Virtual Memory: Windows NT® Implementation

Submitted By: Jacob Buysse
Submitted To: Dr. Barnicki
Submitted On: January 27, 2000

Table of Contents

Table of Figures

Application Memory

From an applications point of view, the system has 4GB of memory (Kath). This can be determined from the 32-bit pointers having a range from 0000-0000 to FFFF-FFFF. The upper portion of this "virtual memory" is reserved by the system while the rest is open for the application to use. On most personal computers, nowhere near 4GB of memory is installed. The operating system works around this problem by providing support for virtual addressing and mapping the virtual pointers a program uses to physical memory.

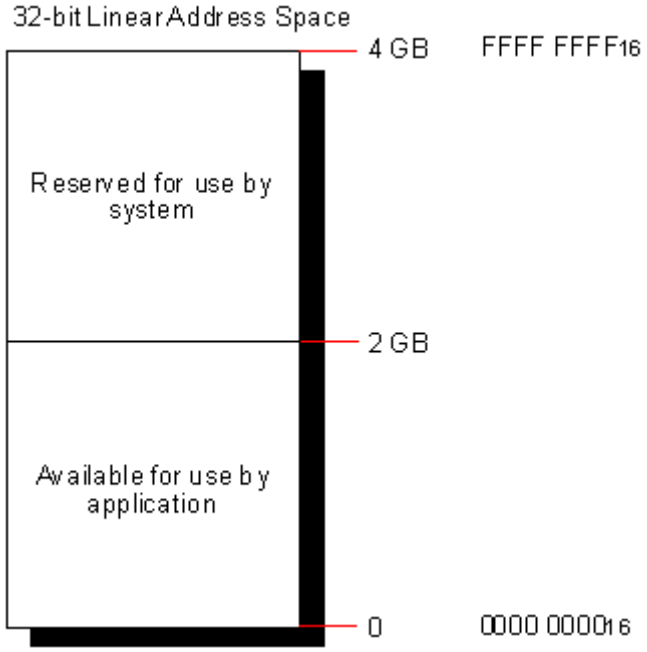

Figure 1 (Kath)

Entire program or portions of them can be swapped in and out of memory to mimic having the full 4GB of memory. However, there is a performance hit that is taken when doing the swapping and some programs cannot handle this kind of a timing issue.

These are mostly the system processes like the memory manager itself and all kernel threads. There is an area in the system memory where nonpaged pools of memory exist (Solomon). This keeps the memory manager from getting swapped out and then, "Whoops! What now".

Paging Model

In order to map 4GB of memory, there needs to be some sort of reference table the operating system can use to find the corresponding physical address. The hierarchy used by Windows NT® is a three (and sometimes four) tier system. At the top level, there is a Page Directory. The physical address of this is always stored in the CPU register CR3. Each process gets its own directory. The directory is a page itself, 4096 bytes long (4K). It contains 1024 32-bit entries, each of which point to a Page Table.

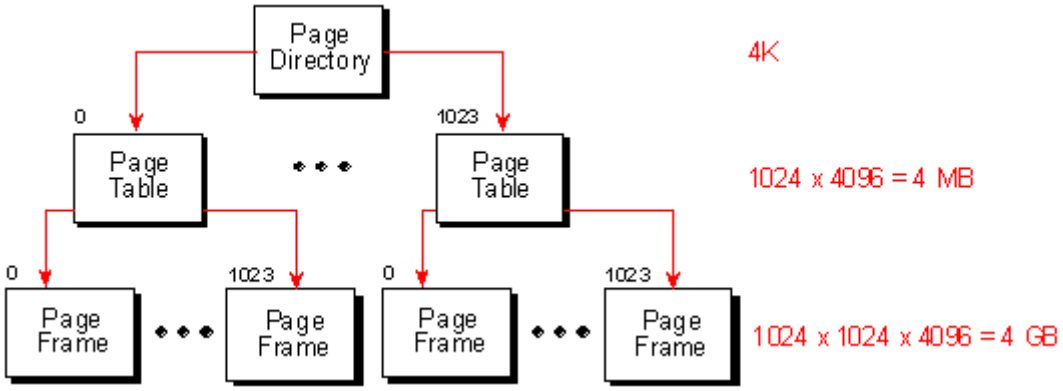

Figure 2 (Kath)

The page table is also a 4K page with 1024 entries. Each of these entries is a Page Table Entry (PTE) which points to a Page Frame (what is though of when a page is

referred to). The page frame is the 4K of physical memory for that page. The entire 4GB can be now accounted for in only about 4M of memory (Kath).

Address Translation

The question now is how to convert from a 32-bit pointer which is pointing somewhere in virtual memory to a physical memory address using the paging hierarchy described above. First we analyze the division of the 32-bit pointer into three segments: the Page Directory Offset, the Page Table Offset, and the Physical Page Offset.

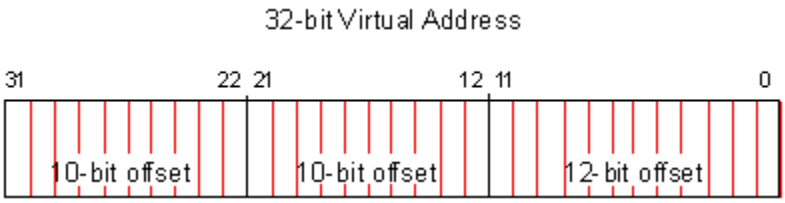

Figure 3 (Kath)

The 10-bit directory offset is shifted left by two (or multiplied by four) to give a DWORD address into the page directory (000 – FFF). This entry points to a page table. We then use the 10-bit page table offset shifted by two to get the DWORD address into the page table. This entry points to a page frame. The final 12-bit value is used to offset the frame in a BYTE format (Kath).

The page table entry is where the logic is performed on whether the referenced memory is both valid and in physical memory. The upper five bites are used for security. They specify whether the process has read, write, readwrite, or no access to the page. The next 20 bits are the page frame address. The next four bits specify the pagefile that

backs this page of memory and the last 3 bits are used to determine the state of this page. The first of those bits is the transition bit. This bit is a one when the page is being placed into physical memory or saved out to the physical disk. The next bit is the dirty bit specifying whether this memory has been modified since it was loaded. The final bit is the valid bit. When this bit is set, the page is a valid page to reference and is in physical memory. However, when this bit is not set, the page is invalid and a page fault occurs (Kath).

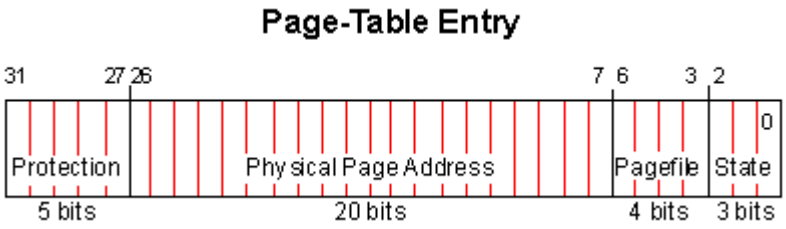

Figure 4 (Kath)

Page Faults

Whenever a page is accessed that is not currently loaded into memory, a page fault exception occurs. There is a system trap that begins which will attempt to load the page from disk into memory. This handler is known as the pager process. The pager will verify that the address is valid to access (i.e. – there has been some memory allocated in that region for this process and it isn't referencing memory that it does not own) and that it can be loaded. Once it is loaded, the pager will return control to the OS and restart the instruction that caused the page fault in the first place.

An instruction that causes a page fault causes the program to start getting performance losses. A single instruction can cause more that just one page fault. The

worst case scenario for any given instruction is that it will need to load the page directory, the page table, and that physical memory page all in one call. This can happen if some other process has paged out our memory pages.

In order for processes to share memory, they need a common page frame between them. Since it would be another performance hit to update multiple page table entries whenever a page frame was modified, Windows NT® came up with a different strategy for handling inter-process memory. The operating system will allocate a Prototype Page Table Entry that points to the shared Page Frames (Kath). This way only the single prototype page table entry needs to be updated when the shared memory is modified. This shared memory also introduces the possibility of four page faults for a given instruction, with the additional fault being the prototype page table entry.

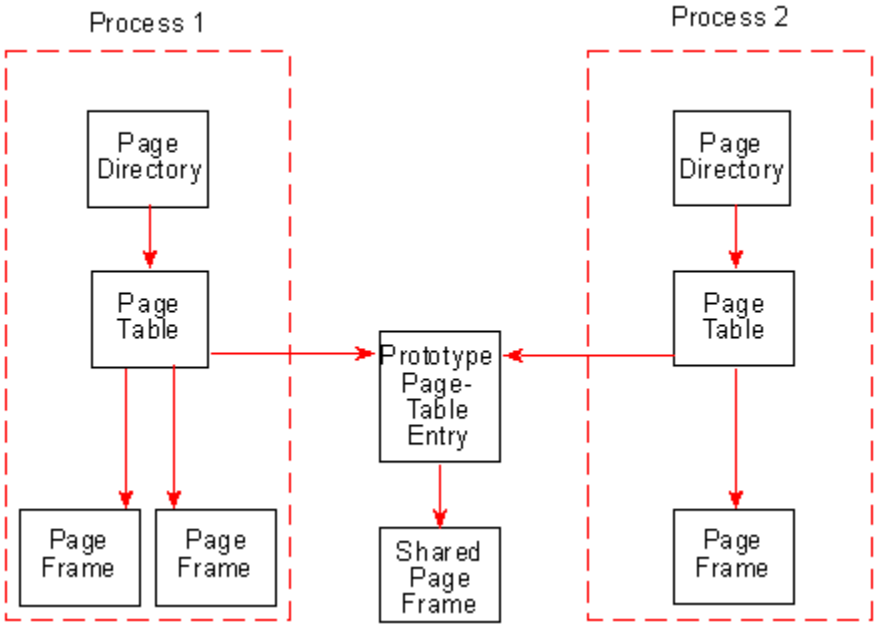

Figure 5 (Kath)

Windows NT® Paging Database

Since not all pages are allocated when a process is created to save memory, there needs to be some sort of recycling program and allocation program going on behind the scenes. Microsoft uses a paging database to keep track of all of the pages that exist. The pages are separated into 8 different categories: Active (or Valid), Transition, Standby, Modified, Modified no write, Free, Zeroed, and Bad (Solomen).

Active pages are pages that are loaded into memory and are part of the working set (described later in this section). These are the pages that are pointed to by all page table entries (PTEs) and each one has a valid PTE pointing to it.

The transition state is a temporary state for a page while it is not active and not on a paging list. This is where some sort of I/O operation is being performed such as reading from the disk or committing to the disk.

Modified pages were just active but were modified (or dirty) and have not been written to the disk yet. They are marked as invalid and in transaction.

Modified no write pages are used by NTFS so that the transaction logs are written before the pages are flushed to disk. NTFS is an entirely different subject so I will refrain from going into detail.

Free pages are pages that have been removed from the active status, but have not been zeroed yet. Zeroed pages are free pages that have been completely written over with zeros. Only read-only pages and kernel threads can allocate free pages while standard new page allocations require zeroed memory.

Bad pages have caused hardware errors or have general parity issues and can no longer be used by any process. They are only mentioned since they never go back into the free or zeroed pools.

By definition, all active pages are managed by the system working set. The active paged pool is not the only memory that is managed, there are also system cache pages, pagable kernel code, pagable device drivers, and system mapped views. The paged pool has pages added and removed according to the working set process. The working set processes are covered in more detail in the next section.

Virtual Memory Manager

The virtual memory manager handles the management of the paging database. There are six threads in all that comprise the memory manager process, all of which reside in the NTOSKRNL.EXE file.

The balance set manager monitors and manages all of the other threads in the process. It will handle the working set trimming which shrinks the number of allocated pages in the working set to free some pages for allocation. It also takes care of aging pages so they can be swapped properly. Window NT® uses a derivation of the First in, First out (FIFO) replacement algorithm that uses the age of the page to determine which is the oldest page (Solomon). It also attempts to minimize page faults by loading adjacent pages whenever a page fault occurs. So if page 6 is being loaded, pages 4,5,7, and 8 might be loaded along with it. The set manager also schedules the modified page writer whenever there are no free or zeroed pages and there are modified pages. This thread has a thread priority of 16 (Solomen).

The process/stack swapper performs thread and kernel thread stack inswapping and outswapping. This is done when the thread scheduling code is context switching to a new thread so each process' stack is properly preserved between switches. This thread has a thread priority of 23 (Solomen).

The modified page writer controls writing pages in the modified list to the paging files. It is awakened when the modified list of pages needs to be reduced for free pages. This thread has a thread priority of 17 (Solomen).

The mapped page writer is the backup writer and writes dirty mapped files to disk. This is in place to prevent deadlock when the modified page writer causes a page fault since it is being run to free a page for the set manager. This thread also has a priority of 17 (Solomen).

The next thread is the dereference segment thread. This thread controls the system cache and manages the page file growth upon request. It has a priority of 18 (Solomen).

The last thread is only run when necessary of when nothing else is running. It is the zero thread. Its purpose is solely to go through the free page list and zero them out so they can go in the zeroed page list. Therefore this thread has a priority of 0 (Solomen).

All of these processes work with the paging database structures. The database is set up as series of linked lists whose members are the pages. A linked list is kept for each type of page: Active, Modified, Standby, Free, Zeroed, and Bad. The standby list exists for a good reason. If modified pages were sent directly to the free list when flushed, then when a page fault came along to re-request that page, a hard page fault would occur and the memory would need to be read from disk. However, by keeping a temporary storage

or recently used pages, several hard page faults can be avoided and replaced with “soft” page faults (Solomen).

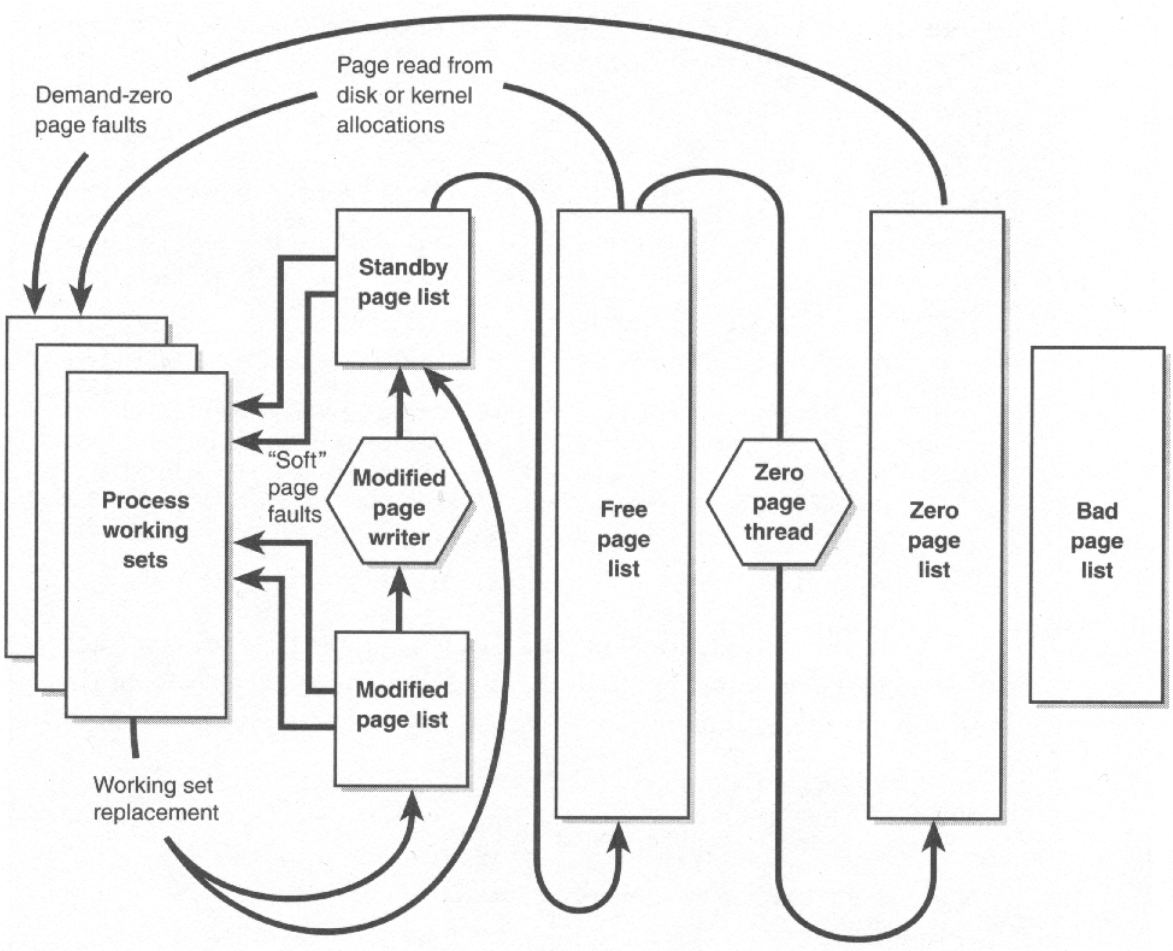

Figure 6 (Solomen)

Performance

There has been a lot of talk about the demand paging system implemented by Windows NT®, but no talk about how well it performs. There are several reasons that the implementation works slightly better than the theory. The first is a hardware accelerated bonus, the Translation Look-Ahead Buffer (TLB) and the second is context switching theory.

The TLB is an on-chip cache that holds 32 values. Each of these values maps a recently used virtual page table to the physical memory. The current virtual address can be compared to all 32 values in parallel since it is hardware supported providing a performance boost over a software implementation (Galvin). If the page table has been recently used, then the physical offset can be directly applied to the physical page that is also stored in the TLB (Kath). This saves a lookup in both the directory and the table (both of which could have caused page faults). Since the TLB can only have 32 entries, whenever a new page is loaded, the oldest page in the buffer is removed (using the aging information the set manager provides).

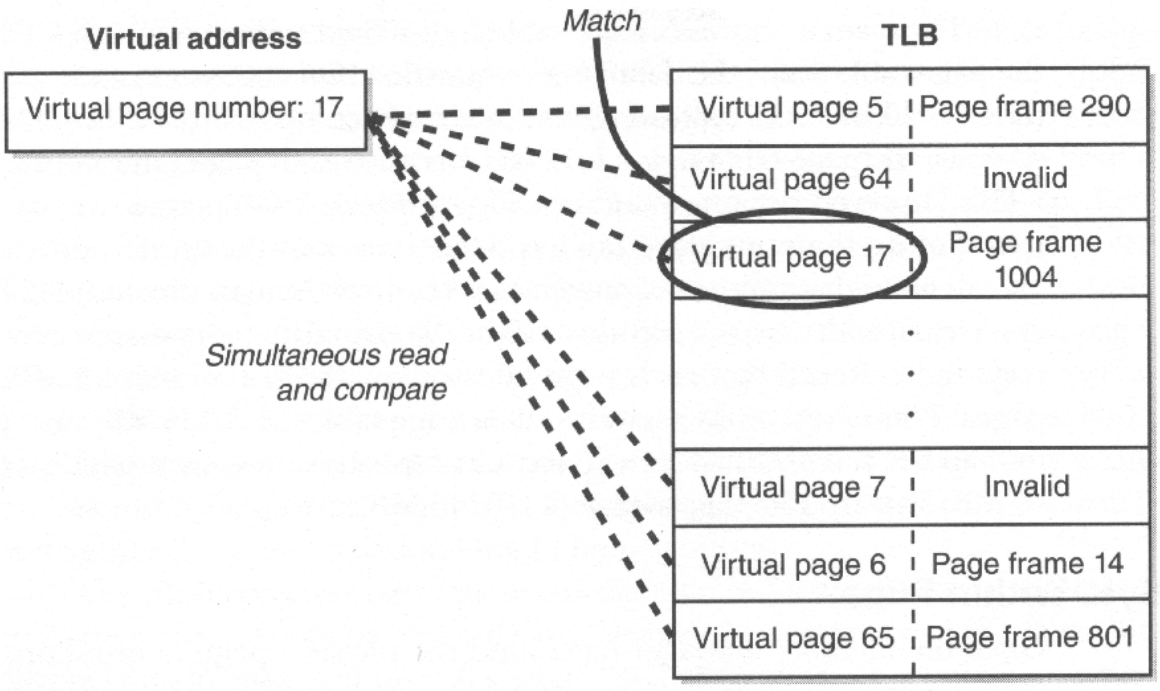

Figure 7 (Solomen)

This is great since most memory references are localized within a page, or within enough pages that could fit onto the TLB. But when multitasking is introduced, the

theory is broken. When a new thread is executed, then the odds that it will be referencing the same page tables as the previous thread are extremely low, causing the entire TLB to be useless for 32 more page faults (Solomen). So every context switch the TLB is useless.

Now look at the problem in a more specific sense and take the x86 platform for an example. An average time slice (the duration between thread context switches) is around 17ms. Given a 5 million instructions per second (MIPS) processor (in perspective, a 33MHz Intel processor runs around 10-15 MIPS) running a program that has about 30% - 50% of its instructions as memory accesses, then you get between 25,500 and 42,500 address instructions per context switch (Kath). Since the TLB will be refilled with useful pages after 32 distinct page references, it will return to full operation soon after the context switch takes place.

Conclusion

The virtual memory management is a key system in the operating system. All user applications and even the operating system itself rely on proper and efficient operation of demand paging. To ensure that performance is not affected under normal operating conditions, an efficient databasing structure is in place along with a plethora of threads to handle the workload. All of these things combined help to create a dependable operating system.

Works Cited

Galvin, Peter B. and Abraham Silberschatz, 1999, Operating System Concepts, 5th Edition: pages 289-336.

Kath, Randy. December 21, 1992. The Virtual-Memory Manager in Windows NT. **http://msdn.microsoft.com/library/techart/msdn_ntvmm.htm**.

Solomen, David A. 1998. Inside Windows NT, 2nd Edition. Microsoft Press, Redmond, WA: 217-304.

Bibliography

Bobbin, Jill, Prescilla McAndrews, and Srinivasa Tata. 1994. Virtual Memory Tutorial. **http://cne.gmu.edu/modules/VM/index.html**.

Galvin, Peter B. and Abraham Silberschatz, 1994, Operating System Concepts, 4th Edition: pages 249-343.

Galvin, Peter B. and Abraham Silberschatz, 1999, Operating System Concepts, 5th Edition: pages 289-336.

Kath, Randy. December 21, 1992. The Virtual-Memory Manager in Windows NT. **http://msdn.microsoft.com/library/techart/msdn_ntvmm.htm**.

Solomen, David A. 1998. Inside Windows NT, 2nd Edition. Microsoft Press, Redmond, WA: pages 217-304.