Portable Systems Group

Windows NT Memory Management Design Note

Author: *Lou Perazzoli*

Revision 1.8, July 24, 1990

1. Overview

This specification discusses memory management issues as related to the Intel 860. The Intel 386/486 architecture has a similar PTE format and the software PTE definition is identical between the two architectures.

The memory management subsystem is responsible for the mapping of physical memory into the virtual address space of a process. The memory management functions are implemented by several distinct pieces of the executive. The *translation-not-valid fault handler* (pager) is the exception service routine that responds to page faults and makes virtual pages resident on behalf of a process. The *modified page writer* is responsible for writing modified pages to the appropriate backing store so the physical pages can be reused. The *balance set manager* is responsible for reducing process working set sizes to gain more pages of memory. Executive routines and system services are provided to allow processes some degree of control over the behavior of their memory while executing and to support various executive functions.

The memory management subsystem has the following requirements:

- A number of processes may exist in memory simultaneously, each only allowed access to its own address space
- Support for the I/O subsystem and process structure.
- A process's pages need not be totally resident at any one time.
- Virtual pages of a process are not physically contiguous.
- Processes executing the same image will share read only code and data.

The memory management subsystem imposes requirements due to the nature of page fault handling that page faults cannot occur at interrupt request levels (**IRQL**) greater than **APC_LEVEL**. This allows the pager to acquire and release mutexes in a deadlock free manner.

Because page faults can occur at **IRQL**s **0** or **APC_LEVEL**, and routines executing at **IRQL 0** can be interrupted by **APCs** at **IRQL APC_LEVEL**, all memory management function which acquire mutexes operate at **IRQL APC_LEVEL**.

2. Address Space Layout

The Intel 860 supports a 4-gigabyte virtual address space. The virtual address space is divided into 3 parts.

- o User Space - Consists of 3 gigabytes which is unique for each address space. The page ownership for this region is user mode.

- o Hyper Space - Consists of 12 megabytes with a page ownership of kernel mode and 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.

- o System Space - Consists of almost 1 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:

```
+------------------------+ 00000000
|                        |
|                        |
|  User Space            |
|                        |
|                        |
+------------------------+ C0000000
|  Hyper Space           |
+------------------------+ C0C00000
|                        |
|  System Space          |
|                        |
+------------------------+ FFFFFFFF
```

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.

The highest 32k bytes of the address space are reserved for mapping non-paged constructs used by the kernel. The Intel 860 dispatches traps to virtual address FFFFFF00, which is in the last page of the virtual address space. The page before the trap page is used to map constructs for the kernel to utilize in trap handling. This page is used for saving and restoring various registers, locating the kernel process object and locating the user's thread control block (TEB). In addition, the page is double mapped into the user portion of the address space as read-only to allow routines executing in user mode to locate the TEB.

The last 64K bytes of user space (BFFF0000 - BFFFFFFF) is set as no access. This allows argument probing within the executive to be accomplished by comparing the specified address to the highest valid address in user space (BFFEFFFF). If the address less than the highest valid address, it resides within user space, and the access is allowed (note that the actual page protection may not permit the access). By setting the last 64K bytes as no access, an argument that straddles the boundary causes an access violation.

If access to the user's data is allowed, the routine requesting the access check eventually attempts to read or write the data. If the page protection prevents the access, an access violation exception is delivered and is handled by the routine attempting the access.

Any time data which resides in the user space portion of the address space is being read or written from kernel mode, an exception handler must be present to handle access violations. This is required because another thread within the process can potentially change the protection or validity of any page within user space after argument validation has occurred.

3. Initial Page Directory at Address Space Creation

When an address space is initially created, the user portion of the address space is set to no access. This is accomplished by zeroing entries 0 through 767 of the new process's PD (page directory).

Nonpaged system space is initialized by copying PD entries 992 through 1023 of the current process to the PD of the new process. This provides an identical view of a 128 megabyte nonpaged system space in every process.

Paged system space is created by building a *Virtual Address Descriptor* which describes the range of paged system space and the section which maps the paged system space. Initial references to paged system space result in page faults which are then resolved to map the corresponding page within the paged system space.

Hyper Space is created by mapping PD entry 768 to the physical page that contains the PD itself, and mapping entry 769 to a private page table page that will contain the page table entries to map the address space control structures. During the life of the process, this region can grow expand by 4 megabytes due to large working sets.

By mapping the PD to itself means that all references through PD entry 768 refer to page table pages. For example, the 32-bit word at address 0xC0000000 is the page table entry (PTE) which describes the page which maps addresses 0 through 4095. By mapping the page directory onto itself, all processes have their page table pages mapped at the same address, and one process does not have a view of another process's page table pages.

4. Page Frame Database (PFN)

The PFN database contains information about each page of physical memory. The fact that this information must be available while the page is being used prevents this information from being stored in the page itself.

Every physical page of memory is in one of 5 states:

- **Active and valid**. A valid PTE refers to this page.
- **Transition**. The page is on either the modified list or the standby list. The page contents are still valid, but the page is not currently in any process's working set. A non-valid, transition PTE refers to this physical page.
- **Free**. The page is on the free list and may be used immediately.
- **Zeroed**. The page is on the zeroed list. It may be used immediately and contains all zeroes.
- **Bad**. The page is on the bad list. The page has parity or hard ECC errors which prevent it from being used.

The PFN database contains information to link all pages except active/valid pages together in lists. This allows pages to be easily manipulated to satisfy page faults.

The PFN database consists an array of records indexed by the physical page number, and has the following structure:

```
+------------------------------------+
| Forward link,  Event Address or    |
|     Working Set Index Hint         |
+------------------------------------+
| Virtual Address of (Proto) PTE     |
+------------------------------------+
| Backward link  or  Share Count     |
+------------------------------------+
| Reference Count | # valid PTEs     |
+------------------------------------+
| Original PTE contents              |
+------------------------------------+
| PFN of PTE      | flags            |
+------------------------------------+
```

When the page is on one of the lists (free, standby, modified, zeroed, or bad), the Forward and Backward link fields link the elements of the list together. Note that when a page is on one of the lists, the *ShareCount* must be zero and therefore can be overlayed with the backward link, but the *ReferenceCount* may not be zero as there could be I/O in progress for this page. This is true when the page is being written to backing storage.

When a page is active/valid or transition the original contents of the PTE (which could be a prototype PTE, see section 7 for information about prototype PTEs) are stored in the PFN element. This allows the PTE to be restored when the physical page is no longer resident. In addition to the contents of the PTE, the virtual address of the PTE and the physical page number of the page which contains the PTE are stored in the PFN element. These fields provide the virtual and the physical address of the PTE which maps the page.

When a page is active/valid, the *ShareCount* field represents the number of PTEs which refer to this page. As long as the *ShareCount* is greater than zero, the page is not eligible for removal from memory.

The *ReferenceCount* field represents the number of reasons the page must be kept in memory. The *ReferenceCount* field is incremented when a page initially becomes valid (*ShareCount* becomes non-zero) and when the page is locked in memory for I/O. The *ReferenceCount* is decremented when the *ShareCount* becomes zero and when pages are unlocked from memory. When the *ReferenceCount* becomes zero, the page is placed on the free, standby or modified list depending on the contents of various flags.

The working set index hint field is only valid when the *ShareCount* is non-zero. This field indicates the index into the working set where the virtual address that maps this physical page resides. If the page is a private page, then the working set index field always contains the proper value as the page is only mapped at a single virtual address. In the case of a shared page, this hint value is only correct for the first process which made the page valid. The process which sets the hint field is guaranteed that the hint field refers to the proper index and does not need to add the Working Set List Entry referenced by the hint index into the working set tree. This reduces the size of the working set tree allowing faster searches for particular entries.

The following information is contained in the flags field:

- **Modified state** - Indicates if the page was modified requiring its contents to be saved if it is removed from memory.

- **Prototype PTE** - Indicates the PTE referenced by the PFN element is a prototype PTE.

- o **In-page read in progress** - Indicates that an in-page operation is in progress for the page. In this case the event address is stored in the field used for the forward link.
- o **Parity error** - Indicates the physical page contains parity or ECC errors.
- o **In-page error** - Indicates an I/O error occurred during the in-page operation on this page.

The number of valid PTEs field contains the count of PTEs within this physical page which are either valid or in transition. If this field is not zero, the page cannot be removed from memory.

5. Paged and Nonpaged Dynamic Memory

At system initialization the memory management subsystem creates two dynamic memory regions for use by the executive for storage allocation and deallocation. These storage pools are located in the system part of the address space and are mapped at the same virtual address in every process.

The *nonpaged pool* consists of a range of virtual addresses which are guaranteed to be resident in physical memory at all times and thus can be accessed from any address space without page faults. This is mapped through a single set of page table pages which are shared by each process.

The *paged pool* consists of a range of virtual addresses which may be paged in and out of a process's working set. Each process has a unique copy of page table pages which refer to the paged portion of system space. This means that there is no guarantee that an address within the paged portion of the system will not cause a page fault at any time. For this reason data structures which are operated at **IRQL** levels greater than **APC_LEVEL** must be allocated from nonpaged pool.

6. Page Fault Handling

A page fault occurs when the valid (present) bit in a PTE is zero indicating the desired page is not resident in memory. When a page fault occurs, the memory management system examines various structures to determine if the fault is a page fault or an access violation.

Upon entry, the page fault handler locates the PTE which describes the page. Note, that the page table page containing the PTE could be non-resident, or could not exist. If it does not exist, the PTE which describes the page is treated as if it were zero.

Note, that before the page is made valid, access checks are performed to ensure the requestor has access to the page.

Once the PTE is found, it can be in one of six states:

1. **Active and valid** - Another thread already faulted the page into memory.

2. **Transition** - The desired page is in memory on either the standby or modified list.

3. **Demand Zero** - The desired page should be satisfied with a page of zeroes.

4. **Page File Format** - The desired page resides within a paging file and should be in-paged.

5. **Prototype PTE Format** - The desired page is potentially shared and this PTE refers to the Prototype PTE for the shared page.

6. **Unknown** - The PTE is zero. This means the *virtual address descriptors* should be examined to determine if this virtual address has been allocated.

The following figures describe the contents of a PTE for the first 5 cases. The below abbreviations are used;

Hardware usage (defined by Intel 860 reference manual):

vld - valid bit (present bit)
wrt - write bit
own - owner (user)
wt - write through cache bit
cd - cache disable bit
acc - accessed bit
dty - dirty bit
rsv - reserved, must be zero when vld is one
gp - guard page
prot. code - protection code

value	protection
0	no-access
1	read-only
2	execute-read
3	execute-only
4	read-write
5	execute-read-write
6	read-writecopy
7	execute-read-writecopy

8+(0-7)	no-cache + protection (0-7)
16+(0-15)	guard page + protection (0-15)

Software usage (defined by **Windows NT** memory management):

trn - if set page is in transition state (pro must be zero)
cow - if set page should be copied on write operation
pro - if set in a PTE, then the PTE refers to a prototype PTE. If set in a prototype PTE, then the prototype PTE is in mapped format.

6.1 Valid PTE

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |t| |c|r|r|d|a|c|w|o|w|v|
| Page Frame Number             |r| |o|s|s|t|c|d|t|w|r|l|
|                               |n| |w|v|v|y|c| | |n|t|d|
|                                0 0   0 0             1|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.2 Transition PTE

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |t|p| protect |c|w|o|w|v|
| Page Frame Number             |r|r|  code   |d|t|w|r|l|
|                               |n|o|         | | |n|t|d|
|                                1 0                   0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.3 Demand Zero PTE

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |t|p| protect | page  |v|
| Page File Offset (all zeroes) |r|r|  code   | file  |l|
|                               |n|o|         | number|d|
|0 0 0 0 0 0 0 0 0 0  0  0 0 0 0 0 0           0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.4 PTE Referring to Page in Paging File

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |t|p| protect | page  |v|
| Page File Offset (20 bits)    |r|r|  code   | file  |l|
|                               |n|o|         | number|d|
|                                0 0                   0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.5 PTE Referring to Prototype PTE (protection code in protoPTE)

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                |p|r|r| Offset to   |v|
| Offset to Prototype PTE        |r|s|s| Proto PTE   |l|
|           <27:7>               |o|v|v|    <6:0>    |d|
|                                 1 0 0               0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.6 PTE Referring to Prototype PTE (protection code here)

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                |p| protect |       |v|
| Offset to Prototype PTE        |r|  code   |       |l|
|           <27:7>               |o|         |       |d|
|1 1 1 1 1 1 1 1 1 1     1 1 1 1 1 1                  0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Note that a PTE in the transition state is indicated by the transition bit being set to 1 and the bit normally occupied by the prototype PTE indicator set to 0. In this case, the state of the prototype PTE indicator is maintained in the page frame database.

7. Prototype PTEs

Prototype PTEs have a similar format as normal PTEs, but are a memory management structure and never reside in a page table page. Prototype PTEs are created as a result of section creation and POSIX fork and provide the mechanism for sharing pages between multiple address spaces.

When a view of a section is mapped, the PTEs in the address space refer to the corresponding prototype PTE for the section. Hence when multiple address spaces map the same view, the same prototype PTEs are referenced allowing sharing of the physical page.

When a PTE refers to a prototype PTE, the address of the prototype PTE is calculated by multiplying the offset value by 4 and adding it to the system prototype PTE base address. Note that prototype PTEs are allocated from paged pool.

Once the prototype PTE has been located, it can be in one of 5 states:

1. **Active and valid** - Another thread already faulted the page

2. **Transition** - The desired page is in memory on one of the memory management lists (standby or modified list)

3. **Demand Zero** - The desired page should be satisfied with a page of zeroes

4. **Page File Format** - The desired page resides within a paging file and should be in-paged

5. **Mapped File Format** - The desired page is in a mapped file and should be in-paged

7.1 Valid Prototype PTE

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             |t|p|c|r|r|d|a|c|w|o|w|v|
| Page Frame Number           |r|r|o|s|s|t|c|d|t|w|r|l|
|                             |n|o|w|v|v|y|c| | |n|t|d|
|                              0 0   0 0             1|
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.2 Transition Prototype PTE

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             |t|p| protect |c|w|o|w|v|
| Page Frame Number           |r|r|  code   |d|t|w|r|l|
|                             |n|o|         | | |n|t|d|
|                              1 0                   0|
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.3 Demand Zero Prototype PTE

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             |t|p| protect | page   |v|
| Page File Offset (all zeroes) |r|r|  code   | file   |l|
|                             |n|o|         | number|d|
|0 0 0 0 0 0 0 0 0 0  0  0 0 0 0 0 0          0 0 0 0 0|
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.4 Prototype PTE Referring to Page in Paging File

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             |t|p| protect | page   |v|
| Page File Offset (20 bits)  |r|r|  code   | file   |l|
|                             |n|o|         | number|d|
|                              0 0                   0|
+-+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.5 Prototype PTE Referring to Page in Mapped File

```
 3                             1 1 1
 1                             2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                |p| protect | Offset|v|
| Offset to SubSection           |r|  code   | Subsct|l|
|          <24:4>                |o|         | <3:0> |d|
|                                 1                   0|
+-+-+-+-+-+-+-+-+-+-+ ~~+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

8. Page Protection

The Intel i860 supports read only and read/write pages. Pages that are protected as either no-access or as guard pages are managed through software. This is accomplished by setting the protection mask appropriately and on the access trap, examining the protection mask and returning the appropriate status.

9. Retrieving a Free Page

The pager must supply a free page for demand zero faults, copy on write operations, and to satisfy in-page requests. For the demand zero case and in-paging less than a full page, the pager checks the zeroed page list for a page. If the zeroed page list is empty, the free list is checked and, if empty, the standby list is checked. If the standby list is empty, then no free pages are available and the thread enters a wait state for a free page.

In the case of copy on write and in-paging a full page, the free list is first checked, then the zeroed list and lastly the standby list.

If the page is to be removed from the zeroed or the free lists, the pager removes the page from the head of the list and updates the appropriate list structures. However, if the page is removed from the standby list, the PTE referenced by the corresponding PFN database element is currently in a transition state and must have its original contents restored.

This is accomplished by using the physical address information located in the PFN database element and mapping the PTE within the current process's hyper space. The transition PTE is then restored to its original state using the mapped virtual address, and the virtual address is unmapped. Note, to prevent unnecessary TB and cache flushes, a pool of PTEs for mapping addresses is maintained in hyper space and the unmapping of these virtual addresses is only done when the pool becomes exhausted. At this time all the reserved mapping PTEs are made invalid and the TB and caches are flushed permitting reuse of the PTEs.

10. In-Paging I/O

In-paging I/O occurs when a read operation must be issued to a file (paging or mapped) to complete a page fault operation. The in-page I/O operation is essentially synchronous. It is not APC interruptible, and the thread waits on an event once the I/O operation is issued.

In order to support in-paging, each thread control block contains two events and two I/O status blocks which are used for I/O requests. There are two events and IOSBs to allow the initial in-page I/O to incur a page fault, which could issue another I/O request. The second I/O request would use the other event/IOSB pair. This allows file retrieval pointers and other information to be kept in the pagable portion of the executive. Note, however, that information relating to a paging file must not be maintained in the pagable portion or the page fault cannot complete.

The pager uses a special modifier in the I/O request function to indicate paging I/O. Upon I/O completion, the *IoCompletion* mechanism recognizes this as paging I/O, and writes the IOSB and sets the event. It does not attempt to deliver a kernel mode APC or unlock buffers. Note that both the event and the IOSB are allocated in nonpaged pool and the event is set to false before the I/O is issued. Setting the event to false prevents a race condition where another thread issues a wait on the event before the I/O system has cleared the event.

When the event is set, the wait is satisfied, allowing the pager to continue in-page processing. The pager checks the IOSB status, updates the PFN database and other structures, and completes the page fault.

While the paging I/O operation is in progress, the faulting thread does not own any mutexes. This allows other threads within the process to incur and handle page faults and issue virtual memory APIs. This exposes a number of interesting conditions which must be recognized by the pager when the I/O completes and the mutexes are again acquired. These conditions are:

- Another thread could have faulted the same page.
- The page could have been deleted (and remapped) from the virtual address space.
- Another process could have faulted the same page.
- The protection on the page could have changed.
- The fault could have been for a prototype PTE and the page which maps the prototype PTE could be out of the working set.

The pager handles these conditions by saving enough state on the kernel stack before the paging I/O request such that when the request is complete, it can detect these conditions, and, in the pathological cases, just dismiss the page fault without making the page valid. When the faulting instruction is reissued, the pager will again be invoked, and the proper action taken.

10.1 Collided Page Faults

The case when another thread or process faults a page which is currently being in-paged is known as a *collided page fault*. This case is detected and handled optimally by the pager as it is a common occurrence in multi-threaded systems.

When an in-page operation occurs, a physical page is allocated for the I/O request, and the corresponding PTE is placed into the transition state referring to the physical page. The PFN database element for the physical page contains the original PTE contents, the physical address of the transition PTE, and the virtual address of the event which is used in the I/O request.

If another thread or process faults the same page, the collided page fault is detected by the pager noticing that the page is in transition and the read-in-progress flag in the PFN database element is set. In this case the pager issues a wait operation on the event address specified in the PFN database element.

When the I/O operation completes, all threads waiting on the event have their wait satisfied. The first thread to acquire the PFN database lock, is responsible for performing the in-page completion operations. This consists of checking the IOSB (which is at a known offset from the event) to ensure the I/O operation completed successfully, clearing the read-in-progress bit in the PFN database, and updating the PTE element.

If the IOSB indicates an in-page I/O failure, the PFN database element for the page has the in-page error flag set. The corresponding PTE element is not updated and an in-page error exception is raised in the faulting thread.

When subsequent threads acquire the PFN database lock to complete the collided page fault, the pager recognizes that the initial updating has been performed as the read-in-progress bit is clear, and checks the in-page error flag in the PFN database element to ensure the in-page completed successfully. If the in-page error flag is set, the PTE is not updated and an in-page error exception is raised in the faulting thread.

11. Sections

When a section is created, a section object is created. The section object is allocated from paged pool and contains the following information:

- Size of the section in bytes.
- Type of section (page file backed, mapped file, or image file).
- Pointer to the segment structures that describes the prototype PTE and the control area structures for the section.

11.1 Segments

A segment structure is created for every section that is backed by a paging file, and for each section that maps a file that is not already described by a segment. That means that multiple sections that map the same file all refer to the same segment. A segment structure is allocated from paged pool and contains:

- Size of the segment in pages and bytes.
- Type of segment (page file backed, mapped file, or image file)
- Pointer to control area for the segment
- Prototype PTEs which describe the segment

11.2 Segment Control Area

The segment control area maintains, in nonpaged pool, the information required to perform I/O to and from mapped files. The segment control area contains:

- A pointer to the file descriptor for the associated file
- Number of valid/transition pages in the section
- Total number of prototype PTEs in section
- Pointer to section
- One or more subsection descriptors

11.3 Subsection Descriptors

A subsection contains the necessary information to calculate the prototype PTE to logical sector number (LSN) correspondence. It contains:

- A pointer to the segment control area
- Address of first prototype PTE for this subsection

- o Base LSN for this subsection
- o Number of LSNs in this subsection
- o Pointer to any extended subsections

When a prototype PTE refers to a subsection, the logical LSN is calculated by subtracting the address of the prototype PTE from the address of first prototype PTE for this subsection. This result is doubled and added to the base LSN for this subsection. This yields the LSN for the I/O.

The size of the I/O, which is 8 sectors for a full page, is checked against the number of LSNs within this subsection and if a full 8 sectors are not read, the remainder of the page is filled with zeroes.

12. Extending Sections

Sections which map data files may be extended. This is accomplished by comparing the current section size to the requested extension size. If the section size is greater than the requested extension size, no extension is done.

If the section size is less than the extended size, the segment which corresponds to the mapped file is examined. If its size is greater than the requested extension, the section is extended by merely adjusting the size value in the section object. If, however, the segment size is smaller than the requested size, the segment is extended.

Segment extension is accomplished by allocating enough prototype PTEs to map the requested extension rounded up to the next 4mb boundary. This allows multiple small extensions to be made without allocating any additional structures. Associated with the prototype PTEs is a new subsection which points to the segment's control area and the extended prototype PTEs. This subsection is added to the singlely linked list of subsections for the segment. In addition, the segment structures are updated to indicate the large size.

Mapping a view to the extended part of the section no different than mapping a view to a non-extended section, however, when a page fault occurs for a page within the extended part of the section, the subsection list is searched until the subsection which contains the page is located. This is accomplished by calculating the PTE offset from the start of the section and then finding the subsection which contains this offset.

13. Image Activation

Image activation is a multi-step process which consists of open the image file for execute access, creating a section for the image, mapping the image into the address space, performing fixup operations on the image, "activating any DLLs", and starting the image at it's main entry point.

Image activation has the following goals:

- o All executable images are automatically shared among all users of that image.
- o A minimum number of disk I/Os are issued to load and fixup the image.
- o Image fixup operations occur in user mode in the context executing the image.
- o Code, data, and fixup information from images is brought into the address space via the pager. No special code exists to read images into memory.

13.1 Activation Process

The following steps occur to activate an image:

- o Open the image file.

  The image file is opened using the NtOpenFile or NTCreateFile service. The specified desired access is *FILE_EXECUTE* and the file sharing specifies reading with no writers.

- o Create the section.

  A section is created using the NtCreateSection service passing in the file handle for the image and an attribute indicating the file should be treated as an image rather than as a data file.

- o Create a new process.

  A new process is created specifying the section handle of the mapped image file. When the new process is created, the image file will be mapped in the user part of the address space at its specified base address. If this image cannot be mapped at the specified base address, the process creation fails.

  During the process creation phase, the system DLL (UDLL.DLL) is also mapped into the address space of the newly created process.

- o Thread startup.

The initial thread in the process is created with a context that will cause it to start at the entry-point specified in the image header. The routine **RtlCreateUserProcess** is provided to perform the above steps.

The executive starts all user-mode threads with a user-mode APC targeted at **LdrInitializeThunk**. This procedure exists in the UDLL.DLL mapped in all address spaces. The **LdrInitializeThunk** routine does the following:

- Determines if this is the initial thread of a "non-forked" process. If this is the not the case, then return as initialization has already been performed. Note, in the future, initialization of such things as thread local storage, or other thread specific initialization would be performed at this time.

- DLL initialization is now performed. The PEB contains the address of the image header for the initial image. Using the image header, the image's DLL table is located and for each DLL referenced by the image, the following steps occur:

  1. If the DLL is already mapped, it doesn't need to be mapped again, skips steps 2 through 4.

  2. Open the file for the DLL image.

  3. Create a section for the DLL image.

  4. Map the DLL image into the address space. If the DLL could not be mapped at its base address, fixups are performed on the DLL image.

  5. For each entry-point used by the main image, resolve references to the mapped DLL.

  6. Examine the newly loaded DLL image for DLL and resolve any external references using these same steps.

  7. If the DLL contains an initialization routine, call the specified initialization routine.

  8. Repeat until all DLL references are resolved.

- At this point all static DLL references have been resolved and **LdrInitializeThunk** returns.

o The thread context is restored and the APC is dismissed. The thread begins execution at the specified address.

13.2 Create Section Operation For Images

Once the image file has been opened, the **NtCreateSection** service is invoked to create the necessary structures to allow the image to mapped and shared between multiple address spaces.

Create section performs the following:

1. Translate the file handle to an object pointer with execute access.
2. Check to see if the section field in the file object is NULL. If the section field is not NULL, a section object for this file already exists and the section has already been created and may be shared (don't do the rest of these steps).
3. Allocate a page of memory and read in the first 4k bytes of the image header (the image header may be less than 4k bytes and the file may be less than 4k bytes, but this read will produce the correct results).
4. Analyze the image header to ensure that the file is a valid image.
5. Using the information found in the image header, create the prototype PTEs, the subsection descriptors, update the section field in the file object, and create the section object.
6. Return the appropriate status and section handle (if successful) to the caller.

13.3 Map view of section

The **NtMapViewOfSection** maps the specified section into the specified address space and returns the base address of the section. For images, the image header mapped is at the returned base address. This allows the image header to be analyzed using offsets from the base address. An attempt is made to map the image at its specified base address thereby eliminating the need for internal fixups on the image.

From the image header information for DLLs, fixups, entry point, debugger, etc. may be obtained.

13.4 Image Fixup

The image file contains the necessary fixup information to resolve internal image addresses if the image cannot be based at the specified address. The information is mapped in the *TBD* section which can be located using the image header.

Image fixup is accomplished by mapping the pages to be modified as read/write, and changing the specified addresses by the difference between the desired base address and the actual base address. Once the fixups have been performed the page protection is changed back to its previous protection.

14. Virtual Address Descriptors

Whenever a range of virtual addresses is created (committed or reserved), a virtual address descriptor is built. A virtual address descriptor contains the following information:

1. Virtual address range for this descriptor
2. Section information, if any
3. Attributes for range including protection and inheritance
4. Link words to form an ordered set of virtual address descriptors

As pages within the virtual address descriptor are referenced, page table entries (which are initially zero) are updated to map the appropriate page.

15. POSIX Fork Support

POSIX fork requires that a new process (the child) be created with its virtual address mappings and contents identical to the process that initiated the fork (the parent). In order to provide efficient fork operation, pages that are shared between the parent and child are shared copy-on-write.

15.1 Structures to Support Fork

There are three structures created during a fork operation:

- o The *fork prototype PTEs* that describe the unique address space that is shared between the parent and the child. A fork prototype PTE is a simple structure that contains a prototype PTE and a reference count indicating the number of processes that refer to the prototype PTE. When the reference count is decremented to zero, any resources (paging file space, physical page) are released.
- o The *fork header* is a system-wide structure allocated from non-paged pool that contains the size and address of the fork prototype PTEs created during the fork operation. It also contains a reference count indicating the number of processes that refer to at least one of the fork prototype PTEs. When the reference count becomes zero, all the fork prototype PTEs and the associated fork header are deleted.

- The *fork descriptor* is a process structure allocated from non-paged pool that contains a pointer to the fork header, the starting and ending address of the fork prototype PTEs, and a reference count indicating the number of PTEs that refer to fork prototype PTEs. When the reference count in the fork descriptor becomes zero, the fork descriptor is deleted and the reference count in the fork header is decremented.

15.2 Fork Operation

When a fork operation is performed, the following steps occur:

- A new address space is created containing a page directory page, hyper space structures and the mapping of the non-paged executive.
- An attach process is issued with the parent process as the target.
- The parent process's virtual address descriptors (VADs) are examined.
- If the VAD indicates the range is inherit on fork and the VAD is not a section with the PAG_COPY attribute, do the following:
  - Allocate a new VAD from non-paged pool and initialize it as a copy of the parent's VAD.
  - Examine the page directory entry in the child process for the starting virtual address for the memory described by the VAD. If no PDE is allocated, get a free page of memory, otherwise, use the page which has previously been allocated.
  - Examine the PTEs described by the VAD and for each PTE take the following action:
    - Valid - Examine the PFN element to assess the PTE type.
      - If it is a prototype PTE and it was not created from a previous fork operation, put the prototype PTE address into the PFN element preserving the current protection values.
      - If it is a fork prototype PTE, put the prototype PTE address into the PFN element preserving the current protection values and increment the reference count for that fork prototype PTE.
      - If it is not a prototype PTE, allocate a fork prototype PTE, put the new PTE into prototype PTE format, set the reference count for the

fork prototype PTE to 2, and change the PFN element to refer to the "fork prototype PTE."

- o Transition

    - Acquire the PFN lock and check the PTE again. If it is still in transition, this page is private and therefore, must be converted into "fork prototype PTE format". This is identical to the valid case, except the parent's PTE is changed to refer to the fork prototype PTE which is placed into the transition state.

- o Demand Zero

    - Make the corresponding PTE in the child process demand zero.

- o Page File Format

    - Allocate a fork prototype PTE, set the reference count to 2, move the page file format PTE into the fork prototype PTE and put the address of the fork prototype PTE into the PTE of both the parent and the child.

- o Prototype PTE Format

    - Copy the PTE contents to the child process and check to see if this is fork prototype PTE format, if so increment the reference count for the fork prototype PTE.

- o If the VAD indicates PAG_COPY, each committed page of the section must be copied into a private page. This is accomplished by creating a VAD to describe the section as a copy-on-write section. Later, in the context of the child process, all pages in the section are read and written, causing a private copy of the page to be created.

- o Once all the VADs have been processed, detach from the parent process and attach to the child process. In the context of the child process, build the structures to manage the VADs, fork descriptors, and force copy-on-write actions to all pages in any PAG_COPY sections.

Note that at no time during the processing of the VADs to build the PTEs in the child process was a PTE created in either the valid or transition state. By avoiding this condition no updates need occur to the working set child while attached to the parent process.

16. Working Set Management

Associated with every process is a working set. Two parameters control the size of the working set. The *WorkingSetMinimum* is the minimum number of pages guaranteed to be resident or made resident when any thread within the process is executing. The other parameter is the *WorkingSetMaximum* which is the maximum number of pages which the process is allowed to have resident. The *WorkingSetMaximum* can be exceeded if there is a large number of free pages available.

Each page fault requires a page to become resident to satisfy the request. The *working set list* is used to track the current number of pages a process has resident in memory. As pages are made valid, the page is added to the working set. When the working set reaches its limit, for every page added, a page is discarded. (Note, that due to cache flushes, etc. it may be the case that a set of pages is removed from the working set when a new page is added. This allows a number of page faults to occur without each one having to invalidate TBs and data caches.) The discarded page(s) are not removed from memory immediately, but rather, are placed on the modified or standby lists. By placing pages on the lists, if a fault occurs for the page while it is still on the list, the fault can be satisfied by removing the page from the list and placing it back into the working set.

A working set is an array of working set list entries (WSLE), with each WSLE describing one resident page of the address space. Associated with the working set list are two indexes. One index is for the start of the dynamic region (all working set list entries before that index are locked in the working set), and the other is the end of the working set list.

Replacement is performed by keeping an index into the dynamic portion of the list. When a page needs to be removed from the working set, the WSLE found by this index becomes the candidate for removal. If, after closer examination, this WLSE should be removed, then it is removed and the index is incremented. If it is not a proper candidate for replacement, the index is incremented and the next WSLE becomes the candidate for replacement. After the last WSLE in the array is examined, the index is changed to reference the first entry in the dynamic portion of the working set.

A WSLE which has been selected as a candidate for removal from the working set has the access bit in its corresponding PTE examined. If the access bit is currently set, it is cleared and the WSLE is not removed. If the access bit is clear, the WSLE is removed from the working set and placed on the modified list or standby list.

In order to limit the time to find a WSLE to replace, if a certain number, say 16, WSLEs are examined for replacement and a suitable candidate cannot be found, the first WSLE examined is the one removed from the working set and the index is adjusted accordingly.

A working set list entry is created for every valid pageable PTE. The working set list entry is 64 bits in size and consists of:

1. Virtual Page Number of this element (20 bits)
2. Attributes (locked in working set, valid, etc.) (12 bits)
3. Link words (16 bits each) to link working set entries together into a sorted list by virtual page number. If both link words are zero, the entry is directly indexed through the PFN working set index hint field and not in the tree.

Because the link words are 16 bits, the maximum size of a working set is limited to 65,535 entries. This means the largest amount of physical memory a process can consume is 512 MB. This does not limit the virtual size of a process. In addition, when this limit becomes a factor, the WSLE will be changed to support a larger working set.

17. Physical Page Management

The memory management subsystem maintains a working set for each process so that physical memory is shared equitably among all the active processes on the system. As the demands for physical memory increase, and the free and zeroed list becomes empty, free pages are obtained by the following methods:

- Removing pages from the standby list
- Writing pages on the modified list and placing those pages on the standby list
- Reducing the size of working sets thereby placing pages on the modified and standby list
- Eliminating processes from memory by reducing the size of their working sets to zero

17.1 Modified Page Writer

The modified page writer is a system thread created during system initialization. The modified page writer is responsible for limiting the size of the modified page list by writing pages to their backing store locations when the list becomes too big.

When invoked, the modified page writer attempts to write as many pages as possible to backing store with a single I/O request. This is accomplished by examining the original PTE field of the PFN database elements for pages on the modified page list to locate pages in contiguous locations in the backing store. Once a list is created, the pages are removed from the modified list, an I/O request is issued, and, at successful completion of the I/O request, the pages are placed at the tail of the standby list.

Pages which are in the process of being written may be faulted back into memory. This is accomplished by incrementing both the *ReferenceCount* and *ShareCount* for the physical page. When the I/O completes, the modified page writer notices that the *ShareCount* is no longer zero, and does not place the page on the standby list.

17.2 Balance Set Manager

The balance set manager is created during system initialization and is responsible for trimming process working sets when physical memory becomes over-committed.

Each process has a minimum working set which is guaranteed to be available when any thread within the process is executing. When the balance set manager is invoked, it creates free pages by reducing working sets towards their minimum size by starting with those processes which have the lowest page fault rates.

If all working sets are reduced to their minimum, and physical memory is still over-committed, the balance set manager removes processes from the system. This is accomplished by selecting a process to eliminate, and calling the kernel function *KeExcludeProcess.* Once the kernel has prevented the process from executing, the balance set manager reduces the process's working set to zero causing pages to be placed on the modified and standby lists.

Once a process has been excluded from the balance set, it becomes eligible for inclusion when either ample physical memory exists for its minimum working set, or threads within the process become computable and the priority is such that the threads should be allowed to execute. In the later case, another process(es) may be removed from the balance set to free ample physical pages.

When the system reaches the state that physical memory is so severely over-committed that processes must be removed from the balance set to allow non-resident processes to be included in the balance set, the system performance suffers greatly. This case should be avoided in all but the most extreme cases, i.e. the amount of physical memory is inadequate for the workload.

Once memory again becomes plentiful the pager allows working sets to expand above their minimum. This is accomplished by satisfying page faults without removing pages from the working set list.

18. I/O Support

The memory management subsystem provides services for operating on virtual memory to support I/O operations. The following operations are available:

- Probing pages for access
- Locking pages in memory
- Unlocking pages from memory
- Mapping locked pages into the current address space
- Unmapping locked pages from the current address space
- Getting the physical address of a locked page
- Mapping I/O space
- Unmapping I/O space

These operations are available from kernel mode at **IRQL 0** or **APC_LEVEL**.

18.1 Locking Pages in Memory

A range of virtual addresses for the current process are checked for proper accessibility and locked in physical memory with the **MmProbeAndLockPages** function:

VOID
MmProbeAndLockPages (
    IN PMDL *MemoryDescriptionList*,
    IN ULONG *AccessMode,*
    IN ULONG *Operation*
    );

Parameters:

MemoryDescriptionList - A pointer to a memory description list which contains the starting virtual address to lock, the size in bytes of the region to lock, and an array of elements that are to be filled with physical page numbers.

AccessMode - Specifies the access mode with which to probe the region (*UserMode* or *KernelMode*).

Operation - Specifies the type of the I/O operation (*IoWriteAccess, IoReadAccess* or *IoModifyAccess*).

This function probes the specified range for access and locks the pages in memory by making any nonresident pages resident and incrementing the *ReferenceCount* in the PFN database for the physical page. Incrementing the *ReferenceCount* prevents the physical page from being reused.

If any pages are found with improper access protection, or the processes working set is not sufficient to lock the range in memory, an exception is raised and none of the pages are locked in memory.

18.2 Unlocking Pages from Memory

Pages which have been locked in memory are unlocked with the **MmUnlockPages** function:

```
VOID
MmUnlockPages (
    IN PMDL MemoryDescriptionList
    );
```

Parameters:

MemoryDescriptionList - A pointer to an memory definition list containing the information about locked pages.

This function analyzes the *MemoryDescriptionList* and unlocks any pages which have been locked. This function is callable within any process's context.

18.3 Mapping Locked Pages into the Current Address Space

Once a page has been locked into memory, the **MmMapLockedPages** function maps the physical pages into the address space of the current process. This provides a mechanism for system processes to virtually address the physical memory within another process.

PVOID
MmMapLockedPages (
    IN PMDL *MemoryDescriptionList*
    IN KPROCESSOR_MODE *AccessMode*
    );

Parameters:

MemoryDescriptorList - Supplies a valid Memory Descriptor List which has been updated by MmProbeAndLockPages.

AccessMode - Supplies an indicator of where to map the pages; KernelMode indicates that the pages should be mapped in the system part of the address space, UserMode indicates the pages should be mapped in the user part of the address space.

Returns the base address where the pages are mapped. The base address has the same offset as the virtual address in the MDL.

This routine will raise an exception if the processor mode is USER_MODE and quota limits or VM limits are exceeded.

18.4 Unmapping Locked Pages from the Current Address Space

Pages which have been mapped into the current process's address space are unmapped with the **MmUnmapLockedPages** function:

VOID
MmUnmapLockedPages (
    IN PVOID *BaseAddress,*
    IN PMDL *MemoryDescriptionList*
    );

Parameters:

BaseAddress - The base address where the pages are mapped.

MemoryDescriptionList - A pointer to an memory definition list containing the information about locked pages.

This function unmaps the pages which were previously mapped. Once the pages have been unmapped, they may be unlocked.

If the *MemoryDescriptionList* indicates the pages are not locked an exception is raised.

18.5 Mapping I/O Space

Physical addresses residing in the processors I/O space can be mapped to virtual addresses within the nonpagable portion of the system with the **MmMapIoSpace** function:

```
PVOID
MmMapIoSpace (
    IN PHYSICAL_ADDRESS PhysicalAddress,
    IN ULONG NumberOfBytes
    );
```

Parameters:

PhysicalAddress - The physical address within I/O space to map.

NumberOfBytes - The number of bytes to map.

This function returns the base virtual address where the requested I/O space was mapped.

18.6 Unmapping I/O Space

Physical addresses residing in the processors I/O space can be unmapped from virtual addresses in the nonpagable portion of the system with the **MmUnmapIoSpace** function:

```
VOID
MmUnmapIoSpace (
    IN PVOID BaseAddress,
    IN ULONG NumberOfBytes
    );
```

Parameters:

BaseAddress - The virtual address within system space to unmap.

NumberOfBytes - The number of bytes to unmap.

18.7 Get Physical Address

The physical address mapped by a virtual address which has been locked in memory may be obtained with the **MmGetPhysicalAddress** function:

PHYSICAL_ADDRESS
    MmGetPhysicalAddress (
    IN PVOID *BaseAddress*
    );

__**Parameters:**__

BaseAddress - Specifies the virtual address of which to provide the physical address.

This function returns the physical address of the page mapped by the specified virtual address.

19. File System Caching Support

A 256MB region of system space is reserved for file system caching support. This region has the following characteristics:

- Not accessible from user-mode.
- Pagable, but has a system-wide working set. This means that if a page is valid in the "system cache" it is valid in any address space. Note, however, that the page could be removed from the working set of the system cache at any time unless it has been explicitly "locked" in the system cache.
- Only views of mapped files may reside in the cache.
- Addresses within the system cache may not be used as arguments for the BaseAddress within NT memory management services, e.g., **NtLockVirtualMemory** supplying the base address argument as an address that resides in the cache.

The following routines are provided to the file systems and server for interacting with the system cache.

19.1 Mapping a View in the Cache

A view to a section can be mapped in the system cache with the **MmMapViewInSystemCache** function:

NTSTATUS
MmMapViewInSystemCache (
    IN PVOID *SectionObject,*
    OUT PVOID **CapturedBase,*
    IN OUT PLARGE_INTEGER *SectionOffset,*
    IN OUT PULONG *CapturedViewSize,*
    IN OUT ULONG *Protect*
    );

Parameters:

SectionObject - An pointer to a section object.

CaputuredAddress - A pointer to a variable that will receive the base address of the view.

SectionOffset - Supplies a pointer to the offset from the beginning of the section to the view in bytes. This value is rounded down to the next allocation granularity size boundary.

CapturedViewSize - A pointer to a variable that will receive the actual size in bytes of the view. If the value of this argument is zero, then a view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise the initial value of this argument specifies the size of the view in bytes and is rounded up to the next host page size boundary.

Protect - The protection desired for the region of initially committed pages.

19.2 Unmapping a View from the Cache

NTSTATUS
NtUnmapViewInSystemCache (
    IN PVOID *BaseAddress*
    );

Parameters:

BaseAddress - A virtual address within the view which is to be unmapped.

18.3 Check and Lock Pages

A range of valid (or transition if the virtual address resides within the system cache) virtual addresses may be locked in memory with the check and lock pages function.

```
ULONG
MmCheckAndLockPages
    IN PEPROCESS Process,
    IN PVOID BaseAddress,
    IN ULONG SizeToLock
    );
```

Parameters:

Process - Supplies a pointer to the process in which these pages are mapped.

BaseAddress - A virtual address within the system cache to begin locking.

SizeToLock - The number of bytes to attempt to lock in the system cache.

The returned value is the number of bytes that were actually locked.

This routine checks to see if the specified pages are resident and if so increments the reference count for the page. For addresses within the system cache, the virtual address is guaranteed to be valid until the pages are unlocked (the reference count for the page becomes zero), for pages not residing in the system cache, the physical page is resident, but a page-fault could occur when referencing this address (though no I/O operation will result from this page fault).

NOTE, this routine is not to be used for general locking of user addresses - use **MmProbeAndLockPages**. Rather, this routine is intended for well file system caches which maps views of files into the address range and guarantees that the mapping will not be modified (deleted or changed) while the pages are mapped.

This routine may be called at DPC_LEVEL and below. If the base address is not within the system cache and the IRQL is at DPC_LEVEL, no pages will be locked and a zero will be returned.

19.3 Unlock Checked Pages

A range of addresses locked in memory with the **MmCheckAndLockPages** function is unlocked with the **MmUnlockCheckedPages**.

VOID
MmUnlockCheckedPages
    IN PEPROCESS *Process,*
    IN PVOID *BaseAddress,*
    IN ULONG *SizeToUnlock*
    );

Parameters:

Process - Supplies a pointer to the process in which these pages are mapped.

BaseAddress - A virtual address within the system cache to begin unlocking.

SizeToUnlock - The number of bytes to attempt to lock in the system cache, this was the function value returned when the range was locked.

If the base address is within the system cache, this routine may be called at DPC_LEVEL or below. If the base address is not within the system cache, it may be called at APC_LEVEL or below.

19.4 Read Mapped File

A range of virtual memory can be made valid with the **MmReadMappedFile** function:

VOID
MmReadMappedFile
    IN PEPROCESS *Process,*
    IN PVOID *BaseAddress,*
    IN ULONG *Size,*
    IN PIOSTATUS_BLOCK *IoStatus*
    );

Parameters:

Process - Supplies a pointer to the process in which these pages are mapped.

BaseAddress - Supplies the virtual address within the system cache to begin reading.

Size - Supplies the number of bytes to read.

IoStatus - Supplies the I/O status value from the in-page operation.

This function checks the corresponding PTEs and makes the specified range valid with a minimum number of I/O operations. Any errors which occur during the in-page sequence are returned in the *IoStatus* argument.

19.5 Purge Section

To support file truncation, pages within a section can be cleared with the **MmPurgeSection** function:

```
BOOLEAN
MmPurgeSection (
    IN PVOID SectionObject,
    IN PLARGE_INTEGER Offset
    );
```

Parameters:

SectionObject - Supplies a pointer to the section object which to purge.

Offset - Supplies the offset into the file where the purge should begin.

This function examines the prototype PTEs beginning at the specified offset. If the PTE is active and valid, a bugcheck code is issued. If the PTE is transition, the page is put on the free list and the prototype PTE is loaded with the original contents from the PFN database.

19.6 Force Section Closed

A section can be disassociated from a file object with the **MmForceSectionClosed** function:

```
BOOLEAN
MmForceSectionClosed
    IN POBJECT_FILE FileObject
    );
```

Parameters:

FileObject - Supplies a pointer to the file object to check for a section and attempt to close and remove the section reference.

If the section cannot be closed due to outstanding references or mapped view, the value FALSE is returned and no action is taken. If the section was successfully closed, the value TRUE is returned.

Revision History:

Original Draft 1.0, January 6, 1989

Revision 1.1, January 20, 1989

1. Fix PTE format to include prototype PTEs.
2. Add virtual address descriptors.
3. General reorganization.

Revision 1.2, March 31, 1989

1. Add I/O routines.
2. Make PPTN database 20 bytes rather than 24.

Revision 1.3, May 3, 1989

1. Make PPTN database 24 bytes rather than 20, the PTE address field cannot overlay the Flink field.
2. Change size of last reserved page of user address space to 64k bytes.
3. Change name of I/O support routines.

Revision 1.5, August 10, 1898

1. Add description of fork structures and operation.
2. Change description of section to add description of segment object.

Revision 1.6, October 25, 1989

1. Clarify modified page writer.

Revision 1.7, July 10, 1990

1. PFN mutex is now an executive spinlock.
2. Add section detailing overview of how create and map file interact with image activiation.
3. Add section on extending mapped sections.
4. Add section on system wide cache.

Revision 1.8, July 25, 1990

1. At working set index hint field in PFN database.

[end of vmdesign.doc]