# Windows Kernel Internals
# NT Registry Implementation

David B. Probert, Ph.D.
Windows Kernel Development
Microsoft Corporation

# Outline

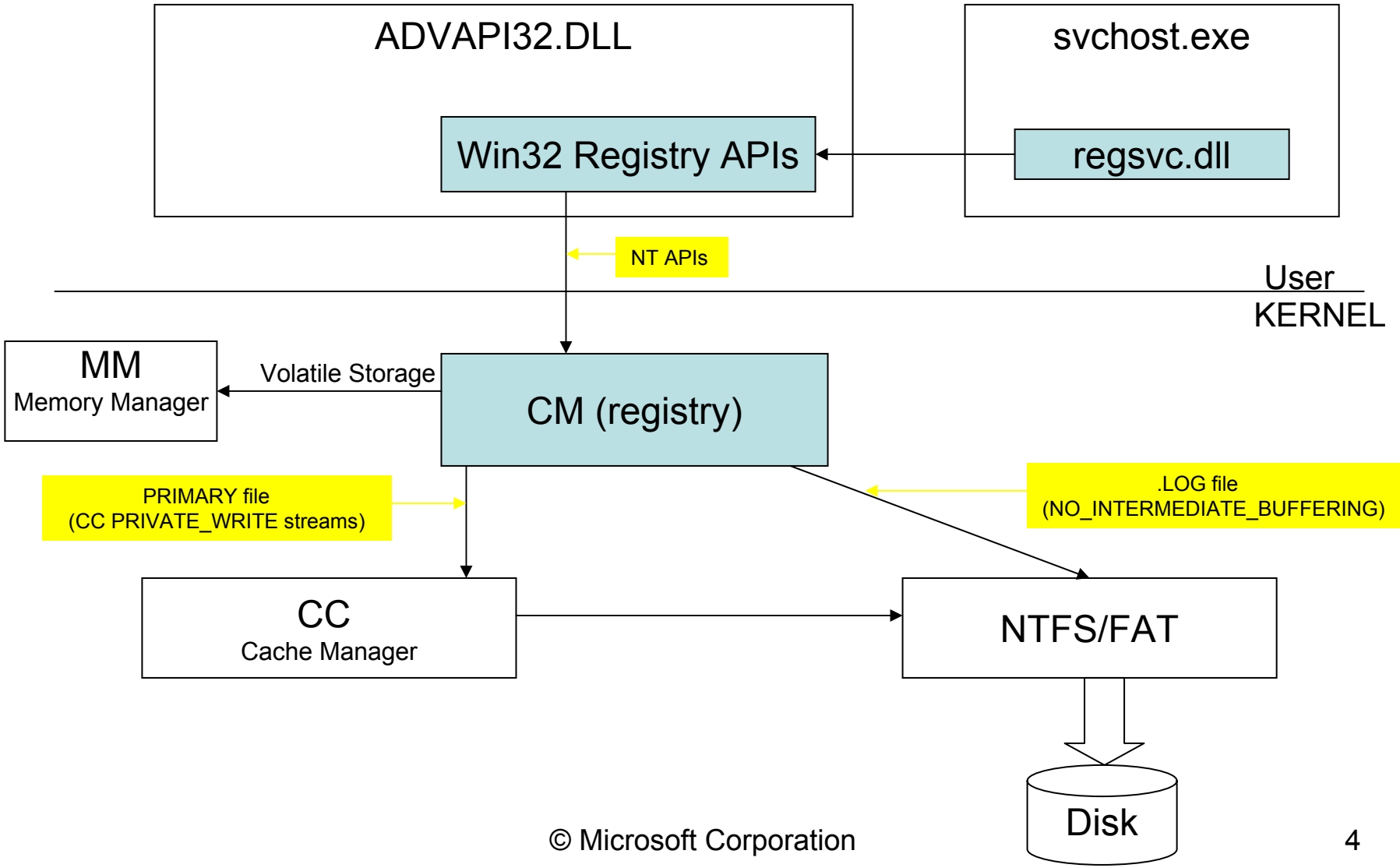
- High level overview
- System whereabouts
- Native registry APIs
- Implementation Details
- I/O
- Mounting a Hive
- Life Span
- Backup/Restore
- Limits

# High Level Overview

- Logical:
  - Registry = “a FS within a file”
  - Keys ←→ directories
  - Values ←→ files

- Physical:
  - Registry = collection of Hives
  - Hive = collection of Bins
  - Bin = collection of Cells
  - Cell = unit of allocation (contains raw data)

# Whereabouts



# NT Registry APIs: **Key Ops**

| | |
|---|---|
| **NtCreateKey** (kname) | open a new or existing key |
| **NtDeleteKey** (khandle) | mark key to delete at last handle close |
| **NtEnumerateKey** (khandle, i) | return the name/info of subkey[i] of key |
| **NtQueryKey** (khandle) | get info about a key |
| **NtSetInformationKey** (khandle, info) | set info on key |
| **NtRenameKey** (khandle, string) | change the name of key |
| **NtFlushKey** (khandle) | flush changes associated with key to disk |
| **NtNotifyChangeKey** (khandle, bsubtree) | notify caller of changes to a key/subtree |
| **NtNotifyChangeMultipleKeys** (knames[], bsubtree) | Multi-key version of NtNotifyChangeKey |

# NT Registry APIs: Value Ops

| | |
|---|---|
| **NtEnumerateValueKey** (khandle, i) | return the name/info of value[i] of key |
| **NtQueryValueKey** (khandle, vname) | get value (data & type) |
| **NtQueryMultipleValueKey** (khandle, vnames[]) | get multiple values |
| **NtSetValueKey** (khandle, vname, value) | set a value |
| **NtDeleteValueKey** (khandle, vname) | delete a value belonging to a key |

## Misc Ops

| | |
|---|---|
| **NtQueryOpenSubKeys** (kpath) | get count of open khandles under kpath |
| **NtCompactKeys** (count, khandles[]) | optimize access to khandles[] |

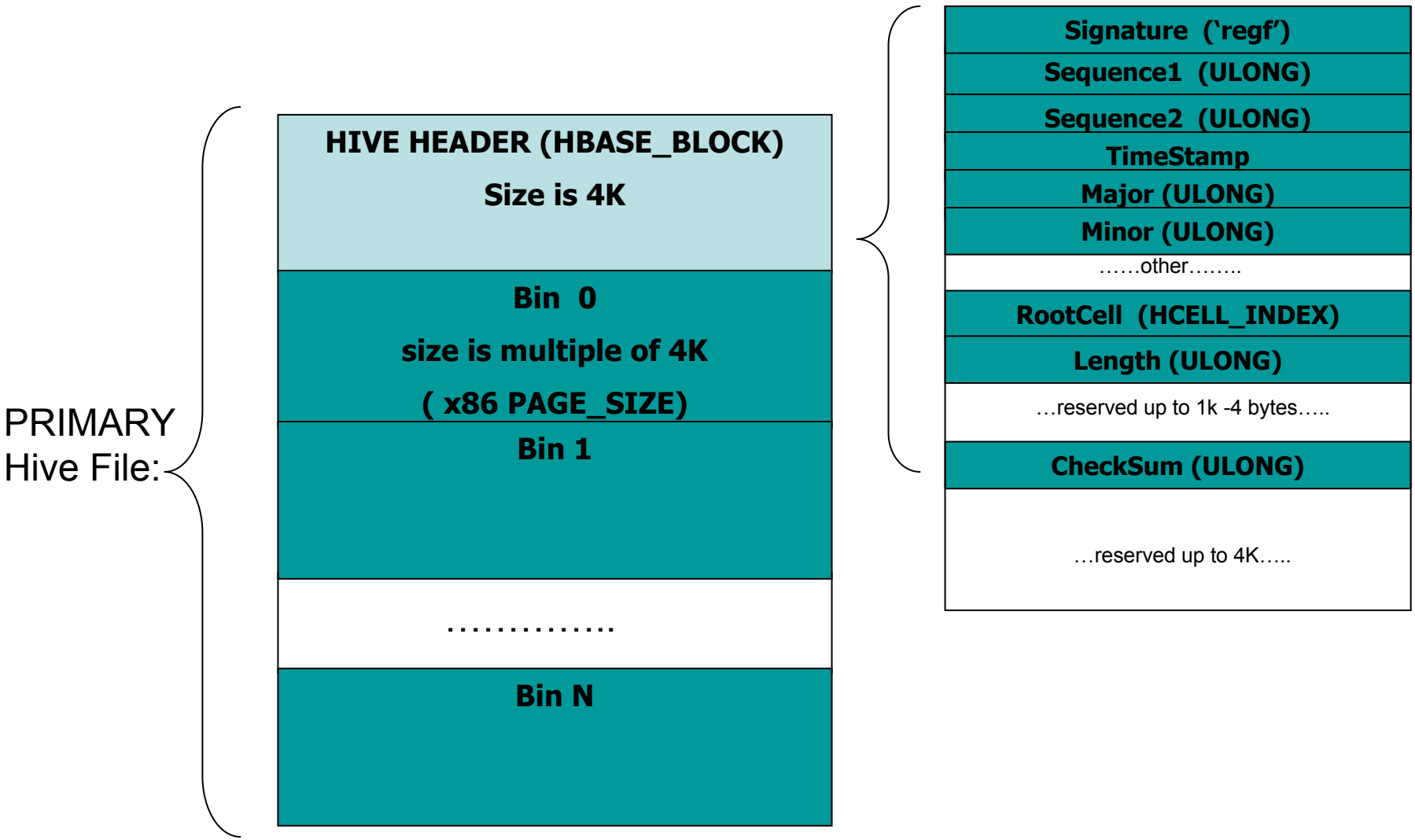
# NT Registry APIs: **Hive Ops**

| | |
|---|---|
| **NtSaveKey** (khandle, fhandle) | write the subtree at key khandle to file via fhandle |
| **NtRestoreKey** (khandle, hivefilename) | copy a subtree or complete hive at key |
| **NtLoadKey** (khandle, hivefilename) | mount a subtree or complete hive at key |
| **NtUnloadKey** (kname) | remove a subtree or hive loaded or restored at key kname |
| **NtReplaceKey** (newfile, roothandle, oldfile) | prepare to replace hive at next reboot |
| **NtCompressKey** (roothandle) | compress hive (inplace SaveKey) |

# Implementation Details

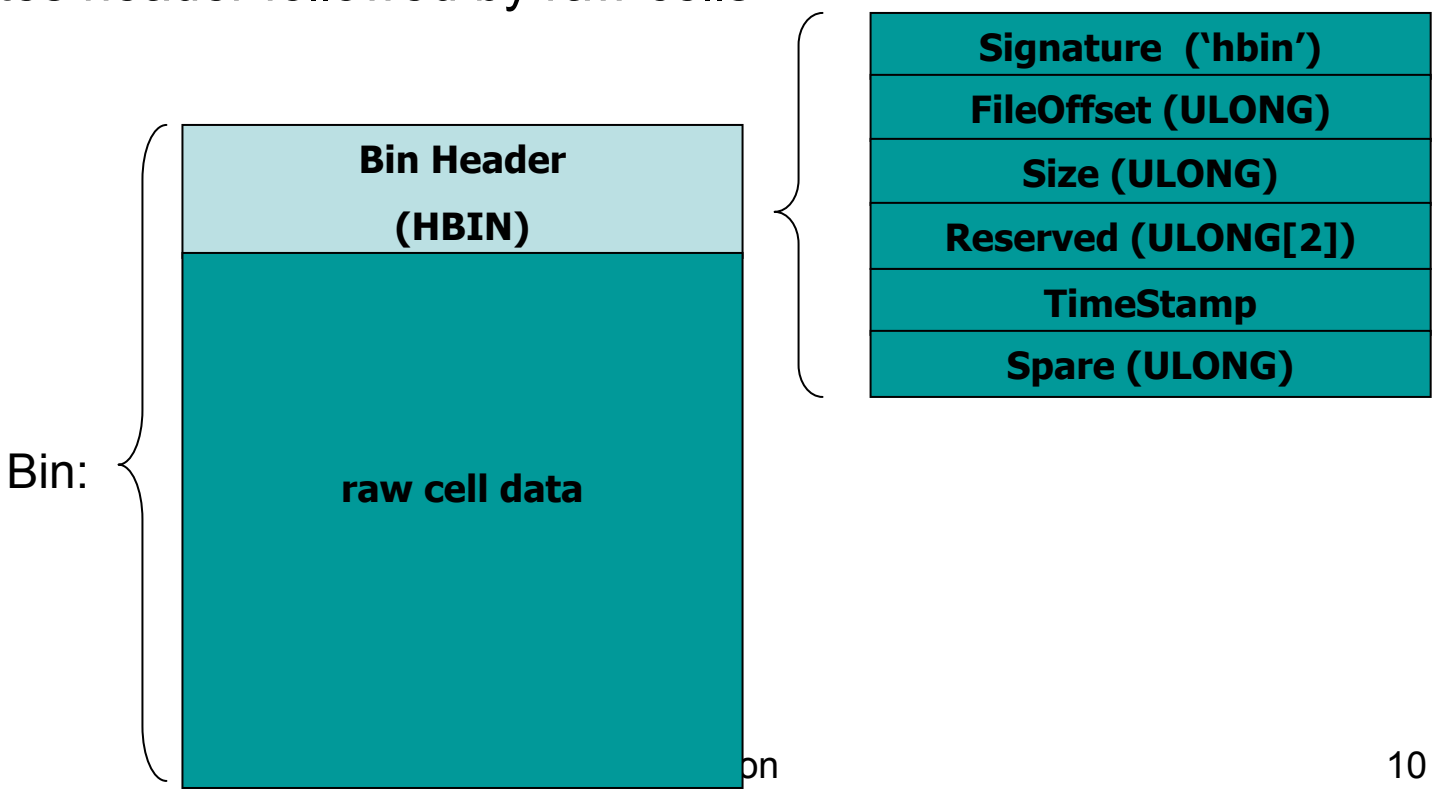
- A Hive is a file (two if you also count the .LOG)
  - PRIMARY – holds the actual hive data
  - .LOG – used only when flushing (crash recovery)
- Two storage mappings:
  - Stable – maps to the backing file
  - Volatile – in paged pool, lost after reboot
- PRIMARY grows in 256K increments – to avoid fragmentation
- First page (4k) is the header
- Followed by chained Bins
- I/O to primary is cached, PRIVATE_WRITE stream (no CC Lazy Flush, no MPW)

# Hive Layout



# Bin

- Collection of cells
- Size is increment of 4K
- Unit of hive growth
- 0x20 bytes header followed by raw cells

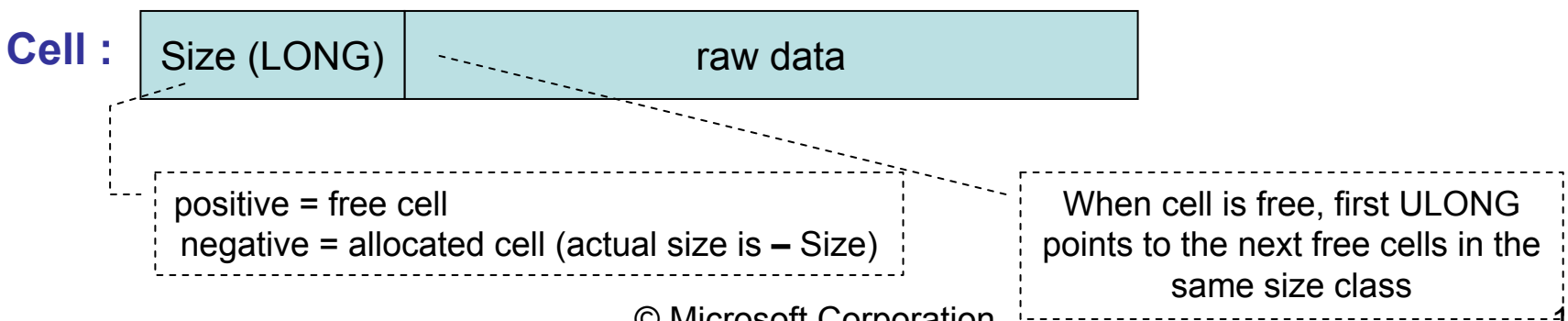


# Reading Stable Storage

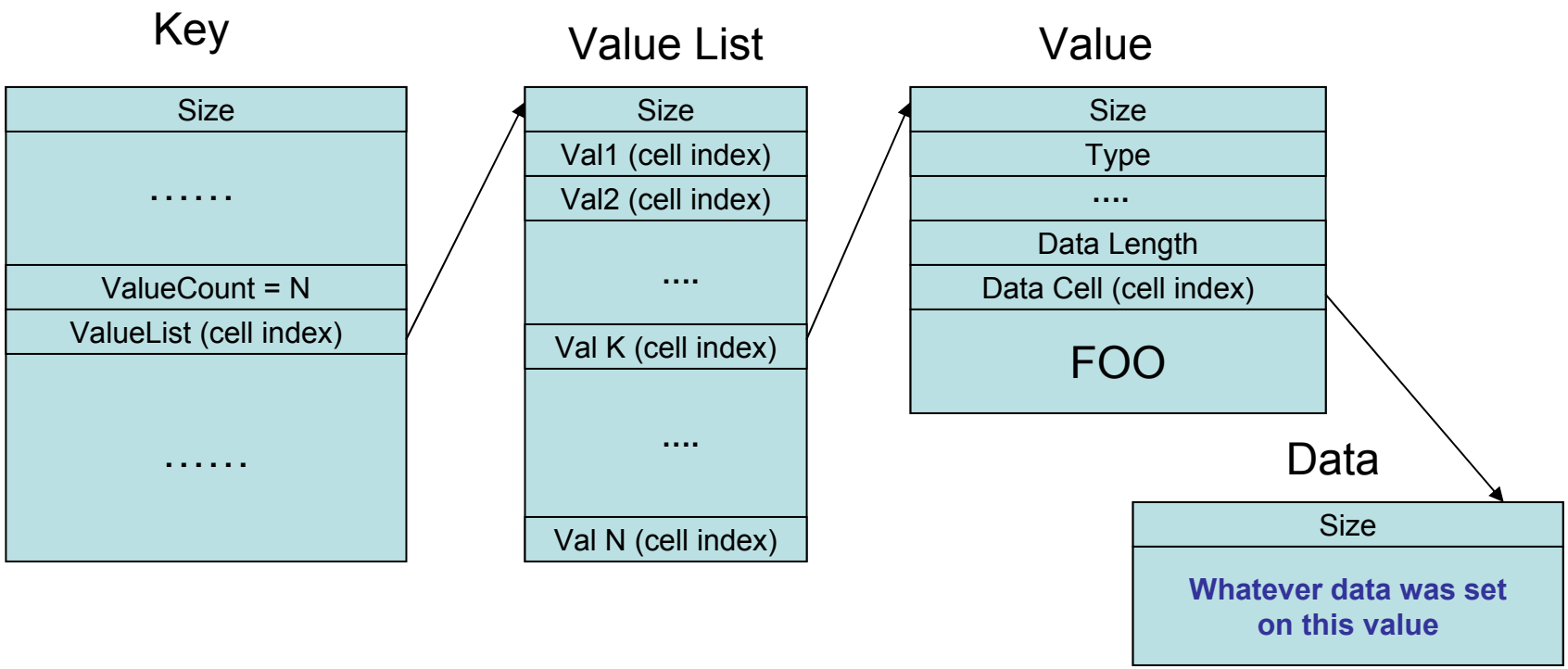
- PRIMARY file is CC PRIVATE_WRITE stream
  - no CC Lazy Flush, no MM MPW, no read ahead
  - complete control over when data hits the disk
- Map/fault in  16K views of the hive in the system cache address space (CcMapData)
  - Views cannot cross 256K boundary
- Max 256 views per hive, then reuse the oldest unused view (LRU)
  - Regardless of hive size, we only commit 4 megs of address space / hive (XP/.NET) → no RSL
- PRIMARY is loaded as Stable storage, Volatile storage is allocated from paged pool
- Dirtied data is pinned (CcPinMappedData) in physical memory up to the first hive flush

# Cell

- Unit of storage allocation within the hive
- Size rounded at 8 bytes boundary
- Referenced as a ‘cell index’ (HCELL_INDEX)
  - Cell index is offset within the file (minus 0x1000 – the header) – ULONG
  - Volatile cell indexes have first MSB set (i.e. 0x8xxxxxxx)
- Free Display bitmap keeps track of free cells with the same size
  - Up to 128 bytes exact match
  - 128 bytes up to 2048 bytes, rounded at power of 2
  - 2048 OR higher in the same list
  - Free cells in the same ‘size class’ linked together
- Always reuse free cells if one with the same size (or bigger) exists
  - If size is bigger than what we need, split & reenlist remaining
- Every time a cell is dirtied the whole page is marked dirty (in the Dirty Vector)



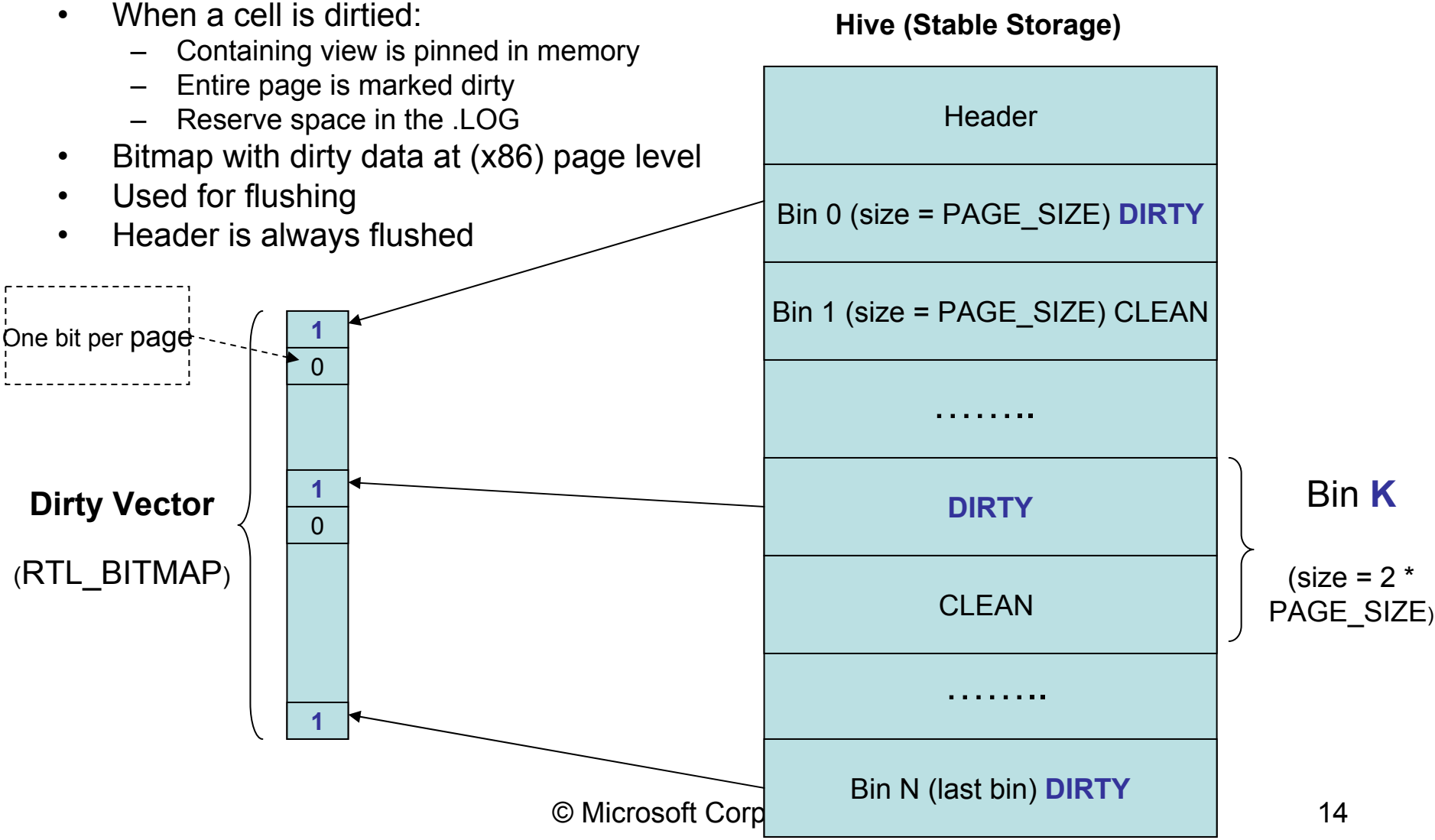
# Example – value lookup “foo”




- Raw cells are used to build up logical data
  - Keys, values, security descriptors, indexes etc all are made up of cells
  - Fetching in a key, might involve several faults spread across the hive file
    → Caching (Win2K) + locality enforcement (XP/.NET) to help with performance

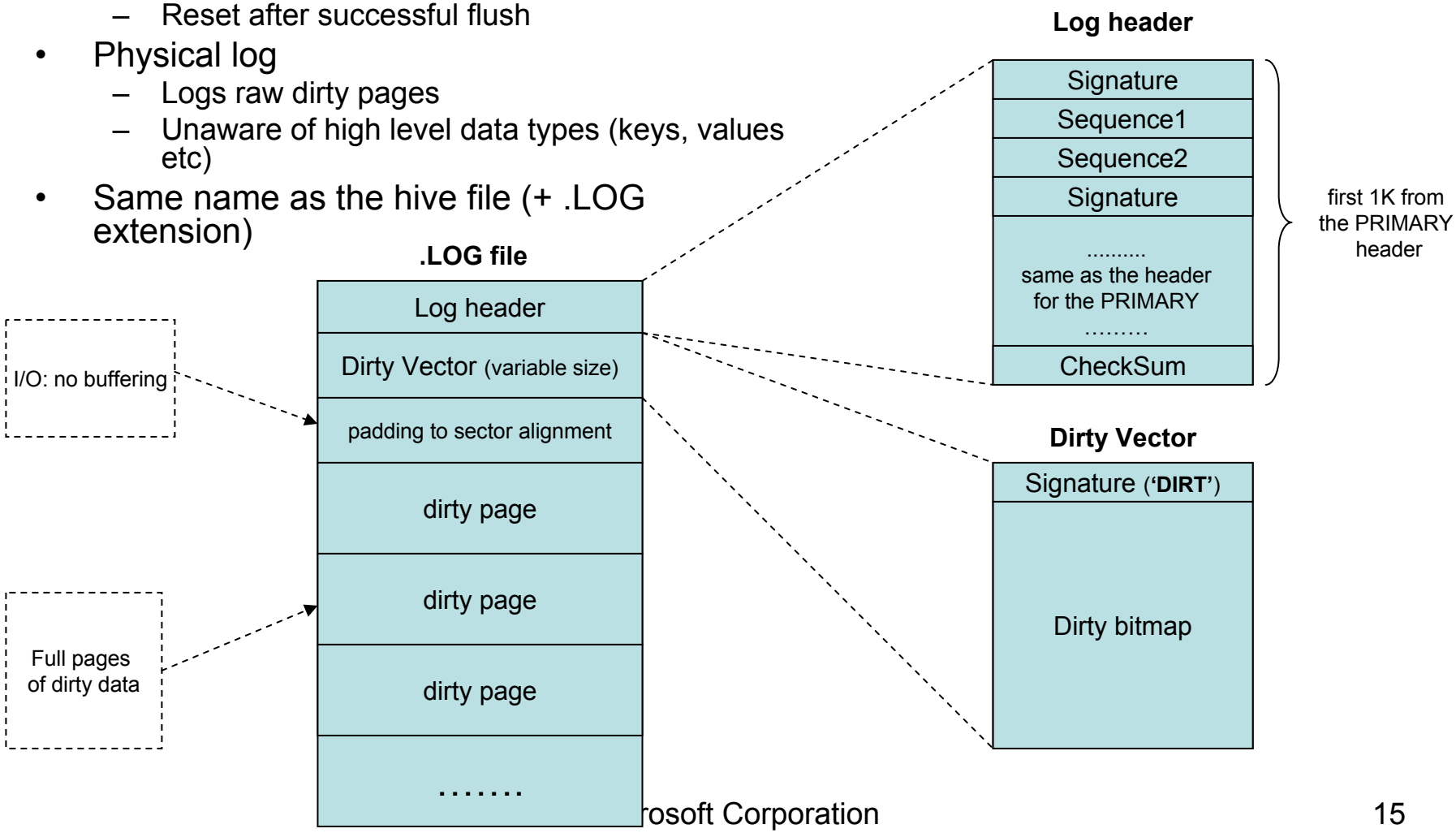
# Dirty Data

- When a cell is dirtied:
  - Containing view is pinned in memory
  - Entire page is marked dirty
  - Reserve space in the .LOG
- Bitmap with dirty data at (x86) page level
- Used for flushing
- Header is always flushed



# .LOG

- .LOG file used only while flushing
  - In the event of a crash during flushing
  - Reset after successful flush
- Physical log
  - Logs raw dirty pages
  - Unaware of high level data types (keys, values etc)
- Same name as the hive file (+ .LOG extension)



# Hive Flush

- The most expensive operation (by far)
- Triggered from outside – NtFlushKey/RegFlushKey
- … or from inside - Lazy Flush
  - Fires off 5 seconds after the write occurs (SetValue/DeleteValue/CreateKey/DeleteKey etc).
  - Walks the list of the hives loaded in the system and flushes every one that has dirty data
  - Ignores hives marked as NO_LAZY_FLUSH
- Others may read during flush, no write allowed
- All dirty data in the hive is written out
- All or none makes it to the backing file
- It is only after the flush that data is persisted to disk
  - i.e. If you:
    - CreateKey + SetValue
    - machine crashes (before lazy flush has a chance to flush the hive)
    - → the key/value is lost
- Automatic flush at hive unload

# Hive Flush – algorithm

1.Write the LOG
2.Flush the LOG

Past this point all dirty data
is in the log (on disk)

3.Header.Sequence1++; compute checksum
4.Write the Header to PRIMARY
5.Flush the PRIMARY

Crash past this point → Sequence1 != Sequence2
so we know the PRIMARY image has partial data

6.Write all dirty pages
7.Flush the PRIMARY

CcSetDirtyPinnedData
CcUnpinData
CcFlushCache

8.Header.Sequence2++; compute checksum
9.Write the Header to PRIMARY
10.Flush the PRIMARY

PRIMARY image is valid (on disk).

11.Reset LOG

# Loading (Mounting) a Hive

- **When:**
  - At boot: boot loader (NTLDR) & kernel (ntoskrnl.exe)
  - Explicitly, by calling NtLoadKey/RegLoadKey
    - Requires Restore privilege
  - File are opened in exclusive mode; and kept open by the kernel

# Loading (Mounting) a Hive

- **How:**
  - Read PRIMARY header; check it's validity (checksum, signature etc)
  - If sequence numbers don't match:
    - Hive has partial data, apply .LOG on top of PRIMARY
  - Build internal mappings as needed (Bins to Views)
  - Physical integrity check:
    - Walk the whole hive, check every single cell
  - Logical integrity check:
    - Walk the tree, check every key/value etc.

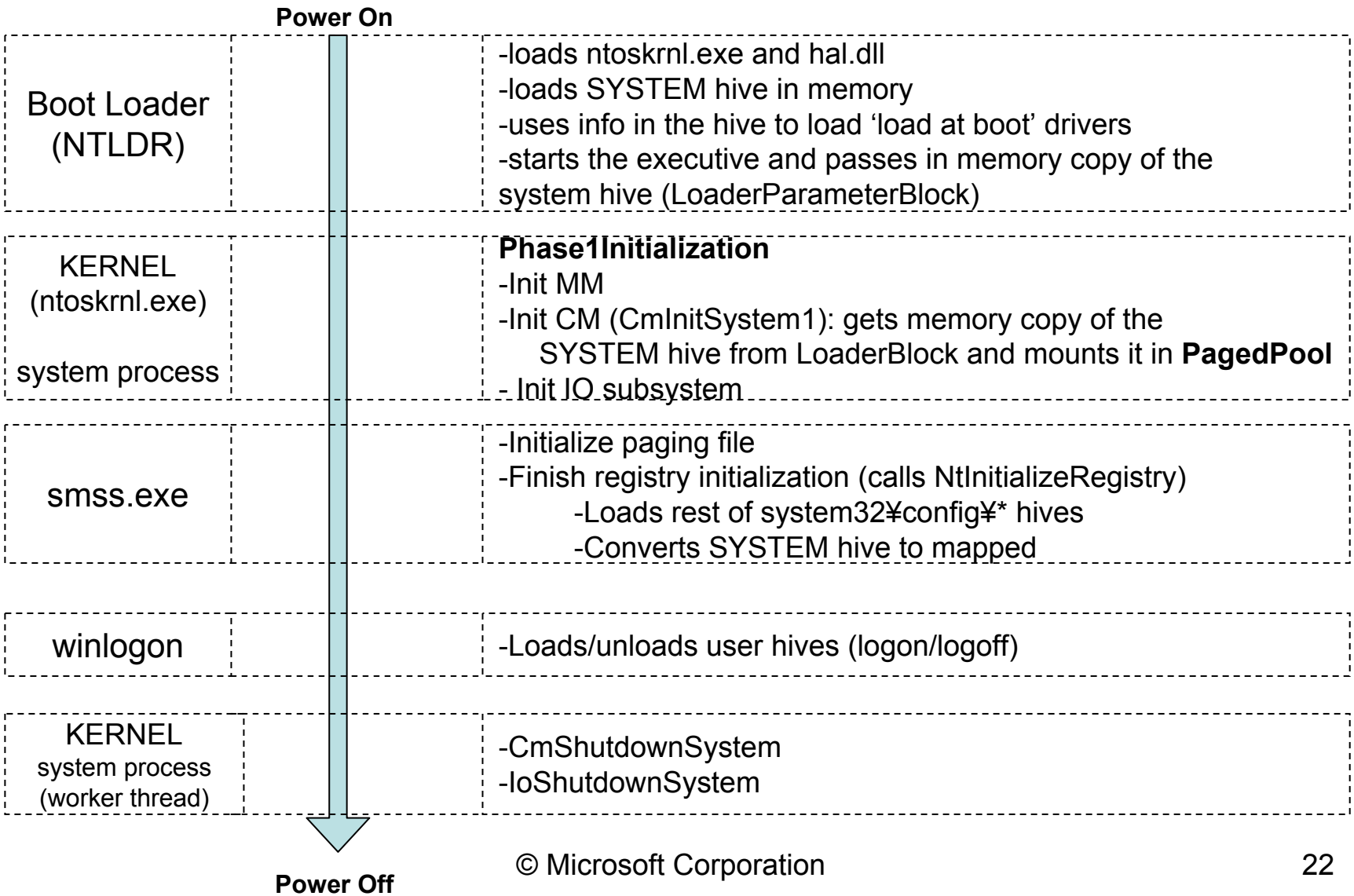
# Hives On a Typical (Clean) System

- Machine hives → %windir%¥system32¥config¥*
  - SYSTEM – mounted at HKLM¥System
  - SOFTWARE – mounted at HKLM¥Software
  - SAM – mounted at HKLM¥SAM
  - SECURITY – mounted at HKLM¥Security
  - .DEFAULT – used when a new account is created
    - Failure to load any of these → OS will not boot

# Hives On a Typical (Clean) System

- User hives → two per each user account
  - NTUSER.DAT – roams (if roaming profile enabled)
    - Mounted under HKEY_USERS¥<SID>
    - Mounted under HKEY_USERS¥<SID>_Classes
  - UsrClass.DAT – local (does not roam) – per user registration data
  - Stored in %USERPROFILE% folder.
  - Loaded at logon, or whenever the user is impersonated
- 'Special' user hives
  - Two per account as above; always loaded
    - S-1-5-18 → SYSTEM account
    - S-1-5-19 → Local Service
    - S-1-5-20 → Network Service
- Clusters – one additional hive: CLUSTER (cluster db)
  - → %windir%¥Cluster¥Cluster
- Any user/app with Restore privilege can mount own hive

# Life Span



# Backup/Restore …of the registry

- **Backup:**
  - NtSaveKey(Ex) – saves an entire hive to a file
  - Also does compression
    - Tree copy to a in memory temporary & volatile hive
    - Save temporary to destination file
      - Slow on big hives
  - Ex called with REG_NO_COMPRESSION much faster
    - Just dumps what's there to the file
    - No compression.
  - Requires SeBackupPrivilege
- **Restore:**
  - NtReplaceKey(ExistingFile,NewFile,OldFileName) – followed by a reboot
    - NewFile hive is mounted/checked/unmounted
    - ExistingFile → OldFileName ; NewFile → ExistingFile
    - Both files are kept open until reboot
      - Any changes made to the hive past this are lost at reboot
        - » Because the hive still maps to the old (existing) file
  - Requires SeRestorePrivilege

# Limits

- Win2K
  - RSL (Registry Size Limit) up to 80% sizeof(PagedPool)
    - Entire hive file loaded in paged pool
  - SYSTEM hive ~ 12 MB
    - sizeof(SYSTEM hive) + sizeof(ntoskrnl.exe) + sizeof(hal.dll) + sizeof(all drivers loaded at boot) <= 16 MB
      - Win2k loader only sees the first 16 MB of physical memory
- XP/WS03
  - No RSL – up to available space on system drive
    - only 4MB commit per hive
  - Sizeof(SYSTEM hive) <= min(200 MB, ¼ physical memory)
    - XP/.NET loader sees ¼ physical memory

# Summary

Registry intended to maintain config info

Win32 registry interfaces in Advapi32

Registry implementation in kernel

Native APIs are NT APIs

Used by kernel, drivers, system, apps, security, policy, etc.

# Discussion