Portable Systems Group

NT OS Memory Management Guide For I/O

Author: *Lou Perazzoli*

Original Draft 1.0, December 15, 1990
Revision 1.1, January 8, 1991
Revision 1.2, January 28, 1991

1. Introduction

This specification describes the memory management support routines available to I/O drivers, their usage and limitations.

2. Overview

The typical device driver has to deal with a number of memory management related issues - allocating buffers, working with MDL's, mapping device registers, etc. By properly designing the interaction between the driver and the memory management support routines, drivers will perform better in throughput, latency and system impact. Architectural differences between various architectures should be considered such that drivers are written to be as portable as possible.

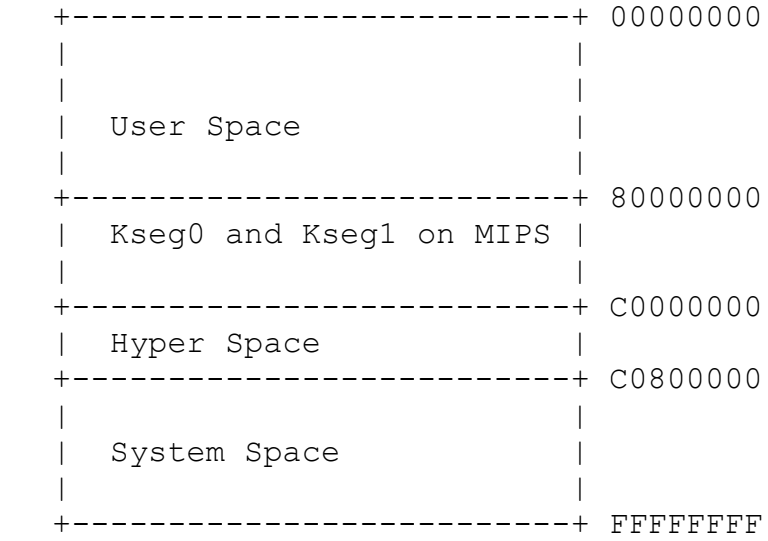
Memory management supports a 4-gigabyte virtual address space. It is important to understand the differences between virtual addresses and physical addresses (and on MIPS how physical addresses appear in the virtual address space).

The 4 GB address space is divided into 3 regions:

- User space - Consists of 2 gigabytes which is unique for each address space. The page ownership for this region is user mode.
- Hyper space - Consists of 8 megabytes with a page ownership of kernel mode and is unique for each address space. Page table pages, working set lists, PTEs reserved for temporary mappings, and other address space unique structures reside in this region.
- System space - Consists of almost 2 gigabyte which is shared among all address spaces and has a page ownership of kernel mode.

The page ownership (user mode or kernel mode) is used for access checks for operations on virtual addresses.

Layout of Virtual Address Space:



System space contains a paged and a non-paged area. The paged area starts at the low addresses and grows upward, while the nonpaged area starts at the high addresses and grows downward.

3. Processes and Working Sets

Each process has a unique virtual address space which is independent from all other processes in both user space and hyper space. However, the system space portions have identical page translations and the non-paged portion of the system space can be referenced in any process at any IRQL. These items are very important for device drivers because when an interrupt occurs the processor could be executing in any thread's context.

As a thread executes and accesses non-valid virtual addresses (addresses that have no corresponding physical address) the pages are made valid (i.e., translated to a physical address), and are placed into the process's working set. Each process has a unique working set which consists of the set of all pageable addresses which are currently valid in the process. This includes both user space and system space addresses. This is important to note since a pageable system address that is resident in one process may not be valid in another process and a reference to that address may cause a page-fault to make the reference valid.

The working set has a minimum and maximum size. As the thread executes and faults more pages into the working set, the working set may exceed its minimum at which time the page fault routine determines if the working set is allowed to grow or if a page is to be removed from the working set. Hence, the working set acts as a process-specific quota.

4. Probe and lock pages

The **MmProbeAndLockPages** function takes as input an MDL which has the *StartVa, ByteOffset,* and *ByteCount* fields initialized. The function checks the specified range for access (read or write) and

locks the physical memory corresponding the the virtual addresses and puts the "page frame numbers" for the physical addresses into the MDL. In addition, the *Process* field of the MDL is initialized to the current process.

The **MmProbeAndLockPages** function keeps track of the number of pages in user-space each process has locked in memory and refuses to lock pages (it raises the exception STATUS_NO_MEMORY) if the total number of locked pages would exceed the working set minimum minus some small constant.

/Darryl should MmBuildMdlForNonPagedPool and MmMapLockedPages fill in the system VA field of the MDL if it is NULL?/

Another important aspect of **MmProbeAndLockPages** is that when the virtual address specified in the MDL is in the user's portion of the address space, when MmProbeAndLockPages returns the completed MDL, the user may change the address space. This means that the virtual address in the MDL may no longer correspond to the physical pages in the MDL. This causes no problem as long as the device never accesses the buffers through both the MDL (either physically or by mapping them in the system address space) and by the user's virtual address. This is a very important point.

To unlock the pages that were locked by **MmProbeAndLockPages** invoke the **MmUnlockPages** function specifying the same MDL that was used in the MmProbeAndLockPages call. This will cause the pages to be unlocked and the locked count to be decremented in the process.

If the buffer resides in the non-paged portion of system space and I/O completion is not invoked to unlock the buffer, the routine **MmBuildMdlForNonPagedPool** can be used to complete the MDL. This routine does not increment any reference counts or checks to ensure pages are resident, it merely updates the MDL with the corresponding page frame numbers.

5. Mapping Locked Pages

Certain devices, such as the standard AT disk, require the buffer to be accessed virtually rather than physically at high IRQL. But the pager, which is invoked when a virtual address does not have a corresponding physical address, can only be called at an IRQL of APC_LEVEL and below and at a mutex level below **MUTEX_LEVEL_WORKING_SET**.

To create virtual addresses that "map" the user's buffer and can be accessed at high IRQLs use the combination of **MmProbeAndLockPages** and **MmMapLockedPages**. The MmProbeAndLockPages function will complete the MDL and the MmMapLockedPages function will create a range of non-paged virtual addresses which map the physical buffer.

When invoking MmMapLockedPages the *AccessMode* argument should always be *KernelMode*. When the argument is specified as *UserMode* the buffer specified by the MDL is mapped into the user-mode portion of the current process and hence can only be referenced in the context of that process. This feature is used only by File System Processes (FSPs).

The **MmUnmapLockedPages** function deletes the mapping to the buffer. It is called with the *BaseAddress* that was returned by MmMapLockedPages and the same MDL that was passed into MmMapLockedPages.

The MmMapLockedPages and MmUnmapLockedPages have a cost which increases dramatically on a multi-processor system. When the pages are locked, the non-paged portion of system space is searched for an empty range to contain the buffer. The time for this search varies based on the number of pages in the MDL. Single page requests complete immediately, whereas multi-page requests may take slightly longer to locate a suitable range. When unmapping the pages, the addresses are marked as unused and returned and the **translation buffer is invalidated on all processors**. Note that if the request is for a single page then only a single address is invalidated if the underlying hardware supports single invalidation. Note that the 386 does not support single invalidation, but the 486 and the R4000 do.

The bottom line is that if the device supports DMA operations the driver should be designed such that **MmUnmapLockedPages** is never invoked.

6. Mapping I/O space

Most devices have control registers which reside in the I/O portion of the physical address space. In order to access these registers a corresponding non-pageable virtual address must be created which refers to the physical I/O address. This is accomplished using the **MmMapIoSpace** service. The returned address is the virtual address which corresponds to the specified physical I/O address. This virtual address is created either cached or non-cached depending on an argument. Note that only certain processors support non-cached memory via the translation hardware (MIPS and 486).

When the driver is being unloaded **MmMapIoSpace** is invoked to return the reserved address space back to the system.

7. Physically Contiguous Memory

On certain archaic systems, devices require the buffers to be physically contiguous in memory. On NT the memory management system has no support for memory compaction or other mechanisms to obtain physically contiguous pages. Hence a simple solution is required; when the system is initialized and non-paged pool created, all physical pages used for non-paged pool are contiguous.

Therefore during system initialization any allocation from non-paged pool is most likely physically contiguous. However, as the demand for nonpaged pool increases and non-paged pool is automatically expanded, the pages added are NOT physically contiguous. This means that after the system has been operating for a some period of time there is a possibility that no non-paged contiguous memory can be allocated.

Drivers that require physically contiguous memory should allocate their memory at driver initialization using the **MmAllocateContiguousMemory** function. The driver should then copy the user's buffer into or out of the physically contiguous area during its operation. Note depending on how the driver is

designed, the copy may not require building a MDL for the user's buffer, rather it can use a simple RtlMoveMemory inside of a try/except block.

When the driver is being unloaded or the need for physically contiguous memory is no longer present, the memory should be deallocated with the **MmDeallocateContiguousMemory** function.

8. Non Cached Memory

Certain devices require that buffers be shared between the device and the driver. These may be ring buffers which present a list of transfers to the device and a protocol is followed to insert and remove from the list, or other types of buffers. But the key thing about these buffers is that reads from and writes to the buffers must go directly to memory and not to the processor's cache where the device cannot see the changes.

The **MmAllocateNonCachedMemory** function allocates a range of nonpaged memory within system space and makes that memory non-cached. If insufficient memory is available, NULL is returned.

Currently, full pages are allocated to the request, so a request for 8 bytes allocates a full page. This is to avoid putting another pool type in the system and having the overhead of managing the pool as allocating noncached memory should be an infrequent function. It is anticipated that certain drivers will obtain some non cached pages at initialization and use only those pages for the life of the driver.

The **MmDeallocatedNonCachedMemory** function returns the non-cached memory back to the system pool.

9. Obtaining Physical Addresses

The **MmGetPhysicalAddress** function returns the physical address for a corresponding virtual address. This function should only be used to obtain the physical address of a virtual address that is in the non-pageable portion of the system.

10. Paged and NonPaged Pool

Paged and NonPaged pool requests have the following characterists:

- requests of PAGE_SIZE or less are always physically contiguous and do not cross a page boundary
- the returned address is aligned on a quadword boundary (low order 3 bits of virtual address are zero)
- pool allocations do not share cache lines with any other pool allocations.

11. Determining Non-Paged System Space Addresses

The **MmIsNonPagedSystemAddressValid** function allows the caller to determine if a given virtual address is within the non-paged portion of the system space and is currently mapped (valid). This includes such regions as non-paged pool, kernel stacks, and mapped locked pages.

12. Useful Macros

The following macros are also provided to aid in dealing with virtual addresses and buffer sizes:

- **ROUND_TO_PAGES**
  - Given a size, round the size up to the next page multiple.
- **BYTES_TO_PAGES**
  - Given a size, compute the number of pages required to contain a buffer of that size.
- **BYTE_OFFSET**

-Given a virtual address, return the byte offset for that virtual address.

- **PAGE_ALIGN**

-Given a virtual address, return the corresponding page aligned virtual address.

- **MM_IS_SYSTEM_VIRTUAL_ADDRESS**

-Given a virtual address, return *TRUE* if the virtual address is within system space.

Revision History:

Original Draft 1.0, December 15, 1990.

Revision 1.1, January 8, 1991

1. Added MmIsAddressInNonPagedSystemSpace.
2. Editorial changes.
3. Added useful macros.

Revision 1.2, January 28, 1991

1. Added CacheEnable argument to MmMapIoSpace.
2. Changed MmIsAddressinNonPagedSystemSpace to MmIsNonPagedSystemAddressValid to more reflect the actions.