<u>RESEARCH PAPER</u>

<u>ON</u>

<u>VIRTUAL MEMORY</u>

Submitted To: Dr. C. Taylor
Submitted By: Sumit Sehgal
Date: February 3, 2003

CONTENTS

INTRODUCTION

In today's world, computers have become an integral and inseparable part of our lives. We use them for many different things in many different ways. However, what is the reason for accepting them so much in our lives? The answer is that computers make our job easier and more efficient. Sometimes, it can be very frustrating or stressful to work with computers that don't run as fast as we want them to or they just cannot handle certain processes due to shortage of system resources. When the limitations of the system resources become a major barrier in achieving our maximum productivity, we often consider the apparent ways of upgrading the system, such as switching to a faster CPU, adding more physical memory (RAM), installing utility programs, and so on. This process of making sure that the operating system uses its resources most efficiently is called System Optimization. To facilitate the process of system optimization, the concept of **Virtual Memory** was introduced. A virtual memory system combines paging with automatic swapping. It uses an imaginary storage area (virtual address space) in conjunction with the existing hardware. This allows a program to use a virtual address space that is larger than the physical memory. Along with the ability to execute a process that is not completely in memory, another advantage of this system includes increased multiprogramming feasibility.

HISTORY

The first virtual memory machine was developed in 1959. It was called the one level storage system. Although it was heavily criticized, it spurred many new prototypes during the early 1960's. Before virtual memory could be regarded as a stable entity, many models, experiments, and theories were developed to overcome the numerous problems that were associated with it. Specialized hardware was developed that would take a "virtual" address and translate it into an actual physical address in memory (secondary or primary). The final debate was laid to rest in 1969 when IBM's research team, lead by David Sayre, showed that the virtual memory overlay system worked consistently better than the best manual-controlled systems. By the late 1970's the virtual memory idea had been perfected enough to use in every commercial computer. Virtual Memory was introduced in Personal computers in 1985 when Intel offered virtual memory along with cache in the 386 microprocessor and Microsoft offered multiprogramming in Windows 3.1. Others finally followed and virtual memory found its place in our everyday lives.

CONCEPTS AND IMPLEMENTATIONS OF VIRTUAL MEMORY

Due to the limitations of physical memory, it is increasingly difficult to store multiple processes in memory to facilitate multiprogramming. One of the techniques to solve this problem is virtual memory. It is defined as a technique that allows the execution of processes that may not be completely in memory (Silberschatz & Galvin). Virtual memory separates logical memory and physical memory. From this separation, the amount of available physical memory is no longer a constraint and less physical memory is needed for each program, thus increasing CPU utilization and throughput, and little if no change in response or turnaround time. Each process has a virtual address, which is used to map the process into main memory. The process can access its data with the virtual address space. On the other hand, the available range of actual memory is known as the physical address space. And the addresses available in main memory are called physical addresses. When executing a process, the virtual address space must be mapped into a physical location. This can be seen below in Figure 1.

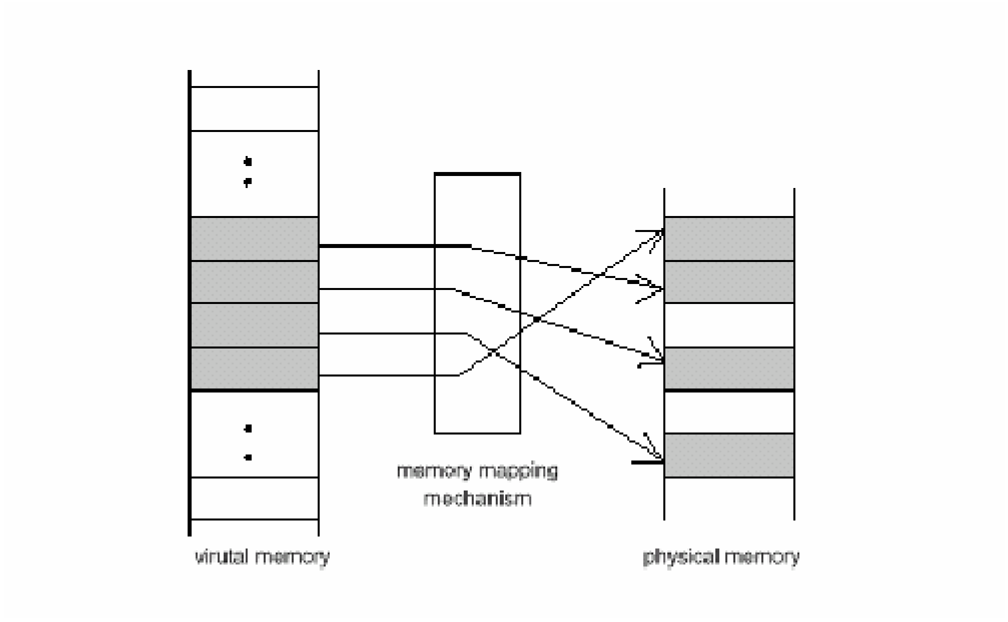

Figure 1: Virtual Memory (Lister)

As shown in the figure above, the memory management systems operation is to translate the virtual address into the actual physical addresses where the data exists.

Implementations of Virtual Memory

Demand Paging

A demand-paging system is a system where the processes reside in secondary memory (usually a disk). When a process is to be executed, it is swapped into the memory. The process is divided into several pages and when needed, that page is restored back into memory instead of the whole process.

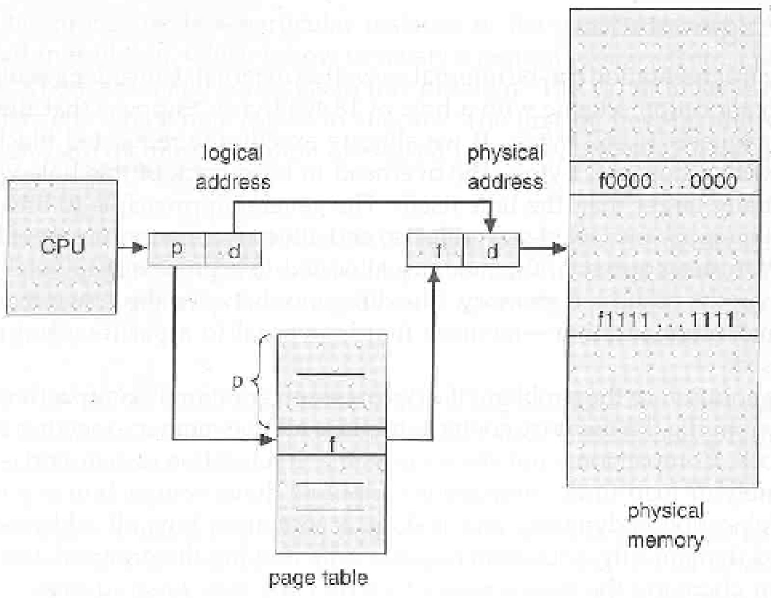

Figure 2: Paging Hardware (Silberschatz & Galvin)

When a process is swapped in, the pager guesses which pages will be used, before the process is swapped out again. Thus it avoids reading into memory, the pages that will not be used anyway hence decreasing the swap time and the amount of physical memory needed. To distinguish between those pages that are in memory and those pages that are on the disk, the valid-invalid bit scheme is used and is clearly illustrated in figure 3.

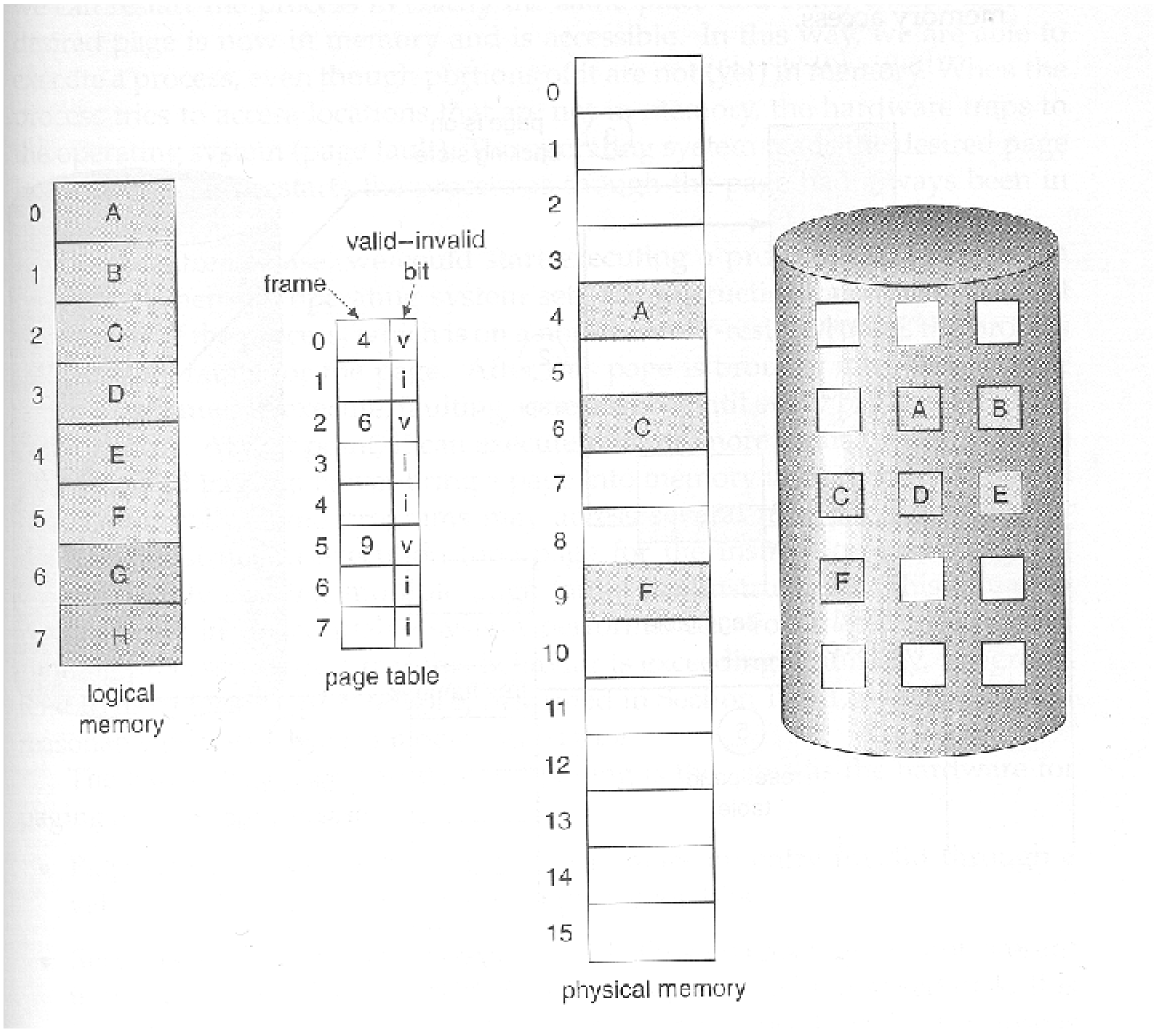

Figure 3: Page table when some pages are not in main memory. (Silberschatz & Galvin)

Viewing the diagram above, it is seen that when this bit is set to "valid," indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page is either not valid or is valid but is currently on the disk. If the process tries to use a page that was not brought into memory causes a *page-fault* trap. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing the trap to be sent to the operating system. It is important to realize that, because we save the state (condition code, registers, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. This makes it possible to execute a process, even though portion of it are not in memory. The hardware to support demand paging is listed as follows:

- Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
- Secondary memory: This memory holds those pages not in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap space or backing store.

Demand Segmentation

When hardware can become an issue, a less efficient way to implement virtual memory is with demand segmentation. The hardware setup used to implement demand segmentation is shown in Figure 4.

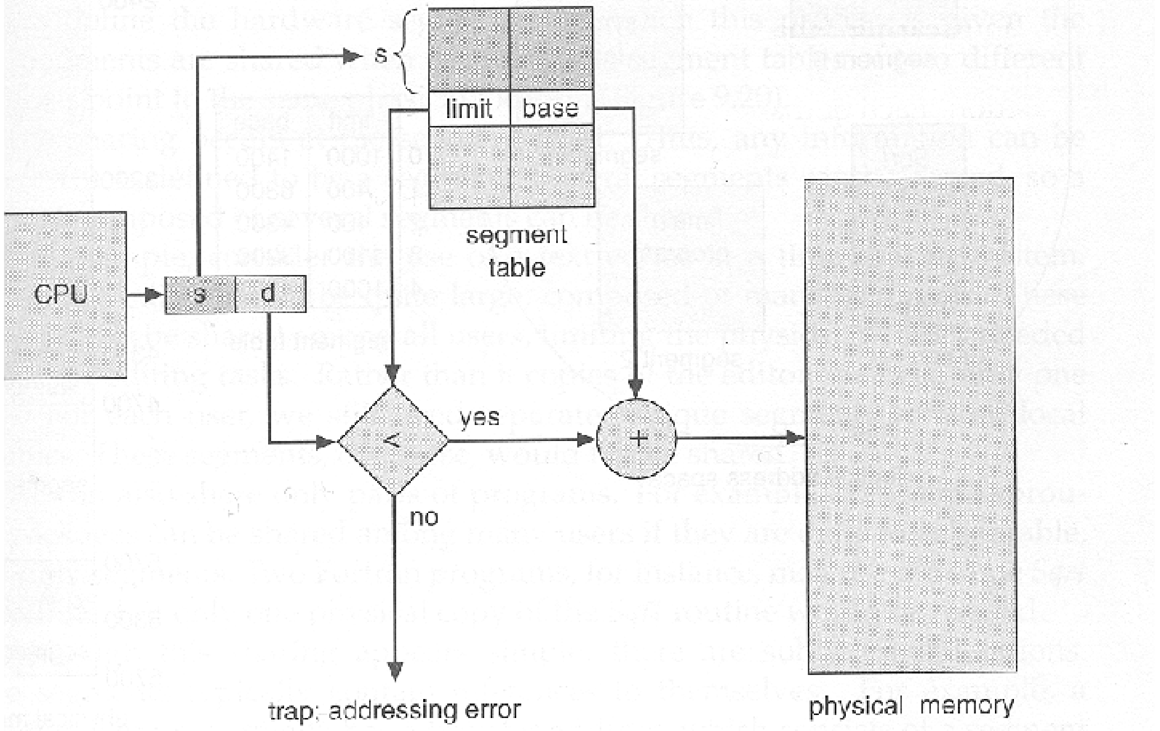

Figure 4: Segmentation Hardware (Silberschatz & Galvin)

The memory is allocated in segments where segment descriptors (includes information about the segment's size, protections, and location) keep track of these segments. Like demand paging, the process does not need to have all of the segments in memory to execute. The segment descriptors contain a valid bit for each segment to indicate whether the segment is in memory. This bit is checked when the process addresses a segment. If the segmentation is not in memory, a segmentation fault occurs and it is brought into memory. The process then continues. The bit in the segment descriptor call accessed bit is used to determine which segment should be replaced in case of a segmentation fault. This bit is set whenever any byte in the segment is either read or written. A queue is kept to contain an entry for each segment in memory with the most recently used segments at the head. When an invalid segment trap occurs, the memory-management routines first determine if there is sufficient free memory space to accommodate the segment. If there is not enough sufficient free memory, the segment at the end of the queue is chosen for replacement, and then the new segment is stored in its place. When the segment descriptor is updated, the new segment is placed at the head of the queue. Demand segmentation may not be an optimal means for making best use of the resources of a computer system because of the required overhead. However, with hardware constraints, demand segmentation is a reasonable compromise when demand paging is impossible.

Page Replacement

Page replacement takes the following approach. If no frame is free, one that is not currently being used is freed. To free a frame, the contents are written to the swap space and the page table is changed to indicate that the page is no longer in memory. The freed frame can now be used to hold the page for which the process is faulted. This can be seen in Figure 5.

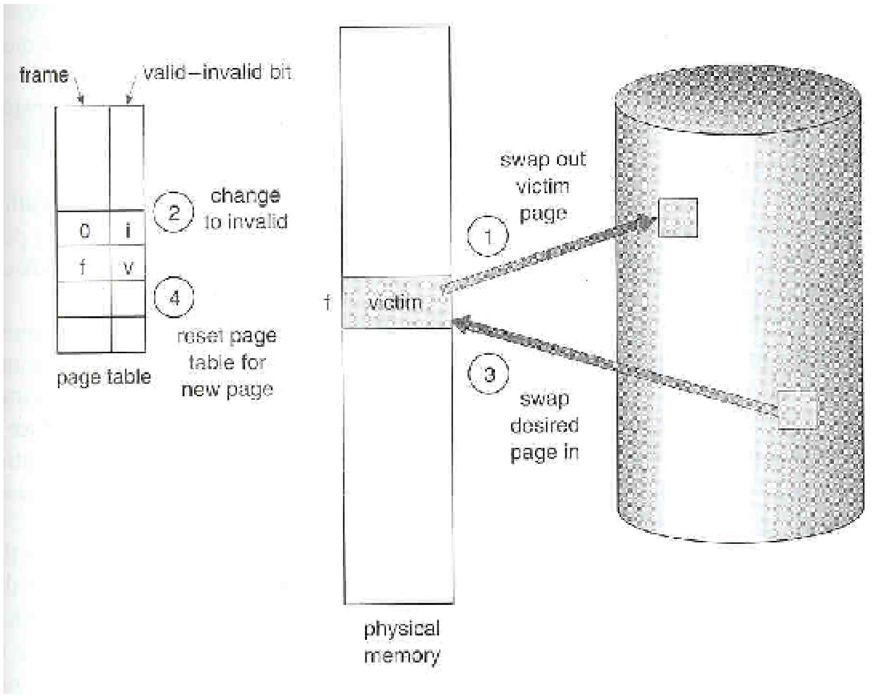

Figure 5: Page Replacement (Silberschatz & Galvin)

ALGORITHMS USED IN VIRTUAL MEMORY

The concept of Page Replacement deals with swapping pages to and from the page table once it becomes full. In the best-case scenario, the page that would be replaced would be the one that is not going to be used for the longest time. Since it's hard to predict the future, the best ways to pick the correct method is by looking at characteristics of the processes and try to predict the future. The algorithms that are used to implement page replacement are explained as follows:

First in, first out (FIFO)

The first-in first-out algorithm is the simplest and oldest algorithm. Each page is time stamped upon entry into the page table, and when the time comes for a page to be replaced, the page with the oldest time gets removed. 'Replace the page which has been resident longest.' (Lister, 42) It ignores the possibility that the oldest page may be the most heavily referenced. Figure 6 illustrates an example of the FIFO algorithm. Note that since the page table in the initially empty; three page faults occur to fill the table. Thereafter, a page fault occurs when the requested page is not in the table.

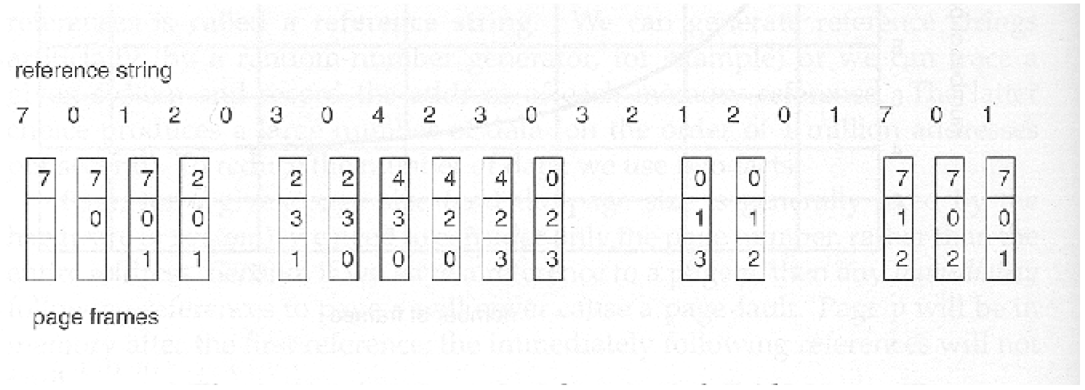

Figure 6: FIFO page-replacement algorithm (Silberschatz & Galvin).

From this figure, initially the first three frames are empty. From the first three references (7, 0, 1) of the reference string, page faults occur, and are brought into these empty frames. Thereafter, a page fault only occurs when the requested page is not in the table. A total of 15 page faults would occur with this reference string.

The FIFO algorithm is fairly simple to understand and implement. On the down side, optimal performance is rarely obtained. Since the page replacement choice is based solely on the age of the pages, the oldest page may be used again fairly soon, thus creating an unnecessary page fault. On the other hand, the oldest page may have been needed only for initialization purpose and will never be used again. The most efficient implementation would be to use a queue. One each page fault, the replaced page would be taken from the exit end (head) of the queue and the new page would add at the entry end (tail) of the queue.

Least-recently-used (LRU)

This algorithm replaces the least-recently-used resident page. In general LRU algorithm performs better than FIFO. The reason is that LRU takes into account the patterns of program behavior by assuming that the page used in the most distant past is least likely to be referenced in the near future. The least-recently-used algorithm belongs to a larger class of the so-called stack replacement algorithm. A stack algorithm is distinguished by the property of performing better, or at least not worse, when more physical memory is made available to the executing program. However, implementation of the LRU algorithm imposes too many overheads to be handled by software alone. One possible implementation is to record the usage of pages by means of a structure similar to the stack. Whenever, a resident page is referenced, it is retrieved from the stack is removed from memory.

VIRTUAL MEMORY IMPLEMENTATION IN WINDOWS NT

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.

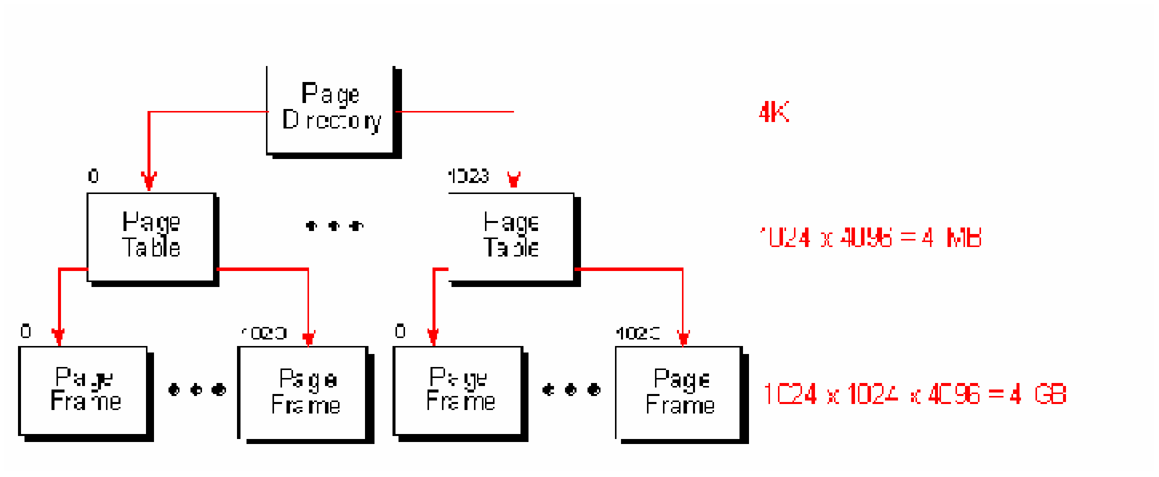

Figure 7 (Kath)

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:

1) the Page Directory Offset
2) the Page Table Offset
3) Physical Page Offset.

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, read-write, or no access to the page. The next 20 bits are the page frame address. The next four bits specify the page file 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)

Windows NT Paging Database

Since not all pages are allocated when a process is created, 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)* - Active pages are pages that are loaded into memory and are part of the working set. These are the pages that are pointed to by all page table entries (PTEs) and each one has a valid PTE pointing to it.
- *Transition* - 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.
- *Standby*
- *Modified* - 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* - Modified no write pages are used by NTFS so that the transaction logs are written before the pages are flushed to disk.
- *Free* - Free pages are pages that have been removed from the active status, but have not been zeroed yet.
- *Zeroed* - 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* - 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.

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. The set manager also schedules the modified page writer whenever there are no free or zeroed pages and there are modified pages. The process/stack swapper performs thread and kernel thread stack in-swapping and out-swapping. The 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 as is shown in the figure below.

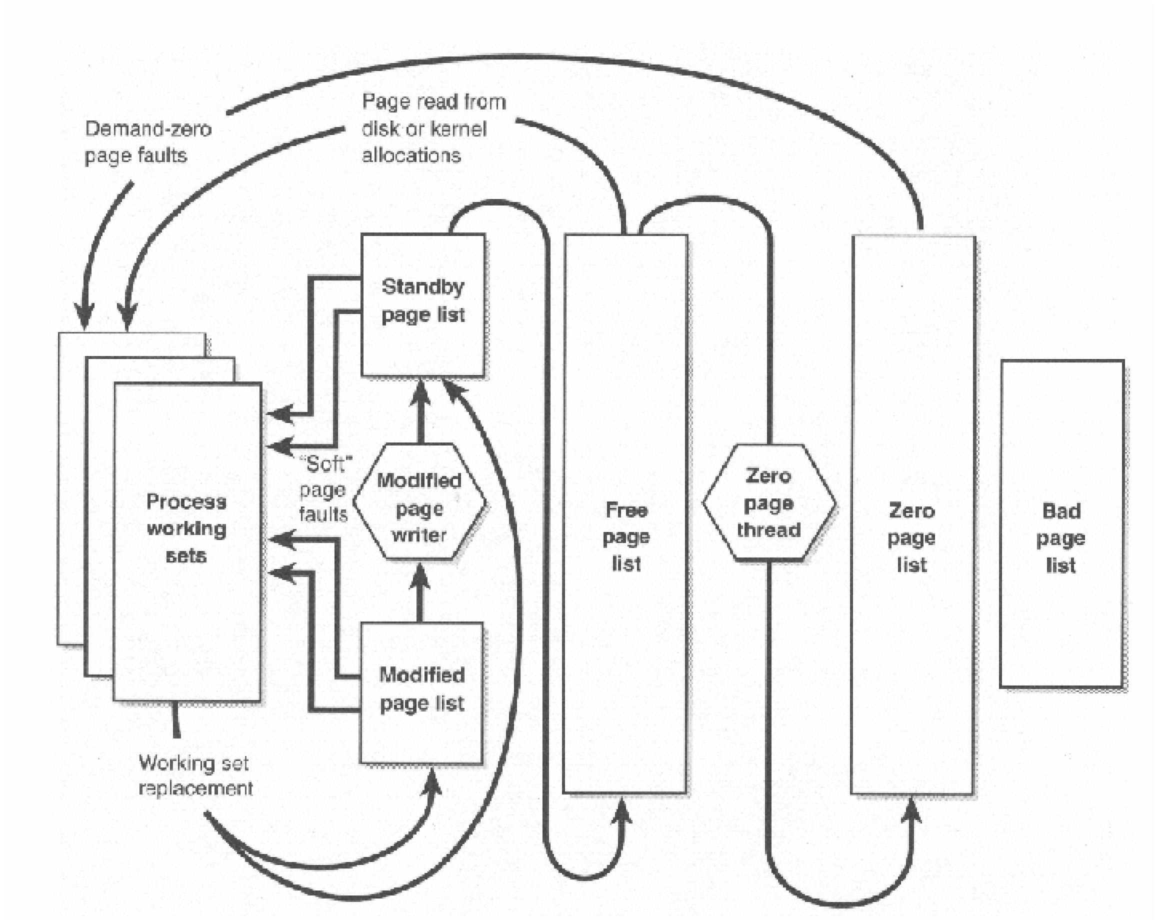

Figure 8 (Solomen)

CONCLUSION

Virtual memory provides the illusion of large address space that almost eliminates considerations imposed by the limited capacity of physical memory. Thus, both system and user programs can provide the desired functionality without concern for the amount of real memory installed in a particular system. However, the main disadvantage of virtual memory is the complex hardware and software needed to support it. Both the space and time complexities of virtual-memory operating systems exceed those of their real-memory counterparts. Large virtual-address space and management of file-map tables contribute to considerably higher table fragmentation.

BIBLIOGRAPHY

1) Silberschatz, Abraham, and Galvin, Peter Baer. Operating Systems Concepts, Sixth Edition. New York: John Wiley and Sons.

2) Virtual Memroy Tutorial.
http://cne.gmu.edu/modules/vm/

3) http://www.cs.duke.edu/~narten/110/nachos/main/node34.html

4) Virtual Memory and Paging
http://www.caa.lcs.mit.edu/~devadas/6.004/Lectures/lect19/

5) Windows NT Virtual Memory
http://www.osr.com/ntinsider/1998/Virtualmem1/virtualmem1.htm

6) Virtual Memory: Issues of Implementation
http://dlib.computer.org/co/books/co1998/pdf/r6033.pdf

7) A Virtual Memory Model for Parallel SupercomputerS", Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)
1063-7133/96 $10.00 © 1996 IEEE
http://dlib.computer.org/conferen/ipps/7255/pdf/72550537.pdf

8) Lister, A.M. *Fundamentals of Operating Systems*. The Macmillan Press LTD, New York, 1980.

9) Milenkovic, Milan. *Operating System Concepts and Design*. McGraw-Hill Book Company, New York, 1987.

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

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