthreads that execute the add_counter() function increment a variable, named count, whenever the associated mutex is held. The third Pthread that executes the monitor_counter() function waits until the count variable reaches a specified value.

*Example 8-10 pthread_cond_variable.c*

```
#include <pthread.h>
#include <stdio.h>

#define NUM_OF_THREADS  3
#define TOTAL_COUNT 5
#define MAX_COUNT 5
#define EXIT_CODE -1

int     count = 0;
int     thread_ids[3] = {0, 1, 2};
pthread_mutex_t a_mutex;
pthread_cond_t a_cond_var;

void *add_counter(void *arg)
{
    unsigned int i, j;
    double result = 0.0;
    int *my_id = arg;

    for (i = 0; i < TOTAL_COUNT; i++) {
        pthread_mutex_lock(&a_mutex);
        count++;

        /*
         * Check the value of count and signal waiting thread when condition is
         * reached.  Note that this occurs while mutex is locked.
         */
        if (count == MAX_COUNT) {
            pthread_cond_signal(&a_cond_var);
            printf(
                "add_counter(): thread %d, count = %d, threshold reached.\n"
                , *my_id, count);
        }
        printf("add_counter(): thread %d, count = %d, unlocking mutex.\n"
            , *my_id, count);
        pthread_mutex_unlock(&a_mutex);

        /* Do some work so threads can alternate on mutex lock. */
        for (j = 0; j < 1000; j++)
            result += j;
        }
        pthread_exit(NULL);
}

void *monitor_counter(void *arg)
{
    int *my_id = arg;
    printf("Starting monitor_counter(): thread %d\n", *my_id);

    /*
     * Lock mutex and wait for signal.  Note that the pthread_cond_wait()
     * routine will automatically and atomically unlock mutex while it waits.
     * Also, note that if MAX_COUNT is reached before this routine is run by
     * the waiting thread, the loop will be skipped to prevent
     * pthread_cond_wait() from never returning.
     */
    pthread_mutex_lock(&a_mutex);
    while (count < MAX_COUNT) {
        pthread_cond_wait(&a_cond_var, &a_mutex);
        printf("monitor_counter(): thread %d Condition signal received.\n"
            , *my_id);
    }
    pthread_mutex_unlock(&a_mutex);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    int i, rc;
    pthread_t threads[NUM_OF_THREADS];
    pthread_attr_t attr;

    /* Initialize mutex and condition variable objects. */
    pthread_mutex_init(&a_mutex, NULL);
    pthread_cond_init (&a_cond_var, NULL);

    /*
     * For portability, explicitly create threads in a joinable state
     * so that they can be joined later.
     */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    if ((rc = pthread_create(&threads[0], &attr, add_counter
                , (void *)&thread_ids[0])) != 0) {
        fprintf(stderr, "pthread_create(#1) is failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    if ((rc = pthread_create(&threads[1], &attr, add_counter
                , (void *)&thread_ids[1])) != 0) {
        fprintf(stderr, "pthread_create(#2) is failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }
    if ((rc = pthread_create(&threads[2], &attr, monitor_counter
                , (void *)&thread_ids[2])) != 0) {
        fprintf(stderr, "pthread_create(#3) is failed with rc = %d.\n", rc);
        exit(EXIT_CODE);
    }

    /* Wait for all threads to complete */
    for (i = 0; i < NUM_OF_THREADS; i++) {
        if ((rc = pthread_join(threads[i], NULL)) != 0) {
            fprintf(stderr, "pthread_join() is failed with rc = %d.\n", rc);
            exit(EXIT_CODE);
        }
    }
    printf ("Main(): Waited on %d threads. Done...\n", NUM_OF_THREADS);

    /* Clean up and exit */
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&a_mutex);
    pthread_cond_destroy(&a_cond_var);
    pthread_exit(NULL);
}
```

The output of this program is:

```
add_counter(): thread 0, count = 1, unlocking mutex.
add_counter(): thread 0, count = 2, unlocking mutex.
Starting monitor_counter(): thread 2
add_counter(): thread 0, count = 3, unlocking mutex.
add_counter(): thread 1, count = 4, unlocking mutex.
add_counter(): thread 0, count = 5, threshold reached.
add_counter(): thread 0, count = 5, unlocking mutex.
add_counter(): thread 1, count = 6, unlocking mutex.
add_counter(): thread 0, count = 7, unlocking mutex.
add_counter(): thread 1, count = 8, unlocking mutex.
monitor_counter(): thread 2 Condition signal received.
add_counter(): thread 1, count = 9, unlocking mutex.
add_counter(): thread 1, count = 10, unlocking mutex.
Main(): Waited on 3 threads. Done...
```

### 8.4.3 Synchronizing Pthreads with read-write locks

Another type of lock primitive, read-write locks allow a Pthread to exclusively lock some shared data while updating that data, or allow any number of Pthreads to have simultaneous read-only access to the data.

Unlike a mutex, a read-write lock distinguishes between reading data and writing data. A mutex excludes all other Pthreads. A read-write lock allows other Pthreads access to the data, providing no Pthread is modifying the data. Thus, a read-write lock is less primitive than either a mutex-condition variable pair or a semaphore.

Application developers should consider using a read-write lock rather than a mutex to protect data that is frequently referenced but seldom modified. Most Pthreads (readers) will be able to read the data without waiting and will only have to block when some other Pthread (a writer) is in the process of modifying the data. Conversely, a Pthread that wants to change the data is forced to wait until there are no readers. This type of lock is often used to facilitate parallel access to data on multiprocessor platforms or to avoid context switches on single processor platforms where multiple Pthreads access the same data.

If a read-write lock becomes unlocked and there are multiple Pthreads waiting to acquire the write lock, the implementation's scheduling policy determines which Pthread shall acquire the read-write lock for writing. If there are multiple Pthreads blocked on a read-write lock for both read locks and write locks, it is unspecified whether the readers or a writer acquires the lock first. However, for performance reasons, implementations often favor writers over readers to avoid potential writer starvation.

A read-write lock object is an implementation-dependent opaque object of type pthread_rwlock_t, as defined in pthread.h. There are two different sorts of locks associated with a read-write lock: a *read lock* and a *write lock*.

The pthread_rwlockattr_init() function initializes a read-write lock attributes object with the default value for all the attributes defined in the implementation. After a read-write lock attributes object has been used to initialize one or more read-write locks, changes to the read-write lock attributes object, including destruction, do not affect previously initialized read-write locks.

The read-write lock attribute can be any of the following:

**PTHREAD_PROCESS_SHARED**

Any Pthread of any process that has access to the memory where the read-write lock resides can manipulate the read-write lock.

**PTHREAD_PROCESS_PRIVATE**

Only Pthreads created within the same process as the Pthread that initialized the read-write lock can manipulate the read-write lock. This is the default value.

The pthread_rwlockattr_setpshared() function is used to set the process-shared attribute of an initialized read-write lock attributes object while the function pthread_rwlockattr_getpshared() obtains the current value of the process-shared attribute. A read-write lock attributes object is destroyed using the pthread_rwlockattr_destroy() function.

A Pthread creates a read-write lock using the pthread_rwlock_init() function. The attributes of the read-write lock can be specified by the application developer; otherwise, the default implementation-dependent read-write lock attributes are used if the pointer to the read-write lock attributes object is NULL. In cases where the default attributes are appropriate, the PTHREAD_RWLOCK_INITIALIZER macro can be used to initialize statically allocated read-write locks.

A Pthread that wants to apply a read lock to the read-write lock can use either pthread_rwlock_rdlock() or pthread_rwlock_tryrdlock(). If pthread_rwlock_rdlock() is used, the Pthread acquires a read lock if a writer does not hold the write lock and there are no writers blocked on the write lock. If a read lock is not acquired, the calling Pthread blocks until it can acquire a lock. However, if pthread_rwlock_tryrdlock() is used, the function returns immediately with the error EBUSY if any Pthread holds a write lock or there are blocked writers waiting for the write lock.

A Pthread that wants to apply a write lock to the read-write lock can use either of two functions: pthread_rwlock_wrlock() or pthread_rwlock_trywrlock(). If pthread_rwlock_wrlock() is used, the Pthread acquires the write lock if no other reader or writer Pthreads holds the read-write lock. If the write lock is not acquired, the Pthread blocks until it can acquire the write lock. However, if pthread_rwlock_trywrlock() is used, the function returns immediately with the error EBUSY if any Pthread is holding either a read or a write lock.

The pthread_rwlock_unlock() function is used to unlock a read-write lock object held by the calling Pthread.

Supported read-write lock sub-routines are listed in Table D-5 on page 480.

For further information about the read-write lock programming on AIX, please refer to the “Threads Programming Guidelines” section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 8.5 Thread-specific data

POSIX provides a mechanism that enables applications to maintain specified data on a per-Pthread basis. The mechanism is motivated by the need of some modules (that is, groups of related functions) to maintain selected data across function invocations (that is, to maintain static data in the C programming language). If such a module is being used by multiple Pthreads of a multi-threaded process, then the module may need to maintain such data separately for each calling Pthread, depending on the particular application.

**Note:** If a Pthread executes only one function during its life-cycle, it less likely needs thread-specific data. Because each Pthread has its own thread stack, all auto storage class variables in that function are allocated in the thread stack, which would be automatically deallocated when the Pthread terminates (see 8.3.4, “Thread stack” on page 292).

For example, consider a module consisting of the push, pop, and clear stack functions. Suppose that the module declares the stack and stack pointer as static data, instead of as function parameters. If the module is being used in a multi-threaded process, in which each Pthread needs its own stack and stack pointer, the module must have a mechanism for maintaining per-thread stacks and stack pointers.

In the POSIX model, per-thread data is maintained through a key/value mechanism, an approach designed for efficiency and ease of use. A key is an opaque object of type pthread_key_t. The value of a key is thread-specific, that is, each key that has been established for a multi-threaded process has a distinct value for each Pthread of the process. For this reason, the thread-specific data of a process is sometimes thought of as a matrix, with rows corresponding to keys and columns to Pthreads, although implementations need not work this way. A process can have up to PTHREAD_KEYS_MAX keys (or rows), where PTHREAD_KEYS_MAX is defined at least at 128 in the POSIX thread standard.[14]

Keys are of opaque data type so that operating system implementations can have the freedom in setting them up to offer efficient access to thread-specific data. Instead of holding thread-specific values directly, keys may hold means of accessing thread-specific values. Conceptually, a key isolates a row of the thread-specific data matrix, and then the key uses the Pthread ID of the calling Pthread (the Pthread calling pthread_getspecific() or pthread_setspecific()) to isolate an entry in the row, thus obtaining the desired key value.

Typically, the value associated with a given key for a given Pthread is a pointer to memory dynamically allocated for the exclusive use of the given Pthread (for example, per-thread stack and stack pointer). The scenario for establishment and use of thread-specific data can be described as follows. A module that needs to maintain static data on a per-thread basis creates a new thread-specific data key as a part of its initialization routine. At initialization, all Pthreads of the process are assigned null values for the new key. Then, upon each Pthread’s first invocation of the module (which can be determined by checking for a null key value), the module dynamically allocates memory for the exclusive use of the

[14] Currently, AIX 5L Version 5.2 supports a maximum of 450 keys per process. However, it can be increased in the future versions of AIX. Call the sysconf() routine with _SC_THREAD_KEYS_MAX to determine the maximum number of keys.

calling Pthread, and stores a pointer to the memory as the calling Pthread's value of the new key. Upon later invocations of the same module by the same Pthread, the module can access the Pthread's data through the new key (that is, the Pthread's value for the key). Other modules can independently create other thread-specific data keys for other per-thread data for their own use.

An application process could maintain thread-specific data in other ways. For example, it could use a hash function on a Pthread ID as a means of access to an area of thread-specific data. Then the application process would have to manage the use of sub-areas of the thread-specific data. The POSIX thread-specific data interfaces, on the other hand, provide Pthreads more direct access to sub-areas of their thread-specific data. Moreover, the sub-areas are independent; different parts of the application process can use different sub-areas without any need for process-wide cooperation.

### 8.5.1 Allocating thread-specific data

The pthread_key_create() function is used to allocate a new key. This key now becomes valid for all Pthreads in our process. When a key is created, the value it points to defaults to NULL. Later on, each Pthread may change its copy of the value as it wishes. The following code fragment shows how to use this function:

```
/* rc is used to contain return values of pthread functions. */
int rc;
/* define a variable to hold the key, once created.          */
pthread_key_t list_key;
/*
 * cleanup_list() is a function that can clean up some data.
 * it is specific to our program, not to thread-specific data.
 */
extern void* cleanup_list(void*);
/* create the key, supplying a function that'll be invoked when it's deleted.*/
rc = pthread_key_create(&list_key, cleanup_list);
```

Please note the following:

- After pthread_key_create() returns, the variable list_key points to the newly created key.
- The function pointer, passed as a second parameter to pthread_key_create(), will be automatically invoked by the Pthread library when our Pthread exits, with a pointer to the key's value as its parameter. We may supply a NULL pointer as the function pointer, and then no function will be invoked for key. Note that the function will be invoked once in each Pthread, even though we created this key only once in one Pthread.
- If we created several keys, their associated destructor functions will be called in an arbitrary order, regardless of the order of keys creation.

- If the pthread_key_create() function succeeds, it returns 0. Otherwise, it returns some error code.
- There is a limit of PTHREAD_KEYS_MAX keys that may exist in our process at any given time. An attempt to create a key after PTHREAD_KEYS_MAX exits will cause a return value of EAGAIN from the pthread_key_create() function.

## 8.5.2 Accessing thread-specific data

After we have created a key, we may access its value using two Pthread functions: pthread_getspecific() and pthread_setspecific(). The first is used to get the value of a given key, and the second is used to set the data of a given key. A key's value is simply a void pointer (void *), so we can store in it anything that we want. The following code fragment shows you how to use these functions, assuming that a_key is a properly initialized variable of type pthread_key_t, which contains a previously created key:

```
/* this variable will be used to store return codes of pthread functions. */
int rc;

/* define a variable into which we'll store some data. */
int* p_num = (int *)malloc(sizeof(int));
if (!p_num) {
    perror("malloc() failed.\n");
    exit(1);
}
/* initialize our variable to some value. */
*p_num = 4;

/*
 * now let’s store this value in our thread-specific data key.
 * note that we don't store p_num in our key.
 * we store the value that p_num points to.
 */
rc = pthread_setspecific(a_key, (void *)p_num);
....
....
/*
 * and somewhere later in our code, get the value of key a_key and print it.
 */
int* p_keyval = (int *)pthread_getspecific(a_key);

if (p_keyval != NULL) {
    printf("value of 'a_key' is: %d\n", *p_keyval);
}
```

Note that if we set the value of the key in one Pthread, and try to get it in another Pthread, we will get a NULL, since this value is distinct for each Pthread.

There are two other cases where pthread_getspecific() might return NULL:

- The key supplied as a parameter is invalid (for example, its key was not created).
- The value of this key is NULL. This means it either was not initialized, or was set to NULL explicitly by a previous call to pthread_setspecific().

## 8.5.3 Deleting thread-specific data

The pthread_key_delete() function may be used to delete keys. It does *not* delete memory associated with this key *or* call the destructor function in C++ defined during the key's creation. Thus, we still need to do memory cleanup on our own if we need to free this memory during run time. However, with global variables (and thus also thread-specific data), we usually do not need to free this memory until the thread terminates, in which case the pthread library will invoke our destructor functions anyway.

Using this function is simple. Assuming list_key is a pthread_key_t variable pointing to a properly created key, use this function like this:

```
int rc = pthread_key_delete(key);
```

The function will return 0 on success, or EINVAL if the supplied variable does not point to a valid thread-specific data key.

### An example of using thread-specific data

The example program shown in Example 8-11 illustrates the use of thread-specific data routines. Each created Pthread maintains its own copy of the data structure tsd_data, which is declared globally.

*Example 8-11 pthread_tsd.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define EXIT_CODE -1
/*
 * Structure used as the value for thread-specific data key.
 */
typedef struct tsd_data {
    pthread_t   thread_id;
    char        *mesg;
} tsd_data_var;
```

```
pthread_key_t tsd_data_key;           /* thread-specific data key */
pthread_once_t key_once = PTHREAD_ONCE_INIT;

/*
 * init_routine() is an one-time initialization routine used with
 * pthread_once().
*/
void init_routine(void)
{
    int rc;

    printf("Initializing key.\n");
    if ((rc = pthread_key_create(&tsd_data_key, NULL)) != 0) {
        fprintf(stderr, “pthread_create() failed with rc = %d.\n”, rc);
        exit(EXIT_CODE);
    }
}

/*
 * tsd_routine() is a routine that uses pthread_once() to dynamically
 * create a thread-specific data key.
 */
void *tsd_routine(void *arg)
{
    tsd_data_var *value;
    int rc;

    if ((rc = pthread_once(&key_once, init_routine)) != 0) {
        fprintf(stderr, “pthread_once() failed with rc = %d.\n”, rc);
        exit(EXIT_CODE);
    }
    value = (tsd_data_var *)malloc(sizeof(tsd_data_var));
    if (value == NULL) {
        perror("Failed to allocate key value.");
        exit(EXIT_CODE);
    }
    if ((rc = pthread_setspecific(tsd_data_key, value)) != 0) {
        fprintf(stderr, “pthread_setspecific() failed with rc = %d.\n”, rc);
        exit(EXIT_CODE);
    }
    printf("%s set thread-specific data value %p\n", arg, value);
    value->thread_id = pthread_self();
    value->mesg = (char *)arg;
    value = (tsd_data_var *)pthread_getspecific(tsd_data_key);
    printf("%s Starting...\n", value->mesg);
    sleep(2);
    value = (tsd_data_var *)pthread_getspecific(tsd_data_key);
    printf("%s Done...\n", value->mesg);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t thread1, thread2;
    int rc;

    if ((rc = pthread_create(&thread1, NULL, tsd_routine, "thread 1")) != 0) {
        fprintf(stderr, “pthread_create(#1) failed with rc = %d.\n”, rc);
        exit(EXIT_CODE);
    }
    if ((rc = pthread_create(&thread2, NULL, tsd_routine, "thread 2")) != 0) {
        fprintf(stderr, “pthread_create(#2) failed with rc = %d.\n”, rc);
        exit(EXIT_CODE);
    }
    pthread_exit (NULL);
}
```

The output of Example 8-11 on page 316 is:

```
Initializing key.
thread 1 set thread-specific data value 20209d58
thread 1 Starting...
thread 2 set thread-specific data value 20209d68
thread 2 Starting...
thread 1 Done...
thread 2 Done...
```

In Example 8-11 on page 316, the following two new Pthread routines are called: pthread_once() and pthread_self().

We have called the pthread_once() routine to dynamically create a Pthread specific data key. The pthread_once() routine calls an initialization routine executed by one Pthread a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple Pthreads or multiple times in the same Pthread.

For example, a mutex or a thread-specific data key must be created exactly once. Calling pthread_once() prevents the code that creates a mutex or thread-specific data from being called by multiple Pthreads. Without this routine, the execution must be serialized so that only one Pthread performs the initialization. Other Pthreads that reach the same point in the code are delayed until the first Pthread has finished the call to pthread_once().

This routine initializes the control record if it has not been initialized and then determines if the client one-time initialization routine has executed once. If it has not executed, this routine calls the initialization routine specified in init_routine().

If the client one-time initialization code has executed once, this routine returns. The pthread_once_t data structure is a record that allows client initialization operations to guarantee mutual exclusion of access to the initialization routine, and that each initialization routine is executed exactly once.

This variable must be initialized using the pthread_once_init macro, as follows:

```
pthread_once_t key_once = PTHREAD_ONCE_INIT;
```

Similarly, we have called the pthread_self() routine to get the unique identifier of the Pthread that is getting executed.

## 8.5.4 Thread-safe and reentrant functions

One very important point to take care of when building multi-threaded programs is the resource handling. To avoid getting in trouble, be sure to call only thread-safe and reentrant functions in multi-threaded applications. Reentrance and thread-safety are separate concepts: A function can be either reentrant, thread-safe, both, or neither.

**Reentrant** A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.

**Thread-safe** A thread-safe function protects shared resources from concurrent access by locks. Thread-safety concerns only the implementation of a function and does not affect its external interface. The use of global data is thread-unsafe. It should be maintained per-Pthread or encapsulated so that its access can be serialized.

Reentrant and thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within Pthreads. Thus, it is a good programming practice to always use and write reentrant and thread-safe functions.

Some of the standard C subroutines are non-reentrant, such as the ctime() and strtok() subroutines. The reentrant version of the subroutines typically have the name of the original subroutine with a suffix _r (underscore r).

When writing multi-threaded programs, the reentrant versions of subroutines should be used instead of the original version. For example, the following code fragment:

```
token[0] = strtok(string, separators);
i = 0;
do {
    i++;
    token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

must be replaced by the following code fragment in the multi-threaded programming:

```
char *pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
    i++;
    token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

Thread-unsafe libraries may be used by only one Pthread in a program. The uniqueness of the Pthread using the library must be ensured by the programmer; otherwise, the program will have unexpected behavior or may even crash.

In order to determine which sub routines provided by the AIX base operating system are reentrant or thread-safe, please refer to the “Threads Programming Guidelines” section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 8.6 Pthread cancellation

POSIX provides facilities for canceling Pthreads. The facilities enable a Pthread to cancel another specified Pthread within the same process. The facilities are aimed at allowing applications to cancel cooperating Pthreads.

A Pthread that is the target of cancellation must be cooperative in the following sense. Each Pthread controls its own cancelability state and type. The cancelability state can be enabled or disabled. For the enabled state, the type can be asynchronous (cancellation requests accepted at any time) or deferred (cancellation accepted only at designated cancellation points). By default, Pthreads are initialized with state enabled and type deferred. An uncooperative Pthread could disable cancellation, and thus thwart cancellation requests.

An application is expected to make cancellation graceful by specifying cancellation cleanup handlers. The cleanup handlers perform actions such as unlocking mutexes owned by the target Pthread (the Pthread being canceled), signaling conditions that the target Pthread may have caused to become true, releasing resources, and so on. These actions enable other Pthreads in the application to make progress after the cancellation occurs.

The Pthread cancellation facilities specified in POSIX include the following functions:

- Canceling execution of a specified Pthread using the pthread_cancel() routine. The cancellation occurs in accordance with the target Pthread's cancelability state and type. If and when cancellation does occur, cancellation cleanup handlers are invoked in last-in-first-out (LIFO) order, followed by thread-specific data destructor functions in unspecified order. Then the Pthread is terminated.
- Setting the cancelability state of the calling Pthread using the pthread_setcancelstate() routine to either enabled or disabled.
- Setting the cancelability type of the calling Pthread using pthread_setcanceltype() to either asynchronous or deferred.
- Creating a cancellation point using the pthread_testcancel() routine. To designate a point in its execution as a cancellation point, a Pthread simply makes a call to the pthread_testcancel() function at that point.
- Establishing cancellation cleanup handlers using the pthread_cleanup_push() and pthread_cleanup_pop() routines. The pthread_cleanup_push() function is used to identify a specified routine as a cleanup handler. Conceptually, the routine is pushed onto the top of the calling Pthread's cancellation cleanup stack. The pthread_cleanup_pop() function is used to remove and optionally execute the cleanup handler at the top of the cancellation cleanup stack.

In order to determine which subroutines provided by the AIX base operating system can be canceled (cancellation point(s) can occur when a Pthread is executing that subroutine), please refer to the "Thread Programming Guidelines" section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 8.7 Pthread priority and scheduling

By default, each Pthread within a process has its own a dynamic or *floating* priority on AIX. This mechanism ensures that all Pthreads on a system have an opportunity to run and not be *locked out* by other Pthreads. As individual Pthreads consume CPU time, they are charged. Pthreads that are not ready to run, and that are not given a processor upon which they consume cycles, are not charged. Over time, CPU intensive Pthreads use more CPU cycles, are charged more, and their priority degrades. Pthreads that run only briefly before being preempted or relinquishing control of the processor tend to retain a fairly constant priority. The net result is that I/O intensive tasks stay near a constant priority, and computational tasks become lower in priority. The I/O intensive tasks usually run briefly. The computational tasks run for most or all of their allotted processor time. This allows all processes on the system to be responsive, ensures that every task is treated fairly with respect to CPU utilization, and prevents long-running tasks from *hogging* the system at the expense of others.

It is also possible to fix the priority of a Pthread at a specific value. This facility allows processes with Pthreads that must run at a certain *response level* to accomplish tasks in a predicable manner. While a normal process can fix the priority of a Pthread up to a certain level, at high priority levels the root user authority is required.

## 8.7.1 Thread models in AIX

Pthreads (user threads) are mapped to underlying kernel threads. A *thread model* refers to the way this mapping is done. A *light-weight process* (LWP) refers to a kernel thread.

There are three possible thread models, corresponding to three different ways to map Pthreads to kernel threads (see Figure 8-4 on page 323).

- The M:1 model[15]

  Also known as the *library thread scheduling,* it refers to the case where threads share a pool of available LWPs.

  In the M:1 model, all user threads run on one VP and are linked to exactly one LWP; the mapping is handled by a POSIX thread library scheduler. All user threads programming facilities are completely handled by the library. This model can be used on any system, especially on traditional single-threaded systems.

- The 1:1 model[16]

  The 1:1 model is sometimes referred to as *bound thread scheduling*. In this model, each user thread is bound to a VP and linked to exactly one kernel thread (LWP). The VP is not necessarily bound to a real CPU (unless binding to a processor was done). Each VP can be thought of as a virtual CPU available for executing user code and system calls. A thread that is bound to a VP is said to have *system scope*, because it is directly scheduled with all the

[15] This was a thread model on AIX Version 3.2 when DCE is installed.

[16] This was the thread model on AIX Version 4.1 and 4.2.

other kernel threads by the kernel scheduler. Most of the user threads programming facilities are directly handled by the kernel threads.

- The M:N model

  In the M:N model (this is also known as *multiplexed thread scheduling*), several user threads can share the same virtual processor or the same pool of VPs. A thread that is not bound to a VP is said to be a local or *process scope*, because it is not directly scheduled with all the other threads by the kernel scheduler. The Pthreads library will handle the scheduling of user threads to the VP and then the kernel will schedule the associated kernel thread. This is the most efficient and most complex thread model; the user threads programming facilities are shared between the threads library and the kernel threads.

  The mapping of user threads to kernel threads is done using virtual processors. A *virtual processor* (VP) is a library entity that is usually implicit. For a user thread, the virtual processor behaves as a CPU for a kernel thread. In the library, the virtual processor is a kernel thread or a structure bound to a kernel thread, that is, user threads sit on top of virtual processors, which are themselves on top of kernel threads.

**Note:** The M:N model is the default thread model on AIX, beginning with Version 4.3.

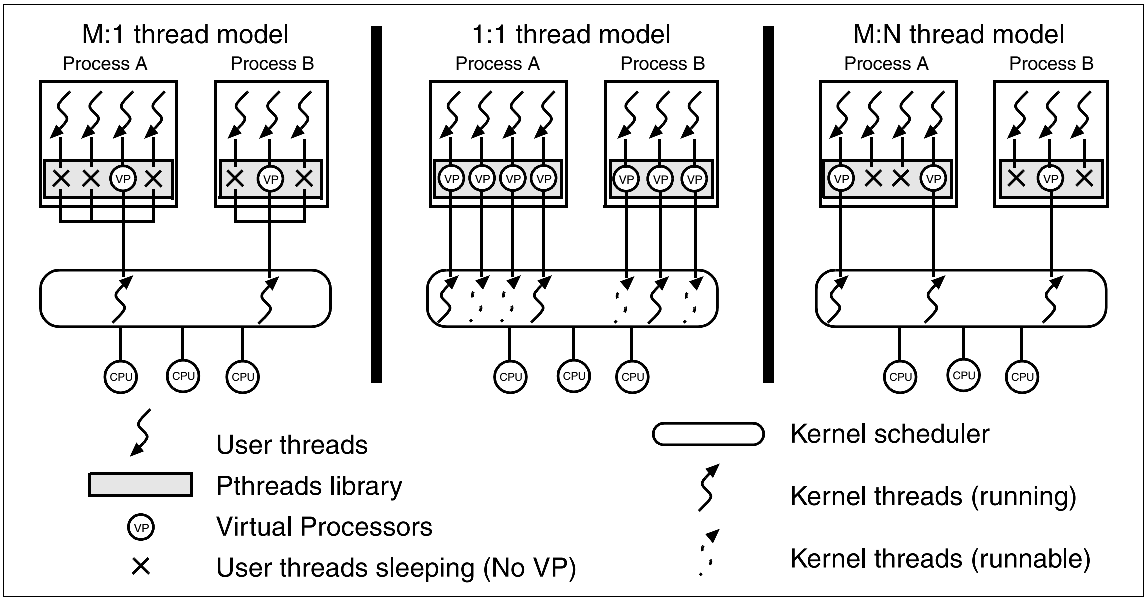


*Figure 8-4 Thread models*

### Comparing bound and multiplexed threads

Bound thread scheduling differs from multiplexed thread scheduling in the following important ways:

- A bound thread is permanently associated to its kernel thread, hence, it is exempt from the intermediate level of scheduling provided by the POSIX threads library used under the M:1 and M:N thread models.
- A bound thread executes exactly when its associated kernel thread is scheduled by the kernel scheduler.
- A bound thread's scheduling policy is related to the underlying kernel thread. In AIX, only kernel threads with root authority can use a fixed-priority scheduling policy.

Multiplexed thread scheduling has the following important characteristic: A multiplexed thread is subject to two levels of scheduling. First, the thread is assigned to a kernel thread and preempted by a POSIX thread library scheduler. Second, the kernel scheduler assigns the LWPs to processors and then preempts them.

## 8.7.2 Scheduling Pthreads

A kernel component, called scheduler, is used to share processor cycles among running processes. The kernel scheduler coordinates ready-to-run tasks of varying priorities, running at least 100 times every second to reevaluate tasks on the run queue and allow processes to run based on the scheduling policy.

Starting with Version 4.3, AIX supports Pthread scheduling at the process level. The user scheduler is provided by the Pthreads library and supports the M:N thread model on AIX. The Pthreads library scheduler has a pool of one or more *virtual processors* (VPs) upon which a user-space Pthread may run. You can think of a VP as a kernel-space thread. This mapping of the user thread to kernel thread provides more effective use of kernel resources for processes that create many threads, not all of which will be running at any given moment. For example, in a producer-consumer thread pair, we will often see only one task running while the other waits to send or receive data. The Pthread library scheduler itself is also created as a Pthread in a process and runs on its own VP.

The Pthreads library allows the programmer to control the execution scheduling of the Pthreads. The control can be performed in two different ways:

- By setting scheduling attributes when creating a Pthread.
- By dynamically changing the scheduling attributes of a created Pthread.

A Pthread has three scheduling parameters:

**Scope** The contention scope of a Pthread is defined by the thread model used in the Pthreads library.

**Policy** The scheduling policy of a Pthread defines how the scheduler treats the Pthread once it gains control of the CPU.

**Priority** The scheduling priority of a Pthread defines the relative importance of the work being done by each Pthread.

The scheduling parameters can be set before the Pthread's creation or during the Pthread's execution. In general, controlling the scheduling parameters of Pthreads is important only for Pthreads that are compute-intensive. Thus, the Pthreads library provides default values that are sufficient for most cases.

Controlling the scheduling of a Pthread is often a complicated task. Because the scheduler can handle all system or process-wide Pthreads, depending on the scope context, the scheduling parameters of a Pthread can interact with those of all other Pthreads in the process and in the other processes on the system.

## Scheduling policies

AIX also provides support for a number of scheduling policies. Each policy has certain characteristics that will benefit a specific workload, but not all policies are suitable for every task. We can create an application that assigns different policies to different Pthreads, tuning our application to make optimum use of the system. AIX provides the following policies at the system, or kernel, level:

- SCHED_FIFO
- SCHED_RR
- SCHED_OTHER
- SCHED_FIFO2
- SCHED_FIFO3

As long as a process is using process-level scope (that is, the M:N thread model is in effect), the process may modify the scheduling policy of the Pthreads within that process. If the process is using system-scope (the 1:1 model), then the root user authority is required to modify the scheduling or priority aspects of the Pthread. Root privileges are also required to modify the policy of a Pthread outside the scope of the process, or to set the priority of a Pthread or process to a fixed value.

### *SCHED_FIFO*

SCHED_FIFO is First-In/First-Out scheduling. A Pthread with fixed priority and SCHED_FIFO scheduling will be allowed to run to completion unless it:

- It yields the processor voluntarily.
- It blocks because something it needs is not available.
- A kernel thread with a higher priority is available to run.

Running Pthreads under a FIFO policy is discouraged because it prevents the system from balancing resource utilization in a way that lets all tasks run.

### *SCHED_RR*

SCHED_RR is round-robin scheduling. Pthreads using this policy will receive a time slice (time slicing is a mechanism that ensures that every Pthread is allowed time to execute by preempting running Pthreads at fixed intervals) of the CPU. Only Pthreads with a fixed priority can take advantage of round-robin scheduling.

As each Pthread completes its allotted time on the processor, it is placed at the end of the queue. The Pthread at the front of the queue is allowed to run. This circular sharing of the processor ensures that systems that require a given number of Pthreads to execute on a regular basis will not *starve* any task.

### *SCHED_OTHER*

SCHED_OTHER is the default scheduling policy used by AIX. Unlike first-in/first-out and round-robin, the SCHED_OTHER policy is used to manage Pthreads that have dynamic priority. In this policy the scheduler runs 100 times per second, and at each interval charges the Pthread(s) running on a processor(s) for its use of the CPU. Floating priorities are computed by using the Pthread priority, the CPU usage, and various tunable kernel parameters.

By default, a Pthread is created with a priority of 40 plus its nice value. The default nice value is 20 and can be changed within a relative range of plus or minus 20 using the `nice` command. This results in a default priority for a newly created Pthread of 60. If the Pthread has control of a processor when the scheduler next runs, the Pthread is charged and its priority is recalculated. The formula used to recalculate Pthread priority is:

recalculated_priority = P_nice + (C * R / 32)

Where:

**P_nice** 40 + NICE.

**C (recent CPU usage)** Old CPU Usage * D / 32 (default D is 16 and ranges from 0-32).

**R / 32** The CPU penalty factor, where R is a kernel tunable and ranges from 0-32.

The values for D and R are tunable. D defaults to 16 and may range from 0 to 32; R also defaults to 16 and ranges in value from 0 to 32.

### *SCHED_FIFO2*

SCHED_FIFO2 is similar to SCHED_FIFO, but allows a Pthread that has slept for a short period of time to be added to the front of the run queue when it awakened. The time period is a kernel parameter that can be modified. This affinity limit is adjusted using the **schedtune** command:

```
/usr/samples/kernel/schedtune -a N
```

Where *N* is the number of clock ticks.

### *SCHED_FIFO3*

SCHED_FIFO3 will always place an awakened Pthread at the front of the queue, regardless of how long it was asleep. While this is similar to FIFO2, we should remember that when a FIFO3 Pthread is on a run queue, the parameters of the queue are modified to prevent a FIFO2 Pthread from being put at the head of the queue.

## Setting Pthread scheduling policies

Scheduling policies are inherited from the process that created the Pthread. If a different scheduling policy is required, the pthread_attr_setinheritsched() routine can be used to set the thread attribute prior to Pthread creation. Again, this applies to Pthreads in a process that uses the M:N model:

```
pthread_attr_setinheritsched(pthread_attr_t *thread_attr, int value)
```

Where:

**thread_attr** Thread attribute structure or object

**Value** Either PTHREAD_INHERIT_SCHED or PTHREAD_EXPLICIT_SCHED

If the Pthread is already created, the pthread_setschedparam() routine can be used to modify a Pthread's scheduling policy. The scheduling change is not realized until the next time the Pthread is put on the run queue. That is, if a Pthread is currently running and a call is made to pthread_setschedparam(), the change will not become effective until the Pthread has consumed its time slice:

```
pthread_setschedparam(pthread_t thread_id, int policy, *param)
```

Where:

**thread_id** Thread ID to operate on.

**policy** Scheduling policy, for example, SCHED_RR or SCHED_FIFO.

**param** This structure contains a policy field but is ignored.

### 8.7.3 Scheduling limitations

Thread scope affects how the underlying kernel thread's scheduling will behave. For system scope threads, there exists one kernel thread for every user thread, that is, pthread. Modifying a system scope thread's scheduling policy modifies the underlying kernel thread's scheduling policy; thus, system scope threads are limited to SCHED_OTHER unless the process has root privileges. Also, in the case of system scope threads, the other scheduling policies (FIFO and round-robin) require a fixed priority and root is the only user who can modify a thread's priority.

As stated earlier, *process scope* threads are threads that have M user threads to N kernel threads. The scheduling policy for process scope threads is handled at the library level and does not affect the underlying kernel thread(s). At the process level, the scheduling policy affects how the ready-to-run Pthreads are placed in the run queue. The scheduling policy of the underlying kernel thread (which implements the virtual processor, or VP) is managed by the kernel.

## 8.8 Pthread specific environment variables in AIX

Tuning multi-threaded applications for optimal performance can be challenging on any system. AIX provides a set of environmental variables that modify the behavior of various threading functions. Let us look at some of these variables one at a time.

### AIXTHREAD_SCOPE

This environment variable controls the mapping of the application-level Pthreads to entries in the OS scheduling queue. The default thread scope for AIX is process, and this applies to the creation of Pthreads only when the attribute argument to pthread_create() is NULL; otherwise, the contents of the attribute structure determines the scope of the Pthread. This implies that a single process may contain Pthreads of both scopes.

AIXTHREAD_SCOPE permits exploring application behavior by being able to change the thread scope without having to modify the application. The default thread scope can be set by setting the variable in the environment:

```
AIXTHREAD_SCOPE=[P|S]
```

Where:

|   |   |
|---|---|
| **P** | For process scope |
| **S** | For system scope |

The preferred setting for any particular application is dependent upon its behavior. The performance difference between process and system scope, in current levels of AIX 5L, has become almost negligible for certain applications.

## AIXTHREAD_MNRATIO

This environment variable controls the scaling factor used within the Pthread library when creating and terminating Pthreads. The default M:N ratio for current levels of the operating system is 8:1, that is, eight Pthreads for every kernel thread. This default ratio may not be suitable for all workloads, however, so the user can modify the M:N ratio by setting and exporting the environmental variable as follows:

```
AIXTHREAD_MNRATIO=M:N
```

Where:

| | |
|---|---|
| **M** | Number of user threads |
| **N** | Number of kernel threads |

AIXTHREAD_MNRATIO applies only to process-scope threads. It may be useful for applications with a very large number of Pthreads.

## SPINLOOPTIME

SPINLOOPTIME controls the number of times a Pthread will attempt to obtain a mutex before blocking on the mutex availability. For example, sometimes it makes more sense for a Pthread to keep trying just a bit longer to acquire a lock in hopes of the other Pthread releasing the lock while the current Pthread is awake and waiting. This can avoid the overhead of putting the Pthread to sleep, managing the event that occurs when the lock is released, and waking the Pthread up and setting it to run so that it can now acquire the lock.

On multiprocessor systems the default value for SPINLOOPTIME is 40. The spin loop time can be modified by setting and exporting the environmental variable:

```
SPINLOOPTIME=N
```

Where:

| | |
|---|---|
| **N** | Number of times to retry a busy lock before yielding to another Pthread. |

## YIELDLOOPTIME

Similar to SPINLOOPTIME, YIELDLOOPTIME controls the number of times a Pthread will yield itself before blocking on a mutex. In some cases, yielding the processor will permit other Pthreads time to run, thus reducing the amount of time the waiting Pthread will have to spin and/or wait for the mutex to be unlocked. This allows work to be accomplished while one Pthread is waiting for a resource to be available. It also keeps the Pthread from going to sleep; instead, the Pthread is put back on the run queue to try again to acquire the lock. Keeping the Pthread from going to sleep can result in improved overall performance, depending upon how much more time the Pthread needs to wait for the lock acquisition.

The default for YIELDLOOPTIME is 0. If the Pthread spins on the lock and cannot get it, it will be put to sleep. Increasing the value of this variable will prevent the Pthread from going to sleep quite so quickly. We can modify the value of yield loop time by setting and exporting the environmental variable:

```
AIXTHREAD_YIELDLOOPTIME=N
```

Where:

**N** Number of times to yield to acquire a busy mutex or spin lock

## AIXTHREAD_MINKTHREADS

Sets the minimum number of kernel threads that should be used for a process. The default is eight, which means that an average threaded process will have at least eight kernel threads available as a resource for scheduling Pthreads. This environment variable can be set as follows:

```
AIXTHREAD_MINKTHREADS=N
```

Where:

**N** Number of kernel threads

## AIXTHREAD_MUTEX_DEBUG

AIX is capable of maintaining, in running threaded processes, information regarding mutexes; this information is then available for application debugging. On current levels of AIX 5L, the default setting for this variable is OFF, but that was not always the case. Since maintaining this information is not free, we may observe unexplainable and undesirable performance characteristics. If we suspect that our application may be impacted, set this variable in our environment to ensure mutex debugging information is not being collected:

```
AIXTHREAD_MUTEX_DEBUG=[ON|OFF]
```

If you plan on debugging the Pthreads in your application, you will want to turn this variable ON.

## AIXTHREAD_COND_DEBUG

Similar to AIXTHREAD_MUTEX_DEBUG, this variable controls the collection of debug information for condition variables. Its default setting is now OFF. For the same conditions described above, we can set the variable in our environment:

```
AIXTHREAD_COND_DEBUG=[ON|OFF]
```

## AIXTHREAD_RWLOCK_DEBUG

Causes the Pthreads library to maintain a list of all read-write locks that can be viewed by debugging programs. Like the other two debugging variables above, the default for this is now OFF. Set the variable in your environment to ON for debugging the application.

```
AIXTHREAD_RWLOCK_DEBUG=[ON|OFF]
```

## AIXTHREAD_GUARDPAGES

Each Pthread is created with its own stack; the stack size is controlled via the thread attributes or via (for a default value) an environment variable. Since the stack is allocated in the process heap, there is some potential for inadequately sized stacks to allow a Pthread to cause a stack to grow beyond its maximum size. To assist in detecting this behavior, and also to guard against errant memory writes, the stack can be protected by setting pages at the top of the stack as read-only. Any attempt to write onto these pages will result in the application receiving a segmentation violation signal, which will occur immediately. Debugging the application will allow us to investigate the conditions at the time of the *stack overflow* and decide on an appropriate course of action (increase the stack size, redesign part of the application, change where the data is stored, and so on).

To enable and set the number of guard pages, we need to export the environmental variable:

```
AIXTHREAD_GUARDPAGES=N
```

Where:

| | |
|---|---|
| **N** | Number of 4 KB size pages |

The default value is 0, which means no guard pages are created. If the application specifies its own stack, or uses large pages[17] for its process heap, no guard pages are created.

## AIXTHREAD_SLPRATIO

The “sleep ratio” value tells the system how many kernel threads should be held in reserve for sleeping Pthreads. This tuning parameter allows greater

[17] See 3.6, “Large page support” on page 157.

management of kernel resources, as sleeping Pthreads do not require a matching kernel thread. And since Pthreads are (usually) woken one at a time, the scheduling and running of the Pthreads can be matched to existing resources without needing to acquire additional kernel threads.

One of the control variables that impact the scheduling of Pthreads is created with a process based contention scope. This environment variable will be set as follows:

```
AIXTHREAD_SLPRATIO=k:p
```

Where:

**k** The number of kernel threads that should be held in reserve for p sleeping pthreads

Any positive integer value may be specified for p and k. If k > p, then the ratio is treated as 1:1 (that is, you cannot specify more kernel threads than Pthreads). The default sleep ratio is 1:12.

### AIXTHREAD_STK

Each Pthread has its own stack; the default size (on current levels of AIX) is 96 KB for 32-bit applications and 192 KS for 64-bit applications. While we will find this default adequate for many applications, it is also true that there are exceptions. Instead of modifying code to adjust a parameter in the Pthread attribute structure and then recompiling and rebuilding our application, we can use this environment variable to specify stacks of up to 256MB in size. Keep in mind, however, that Pthreads stacks are created in the process heap, and thus impact the amount of space available for dynamically allocated memory.

We can modify the stack size for Pthreads created without specifying the stack size programmatically by exporting the environment variable:

```
AIXTHREAD_STK=N
```

Where:

**N** number of bytes

## 8.9 User API for Solaris threaded applications

The new user thread library provides for source compatibility with Solaris user thread routines. This allows applications that are run on Solaris systems to be recompiled without changing their application source code so that they can run on AIX beginning with Version 5.2. The API for Solaris threaded applications for

AIX 5L Version 5.2 is designed to be compatible with Solaris Version 8 of the thread library.

The Solaris user thread library does not alter the Pthread library, so compatibility with POSIX and X/Open standards for Pthreads are maintained on AIX. The Solaris user threads support is implemented on top of the Pthreads; therefore, existing POSIX thread applications will not be affected by the Solaris user thread support.

There is, however, no binary compatibility with applications compiled under Solaris. All source code is required to be recompiled on AIX.

> **Note:** Solaris user thread library function names have the prefix *thr_*, while POSIX thread library function names have the prefix *pthread_*.

## 8.9.1 Application binary interface (ABI)

The design of the existing ABI of the pthread library is not altered with respect to:

- Exported function names
- Exported function signatures
- Exported data structures
- Exported data structures used in file formats

### AIX LPP packaging

The filesets listed in Table 8-1 contain the AIX files needed for the user API for Solaris threaded applications.

*Table 8-1 Filesets for Solaris user thread library*

| File | Fileset |
|---|---|
| /usr/ccs/lib/libthread.a | bos.adt.lib |
| /usr/include/thread.h | bos.adt.include |
| /usr/include/synch.h | bos.adt.include |

There are no user interfaces required for either the command line, SMIT, or Web-based System Manager. All applications need to be recompiled.

# 9

# Program parallelization using OpenMP

The latest versions of the IBM C and C++ compiler products for AIX support OpenMP, which parallelizes your C and C++ applications. By inserting simple OpenMP directives[1] (start with “#pragma omp”) into your source code, your application can be parallelized and efficiently performs on SMP systems.

This chapter explains how to use the OpenMP support of the IBM C and C++ compiler products for AIX.

The OpenMP support on AIX internally uses the POSIX threads explained in Chapter 8, “Introduction to POSIX threads” on page 275. Although the application programmer are not exposed to the POSIX threads when programming C and C++ applications with OpenMP, several concepts in the POSIX threads programming model are sometimes beneficial.

For further information about the OpenMP support provided by the compiler products, please refer to *C for AIX Compiler Reference*, SC09-4960 and *VisualAge C++ for AIX Compiler Reference*, SC09-4959.

[1] In addition to the OpenMP directives, the IBM C and C++ compiler products for AIX also have been supporting IBM SMP directives to parallelize your C and C++ applications. See Appendix E, “Supported IBM SMP directives” on page 483 for more information about the IBM SMP directives.

# 9.1 Introduction to OpenMP

OpenMP is an industry specification describing a common set of APIs for multi-platform SMP programming. OpenMP is a portable, scalable programming model designed to provide SMP programmers with a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the super computer. It is comprised of three components, as shown in Table 9-1.

*Table 9-1 OpenMP components*

| Components | Relevant section(s) |
|---|---|
| Compiler directives | From 9.3, "Classification of OpenMP directives" on page 337 to 9.9, "Data-sharing attribute clauses" on page 355 |
| Run-time library functions | Section 9.10, "Run-time library functions" on page 363 |
| Environment variables | Section 9.11, "Environment variables" on page 373 |

For further information about OpenMP, please visit the following URL:

http://www.openmp.org

# 9.2 The OpenMP programming model

OpenMP uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives will be ignored and a stub library will be linked).

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the *master* thread. The master thread executes in a serial region until the first parallel construct is executed. In the OpenMP C/C++ API, the parallel directive constitutes a *parallel construct*. When a parallel construct is executed, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the *work-sharing constructs*. Work-sharing constructs must be executed by all threads in the team in the same order, and the statements within the associated structured block are executed by one or more of the threads.

# 9.3 Classification of OpenMP directives

The IBM C and C++ compiler products for AIX support OpenMP pragma directives that exploit shared memory parallelism.[2] Directives are based on the #pragma directive defined in the C and C++ standards. Compilers that support the OpenMP C and C++ API will include a command line option that activates and allows interpretation of all OpenMP compiler directives.

We categorized OpenMP pragmas into the four categories shown in Table 9-2.

*Table 9-2 OpenMP directive categories*

| Category | Brief explanation and related sections |
|---|---|
| Parallel constructs | These pragmas enable the programmer to define the parallel regions in which work is done by threads in parallel. Most of the OpenMP directives either statically or dynamically bind to an enclosing parallel region. See Section 9.4, “Parallel region construct” on page 338 for more information. |
| Work-sharing constructs | This category of pragmas enables the programmer to define how work will be distributed across the threads in a parallel region. See Section 9.5, “Work-sharing constructs” on page 340 for more information. |
| Synchronization constructs | This category enables the programmer to control synchronization among threads. See Section 9.7, “Synchronization constructs” on page 348 for more information. |
| Data sharing attributes | This category of pragmas enables the programmer to define the private/shared context of data within a parallel region. See Section 9.9, “Data-sharing attribute clauses” on page 355 for more information. |

**Note:** Throughout this chapter, the term *construct* refers to the pragma line and the following single code block. If a pragma line does not have any following code blocks, then we use the term *directive*.

## 9.3.1 The OpenMP directive format

The syntax of an OpenMP directive is specified as follows:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

[2] VisualAge C++ for AIX supports OpenMP since Version 6, while C for AIX has been supporting it since Version 5.

Each directive starts with “#pragma omp”, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) pragma directives with the same names. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the #, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the #pragma omp are subject to macro replacement.

Directives are case-sensitive. The order in which clauses appear in directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If a variable-list appears in a clause, it must specify only variables. Only one directive-name can be specified per directive.

For example, the following directive is not allowed:

```
/* ERROR - multiple directive names not allowed */
#pragma omp parallel barrier
```

An OpenMP directive applies to at most one succeeding statement, which must be a structured block. For those directives, which allow a structured block following the directive name, are termed Constructs. Throughout this chapter, the term Construct is used as appropriate.

## 9.4 Parallel region construct

The following directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel. This is the fundamental construct that starts parallel execution:

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
       structured-block
```

The clause is one of the following:

- if(scalar-expression)
- private(variable-list)
- firstprivate(variable-list)
- default(shared | none)
- shared(variable-list)
- copyin(variable-list)
- reduction(operator: variable-list)
- num_threads(integer-expression)

For a detailed description of the clauses, see 9.9, “Data-sharing attribute clauses” on page 355.

When a thread encounters a parallel construct, a team of threads is created if one of the following cases is true:

- No if clause is present.
- The if expression evaluates to a nonzero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads in the team, including the master thread, execute the region in parallel. If the value of the if expression is zero, the region is serialized.

The number of threads in a parallel region is determined by the following factors, in order of precedence:

1. Use of the omp_set_num_threads() library function.[3]
2. Setting of the OMP_NUM_THREADS environment variable.[4]
3. Implementation default.[5]

The sample code in Example 9-1 illustrates the parallel region execution. In this program, every thread executes and prints the "Hello World." enclosed in the parallel region.

*Example 9-1 omp_parallel.c*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Before forking a parallel region.\n");

    /* Fork a team of threads giving them their own copies of variables. */
    #pragma omp parallel
    {
        printf("Hello World.\n");
    } /* All threads join master thread and terminate. */

    printf("Exiting the program... Bye.\n");
}
```

To compile this code, run:

```
$ cc_r -qsmp=omp omp_parallel.c
```

[3] See 9.10.1, "Execution environment functions" on page 364.

[4] See 9.11.2, "OMP_NUM_THREADS" on page 374.

[5] On AIX, it is equivalent to the number of equipped processors.

**Note:**

- Use **cc_r** when compiling C programs with OpenMP directives. Use **xlC_r** when compiling C++ programs with OpenMP directives.
- The -qsmp=omp compiler option should be specified when compiling C and C++ programs with OpenMP directives. If not specified, a stub library (/usr/lib/libxlomp_ser.a), which does not create any threads, will be in-lined instead of the actual OpenMP library (/usr/lib/libxlsmp.a).

Before running this program, set the environment variable OMP_NUM_THREADS in order to set the number of threads that you want to execute in parallel. For example,

```
$ export OMP_NUM_THREADS=3 a.out
```

The output of the program is as follows:

```
Before forking a parallel region
Hello World.
Hello World.
Hello World.
Exiting the program... Bye.
```

As instructed by the environment variable, the program forked three threads and executed the parallel region construct.

## 9.5 Work-sharing constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

OpenMP defines the three work-sharing constructs, *for*, *sections*, and *single*, explained in the following sections:

- Section 9.5.1, “for construct” on page 341
- Section 9.5.2, “sections construct” on page 344
- Section 9.5.3, “single construct” on page 345

A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel. They must be executed by all members of a team or none at all. Successive work-sharing constructs must be executed in the same order by all members of a team.

### 9.5.1 for construct

The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated; otherwise, it executes in serial on a single processor.

The syntax for the for construct is as follows:

```
#pragma omp for [clause[[,] clause] ... ] new-line
   for-loop
```

The clause is one of the following:

- private(variable-list)
- firstprivate(variable-list)
- lastprivate(variable-list)
- reduction(operator: variable-list)
- ordered
- schedule(kind[, chunk_size])
- nowait

Let us discuss a few of the clauses in brief. For a detailed description of the other clauses, see 9.9, “Data-sharing attribute clauses” on page 355.

The *ordered* clause must be present when ordered directives bind to for construct.

The *schedule* clause describes how iterations of the loop are divided among the threads in the team. The *schedule kind* may be one of the following:

**static** Loop iterations are divided into pieces of *chunk_size* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

**dynamic** Loop iterations are divided into pieces of size chunk_size and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk_size is 1.

**guided** The chunk_size is exponentially reduced with each dispatched piece of the iteration space. The chunk_size specifies the minimum number of iterations to dispatch each time. The default chunk_size is 1.

**runtime** The scheduling decision is deferred until run time by the environment variable OMP_SCHEDULE. It is not allowed to specify a chunk_size for this clause.

If the *nowait* clause is specified*,* then threads do not synchronize at the end of the parallel loop. Threads proceed directly to the next statements after the loop.

The restrictions to the for directive are as follows:

- The for loop must be a structured block, and, in addition, its execution must not be terminated by a *break* statement.
- The values of the loop control expressions of the for loop associated with a for directive must be the same for all the threads in the team.
- The for loop iteration variable must have a signed integer type.
- Only a single schedule clause can appear on a for directive.
- Only a single ordered clause can appear on a for directive.
- Only a single nowait clause can appear on a for directive.
- The value of the chunk_size expression must be the same for all threads in the team.

The for directive requires that the for loop must have *canonical form*. In the OpenMP specifications, the canonical form allows the number of loop iterations to be computed on entry to the loop. For example, the following code fragment:

```
#pragma omp for reduction(+:overallsum)
for (n = 0; n != ARRSIZE; n++) {
    overallsum += array[n];
}
```

would yield the following error when it is compiled:

```
" 1506-818 (S) Controlling expression of the for loop is not in the canonical
form."
```

because the loop test expression `n != ARRSIZE`, should be replaced by `n < ARRSIZE`.

The example program shown in Example 9-2 on page 343 illustrates the use of the for directive. In this example, the arrays a, b, and total are shared by all threads. The variable i is private to each thread and each thread has its own copy of it. The iterations of the for is distributed dynamically in chunk_size pieces. The nowait clause ensures that threads will not synchronize upon completing their individual pieces of work.

*Example 9-2 omp_ws_for.c*

```
#include <stdio.h>
#define CHUNKSIZE 5
#define N         10

int main(int argc, char *argv[])
{
    int i, chunk_size;
    float a[N], b[N], total[N];

    /* Some initializations. */
    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk_size = CHUNKSIZE;

    #pragma omp parallel shared(a,b,total,chunk_size) private(i)
    {
        #pragma omp for schedule(dynamic,chunk_size) nowait
        for (i=0; i < N; i++)
            total[i] = a[i] + b[i];
    }  /* end of pragma omp parallel for. */
    for (i = 0; i < N; i++)
        printf("The total value is = %f\n", total[i]);
}
```

When executed, the program prints the output shown in Example 9-3.

*Example 9-3 Output fromomp_ws_for.c*

```
The total value is = 0.000000
The total value is = 2.000000
The total value is = 4.000000
The total value is = 6.000000
The total value is = 8.000000
The total value is = 10.000000
The total value is = 12.000000
The total value is = 14.000000
The total value is = 16.000000
The total value is = 18.000000
```

### 9.5.2 sections construct

The sections directive identifies a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team. The syntax of the sections directive is as follows:

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
   [#pragma omp section new-line]
      structured-block
   [#pragma omp section new-line
      structured-block ]
   ...
}
```

The clause is one of the following:

- private(variable-list)
- firstprivate(variable-list)
- lastprivate(variable-list)
- reduction(operator: variable-list)
- nowait

Each section is preceded by a section directive, although the section directive is optional for the first section. The section directives must appear within the lexical extent of the sections directive. There is an implicit barrier at the end of a sections construct, unless a nowait is specified.

Restrictions to the sections directive are as follows:

- A section directive must not appear outside the lexical extent of the sections directive.
- Only a single nowait clause can appear on a sections directive.

The example shown in Example 9-4 on page 345 illustrates the use of the sections directive. The program is a slight modification of Example 9-2 on page 343, which illustrated the use of the for directive. The first N/2 iterations of the for loop will be distributed to the first thread, and the rest will be distributed to the second thread. When each thread finishes its block of iterations, it proceeds with whatever code comes next (nowait).

*Example 9-4   omp_ws_section.c*

```
#include <stdlib.h>
#define N     10

int main(int argc, char *argv[])
{
    int i;
    float a[N], b[N], total[N];

    /* Some initializations. */
    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    #pragma omp parallel shared(a,b,total) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i = 0; i < N / 2; i++)
                total[i] = a[i] + b[i];
            #pragma omp section
            for (i = N / 2; i < N; i++)
                total[i] = a[i] + b[i];
        } /* end of pragma omp sections nowait. */
    } /* end of pragma omp parallel shared. */
    for (i = 0; i < N; i++)
        printf("The total value is = %f\n", total[i]);
}
```

When executed, the program prints the same output shown in Example 9-3 on page 343.

### 9.5.3 single construct

The single directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). This may be useful when dealing with sections of code that are not thread safe (such as I/O). The syntax of the single directive is as follows:

```
#pragma omp single [clause[[,] clause] ...] new-line
   structured-block
```

The clause is one of the following:

- private(variable-list)
- firstprivate(variable-list)
- copyprivate(variable-list)

- nowait

Restrictions to the single directive are as follows:

- Only a single nowait clause can appear on a single directive.
- The copyprivate clause must not be used with the nowait clause.
- single constructs cannot be nested within sections construct.

The following example program illustrates the use of a single construct:

```
/* File : omp_ws_single.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Before forking a parallel region.\n");

    /* Fork a team of threads giving them their own copies of variables. */
    #pragma omp parallel
    {
        /* The printf() line is executed only once by any of the thread. */
        #pragma omp single
        printf("Before printing Hello World...\n");

        /* The printf() line is executed by all the thread. */
        printf("Hello World.\n");

        /* The printf() line is executed only once by any of the thread. */
        #pragma omp single
        printf("Done.. Leaving.\n");
    }
}
```

If executed, the program prints the following output:

```
Before forking a parallel region.
Before printing Hello World...
Hello World.
Done.. Leaving.
Hello World.
Hello World.
```

In the above output, the statements in the single construct were executed only once by one of the threads from the team.

## 9.6 Combined parallel work-sharing constructs

Combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a *parallel* directive followed by a single work-sharing construct.

The following sections describe the combined parallel work-sharing constructs:

- Section 9.6.1, “parallel for construct” on page 347
- Section 9.6.2, “parallel sections construct” on page 348

### 9.6.1 parallel for construct

The parallel for directive is a shortcut for a parallel region that contains only a single for directive. The syntax of the parallel for directive is as follows:

```
#pragma omp parallel for [clause[[,] clause] ...] new-line
   for-loop
```

This directive allows all the clauses of the parallel directive and the for directive, except the nowait clause, with identical meanings and restrictions. The semantics are identical to explicitly specifying a parallel directive immediately followed by a for directive.

The following example illustrates the use of the parallel for construct. Iterations of the for loop will be distributed in equal sized blocks to each thread in the team (*schedule static*):

```
/* File : omp_parallel_for.c */
#include <stdlib.h>
#define N       10
#define CHUNKSIZE   5

int main(int argc, char *argv[])
{
    int i, chunk_size;
    float a[N], b[N], total[N];

    /* Some initializations. */
    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk_size = CHUNKSIZE;

    #pragma omp parallel for \
        shared(a,b,total,chunk_size) private(i) \
        schedule(static,chunk_size)
    for (i = 0; i < N; i++)
```

```
        total[i] = a[i] + b[i];

    for (i = 0; i < N; i++)
        printf("Total value is = %f\n", total[i]);
}
```

When executed, the program prints the same output shown in Example 9-3 on page 343.

### 9.6.2 parallel sections construct

The *parallel sections* directive provides a shortcut form for specifying a parallel region containing only a single sections directive. The semantics are identical to explicitly specifying a parallel directive immediately followed by a sections directive. The syntax of the parallel sections directive is as follows:

```
#pragma omp parallel sections [clause[[,] clause] ...] new-line
{
   [#pragma omp section new-line]
      structured-block
   [#pragma omp section new-line
      structured-block ]
   ...
}
```

The clause can be one of the clauses accepted by the parallel and sections directives, except the nowait clause.

## 9.7 Synchronization constructs

Synchronization refers to the time order in which threads access shared or global variables. Because several parallel threads may need to access or update the same shared variable at about the same time, explicit control of synchronization is often required. A program with faulty synchronization may compile and run without incident, but will produce incorrect results.

OpenMP provides a variety of synchronization constructs, explained in the following sections, that control how the execution of each thread proceeds relative to other team threads:

- Section 9.7.1, “master construct” on page 349
- Section 9.7.2, “critical construct” on page 349
- Section 9.7.3, “barrier directive” on page 350
- Section 9.7.4, “atomic construct” on page 352

- Section 9.7.5, “flush directive” on page 352
- Section 9.7.6, “ordered construct” on page 353

## 9.7.1 master construct

The *master* directive identifies a construct that specifies a structured block that is executed by the master thread of the team. The syntax of the master directive is as follows:

```
#pragma omp master new-line
   structured-block
```

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to or exit from the master construct.

It is not allowed to branch into or out of master block.

## 9.7.2 critical construct

The *critical* directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. The syntax of the critical directive is as follows:

```
#pragma omp critical [(name)] new-line
   structured-block
```

An optional name may be used to identify the critical region. In the following sample code, all threads in the team will attempt to execute in parallel; however, because of the critical construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time:

```
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int x = 0;
    #pragma omp parallel shared(x)
    {
        /* access allowed for one thread only at a time. */
        #pragma omp critical
        x = x + 1;
    } /* end of pragma omp parallel shared. */
}
```

A small example program omp_reduction2.c, illustrating the use of the critical directive, is shown in 9.9.6, “reduction clause” on page 358.

### 9.7.3 barrier directive

The *barrier* directive synchronizes all the threads in a team. When executed, each thread in the team waits until all of the others have reached this point (see Figure 9-1). In this figure, the thread 2 has not been reached at the barrier statement; therefore, the other threads cannot proceed their execution flow.

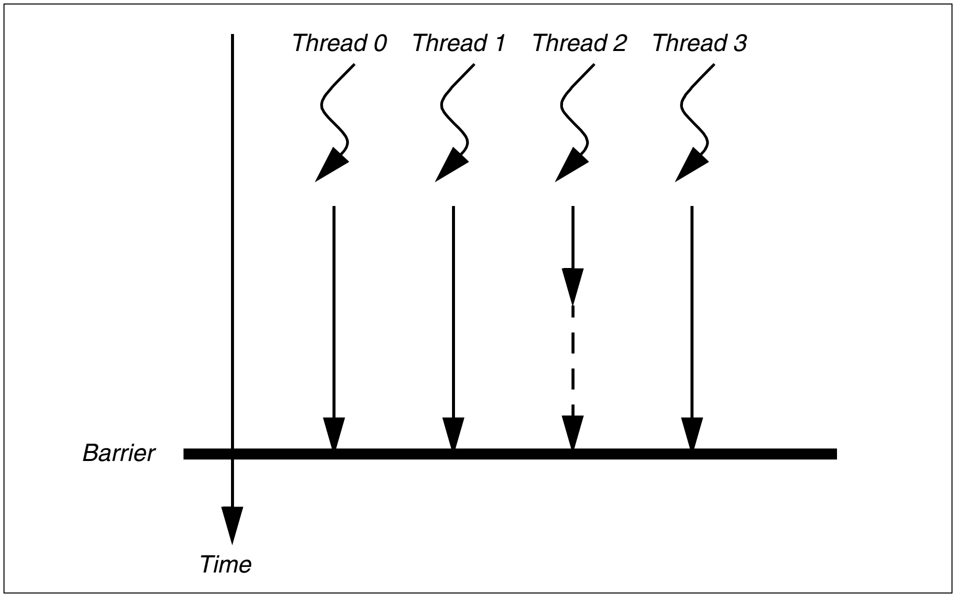


*Figure 9-1 Concept of barrier*

The syntax of the barrier directive is as follows:

```
#pragma omp barrier new-line
```

After all threads in the team have executed the barrier, each thread in the team begins executing the statements after the barrier directive in parallel.

In the following program, though the printf() statement is inside the parallel construct, the value of x is printed only once after all the threads have finished incrementing the variable. It is because the inclusion of the barrier directive has made it possible:

```
/* File : omp_barrier.c */
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
        /*
         * The following block will be executed after all the threads finish
         * their work.
         */
        #pragma omp barrier
        {
            #pragma omp single
            printf("The value of x is : %d\n", x);
        } /* end of pragma omp barrier */
    } /* end of pragma omp parallel shared. */
}
```

The output is printed as:

```
The value of x is : 3
```

If we remove the barrier construct from the code, we might not have achieved the expected results. The output would have been any of the following lines:

```
The value of x is : 1
The value of x is : 2
The value of x is : 3
```

depending upon the thread that gets a chance to execute the single block.

> **Note:** Because the barrier directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The example below illustrates these restrictions:
>
> ```
> /*
>  * ERROR - The barrier directive cannot be the immediate substatement of
>  * an if statement.
>  */
> if (x != 0)
>     #pragma omp barrier
>     your_statement_line;
> ...
>
> /* OK - The barrier directive is enclosed in a compound statement. */
> if (x != 0) {
>     #pragma omp barrier
>     your_statement_line;
> }
> ...
> ```

## 9.7.4 atomic construct

The *atomic* directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

The syntax of the atomic directive is as follows:

```
#pragma omp atomic new-line
   expression-stmt
```

The expression statement must have one of the following forms:

- x binop= expr
- x++
- ++x
- x--
- --x

Where:

- x is an lvalue expression with scalar type.
- expr is an expression with scalar type, and it does not reference the object designated by x.
- binop is not an overloaded operator and is one of +, *, -, /, &, ^, |, <<, or >>.

Only the load and store of the object designated by x are atomic; the evaluation of expr is not atomic. To avoid race conditions, all updates of the location in parallel should be protected with the atomic directive, except those that are known to be free of race conditions.

## 9.7.5 flush directive

The *flush* directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point. The syntax of the flush directive is as follows:

```
#pragma omp flush [(variable-list)] new-line
```

The optional variable list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, note that the pointer itself is flushed, not the object it references to.

Note that because the flush directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The example below illustrates these restrictions:

```
/*
 * ERROR - The flush directive cannot be the immediate substatement of
 * an if statement.
 */
if (x != 0)
    #pragma omp flush (x)
    your_statement_line;
...
/* OK - The flush directive is enclosed in a compound statement. */
if (x != 0) {
    #pragma omp flush (x)
    your_statement_line;
}
...
```

A variable specified in a flush directive must not have a reference type provided by the C++ language. The flush directive only provides consistency between the executing thread and global memory. To achieve a globally consistent view across all threads, each thread must execute a flush operation.

## 9.7.6 ordered construct

The *ordered* directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. The syntax of the ordered directive is as follows:

```
#pragma omp ordered new-line
   structured-block
```

An ordered directive can only appear in the dynamic extent of the for or parallel for directive. Only one thread is allowed in an ordered section at any time. It is illegal to branch into or out of an ordered block. A loop which contains an ordered directive must be a loop with an ordered clause. The following code fragment illustrates the use of the ordered directive:

```
#pragma omp for ordered
for (i = 0; i  <n; i++) {
    ...
    if (i <= 10) {
    ...
    #pragma omp ordered
    { ... }
    }
    ...
    if (i > 10) {
        ...
        #pragma omp ordered
        { ... }
    }
    ...
}
```

## 9.8 Data environment: The threadprivate directive

The *threadprivate* directive is provided to make file-scope, namespace-scope, or static block-scope variables local to a thread. The syntax of the threadprivate directive is as follows:

```
#pragma omp threadprivate(variable-list) new-line
```

Each copy of a threadprivate variable is initialized once, at an unspecified point in the program prior to the first reference to that copy, and in the usual manner (that is, as the master copy would be initialized in a serial execution of the program).

As with any private variable, a thread must not reference another thread's copy of a threadprivate object. During serial regions and master regions of the program, references will be to the master thread's copy of the object.

After the first parallel region executes, the data in the threadprivate objects is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads remains unchanged for all parallel regions.

The following code sample illustrates the use of the threadprivate variable:

```
/* File: omp_th_private.c */

#include <stdlib.h>
int a, b;
#pragma omp threadprivate(a)

int main(int argc, char *argv[])
{
    /* First parallel region. */
    #pragma omp parallel private(b)
    a = b = 5;

    /* Second parallel region. */
    #pragma omp parallel
    {
        printf("The value of a is %d\n", a);
        printf("The value of b is %d\n", b);
    }
}
```

When executed, the program prints the following output:

```
The value of a is 5
The value of b is 0
The value of a is 5
The value of b is 0
The value of a is 5
The value of b is 0
```

threadprivate variables differ from private variables because they are able to persist between different parallel sections of a code. That is the reason, in the above output, the value of a is 5 and the value of b is zero or undefined.

## 9.9 Data-sharing attribute clauses

Several directives accept clauses that allow a user to control the sharing attributes of variables for the duration of the region. Sharing attribute clauses applies only to variables in the lexical extent of the directive on which the clause appears. Not all of the following clauses are allowed on all directives. The list of clauses that are valid on a particular directive are described with the directive.

The following sections describe the data-sharing attribute clauses:

- Section 9.9.1, “private clause” on page 355
- Section 9.9.2, “firstprivate clause” on page 356
- Section 9.9.3, “lastprivate clause” on page 357
- Section 9.9.4, “shared clause” on page 357
- Section 9.9.5, “default clause” on page 358
- Section 9.9.6, “reduction clause” on page 358
- Section 9.9.7, “copyin clause” on page 362
- Section 9.9.8, “copyprivate clause” on page 363

### 9.9.1 private clause

The *private* clause declares the variables in variable-list to be private to each thread in a team. The syntax of the private clause is as follows:

```
private(variable-list)
```

The behavior of a variable specified in a private clause is as follows:

A new object with automatic storage duration is allocated for the construct. The size and alignment of the new object are determined by the type of the variable. This allocation occurs once for each thread in the team, and a default constructor is invoked for a class object (if any) if necessary; otherwise, the initial value is indeterminate. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

The restrictions to the private clause are as follows:

- A variable with a class type that is specified in a private clause must have an accessible, unambiguous default constructor.
- A variable specified in a private clause must not have a const-qualified type unless it has a class type with a mutable member.
- A variable specified in a private clause must not have an incomplete type or a reference type.
- Variables that appear in the *reduction* clause (explained in 9.9.6, “reduction clause” on page 358) of a parallel directive cannot be specified in a private clause on a work-sharing directive that binds to the parallel construct.

## 9.9.2 firstprivate clause

The *firstprivate* clause provides a superset of the functionality provided by the private clause. The syntax of the firstprivate clause is as follows:

```
firstprivate(variable-list)
```

The firstprivate clause combines the behavior of the private clause with automatic initialization of the variables in its list. The initialization or construction happens as if it were done once per thread, prior to the thread’s execution of the construct. For a firstprivate clause on a parallel construct, the initial value of the new private object is the value of the original object that exists immediately prior to the parallel construct for the thread that encounters it. For a firstprivate clause on a work-sharing construct, the initial value of the new private object for each thread that executes the work-sharing construct is the value of the original object that exists prior to the point in time that the same thread encounters the work-sharing construct. In addition, for C++ objects, the new private object for each thread is copy constructed from the original object.

The restrictions to the firstprivate clause are as follows:

- A variable specified in a firstprivate clause must not have an incomplete type or a reference type.
- A variable with a class type that is specified as firstprivate must have an accessible, unambiguous copy constructor.
- Variables that are private within a parallel region or that appear in the reduction clause of a parallel directive cannot be specified in a firstprivate clause on a work-sharing directive that binds to the parallel construct.

## 9.9.3 lastprivate clause

The *lasprivate* clause combines the behavior of the private clause with a copy from the last loop iteration or section to the original variable object. The syntax of the lastprivate clause is as follows:

```
lastprivate(variable-list)
```

The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

For example, the team member which executes the final iteration for a for section, or the team member which does the last section of a sections context performs the copy with its own values.

The restrictions to the lastprivate clause are as follows:

- All restrictions for private apply.
- A variable with a class type that is specified as lastprivate must have an accessible, unambiguous copy assignment operator.
- Variables that are private within a parallel region or that appear in the reduction clause of a parallel directive cannot be specified in a lastprivate clause on a work-sharing directive that binds to the parallel construct.

## 9.9.4 shared clause

This clause shares variables that appear in the variable-list among all the threads in a team. All threads within a team access the same storage area for shared variables. The syntax of the shared clause is as follows:

```
shared(variable-list)
```

It is the programmer's responsibility to ensure that multiple threads properly access shared variables (such as via critical sections).

## 9.9.5 default clause

The default clause allows the user to affect the data-sharing attributes of variables. The syntax of the default clause is as follows:

```
default(shared | none)
```

Specifying default(shared) is equivalent to explicitly listing each currently visible variable in a shared clause, unless it is threadprivate or const-qualified. In the absence of an explicit default clause, the default behavior is the same as if default(shared) were specified.

Specifying default(none) requires that at least one of the following must be true for every reference to a variable in the lexical extent of the parallel construct:

- The variable is explicitly listed in a data-sharing attribute clause of a construct that contains the reference.
- The variable is declared within the parallel construct.
- The variable is threadprivate.
- The variable has a const-qualified type.

Only a single default clause may be specified on a parallel directive.

## 9.9.6 reduction clause

This clause performs a reduction on the scalar variables that appear in variable-list, with the operator op. The syntax of the reduction clause is as follows:

```
reduction(op:variable-list)
```

A reduction is typically specified for a statement with one of the following forms:

- x binop= expr
- x = x op expr
- x = expr op x (except for subtraction)
- x++
- ++x
- x--
- --x

Where:

- x is one of the reduction variables specified in the list.
- expr is an expression with scalar type, and it does not reference the object designated by x.
- binop is not an overloaded operator and is one of +, *, -, &, ^, or |.

- op is not an overloaded operator, but one of +, *, -, &, ^, |, &&, or ||.

The term *reduction* refers to repeated updating of a variable via +, -, *, or a binary logical operator, with some other value. As will be apparent in the following, these binary operators must be commutative; division is not permitted as a reduction operator.

A reduction variable is a hybrid shared/private variable. It is initialized prior to the parallel construct, such as a shared or global variable. Within the parallel construct, it behaves as a private variable. But at the conclusion of the parallel construct, the threads' private values are amalgamated per the designated binary operation into the original shared variable.

Let us take the following example (Example 9-5). In this program, we are printing Hello inside the nested for loop of a parallel for construct. We are also printing the number of times the word Hello is printed. Let us look at the output of the program.

*Example 9-5 omp_reduction1.c*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0, j = 0;
    int result = 0;

    #pragma omp parallel for private(i)
    for (i = 0; i < 3; i++) {
        for (j = i + 1; j < 4; j++) {
            printf("Hello.\n");
            result = result + 1;
        }
    }

    printf("Number of times printed Hello = %d\n", result);
}
```

When executed, the program prints the following output:

```
Hello.
Hello.
Hello.
Hello.
Hello.
Hello.
Number of times printed Hello = 2
```

Alhough the word Hello is printed six times, the *result* variable is counted only twice when it should have been six. Clearly, a synchronization problem exists in the reading and writing of result. This problem can be resolved in three ways.

First, as we discussed earlier, the critical directive can be used to avoid the race condition to update the result variable, as shown in Example 9-6.

*Example 9-6 omp_reduction2.c*

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i = 0, j = 0;
    int result = 0;

    #pragma omp parallel for private(i)
    for (i = 0;i < 3; i++) {
        for (j = i+1; j <4; j++) {
            printf("Hello.\n");
            #pragma omp critical /* To avoid race condition. */
            result = result + 1;
        }
    }
    printf("Number of times printed Hello = %d\n", result);
}
```

When executed, the program prints the following output:

```
Hello.
Hello.
Hello.
Hello.
Hello.
Hello.
Number of times printed Hello = 6
```

Second is the use of the reduction clause. It supersedes the use of synchronization constructs in these situations. It not only prevents the race condition but also reduces the time of execution as it happens while using synchronization constructs. In Example 9-6, we may treat result as a reduction variable by modifying the parallel construct, as shown in Example 9-7 on page 361.

*Example 9-7 omp_reduction3.c*

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0, j = 0;
    int result = 0;

    /* reduction clause is used here for result. */
    #pragma omp parallel for private(i) reduction(+:result)
    for (i = 0; i < 3; i++) {
        for (j = i + 1 ; j < 4; j++) {
            printf("Hello.\n");
            result = result + 1;
        }
    }
    printf("Number of times printed Hello = %d\n",result);
}
```

When executed, the program prints the following output:

```
Hello.
Hello.
Hello.
Hello.
Hello.
Hello.
Number of times printed Hello = 6
```

For the third method of solving this problem using run-time library functions, see 9.10.2, “Lock functions” on page 368.

The restrictions to the reduction clause are as follows:

- The type of the variables in the reduction clause must be valid for the reduction operator, except that pointer types and reference types are never permitted.
- A variable that is specified in the reduction clause must not be const-qualified.
- Variables that are private within a parallel region or that appear in the reduction clause of a parallel directive cannot be specified in a reduction clause on a work-sharing directive that binds to the parallel construct, as shown in the following example.

```
#pragma omp parallel private(y)
{
    /* ERROR - private variable y cannot be specified in a reduction clause. */
    #pragma omp for reduction(+: y)
}
```

```
/* ERROR -
 * variable x cannot be specified in both a shared and a reduction clause.
 */
#pragma omp parallel for shared(x) reduction(+: x)
```

### 9.9.7 copyin clause

The *copyin* clause provides a mechanism to assign the same value to threadprivate variables for each thread in the team executing the parallel region. For each variable specified in a copyin clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the threadprivate copies at the beginning of the parallel region. The syntax of the copyin clause is as follows:

```
copyin(variable-list)
```

The following code sample illustrates this:

```
/* File : omp_copyin.c */
#include <include.h>
int a, b;
#pragma omp threadprivate(a)

int main(int argc, char *argv[])
{
    /* First parallel region. */
    a = 10; /* a is initialized before the parallel region. */
    /*
     * copies the value of ‘a’ by assignment, if copyin is not used, ‘a’ is
     * visible only to the master thread.
     */
    #pragma omp parallel private(b) copyin(a)
    b = 8;
    /* Second parallel region. */
    #pragma omp parallel
    {
        printf("The value of a is %d\n", a);
        printf("The value of b is %d\n", b);
    }
}
```

When executed, the program prints the following output:

```
The value of a is 10
The value of b is 0
The value of a is 10
The value of b is 0
The value of a is 10
The value of b is 0
The value of a is 10
The value of b is 0
```

The restrictions to the copyin clause are as follows:

- A variable that is specified in the copyin clause must have an accessible, unambiguous copy assignment operator (for C++).
- A variable that is specified in the copyin clause must be a threadprivate variable.

### 9.9.8 copyprivate clause

The *copyprivate* clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The copyprivate clause can only appear on the single directive. The syntax of the copyprivate clause is as follows:

```
copyprivate(variable-list)
```

Restrictions to the copyprivate clause are as follows:

- A variable that is specified in the copyprivate clause must not appear in a private or firstprivate clause for the same single directive.
- If a single directive with a copyprivate clause is executed in the dynamic extent of a parallel region, all variables specified in the copyprivate clause must be private in the enclosing context.
- A variable that is specified in the copyprivate clause must have an accessible unambiguous copy assignment operator (for C++).

## 9.10 Run-time library functions

In addition to the directives described in previous sections, OpenMP provides a set of run-time library functions. It includes run-time execution functions and lock functions. The run-time functions allow an application to specify the mode in which to run. An application developer may wish to maximize throughput performance of the system, rather than time to completion. In such cases, the developer may tell the system to dynamically set the number of processes used to execute parallel regions. This can have a dramatic effect on the throughput performance of a system with only a minimal impact on the time to completion for a program.

The run-time functions also allow a developer to specify when to enable nested parallelism. Enabling nested parallelism allows the system to act accordingly when it encounters nested parallel constructs. On the other hand, by disabling nested parallelism, a developer can write a parallel library that will perform in an easily predictable fashion, whether executed dynamically from within or outside a parallel region.

For further information about these library functions, please refer to the “Parallel Processing Support” section of the *VisualAge C++ for AIX Compiler Reference*, SC09-4959.

## 9.10.1 Execution environment functions

In this section, let us look at the following important OpenMP run-time functions supported on AIX. The functions described in this section affect and monitor threads, processors, and the parallel environment:

- omp_set_num_threads
- omp_get_num_threads
- omp_get_max_threads
- omp_get_thread_num
- omp_get_num_procs
- omp_set_dynamic
- omp_get_dynamic
- omp_in_parallel
- omp_set_nested
- omp_get_nested

An example program using these functions is provided in 9.10.3, “Example usage of run-time library functions” on page 370.

### omp_set_num_threads

The omp_set_num_threads() function sets the default number of threads to use for subsequent parallel regions that do not specify a num_threads clause. The syntax is as follows:

```
void omp_set_num_threads(int num_threads);
```

The dynamic threads mechanism modifies the effect of this routine.

- Enabled: Specifies the maximum number of threads that can be used for any parallel region.
- Disabled: Specifies exact number of threads to use until next call to this routine.

This routine can only be called from the serial portions of the code. This call has precedence over the OMP_NUM_THREADS environment variable (see 9.11, "Environment variables" on page 373).

### omp_get_num_threads

The omp_get_num_threads() function returns the number of threads currently in the team executing the parallel region from which it is called. The syntax is as follows:

```
int omp_get_num_threads(void);
```

The num_threads clause, the omp_set_num_threads() function, and the OMP_NUM_THREADS environment variable control the number of threads in a team. If the number of threads has not been explicitly set by the user, the default value is chosen. In AIX, it is the number of equipped processors on the system. This function binds to the closest enclosing parallel directive. If called from a serial portion of a program, or from a nested parallel region that is serialized, this function returns 1.

### omp_get_max_threads

It returns the maximum value that can be returned by calls to the omp_get_num_threads() function. The syntax is as follows:

```
int omp_get_max_threads(void);
```

This function returns the maximum value, whether executing from a serial region or from a parallel region. If a program uses omp_set_num_threads to change the number of threads, subsequent calls to omp_get_max_threads() will return the new value.

When the omp_set_dynamic() routine is set to TRUE, we can use omp_get_max_threads() to allocate data structures that are maximally sized for each thread.

### omp_get_thread_num

The omp_get_thread_num() function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and omp_get_num_threads() – 1, inclusive. The master thread of the team is thread 0. The syntax is as follows:

```
int omp_get_thread_num(void);
```

This function binds to the closest enclosing parallel directive. The function returns zero when called from a serial region or from within a nested parallel region that is serialized.

### omp_get_num_procs

The omp_get_num_procs() function returns the number of processors that are available to the program at the time the function is called. The syntax is as follows:

```
int omp_get_num_procs(void);
```

### omp_set_dynamic

The omp_set_dynamic() function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The syntax is as follows:

```
void omp_set_dynamic(int dynamic_threads);
```

To obtain the best use of system resources, certain run-time environments automatically adjust the number of threads that are used for executing subsequent parallel regions. This adjustment is enabled only if the value of the scalar logical expression to omp_set_dynamic is set to TRUE. If the value of the scalar logical expression is set as FALSE, dynamic adjustment is disabled.

When dynamic adjustment is enabled, the number of threads specified by the user becomes the maximum thread count. The number of threads remains fixed throughout each parallel region and is reported by omp_get_num_threads(). A call to omp_set_dynamic() has precedence over the OMP_DYNAMIC environment variable.

The default for dynamic thread adjustment is implementation dependent. In AIX, by default, dynamic adjustment is enabled. A user code that depends on a specific number of threads for correct execution should explicitly disable dynamic threads. Implementations are not required to provide the ability to dynamically adjust the number of threads, but they are required to provide the interface in order to support portability across platforms.

## omp_get_dynamic

The omp_get_dynamic() function returns a nonzero value if dynamic adjustment of threads is enabled, and returns 0 otherwise. The syntax is as follows:

```
int omp_get_dynamic(void);
```

This function returns 1 if dynamic thread adjustment is enabled; otherwise, it returns 0. The function always returns 0 if dynamic adjustment of the number of threads is not implemented.

## omp_in_parallel

The omp_in_parallel() function returns a nonzero value if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0. The syntax is as follows:

```
int omp_in_parallel(void);
```

The omp_in_parallel() function determines whether a region is executing in parallel. A parallel region that is serialized is not considered to be a region executing in parallel.

## omp_set_nested

The omp_set_nested() function enables or disables nested parallelism. The syntax is as follows:

```
void omp_set_nested(int nested);
```

If the value of the scalar logical expression is FALSE, nested parallelism is disabled, and nested parallel regions are serialized and executed by the current thread. This is the default.

## omp_get_nested

The omp_get_nested() function returns a nonzero value if nested parallelism is enabled and 0 if it is disabled. The syntax is as follows:

```
int omp_get_nested(void);
```

If an implementation does not implement nested parallelism, this function always returns 0.

> **Note:** In the current implementation, nested parallel regions are always serialized. As a result, omp_set_nested() does not have any effect, and omp_get_nested() always returns 0 on AIX.

## 9.10.2 Lock functions

The functions described in this section manipulate locks used for synchronization. For the following functions, the lock variable must have type omp_lock_t. This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to omp_lock_t type.

- omp_init_lock
- omp_destroy_lock
- omp_set_lock
- omp_unset_lock
- omp_test_lock

For the following functions, the lock variable must have type omp_nest_lock_t. This variable must only be accessed through these functions. All nestable lock functions require an argument that has a pointer to omp_nest_lock_t type.

- omp_init_nest_lock
- omp_destroy_nest_lock
- omp_set_nest_lock
- omp_unset_nest_lock
- omp_test_nest_lock

An example program using these functions is provided in 9.10.3, “Example usage of run-time library functions” on page 370.

### omp_init_lock and omp_init_nest_lock

These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter lock for use in subsequent calls. The syntax is as follows:

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero. It is noncompliant to call either of these routines with a lock variable that has already been initialized.

### omp_destroy_lock and omp_destroy_nest_lock

These functions ensure that the pointed to lock variable lock is uninitialized. The syntax is as follows:

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

It is noncompliant to call either of these routines with a lock variable that is uninitialized or unlocked.

### omp_set_lock and omp_set_nest_lock

Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function. The syntax is as follows:

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

For a simple lock, the argument to the omp_set_lock() function must point to an initialized lock variable. Ownership of the lock is granted to the thread executing the function. For a nestable lock, the argument to the omp_set_nest_lock() function must point to an initialized lock variable. The nesting count is incremented, and the thread is granted, or retains, ownership of the lock.

### omp_unset_lock and omp_unset_nest_lock

These functions provide the means of releasing ownership of a lock. The syntax is as follows:

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

The argument to each of these functions must point to an initialized lock variable owned by the thread executing the function. The behavior is undefined if the thread does not own that lock.

For a simple lock, the omp_unset_lock() function releases the thread executing the function from ownership of the lock. For a nestable lock, the omp_unset_nest_lock() function decrements the nesting count, and releases the thread executing the function from ownership of the lock if the resulting count is zero.

### omp_test_lock and omp_test_nest_lock

These functions attempt to set a lock but do not block execution of the thread. The syntax is as follows:

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

The argument must point to an initialized lock variable. These functions attempt to set a lock in the same manner as omp_set_lock() and omp_set_nest_lock(), except that they do not block execution of the thread.

For a simple lock, the omp_test_lock() function returns a nonzero value if the lock is successfully set; otherwise, it returns zero. For a nestable lock, the omp_test_nest_lock() function returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

## 9.10.3 Example usage of run-time library functions

This section provides two example source codes in order to demonstrate the usage of run-time library functions.

**Note:** Before calling any OpenMP run-time library functions, the omp.h header file must be included in the source code.

### Usage of run-time execution functions

The sample code shown in Example 9-8 illustrates the basic use of various run-time routines that we have discussed so far. It displays the values returned by the various run-time routines both in the serial and parallel region. For simplicity, the single directive is used to just print the values returned by one instance of a executing thread.

*Example 9-8 omp_runtime.c*

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    printf("Before forking a parallel region.\n");
    printf("---------------------------------\n");
    printf("omp_get_num_threads returns     %d\n", omp_get_num_threads());
    printf("omp_get_max_threads return  %d\n", omp_get_max_threads());
    printf("omp_get_thread_num returns  %d\n", omp_get_thread_num());
    printf("omp_get_num_procs returns   %d\n", omp_get_num_procs());
    printf("omp_get_dynamic returns     %d\n", omp_get_dynamic());
    printf("omp_in_parallel returns     %d\n", omp_in_parallel());
    printf("omp_get_nested returns      %d\n", omp_get_nested());
    printf("\nAfter forking a parallel region.\n");
    printf("---------------------------------\n");

    omp_set_num_threads(6);/* set the number of threads at run time to 10. */
    /* does not have any effect, just to illustrate that here. */
    omp_set_nested(1);

    #pragma omp parallel
    {
        #pragma omp single /* to print the values once. */
        {
            printf("omp_get_num_threads returns     %d\n"
                , omp_get_num_threads());
            printf("omp_get_max_threads return  %d\n", omp_get_max_threads());
            printf("omp_get_thread_num returns  %d\n", omp_get_thread_num());
            printf("omp_get_num_procs returns   %d\n", omp_get_num_procs());
            printf("omp_get_dynamic returns     %d\n", omp_get_dynamic());
            printf("omp_in_parallel returns     %d\n", omp_in_parallel());
            printf("omp_get_nested returns      %d\n", omp_get_nested());
        }
    } /* All threads join master thread and terminate. */
}
```

When executed, the program prints the following output:

```
Before forking a parallel region.
---------------------------------
omp_get_num_threads returns     1
omp_get_max_threads return      3
omp_get_thread_num returns      0
omp_get_num_procs returns       3
omp_get_dynamic returns         1
omp_in_parallel returns         0
omp_get_nested returns          0

After forking a parallel region.
---------------------------------
omp_get_num_threads returns     6
omp_get_max_threads return      6
omp_get_thread_num returns      4
omp_get_num_procs returns       3
omp_get_dynamic returns         1
omp_in_parallel returns         1
omp_get_nested returns          0
```

### Usage of lock functions

The race condition problem that we discussed in 9.9.6, “reduction clause” on page 358 can also be solved by using the lock functions discussed so far. In the modified version of the program shown in Example 9-9 on page 372, we lock and unlock the inner for loop section using the omp_set_lock() and omp_unset_lock() functions respectively. This prevents the simultaneous entry of threads into the for block; therefore, the variable result is incremented properly.

*Example 9-9 omp_lock.c*

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int i = 0, j = 0;
    int result = 0;
    omp_lock_t lock; /* lock variable to be initialized. */

    omp_init_lock(&lock); /* Initializes the lock before using it. */

    #pragma omp parallel for private(i)
    for (i = 0; i < 3; i++) {
        omp_set_lock(&lock); /* sets the lock here. */
        for (j = i + 1; j < 4; j++) {
            /*
             * already locked, therefore it doesn't block the execution of the
             * thread.
             */
            if (!omp_test_lock(&lock)) {
                printf("Hello.\n");
                result = result + 1;
            }
        }
        omp_unset_lock(&lock); /* releases the lock. */
    }
    printf("Number of times printed Hello = %d\n", result);
    omp_destroy_lock(&lock); /*Finally, destroys the lock */
}
```

When executed, the program prints the following output:

```
Hello.
Hello.
Hello.
Hello.
Hello.
Hello.
Number of times printed Hello = 6
```

This is the same output produced by Example 9-7 on page 361.

## 9.11 Environment variables

This section explains the OpenMP C and C++ API environment variables that control the execution of parallel code. The names of environment variables must be uppercase. The values assigned to them are case insensitive and may have leading and trailing white space. Modifications to the values after the program has started are ignored. The environment variables are as follows:

**OMP_SCHEDULE** Sets the run-time schedule type and chunk size.

**OMP_NUM_THREADS** Sets the number of threads to use during execution.

**OMP_DYNAMIC** Enables or disables dynamic adjustment of the number of threads.

**OMP_NESTED** Enables or disables nested parallelism.

For further information about these environment variables, please refer to the “Parallel Processing Support” section in the *VisualAge C++ for AIX Compiler Reference*, SC09-4959.

### 9.11.1 OMP_SCHEDULE

OMP_SCHEDULE applies only to for and parallel for directives that have the schedule type run time. The schedule type and chunk size for all such loops can be set at run time by setting this environment variable to any of the recognized schedule types and to an optional chunk_size.

For for and parallel for directives that have a schedule type other than run time, OMP_SCHEDULE is ignored. The default value for this environment variable is implementation-defined. If the optional chunk_size is set, the value must be positive. If chunk_size is not set, a value of 1 is assumed, except in the case of a static schedule. For a static schedule, the default chunk size is set to the loop iteration space divided by the number of threads applied to the loop.

The syntax of OMP_SCHEDULE is:

```
OMP_SCHEDULE=algorithm
```

where algorithm is one of the following:

- dynamic[,N]
- guided[,N]
- runtime
- static[,N]

If specified, the value of N must be an integer value of 1 or greater.

## 9.11.2 OMP_NUM_THREADS

The OMP_NUM_THREADS environment variable sets the default number of threads to use during execution, unless that number is explicitly changed by calling the omp_set_num_threads library routine or by an explicit num_threads clause on a parallel directive.

Its effect depends upon whether dynamic adjustment of the number of threads is enabled. If no value is specified for the OMP_NUM_THREADS environment variable, or if the value specified is not a positive integer, or if the value is greater than the maximum number of threads the system can support, the number of threads to use is implementation-defined. On AIX, it depends on the number of available processors.

The syntax of OMP_NUM_THREADS is:

```
OMP_NUM_THREADS=N
```

where N is the number of parallel threads requested, which must be a positive integer. The number can be overridden during program execution by calling the omp_set_num_threads() run-time library function.

## 9.11.3 OMP_DYNAMIC

The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The setting of this environment variable can be overridden by calling the omp_set_dynamic() run-time library function.

The syntax of OMP_DYNAMIC is:

```
OMP_DYNAMIC=[TRUE | FALSE]
```

If set to TRUE, the number of threads that are used for executing parallel regions may be adjusted by the run-time environment to best utilize system resources. If set to FALSE, dynamic adjustment is disabled. The default condition is implementation-defined.

## 9.11.4 OMP_NESTED

The OMP_NESTED environment variable enables or disables nested parallelism. The setting of this environment variable can be overridden by calling the omp_set_nested() run-time library function.

The syntax of OMP_NESTED is:

```
OMP_NESTED=[TRUE | FALSE]
```

If set to TRUE, nested parallelism is enabled; if it is set to FALSE, nested parallelism is disabled. The default value is FALSE.

# 10

# Dealing with C++ templates

This chapter explains how to deal with C++ templates on AIX for C++ application programmers by providing the following sections:

- Section 10.1, “What is a template” on page 378
- Section 10.2, “AIX template implementations” on page 378
- Section 10.3, “Simple code layout method” on page 381
- Section 10.4, “Template instantiation file method” on page 383
- Section 10.5, “Template registry: The preferred method” on page 387

Also, the ISO Standard C++ Library, including the Standard Template Library, is explained in the following section:

- Section 10.6, “Standard C++ Library and STL” on page 388

## 10.1 What is a template

A template defines a family of related classes or functions, where the parameters in the declaration describe how they are specialized. Templates are generic entities that can be instantiated to create specific user-defined types. The compiler generates new classes or functions when you supply arguments for the parameters. The class or function definition generated from a template with a set of template parameters is called a specialization.

Templates are an area of the C++ language that provide a great deal of flexibility for developers. The ISO C++ standard defines the language facilities and features for templates. Unfortunately, the standard does not specify how a compiler should implement templates. This means that there are sometimes significant differences between the methods used to implement templates in compiler products from different vendors.

Developers porting C++ code that uses templates to the AIX platform sometimes have problems with the implementation model. The main problems experienced are:

- Long compile and link times
- Linker warnings of duplicate symbols
- Increase in code and executable size

Up until VisualAge C++ for AIX Version 6.0, all of the above problems are generally caused by inefficient use of the compiler implementation of templates. The Version 6 compiler provides new options that handle templates more efficiently. The number of problems experienced will depend on the platform the code is being ported from and the template implementation method used on that platform. Sometimes, the problems can be fixed on AIX by simply adding a few compiler options. In other instances, the code layout needs to be changed in order to utilize the most efficient implementation method on AIX. In most of these rare cases, the code changes are backwards compatible with the original platform the code is being ported from. This is very important for developers who maintain a single source tree that must compile correctly on multiple platforms.

## 10.2 AIX template implementations

The template mechanism provides a way of defining general container types, such as list, vector, and stack, where the specific type of the elements is left as a parameter. Two types of templates can be defined:

**Class templates** Specify how individual classes can be constructed.
**Function templates** Specify how individual functions can be constructed.

Regardless of the type of template being used, the code is essentially split into three parts:

**Template declaration** This is the part of the source code that declares the template class or function. It does not necessarily contain the definition of the class or function, although it may optionally do so. For example, a template declaration may describe a Stack template class, as shown in Example 10-1.

**Template definition** This portion of code is the definition of the template function itself or the template class member functions. For example, using the Stack class template, this portion of code would define the member functions used to manipulate the stack, as shown in Example 10-2 on page 380.

**Template instance** The template instance code is generated by the compiler for each instance of the template. For example, this would be the code to handle a specific instance of the stack template class, such as a stack of integer values.

The difference between the components is that the template declaration must be visible in every compilation unit that uses the template. The template definition and code for each instance of the template need only be visible once in each group of compilation units that are linked together to make a single executable.

*Example 10-1 Stack template declaration*

```
template <class T> class stack
{
private:
    T* v;
    T* p;
    int sz;
public:
    stack(int);
    ~stack();
    void push(T);
    T pop();
};
```

*Example 10-2 Stack template member function definition*

```
template <class T> stack<T>::stack(int s)
{
    v = p = new T[sz=s];
}

template <class T> stack<T>::~stack()
{
    delete [] v;
}

template <class T> void stack<T>::push(T a)
{
    *p++ = a;
}

template <class T> T stack<T>::pop()
{
    T ret = *p;
    p--;
    return ret;
}
```

### 10.2.1 Generated function bodies

When you use class templates and function templates in your program, the compiler automatically generates function bodies for all template functions that are instantiated. The compiler follows four basic rules to determine when to generate template functions. The compiler applies the rules in the following order:

1. If a template declares a function to have internal linkage, the function must be defined within the same compilation unit. The compiler generates the function with internal linkage, and it is not shared with other compilation units. This is the case if the template class has in-line member functions.
2. If a template function is referenced in a compilation unit, but it is not declared to have internal linkage, the compiler looks for a definition of the function in the same compilation unit. If a definition is found, the function is instantiated. If the -qtemplateregistry option is in effect, and the function is already instantiated in another compilation unit, the function is not instantiated in this compilation unit.
3. If the -qtempinc option is in effect, the compiler creates a template instantiation file when the template function is declared but not defined. The template function is instantiated when the template instantiation file compiles.

4. If none of the preceding rules applies, the compiler does not generate the definition of the template function. It must be instantiated in another compilation unit.

## 10.3 Simple code layout method

The simplest method of using template code is to include both the declaration and definition of the template in every compilation unit that uses instances of the template. From a code layout point of view, this is very easy, since the template declaration and definition can be kept in a single header file. Using the stack example, the code in Example 10-1 on page 379 and Example 10-2 on page 380 would be combined into a single header file, for example, stack.h, which is then included by every compilation unit that wishes to use the template. Alternatively, the header file for a template declaration can include the source file that contains the template definition. Using the stack template example, the header file, stack.h, would #include the source file, stack.C.

There are a number of disadvantages to using this method. Some of them can be overcome; others can not.

### 10.3.1 Disadvantages of the simple method

The first disadvantage is that using the header files can become complicated, particularly when other header files need to declare an instance of the template. In order to do this, they must #include the stack.h file, which potentially leads to multiple #include's of the file, resulting in multiple definitions of the member functions. This problem can be fixed with the addition of preprocessor macros in the header file to protect against multiple #include operations. For example:

```
#if !defined(stack_h)
#define stack_h
....
....declaration and definition of stack template
....
#endif
```

Using the macros shown above, the contents of the header file will only appear once in the compilation unit, regardless of the number of times the file is included. This resolves the problems of multiple definitions within a compilation unit.

### Template code bloat

The second disadvantage is that the code for each template instance will potentially appear multiple times in the final executable, resulting in the problems of large executable size and multiple symbol definition warnings from the linker.

As an example, consider an executable made up of two compilation units, main.C and functions.C. If both compilation units include the stack.h header file, and declare variables of the type stack<int>, then after the first stage of compilation, both object files, main.o and functions.o, will contain the code for the member functions of the stack<int> class. When the system linker parses the object files to create the final executable, it cannot remove the duplicate symbols since, by default, each compilation unit is treated as an atomic object by the linker. This results in duplicate symbol linker warnings and a final executable that contains redundant code.

The size of the final executable can be reduced by using the compiler option, -qfuncsect, when compiling all of the source code modules. This option causes the compiler to slightly change the format of the output object files. Instead of creating an object file, which contains a single code section (CSECT) and must be treated by the system linker as an atomic unit, the compiler creates an object file where each function is contained in its own CSECT. This means that the object files created are slightly larger than their default counterparts since they contain extra format information in addition to the executable code. This option does not remove the linker warnings, since at link time, there are still multiple symbol definitions. The benefit of this option is that the linker can discard multiple, identical function definitions by discarding the redundant CSECTs, resulting in a smaller final executable. When the -qfuncsect option is not used, the compiler cannot discard the redundant function definitions if there are other symbols in the same CSECT that are required in the final executable.

Refer to 6.1.9, “Virtual functions” on page 226 for information on another potential cause of C++ code bloat.

### Template compile time

The use of the -qfuncsect option reduces the code size of the final executable. It does not resolve the other disadvantage of using this method, that is, longer than required compile times. The reason for this is that each compilation unit contains the member functions for the templates that it instantiates. Using an extreme example with the stack class, consider the situation where an application is built from 50 source files, and each source file instantiates a stack<int> template. This means the member functions for the class are generated and compiled 50 times, yet the result of 49 of those compiles are discarded by the linker since they are not needed. In the example used here, the code for the stack class is trivial, so in absolute terms, the time saved would be minimal. In real life situations, where the template code is complex, the time savings that can be made when compiling a large application are considerable.

Because of the fact that not all of the disadvantages of the simple template method can be overcome, it is only recommended for use when experimenting with templates. An alternative method can be used, which solves all of the problems of the simple method and scales very well for large applications.

## 10.4 Template instantiation file method

The template instantiation file method on AIX basically means letting the compiler decide which template code to instantiate as a final step in the compile and link process. This solves the long compile time disadvantage of the simple template code layout method because the compiler need only compile each template instance once.

This method requires that the declaration and definition of the template are kept in separate files. This is because only the template declaration must be included in every compilation unit that uses the template. If the definition of the template were also in the header file, it would also be included in the source file, and thus compiled, resulting in a situation similar to that in the simple method.

This template model can also benefit from the use of the -qfuncsect compiler option, since it means the linker can discard code sections that are not referenced in the final executable.

The template declaration should be left in the header file, as in the simple template method. The definition of the template member functions needs to be in a file with the same basename as the header file but with a .c (lower case C) file name extension.

**Note:** By default, the file containing the template definition code must have the same name as the template declaration header file, but with a file name extension of .c (lowercase c), even though this extension normally indicates a C language source file. It must also exist in the same directory as the template declaration header file. If the template definition file is not in the same directory, has a different basename, or has a different file name extension (such as .C, .cxx, or .cpp, which are normally used for C++ source files), then the compiler will not detect the presence of the template code to be used with the template declaration header file.

Using the stack template example introduced earlier, the template declaration shown in Example 10-1 on page 379 would be in the stack.h file, while the template definition code shown in Example 10-2 on page 380 would be in the stack.c file in the same directory. If the template definition code file was named stack.cxx or stack_code.c, then the compiler will not associate the file with the template declaration in the stack.h header file.

The name of the template definition file can be changed, if desired, using the implementation pragma directive as follows:

```
#pragma implementation(string-literal)
```

where `string-literal` is the path name for the template definition file enclosed in double quotes. For example, if the stack template definition code were to be stored in the stack_code.cxx file, then the stack.h header file would have the following directive:

```
#pragma implementation(“stack_code.cxx”)
```

Once the structure of the source code has been altered to conform to the required layout, the templates can be used with this method.

### 10.4.1 The -qtempinc option

The -qtempinc option is used when compiling source code that instantiates templates. When no directory is specified with the option, the compiler will create a directory called *tempinc* in the current directory. For example:

```
$ xlC main.C -qtempinc
```

The user may optionally specify the name of a directory to use for storing the information on the templates to be generated. This allows the same tempinc directory to be used when creating an executable that consists of object files that are compiled in different directories. For example:

```
$ xlC -c file1.C file2.C -qtempinc=../app1/templates
$ cd ../app1
$ xlC -o app1 main.C ../src/file1.o ../src/file2.o -qtempinc=./templates
```

The tempinc directory is used to store information about the templates that are required to be generated. When invoked with the -qtempinc option, the compiler collects information about template instantiations and stores the information in the tempinc directory. As the last step of the compilation before linking, the compiler generates the code for the required template instantiations. It then compiles the code and includes it with the other object files and libraries that are passed to the linker to create the final executable.

If the compiler detects a code layout structure that enables the tempinc method to be used, it will automatically enable the -qtempinc option, even if it was not specified on the command line. This causes the template instantiation information to be stored in the tempinc directory. If you want to specify a different directory, you should explicitly use the -qtempinc=dir_name option on the command line. If you want to prevent the compiler from automatically generating the template information, which may be the case when creating a shared object, then use the -qnotempinc option. See 11.2, “Shared objects with templates” on page 402 for more information on the use of the -qnotempinc option when creating shared objects.

One important point to note about the -qtempinc option is that you should use the same value when compiling all compilation units that will be linked together. In other words, do not compile half of the application with -qtempinc, and the other half with -qtempinc=dir_name. Only one tempinc directory can be specified on the final C++ command line that is used to link the application, which means that half of the template instance information will be missing. If more than one tempinc option is specified on the command line, the last one encountered will prevail.

## 10.4.2 Contents of the tempinc directory

The compiler generates a file in the tempinc directory for each template header file that has templates instantiated. The file has the same name as the header file, but with a .C (uppercase C) file name extension. The compiler generates the file when it detects the first instantiation of a template that is declared in the header file with the same name. Information on the subsequent instances of the template is added to the file.

As the final step of the compilation before linking, the compiler compiles all of the files in the tempinc directory and passes the object files to the linker along with the user specified files.

The contents of a template information file are shown in Example 10-3.

*Example 10-3 A sample template information file*[1]

```
1: /*0965095125*/#include "/redbooks/examples/C++/stack.h"
2: /*0000000000*/#include "/redbooks/examples/C++/stack_code.cxx"
3: template stack<int>::stack(int);
4: template stack<int>::~stack();
5: template void stack<int>::push(int);
6: template int stack<int>::pop();
```

The code on line 1 includes the header file that declares the template. The comment at the start of the line is a time stamp and is used by the compiler to

[1] The line number at the beginning of each line is added intentionally for explanation. These numbers and the colon characters “:” are not part of the file.

determine if the header file has changed, which would require the template instance information file to be recompiled.

The code on line 2 includes the template implementation file that corresponds to the header file in line 1. A time stamp consisting of all zeros indicates that the compiler should ignore the time stamp. The file may include other header files that define the classes that are used in template instantiations. For example, if there was a user defined class Box, and the compiler detected an instantiation of stack<Box>, then the header file that defines the class Box would be included in the instance information file.

The subsequent lines in the example shown above cause the individual member functions to be instantiated.

### 10.4.3 Forcing template instantiation

You can, if you wish, structure your program so that it does not use automatic template instantiation. In order to do this, you must know which template classes and functions need to be instantiated.

The #pragma define directive is used to force the instantiation of a template, even if no reference is made to an instance of the generated template. For example:

```
#pragma define(stack<double>);
```

This, however, means that the template implementation file needs to be included in the compilation units that have the #pragma define directives, which results in the same disadvantages of the simple template method described in 10.3, “Simple code layout method” on page 381.

An alternative to this is to manually emulate the process used by the compiler to automatically create the appropriate template instances. Using the stack class as an example, the following compilation unit shown in Example 10-4 could be used to force the creation of the desired stack template classes, even though no objects of those types are referenced in the source code.

*Example 10-4 A sample usage of #pragma define*[2]

```
1: #include "/redbooks/examples/C++/stack.h"
2: #include "/redbooks/examples/C++/stack_code.cxx"
3: #include “/redbooks/examples/C++/Box.h”         // definition of class Box
4: #pragma define(stack<int>);
5: #pragma define(stack<Box>);
6: #pragma define(stack<char>);
```

[2] The line number at the beginning of each line is added intentionally for explanation. These numbers and the colon characters “:” are not part of the file.

```
7: #pragma define(stack<short>);
```

This type of method will be useful when creating shared objects with the **makeC++SharedLib** command. Users of the VisualAge C++ for AIX product should use the -qmkshrobj option instead. See 11.2, “Shared objects with templates” on page 402 for more information.

## 10.5 Template registry: The preferred method

New in VisualAge C++ for AIX Version 6.0 compiler is the template registry method of template handling. The compiler decides which template code to instantiate by referring to and updating a template registry file as compilation occurs. This method is fast, efficient, and best of all, does not require reorganization of your template code.

### 10.5.1 The -qtemplateregistry option

The -qtemplateregistry option maintains, in a registry file, records of all template instantiations as they are encountered in the compilation unit, and ensures that only one instantiation of each template is generated. By default, the compiler writes to a file named templateregistry in the current directory. You may optionally specify a different registry file name with the option. However, you must use the same registry file for the entire program.

Unlike the template instantiation file method of template handling supported with the -qtempinc option, the -qtemplateregistry option does not require the template code to be structured in a certain way. The option is mutually exclusive with the -qtempinc option, and you must not use both options at the same time. Before using the -qtemplateregistry option, remove all instantiation files in the tempinc directory. Also, any existing program that compiles successfully with -qnotempinc, that is, when every reference is instantiated in every compilation unit, will compile successfully with -qtemplateregistry. In other words, there is no migration impact should you want to take advantage of this improved template handling method.

The -qtemplateregistry option works by storing template instantiation information in a registry as compilation occurs. The registry is read for each compilation, and a check is done when an instantiation is required. If the instantiation has already been seen, nothing will happen; otherwise the template will be instantiated in the object file. In either case, a record is added for each template reference to the registry to keep track of the information about use and instantiation for all compilation units.

### 10.5.2 The -qtemplaterecompile option

If a change to a compilation unit removes a template instantiation, recompiling only the changed source file will result in undefined symbols at link time, since other compilation units may still require the template instantiation that is now missing. This dependency information is maintained in the template registry file in the -qtemplateregistry compiles. In this case, the -qtemplaterecompile option will cause a recompilation of all source files that rely on the template.

The -qtemplaterecompile option is, by default, turned on with the -qtemplateregistry option. It manages dependencies among compilation units, and ensures that affected source files are recompiled automatically by consulting the template registry file. It requires that object files originally generated with the -qtemplateregistry option remain in the same directory. Should you need to disable automatic recompilation for any reason, specify the -qnotemplaterecompile option. This may be the case, for example, if your build process moves the generated object files to a different directory.

## 10.6 Standard C++ Library and STL

With the removal of the IBM Open Class (IOC) Library in Version 6, the VisualAge C++ for AIX compiler has standardized on the use of the Standard C++ Library, including the Standard Template Library (STL).

When migrating from a previous version of the VisualAge C++ for AIX compiler, refer to the *IBM Open Class Library Transition Guide*, SC09-4948 to determine whether your application uses IOC, what version of IOC it is using, and the general migration suggestions for application owners whose applications use the IOC. It also contains migration suggestions for most classes included in the IBM Open Class Library.

In general, where an overlap in functions exists between the IBM Open Class Library and the Standard C++ Library, including the Standard Template Library, the following is recommended:

- Use the Standard Template Library (STL) containers, iterators, and algorithms instead of the IOC collection classes.
- Use the Standard C++ exception classes instead of the IOC exception classes.
- Use the Standard C++ string template classes instead of the IOC string classes.

However, there are many classes in the IBM Open Class Library for which there is no equivalent in the Standard C++ Library. The *IBM Open Class Library*

*Transition Guide*, SC09-4948 identifies some of the options available to application owners to deal with this situation. The decision as to which option is best depends on the version of the IBM Open Class Library you use and the extent to which you use classes without an equivalent replacement in the Standard C++ Library.

## 10.6.1 Standard Template Library

The motivation behind the design of the Standard Template Library, or STL, is to support object-oriented programming with type generality. Implemented using templates, STL provides a set of commonly used abstract data types, access methods, and operation methods, allowing components to be easily created, and operations performed with minimal lines of code.

STL is composed of three major components: containers, iterators, and algorithms. The following sections explore each component in more detail.

### Containers

A container is an abstract object data type. It is a template class that manages a sequence or set of elements. Conceptually, it is an object containing other objects, much like an ordinary array. The elements of a container can be of any object type, and the allocation and deallocation of the elements are controlled by the container methods. You can also perform operations such as insert, remove, copy, or compare elements with operators provided by the container.

STL containers come in two different varieties: sequence containers and associative containers. A sequence container is a linear block of objects, where elements are referred to by their position in the container. STL provides the following sequence containers:

**Vector** A contiguous block of objects that allows random access and fast insertions at the end.

**List** Supports only sequential access, but has equally fast insertions and deletions from anywhere within the list.

**Deque** Allows random access and fast insertions and deletions from either end of the queue.

**Stack** Specialized container that supports stack operations only (LIFO).

**Queue** Specialized container that supports queue operations only (FIFO).

Associative containers, on the other hand, do not store elements in any order, and retrieval of elements are fast by using keys. Associative container template classes are parameterized either on a key, or a key and a type. They can be based on unique keys, where there is at most one element in the container for each key. Alternatively, they can be based on equivalent keys, where there is possibly multiple copies of the same key in the container. STL provides the following associative containers:

| | |
|---|---|
| **Set** | Supports unique keys and fast retrieval of keys themselves. |
| **Map** | Supports unique keys and fast retrieval of values (of the parameterized type) based on the keys. |
| **Multiset** | Supports equivalent keys and fast retrieval of keys themselves. |
| **Multimap** | Supports equivalent keys and fast retrieval of values (of the parameterized type) based on the keys. |

To use a container, simply declare a variable with the desired container class. For example, to create an empty vector of integers:

```
vector<int> v;
```

To add an integer n to the vector:

```
v.push_back(n);
```

To get the size of the vector:

```
int size = v.size();
```

See the *VisualAge C++ Standard C++ Library Reference*, SC09-4949 for more information on the methods provided by the various container classes.

### Iterators

Iterators are abstract pointers to containers, that allow you to work with the various containers in much the same way as you would with normal C++ pointers. You can refer to a location in a container using an iterator, increment it to get to the next position, dereference it to get the value of the element, and so on. Two iterators can be compared to find the relative locations with respect to each other, or subtracted to find the distance between the two locations. For example:

```
vector<int>::iterator start = v.begin();
vector<int>::iterator end = v.end();
int size = end - start;
```

the above lines of code are equivalent to:

```
int size = v.size();
```

And to print the value of the first element of the vector:

```
cout << *start << endl;
```

The standard defines five categories of iterators, according to the type of operations they perform:

**Input** Provides data read access to a container in a sequential manner.

**Output** Provides data write access to a container in a sequential manner.

**Forward** Allows forward traversal (that is, increment only) of the container in a sequential manner. This can replace an input or output iterator.

**Bidirectional** Allows traversal of the container in both directions (that is, increment and decrement) in a sequential manner. This can replace a forward iterator.

**Random access** Allows random access of a container. This can replace a bidirectional iterator.

Refer to the *VisualAge C++ Standard C++ Library Reference*, SC09-4949 for more details.

## Algorithms

Algorithms are template functions that are specialized to perform operations on containers. They are parameterized by iterators instead of containers, in order to support other user-defined types.

There are three main types of algorithmic operations. Non-modifying sequence operations, as the name suggests, do not alter the sequence of the container in any manner. It includes template functions such as find, count, equal, mismatch, and search.

Mutating sequence operations, including functions such as copy, swap, transform, replace, fill, generate, remove, unique, reverse, rotate, random_shuffle, and partition, change the sequence of the container while the operation is carried out.

Sorting and related operations may or may not change the sequence, and include such functions as sort, nth_element, lower_bound, upper_bound, equal_range, binary_search, merge, and so on. There are also set operations functions like set_union, set_intersection, and set_difference that operate on sorted containers.

For more information on all available algorithmic functions, refer to the *VisualAge C++ Standard C++ Library Reference*, SC09-4949 for details.

### 10.6.2 A STL example

Putting it all together, the following is a simple example that takes an integer number as input, and randomly generates 10 numbers between 0 to 100, using the input as the seed value for the random generator function. The numbers are stored in a vector container. They are then sorted and output to the terminal:

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{
    unsigned int seed;
    if (argc >= 2 && (seed = atoi(argv[1])))
        srand(seed);

    vector<int> v;
    for (int i = 0; i < 10; i++)
        v.push_back(1+(int)(100.0*rand()/(RAND_MAX+1.0)));

    sort(v.begin(), v.end());
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << endl;
}
```

# 11

# Creating shared objects from C++ source codes

Creating shared objects from C++ source codes on AIX require some additional knowledge in addition to the tasks required for creating shared objects from C source codes.

This chapter explains how to create and manage shared objects from C++ source codes for C++ application programmers by providing the following sections:

- Section 11.1, “Creating shared objects from C++ source codes” on page 394
- Section 11.2, “Shared objects with templates” on page 402

## 11.1 Creating shared objects from C++ source codes

The C++ language, although similar in some respects to the C language, offers many additional facilities. One of these is known as *function overloading*, which makes it possible to have multiple functions with the same name but different parameter lists. This feature means it is not possible to use the function name alone as a unique identifier in the symbol table of an object file. For this reason, function names in C++ are *mangled* to produce the symbol name. The mangling uses a code to indicate the number, type, and ordering of parameters to the function.

The term *mangle* has a specific meaning in compiler products, especially in the C++ compilers, because the language standard allows you to have multiple instances of the same name function, but each function will be used with specified parameter variable types. For example, the following two function definitions can be mangled (this is a pseudo-mangling example):

```
int add(int, int)
double add(double, double)
```

to the following function names during the compilation process:

```
add_Fii(int, int)
add_Fdd(double, double)
```

Please note that the mangled names are referred to as symbols only after the compilation; therefore, you do not have to understand what mangled names will be produced after the compilation process (see 6.3.4, "The c++filt utility" on page 243 for further information about the mangling process).

It is the name mangling feature of C++ that means the process of creating a shared object, which includes object code created by the C++ compiler, is slightly more complicated than when using code produced by the C compiler.

Although it would be possible to create import and export files manually, the process is time consuming, since a unique symbol name is required for each instance of an overloaded function.

## 11.1.1 Creating a C++ shared object

VisualAge C++ for AIX provides the **makeC++SharedLib** command, which performs most of the dirty work behind the scenes. It is included in the vacpp.cmd.tools fileset and installed automatically when the product is installed, as shown in the following example:

```
# lslpp -w /usr/vacpp/bin/makeC++SharedLib
  File                                        Fileset               Type
  ----------------------------------------------------------------------------
  /usr/vacpp/bin/makeC++SharedLib             vacpp.cmp.tools       File
```

The process of creating a shared object from C++ source codes is very similar to the process of creating a shared object from C source codes. See 2.8, "Creating shared objects" on page 92 how to create shared objects from C program source codes.

To create a shared object from C++ source codes, do the following:

1. Compile the C++ source code from which you wish to create to a shared object. For example, the following example will produce two object modules, cppsrc1.o and cppsrc2.o, in the current directory:

   ```
   $ xlC -c cppsrc1.C cppsrc2.C
   ```

2. Run the **makeC++SharedLib** command[1] to create the shared object with the object file names. The following example shows how to create a shared object, shr_cpp_12.o, from the object modules created in the previous step:

   ```
   $ /usr/vacpp/bin/makeC++SharedLib -o shr_cpp_12.o cppsrc1.o cppsrc2.o
   ```

Use the -G option when you want to create a shared object enabled for use with run-time linking, or one that uses the libname.so format. For example:

```
$ /usr/vac/bin/makeC++SharedLib -G -o shr_cpp_12.so cppsrc1.o cppsrc2.o
```

If the **makeC++SharedLib** command is used to build the C++ shared libraries and export symbols, make sure that any system libraries required by the shared object are always specified with the -l option (for example, -lX11) and not by name (for example, /usr/lib/libX11.a) on the command line. This allows the system libraries to be simply referenced by the shared object being created and not go through the special C++ related processing.

## 11.1.2 Generating an export file

Another very useful option of the **makeC++SharedLib** command is the ability to save the export file that is generated behind the scenes and normally discarded

[1] If you add the directory path, /usr/vacpp/bin, into the PATH environment value, you do not have to invoke the **makeC++SharedLib** command with the full pathname.

after use. If saved, this export file can then be used as an import file when creating another shared object. The -e expfile option is used to save the export file. Note that the export file produced does not have an object file name field (#!) on the first line, so one will have to be manually added, if required.

### 11.1.3 The -qmkshrobj option

Starting from Version 5, VisualAge C++ for AIX provides a useful compiler option, -qmkshrobj, which is used to instruct the compiler to create a shared object from previously created object files and archive libraries. It provides similar functionality to the **makeC++SharedLib** command and, in addition, makes it much easier to create shared objects that use template functions. See 11.2, “Shared objects with templates” on page 402 for more detailed information.

For example, to create a shared object shr1.o from the source1.o and source2.o files, use the following command:

```
$ xlC -qmkshrobj -o shr1.o source1.o source2.o
```

The -G option can be also used in conjunction with the -qmkshrobj option to create an object that uses the run-time linking shared object, as shown in the following example:

```
$ xlC -G -qmkshrobj -o libshr1.so source1.o source2.o
```

To specify the priority of the shared object, which determines the initialization order of the shared objects used in an application, append the priority number to the -qmkshrobj option. For example, to create the shared object shr1.o, which has an initialization priority of -100, use the following command:

```
xlC -qmkshrobj=-100 -o shr1.o source1.o source2.o
```

If none of the -bexpall, -bE:, -bexport:, or -bnoexpall options are specified, then using the -qmkshrobj option will force the compiler to generate an exports file that exports all symbols. This file can be saved for use as an import file when creating other shared objects, if desired. This is done using the -qexpfile=*filename* option. For example:

```
xlC -qmkshrobj -qexpfile=shr1.exp -o shr1.o source1.o source2.o
```

### 11.1.4 Mixing C and C++ object files

In addition to the mangling of symbol names, the C++ language differs from the C language in the way function arguments are passed on the calling stack. The C language pushes arguments onto the stack right to left, which means the left-most argument is top-most on the stack. For various reasons, the C++ language uses right to left instead. This is termed *linkage*, and a complication unit can have both the C and C++ linkages.

When mixing C and C++ code together, it is necessary to use a linkage block to call a C routine from a C++ routine. This is to prevent the compiler from mangling the name of the C routine, which would result in a symbol name that could not be resolved. For example, to call the C function, foo(), from C++ code, the declaration of foo must be in an external linkage block, as shown in the following code fragment:

```
extern “C” {
    void foo(void);
}
class1::class1(int a)
{
    foo();
}
```

If the declaration of foo() was not contained in the extern “C” block, the C++ compiler would mangle the symbol name to foo__Fv.

When mixing C and C++ objects within a single shared object, either the **makeC++SharedLib** command (which uses the C++ compiler) or the -qmkshrobj option of the C++ compiler should be used to create the shared object. Do not use the C compiler or the linker, since they may not produce the correct result, as they are not aware of C++ constructors, destructors, templates, and other C++ language features.

## 11.1.5 Order of initialization

There are situations where the order of initialization of data objects within a program is important to the correct operation of the application. A priority can be assigned to an individual object file when it is compiled. This is done using the -qpriority option. For example:

```
$ xlC -c zoo.C -qpriority=-50
```

The C++ compiler and the **makeC++SharedLib** command also support options that can be used to indicate the relative order of initialization of shared objects. There is a slight difference in the way the priority is specified when using each command. When using the C++ compiler, the priority is specified as an additional value with the -qmkshrobj option. For example:

```
$ xlC -qmkshrobj=-100 -o shr1.o source1.o
```

When using the **makeC++SharedLib** command, the priority is specified with the -p option. For example:

```
$ makeC++SharedLib -p -100 -o shr1.o source1.o
```

Priority values can also be indicated within C++ code by inserting the priority compiler directive as follows:

```
#pragma priority(value)
```

These *values* alter the order of initialization of data objects within the object module.

### Priority values

Priority values may be any number from -214782623 to 214783647. A priority value of -214782623 is the highest priority. Data objects with this priority are initialized first. A priority value of 214783647 is the lowest priority. Data objects with this priority are initialized last. Priority values from -214783648 to -214782624 are reserved for system use. If no priority is specified, the default priority of 0 is used.

The explanation of priority values uses the example data objects and files shown in Figure 11-1 on page 399.

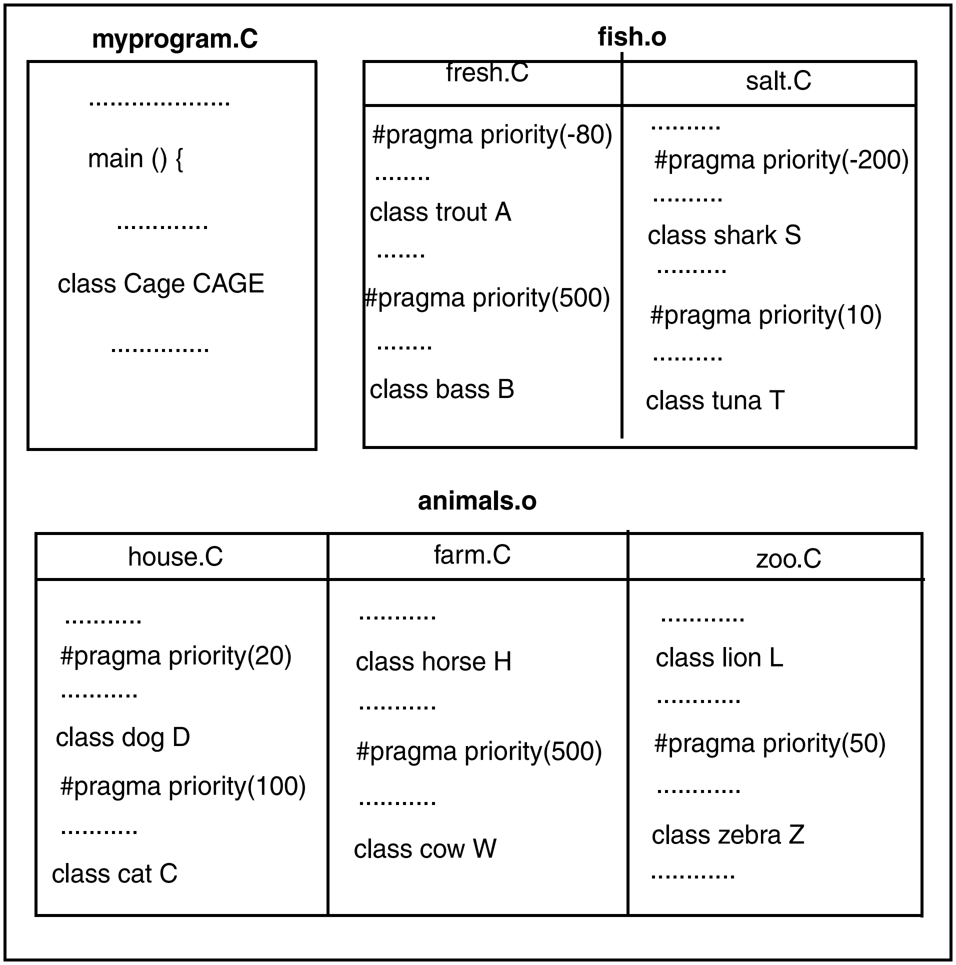


*Figure 11-1 Illustration of objects in fish.o and animals.o*

This example shows how to specify priorities when creating shared objects to guarantee the order of initialization. The user should first of all determine the order in which they want the objects to be initialized, both within each file and between shared objects:

1. Develop an initialization order for the objects in house.C, farm.C, and zoo.C:

   a. To ensure that the object lion L in zoo.C is initialized before any other objects in either of the other two files in the shared object animals.o, compile zoo.C using a -qpriority=*nn* option with nn less than zero so that data objects have a priority number less than any other objects in farm.C and house.C:

   ```
   $ xlC -c -qpriority=-50 zoo.C
   ```

   b. Compile the house.C and farm.C files without specifying the -qpriority=*nn* option. This means the priority will default to zero. This means data objects within the files retain the priority numbers specified by their #pragma priority(nn) directives:

   ```
   $ xlC -c house.C farm.C
   ```

   c. Combine these three files into a shared library. Use the **makeC++SharedLib** command to construct the shared object animals.o with a priority of 40:

   ```
   $ makeC++SharedLib -o animals.o -p 40 house.o farm.o zoo.o
   ```

2. Develop an initialization order for the objects in fresh.C and salt.C, and use the #pragma priority(value) directive to implement it:

   a. Compile the fresh.C and salt.C files:

   ```
   $ xlC -c fresh.C salt.C
   ```

   b. To assure that all the objects in fresh.C and salt.C are initialized before any other objects, including those in other shared objects and the main application, use **makeC++SharedLib** to construct a shared object fish.o with a priority of -100:

   ```
   $ makeC++SharedLib -o fish.o -p -100 fresh.o salt.o
   ```

   Because the shared object fish.o has a lower priority number (-100) than animals.o (40), when the files are placed in an archive file with the **ar** command, the objects are initialized first.

3. To create a library that contains the two shared objects, so that the objects are initialized in the order you have specified, use the **ar** command. To produce an archive file, libprio.a, enter the command:

   ```
   $ ar rv libprio.a animals.o fish.o
   ```

   Where libprio.a is the name of the archive file that will contain the shared library files, and animals.o and fish.o are the two shared files created with **makeC++SharedLib**.

4. Compile the main program, myprogram.C, that contains the function main to produce an object file, myprogram.o. By not specifying a priority, this file is compiled with a default priority of zero, and the objects in main have a priority of zero:

   ```
   $ xlC -c myprogram.C
   ```

5. Produce an executable file, animal_time, so that the objects are initialized in the required order, and enter:

   ```
   $ xlC -o animal_time main.o -lprio -L.
   ```

   When the animal_time executable is run, the order of initialization of objects is as shown in Table 11-1.

*Table 11-1 Order of initialization of objects in prriolib.a*

| Object | Priority value | Comment |
|---|---|---|
| fish.o | -100 | All objects in fish.o are initialized first because they are in a library prepared with **makeC++SharedLib -p -100** (lowest priority number; -p -100 specified for any files in this compilation). |
| shark S | -100(-200) | Initialized first in fish.o because within file #pragma priority(-200). |
| trout A | -100(-80) | #pragma priority(-80). |
| tuna T | -100(10) | #pragma priority(10). |
| bass B | -100(500) | #pragma priority(500). |
| myprog.o | 0 | File generated with no priority specifications; default is 0. |
| CAGE | 0(0) | Object generated in main with no priority specifications; default is 0. |
| animals.o | 40 | File generated with **makeC++SharedLib** with -p 40. |
| lion L | 40(-50) | Initialized first in file animals.o compiled with -qpriority=-50. |
| horse H | 40(0) | Follows with priority of 0 (since -qpriority=nn not specified at compilation and no #pragma priority(nn) directive). |
| dog D | 40(20) | Next priority number (specified by #pragma priority(20)). |
| zebra N | 40(50) | Next priority number from #pragma priority(50). |
| cat C | 40(100) | Next priority number from #pragma priority(100). |
| cow W | 40(500) | Next priority number from #pragma priority(500). |

## 11.2 Shared objects with templates

Templates are usually declared in a header file. Each time a template is used, code is generated to instantiate the template with the desired parameters. Most C++ compilers work with a *template repository*. No template code is generated at compile time; the compiler just remembers where the template code came from. Then, at link time, as the compiler/linker puts all parts together, it notices which templates actually need to be generated. The code is then produced, compiled, and linked into the application.

This becomes a problem when using templates with shared libraries, where no actual linking takes place. Therefore, you must make sure that the template code is generated when producing the shared library.

Therefore, one should keep track of compilation and inclusion of template instantiations. This would mean that one has to manually keep track of all the template instantiation and address them during the linking phase.

Starting from Version 5, the VisualAge C++ for AIX product supports a compiler option, -qmkshrobj, to create a shared objects. This option, together with the -qtempinc option, should be used in preference to the **makeC++SharedLib** command when creating a shared object that uses templates. The advantage of using these options instead of **makeC++SharedLib** is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

### 11.2.1 Templates and makeC++SharedLib

The **makeC++SharedLib** command is supplied with all IBM C++ command line compiler drivers for the AIX platform.[2] The command is implemented as a shell script that gathers the supplied input and then calls the linker to create the shared object.

When creating a shared object that uses templates, the **makeC++SharedLib** command needs to somehow find information on the templates that are to be instantiated. Because the script calls the linker, and not the compiler, it does not look at the contents of the tempinc directory. This means the method of creating a shared object that uses templates relies on either using the simple template method code layout, as described in 10.3, “Simple code layout method” on page 381, or forcing templates to be instantiated, as described in 10.4.3, “Forcing template instantiation” on page 386.

The best method to use will depend on the circumstances. Using the simple code layout method means that all the required templates are generated automatically.

[2] Type **ls -l /usr/vacpp/bin/makeC++SharedLib*** on the command line.

However, it also comes with the disadvantages of slower compile times and larger code size. Forcing the templates to be instantiated is better from both the code size and compile time aspect, but it does mean that the user needs to maintain files that instantiate the required templates.

Suppose you want to create a shared object from the following two source files, which use the preferred code layout method.

File source1.C contains the following code:

```
#include “stack.h”
stack<int> counter1;

void function1(int a)
{
    counter1.push(a);
}
```

The file source2.C contains the following code:

```
include “stack.h”
stack<int> counter2;

void function2(int a)
{
    counter2.push(a);
}
```

Using the **makeC++SharedLib** command, an attempt is made to create a shared object as follows:

```
$ xlC -c source1.C source2.C
$ /usr/vacpp/bin/makeC++SharedLib -o shr1.o -p0 source1.o source2.o
ld: 0711-317 ERROR: Undefined symbol: .stack<int>::stack(int)
ld: 0711-317 ERROR: Undefined symbol: .stack<int>::~stack()
ld: 0711-317 ERROR: Undefined symbol: .stack<int>::push(int)
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

The command failed, and based on the output, it is easy to see that the required template functions have not been instantiated. At this point, note that because the code uses the preferred code layout method, the compiler has, in fact, automatically created the file tempinc/stack.C, which, if compiled, would supply the required template definitions. You can, if you wish, copy this file and make it an explicit compilation unit as part of your source code. In this case, that would mean adding the following command to the sequence:

```
$ xlC -c tempinc/stack.C -o stack.o
```

The object, file stack.o, would then be passed to the **makeC++SharedLib** command along with source1.o and source2.o.

### 11.2.2 Templates and -qmkshrobj

You should use the -qmkshrobj option instead of the **makeC++SharedLib** command when creating a shared object. Because the option is a compiler option, the compiler will automatically look in the tempinc directory (or the directory specified with the -qtempinc=dirname option) for the automatically generated template instance information. Using the same source files as described in the previous section, the following commands can be used to create the shared object:

```
$ xlC -c source1.C source2.C
$ xlC -qmkshrobj -o shr1.o source1.o source2.o
```

This time, the command works correctly, since the compiler looks in the tempinc directory. Remember to use the same -qtempinc option (if any) that was used when compiling the modules being used to create the shared object.

This option solves the problems associated with creating shared objects that use template classes and functions. If you want to create a shared object that contains pre-instantiated template classes and functions for use by other developers, then you can create an additional compilation unit that explicitly defines the required templates using the #pragma define directive.

# 12

# Packaging your applications

If your applications are only executed on the systems where you are developing and debugging, no software installation is required. However, applications are most likely developed on development and test systems, then installed on production systems.

The software installation tasks are commonly performed by customers or other people who might not have enough knowledge about your applications. Even if you have prepared the detailed software installation instruction for your applications, there is no guarantee that those people follow your instruction; you have no control where they try to install your applications.

By creating your application packages, the required conditions, such as supported operating system software levels and adequate physical memory size for your applications, will be enforced.

This chapter provides the following sections to explain how to package your applications for AIX:

- Section 12.1, “Understanding the AIX standard packaging” on page 406
- Section 12.2, “Packaging applications using mkinstallp” on page 418

# 12.1 Understanding the AIX standard packaging

This section provides you with a basic understanding of the AIX standard packaging. For further information about AIX standard packaging, please refer to the *AIX 5L Version 5.2 Installation Guide and Reference*.

## 12.1.1 Filesets and package files

In AIX, the smallest installable unit is a fileset. A fileset logically groups files and directories to be installed. A fileset also includes required control files and optional installation customization files.

Several filesets can be packaged in a package file. A package file is an AIX backup-format file; therefore, it is sometimes referred to as a *bff-file*.

Figure 12-1 illustrates the relationship among filesets, packages, and bundles.

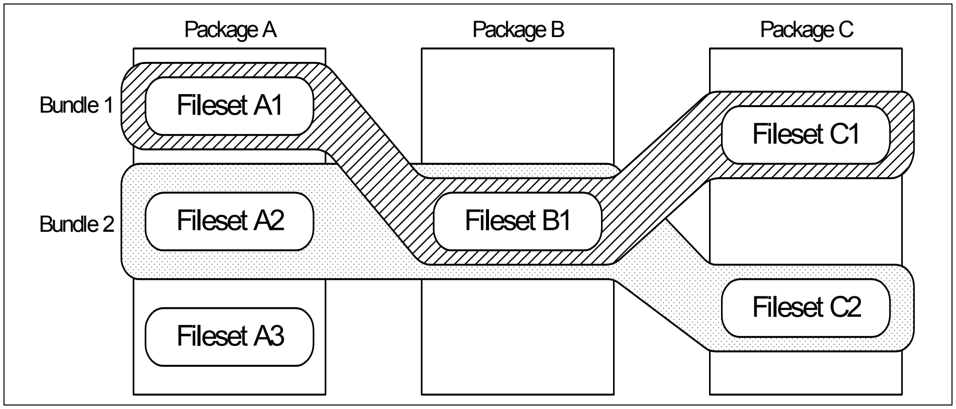


*Figure 12-1 Relationship among filesets, packages, and bundles*

## 12.1.2 Bundles

In AIX, you have hundreds of filesets in the base operating system (BOS). Therefore, to easily select many filesets, AIX offers simple facilities, called bundles. In Figure 12-1, if you select bundle 1, then you specify installing the filesets A1, B1, and C1. Please note that multiple bundles can include the same filesets. For example, in Figure 12-1, both bundle 1 and 2 include the fileset B1.

A bundle is defined by a file installed in either the /usr/sys/inst.data/sys_bundles or /usr/sys/inst.data/user_bundles directories. In the /usr/sys/inst.data/sys_bundles directory, you can see the system defined[1] bundles are already installed by default (see Example 12-1 on page 407).

*Example 12-1 System defined bundles on AIX 5L Version 5.2*

```
# cd /usr/sys/inst.data/sys_bundles; ls -l *.bnd
-rw-r--r--   1 bin      bin            896 Sep 13 09:53 AllDevicesKernels.bnd
-rw-r--r--   1 bin      bin            879 Sep 13 09:53 Alt_Disk_Install.bnd
-rw-r--r--   1 bin      bin           1391 Sep 13 09:53 App-Dev.bnd
-rw-r--r--   1 bin      bin           1051 Sep 13 09:53
CC_EVAL.DocServices.bnd
-rw-r--r--   1 bin      bin           1155 Sep 13 09:53 CC_EVAL.Graphics.bnd
-rw-r--r--   1 bin      bin           1176 Sep 13 11:03 CDE.bnd
-rw-r--r--   1 bin      bin           1026 Sep 13 09:53 DocServices.bnd
-rw-r--r--   1 bin      bin           1292 Sep 13 09:53 GNOME.bnd
-rw-r--r--   1 bin      bin           1517 Sep 13 09:53 Graphics.bnd
-rw-r--r--   1 bin      bin            879 Sep 13 09:53 HTTP_Server.bnd
-rw-r--r--   1 bin      bin            972 Sep 13 09:53 KDE.bnd
-rw-r--r--   1 bin      bin            829 Sep 13 09:53 Kerberos_5.bnd
-rw-r--r--   1 bin      bin           1201 Sep 13 09:53 Media-Defined.bnd
-rw-r--r--   1 bin      bin           1363 Sep 13 09:53 Netscape.bnd
-rw-r--r--   1 bin      bin           1214 Sep 13 11:03 Server.bnd
-rw-rw-r--   1 root     system         409 Jun 06 2002  devices.bnd
-rw-r--r--   1 bin      bin           2030 Sep 13 09:53 openssh_client.bnd
-rw-r--r--   1 bin      bin           2030 Sep 13 09:53 openssh_server.bnd
-rw-r--r--   1 bin      bin           1373 Sep 13 11:03 wsm_remote.bnd
```

In order to install filesets using a bundle, do the following:

1. Select the following SMIT menus (you can access it using the SMIT fast path **smit easy_install**):

   ```
   # smit
       Software Installation and Maintenance
           Install and Update Software
               Install Software Bundle
   ```

2. Specify the installation device (typically /dev/cd0):

   ```
   INPUT device / directory for software        [ ]
   ```

3. Select a bundle name on the panel, then press the Enter key; all the filesets defined in the selected bundle file, shown in Example 12-1, will be installed.

Please note that you can only use bundles for installation purposes. AIX does not offer simple methods to uninstall bundles or to confirm whether bundles are correctly installed.

[1] You can also create user defined bundles under the /usr/sys/inst.data/user_bundles directory.

### 12.1.3 Managing filesets

The installed filesets can be classified under several states, as shown in Table 12-1. The successfully installed filesets should be in either an applied or committed state.

*Table 12-1 Fileset state*[2]

| Status | Description |
|---|---|
| APPLIED | The specified fileset update is installed on the system. The APPLIED state means that the fileset update can be rejected with the **installp** command and the previous level of the fileset restored. This state is only valid for fileset updates. |
| APPLYING* | An attempt was made to apply the specified fileset, but it did not complete successfully, and cleanup was not performed. |
| BROKEN | The specified fileset or fileset update is broken and should be reinstalled before being used. |
| COMMITTED | The specified fileset is installed on the system. The COMMITTED state means that a commitment has been made to this level of the software. A committed fileset update cannot be rejected, but a committed fileset base level and its updates (regardless of state) can be removed or deinstalled by the **installp** command. |
| OBSOLETE | The specified fileset was installed with an earlier version of the operating system but has been replaced by a repackaged (renamed) newer version. Some of the files that belonged to this fileset have been replaced by versions from the repackaged fileset. |
| COMMITTING* | An attempt was made to commit the specified fileset, but it did not complete successfully, and cleanup was not performed. |
| REJECTING* | An attempt was made to reject the specified fileset, but it did not complete successfully, and cleanup was not performed. |

In order to confirm the fileset status, you can use the **lslpp -L** command, as shown in Example 12-2 on page 409. In this case, the fileset bos.rte.install is in committed status (C).

[2] Filesets with the state specified with an asterisk (*) are not shown on the **lslpp -L** command output, since they are considered to be in a transient state.

*Example 12-2 Listing a fileset status*

```
# lslpp -L bos.rte.install
  Fileset                      Level  State  Type  Description (Uninstaller)
  ----------------------------------------------------------------------------
  bos.rte.install            5.2.0.0    C     F    LPP Install Commands


State codes:
 A -- Applied.
 B -- Broken.
 C -- Committed.
 O -- Obsolete.  (partially migrated to newer version)
 ? -- Inconsistent State...Run lppchk -v.
```

If a fileset status is broken, then you should deinstall and reinstall the fileset.[3] If a fileset status is obsolete, then the fileset may or may not be supported on your system. If a fileset status is inconsistent, you should check it using the **`lppchk -v`** command.

Figure 12-2 on page 410 illustrates the state diagram of the applied and committed states. Once a base level fileset (fileset level 1.0.0.0) is installed, it is always in the committed status. If you apply an update fileset (fileset level 1.0.1.0),[4] the status is changed to the applied status.

If you commit the applied update fileset, then the updated fileset level is persistent. In order to revert to the previous fileset level, you have to uninstall the fileset and reinstall it. If you reject the applied update fileset, then the fileset level is reverted to the last committed level.

[3] If you install filesets over the network, you should verify the size and the checksum of the installing package file on the target system.
[4] An update fileset is sometimes referred to as a PTF (Program Temporary Fix).

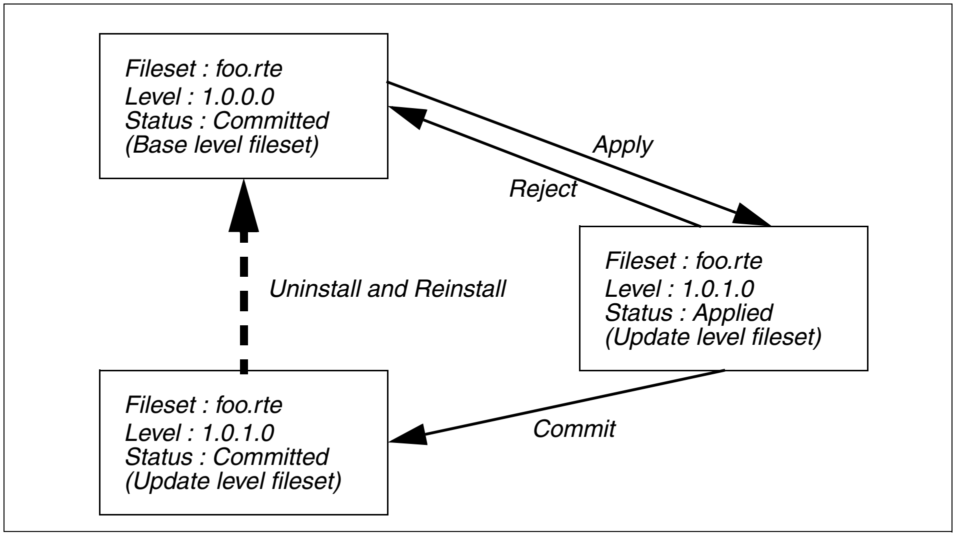


*Figure 12-2 State diagram between applied and committed state*

This mechanism is used to precisely control the software levels on the running system. When you encounter a software problem, you should investigate to solve the problem. If the problem is caused by some defects, software vendors supporting the software products might provide some software fixes.

## APARs and fileset updates

In the IBM terminology, a software defect is uniquely identified by an identifier called an APAR (authorized program analysis reports). An APAR can be, but does not have to be, addressed by more than one fileset updates. If an APAR is addressed by multiple fileset updates, then the software defect affects many files included in multiple filesets. In Figure 12-3 on page 411,[5] the APAR IZ98765 includes the fileset update of foo.rte with an update level 1.0.1.0. Therefore, in order to fix the software defect identified by the APAR IZ98765, you have to apply this single fileset update only. In order to fix the software defect identified by the APAR IZ56789, you have to apply both fileset updates, for foo.rte and gnat.rte, as shown in Figure 12-3 on page 411.

[5] We use these two identifies, IZ98765 and IZ56789, for illustrative purposes only. They do not exist.

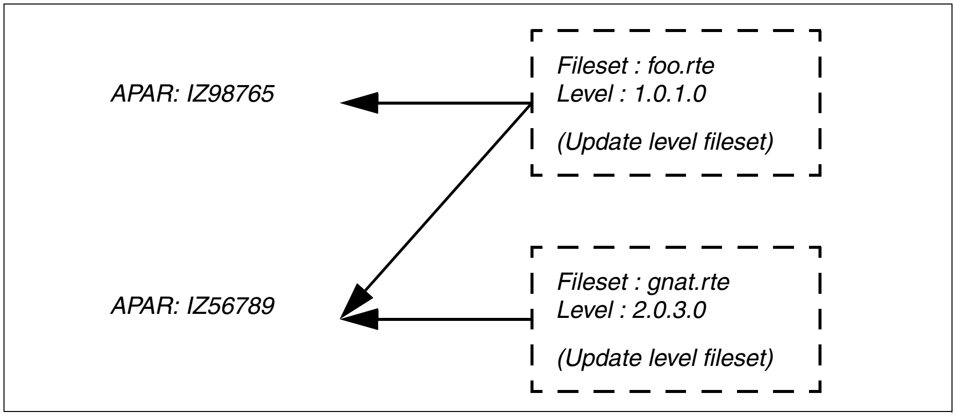


*Figure 12-3 Relationship between APARs and update fileset*

You can confirm whether the specific APARs are applied or not using the **instfix** command. In the following example, the APAR IY35444 is applied on the system:

```
# instfix -ivk IY35444
IY35444 Abstract: Add snapshot backup support to JFS2

    Fileset bos.mp:5.2.0.1 is applied on the system.
    Fileset bos.mp64:5.2.0.1 is applied on the system.
    Fileset bos.up:5.2.0.1 is applied on the system.
    All filesets for IY35444 were found.
```

If an APAR is not applied[6] on the system, the **instfix -ik APAR_ID** command shows the error message similar to either of the following examples:

```
All filesets for IZ98765 were not found.
```

or

```
There was no data for IZ56789 in the fix database.
```

Therefore, you do not have to remember which fileset update would fix the specific software defect, as long as you know the corresponding APAR.

To find APARs for AIX, visit the following URL:

http://techsupport.services.ibm.com/server/support?view=pSeries

[6] An APAR classified as a packaging APAR also shows this message. A packaging APAR is provided to specify multiple APARs using one identifier, mainly for ordering and distribution purposes.

### Recommended maintenance level

Sometimes IBM ships a recommended maintenance level (also referred to as RML), which includes a series of fileset updates. By using recommended maintenance levels, you can easily track the latest level of all the filesets included in AIX. The latest AIX installation media set usually includes the latest recommended maintenance level in the Update CD.

To simply determine the latest recommended maintenance level applied on the system, you can use the **oslevel -r** command as follows:

```
# oslevel -r
5200-01
```

The command output `5200-01` shows that RML 5200-01 is applied on that system.

In order to determine what recommended maintenance levels are applied, you can use the **instfix** command as follows:

```
# instfix -i | grep ML
    All filesets for 5.0.0.0_AIX_ML were found.
    All filesets for 5200-01_AIX_ML were found.
```

If some lines show the message `All filesets for XXXX-YY_AIX_ML were not found`, then you can confirm which of the fileset updates included in the recommended maintenance level are not applied on the system by running the following command:

```
# instfix -ivk XXXX-YY_AIX_ML | grep not | grep :
```

To download the recommended maintenance level, visit the following URL:

http://techsupport.services.ibm.com/server/support?view=pSeries

**Note:** We strongly recommend that you purchase a software program support contract for each AIX system, even if you understand the AIX software packaging mechanism and can easily download the required fileset updates. To purchase software program support, please contact your IBM sales representative or the IBM Business Partner from which you purchased your systems.

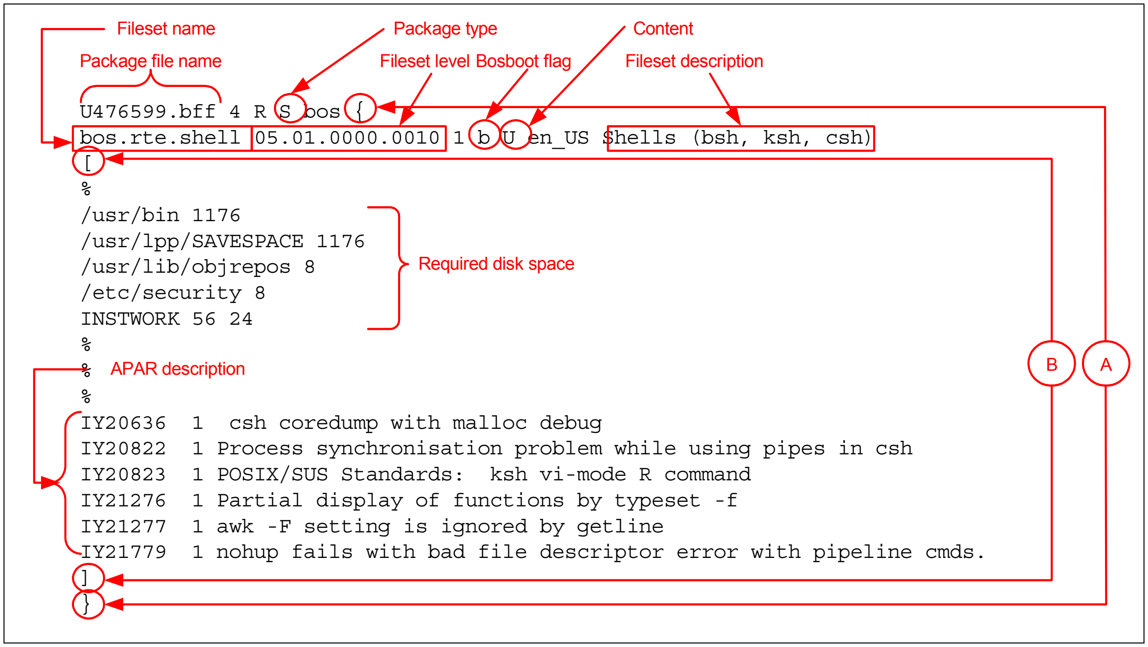
## 12.1.4 Viewing the TOC file (.toc)

Each AIX standard package file contains a table of contents (TOC). Before installation, this information has to be retrieved from package files and placed in the directory as a TOC file named .toc.

If you download some APARs or copy some packages from the install media to a file system, you have to issue the **inutoc** command to create the .toc file. If you copy additional packages into the /usr/sys/inst.images directory, you have to manually rebuild the .toc file using the **inutoc** command. The following example shows you how to create or update the .toc file in the /usr/sys/inst.images directory:

```
# inutoc /usr/sys/inst.images
# cd /usr/sys/inst.images; ls -l
total 800
-rw-r--r--    1 root     system          552 Apr 10 15:55 .toc
-rw-r--r--    1 root     system       403456 Apr 10 15:55 U476599.bff
```

The created .toc file is a text file, so you can view the contents using a viewer command, such as **pg** or **more**. Figure 12-4 shows an example entry of the .toc file. A package has a block shown as A in Figure 12-4. A fileset has a block shown as B in Figure 12-4. If a package contains multiple filesets, then you will see multiple blocks of B in the package block.



```
U476599.bff 4 R S bos {
bos.rte.shell 05.01.0000.0010 1 b U en_US Shells (bsh, ksh, csh)
[
%
/usr/bin 1176
/usr/lpp/SAVESPACE 1176
/usr/lib/objrepos 8
/etc/security 8
INSTWORK 56 24
%
%
%
IY20636  1  csh coredump with malloc debug
IY20822  1 Process synchronisation problem while using pipes in csh
IY20823  1 POSIX/SUS Standards:  ksh vi-mode R command
IY21276  1 Partial display of functions by typeset -f
IY21277  1 awk -F setting is ignored by getline
IY21779  1 nohup fails with bad file descriptor error with pipeline cmds.
]
}
```

*Figure 12-4   Sample .toc file*

Table 12-2 on page 414 explains entries shown in Figure 12-4.

*Table 12-2 Fields description of the .toc file*

| Field name | Description |
|---|---|
| Package file name | The file name of the package. |
| Package type | Indicates package type:<br>► I (Installation): All the filesets contained in this package are base filesets.<br>► S (Single update): All the filesets contained in this package are fileset updates. |
| Fileset name | The name of the fileset. |
| Fileset level | The fileset level represented by (V.R.M.F):<br>► V: Version.<br>► R: Release.<br>► M: Maintenance level.<br>► F: Fix level. |
| Bosboot flag | Indicates whether a bosboot is needed after installation:<br>► N: Do not invoke bosboot.<br>► b: Invoke bosboot. |
| Content | Indicates the parts included in the fileset or the fileset update:<br>► B: usr and root part.<br>► H: share part.<br>► U: usr part only. |
| Fileset description | The description of the fileset. |
| Required disk space | Size required for each install target directory. |
| APAR descriptions | Information regarding the APARs contained in the fileset update. |

For further information about the format of the .toc file, please refer to *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

Once a .toc file is created, you can list the table of contents using the **installp** command, as shown in Example 12-3 on page 415.

*Example 12-3 Listing the table of contents*

```
# installp -ld /usr/sys/inst.images
Fileset Name                Level                     I/U Q Content
  ====================================================================
  bos.mp64                    5.2.0.1                   S  b usr
#   Base Operating System 64-bit Multiprocessor Runtime

  bos.mp64                    5.2.0.2                   S  b usr
#   Base Operating System 64-bit Multiprocessor Runtime
```

## 12.1.5 Viewing package files

The AIX standard packaging uses the **backup** command to archive package files. Therefore, you can un-archive it using the **restore** command. Example 12-4 shows how to view the contents of a package file.

*Example 12-4 Viewing the contents of a package file*

```
# restore -qTf bos.mp64.5.2.0.2.U
New volume on bos.mp64.5.2.0.2.U:
Cluster size is 51200 bytes (100 blocks).
The volume number is 1.
The backup date is: Mon Nov 18 11:24:17 CST 2002
Files are backed up by name.
The user is BUILD.
./
./lpp_name
./usr
./usr/lpp
./usr/lpp/bos.mp64/bos.mp64/5.2.0.2
./usr/lpp/bos.mp64/bos.mp64/5.2.0.2/liblpp.a
./usr/lib/boot/unix_64
The number of archived files is 7.
```

### TOC information file (lpp_name)

A package always contains a file, named lpp_name, as its first archived file. This file is the table of contents of this package file, as shown in Example 12-5 on page 416.

*Example 12-5 Extracting the lpp_name file*

```
# mp64.5.2.0.2.U ./lpp_name
x ./lpp_name
# ls -ld ./lpp_name
-r-xr-xr-x   1 root     system         1823 Nov 18 11:24 ./lpp_name
# cat ./lpp_name
4 R S bos.mp64 {
bos.mp64 5.2.0.2 01 b U en_US Base Operating System 64-bit Multiprocessor
Runtime
[
%
/usr/lib/boot 19776
/usr/lpp/SAVESPACE 19776
/usr/lib/objrepos 8
/tmp 0 14656
INSTWORK 64 32
%
%
%
IY35438  3 dio read to an unpinned page
IY35432  3 Fixes for JFS2 on filesystems larger than 2TB
... many lines are skipped ...
]
}
```

Upon invoking the **inutoc** command, it parses the specified directory and extracts the lpp_name file from each package file, then concatenates the extracted information to the .toc file.

## Installation control library file (liblpp.a)

A package also has to contain an installation control library file, named liblpp.a, under the appropriate sub-directory. This file is an ar format archive file, which contains the files, as shown in Example 12-6.

*Example 12-6 Extracting the liblpp.a file*

```
# restore -qxf bos.mp64.5.2.0.2.U ./usr/lpp/bos.mp64/bos.mp64/5.2.0.2/liblpp.a
x ./usr/lpp/bos.mp64/bos.mp64/5.2.0.2/liblpp.a
# cd ./usr/lpp/bos.mp64/bos.mp64/5.2.0.2
# ls -l liblpp.a
-r-xr-xr-x   1 root     system        11382 Nov 18 11:24 liblpp.a
# file liblpp.a
liblpp.a:       archive
# ar -t liblpp.a
productid
bos.mp64.copyright
bos.mp64.inventory
bos.mp64.size
bos.mp64.al
bos.mp64.fixdata
```

The <fileset_name>.al file contains the list of files in the fileset. Example 12-7 shows you the contents of the bos.mp64.al file, which contains one file that is specified using the relative path name starting from the root directory.

*Example 12-7 Contents of the bos.mp64.al file*

```
./usr/lib/boot/unix_64
```

The <fileset_name>.inventory file contains the required information about files within the fileset (see Example 12-8). Each file has entries explained in Table 12-3.

*Table 12-3 Definition of entries in <fileset_name>.inventory*

| Entry name | Description |
|---|---|
| owner | Specifies the file owner. |
| group | Specifies the file group. |
| mode | Specifies the permission bit of the file. |
| type | Specifies the file type. |
| links | Specifies a hard-link of the file (if available). |
| class | The logical group of the file. |
| size | Specifies the size of the file. |
| checksum | Specifies the result of the **cksum** command to the file. |

These values are used to verify the contents of restored files from the package file if those files are restored correctly.

*Example 12-8 Contents of the bos.mp64.inventory file*

```
/usr/lib/boot/unix_64:
          owner = root
          group = system
          mode = 555
          type = FILE
          class = apply,inventory,bos.mp64
          size = 10122374
          checksum = "07429  9886 "
```

For further information about the format of these files contained in the liblpp.a file, please refer to *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

## 12.2 Packaging applications using mkinstallp

Beginning with AIX 5L Version 5.2 plus 5200-01 Recommended Maintenance Level, AIX provides a command, **`/usr/sbin/mkinstallp`**, to allow users to create their own software packages for AIX. Packages created with **`mkinstallp`** are in the AIX standard packaging format and can be installed or removed with SMIT or the **`installp`** command.

### 12.2.1 mkinstallp

A new fileset, bos.adt.insttools, includes the following files:

| | |
|---|---|
| **/usr/sbin/mkinstallp** | Front-end command |
| **/usr/sbin/makebff.pl** | Back-end command |
| **/usr/lpp/bos/README.MKINSTALLP** | Documentation |

**Note:** The makebff.pl Perl script was originally written in the development phase of the IBM Redbook *Managing AIX Server Farms*, SG24-6606. Therefore, it is possible to package your applications on previous versions of AIX; however, the script provided in the redbook, which is downloadable, might be obsolete and is not supported by IBM.

Files to be packaged by **`mkinstallp`** must be in a directory structure so that the location of the file relative to the root build directory is the same as the destination of the file after installation.

For example, if /usr/your_app/bin/*command1* is to be installed from a package, *command1* must be in the <build_root>/usr/your_app/bin directory when **`mkinstallp`** is invoked as shown in Figure 12-5 on page 419.

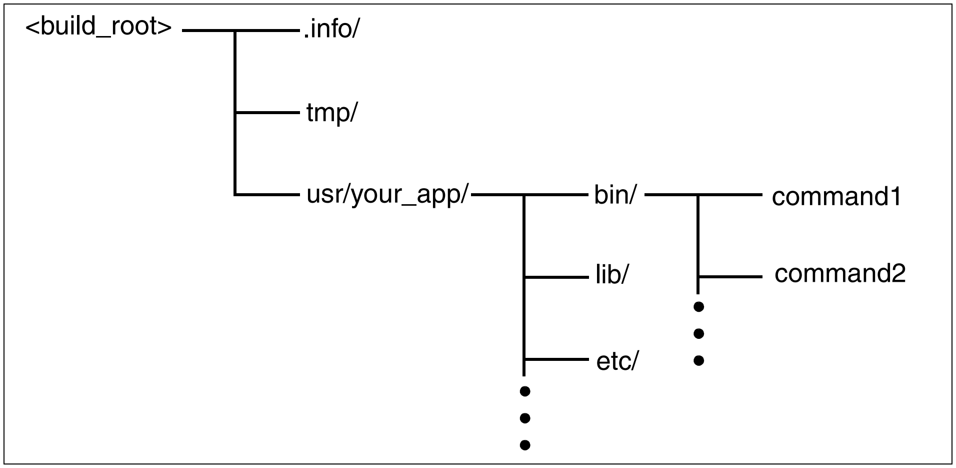


*Figure 12-5 Directory structure for packaging*

Once the contents of a package are in the correct directory structure, **mkinstallp** prompts for basic package data from the command line interface. This data includes the package name, requisites, descriptions of files to be packaged, and so on. The **mkinstallp** command will then generate a template file based on responses given by the user. Template files can be created or edited directly by the user and passed to the **mkinstallp** command with the -T flag to prevent command line prompting.

The **mkinstallp** command has the following command syntax:

```
mkinstallp [ -d build_root_dir ] [ -T template_file ]
```

Where:

**-d build_root_dir** Specifies the build root directory containing the files to be packaged. If omitted, the current working directory is used.

**-T template_file** Specifies the full path name of the template file to be passed to **mkinstallp**. If omitted, **mkinstallp** will prompt for package information and create a new template file based on user responses.

## 12.2.2 Packaging examples

We demonstrate how to use the **mkinstallp** command in order to package a sample program whose path name is /usr/redbooks/bin/vmgetinfo in this section. Our **vmgetinfo** command is an executable file compiled from the C source file shown in Example 3-17 on page 149.

The sample application, vmgetinfo, would fail with errno 109 (ENOSYS), as shown in the following example, if it is executed on AIX 5L Version 5.1, because it calls the vmgetinfo() system call, which is supported from Version 5.2:

```
$ oslevel -r
5100-03
$ vmgetinfo
vmgetinfo() at 16 in my_application.c failed with errno = 109: Function not
implemented
```

Assuming that the build root directory is /tmp/packages, we have prepared the install target files (the command itself and its source file), as shown in the following directory structure:

```
# pwd
/tmp/packages
# ls -lFR .
total 8
drwxr-xr-x   3 root     system          512 Feb 24 14:37 usr/
./usr:
total 8
drwxr-xr-x   4 root     system          512 Feb 24 14:37 redbooks/

./usr/redbooks:
total 16
drwxr-xr-x   2 root     system          512 Feb 24 14:39 bin/
drwxr-xr-x   2 root     system          512 Feb 24 14:39 src/

./usr/redbooks/bin:
total 16
-r-xr-xr-x   1 root     system         5898 Feb 24 14:39 vmgetinfo*

./usr/redbooks/src:
total 8
-rw-r--r--   1 root     system          922 Feb 24 14:39 vmgetinfo.c
```

**Note:** The owner, group, and permission modes must be carefully verified before actually creating your packages.

## Preparing a template file

We have prepared the template file shown in Example 12-9 on page 421 in order to create a package, called vmgetinfo. In this example, the vmgetinfo package consists of two filesets: vmgetinfo.rte and vmgetinfo.src.

As shown in the high-lighted lines in Example 12-9 on page 421, vmgetinfo.rte requires the bos.rte.libc fileset to be installed with level 5.2.0.0 or higher. This requisite condition would prevent users from installing this fileset on AIX 5L Version 5.1 and earlier. Also, vmgetinfo.src requires that vmgetinfo.rte be installed. Therefore, even if a user selects vmgetinfo.src only, vmgetinfo.rte will be installed automatically.

For the complete definition of requisite condition keywords, refer to the “Packaging Software for Installation” section in *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

*Example 12-9 A template file (redbook.tmplt)*

```
Package Name: vmgetinfo
Package VRMF: 1.0.0.1
Update: N
Fileset
  Fileset Name: vmgetinfo.rte
  Fileset VRMF: 1.0.0.1
  Fileset Description: vmgetinfo runtime
  Bosboot required: N
  License agreement acceptance required: N
  Name of license agreement:
  Include license files in this package: N
  License file path:
  Requisites: *prereq bos.rte.libc 5.2.0.0
  Files
    /usr/redbooks
    /usr/redbooks/bin
    /usr/redbooks/bin/vmgetinfo
  EOFiles
EOFileset
Fileset
  Fileset Name: vmgetinfo.src
  Fileset VRMF: 1.0.0.0
  Fileset Description: vmgetinfo source
  Bosboot required: N
  License agreement acceptance required: N
  Name of license agreement:
  Include license files in this package: N
  License file path:
  Requisites: *coreq vmgetinfo.rte 1.0.0.1
  Files
    /usr/redbooks/src
    /usr/redbooks/src/vmgetinfo.c
  EOFiles
EOFileset
```

The supported keywords in template files are shown in Table 12-4 on page 422.

*Table 12-4 Template file keywords*

| Keyword | Description |
|---|---|
| Package Name* | Name of the package. |
| Package VRMF* | Version, Release, Modification, and Fix level of the package. |
| Update* | Is this an update package? |
| Fileset* | Start of a new fileset. |
| Fileset Name* | Name of the fileset. |
| Fileset VRMF* | VRMF of the fileset. |
| Fileset Description | Description of the fileset. |
| Bosboot required* | Is a bosboot required when installing this fileset? |
| License agreement acceptance required | Is license agreement acceptance required for this fileset? |
| Name of license agreement | Name of the license agreement.[a] |
| Include license files in this package | Are the license files included in this package? |
| License file path | Path of the license file(s).[b] |
| Requisites | Requisite conditions (coreq, ifreq, instreq, or prereq) for the fileset.[c] |
| Files* | Start of the files section. |
| /path/to/file | File path.[d] |
| EOFiles* | End of the files section. |
| EOFileset* | End of the fileset. |

a. The Name of the license agreement is defined as LAR/path/to/license/agreement. The %L tag can be used in place of a hardcoded path to represent the locale of the machine that the package will be installed on. For example, if a package is installed in the en_US locale, %L will be converted to en_US.

b. The License file path is defined as LAF/path/to/license/file. Multiple license files are separated by semicolons.

c. Requisites are defined as *Type_keyword Name VRMF; Type_keyword is either coreq, ifreq, instreq, or prereq. Multiple requisites are separated by semicolons.

d. The full path name for each file in the fileset must be listed in the Files section. Any custom directories should also be listed in this section. For example, to package the /usr/xyz/foo file, list both the /usr/xyz directory and the /usr/xyz/foo file in the Files section. Each entity in the final package will have the same attributes (owner/group/permissions) that it had at build time. The user must ensure that file attributes in the build root directory are correct prior to running **mkinstallp**.

**Note:** Keywords with a * are required, and will cause **mkinstallp** to fail if left blank or omitted in the template file.

## Creating a package file

Once the template file is prepared, run the **mkinstallp** command to create a package file. In the following example, the template file is redbook.tmplt and we invoked the command in the build root directory, /tmp/packages:

```
# pwd
/tmp/packages
# ls -l
total 16
-rw-r--r--   1 root     system          837 Feb 24 15:01 redbook.tmplt
drwxr-xr-x   4 root     system          512 Feb 24 15:00 usr/
# mkinstallp -T redbook.tmplt
Using '/tmp/packages' as the base package directory.
Cannot find '/tmp/packages/.info'. Attempting to create.
Using '/tmp/packages/.info' to store package control files.
Cleaning intermediate files from '/tmp/packages/.info'.

Using 'redbook.tmplt' as the template file.
vmgetinfo 1.0.0.1 I
processing vmgetinfo.rte
processing vmgetinfo.src
creating ./.info/liblpp.a
ar: Creating an archive file ./.info/liblpp.a.
creating ./tmp/vmgetinfo.1.0.0.1.bff
```

The command will generate the package file in the ./tmp directory underneath the build root directory as shown in the following example:

```
# ls ./tmp
vmgetinfo.1.0.0.1.bff
# inutoc ./tmp
# ls ./tmp
.toc                    vmgetinfo.1.0.0.1.bff
# installp -ld ./tmp
  Fileset Name                Level                     I/U Q Content
  ====================================================================
  vmgetinfo.rte               1.0.0.1                    I  N usr
#   vmgetinfo runtime
```

```
  vmgetinfo.src             1.0.0.0                     I  N usr
#   vmgetinfo source
```

## 12.2.3 Verification of packages

Once the package is created, it must be verified before the actual software deployment phase on production systems.

### Installation verification

The created package must be smoothly installed on the target system if it satisfies the install requisite conditions. If the target system does not satisfy the conditions, the package must be prevented from being installed.

In this example, the following tests must be done:

- Installing only the vmgetinfo.rte fileset on AIX 5L Version 5.2 and later.

  The vmgetinfo.src fileset should not be installed automatically.
- Installing only the vmgetinfo.src fileset on AIX 5L Version 5.2 and later.

  The vmgetinfo.src fileset should not be installed automatically, unless the following conditions are met:
  - The -g flag of **installp**, which instructs the **installp** command to automatically install prerequisite filesets, is specified.
  - The vmgetinfo.rte fileset has been already installed.
- Installing the filesets on AIX 5L Version 5.1 and earlier.

  The filesets must not be installed.

### Verification of the installed files

Check if the installed files are placed, according to the fileset inventory information, with the **lppchk** command. There are four flags of the **lppchk** command for verifying the files:

**-f** Fast check (file existence, file length)

**-c** Checksum verification

**-v** Fileset version consistency check

**-l** File link verification

All four calls of **lppchk** with the different flags has to return with a zero exit code; if not, there are some inconsistent files, as shown in the following example:

```
$ lppchk -f -m 3 vmgetinfo.rte; echo $?
lppchk: 0504-230  3 files have been checked.
0
$ lppchk -c -m 3 vmgetinfo.rte; echo $?
lppchk: 0504-230  3 files have been checked.
0
$ lppchk -l -m 3 vmgetinfo.rte; echo $?
0
$ lppchk -v -m 3 vmgetinfo.rte; echo $?
0
```

## Verification of correct functionality

Use the **lslpp** command to show all installed files:

```
# lslpp -L vmgetinfo.rte
  Fileset                      Level  State  Type  Description (Uninstaller)
  ----------------------------------------------------------------------------
  vmgetinfo.rte              1.0.0.1    C     F    vmgetinfo runtime

# lslpp -f vmgetinfo.rte
  Fileset               File
  ----------------------------------------------------------------------------
Path: /usr/lib/objrepos
  vmgetinfo.rte 1.0.0.1
                        /usr/redbooks/bin/vmgetinfo
                        /usr/redbooks
                        /usr/redbooks/bin
```

After installing and validating of the installed package, test your installed application to see if it works as designed. In this example, we have confirmed that the command returned the same output shown in Example 3-16 on page 148.

## Uninstallation verification

As the last step of the verification process, check if the package can be un-installed correctly by using the **installp -u** command, as shown in the following example:

```
# installp -u vmgetinfo.rte
+-----------------------------------------------------------------------------+
                    Pre-deinstall Verification...
+-----------------------------------------------------------------------------+
Verifying selections...done
Verifying requisites...done
Results...
```

```
SUCCESSES
---------
  Filesets listed in this section passed pre-deinstall verification
  and will be removed.

  Selected Filesets
  -----------------
  vmgetinfo.rte 1.0.0.1                       # vmgetinfo runtime

  << End of Success Section >>

FILESET STATISTICS
------------------
    1  Selected to be deinstalled, of which:
        1  Passed pre-deinstall verification
  ----
    1  Total to be deinstalled

+-----------------------------------------------------------------------------+
                           Deinstalling Software...
+-----------------------------------------------------------------------------+

installp:  DEINSTALLING software for:
        vmgetinfo.rte 1.0.0.1

Finished processing all filesets.  (Total time:  0 secs).

+-----------------------------------------------------------------------------+
                                Summaries:
+-----------------------------------------------------------------------------+

Installation Summary
--------------------
Name                        Level           Part        Event       Result
-------------------------------------------------------------------------------
vmgetinfo.rte               1.0.0.1         USR         DEINSTALL   SUCCESS
```

After removing the fileset, the installation target directory (if it is unique to the fileset) should be removed also. In this example, all files and sub-directories under /usr/redbooks are removed.

Now this package is ready to be deployed on production systems.

### 12.2.4 Optional installation control executable files

The AIX standard packaging format allows you not only to package your applications, but also to add optional files in order to control the installation process. For example, if your application requires a configuration file that depends on the host name where the application is installed, an optional executable file, *fileset*.config, can be used to automatically create the configuration file upon the fileset installation.

The optional installation control executable files shown in Table 12-5 are called during the installation process. Unless otherwise noted, file names that end in _i are used during installation processing only, and file names that end in _u are used in file set update processing only. All files in Table 12-5 are optional and can be either shell scripts or executable object modules. Each program should have a return value of 0 (zero), unless the program is intended to cause the installation or update to fail.

In order to include these optional installation control executable files, create the files in the .info sub-directory underneath the build root directory before creating the package.

*Table 12-5 Optional installation control executable files*

| File name | Description |
|---|---|
| *fileset*.config<br>*fileset*.config_u | Modifies configuration near the end of the default installation or update process. Fileset.config is used during installation processing only. |
| *fileset*.odmdel<br>*fileset*.*.odmdel | Updates ODM database information for the file set prior to adding new ODM entries for the file set. The odmdel file naming conventions enables a file set to have multiple odmdel files. |
| *fileset*.pre_d | Indicates whether a file set may be removed. The program must return a value of 0 (zero) if the file set may be removed. File sets are removable by default. The program should generate error messages indicating why the file set is not removable. |
| *fileset*.pre_i<br>*fileset*.pre_u | Runs prior to restoring or saving the files from the apply list in the package, but after removing the files from a previously installed version of the file set. |
| *fileset*.pre_rm | Runs during a file set installation prior to removing the files from a previously installed version of the file set. |
| *fileset*.post_i<br>*fileset*.post_u | Runs after restoring the files from the apply list of the file set installation or update. |
| *fileset*.unconfig<br>*fileset*.unconfig_u | Undoes configuration processing performed in the installation or update. Fileset.unconfig is used during installation processing only. |
| *fileset*.unodmadd | Deletes entries that were added to ODM databases during the installation or update. |

| File name | Description |
|---|---|
| *fileset*.unpost_i_0<br>*fileset*.unpost_u | Undoes processing that is performed following the restoration of the files from the apply list in the installation or update. |
| *fileset*.unpre_i<br>*fileset*.unpre_u | Undoes processing performed prior to restoring the files from the apply list in the installation or update. |

For more information, please refer to the “Packaging Software for Installation” section in the *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

# A

# Previous versions of C and C++ compiler products

Over the years, IBM has offered a variety of compiler products to perform the compilation of C and C++ programs on AIX. These C and C++ compilers have evolved over time and can be tracked by their different version numbers. The C and C++ compilers introduced new compiler features to take advantage of the functionality included in new releases of the AIX operating system.

This appendix gives you background information on versions of C and C++ compilers prior to C for AIX Version 6.0 and VisualAge C++ for AIX Version 6.0, by providing the following sections.

- “Compiler product similarities” on page 430
- “IBM C compilers” on page 431
- “IBM C++ compilers” on page 435

# Compiler product similarities

All of the IBM C and C++ compiler products for AIX Version 4 share some similar characteristics, in particular, the way the products are installed on the system and the configuration options available when using the products.

## Multiple command line drivers

Each compiler product, with the exception of VisualAge C++ Professional for AIX Version 4.0, has multiple command line driver interfaces available, each causing a different set of default arguments to be used. For example, the C compiler products provide commands, such as **`cc`**, **`xlc`**, **`c89`**, **`cc_r`**, and so on. These commands are all links to a single compiler core, which uses a specific set of options, depending on the name of the command used to invoke it.

In addition to the default invocation commands provided when the compiler is installed, the system administrator can create new commands, which result in the compiler being invoked with a customized set of default options. This feature is controlled by the compiler configuration file, which lists the options to be used for each invocation command. The exact name of the configuration file differs between the compiler products, but generally has a name of the form /etc/*comp*.cfg, where *comp* indicates the compiler product that uses the configuration file.

### Finding the compiler drivers

The earlier versions of the compiler products automatically created symbolic links in /usr/bin for each invocation command supplied by the compiler. For example, this means that if a user has the directory /usr/bin as part of their PATH environment variable (which it is, by default), they need only type **`cc`** on the command line to invoke the **`/usr/bin/cc`** command.

The later versions of the compiler products are designed to co-exist with earlier versions, and, as a consequence, they do not automatically create the symbolic links in /usr/bin when they are installed. This means that a user may have trouble invoking the compiler on a system that only has a new version compiler product installed. There are two solutions available in this instance:

- When logged in as the root user, invoke the **`replaceCSET`** command supplied with the compiler. This will create appropriate symbolic links in /usr/bin to the compiler driver programs.
- Alter the PATH environment variable to add the directory that contains the compiler driver programs. For example:

  ```
  PATH=/usr/vac/bin:$PATH; export PATH
  ```

The second solution should be used if two compilers are installed on a system, since it allows each user to choose which version of the compiler they wish to use. If the system only has one compiler installed, it makes sense to use the first solution. If required, the root user can reverse the action of the **`replaceCSET`** command by using the **`restoreCSET`** command, which is also supplied with the compiler. The exact location of the **`replaceCSET`** and **`restoreCSET`** commands will depend on the version of the compiler you are using. Note that the **`replaceCSET`** and **`restoreCSET`** tools are removed in Version 6.0 of the C and C++ compiler products.

## Installation directory

The main components of the compiler product are installed on the system in the /usr file system. The exact directory used depends on the compiler product. Table A-1 on page 435 shows the location of this directory for C compiler products. Table A-2 on page 439 shows the location of this directory for C++ compiler products.

# IBM C compilers

This section describes the various IBM C compilers for AIX. The details provided are limited to compatibility issues of the various compilers with the different versions of the AIX operating system. This information can help in deciding which version of a compiler product should be used in a given situation, based on the version of AIX that will be used. The information can also help avoid problems when upgrading a compiler product to a newer version.

AIX Version 3.2 included a bundled C compiler as part of the operating system. This compiler was known as XL C Version 1.3. When AIX Version 4.1 was introduced, the compiler product was unbundled from the operating system and offered for sale as a separately orderable product. This appendix covers the C and C++ compiler products prior to C for AIX Version 6.0.

## IBM XL C Version 3

The IBM XL C Version 3 product was the first C compiler product from IBM for AIX Version 4.1. The *Version 3* in the name of the product specifies the version number of the compiler product rather than the version of the AIX operating system the compiler is compatible with.

Initially released as Version 3.1.0, this compiler evolved over time with the addition of Program Temporary Fixes (PTFs) to become XL C Version 3.1.4, which was supported on AIX Version 4.1 and AIX Version 4.2. This compiler was not supported on AIX Version 3.2 and is not supported on AIX Version 4.3.

C programs written using C Set ++ Version 2 and XL C Version 1.3 on AIX Version 3.2 are source compatible with XL C Version 3, with some exceptions in detecting invalid programs or areas where results are undefined. These exceptions are documented in the product README file.

The compiler product itself is installed in /usr/lpp/xlC, and symbolic links are created in /usr/bin for the command line driver programs, for example, /usr/bin/cc and /usr/bin/c89. The default PATH environment variable means that most users need only type `cc` on the command line to invoke the /usr/bin/cc driver program, which, in turn, is a symbolic link to the driver /usr/lpp/xlC/bin/xlC. The compiler configuration file is /etc/xlC.cfg.

The XL C Version 3 compiler product uses the Net/LS licensing system to control usage of the product.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 6.0 compiler product.

## IBM C for AIX Version 4.1

A new version of the C compiler product was introduced with AIX Version 4.3.0. The compiler product, IBM C for AIX Version 4.1, had a number of new features, including new optimization routines for improved execution performance, new inter-procedural analysis tools, precompiled headers for improved compiler performance, improved memory management, and improved prototyping of programs. This version of the compiler was supported on AIX Version 4.1.4, AIX Version 4.2, and AIX Version 4.3.

C programs written using either Version 2 or 3 of IBM C Set ++ for AIX Version 3 of IBM XL C, or the XL C compiler component of AIX Version 3.2 are source compatible with C for AIX Version 4.1, with some exceptions to detect invalid programs or areas where results are undefined.

Two important configuration differences introduced with this version of the compiler are:

1. The compiler product is now installed under /usr/vac rather than /usr/lpp/xlC.
2. The installation process does not create symbolic links to the driver programs from the /usr/bin directory. This is because the C compiler has been designed to co-exist on a system that already has the previous version of the C compiler or a version of the C Set ++ compiler installed.

If the system does not have another version of the compiler installed, the symbolic links in the /usr/bin directory can be created by invoking the **/usr/vac/bin/replaceCSET** command, which, as the name implies, replaces the symbolic links to the C Set driver programs.

The compiler product also includes the **/usr/vac/bin/restoreCSET** command, which can be used to reverse the actions of the **replaceCSET** command.

Alternatively, if multiple versions of the compiler exist, or if the user does not want to create symbolic links in /usr/bin, the setting of the PATH environment variable can be used to determine which compiler product is used.

For example, setting the PATH environment variable as follows:

```
PATH=/usr/vac/bin:$PATH; export PATH
```

will result in the C for AIX Version 4.1 compiler being used when the **cc** command is invoked.

The compiler configuration file is /etc/vac.cfg.

The C for AIX Version 4.1 compiler uses the License Use Management (LUM) licensing system to control usage of the product. Refer to 1.4, “Activating the compilers” on page 23 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 6.0 compiler product.

## IBM C for AIX Version 4.3

The IBM C for AIX Version 4.3 compiler product was introduced shortly after the release of AIX Version 4.3.0 and the 64-bit hardware models of the RS/6000 family. This version of the compiler was similar to the IBM C for AIX Version 4.1 compiler, except that it added support for creating and debugging 64-bit application binaries for use on the 64-bit hardware. This version of the compiler is installed under /usr/vac, and uses the /etc/vac.cfg configuration file. If C for AIX Version 4.1 is already installed, installing C for AIX Version 4.3 will overwrite and upgrade the previous version.

The C for AIX Version 4.3 compiler uses the LUM licensing system to control usage of the product. Refer to 1.4, “Activating the compilers” on page 23 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 6 compiler product.

## IBM C for AIX Version 4.4

The IBM C for AIX Version 4.4 compiler product was an improved version of the previously released C for AIX Version 4.3. The main enhancement was that this compiler was designed to exploit the RS/6000 Symmetric Multi-Processing (SMP) architecture. It supported automatic parallellization of a C program as well as explicit parallellization through a set of directives that enabled the user to parallelize selected sections of the application program. This version of the C compiler was supported only by AIX Version 4.2 and 4.3.

C programs written using either Version 2 or Version 3 of IBM C Set ++ for AIX, the XL C compiler component of AIX Version 3.2, or previous versions of the C for AIX Version 4.x compilers, are source compatible with C for AIX Version 4.4, with some exceptions to detect invalid programs or areas where results are undefined.

The compiler is installed under /usr/vac, and uses the /etc/vac.cfg configuration file. If a previous version of C for AIX 4.x is installed, installing C for AIX Version 4.4 will overwrite and upgrade the previous version.

The C for AIX Version 4.4 compiler uses the LUM licensing system to control usage of the product. Refer to 1.4, “Activating the compilers” on page 23 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to the IBM C for AIX Version 6.0 compiler product.

## IBM C for AIX Version 5.0

The C for AIX Version 5.0 compiler extends the existing symmetric multi-processing (SMP) support available with C for AIX Version 4.4 by supporting the OpenMP industry specification. OpenMP provides a model for parallel programming that allows a program to be portable across shared memory architectures from different vendors by using a common set of application program interfaces. The compiler generates highly-optimized code for all RS/6000 processors and can provide run-time address checking to detect memory errors.

This compiler is supported only by IBM AIX Version 4.2.1 or later. Also, note that 64-bit applications will run only on AIX Version 4.3 and later when running on

64-bit hardware. C for AIX Version 5.0.2 adds support for AIX Version 5.1 and Version 5.2.

C programs written using Version 3 of XL C or Version 4 of IBM C for AIX are source compatible with IBM C for AIX Version 5.0. C programs written using either Version 2 or 3 of IBM Set ++ for AIX or the XL C compiler component of AIX Version 3.2 are source compatible with IBM C for AIX Version 5.0 with exceptions to detect invalid programs or areas where results are undefined.

This version of the compiler is installed under /usr/vac and uses the /etc/vac.cfg configuration file. If C for AIX Version 4.x is installed on a system, installing C for AIX Version 5.0 will overwrite and upgrade the previous version.

The C for AIX Version 5.0 compiler uses the LUM licensing system to control usage of the product. Refer to 1.4, "Activating the compilers" on page 23 for information on configuring the licence system.

## C compiler summary

Table A-1 summarizes the various versions of IBM C compiler products for AIX.

*Table A-1 IBM C compilers for AIX*

| **Compiler** | **Installation directory** | **Configuration file** | **Supported AIX levels** | **Licensing method** | **Drivers in /usr/bin** |
|---|---|---|---|---|---|
| XL C Version 3 | /usr/lpp/xlC | /etc/xlC.cfg | 4.1 and 4.2 | Net/LS | Yes |
| C for AIX Version 4.1 | /usr/vac | /etc/vac.cfg | 4.1.4, 4.2, and 4.3 | iFOR/LS | No |
| C for AIX Version 4.3 | /usr/vac | /etc/vac.cfg | 4.1.5, 4.2, and 4.3 | LUM | No |
| C for AIX Version 4.4 | /usr/vac | /etc/vac.cfg | 4.2 and 4.3 | LUM | No |
| C for AIX Version 5.0 | /usr/vac | /etc/vac.cfg | 4.2 and 4.3 | LUM | No |

# IBM C++ compilers

This section describes the various IBM C++ Compilers for AIX. The details provided here are, again, limited to compatibility issues of the various compilers with the different versions of the AIX operating system. This information can help decide which C++ Compiler product to use for a particular project, based on the target version of AIX, and the nature of the C++ source code being compiled.

## IBM C Set ++ for AIX Version 3

The IBM C Set ++ for AIX Version 3 product was the first C++ compiler product from IBM for AIX Version 4.1. The *Version 3* in the name of the product specifies the version of the compiler product rather than the version of the AIX operating system the compiler is compatible with. The C Set ++ for AIX Version 3 product is, in effect, an extension of the C for AIX Version 3 compiler. An alternative view is that the C for AIX Version 3 compiler is a subset of the C Set ++ for AIX compiler.

This compiler was initially released as Version 3.1.0, and evolved over time with the addition of Program Temporary Fixes (PTFs) to become C Set ++ for AIX Version 3.1.4, which was supported on AIX Version 4.1 and AIX Version 4.2. It was not supported on AIX Version 3.2 and is not supported on AIX Version 4.3.

C++ programs written using C Set ++ Version 2 on AIX Version 3.2 are source compatible with C Set ++ for AIX Version 3, with some exceptions to detect invalid programs or areas where results are undefined. These exceptions are documented in the product README file.

The compiler product itself is installed in /usr/lpp/xlC, and symbolic links are created in /usr/bin for the command line driver programs, for example, /usr/bin/xlC. The default PATH environment variable means that most users need only type `xlC` on the command line to invoke the /usr/bin/xlC driver program, which, in turn, is a symbolic link to the driver /usr/lpp/xlC/bin/xlC. The compiler configuration file is /etc/xlC.cfg.

The C Set ++ for AIX Version 3 compiler product uses the Net/LS licensing system to control usage of the product.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to IBM VisualAge C++ for AIX Version 6.0.

## IBM C and C++ compilers Version 3.6

The official name of this product is IBM C and C++ compilers for AIX, OS/2®, and Windows NT. The product is part of a family of related compilers, with versions available for each of the mentioned platforms. The product is sometimes referred to as C Set ++ Version 3.6. The AIX Version of the product is the follow on to C Set ++ Version 3 for AIX.

The product offered a number of facilities to assist in the development of cross-platform applications, where the same source code is used on multiple platforms. The product includes a rich set of IBM class libraries, memory management routines, graphical debuggers, and resource tools for creating and compiling resources and converting between platform formats. The C compiler component of the product can produce either 32-bit or 64-bit executable files when used on AIX Version 4.3.

The product is supported on AIX Version 4.1.4, Version 4.2, and Version 4.3.

C++ programs written using Version 3 of C Set ++ for AIX and earlier are source compatible with the C++ compiler of C Set ++ for AIX Version 3.6.

As with the other post Version 3.1 compiler products, the compiler command drivers are not created in /usr/bin when the product is installed.

The installation directory is /usr/ibmcxx, and the configuration file is /etc/ibmcxx.cfg.

The product uses the LUM license management system to control usage of the product. Refer to 1.4, “Activating the compilers” on page 23 for information on configuring the licence system.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to IBM VisualAge C++ for AIX Version 6.0.

## IBM VisualAge C++ Professional for AIX Version 4

VisualAge C++ Professional for AIX Version 4 is a powerful rapid application development (RAD) tool for building C and C++ applications. This heterogeneous RAD environment provides:

- Tools, including a graphical debugger
- Visual Builder and Data Access Builder
- Incremental compiler and linker
- A rich set of class libraries
- Online help and a powerful full-text search engine

VisualAge C++ Professional for AIX provides a standards-compliant C++ compiler. Its incremental development environment and visual programming tools improve programmer productivity.

This product features an incremental compiler and linker and, as such, is not ideally suited for use when working with existing application code that uses Makefiles or for a development environment that maintains a single source tree for multiple platforms and uses Makefiles. For this reason, the product includes a copy of the IBM C Set ++ Version 3.6 compiler for use in a batch compile environment.

This compiler product runs on IBM AIX Version 4.1.5, Version 4.2, and Version 4.3 for RS/6000.

C++ programs written using Version 3.6 of IBM C and C++ compilers and earlier are source compatible with the C++ compiler component of VisualAge C++ Professional for AIX Version 4.

> **Note:** As described above, this version of VisualAge features an incremental compiler. The implications of this for productivity and the code are impressive, but if the application is moving from a batch environment, do spend time with the application to adapt to the VisualAge products. For example, makefiles cannot be processed directly by the incremental compiler.
>
> But, once the migration is done, then the advantages of VisualAge products are very impressive. This then would reduce the amount of time and memory required to do each build as well as the time spent on rebuilding when some changes are made to the source files.

This product has been withdrawn from marketing and is no longer available for purchase. Support for this product has also been discontinued. Current users of this product are encouraged to upgrade to IBM VisualAge C++ for AIX Version 6.0.

## IBM VisualAge C++ Professional for AIX Version 5

VisualAge C++ Professional for AIX Version 5.0 features a fully incremental compiler and a batch compiler. The Integrated Development Environment (IDE) operates with the incremental compiler when used in the AIX Common Desktop Environment (CDE). The batch compiler is run from the command line and is suitable for use in a development environment that uses Makefiles. Both compilers support the latest ANSI/ISO C++ language standard and the latest version (Version 5) of the IBM Open Class library.

The main differences between Version 4 and Version 5 of this product are:

- Version 5 supports multiple codestores in a single project.
- Version 5 is a single product featuring both batch and incremental compilers.

The graphical interface of Version 5 has been redesigned with a host of helpful features. Version 5 has improved optimization techniques and provides the programmer with effective and efficient ways handling of C++ object code. Also, this product allows the developer to carry out performance analysis to determine the applications usage of system resources.

This product is supported on IBM AIX Version 4.2.1 and later versions for RS/6000 hardware.

C++ programs written using Version 4 of IBM VisualAge C++ Professional for AIX are source compatible with the VisualAge C++ Professional for AIX Version 5. However, programs written using IBM C and C++ Compilers for AIX Version 3.6, and earlier are not source compatible because the former compilers were based on the ISO C++ Draft.

The C compiler component of VisualAge C++ Professional for AIX Version 5 is provided by the IBM C for AIX Version 5 compiler.

## C++ compiler summary

Table A-2 summarizes the various IBM C++ compiler products for AIX.

*Table A-2 C++ compiler products*

| Compiler | Installation directory | Configuration file | Supported AIX levels | Licensing method | Drivers in /usr/bin |
|---|---|---|---|---|---|
| C Set ++ for AIX Version 3 | /usr/lpp/xlC | /etc/xlC.cfg | 4.1, 4.2 | Net/LS | Yes |
| IBM C and C ++ compilers Version 3.6 | /usr/ibmcxx | /etc/ibmcxx.cfg | 4.1.4, 4.2, 4.3 | LUM | No |
| VisualAge C++ Professional for AIX Version 4 | /usr/vacpp | /etc/vacpp.cfg | 4.1.5, 4.2, 4.3 | LUM | No |
| VisualAge C++ Professional for AIX Version 5 | /usr/vacpp | /etc/vacpp.cfg<br>/etc/vac.cfg | 4.2.1, 4.3 | LUM | No |

B

# Useful information for linking and loading on AIX

This appendix gives you the back ground information about the linking and loading processes on AIX by providing the following sections:

- “A brief history of UNIX programming development” on page 442
- “Historical view of linking and loading in AIX” on page 443
- “Definitions” on page 443

# A brief history of UNIX programming development

The traditional UNIX compilation technique historically involved *static linking*. With this technique, multiple object files defining global symbols and containing code and data were combined and written to an executable file with all the references resolved. The executable was a single self contained file, and often became quite large. However, since the name and location of all symbols were resolved at link-time, a program could be executed by simply reading it into memory and transferring control to its entry point. Static linking, which is still used in certain circumstances, has the following drawbacks:

- If any of the libraries used by the program are updated, the program needs to be re-linked to take advantage of the updated libraries.
- Disk space is wasted because every program on the system contains private copies of every library functions that it needs.
- System memory is wasted because every running process loads into memory its own private copy of the same library functions.

To allow the libraries to be shared among different programs, the concept of *shared libraries* evolved. When a program is linked with a shared library, the shared library code is not included in the generated program executable file. Instead, information is saved in the program that allows the library to be found and loaded when the program is executed. Shared library code is loaded into global system memory by the first program that needs it and is shared by all programs that use it.

The advantages of shared libraries are:

- Less disk space is used because the shared library code is not included in the executable programs.
- Less memory is used because the shared library code is only loaded once.
- The time taken to start an application may be reduced because the shared library code may already be in memory.
- Performance may be improved because fewer page faults will be generated when the shared library code is already in memory.

When a program is linked with shared libraries, the resulting executable contains a list of symbols imported by it. The actual code is not included. Therefore, if one of the shared libraries is updated, programs using that library do not need to be re-linked, automatically picking up the current version of the library on its next execution. This makes application service and support easier since new versions of individual libraries containing patches and fixes can be shipped to customers without having to rebuild and ship a whole new version of the application.

When a program uses imported symbols, each of those symbols is associated with a specific shared library needed by the program. During program execution, the symbols do not have to be searched, but can be found by simply looking up their addresses in the specified shared library.

## Historical view of linking and loading in AIX

The linker and system loader on AIX are designed so that modules are self contained entities with well defined sets of imported and exported symbols. Symbol resolution is performed at link-time, simplifying the work of the system loader when a module is loaded. The system loader looks up symbols to relocate references, but does not perform symbol resolution. As a result, a shared module and its dependents can be *pre-relocated* in global memory. Once a set of modules has been pre-relocated, a program using the modules can be loaded more effectively since symbol lookup and relocation in the pre-relocated modules does not have to be performed.

However, there has been several circumstances where the traditional AIX linking mechanism did not satisfy the application programmers' requirements. For example, in many new applications, the location of some symbol definitions may not be known at link-time.

To address this requirement, AIX has been supporting the run-time linking method, which provide more flexibility in how and when symbol resolution takes place (see 2.5, "Run-time linking" on page 68).

## Definitions

The definitions shown in Table B-1 are very useful in understanding the technical details of the linking and loading process on AIX.

*Table B-1 Definitions of terms regarding linking and loading process on AIX*

| Term | Description |
|---|---|
| Object file | A generic term for a file containing executable code, data, relocation information, a symbol table, and other information. Object files on AIX for POWER processor architecture are defined by XCOFF (eXtended Common Object File Format). Please refer to the *AIX 5L Version 5.2 Files Reference* for further information about the XCOFF format. |
| CSECT | The atomic unit of relocation as far as link-editing is concerned. A CSECT can contain code or data. One CSECT can contain references to other CSECTs. These references are described by relocation entries (RLDs) contained in a section of an object file. |

| Term | Description |
|---|---|
| Module | The smallest, separately loadable and relocatable unit. A module is an object file containing a loader section. Modules are loaded implicitly when they are dependents of another loaded module. A module may have an entry point and may export a set of symbols. |
| Dependent module | A module loaded automatically as part of the process of loading another module. A module is loaded automatically if it is listed in the loader section of another module being loaded. |
| Executable | A module with an entry point. An executable may have exported symbols. |
| Linker | The application program that combines multiple object modules into an executable program. On AIX, the `ld` command is the system linker. The `ld` command processes command line arguments and generates a list of commands, which are piped to another program called the binder (`/usr/ccs/bin/bind`). For the purpose of this document, the terms linker and binder are used interchangeably. |
| System loader | A kernel subsystem that creates a process image from an executable file. The loader loads the executable file into memory, loads (or finds) the program's dependent modules, looks up imported symbols, and relocates references. |
| Load-time | The time period during which the system loads a module and its dependents. This includes the processing required to resolve symbols and relocate the modules. A module can be loaded with the exec(), dlopen(), or load() system calls. The dlopen() and loadAndInit() functions are higher level calls that ultimately call load(). |
| Exec-time | Exec-time is load-time for the main program. A new program is loaded when the exec() function is called. |
| Run time | Run time starts when the control enters main() and ends when the program exits. |
| Resolution | The act of associating a reference to a symbol, with the definition of that symbol. Symbol resolution is performed by the linker at link-time and by the system loader at load-time. Some special kinds of symbol resolution can occur during run time. |
| Archives | Archive files are composite objects that usually contain object files. On AIX, archives can contain shared objects, import files, or other kinds of members. By convention, the name of an archive usually ends with *.a*, but the magic number of a file (that is, the first few bytes of the file) is used to determine whether a file is an archive or not. |
| Library | A library is a file that contain the definitions of multiple symbols. |
| Export and Import lists | Import files are ASCII files that list symbols to be resolved at load-time, and often times, their defining modules. This information is saved in the loader section of the generated module, and is used by the system loader when the module is loaded. The import file can be used in place of a corresponding shared object as an input file to the `ld` command. Export files are ASCII files that list global symbols to be made available for another module to import. The file formats of the import and export file are the same. |

| Term | Description |
|---|---|
| Strip | A system command that reduces the size of an XCOFF module by removing the linking and debugging information. Object files that have been stripped cannot be used as input to the linker. A module that has been stripped can still be loaded, since the module's loader section contains the information needed to load the module. |
| Static linking | Executing the `ld` command so that shared objects are treated as ordinary (non shared) object files. A shared object that has been stripped cannot be linked statically. |
| load() call | The load() system call can be used to add a module into a running process, allowing a program to expand its own capabilities. The unload() system call can be used to remove object modules from an executing program, which were loaded with the load() routine. |
| loadbind() call | A system call that allows *deferred* references to be resolved after a module has been loaded. |
| Dynamic binding | The technique where imported symbols are looked up or resolved at load-time or run time. |
| Run-time linking | The technique where imported symbols are resolved at load-time. When run-time linking is used, symbols may resolve to alternate definitions that were not available at link-time. |
| Dynamic loading | The addition of modules to a running (executing) process. Modules can be loaded with the load(), dlopen(), or loadAndInit() functions. |
| Lazy loading | The ability to defer loading of a dependent module until a function in the dependent module is first called by the process. |
| Rebinding | Associating an alternate definition with an imported symbol at run time. By default, the linker identifies a specific dependent module with each imported symbol. When the run-time linker is used, the imported symbol can be associated with a symbol in a different module. |

C

# Subroutine references for shmat and mmap services

This appendix contains the following subroutine references:

For detailed information about these routines, please refer to the *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions*.

# References for shmat services

This section includes the following subroutine references:

- “The ftok() subroutine” on page 448
- “The shmat() routine” on page 449
- “The shmctl() subroutine” on page 454
- “The shmget() routine” on page 457
- “The shmdt() subroutine” on page 460

## The ftok() subroutine

Generates a standard interprocess communication key.

### Library

Standard C library (libc.a)

### Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(Path, ID)
char *Path;
int ID;
```

### Description

The ftok() subroutine returns a key, based on the Path and ID parameters, to be used to obtain interprocess communication identifiers. The ftok() subroutine returns the same key for linked files if called with the same ID parameter. Different keys are returned for the same file if different ID parameters are used.

All interprocess communication facilities require you to supply a key to the msgget(), semget(), and shmget() subroutines in order to obtain interprocess communication identifiers. The ftok() subroutine provides one method for creating keys, but other methods are possible. For example, you can use the project ID as the most significant byte of the key, and use the remaining portion as a sequence number.

**Attention:**

- If the Path parameter of the ftok() subroutine names a file that has been removed while keys still refer to it, the ftok() subroutine returns an error. If that file is then re-created, the ftok() subroutine will probably return a key different from the original one.
- Each installation should define standards for forming keys. If standards are not adhered to, unrelated processes may interfere with each other's operation.
- The ftok() subroutine does not guarantee unique key generation. However, the occurrence of key duplication is very rare and mostly for across file systems.

### Parameters

**Path** Specifies the path name of an existing file that is accessible to the process.

**ID** Specifies a character that uniquely identifies a project.

### Return values

When successful, the ftok() subroutine returns a key that can be passed to the msgget(), semget(), or shmget() subroutines.

### Error codes

The ftok() subroutine returns the value (key_t)-1 if one or more of the following are true:

- The file named by the Path parameter does not exist.
- The file named by the Path parameter is not accessible to the process.
- The ID parameter has a value of 0.

## The shmat() routine

The shmat() subroutine attaches a shared memory segment or a mapped file to the current process's address space.

### Library

Standard C library (libc.a)

### Syntax

```
#include <sys/shm.h>
void *shmat(shm_mem_id, shm_mem_addr, shm_mem_flag)
```

```
int         shm_mem_id, shm_mem_flag;
const void *shared_mem_addr;
```

## Description

The shmat() subroutine attaches the shared memory segment or mapped file specified by the SharedMemoryID parameter (returned by the shmget() subroutine), or file descriptor specified by the SharedMemoryID parameter (returned by the openx() subroutine) to the address space of the calling process.

The following limits apply to shared memory:

- The maximum shared-memory segment size is:
  - 256 MB before AIX Version 4.3.1
  - 2 GB for AIX Version 4.3.1 through AIX 5L Version 5.1
  - 64 GB for 64-bit applications for AIX 5L Version 5.1 and later
- The minimum shared-memory segment size is 1 byte.
- The maximum number of shared memory IDs is 4096 for operating system releases before AIX Version 4.3.2 and 131072 for AIX Version 4.3.2 and following.

**Note:** The following applies to AIX Version 4.2.1 and later releases for 32-bit processes only.

An extended shmat capability is available. If an environment variable EXTSHM=ON is defined, then processes executing in that environment will be able to create and attach more than eleven shared memory segments.

The segments can be of size, from 1 byte to 2 GB, although for segments larger than 256 MB in size the environment variable EXTSHM=ON is ignored. The process can attach these segments into the address space for the size of the segment. Another segment could be attached at the end of the first one in the same 256 MB segment region. The address at which a process can attach is at page boundaries (a multiple of SHMLBA_EXTSHM bytes). For segments larger than 256 MB in size, the address at which a process can attach is at 256 MB boundaries, which is a multiple of SHMLBA bytes.

The segments can be of size from 1 byte to 256 MB. The process can attach these segments into the address space for the size of the segment. Another segment could be attached at the end of the first one in the same 256 MB segment region. The address at which a process can attach will be at page boundaries (a multiple of SHMLBA_EXTSHM bytes).

The maximum address space available for shared memory with or without the environment variable and for memory mapping is 2.75 GB. An additional segment register "0xE" is available so that the address space is from 0x30000000 to 0xE0000000. However, a 256 MB region starting from 0xD0000000 will be used by the shared libraries and is therefore unavailable for shared memory regions or mmapped regions.

There are some restrictions on the use of the extended shmat feature. These shared memory regions cannot be used as I/O buffers where the unpinning of the buffer occurs in an interrupt handler. The restrictions on the use are the same as that of mmap buffers.

The smaller region sizes are not supported for mapping files. Regardless of whether EXTSHM=ON or not, mapping a file will consume at least 256 MB of address space.

SHM_SIZE shmctl() is not supported for segments created with EXTSHM=ON.

A segment created with EXTSHM=ON can be attached by a process without EXTSHM=ON. This will consume a 256 MB area of the address space irrespective of the size of the shared memory region.

A segment created without EXTSHM=ON can be attached by a process with EXTSHM=ON. This will consume a 256 MB area of the address space irrespective of the size of the shared memory region.

The environment variable provides the option of executing an application either with the additional functionality of attaching more than 11 segments when EXTSHM=ON, or the higher-performance access to 11 or fewer segments when the environment variable is not set.

### Parameter

**shm_mem_id** Specifies an identifier for the shared memory segment.

**shm_mem_addr** Identifies the segment or file attached at the address specified by the shm_mem_addr parameter, as follows:

If the shm_mem_addr parameter is not equal to 0, and the SHM_RND flag is set in the shm_mem_flag parameter, the segment or file is attached at the next lower segment boundary. This address is given by (shm_mem_addr - (shm_mem_addr modulo SHMLBA_EXTSHM, if environment variable EXTSHM=ON or SHMLBA if not). SHMLBA specifies the low boundary address multiple of a segment.

If the shm_mem_addr parameter is not equal to 0 and the SHM_RND flag is not set in the shm_mem_flag parameter, the segment or file is attached at the address given by the shm_mem_addr parameter. If this address does not point to a SHMLBA_EXTSHM boundary if the environment variable EXTSHM=ON or SHMLBA boundary if not, the shmat() subroutine returns the value -1 and sets the errno global variable to the EINVAL error code. SHMLBA specifies the low boundary address multiple of a segment.

**shm_mem_flag** Specify several options. Its value is either 0 or is constructed by logically ORing one or more of the values in Table C-1.

*Table C-1 Values for the third parameter of shmat()*

| Value | Description |
|---|---|
| SHM_MAP | Maps a file onto the address space instead of a shared memory segment. The shm_mem_id parameter must specify an open file descriptor in this case. |
| SHM_RDONLY | Specifies read-only mode instead of the default read-write mode. |
| SHM_RND | Rounds the address given by the shm_mem_addr parameter to the next lower segment boundary, if necessary. |

The shmat() subroutine makes a shared memory segment addressable by the current process. The segment is attached for reading if the SHM_RDONLY flag is set and the current process has read permission. If the SHM_RDONLY flag is not set and the current process has both read and write permission, it is attached for reading and writing.

If the SHM_MAP flag is set, file mapping takes place. In this case, the shmat subroutine maps the file open on the file descriptor specified by the SharedMemoryID onto a segment. The file must be a regular file. The segment is then mapped into the address space of the process. A file of any size can be mapped if there is enough space in the user address space.

When file mapping is requested, the SharedMemoryFlag parameter specifies how the file should be mapped. If the SHM_RDONLY flag is set, the file is mapped read-only. To map read-write, the file must have been opened for writing.

All processes that map the same file read-only or read-write map to the same segment. This segment remains mapped until the last process mapping the file closes it.

A mapped file opened with the O_DEFER update has a deferred update. That is, changes to the shared segment do not affect the contents of the file resident in the file system until an fsync subroutine is issued to the file descriptor for which the mapping was requested. Setting the SHM_COPY flag changes the file to the deferred state. The file remains in this state until all processes close it. The SHM_COPY flag is provided only for compatibility with Version 2 of the operating system. New programs should use the O_DEFER open flag.

A file descriptor can be used to map the corresponding file only once. To map a file several times requires multiple file descriptors.

When a file is mapped onto a segment, the file is referenced by accessing the segment. The memory paging system automatically takes care of the physical I/O. References beyond the end of the file cause the file to be extended in page-sized increments. The file cannot be extended beyond the next segment boundary.

## Return values

When successful, the segment start address of the attached shared memory segment or mapped file is returned. Otherwise, the shared memory segment is not attached, the errno global variable is set to indicate the error, and a value of -1 is returned.

## Error codes

The shmat() subroutine is unsuccessful and the shared memory segment or mapped file is not attached if one or more of the following are true:

**EACCES** The calling process is denied permission for the specified operation.

**EAGAIN** The file to be mapped has enforced locking enabled, and the file is currently locked.

**EBADF** A file descriptor to map does not refer to an open regular file.

**EEXIST** The file to be mapped has already been mapped.

**EINVAL** The SHM_RDONLY and SHM_COPY flags are both set.

**EINVAL** The shm_mem_id parameter is not a valid shared memory identifier.

**EINVAL** The shm_mem_addr parameter is not equal to 0, and the value of (shm_mem_addr - (shm_mem_addr modulo SHMLBA_EXTSHM if the environment variable EXTSHM=ON or SHMLBA if not) points outside the address space of the process.

**EINVAL** The shm_mem_addr parameter is not equal to 0, the SHM_RND flag is not set in the SharedMemoryFlag parameter, and the shm_mem_addr parameter points to a location outside of the address space of the process.

**EMFILE** The number of shared memory segments attached to the calling process exceeds the system-imposed limit.

**ENOMEM** The available data space in memory is not large enough to hold the shared memory segment. ENOMEM is always returned if a 32-bit process tries to attach a shared memory segment larger than 2 GB.

**ENOMEM** The available data space in memory is not large enough to hold the mapped file data structure.

**ENOMEM** The requested address and length crosses a segment boundary. This is not supported when the environment variable EXTSHM=ON.

## The shmctl() subroutine

Controls shared memory operations.

### Library

Standard C library (libc.a)

### Syntax

```
#include <sys/shm.h>
int shmctl (SharedMemoryID, Command, Buffer)
int SharedMemoryID, Command;
struct shmid_ds * Buffer;
```

### Description

The shmctl() subroutine performs a variety of shared-memory control operations as specified by the Command parameter.

The following limits apply to shared memory:

- The maximum shared-memory segment size is:
  - 256 MB before AIX Version 4.3.1
  - 2 GB for AIX Version 4.3.1 through AIX 5L Version 5.1
  - 64 GB for 64-bit applications for AIX 5L Version 5.1 and later
- The minimum shared-memory segment size is 1 byte.

- The maximum number of shared memory IDs is 4096 for operating system releases before AIX Version 4.3.2 and 131072 for AIX Version 4.3.2 and following.

## Parameters

**SharedMemoryID** Specifies an identifier returned by the shmget subroutine.

**Buffer** Indicates a pointer to the shmid_ds structure. The shmid_ds structure is defined in the sys/shm.h file.

**Command** The commands listed in Table C-2 are available.

*Table C-2 Values for the third parameter of shmctl()*

| Value | Description |
|---|---|
| IPC_STAT | Obtains status information about the shared memory segment identified by the SharedMemoryID parameter. This information is stored in the area pointed to by the Buffer parameter. The calling process must have read permission to run this command. |
| IPC_ SET | Sets the user and group IDs of the owner as well as the access permissions for the shared memory segment identified by the SharedMemoryID parameter. This command sets the following fields:<br>`shm_perm.uid  /* owning user ID       */`<br>`shm_perm.gid  /* owning group ID      */`<br>`shm_perm.mode /* permission bits only */`<br>You must have an effective user ID equal to root or to the value of the shm_perm.cuid or shm_perm.uid field in the shmid_ds data structure identified by the SharedMemoryID parameter. |
| IPC_RMID | Removes the shared memory identifier specified by the SharedMemoryID parameter from the system and erases the shared memory segment and data structure associated with it. This command is only executed by a process that has an effective user ID equal either to that of superuser or to the value of the shm_perm.uid or shm_perm.cuid field in the data structure identified by the SharedMemoryID parameter. |

| Value | Description |
|---|---|
| SHM_SIZE | Sets the size of the shared memory segment to the value specified by the shm_segsz field of the structure specified by the Buffer parameter. This value can be larger or smaller than the current size. The limit is the maximum shared-memory segment size. This command is only executed by a process that has an effective user ID equal either to that of a process with the appropriate privileges or to the value of the shm_perm.uid or shm_perm.cuid field in the data structure identified by the SharedMemoryID parameter. This command is not supported for regions created with the environment variable EXTSHM=ON. This results in a return value of -1 with errno set to EINVAL. Attempting to use the SHM_SIZE on a shared memory region larger than 256 MB or attempting to increase the size of a shared memory region larger than 256 MB results in a return value of -1 with errno set to EINVAL. |

## Return values

When completed successfully, the shmctl() subroutine returns a value of 0. Otherwise, it returns a value of -1 and the errno global variable is set to indicate the error.

## Error codes

The shmctl() subroutine is unsuccessful if one or more of the following are true:

**EACCES** The Command parameter is equal to the IPC_STAT value and read permission is denied to the calling process.

**EFAULT** The Buffer parameter points to a location outside the allocated address space of the process.

**EINVAL** The SharedMemoryID parameter is not a valid shared memory identifier.

**EINVAL** The Command parameter is not a valid command.

**EINVAL** The Command parameter is equal to the SHM_SIZE value and the value of the shm_segsz field of the structure specified by the Buffer parameter is not valid.

**EINVAL** The Command parameter is equal to the SHM_SIZE value and the shared memory region was created with the environment variable EXTSHM=ON.

**ENOMEM** The Command parameter is equal to the SHM_SIZE value, and the attempt to change the segment size is unsuccessful because the system does not have enough memory.

**EOVERFLOW** The Command parameter is IPC_STAT and the size of the shared memory region is greater than or equal to 4 GB. This only happens with 32-bit programs.

**EPERM** The Command parameter is equal to the IPC_RMID or SHM_SIZE value, and the effective user ID of the calling process is not equal to the value of the shm_perm.uid or shm_perm.cuid field in the data structure identified by the SharedMemoryID parameter. The effective user ID of the calling process is not the root user ID.

## The shmget() routine

The shmget() subroutine returns the shared memory identifier associated with the specified key parameter.

### Library

Standard C library (libc.a)

### Syntax

```
#include <sys/shm.h>
int shmget(Key, Size, Shm_mem_flag)
key_t       Key;
size_t      Size;
int         Shm_mem_flag;
```

### Description

The shmget() subroutine returns the shared memory identifier associated with the specified Key parameter.

The following limits apply to shared memory:

- The maximum shared-memory segment size is:
  - 256 MB before AIX Version 4.3.1
  - 2 GB for AIX Version 4.3.1 through AIX 5L Version 5.1
  - 64 GB for 64-bit applications for AIX 5L Version 5.1 and later
- The minimum shared-memory segment size is 1 byte.
- The maximum number of shared memory IDs is 4096 for operating system releases before AIX Version 4.3.2 and 131072 for AIX Version 4.3.2 and following.

## Parameters

**Key** Specify either the IPC_PRIVATE value or an IPC key constructed by ftok() routine.

**Size** Specify the number of bytes of shared memory required.

**Shm_mem_flag** Constructed by logically ORing one or more of the values in Table C-3.

*Table C-3 Values for the third parameter of shmget()*

| Value | Description |
|---|---|
| IPC_CREATE | Creates the data structure if it does not already exist. |
| IPC_EXCL | Cause the shmget() subroutine to be unsuccessful if the IPC_CREATE flag is also set, and the data structure already exists. |
| SHM_LGPAGE | Attempts to create the region so it can be mapped through hardware-supported, large-page mechanisms, if enabled. This is purely advisory. For the system to consider this flag, it must be used in conjunction with the SHM_PIN flag and enabled with the **vmtune** command (-L to reserve memory for the region (which requires a reboot) and -S to enable SHM_PIN). To successfully get large-pages, the user requesting large-page shared memory must have CAP_BYPASS_RAC_VMM capability. This has no effect on shared memory regions created with the EXTSHM=ON environment variable. |
| SHM_PIN | Attempts to pin the shared memory region if enabled. This is purely advisory. For the system to consider this flag, the system must be enabled with the **vmtune** command. This has no effect on shared memory regions created with the EXTSHM=ON environment variable. |
| S_IRUSR | Permits the process that owns the data structure to read it. |
| S_IWUSR | Permits the process that owns the data structure to modify it. |
| S_IRGRP | Permits the group associated with the data structure to read it. |
| S_IWGRP | Permits the group associated with the data structure to modify it. |
| S_IROTH | Permits others to read the data structure. |
| S_IWOTH | Permits others to modify the data structure. |
| Values that begin with S_I prefix are defined in the /usr/include/sys/mode.h file and are a subset of the access permissions that apply to files. | |

A shared memory identifier, its associated data structure, and a shared memory segment equal in number of bytes to the value of the Size parameter are created for the Key parameter if one of the following is true:

- The Key parameter is equal to the IPC_PRIVATE value.
- The Key parameter does not already have a shared memory identifier associated with it, and the IPC_CREAT flag is set in the SharedMemoryFlag parameter.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- The shm_perm.cuid and shm_perm.uid fields are set to the effective user ID of the calling process.
- The shm_perm.cgid and shm_perm.gid fields are set to the effective group ID of the calling process.
- The low-order 9 bits of the shm_perm.mode field are set to the low-order 9 bits of the SharedMemoryFlag parameter.
- The shm_segsz field is set to the value of the Size parameter.
- The shm_lpid, shm_nattch, shm_atime, and shm_dtime fields are set to 0.
- The shm_ctime field is set to the current time.

Once created, a shared memory segment is deleted only when the system reboots, by issuing the **ipcrm** command, or by using the following shmctl subroutine:

```
if (shmctl(id, IPC_RMID, 0) == -1)
    perror("error in closing segment"), exit (1);
```

## Return values

Upon successful completion, a shared memory identifier is returned. Otherwise, the shmget() subroutine returns a value of -1 and sets the errno global variable to indicate the error.

## Error codes

The shmget() subroutine is unsuccessful if one or more of the following are true:

**EACCES** A shared memory identifier exists for the Key parameter, but operation permission, as specified by the low-order 9 bits of the SharedMemoryFlag parameter, is not granted.

**EEXIST** A shared memory identifier exists for the Key parameter, and both the IPC_CREAT and IPC_EXCL flags are set in the SharedMemoryFlag parameter.

**EINVAL** A shared memory identifier does not exist and the Size parameter is less than the system-imposed minimum or greater than the system-imposed maximum.

**EINVAL** A shared memory identifier exists for the Key parameter, but the size of the segment associated with it is less than the Size parameter, and the Size parameter is not equal to 0.

**ENOENT** A shared memory identifier does not exist for the Key parameter, and the IPC_CREAT flag is not set in the SharedMemoryFlag parameter.

**ENOMEM** A shared memory identifier and associated shared memory segment are to be created, but the amount of available physical memory is not sufficient to meet the request.

**ENOSPC** A shared memory identifier will be created, but the system-imposed maximum of shared memory identifiers allowed will be exceeded.

## The shmdt() subroutine

Detaches a shared memory segment.

### Library

Standard C library (libc.a)

### Syntax

```
#include <sys/shm.h>
int shmdt (SharedMemoryAddress)
const void * SharedMemoryAddress;
```

### Description

The shmdt() subroutine detaches, from the data segment of the calling process, the shared memory segment located at the address specified by the SharedMemoryAddress parameter.

Mapped file segments are automatically detached when the mapped file is closed. However, you can use the shmdt() subroutine to explicitly release the segment register used to map a file. Shared memory segments must be explicitly detached with the shmdt() subroutine.

If the file was mapped for writing, the shmdt() subroutine updates the mtime and ctime time stamps.

The following limits apply to shared memory:

- The maximum shared-memory segment size is:
  - 256 MB before AIX Version 4.3.1
  - 2 GB for AIX Version 4.3.1 through AIX 5L Version 5.1
  - 64 GB for 64-bit applications for AIX 5L Version 5.1 and later
- The minimum shared-memory segment size is 1 byte.
- The maximum number of shared memory IDs is 4096 for operating system releases before AIX Version 4.3.2 and 131072 for AIX Version 4.3.2 and following.

### Parameters

**SharedMemoryAddress** Specifies the data segment start address of a shared memory segment.

### Return values

When successful, the shmdt subroutine returns a value of 0. Otherwise, the shared memory segment at the address specified by the SharedMemoryAddress parameter is not detached, a value of 1 is returned, and the errno global variable is set to indicate the error.

### Error codes

The shmdt() subroutine is unsuccessful if the following condition is true:

**EINVAL** The value of the SharedMemoryAddress parameter is not the data-segment start address of a shared memory segment.

# References for mmap services

This section includes the following subroutine references:

- “The mmap() subroutine” on page 461
- “The mprotect() subroutine” on page 467
- “The msync() subroutine” on page 469
- “The munmap() subroutine” on page 471

## The mmap() subroutine

Maps a file-system object into virtual memory.

## Library

Standard C library (libc.a)

## Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
void *mmap(addr, len, prot, flags, filedes, offset)
void *      addr;
size_t      len;
int         prot, flags, filedes;
off_t       offset;
```

## Description

The mmap() subroutine creates a new mapped file or anonymous memory region by establishing a mapping between a process-address space and a file-system object. Care needs to be taken when using the mmap() subroutine if the program attempts to map itself. If the page containing executing instructions is currently referenced as data through an mmap mapping, the program will hang. Use the -H4096 binder option, and that will put the executable text on page boundaries. Then reset the file that contains the executable material, and view via an mmap mapping.

A region created by the mmap() subroutine cannot be used as the buffer for read or write operations that involve a device. Similarly, an mmap region cannot be used as the buffer for operations that require either a pin or xmattach operation on the buffer.

Modifications to a file-system object are seen consistently, whether accessed from a mapped file region or from the read or write subroutine.

Child processes inherit all mapped regions from the parent process when the fork subroutine is called. The child process also inherits the same sharing and protection attributes for these mapped regions. A successful call to any exec subroutine will unmap all mapped regions created with the mmap() subroutine.

The mmap64() subroutine is identical to the mmap subroutine() except that the starting offset for the file mapping is specified as a 64-bit value. This permits file mappings which start beyond OFF_MAX.

In the large file enabled programming environment, mmap() is redefined to be mmap64().

If the application has requested SPEC1170 compliant behavior, then the st_atime field of the mapped file is marked for update upon successful completion of the mmap() call.

If the application has requested SPEC1170 compliant behavior, then the st_ctime and st_mtime fields of a file that is mapped with MAP_SHARED and PROT_WRITE are marked for update at the next call to msync() subroutine or munmap() subroutine if the file has been modified.

> **Note:** A file-system object should not be simultaneously mapped using both the mmap() and shmat() subroutines. Unexpected results may occur when references are made beyond the end of the object.

## Parameters

**addr**

Specifies the starting address of the memory region to be mapped. When the MAP_FIXED flag is specified, this address must be a multiple of the page size returned by the sysconf() subroutine using the _SC_PAGE_SIZE value. A region is never placed at address zero, or at an address where it would overlap an existing region.

**len**

Specifies the length, in bytes, of the memory region to be mapped. The system performs mapping operations over whole page only. If the len parameter is not a multiple of the page size, the system will include, in any mapping operation, the address range between the end of the region and the end of the page containing the end of the region.

**prot**

Specifies the access permission for the mapped region. The /usr/include/sys/mman.h file defines the following access options:

| | |
|---|---|
| **PROT_READ** | Region can be read. |
| **PROT_WRITE** | Region can be written. |
| **PROT_EXEC** | Region can be executed. |
| **PROT_NONE** | Region cannot be accessed. |

The prot parameter can be the PROT_NONE flag, or any combination of the other values logically ORed together. If the PROT_NONE flag is not specified, access permissions may be granted to the region in addition to those explicitly requested. However, write access will not be granted unless the PROT_WRITE flag is specified.

**Note:** The operating system generates a SIGSEGV signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the PROT_WRITE flag is not specified and a program attempts a write access, a SIGSEGV signal results.

If the region is a mapped file that was mapped with the MAP_SHARED flag, the mmap() subroutine grants read or execute access permission only if the file descriptor used to map the file was opened for reading. It grants write access permission only if the file descriptor was opened for writing.

If the region is a mapped file that was mapped with the MAP_PRIVATE flag, the mmap() subroutine grants read, write, or execute access permission only if the file descriptor used to map the file was opened for reading. If the region is an anonymous memory region, the mmap subroutine grants all requested access permissions.

**filedes** Specifies the file descriptor of the file-system object to be mapped. If the MAP_ANONYMOUS flag is set, the filedes parameter must be -1. After the successful completion of the mmap() subroutine, the file specified by the filedes parameter may be closed without executing the mapped region or the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, which prevents the file data from been deallocated.

**Note:** The mmap() subroutine supports the mapping of regular files only. An mmap() call that specifies a file descriptor for a special file fails, returning the ENODEV error. An example of a file descriptor for a special file is one that might be used for mapping either I/O or device memory.

**offset** Specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by the sysconf() routine using the _SC_PAGE_SIZE value.

**flags** Specify attributes of the mapped region. Values for the flags parameter are constructed by a bitwise-inclusive ORing of values listed in Table C-4 on page 465, which lists symbolic names defined in the /usr/include/sys/mman.h file.

*Table C-4 Values for the sixth parameter of mmap()*

| Value | Description |
|---|---|
| MAP_FILE | Specifies the creation of a new mapped file region by mapping the file associated with the filedes file descriptor. The mapped region can extend beyond the end of the file, both at the time when the mmap() subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the mmap() subroutine, or if a file was later truncated. However, references to whole pages following the end of the file result in the delivery of a SIGBUS signal. Only one of the MAP_FILE and MAP_ANONYMOUS flags must be specified with the mmap() subroutine. |
| MAP_ANONYMOUS | Specifies the creation of a new, anonymous memory region that is initialized to all zeros. This memory region can be shared only with the descendants of the current process. When using this flag, the filedes parameter must be -1. Only one of the MAP_FILE and MAP_ANONYMOUS flags must be specified with the mmap() subroutine. |
| MAP_VARIABLE | Specifies that the system select an address for the new memory region if the new memory region cannot be mapped at the address specified by the addr parameter, or if the addr parameter is NULL. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified with the mmap() subroutine. |
| MAP_FIXED | Specifies that the mapped region be placed exactly at the address specified by the addr parameter. If the application has requested SPEC1170 complaint behavior and the mmap() request is successful, the mapping replaces any previous mappings for the process's pages in the specified range. If the application has not requested SEPC170 compliant behavior and a previous mapping exists in the range, then the request fails. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified with the mmap() subroutine. |

| Value | Description |
|---|---|
| MAP_SHARED | When the MAP_SHARED flag is set, modifications to the mapped memory region will be visible to other processes that have mapped the same region using this flag. If the region is a mapped file region, modifications to the region will be written to the file.<br>You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the mmap() subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you request SPEC1170compliant behavior. In this case, you must choose either MAP_SHARED or MAP_PRIVATE. |
| MAP_PRIVATE | When the MAP_PRIVATE flag is set, modifications to the mapped region by the calling process are not visible to other processes that have mapped the same region. If the region is a mapped file region, modifications to the region are not written to the file.<br>If this flag is specified, the initial write reference to an object page creates a private copy of that page and redirects the mapping to the copy. Until then, modifications to the page by processes that have mapped the same region with the MAP_SHARED flag are visible.<br>You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the mmap() subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you request SPEC1170compliant behavior. In this case, you must choose either MAP_SHARED or MAP_PRIVATE. |

## Return values

If successful, the mmap() subroutine returns the address at which the mapping was placed. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

## Error codes

Under the following conditions, the mmap subroutine fails and sets the errno global variable to:

**EACCES** The file referred to by the fildes parameter is not open for read access, or the file is not open for write access and the PROT_WRITE flag was specified for a MAP_SHARED mapping operation, or the file to be mapped has enforced locking enabled and the file is currently locked.

**EBADF** The fildes parameter is not a valid file descriptor, or the MAP_ANONYMOUS flag was set and the fildes parameter is not -1.

**EFBIG** The mapping requested extends beyond the maximum file size associated with fildes.

**EINVAL** The flags or prot parameter is invalid, or the addr parameter or off parameter is not a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter.

**EINVAL** The application has requested SPEC1170 compliant behavior and the value of flags is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).

**EMFILE** The application has requested SPEC1170 compliant behavior and the number of mapped regions would exceed an implementation-dependent limit (per process or per system).

**ENODEV** The fildes parameter refers to an object that cannot be mapped, such as a terminal.

**ENOMEM** There is not enough address space to map len bytes or the application has not requested Single UNIX Specification Version 2 compliant behavior, and the MAP_FIXED flag was set and part of the address-space range (addr, addr+len) is already allocated.

**ENXIO** The addresses specified by the range (off, off+len) are invalid for the fildes parameter.

**EOVERFLOW** The mapping requested extends beyond the offset maximum for the file description associated with fildes.

## The mprotect() subroutine

Modifies access protections for memory mapping.

### Library

Standard C library (libc.a)

## Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int mprotect ( addr,  len,  prot)
void *addr;
size_t len;
int prot;
```

## Description

The mprotect() subroutine modifies the access protection of a mapped file region or anonymous memory region created by the mmap() subroutine. The behavior of this function is unspecified if the mapping was not established by a call to the mmap() subroutine.

## Parameters

**addr** Specifies the address of the region to be modified. Must be a multiple of the page size returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter.

**len** Specifies the length, in bytes, of the region to be modified. If the len parameter is not a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the Name parameter, the length of the region will be rounded off to the next multiple of the page size.

**prot** Specifies the new access permissions for the mapped region. Legitimate values for the prot parameter are the same as those permitted for the mmap (mmap or mmap64()) subroutine, as follows:

| | |
|---|---|
| **PROT_READ** | Region can be read. |
| **PROT_WRITE** | Region can be written. |
| **PROT_EXEC** | Region can be executed. |
| **PROT_NONE** | Region cannot be accessed. |

## Return values

When successful, the mprotect() subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

**Note:** The return value for mprotect() is 0 if it fails, because the region given was not created by mmap() unless XPG 1170 behavior is requested by setting the environment variable XPG_SUS_ENV to ON.

### Error codes

**Attention:** If the mprotect() subroutine is unsuccessful because of a condition other than that specified by the EINVAL error code, the access protection for some pages in the (addr, addr + len) range may have been changed.

If the mprotect() subroutine is unsuccessful, the errno global variable may be set to one of the following values:

**EACCES** The prot parameter specifies a protection that conflicts with the access permission set for the underlying file.

**EINVAL** The prot parameter is not valid, or the addr parameter is not a multiple of the page size as returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter.

**ENOMEM** The application has requested Single UNIX Specification Version 2 compliant behavior, and addresses in the range are invalid for the address space of the process or specify one or more pages which are not mapped.

## The msync() subroutine

Synchronizes a mapped file.

### Library

Standard C library (libc.a).

### Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
int msync ( addr,  len,  flags)
void *addr;
size_t len;
int flags;
```

## Description

The msync() subroutine controls the caching operations of a mapped file region. Use the msync() subroutine to transfer modified pages in the region to the underlying file storage device.

If the application has requested Single UNIX Specification Version 2 compliant behavior then the st_ctime and st_mtime fields of the mapped file are marked for update upon successful completion of the msync() subroutine call if the file has been modified.

## Parameters

**addr** Specifies the address of the region to be synchronized. Must be a multiple of the page size returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter.

**len** Specifies the length, in bytes, of the region to be synchronized. If the len parameter is not a multiple of the page size returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter, the length of the region is rounded up to the next multiple of the page size.

**flags** Specifies one or more of the symbolic constants listed in Table C-5 that determine the way caching operations are performed.

*Table C-5 The third parameter of msync()*

| Value | Description |
|---|---|
| MS_SYNC | Specifies synchronous cache flush. The msync subroutine does not return until the system completes all I/O operations.<br>This flag is invalid when the MAP_PRIVATE flag is used with the mmap() subroutine. MAP_PRIVATE is the default privacy setting. When the MS_SYNC and MAP_PRIVATE flags both are used, the msync() subroutine returns an errno value of EINVAL. |
| MS_ASYNC | Specifies an asynchronous cache flush. The msync() subroutine returns after the system schedules all I/O operations.<br>This flag is invalid when the MAP_PRIVATE flag is used with the mmap() subroutine. MAP_PRIVATE is the default privacy setting. When the MS_SYNC and MAP_PRIVATE flags both are used, the msync() subroutine returns an errno value of EINVAL. |
| MS_INVALIDATE | Specifies that the msync() subroutine invalidates all cached copies of the pages. New copies of the pages must then be obtained from the file system the next time they are referenced. |

### Return values

When successful, the msync() subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

### Error codes

If the msync() subroutine is unsuccessful, the errno global variable is set to one of the following values:

**EIO** An I/O error occurred while reading from or writing to the file system.

**ENOMEM** The range specified by (addr, addr + len) is invalid for a process' address space, or the range specifies one or more unmapped pages.

**EINVAL** The addr argument is not a multiple of the page size, as returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter, or the flags parameter is invalid. The address of the region is within the process' inheritable address space.

## The munmap() subroutine

Unmaps a mapped region.

### Library

Standard C library (libc.a)

### Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
int munmap ( addr,  len)
void *addr;
size_t len;
```

### Description

The munmap() subroutine unmaps a mapped file region or anonymous memory region. The munmap() subroutine unmaps regions created from calls to the mmap() subroutine only.

If an address lies in a region that is unmapped by the munmap subroutine and that region is not subsequently mapped again, any reference to that address will result in the delivery of a SIGSEGV signal to the process.

## Parameters

**addr** Specifies the address of the region to be unmapped. Must be a multiple of the page size returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter.

**len** Specifies the length, in bytes, of the region to be unmapped. If the len parameter is not a multiple of the page size returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter, the length of the region is rounded up to the next multiple of the page size.

## Return values

When successful, the munmap() subroutine returns 0. Otherwise, it returns -1 and sets the errno global variable to indicate the error.

## Error codes

If the munmap() subroutine is unsuccessful, the errno global variable is set to the following value:

**EINVAL** The addr parameter is not a multiple of the page size, as returned by the sysconf() subroutine using the _SC_PAGE_SIZE value for the Name parameter.

**EINVAL** The application has requested Single UNIX Specification Version 2 compliant behavior and the len argument is 0.

# D

# Subroutine references for POSIX threads

This appendix provides Pthread subroutine references supported on AIX 5L Version 5.2 using the categories defined by the following sections:

- “Subroutines defined in the POSIX thread standard” on page 474
- “Subroutines defined in the UNIX 98 Specification” on page 479
- “Extensions to POSIX thread” on page 481

In each category, Pthread subroutines are described by routine names and short descriptions. For complete information about Pthread subroutines, please refer to *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions*.

In addition to the Pthread subroutines (starting from pthread_), AIX provides a set of subroutines starting from pthdb_, which are called as Pthread *debug library* subroutines, in order to offer additional functions over the POSIX thread standard. For further information about the Pthread debug library, please refer to the “Parallel Programming“ section of *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*.

# Subroutines defined in the POSIX thread standard

We have categorized the Pthread subroutines defined in the POSIX thread standard, which are supported on AIX, into the following:

| | |
|---|---|
| **Thread management sub-routines** | See Table D-1. |
| **Execution scheduling sub-routines** | See Table D-2 on page 476. |
| **Synchronization sub-routines** | See Table D-3 on page 477. |
| **Thread-specific data sub-routines** | See Table D-4 on page 479. |

*Table D-1 Thread management sub-routines*

| Name | Description |
|---|---|
| pthread_attr_init() | Initializes a thread attributes object. |
| pthread_attr_destroy() | Destroys a thread attributes object. |
| pthread_attr_setdetachstate() | Sets the detachstate attribute of a thread attributes object. This attribute determines if a thread created with this thread attributes object is in a detached state or not. |
| pthread_attr_getdetachstate() | Gets the detach state attribute from a thread attributes object. |
| pthread_attr_setstackaddr() | Sets the value of the stackaddr attribute of a thread attributes object. This attribute specifies the stack address of a thread created with this attributes object. |
| pthread_attr_getstackaddr() | Gets the stackaddr attribute from a thread attributes object. |
| pthread_attr_setstacksize() | Sets the value of the stacksize attribute of a thread attributes object. This attribute specifies the minimum stack size, in bytes, of a thread created with this thread attributes object. |
| pthread_attr_getstacksize() | Gets the stacksize attribute from a thread attributes object. |
| pthread_testcancel() | Creates a cancellation point in the calling thread. |
| pthread_setcancelstate() | Atomically sets the calling thread's cancelability state to the indicated state and returns the previous cancelability state at a specified location reference. |
| pthread_setcanceltype() | Atomically sets the calling thread's cancelability type to the indicated type and returns the previous cancelability type at a specified location reference. |

| Name | Description |
|---|---|
| pthread_create() | Creates a new thread and initializes its attributes using the thread attributes object specified, or standard values instead, if the NULL pointer is specified. After thread creation, a thread attributes object can be reused to create another thread, or deleted. |
| pthread_exit() | Terminates the calling thread safely, and stores a termination status for any thread that may join the calling thread. |
| pthread_cancel() | Requests the cancellation of the specified thread. The action depends on the cancelability of the target thread. |
| pthread_kill() | Sends the specified signal to the specified thread. It acts with threads like the kill subroutine with single-threaded processes. |
| pthread_join() | Blocks the calling thread until the specified thread terminates. If the specified thread is in a detached state (non-joinable), an error is returned. |
| pthread_detach() | Used to indicate to the implementation that storage for the specified thread can be reclaimed when that thread terminates. |
| pthread_once() | Executes the specified routine exactly once in a process. The first call to this subroutine by any thread in the process executes the given routine, without parameters. Any subsequent call will have no effect. |
| pthread_self() | Returns the calling thread's ID. |
| pthread_equal() | Compares the two specified thread IDs. Returns zero if and only if the IDs are equal. |
| pthread_atfork() | Threads can fork processes. This routine registers fork cleanup handlers. Three handlers can be specified: prepare, parent, and child. The prepare handler is called before the processing of the fork subroutine commences. The parent handler is called after the processing of the fork subroutine completes in the parent process. The child handler is called after the processing of the fork subroutine completes in the child process. |

| Name | Description |
|---|---|
| pthread_cleanup_push() | Pushes the specified cancellation cleanup handler routine onto the calling thread's cancellation cleanup stack. |
| pthread_cleanup_pop() | Removes the routine at the top of the calling thread's cancellation cleanup stack and optionally invokes it (if execute is non-zero). |

*Table D-2 Execution scheduling sub-routines*

| Name | Description |
|---|---|
| pthread_attr_setschedparam() | Sets the value of the schedparam attribute of the specified thread attributes object. The given schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. |
| pthread_attr_getschedparam() | Gets the value of the schedparam attribute of the specified thread attributes object. |
| pthread_attr_setscope() | The contention scope can only be set before thread creation by setting the contention-attribute of a thread attributes object. The pthread_attr_setscope subroutine sets the attribute to the specified value. |
| pthread_attr_getscope() | Gets the contention-scope attribute of the specified thread attributes object. |
| pthread_attr_setinheritsched() | Sets the inheritsched attribute of the specified thread attributes object to a given value. |
| pthread_attr_getinheritsched() | Gets the inheritsched attribute of the specified thread attributes object. |
| pthread_attr_setschedpolicy() | Sets the schedpolicy attribute of the specified thread attributes object. |
| pthread_attr_getschedpolicy() | Gets the schedpolicy attribute of the specified thread attributes object. |

| Name | Description |
|---|---|
| pthread_setschedparam() | Dynamically sets the schedpolicy and schedparam attributes of the specified thread. The given schedpolicy attribute specifies the scheduling policy of the thread. The given schedparam attribute specifies the scheduling parameters. The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system. If the target thread has system contention-scope, the process must have root authority to set the scheduling policy to either SCHED_FIFO or SCHED_RR. |
| pthread_getschedparam() | Returns the current schedpolicy and schedparam attributes of the thread<br>thread. The schedpolicy attribute specifies the scheduling policy of a thread. |

*Table D-3 Synchronization sub-routines*

| Name | Description |
|---|---|
| pthread_mutexattr_init() | Initializes a mutex attributes object with the default value for all of the attributes defined by the implementation. |
| pthread_mutexattr_destroy() | Destroys a mutex attributes object; the object becomes, in effect, uninitialized. |
| pthread_mutexattr_setpshared() | Sets the process-shared attribute in a given initialized attributes object. |
| pthread_mutexattr_getpshared() | Obtains the value of the process-shared attribute from the given attributes object. |
| pthread_mutex_init() | Initializes the given mutex with attributes specified by a given attributes object. If the attributes object is NULL, the default mutex attributes are used. |
| pthread_mutex_destroy() | Destroys the specified mutex object; the mutex object becomes, in effect, uninitialized. |
| pthread_mutex_lock() | The specified mutex object is locked by calling. If the mutex is already locked, the calling thread blocks until the mutex becomes available. |

| Name | Description |
|---|---|
| pthread_mutex_trylock() | Identical to pthread_mutex_lock(), except that if the referenced mutex object is currently locked (by any thread, including the current thread), the call returns immediately. |
| pthread_mutex_unlock() | Releases the referenced mutex object. The manner in which a mutex is released is dependent upon the mutex's type attribute. |
| pthread_condattr_init() | Initializes a specified condition variable attributes object with the default value for all of the attributes defined by the implementation. |
| pthread_condattr_destroy() | Destroys a specified condition variable attributes object; the object becomes, in effect, uninitialized. |
| pthread_condattr_setpshared() | Sets the value of the pshared attribute of the specified condition attributes object. This attribute specifies the process sharing of the condition variable created with this attributes object. |
| pthread_condattr_getpshared() | Returns the value of the pshared attribute of the specified condition attribute object. This attribute specifies the process sharing of the condition variable created with this attributes object. |
| pthread_cond_init() | Initializes the given condition variable with attributes given by a condition attributes object. If that object is NULL, the default condition variable attributes are used. |
| pthread_cond_destroy() | Destroys the given condition variable; the object becomes, in effect, uninitialized. |
| pthread_cond_wait() | Blocks on a condition variable. Must be called with a specified mutex locked by the calling thread or undefined behavior will result. |
| pthread_cond_timedwait() | Same as pthread_cond_wait(), except that an error is returned if the specified absolute time passes (that is, system time equals or exceeds the specified absolute time) before the specified condition is signaled or broadcasted, or if the absolute time specified has already been passed at the time of the call. |
| pthread_cond_signal() | Unblocks one or more threads blocked on the specified condition. |

| Name | Description |
|---|---|
| pthread_cond_broadcast() | Unblocks all the blocked threads on the specified condition. |

*Table D-4 Thread-specific data sub-routines*

| Name | Description |
|---|---|
| pthread_key_create() | Creates a thread-specific data key. The key is shared among all threads within the process, but each thread has specific data associated with the key. The thread-specific data is a void pointer, initially set to NULL. An optional destructor routine can be specified. It will be called for each thread when it is terminated and detached, after the call to the cleanup routines, if the specific value is not NULL. |
| pthread_key_delete() | Deletes the given thread-specific data key previously created with the pthread_key_create() subroutine. The application must ensure that no thread-specific data is associated with the key. |
| pthread_setspecific() | Associates a thread-specific value with a key obtained through a previous call to pthread_key_create(). Different threads may bind different values to the same key. |
| pthread_getspecific() | Returns the value currently bound to the specified key on behalf of the calling thread. |

**Note:** Currently, AIX does not support the following sub-routines. Although the symbols are provided in the Pthread library, but calls to these routines always return with ENOSYS:

- pthread_mutexattr_setprioceiling()
- pthread_mutexattr_getprioceiling()
- pthread_mutexattr_setprotocol()
- pthread_mutexattr_getprotocol()
- pthread_mutex_setprioceiling()
- pthread_mutex_getprioceiling()

# Subroutines defined in the UNIX 98 Specification

We have categorized the Pthread subroutines defined in the UNIX 98 Specification, which are supported on AIX 5L Version 5.2, into the following:

| | |
|---|---|
| **Read-write lock sub-routines** | See Table D-5 on page 480. |
| **Additional POSIX threads sub-routines** | See Table D-6 on page 481. |

*Table D-5 Read-write lock sub-routines*

| Name | Description |
|---|---|
| pthread_rwlockattr_init() | Initializes the read-write specified lock with the attributes referenced by the given read-write lock attribute object. If that object is NULL, the default read-write lock attributes are used. |
| pthread_rwlockattr_destroy() | Destroys the specified read-write lock attribute object and releases any resources used by the lock. |
| pthread_rwlockattr_setpshared() | Sets the process-shared attribute in the given initialized read-write lock attributes object. |
| pthread_rwlockattr_getpshared() | Obtains the value of the process-shared attribute from the given initialized read-write lock attributes object. |
| pthread_rwlock_init() | Initializes the specified read-write lock with the attributes from a given read-write lock attributes object. If that object is NULL, the default read-write lock attributes are used. |
| pthread_rwlock_destroy() | Destroys the specified read-write lock object and releases any resources used by the lock. |
| pthread_rwlock_rdlock() | Applies a read lock to the given read-write lock. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. |
| pthread_rwlock_tryrdlock() | Applies a read lock as in the pthread_rwlock_rdlock() function with the exception that the function fails if any thread holds a write lock on the specified read-write lock or there are writers that sblocked the lock. |
| pthread_rwlock_wrlock() | Applies a write lock to the given read-write lock. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock. Otherwise, the thread blocks (that is, does not return from the pthread_rwlock_wrlock() call) until it can acquire the lock. |
| pthread_rwlock_trywrlock() | Applies a write lock like the pthread_rwlock_wrlock() function, with the exception that the function fails if any thread currently holds the specified read-write lock (for reading or writing). |
| pthread_rwlock_unlock() | Releases a lock held on the specified read-write lock object. |

*Table D-6 Additional POSIX threads sub-routines defined in UNIX 98*

| Name | Description |
|---|---|
| pthread_setconcurrency() | Allows an application to inform the threads implementation of its desired concurrency level. The actual level of concurrency provided by the implementation as a result of this function call is unspecified. |
| pthread_getconcurrency() | Returns the value set by a previous call to the pthread_setconcurrency() function. |
| pthread_attr_setguardsize() | Sets the guardsize attribute in a thread attribute object. The guardsize attribute controls the size of the guard area for the created thread's stack. The guardsize attribute provides protection against overflow of the thread's stack pointer. |
| pthread_attr_getguardsize() | Gets the guardsize attribute of a thread attributes object. |
| pthread_mutexattr_settype() | Sets the mutex type attribute of a mutex attributes object to a given type. |
| pthread_mutexattr_gettype() | Gets the type attribute of a given mutex attributes object. |
| pthread_suspend() | Suspends execution of specified thread. |
| pthread_continue() | Resumes execution of specified thread. |

# Extensions to POSIX thread

The POSIX thread standard leaves certain aspects of implementation defined, unspecified, or even undefined. Therefore, most UNIX operating system vendors provide several extensions in order to complement those areas in the POSIX thread standard.

On AIX, those extension subroutines are starting from pthread_ and ending in _np to signify that a library routine is non-portable and should not be used in code that will be ported to other UNIX-based systems (see Table D-7 on page 482).

*Table D-7 Non-portable thread routines in AIX*

| Routine | Routine (continued) |
|---|---|
| pthread_atfork_unregister_np()<br>pthread_attr_getsuspendstate_np()<br>pthread_attr_setstacksize_np()<br>pthread_attr_setsuspendstate_np()<br>pthread_cleanup_information_np()<br>pthread_cleanup_pop_np()<br>pthread_cleanup_push_np()<br>pthread_clear_exit_np()<br>pthread_continue_np()<br>pthread_continue_others_np()<br>pthread_delay_np()<br>pthread_get_expiration_np()<br>pthread_getrusage_np() | pthread_getthrds_np()<br>pthread_getunique_np()<br>pthread_join_np()<br>pthread_lock_global_np()<br>pthread_mutexattr_getkind_np()<br>pthread_mutexattr_setkind_np()<br>pthread_set_mutexattr_default_np()<br>pthread_setcancelstate_np()<br>pthread_signal_to_cancel_np()<br>pthread_suspend_np()<br>pthread_suspend_others_np()<br>pthread_test_exit_np()<br>pthread_unlock_global_np() |

Relevant information about these non-portable subroutines could be find in the following publications:

- *AIX Version 4.3 Differences Guide*, SG24-2014
- *AIX 5L Differences Guide Version 5.2 Edition,* SG24-5765

E

# Supported IBM SMP directives

This appendix explains IBM SMP directives supported by the following compiler products:

- C for AIX Version 6.0
- VisualAge C++ for AIX Version 6.0

When developing new applications, it is recommended that you use OpenMP directives explained in Chapter 8, “Introduction to POSIX threads” on page 275.

For further information about IBM directives, please refer to:

- *C for AIX Compiler Reference*, SC09-4960
- *VisualAge C++ for AIX Compiler Reference*, SC09-4959

# IBM SMP directives

IBM SMP directives for parallelization are based on the possibility of parallelizing *countable loops*. A loop is considered countable when the following rules can be applied:

- There is no branching into or outside of the loop.
- The incremental expression (incr_expr) is not within a critical section.

Table E-1 shows the C language control flow statements and the regular expressions that define when they can be treated as countable loops.

*Table E-1 Regular expressions for countable loops*

| C control flow statement keywords | Regular expression |
|---|---|
| `for` | `for ([iv]; exit_cond; incr_expr)`<br>`   statement`<br><br>`for ([iv]; exit_cond; [expr] {`<br>`   [declaration_list]`<br>`   [statement_list]`<br>`   incr_expr;`<br>`   [statement_list]`<br>`}` |
| `while` | `while (exit_cond) {`<br>`   [declaration_list]`<br>`   [statement_list]`<br>`   incr_expr;`<br>`   [statement_list]`<br>`}` |
| `do` | `do {`<br>`   [declaration_list]`<br>`   [statement_list]`<br>`   incr_expr;`<br>`   [statement_list]`<br>`} while (exit_cond)` |

Where:

| | |
|---|---|
| **exit_cond** | iv <= ub |
| | iv < ub |
| | iv >= ub |
| | iv > ub |
| **incr_expr** | ++iv |

iv++

--iv

i--

iv += incr

iv -= incr

iv = iv + incr

iv = incr + iv

iv = iv - incr

**iv** Iteration variable. The iteration variable is a signed integer that has either automatic or register storage class, does not have its address taken, and is not modified anywhere in the loop except in *incr_expr*.

**incr** Loop invariant signed integer expression. The value of the expression is known at compile-time and is not 0. *incr* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

**ub** Loop invariant signed integer expression. *ub* cannot reference extern or static variables, pointers or pointer expressions, function calls, or variables that have their address taken.

In general, a countable loop is automatically parallelized only if all of the following conditions are met:

- The order in which loop iterations start or end does not affect the results of the program.
- The loop does not contain I/O operations.
- Floating point reductions inside the loop are not affected by round-off error, unless the -qnostrict option is in effect.
- The -qnostrict_induction compiler option is in effect.
- The -qsmp compiler option is in effect without its omp sub option. The compiler must be invoked using a thread-safe compiler mode.

## The IBM SMP directives syntax

When using IBM SMP directives for explicitly defining parallel portions of code, use the following syntax:

```
#pragma ibm pragma_name_and_args
```

```
<countable for|while|do loop>
```

Pragma directives must appear immediately before the section of code to which they apply. For most parallel processing pragma directives, this section of code must be a countable loop, and the compiler will report an error if one is not found.

More than one parallel processing pragma directive can be applied to a countable loop. For example:

```
#pragma ibm independent_loop
#pragma ibm independent_calls
#pragma ibm schedule(static,5)
<countable for|while|do loop>
```

Some pragma directives are mutually-exclusive. If mutually-exclusive pragmas are specified for the same loop, the last pragma specified applies to the loop. In the example below, the parallel_loop pragma directive is applied to the loop, and the sequential_loop pragma directive is ignored:

```
#pragma ibm sequential_loop
#pragma ibm parallel_loop
```

Other pragmas, if specified repeatedly for a given loop, have an additive effect. For example:

```
#pragma ibm permutation (a,b)
#pragma ibm permutation (c)
```

is equivalent to:

```
#pragma ibm permutation (a,b,c)
```

Table E-2 shows all IBM pragma directives supported by the latest compilers.

*Table E-2 Supported IBM pragma directives*

| Pragma | Description |
|---|---|
| #pragma ibm critical | Instructs the compiler that the statement or statement block immediately following this pragma is a critical section. |
| #pragma ibm independent_calls | Asserts that specified function calls within the chosen loop have no loop-carried dependencies. |
| #pragma ibm independent_loop | Asserts that iterations of the chosen loop are independent, and that the loop can therefore be parallelized. |
| #pragma ibm iterations | Specifies the approximate number of loop iterations for the chosen loop. |

| Pragma | Description |
|---|---|
| #pragma ibm parallel_loop | Explicitly instructs the compiler to parallelize the chosen loop. |
| #pragma ibm permutation | Asserts that specified arrays in the chosen loop contain no repeated values. |
| #pragma ibm schedule | Specifies scheduling algorithms for parallel loop execution. |
| #pragma ibm sequential_loop | Explicitly instructs the compiler to execute the chosen loop sequentially. |

# F

# Sample compiler listing

This appendix includes a sample compiler listing, as explained in 6.2, “Diagnosing compile-time errors” on page 227.

# Compiler listing

The following is a sample compiler listing showing the various sections described in “Compiler listing” on page 228.

*Example: F-1 Sample compiler listing*

```
C for AIX Compiler Version 6.0.0.2 --- hello.c 02/11/03 13:31:29 (C)

>>>>> SOURCE SECTION <<<<<

        1 | #include <stdio.h>
        2 |
        3 | void main() {
        4 |         printf(“hello\n”);
        5 | }

>>>>> OPTIONS SECTION <<<<<

C for AIX Compiler Version 6.0.0.2 ---
***   Command Line Invocation ***
***   Options In Effect   ***

NOA                 NOAE                NOALLOCA            NOBROWSE
NOCOMPACT           NOCPLUSCMT          NODBCS              NODBXEXTRA
NODIGRAPH           DOLLAR              NOEXTCHK            NOFDPR
NOFULLPATH          NOFUNCSECT          NOG                 NOGRAPHICS
NOHEAPDEBUG         NOIDIRFIRST         NOIGNERRNO          NOINLGLUE
NOLARGEPAGE         NOLIBANSI           NOLINEDEBUG         LINEDIR
LIST                LISTOPT             LONGLONG            NOMACPSTR
NOMAKEDEP           NOMBCS              NOOFFSET            NOP
NOPASCAL            NOPDF1              NOPDF2              NOPHSINFO
PRINT               NOPROTO             NOREPORT            NORO
NOROCONST           NOSMALLSTACK        SOURCE              NOSTATSYM
STDINC              STRICT              STRICT_INDUCTION    NOSYNTAXONLY
NOTHREADED          NOTOCMERGE          UNWIND              UPCONV
NOWARN64            NOXCALL             NOXCOFF             NOXPH2

OPTIMIZE=0
INLINE THRESHOLD=20
AGGRCOPY=NOOVERLAP
ALIAS=NOANSI:NOTYPEPTR:NOALLPTRS:NOADDRTAKEN
ALIGN=POWER
ATTR
BITFIELDS=UNSIGNED
CHARS=UNSIGNED
DATAIMPORTED
ENUM=INT
FLAG=I:I
```

```
FLOAT=NOHSFLT:NORNDSNG:NOHSSNGLE:MAF:NORSQRT:NORRM:FOLD:NOSPNANS:NOFLTINT:NOEMU
LATE
FLTTRAP=NOOV:NOUND:NOZERO:NOINV:NOINEX:NOEN:NOIMP
NOGENPROTO
HALT=S
NOHOT
INFO=NOCLS:NOCMP:NOCND:NOCNS:NOCNV:NOCPY:NODCL:NOEFF:NOENU:NOEXT:NOGEN:NOGNR:NO
GOT:NOINI:NOLAN:NOOBS:NOORD:NOPAR:NOPOR:NOPPC:NOPPT:NOPRO:NOREA:NORET:NOTRD:NOT
RU:TRX:NOUND:NOUNI:NOUSE:NOVFT:NOPRIVATE:NOREDUCTION:NOC99
LANGLVL=EXTENDED:NOUCS
LONGDOUBLE=128
NOMAXERR
MAXMEM=8192
OS=AIX
PROCUNKNOWN
SHOWINC=NOSYS:NOUSR
NOSMP

SPILL=512
TBTABLE=DEFAULT
TUNE=DEFAULT
UNROLL=AUTO
XREF
YN (ROUND NEAR)
REACHABLE=setjmp
REACHABLE=_setjmp
REACHABLE=sigsetjmp
REACHABLE=_sigsetjmp



>>>>> ATTRIBUTE AND CROSS REFERENCE SECTION <<<<<

fhandle                           struct tag
                                  4-528.8$  4-528.16{  4-530.1}  4-531.16

fid                               struct tag
                                  4-521.8$  4-521.12{  4-524.1}  4-525.16

fileid                            struct tag
                                  4-500.8$  4-500.15{  4-505.1}  4-535.16

label_t                           struct tag
                                  8-48.16$  8-51.16   8-49.1{  8-67.1}

main                              extern function returning void
                                  0-3.6Y

printf                            extern function returning int
```

```
                                 1-263.17X  0-4.9Z

unique_id                        struct tag
                                 4-545.8$  4-545.18{  4-550.1}  4-551.16


>>>>> FILE TABLE SECTION <<<<<

                                      FILE CREATION        FROM
FILE NO   FILENAME                   DATE       TIME       FILE    LINE
     0    hello.c                    02/11/03   13:23:32
     1    /usr/include/stdio.h       09/13/02   10:34:34      0       1
     2    /usr/include/standards.h   09/13/02   10:27:52      1      43
     3    /usr/include/va_list.h     09/13/02   10:27:45      1     185
     4    /usr/include/sys/types.h   09/13/02   10:27:48      1     399
     5    /usr/include/sys/inttypes.h 09/13/02  10:27:52      4      55
     6    /usr/include/stdint.h      09/13/02   10:27:52      5      62
     7    /usr/include/standards.h   09/13/02   10:27:52      6      28
     8    /usr/include/sys/m_types.h  09/13/02  10:52:54      4     393
     9    /usr/include/sys/vm_types.h 09/13/02  10:53:54      8      40
    10    /usr/include/va_list.h     09/13/02   10:27:45      1     432
    11    /usr/include/sys/limits.h  09/13/02   10:28:31      1     466
    12    /usr/include/float.h       09/13/02   10:51:20     11     263


>>>>> COMPILATION EPILOGUE SECTION <<<<<

C for AIX Summary of Diagnosed Conditions

TOTAL   UNRECOVERABLE  SEVERE       ERROR     WARNING    INFORMATIONAL
               (U)       (S)         (E)        (W)          (I)
    0           0         0           0          0            0


Source records read.........................................    3591

1501-008  Compilation successful for file hello.c. Object file created.


>>>>> OBJECT SECTION, NO OPTIMIZATION <<<<<


 GPR's set/used:   ss-s ssss ssss s---  ---- ---- ---- ---s
 FPR's set/used:   ssss ssss ssss ss--  ---- ---- ---- ----
 CCR's set/used:   ss-- -sss

     | 000000                          PDEF     main
    3|                                 PROC
    0| 000000 mfspr    7C0802A6   1     LFLR     gr0=lr
    0| 000004 stw      93E1FFFC   0     ST4A     #stack(gr1,-4)=gr31
    0| 000008 stw      90010008   2     ST4A     #stack(gr1,8)=gr0
    0| 00000C stwu     9421FFC0   0     ST4U     gr1,#stack(gr1,-64)=gr1
    3| 000010 lwz      83E20004   1     L4A      gr31=.$STATIC(gr2,0)
    4| 000014 addi     387F0008   2     AI       gr3=gr31,8
    4| 000018 bl       4BFFFFE9   0     CALL
gr3=printf,1,gr3,printf”,gr1,cr[01567]”,gr0”,gr4”-gr12”,fp0”-fp13”
    4| 00001C ori      60000000   1
    5|                               CL.1:
    5| 000020 lwz      80010048   1     L4A      gr0=#stack(gr1,72)
    5| 000024 mtspr    7C0803A6   2     LLR      lr=gr0
    5| 000028 addi     38210040   1     AI       gr1=gr1,64
    5| 00002C lwz      83E1FFFC   0     L4A      gr31=#stack(gr1,-4)
    5| 000030 bclr     4E800020   2     BA       lr
     |              Tag Table
     | 000034       00000000 00002041 80010001 00000034 00046D61 696E
     |              Instruction count          13
     |              Straight-line exec time    13
```

# Abbreviations and acronyms

| | |
|---|---|
| **ANSI** | American National Standard Institute |
| **APAR** | Authorized Problem Analysis Report |
| **API** | Application Programming Interface |
| **ASCII** | American National Standard Code for Information Interchange |
| **BOS** | Base Operating System |
| **BSD** | Berkeley Software Distribution |
| **BSS** | Block Started by Symbol |
| **CD** | Compact Disk |
| **CDE** | Common Desktop Environment |
| **CD-ROM** | CD-Read Only Media |
| **CDT** | Central Daylight Time |
| **CPU** | Central Processing Unit |
| **CST** | Central Standard Time |
| **DCE** | Distributed Computer Environment |
| **DLL** | Dynamic Link Library |
| **DLPAR** | Dynamic Logical Partitioning |
| **DSA** | Dynamic Segment Allocation |
| **DVD** | Digital Versatile Disk |
| **DVD-R** | DVD-Recordable |
| **DVD-RAM** | DVD-Random Access Media |
| **DVD-ROM** | DVD-Read Only Media |
| **EB** | Exabyte |
| **EXTSHM** | Extended Mode Shared Memory Segments |
| **FC** | Feature Code |
| **FIFO** | First-in First-out |
| **FTSS** | Field Technical Support Specialist |
| **GB** | Gigabyte |
| **GID** | Group ID |
| **GMT** | Greenwich Mean Time |
| **GNU** | GNU is Not UNIX |
| **GPR** | General Purpose Register |
| **GUI** | Graphical User Interface |
| **HMC** | IBM Hardware Management Console for pSeries |
| **HPC** | High-Performance Computing |
| **HTML** | Hyper-Text Markup Language |
| **HTTP** | Hyper-Text Transfer Protocol |
| **I/O** | Input/Output |
| **IBM** | International Business Machines Corporation |
| **IDE** | Integrated Development Environment |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **ILP32** | integer/long/pointer 32-bit |
| **IOC** | IBM Open Class Library |
| **IPA** | Interprocedural Analysis |
| **IPC** | Inter-Process Communication |
| **ISBN** | International Standard Book Number |
| **ISO** | International Organization for Standardization |
| **ISV** | Independent Software Vendor |
| **ITSO** | International Technical Support Organization |
| **JFS** | Journaled File System |
| **JFS2** | Enhanced Journaled File System |

| | |
|---|---|
| **K&R** | The C programming language standard defined by Brian W. Kernighan and Dennis M. Ritchie |
| **KB** | Kilobyte |
| **LIFO** | Last-In First-Out |
| **LP64** | Long/Pointer 64-bit |
| **LPP** | License Program Product |
| **LUM** | License Use Management |
| **LWP** | Light-Weight Process |
| **MB** | Megabyte |
| **MP** | Multiprocessor |
| **NFS** | Network File System |
| **NIS** | Network Information Service |
| **OS** | Operating System |
| **PB** | Petabyte |
| **PCI** | Peripheral Component Interconnect |
| **PDF** | Portable Document Format or Profile-Directed Feedback |
| **PID** | Process ID |
| **POSIX** | Portable Operating System Interface |
| **POWER** | Performance Optimization with Enhanced RISC |
| **PPID** | Parent Process ID |
| **PTF** | Program Temporary Fix |
| **RAD** | Rapid Application Development |
| **RISC** | Reduced Instruction Set Computer |
| **RMC** | Resource Monitoring and Control |
| **RML** | Recommended Maintenance Level |
| **RSCT** | Reliable Scalable Cluster Technology |
| **SMIT** | System Management Interface Tool |
| **SMP** | Symmetric Multiple Processor |
| **STL** | Standard Template Library |
| **TB** | Terabyte |
| **TID** | Thread ID |
| **TLB** | Translation Look-Aside Buffer |
| **TOC** | Table of Contents |
| **TSD** | Thread-Specific Data |
| **TTY** | Teletypewriter |
| **UCS** | Universal Coded Character Set |
| **UID** | User ID |
| **UP** | Uniprocessor |
| **URL** | Universal Resource Locator |
| **VMM** | Virtual Memory Manager |
| **VP** | Virtual Processor |
| **WLM** | Work Load Manager |
| **XCOFF** | eXtended Common Object File Format |

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 500.

- *AIX 5L Differences Guide Version 5.2 Edition,* SG24-5765
- *AIX 5L Porting Guide*, SG24-6034
- *AIX Version 4.3 Differences Guide*, SG24-2014
- *The Complete Partitioning Guide for IBM @server pSeries Servers,* SG24-7039
- *IBM @server pSeries 670 and pSeries 690 System Handbook*, SG24-7040
- *Managing AIX Server Farms*, SG24-6606
- *The POWER4 Processor Introduction and Tuning Guide*, SG24-7041
- *A Practical Guide for Resource Monitoring and Control (RMC)*, SG24-6615

## AIX official publications

The following publications are contained in the AIX 5L for POWER V 5.2 Documentation CD, 5765-E62, which is shipped as a part of the AIX 5L Version 5.2 CD-ROM media set. These publications are also available on the following URL (click **AIX 5.2** after arriving at the Web page):

http://techsupport.services.ibm.com/server/library

- *AIX 5L Version 5.2 Files Reference*
- *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*
- *AIX 5L Version 5.2 Installation Guide and Reference*
- *AIX 5L Version 5.2 Installation in a Partitioned Environment*
- *AIX 5L Version 5.2 National Language Support Guide and Reference*
- *AIX 5L Version 5.2 Performance Management Guide*

- *AIX 5L Version 5.2 Reference Documentation: Commands Reference*
- *AIX 5L Version 5.2 System Management Guide: Communications and Networks*
- *AIX 5L Version 5.2 System Management Guide: Operating System and Devices*
- *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions*
- *AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems*
- *AIX 5L Version 5.2 Web-based System Manager Administration Guide*
- *IBM Reliable Scalable Cluster Technology for AIX 5L: Administration Guide*, SA22-7889
- *IBM Reliable Scalable Cluster Technology for AIX 5L: Messages*, GA22-7891
- *IBM Reliable Scalable Cluster Technology for AIX 5L: Technical Reference*, SA22-7890

## pSeries hardware related publications

The following pSeries hardware related publications are available by accessing this URL:

http://www.ibm.com/servers/eserver/pseries/library/hardware_docs/index.html

- *IBM Hardware Management Console for pSeries Installation and Operations Guide*, SA38-0590
- *PCI Adapter Placement References*, SA38-0538

## C for AIX official publications

The following publications are contained in the C for AIX Version 6.0 product CD-ROM media set as soft copy files. These publications are also available on the following URL (click **Product documentation** after arriving at the Web site):

http://www.ibm.com/software/ad/caix/support.html

- *C for AIX C/C++ Language Reference*, SC09-4958
- *C for AIX Compiler Reference*, SC09-4960
- *Getting Start with C for AIX Introduction and Installation Guide*, SC09-4961

## VisualAge C++ for AIX official publications

The following publications are contained in the VisualAge C++ for AIX Version 6.0 product CD-ROM media set as soft copy files. These publications are also available on the following URL (click **Library** after arriving at the Web site):

http://www.ibm.com/software/ad/vacpp/

- *Getting Started with VisualAge C++ for AIX Introduction, Installation, and Migration Guide*, SC09-4962
- *IBM Open Class Library Transition Guide*, SC09-4948
- *VisualAge C++ for AIX Compiler Reference*, SC09-4959
- *VisualAge C++ Professional for AIX C/C++ Language Reference*, SC09-4957
- *VisualAge C++ Professional for AIX Programming Tasks and Library Reference*, SC09-4963
- *VisualAge C++ Standard C++ Library Reference*, SC09-4949

## Other publications

The following publications are referenced during the development phase of this redbook:

- Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997, ISBN 0201633922
- Lewine, *POSIX Programmer's Guide: Writing Portable UNIX Programs*, O'Reilly & Associates, 1992, ISBN 0937175730
- Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992, ISBN 0201563177

# Referenced Web sites

These Web sites are also relevant as further information sources:

- AIX large page support white paper, found at:

  http://www.ibm.com/servers/aix/whitepapers/large_page.html

- AIX toolkit for Linux applications

  http://www.ibm.com/servers/aix/products/aixos/linux/download.html

- C for AIX

  http://www.ibm.com/software/ad/caix/

- Fix Delivery Center for AIX Version 5

  `http://techsupport.services.ibm.com/server/aix.fdc`
- IBM AIX Library

  `http://www.ibm.com/servers/aix/library`
- IBM @server pSeries Support

  `http://techsupport.services.ibm.com/server/support?view=pSeries`
- IBM PartnerWorld

  `http://www.ibm.com/partnerworld/developer`
- IBM Supported Products List

  `http://www.ibm.com/servers/aix/products/ibmsw/list`
- OpenMP API Web site

  `http://www.openmp.org`
- *Using License Use Management Runtime for AIX*

  `ftp://ftp.software.ibm.com/software/lum/aix/doc/V4.6.0/lumusg.pdf`
- VisualAge C++ for AIX

  `http://www.ibm.com/software/ad/vacpp/`
- VisualAge C++ Support

  `http://www.ibm.com/software/ad/vacpp/support.html`
- *Writing Multithreaded Applications For Aix, Part 1: Tutorial* white paper, found at:

  `http://www.ibm.com/servers/esdd/tutorials/multi_aix.html`

# How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

**ibm.com**/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

## IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

# Index

IBM

Redbooks

# Developing and Porting C and C++ Applications on AIX

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

# Developing and Porting C and C++ Applications on AIX

**Detailed explanations about 32- and 64-bit process models**

**Effective management of shared objects and libraries**

**Exploring parallel programming using OpenMP**

This IBM Redbook will help experienced UNIX application developers who are new to the AIX operating system. The book explains the many concepts in detail, including the following:

- Enhancements and new features provided by the latest C and C++ compilers for AIX
- Compiling and linking tasks required to effectively use and manage shared libraries and run-time linking
- Use of process heap and shared memory in the 32- and 64-bit user process models
- A new programming paradigm in a partitioned environment where resources can be dynamically changed
- Parallel programming using POSIX threads and OpenMP

The following chapters are also useful for system administrators who are responsible for the software problem determination and application software release level management on AIX systems:

Chapter 3, “Understanding user process models”
Chapter 7, “Debugging your applications”
Chapter 12, “Packaging your applications”

**INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION**

**BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**
**ibm.com**/redbooks

SG24-5674-01 ISBN 073842935X