Porting of Win32 API WaitFor to Solaris [tm]

By Nagendra Nagarajayya and Alka Gupta
Market Development Engineering

Introduction

This paper details how the NT based API `WaitForMultipleObjects()` was successfully ported to a Solaris[tm] operating environment. A thread-safe emulation based on the POSIX thread library and its basic synchronization primitives was used to accomplish this task.

Emulation of Win32 API WaitForMultipleObjects() Functionality Under The Solaris™ Operating Environment

This project was undertaken to assist an ISV with the port of a server application from NT to Solaris, particularly the port of the `WaitForMultipleObjects()` API, which is not straightforward since there is no concept of events and handles in Solaris. Our implementation of this API is generic and can be applied to any handle type on NT. This paper discusses the design issues we faced and could be applicable to other developers facing similar port issues. The Appendix includes the source code for `WaitforMultipleObject`s API implementation and is POSIX compliant.

WaitFor

The `WaitFor` method is the basic thread synchronization mechanism for the NT operating system, and its successful implementation is critical to any NT emulation effort.

`WaitFor` allows NT threads to synchronize on the setting of one or more varieties of NT objects. The thread issuing a `WaitFor` is then blocked until either a timeout occurs after a pre-specified time, or when one or all of the objects it is waiting for gets "set".

The `WaifFor` is not blocking if one of the NT objects is already set when the `WaifFor` is first issued.

WaitForSingleObject

The `WaitForSingleObject` function takes a handle object as the first parameter and will not return before the object that is referenced by the handle attains the signaled state or the timeout value that is specified as the second parameter expires. This is simple.

WaitForMultipleObjects

The `WaitForMultipleObjects` call is similar, except that its main parameter (the second) is an array of Windows NT handle objects. The first parameter passed to it specifies the number of handles in this array. The third parameter is FALSE if waiting for ANY object is desired and TRUE if the function is to return only when all objects in the array have been set; the wait for ALL objects is desired.

Note that the handle array passed to `WaitForMultipleObject`s can be composed of an arbitrary mix of Windows NT handle objects. This includes thread handles, handles to processes, mutexes, semaphores, and events. Each handle type is a subclass of the base handle class.

A thread can wait for multiple objects to be set in one of two ways:

1. ALL

The issuing thread will unblock after ALL objects specified in the call have been set. This is fairly simple to emulate. Each object is waited for in turn until all have been set or until timeout occurs. At that point this function returns.

2. ANY

The issuing thread will unblock after ANY one specified object in the call has been set.

`WaitForMultipleObjects` for the case ANY is the most difficult of all Win32 routines to emulate since UNIX® (thus Solaris) does not support anything like it. The problem is that the issuing thread has to actually be blocked on a single Solaris variable, yet the setting of any one of a series of NT objects has to result in the signaling of that variable.

Following is a list of requirements for the solution:

1. More than one thread could be waiting at any one time on the same handle.
2. Spurious signals may not be sent to threads.
   Note: For example, if thread 1 is waiting on two handles, A and B, and thread 2 is waiting on two handles A and C, and handle B is set, a signal may not be sent to thread 2 just because it is waiting on handle A common to threads 1 and 2.
3. No polling.

The solution detailed in this paper makes use of the ***subscription model***, that is, the interested thread subscribes itself to the interested wait object and goes to sleep until **signaled**.

The handle is a composite object containing a list of `mutex_cond` objects. Each `mutex_cond` object contains a condition variable and a mutex.

The handle structure will look like this:

```
Handle{
      List mutex_cond;
      Boolean is_signaled;

}

mutex_cond{
      Mutex mutex;
      Cond_var cond_var;
}
```

Instead of having two lists, one of mutex and the other of `cond_var`, the `mutex_cond` object is created so that the action of adding the pair of mutex and condition variables to the handle is atomic.

The model proposes that each thread that issues or executes on a `WaitFor` call on any one or more handles subscribes itself to those handles by adding a pair of `mutex/cond_var` to that handle list and then waits on that `mutex/cond_var` pair. When a handle is set, all threads waiting on that handle are signaled. Thus, if two threads wait on a handle, there will be two elements in the list of that handle. And when the handle is set, both of the threads are signaled.

ALL or logical AND

In the case of ALL or logical AND, the thread needs to wait for all handles to be set, or for timeout, before exiting from the `WaitFor` call. One `mutex_cond` object containing a condition variable and a mutex is created for every handle and added to that handle's list.

The code will look like this:

```
Boolean WaitForMultipleObjects(int Count, void **Handle, boolean fWaitall, long
msTimeOut)
{
 mutex_cond_t mutex_cond[Count];

 for(i=0; i < Count; i++){
   initialize the mutex and condition variable
   Create a new mutex_cond object mutex_cond[i]
   Add the mutex_cond object to the list of the Handle[i]
 }
```

```
   for(i=0; i < Count; i++){
   Lock mutex_cond[i]->mutex
   while (! Handle[i]->is_signaled) {
     Wait until timeout on mutex_cond[i]->cond_var
   }
   UnLock mutex_cond[i]->mutex
 }

   for(i=0; i < Count; i++){
   Delete mutex_cond[i] from Handle List
 }
}
```

ANY or logical OR

In the case of ANY or logical OR, the preceding example illustrates that instead of the thread creating a new `mutex_cond` object for each handle, there is a single common `mutex_cond` object for all the handles specified, and the thread wait on this common `mutex_cond` object for all the handles. In other words, the thread subscribes to each handle with the common `mutex_cond` object, and waits on it. So, if any handle gets set, the thread returns from the `WaitFor` method as expected. This prevents the need for any kind of polling, which can be highly CPU intensive.

The code will look like this:

```
Boolean WaitForMultipleObjects(int Count, void **Handle, boolean fWaitall, long
TimeOut)
{
 Create one common mutex_cond object: common_mutex_cond
 Initialize the mutex and condition variable

 for(i=0; i < Count; i++){
   Add the common_mutex_cond object to the list of the Handle[i]
   }

   Lock common_mutex_cond->mutex

   for(i=0; i < Count; i++){
   Check for each Handle if it has been set, if set exit
   }

   if (none of the handles have been set) {
   Wait until timeout on common_mutex_cond->cond_var, common_mutex_cond->mutex
   }

 UnLock common_mutex_cond->mutex

 for(i=0; i < Count; i++){
   Delete common_mutex_cond element from List of Handle[i]
 }
}
```

SIGNALING

When a handle is set, a signal is sent to all threads waiting on that handle.

```
void HandleSet(void *Handle ){
 Iterate through the Handle List {
   Lock Handle->List_element->mutex
 }

 Iterate through the Handle List {
   Signal Handle->List_element->cond_var
 }

 Iterate through the Handle List {
   Unlock Handle->List_element->mutex
 }
}
```

The complete code for the case of an event handle and a test driver is available in the Appendix. The design is generic and this code can be modified to use a mix of one or more handle types. The code uses the POSIX thread library.

The same subscription model can be used for the case of `WaitForSingleObject` if the Count parameter in the case of `WaitForMultipleObjects` is set to 1.

For example:

```
Boolean WaitForSingleObjects(void **Handle, long msTimeOut)
{
     WaitForMultipleObjects(1, **Handle, AND, msTimeOut);
}
```

LIMITATION

1. Synchronized objects states are not modified. In other words, they are not non-signaled.

2. Named handles require the handles to reside in shared space, and could be acted upon by other processes. This requires that the mutex and condition variables be created with system scope (the threads are already created with system scope). The shared space could be `mmaped` on startup and shared by different processes.

3. Handles can be closed multiple times and need to be freed when the reference count goes to zero.

4. There is no check for duplicate handles.

5. Handle type is not checked.

CONCLUSIONS

The port of `WaitForMultipleObjects()` Win32 API has been successfully done and can be used by other groups for porting Win32 applications from NT to the Solaris operating environment. Much more than that, this implementation of the handle structure forms the basic building block, making use of the Solaris synchronization variables, on top of which complex synchronization objects can be built.

REFERENCES

1. Kleinman, R. (1995) Emulation of Server-side Win32 Functionality under Solaris.
2. Ruediger R. Asche (1994) Compound Win32 Synchronization Objects

APPENDIX

The source code includes a file called `WaitForMultipleObjects.c`, a file called `include_type.h` and a file called

`driver.c`. `driver.c` is a test driver that can be used to test the ported API. `include_type.h` defines the `mutex_cond` and event objects. `WaitForMultipleObjects.c` is the source code for the port of the API. This code uses the POSIX library. (Source code `tar` file and README.)

The source code can be compiled on Solaris 8 as:

```
cc-o driver driver.c
WaitForMultipleObjects.c-DREENTRANT-D_POSIX_C_SOURCE=199506L-lrt-lpthread


/*************************************************************************/


/* File include_type.h */


#define AND 1
#define OR 0
#define      NULLP ((void *)0)


/* Type of the mutex_cond object */

typedef struct mutex_cond{
        pthread_mutex_t i_mutex;
        pthread_cond_t i_cv;
}mutex_cond_t;

/* Type of the List element object */

typedef struct List_element{
        struct List_element *next;
        struct List_element *prev;
        struct mutex_cond *i_mutex_cond;
}List_element_t;

/* Type of the List object, Any other implementation
 * of List can be used
 */

typedef struct List{
        struct List_element *start_list;
        struct List_element *end_list;
        /* This mutex is used to protect the List critical section */
        pthread_mutex_t i_mutex;

}List_t;

/* Type of the Event Handle object */

typedef struct win32_event {
        boolean_t i_signalled;
        struct List *i_list;
}solaris_win32_event_t;
```

```
/* End include_type.h */
/*****************************************************************************/
/* File WaitForMultipleObjects.c */

#include < stdio.h >
#include < stdlib.h >
#include < pthread.h >
#include < time.h >
#include < errno.h >
#include "include_type.h"

void AddToList();
void DeleteFromList();

/* This method gets the actual system time at which the
 * timeout must occur, based on the timeout parameter
 */

static boolean_t ClockGetTime(struct timespec *tp, long msTimeOut)
{
      long seconds, nseconds;        /* for time calculations */

      if (clock_gettime(CLOCK_REALTIME, tp) != 0)
            return(_B_FALSE);


   /* since timeOutPeriod is in milliseconds and tv_nsec is nanoseconds,
    * and both are long integers, we'd rather extract the second and
    * nanoseconds from the timeOutPeriod and add them separately, making
    * sure we're not over 1,000,000,000 nanoseconds.
    */
       seconds = msTimeOut / 1000;
       nseconds = (msTimeOut - seconds*1000)*1000000;

       tp->tv_sec += seconds;
       tp->tv_nsec += nseconds;
       if (tp->tv_nsec >= 1000000000)
       {
               tp->tv_nsec -= 1000000000;
               tp->tv_sec += 1;
       }

       return(_B_TRUE);
}


/* Method for setting the Event object. Signal is sent to all
 * threads waiting on that event to be set.
 */

void
EventSet(void *Event)
{
      solaris_win32_event_t *event = (solaris_win32_event_t *)Event;
```

```
	struct List_element *start;
	struct List_element *p;

	event->i_signalled = _B_TRUE;

	/*
	 * At this point iterate through the list
	 */

	start = event->i_list->start_list;

	if ( start == NULLP )
	{
		fprintf(stderr, " Nothing to iterate...no elements in the list!\n");
		return;
	}
	/*
	 * All the mutexes on the Event Handle List are first
	 * locked, then all condition variables are signalled
	 * and then all mutexes are unlocked. This is done to
	 * prevent any race condition.
	 */

	p = start;
	do {
		pthread_mutex_lock( &p->i_mutex_cond->i_mutex );
		p = p->next;

	} while ( p != NULLP );

	p = start;
	do {
		pthread_cond_signal( &p->i_mutex_cond->i_cv );
		 p = p->next;

	} while ( p != NULLP );

	p = start;
	do {
		pthread_mutex_unlock( &p->i_mutex_cond->i_mutex );
		p = p->next;

	} while ( p != NULLP );

} /* EventSet */

boolean_t
WaitForMultipleObjects(int Count, void **Events, boolean_t fWaitall, long msTimeOut)
{
	solaris_win32_event_t *event;
	boolean_t Result;
	int i;
	struct timespec to_time; /* time value */
	struct timespec delta_time; /* time value */
	mutex_cond_t *p, *common_mutex_cond;
```

```
     List_element_t *q;
     /*
      * Local array of mutex_cond objects and List is maintained
      * so that a search through the Event list is not required
      * later when these objects are needed.
      */

      mutex_cond_t **mutex_cond;
      List_element_t **list_element;
      mutex_cond = (mutex_cond_t **)malloc(sizeof(mutex_cond_t *)* Count);
      list_element = (List_element_t **)malloc(sizeof(List_element_t *)* Count);

      for (i = 0; i < Count; i++)
      {
            event = (solaris_win32_event_t*)Events[i];
            if (event == NULL) return(_B_FALSE);
      }

      if (msTimeOut != -1)
      {
              if (ClockGetTime(&to_time, msTimeOut) == _B_FALSE)
              {
                      return(_B_FALSE);
              }
      }


      switch (fWaitall){

      case AND:
              Result = _B_TRUE;
              /*
               * mutex_cond objects are created and initialized, List
               * object elements are created, and mutex_cond object is
               * is added to each List element.
               */

             for(i=0; i < Count; i++){
                     solaris_win32_event_t *event = (solaris_win32_event_t
*)Events[i];
                     p = (mutex_cond_t *) malloc(sizeof(mutex_cond_t));
                     pthread_mutex_init ( &p->i_mutex, NULL);
                     pthread_cond_init ( &p->i_cv, NULL);
                     q = (List_element_t *) malloc(sizeof(List_element_t));
                     mutex_cond[i] = p;
                     q->i_mutex_cond = p;
                     list_element[i]=q;

                      /*
                       * At this point we need to add the malloced
                       * object to the list
                       */

                      AddToList( event, q);
              }
```

```
                /* Wait on each event to be set or timedout */
                for(i=0; i < Count; i++){
                        solaris_win32_event_t *event = (solaris_win32_event_t
*)Events[i];
                        pthread_mutex_lock(&(mutex_cond[i]->i_mutex));
                        while(!event->i_signalled){
                                if (msTimeOut != -1) {
                                        if
(pthread_cond_timedwait(&mutex_cond[i]->i_cv,
                                        &mutex_cond[i]->i_mutex, &to_time) ==
ETIMEDOUT)

                                        {
                                                Result = _B_FALSE;
                                                break;
                                        }
                                }
                                else
                                {
                                 pthread_cond_wait(&mutex_cond[i]->i_cv,
                                 &mutex_cond[i]->i_mutex);
                                }
                        }
                        pthread_mutex_unlock(&mutex_cond[i]->i_mutex);
                        if (Result == _B_FALSE)
                        {
                            break;
                        }
                }
                /*
                 * Delete all the List elements created by this thread for
                 * each event.
                 */

                for(i=0; i < Count; i++){
                        solaris_win32_event_t *event = (solaris_win32_event_t
*)Events[i];
                        DeleteFromList(event, list_element[i]);
                }
                break;

        case OR:
                Result = _B_FALSE;
                 /*
                  * A common mutex_cond object is created and initialized, List
                  * object elements are created, and same common mutex_cond
                  * object is added to each List element.
                  */

                common_mutex_cond = (mutex_cond_t *)malloc(sizeof(mutex_cond_t));
                pthread_mutex_init(&common_mutex_cond->i_mutex, NULL);
                pthread_cond_init(&common_mutex_cond->i_cv, NULL);
                for(i=0; i < Count; i++){
                        solaris_win32_event_t *event = (solaris_win32_event_t
*)Events[i];
```

```
                        q = (List_element_t *) malloc(sizeof(List_element_t));
                        q->i_mutex_cond = common_mutex_cond;
                        list_element[i]=q;
                /*
                 * At this point the malloced List element is
                 * added to the Event Handle.
                 */
                        AddToList(event, q);
               }
                /*
                 * It is first tested if any of the event has already
                 * been set, if yes, the function returns as expected.
                 * Or else, wait on the common mutex for any event to
                 * be set or time out.
                 */

                pthread_mutex_lock(&common_mutex_cond->i_mutex);
                for(i=0; i < Count; i++){
                        solaris_win32_event_t *event = (solaris_win32_event_t
*)Events[i];
                        if(event->i_signalled){
                                Result = _B_TRUE;
                                break;
                        }
                }
                if(Result == _B_FALSE){
                        if (msTimeOut != -1) {
                                if
(pthread_cond_timedwait(&common_mutex_cond->i_cv,
                                    &common_mutex_cond->i_mutex,
                                    &to_time) != ETIMEDOUT)
                                {
                                        Result = _B_TRUE;
                                }
                        }
                        else
                        {
                                pthread_cond_wait(&common_mutex_cond->i_cv,
                                &common_mutex_cond->i_mutex);
                                Result = _B_TRUE;
                        }
                }
                pthread_mutex_unlock(&common_mutex_cond->i_mutex);
                /*
                 * Delete all the List elements created by this thread for
                 * each event.
                 */
                for(i=0; i < Count; i++){
                        solaris_win32_event_t *event = (solaris_win32_event_t
*)Events[i];
                        DeleteFromList(event, list_element[i]);
                }
                break;
        default:
                break;
```

```
          }
          free(mutex_cond);
          free(list_element);
          return Result;

} /* WaitForMultipleObjects */

void
AddToList( solaris_win32_event_t *event, List_element_t *new_element)
{
        struct List_element *start, *end;
        struct List_element *p = (struct List_element*)new_element ;

        pthread_mutex_lock(&event->i_list->i_mutex);

        start = event->i_list->start_list;
        end = event->i_list->end_list;

        if ( start == NULLP ) {
                /* First element of the list */
                start = p;
                p->next = NULLP;
                p->prev = NULLP;
                end = p;
                event->i_list->start_list = p;
                event->i_list->end_list = p;
        }
        else {

               /*
                * Already some elements in the list...
                * add this element at the end
                */
               end->next = p;
               p->prev = end;
               p->next = NULLP;
               end = p;
               event->i_list->end_list = p;
       }
       pthread_mutex_unlock(&event->i_list->i_mutex);

} /* AddToList */

/*
 * Delete element from the list of mutex_cond_t...
 */
void
DeleteFromList( solaris_win32_event_t *event, List_element_t *element )
{
        struct List_element *start, *end, *q;
        struct List_element *p = (struct List_element *)element ;

        pthread_mutex_lock(&event->i_list->i_mutex);

        start = event->i_list->start_list;
```

```
        end = event->i_list->end_list;

        if ( start == p && end == p )
        {
                /* only element in the list */
                start = end = NULLP;
                /*
                 * Call the appropriate function for freeing
                 * the memory
                 */
                free(p);
                event->i_list->start_list = NULLP;
                event->i_list->end_list = NULLP;
        }
        else if ( start == p ) {
                q = p->next;
                start = q;
                q->prev = NULLP;
                free(p);
                event->i_list->start_list = start;
        }
        else if ( end == p ) {
                end = p->prev;
                end->next = NULLP;
                free(p);
                event->i_list->end_list = end;
        }
        else {
                /*
                 * Element is somehwre in the middle of the list
                 */
                (p->prev)->next = p->next;
                (p->next)->prev = p->prev;
                free(p);
        }
        pthread_mutex_unlock(&event->i_list->i_mutex);
} /* DeleteFromList */

/* End WaitForMultipleObjects.c */
/*************************************************************************/
/*
 * File driver.c
 * This is the driver file for testing the ported API
 */

#include
#include
#include
#include "include_type.h"


#define NUM_THREADS 5 /* Number of threads created */
#define EVENT_COUNT 3 /* Actual number of events passed to the WaitFor object */

void  *or_wait_thread(void *);
```

```
void  *and_wait_thread(void *);
void  *signal_thread(void *);
solaris_win32_event_t *event[EVENT_COUNT];

pthread_t thread[NUM_THREADS];
pthread_attr_t attr;

main() {
        int i;
        for(i=0; i < EVENT_COUNT; i++){
                event[i]= (solaris_win32_event_t
*)malloc(sizeof(solaris_win32_event_t));
                event[i]->i_signalled = _B_FALSE;
                event[i]->i_list = (struct List *)malloc(sizeof(struct List *));
                pthread_mutex_init(&(event[i]->i_list->i_mutex), NULL);
                event[i]->i_list->start_list = NULLP;
                event[i]->i_list->end_list = NULLP;

        }
        pthread_attr_init(&attr);
        pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
        for(i=0; i < NUM_THREADS-3; i++){
                pthread_create(&thread[i], &attr, or_wait_thread, (void *)NULL);
        }
        pthread_create(&thread[NUM_THREADS-3], &attr, signal_thread, (void *)NULL);
        pthread_create(&thread[NUM_THREADS-2], &attr, and_wait_thread, (void *)NULL);
        pthread_create(&thread[NUM_THREADS-1], &attr, and_wait_thread, (void *)NULL);


        pthread_exit(0);

}

void *or_wait_thread(void *arg){
        WaitForMultipleObjects(EVENT_COUNT, event, OR, 100000);
}


void *and_wait_thread(void *arg){
        WaitForMultipleObjects(EVENT_COUNT, event, AND, 100000);
}

void *signal_thread(void *arg){
        sleep(5);
        EventSet(event[2]);
        sleep(5);
        EventSet(event[1]);
        sleep(5);
        EventSet(event[0]);
}

/* End of driver.c */
/*************************************************************/
```