UNIT 2 VIRTUAL MEMORY

Structure **Page Nos.**

2.0 INTRODUCTION

In the earlier unit, we have studied Memory Management covering topics like the overlays, contiguous memory allocation, static and dynamic partitioned memory allocation, paging and segmentation techniques. In this unit, we will study an important aspect of memory management known as Virtual memory.

Storage allocation has always been an important consideration in computer programming due to the high cost of the main memory and the relative abundance and lower cost of secondary storage. Program code and data required for execution of a process must reside in the main memory but the main memory may not be large enough to accommodate the needs of an entire process. Early computer programmers divided programs into the sections that were transferred into the main memory for the period of processing time. As the program proceeded, new sections moved into the main memory and replaced sections that were not needed at that time. In this early era of computing, the programmer was responsible for devising this overlay system.

As higher-level languages became popular for writing more complex programs and the programmer became less familiar with the machine, the efficiency of complex programs suffered from poor overlay systems. The problem of storage allocation became more complex.

Two theories for solving the problem of inefficient memory management emerged -- static and dynamic allocation. ***Static*** allocation assumes that the availability of memory resources and the memory reference string of a program can be predicted. ***Dynamic*** allocation relies on memory usage increasing and decreasing with actual program needs, not on predicting memory needs.

Program objectives and machine advancements in the 1960s made the predictions required for static allocation difficult, if not impossible. Therefore, the dynamic allocation solution was generally accepted, but opinions about implementation were still divided. One group believed the programmer should continue to be responsible

for storage allocation, which would be accomplished by system calls to allocate or deal locate memory. The second group supported **automatic storage allocation** performed by the operating system, because of increasing complexity of storage allocation and emerging importance of multiprogramming. In 1961, two groups proposed a one-level memory store. One proposal called for a very large main memory to alleviate any need for storage allocation. This solution was not possible due to its very high cost. The second proposal is known as virtual memory.

In this unit, we will go through virtual memory and related topics.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- discuss why virtual memory is needed;
- define virtual memory and its underlying concepts;
- describe the page replacement policies like Optimal, FIFO and LRU;
- discuss the concept of thrashing, and
- explain the need of the combined systems like Segmented paging and Paged segmentation.

2.2 VIRTUAL MEMORY

It is common for modern processors to be running multiple processes at one time. Each process has an address space associated with it. To create a whole complete address space for each process would be much too expensive, considering that processes may be created and killed often, and also considering that many processes use only a tiny bit of their possible address space. Last but not the least, even with modern improvements in hardware technology, machine resources are still finite. Thus, it is necessary to share a smaller amount of physical memory among many processes, with each process being given the appearance of having its own exclusive address space.

The most common way of doing this is a technique called virtual memory, which has been known since the 1960s but has become common on computer systems since the late 1980s. The virtual memory scheme divides physical memory into blocks and allocates blocks to different processes. Of course, in order to do this sensibly it is highly desirable to have a protection scheme that restricts a process to be able to access only those blocks that are assigned to it. Such a protection scheme is thus a necessary, and somewhat involved, aspect of any virtual memory implementation. One other advantage of using virtual memory that may not be immediately apparent is that it often reduces the time taken to launch a program, since not all the program code and data need to be in physical memory before the program execution can be started. Although sharing the physical address space is a desirable end, it was not the sole reason that virtual memory became common on contemporary systems. Until the late 1980s, if a program became too large to fit in one piece in physical memory, it was the programmer's job to see that it fit. Programmers typically did this by breaking programs into pieces, each of which was mutually exclusive in its logic. When a program was launched, a main piece that initiated the execution would first be loaded into physical memory, and then the other parts, called overlays, would be loaded as needed.

It was the programmer's task to ensure that the program never tried to access more physical memory than was available on the machine, and also to ensure that the proper overlay was loaded into physical memory whenever required. These responsibilities made for complex challenges for programmers, who had to be able to divide their programs into logically separate fragments, and specify a proper scheme to load the right fragment at the right time. Virtual memory came about as a means to relieve programmers creating large pieces of software of the wearisome burden of designing overlays.

Virtual memory automatically manages two levels of the memory hierarchy, representing the main memory and the secondary storage, in a manner that is invisible to the program that is running. The program itself never has to bother with the physical location of any fragment of the virtual address space. A mechanism called relocation allows for the same program to run in any location in physical memory, as well. Prior to the use of virtual memory, it was common for machines to include a relocation register just for that purpose. An expensive and messy solution to the hardware solution of a virtual memory would be software that changed all addresses in a program each time it was run. Such a solution would increase the running times of programs significantly, among other things.

Virtual memory enables a program to ignore the physical location of any desired block of its address space; a process can simply seek to access any block of its address space without concern for where that block might be located. If the block happens to be located in the main memory, access is carried out smoothly and quickly; else, the virtual memory has to bring the block in from secondary storage and allow it to be accessed by the program.

The technique of virtual memory is similar to a degree with the use of processor caches. However, the differences lie in the block size of virtual memory being typically much larger (64 kilobytes and up) as compared with the typical processor cache (128 bytes and up). The hit time, the miss penalty (the time taken to retrieve an item that is not in the cache or primary storage), and the transfer time are all larger in case of virtual memory. However, the miss rate is typically much smaller. (This is no accident−since a secondary storage device, typically a magnetic storage device with much lower access speeds, has to be read in case of a miss, designers of virtual memory make every effort to reduce the miss rate to a level even much lower than that allowed in processor caches).

Virtual memory systems are of two basic kinds—those using fixed-size blocks called pages, and those that use variable-sized blocks called segments.

Suppose, for example, that a main memory of 64 megabytes is required but only 32 megabytes is actually available. To create the illusion of the larger memory space, the memory manager would divide the required space into units called pages and store the contents of these pages in mass storage. A typical page size is no more than four kilobytes. As different pages are actually required in main memory, the memory manager would exchange them for pages that are no longer required, and thus the other software units could execute as though there were actually 64 megabytes of main memory in the machine.

In brief we can say that virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that the program can be larger than physical memory. Virtual memory can be implemented via demand paging and demand segmentation.

2.2.1 Principles of Operation

The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilise. For example, virtual memory might contain twice as many addresses as main memory. A program using all of virtual memory, therefore, would not be able to fit in main memory all at once. Nevertheless, the computer could execute such a program by copying into the main memory those portions of the program needed at any given point during execution.

To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.

Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory management unit

(MMU) translates virtual addresses to physical addresses at run-time. The process of translating virtual addresses into real addresses is called ***mapping***. The copying of virtual pages from disk to main memory is known as ***paging*** or ***swapping***.

Some physical memory is used to keep a list of references to the most recently accessed information on an I/O (input/output) device, such as the hard disk. The optimisation it provides is that it is faster to read the information from physical memory than use the relevant I/O channel to get that information. This is called ***caching***. It is implemented inside the OS.

2.2.2 Virtual Memory Management

This section provides the description of how the virtual memory manager provides virtual memory. It explains how the logical and physical address spaces are mapped to one another and when it is required to use the services provided by the Virtual Memory Manager.

Before going into the details of the management of the virtual memory, let us see the functions of the virtual memory manager. It is responsible to:

- Make portions of the logical address space resident in physical RAM
- Make portions of the logical address space immovable in physical RAM
- Map logical to physical addresses
- Defer execution of application-defined interrupt code until a safe time.

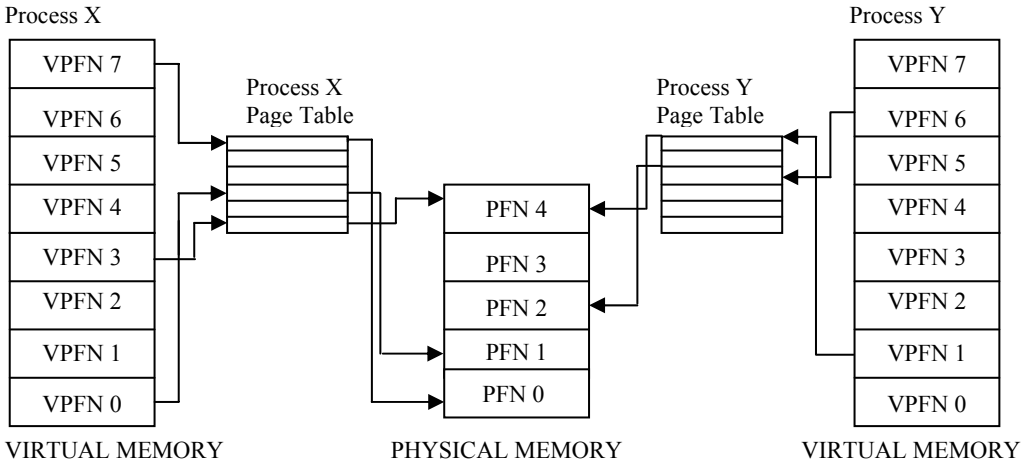

Figure 1: Abstract model of Virtual to Physical address mapping

Before considering the methods that various operating systems use to support virtual memory, it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location of operands in the memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into small blocks called ***pages***. These pages are all of the same size. (It is not necessary that all the pages should be of same size but if they were not, the system would be very hard to administer). Linux on Alpha AXP systems uses 8 Kbytes pages and on Intel x86 systems it uses 4 Kbytes pages. Each of these pages is given a unique number; the page frame number (PFN) as shown in the *Figure 1*.

In this paged model, a virtual address is composed of two parts, an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses ***page tables***.

The *Figure1* shows the virtual address spaces of two processes, process *X* and process *Y*, each with their own page tables. These page tables map each processes virtual pages into physical pages in memory. This shows that process *X's* virtual page frame number 0 is mapped into memory in physical page frame number 1 and that process *Y's* virtual page frame number 1 is mapped into physical page frame number 4. Each entry in the theoretical page table contains the following information:

- *Valid flag* : This indicates if this page table entry is valid.
- *PFN* : The physical page frame number that this entry is describing.
- *Access control information* : This describes how the page may be used. Can it be written to? Does it contain executable code?

The page table is accessed using the virtual page frame number as an offset. Virtual page frame 5 would be the 6th element of the table (0 is the first element).

To translate a virtual address into a physical one, the processor must first work out the virtual addresses page frame number and the offset within that virtual page. By making the page size a power of 2 this can be easily done by masking and shifting. Looking again at the *Figure1* and assuming a page size of *0x2000* bytes (which is decimal 8192) and an address of *0x2194* in process *Y's* virtual address space then the processor would translate that address into offset *0x194* into virtual page frame number 1.

The processor uses the virtual page frame number as an index into the processes page table to retrieve its page table entry. If the page table entry at that offset is valid, the processor takes the physical page frame number from this entry. If the entry is invalid, the process has accessed a non-existent area of its virtual memory. In this case, the processor cannot resolve the address and must pass control to the operating system so that it can fix things up.

Just how the processor notifies the operating system that the correct process has attempted to access a virtual address for which there is no valid translation is specific to the processor. However, the processor delivers it, this is known as a ***page fault*** and the operating system is notified of the faulting virtual address and the reason for the page fault. A page fault is serviced in a number of steps:

i) Trap to the OS.

ii) Save registers and process state for the current process.

iii) Check if the trap was caused because of a page fault and whether the page reference is legal.

iv) If yes, determine the location of the required page on the backing store.

v) Find a free frame.

vi) Read the required page from the backing store into the free frame. (During this I/O, the processor may be scheduled to some other process).

vii) When I/O is completed, restore registers and process state for the process which caused the page fault and save state of the currently executing process.

viii) Modify the corresponding PT entry to show that the recently copied page is now in memory.

ix) Resume execution with the instruction that caused the page fault.

Assuming that this is a valid page table entry, the processor takes that physical page frame number and multiplies it by the page size to get the address of the base of the page in physical memory. Finally, the processor adds in the offset to the instruction or data that it needs.

Using the above example again, process *Y's* virtual page frame number 1 is mapped to physical page frame number 4 which starts at *0x8000* (4 x *0x2000*). Adding in the *0x194* byte offset gives us a final physical address of *0x8194*. By mapping virtual to physical addresses this way, the virtual memory can be mapped into the system's physical pages in any order.

2.2.3 Protection and Sharing

Memory protection in a paged environment is realised by protection bits associated with each page, which are normally stored in the page table. Each bit determines a page to be read-only or read/write. At the same time the physical address is calculated with the help of the page table, the protection bits are checked to verify whether the memory access type is correct or not. For example, an attempt to write to a read-only memory page generates a trap to the operating system indicating a memory protection violation.

We can define one more protection bit added to each entry in the page table to determine an invalid program-generated address. This bit is called valid/invalid bit, and is used to generate a trap to the operating system. Valid/invalid bits are also used to allow and disallow access to the corresponding page. This is shown in *Figure 2*.

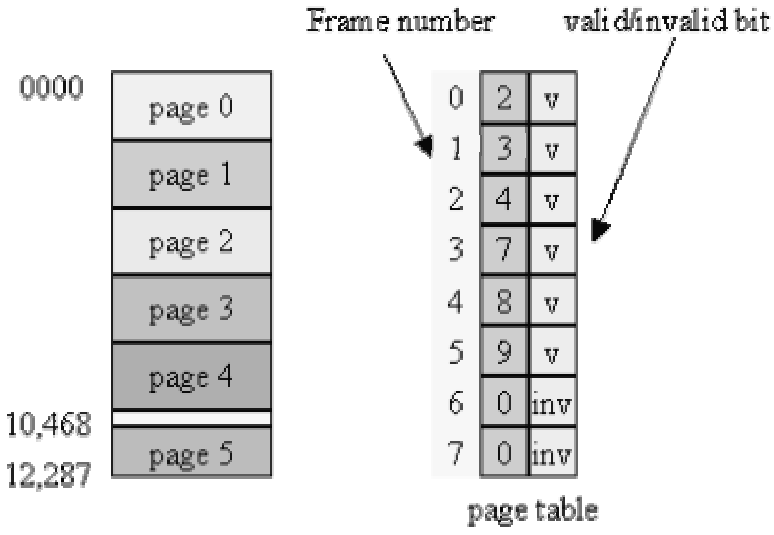

Figure 2: Protection bit in the page table Shared pages

The paging scheme supports the possibility of sharing common program code. For example, a system that supports 40 users, each of them executes a text editor. If the text editor consists of 30 KB of code and 5 KB of data, we need 1400 KB. If the code is reentrant, i.e., it never changes by any write operation during execution (non-self-modifying code) it could be shared as presented in *Figure 3*.

Only one copy of the editor needs to be stored in the physical memory. In each page table, the included editor page is mapped onto the same physical copy of the editor, but the data pages are mapped onto different frames. So, to support 40 users, we only need one copy of the editor, i.e., 30 KB, plus 40 copies of the 5 KB of data pages per user; the total required space is now 230 KB instead of 1400 KB.

Other heavily used programs such as assembler, compiler, database systems etc. can also be shared among different users. The only condition for it is that the code must be reentrant. It is crucial to correct the functionality of shared paging scheme so that the pages are unchanged. If one user wants to change a location, it would be changed for all other users.

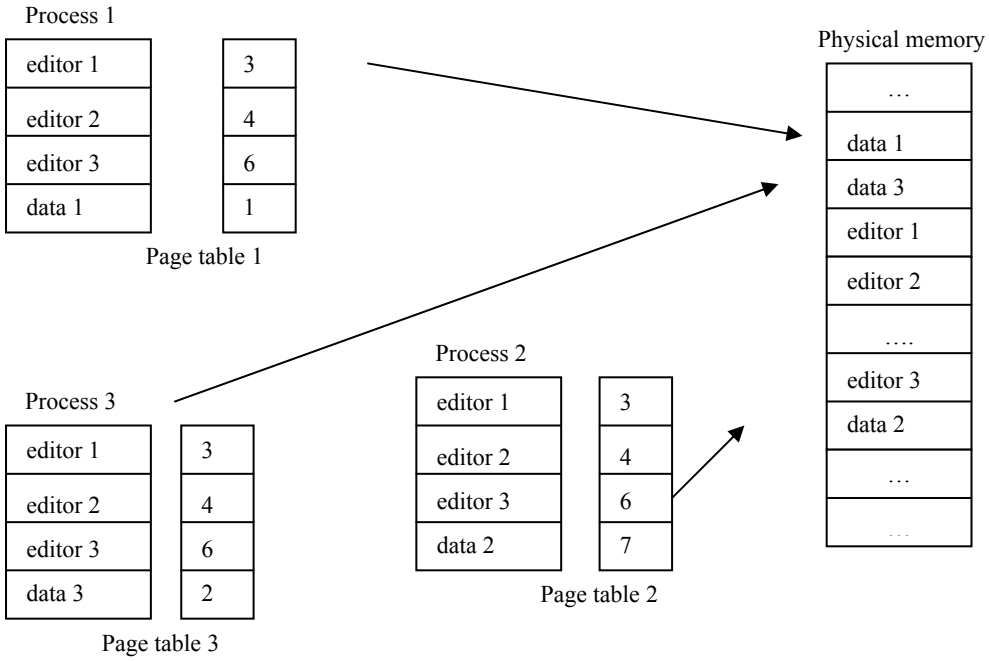

Figure 3: Paging scheme supporting the sharing of program code

2.3 DEMAND PAGING

In a multiprogramming system memory is divided into a number of fixed-size or variable-sized partitions or regions that are allocated to running processes. For example: a process needs *m* words of memory may run in a partition of *n* words where $n \geq m$. The variable size partition scheme may result in a situation where available memory is not contiguous, but fragmented into many scattered blocks. We distinguish between *internal fragmentation* and *external fragmentation*. The difference *(n – m)* is called internal fragmentation, memory that is internal to a partition but is not being used. If a partition is unused and available, but too small to be used by any waiting process, then it is accounted for external fragmentation. These memory fragments cannot be used.

In order to solve this problem, we can either ***compact*** the memory making large free memory blocks, or implement ***paging*** scheme which allows a program's memory to be non-contiguous, thus permitting a program to be allocated to physical memory.

Physical memory is divided into fixed size blocks called ***frames***. Logical memory is also divided into blocks of the same, fixed size called ***pages***. When a program is to be executed, its pages are loaded into any available memory frames from the disk. The disk is also divided into fixed sized blocks that are the same size as the memory frames.

A very important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. Normally, a user believes that memory is one contiguous space containing only his/her program. In fact, the logical memory is scattered through the physical memory that also contains other programs. Thus, the user can work correctly with his/her own view of memory because of the address translation or address mapping. The address mapping, which is controlled by the operating system and transparent to users, translates logical memory addresses into physical addresses.

Because the operating system is managing the memory, it must be sure about the nature of physical memory, for example: which frames are available, which are allocated; how many total frames there are, and so on. All these parameters are kept in a data structure called ***frame table*** that has one entry for each physical frame of

memory indicating whether it is free or allocated, and if allocated, to which page of which process.

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to load only virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. Also, if the database query is a search query then it is not necessary to load the code from the database that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as ***demand paging***.

When a process attempts to access a virtual address that is not currently in memory the CPU cannot find a page table entry for the virtual page referenced. For example, in *Figure 1*, there is no entry in *Process X's* page table for virtual PFN 2 and so if *Process X* attempts to read from an address within virtual PFN 2 the CPU cannot translate the address into a physical one. At this point the CPU cannot cope and needs the operating system to fix things up. It notifies the operating system that a ***page fault*** has occurred and the operating system makes the process wait whilst it fixes things up. The CPU must bring the appropriate page into memory from the image on disk. Disk access takes a long time, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual PFN is added to the processes page table. The process is then restarted at the point where the memory fault occurred. This time the virtual memory access is made, the CPU can make the address translation and so the process continues to run. This is known as demand paging and occurs when the system is busy but also when an image is first loaded into memory. This mechanism means that a process can execute an image that only partially resides in physical memory at any one time.

The valid/invalid bit of the page table entry for a page, which is swapped in, is set as valid. Otherwise it is set as invalid, which will have no effect as long as the program never attempts to access this page. If all and only those pages actually needed are swapped in, the process will execute exactly as if all pages were brought in.

If the process tries to access a page, which was not swapped in, i.e., the valid/invalid bit of this page table, entry is set to invalid, then a page fault trap will occur. Instead of showing the "invalid address error" as usually, it indicates the operating system's failure to bring a valid part of the program into memory at the right time in order to minimize swapping overhead.

In order to continue the execution of process, the operating system schedules a disk read operation to bring the desired page into a newly allocated frame. After that, the corresponding page table entry will be modified to indicate that the page is now in memory. Because the state (program counter, registers etc.) of the interrupted process was saved when the page fault trap occurred, the interrupted process can be restarted at the same place and state. As shown, it is possible to execute programs even though parts of it are not (yet) in memory.

In the extreme case, a process without pages in memory could be executed. Page fault trap would occur with the first instruction. After this page was brought into memory, the process would continue to execute. In this way, page fault trap would occur further until every page that is needed was in memory. This kind of paging is called ***pure demand paging***. Pure demand paging says that "never bring a page into memory until it is required".

2.4 PAGE REPLACEMENT POLICIES

Basic to the implementation of virtual memory is the concept of ***demand paging***. This means that the operating system, and not the programmer, controls the swapping of pages in and out of main memory, as the active processes require them. When a process needs a non-resident page, the operating system must decide which resident page is to be replaced by the requested page. The part of the virtual memory which makes this decision is called the ***replacement policy***.

There are many approaches to the problem of deciding which page is to replace but the object is the same for all-the policy that selects the page that will not be referenced again for the longest time. A few page replacement policies are described below.

2.4.1 First In First Out (FIFO)

The First In First Out (FIFO) replacement policy chooses the page that has been in the memory the longest to be the one replaced.

Belady's Anomaly

Normally, as the number of page frames increases, the number of page faults should decrease. However, for FIFO there are cases where this generalisation will fail! This is called Belady's Anomaly. Notice that OPT's never suffers from Belady's anomaly.

2.4.2 Second Chance (SC)

The Second Chance (SC) policy is a slight modification of FIFO in order to avoid the problem of replacing a heavily used page. In this policy, a reference bit *R* is used to keep track of pages that have been recently referenced. This bit is set to 1 each time the page is referenced. Periodically, all the reference bits are set to 0 by the operating system to distinguish pages that have not been referenced recently from those that have been. Using this bit, the operating system can determine whether old pages are still being used (i.e., $R = 1$). If so, the page is moved to the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues. Thus heavily accessed pages are given a "second chance."

2.4.3 Least Recently Used (LRU)

The Least Recently Used (LRU) replacement policy chooses to replace the page which has not been referenced for the longest time. This policy assumes the recent past will approximate the immediate future. The operating system keeps track of when each page was referenced by recording the time of reference or by maintaining a stack of references.

2.4.4 Optimal Algorithm (OPT)

Optimal algorithm is defined as replace the page that will not be used for the longest period of time. It is optimal in the performance but not feasible to implement because we cannot predict future time.

Example:

Let us consider a 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	1	4
	2	2	2	2	2
		3	3	3	3
			4	5	5

In the above *Figure*, we have 6 page faults.

2.4.5 Least Frequently Used (LFU)

The Least Frequently Used (LFU) replacement policy selects a page for replacement if the page had not been used often in the past. This policy keeps count of the number of

times that a page is accessed. Pages with the lowest counts are replaced while pages with higher counts remain in primary memory.

2.5 THRASHING

Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults it is necessary to be in order to appreciate the benefits of virtual memory, thrashing has a negative effect on the system.

As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault. While the transactions in the system are waiting for the paging device, CPU utilisation, system throughput and system response time decrease, resulting in below optimal performance of a system.

Thrashing becomes a greater threat as the degree of multiprogramming of the system increases.

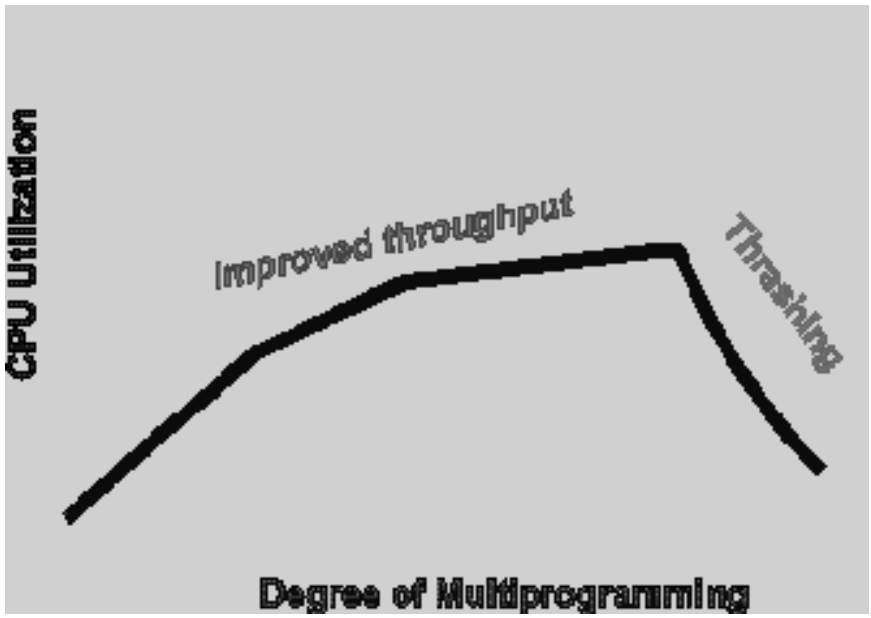

Figure 4: Degree of Multiprogramming

The graph in *Figure 4* shows that there is a degree of multiprogramming that is optimal for system performance. CPU utilisation reaches a maximum before a swift decline as the degree of multiprogramming increases and thrashing occurs in the over-extended system. This indicates that controlling the load on the system is important to avoid thrashing. In the system represented by the graph, it is important to maintain the multiprogramming degree that corresponds to the peak of the graph.

The selection of a replacement policy to implement virtual memory plays an important part in the elimination of the potential for thrashing. A policy based on the local mode will tend to limit the effect of thrashing. In local mode, a transaction will replace pages from its assigned partition. Its need to access memory will not affect transactions using other partitions. If other transactions have enough page frames in the partitions they occupy, they will continue to be processed efficiently.

A replacement policy based on the global mode is more likely to cause thrashing. Since all pages of memory are available to all transactions, a memory-intensive transaction may occupy a large portion of memory, making other transactions susceptible to page faults and resulting in a system that thrashes. To prevent thrashing,

we must provide a processes as many frames as it needs. There are two techniques for this– *Working-Set Model and Page-Fault Rate.*

2.5.1 Working-Set Model

Principle of Locality

Pages are not accessed randomly. At each instant of execution a program tends to use only a small set of pages. As the pages in the set change, the program is said to move from one phase to another. The principle of locality states that most references will be to the current small set of pages in use. The examples are shown below:

Examples:

1) Instructions are fetched sequentially (except for branches) from the same page.
2) Array processing usually proceeds sequentially through the array functions repeatedly, access variables in the top stack frame.

Ramification

If we have locality, we are unlikely to continually suffer page-faults. If a page consists of 1000 instructions in self-contained loop, we will only fault once (at most) to fetch all 1000 instructions.

Working Set Definition

The working set model is based on the assumption of locality. The idea is to examine the most recent page references in the working set. If a page is in active use, it will be in the Working-set. If it is no longer being used, it will drop from the working set.

The set of pages currently needed by a process is its working set.

WS(k) for a process P is the number of pages needed to satisfy the last k page references for process P.

WS(t) is the number of pages needed to satisfy a process's page references for the last *t* units of time.

Either can be used to capture the notion of locality.

Working Set Policy

Restrict the number of processes on the ready queue so that physical memory can accommodate the working sets of all ready processes. Monitor the working sets of ready processes and, when necessary, reduce multiprogramming (i.e. swap) to avoid thrashing.

Note: Exact computation of the working set of each process is difficult, but it can be estimated, by using the reference bits maintained by the hardware to implement an aging algorithm for pages.

When loading a process for execution, pre-load certain pages. This prevents a process from having to "fault into" its working set. May be only a rough guess at start-up, but can be quite accurate on swap-in.

2.5.2 Page-Fault Rate

The ***working-set model*** is successful, and knowledge of the working set can be useful for pre-paging, but it is a scattered way to control thrashing. A page-fault frequency (page-fault rate) takes a more direct approach. In this we establish upper and lower bound on the desired page- fault rate. If the actual page fault rate exceeds the upper limit, we allocate the process another frame. If the page fault rate falls below the lower limit, we remove a Frame from the process. Thus, we can directly measure and control the page fault rate to prevent thrashing.

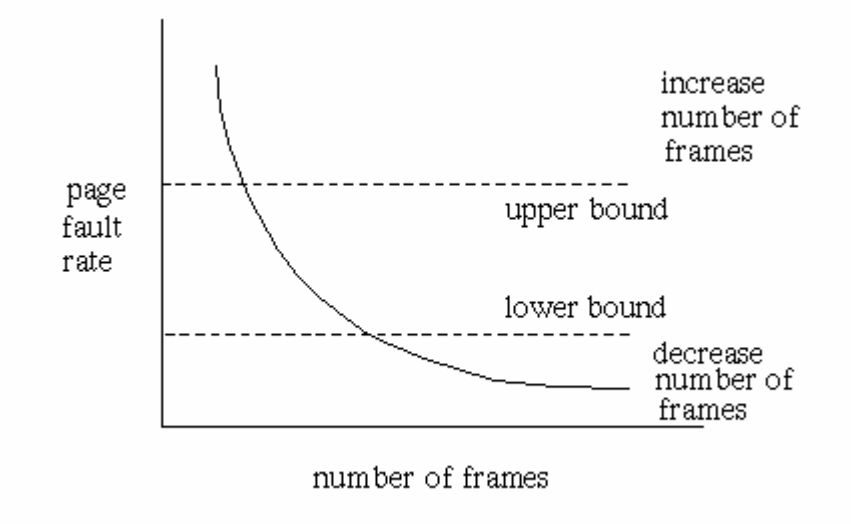

Figure 5: Page-fault frequency

Establish "acceptable" page-fault rate.

- If actual rate too low, process loses frame.
- If actual rate too high, process gains frame.

2.6 DEMAND SEGMENTATION

Segmentation

Programs generally divide up their memory usage by function. Some memory holds instructions, some static data, some dynamically allocated data, some execution frames. All of these memory types have different protection, growth, and sharing requirements. In the monolithic memory allocation of classic Virtual Memory systems, this model isn't well supported.

Segmentation addresses this by providing multiple sharable, protectable, growable address spaces that processes can access.

In pure segmentation architecture, segments are allocated like variable partitions, although the memory management hardware is involved in decoding addresses. Pure segmentation addresses replace the page identifier in the virtual address with a segment identifier, and find the proper segment (not page) to which to apply the offset.

The segment table is managed like the page table, except that segments explicitly allow sharing. Protections reside in the segment descriptors, and can use keying or explicit access control lists to apply them.

Of course, the segment name space must be carefully managed, and thus OS must provide a method of doing this. The file system can come to the rescue here–a process can ask for a file to be mapped into a segment and have the OS return the segment register to use. This is known as memory mapping files. It is slightly different from memory mapping devices, because one file system abstraction (a segment) is providing an interface to another (a file). Memory mapped files may be reflected into the file system or not and may be shared or not at the process's discretion.

The biggest problem with segmentation is the same as with variable sized real memory allocation: managing variable sized partitions can be very inefficient, especially when the segments are large compared to physical memory. External fragmentation can easily result in expensive compaction when a large segment is loaded, and swapping large segments (even when compaction is not required) can be costly.

Demand Segmentation

Same idea as demand paging applied to segments.

If a segment is loaded, base and limit are stored in the Segment Table Entry (STE) and the valid bit is set in the Page Table Entry (PTE). The PTE is accessed for each memory reference. If the segment is not loaded, the valid bit is unset. The base and limit as well as the disk address of the segment is stored in the OS table. A reference to a non-loaded segment generates a segment fault (analogous to page fault). To load a segment, we must solve both the placement question and the replacement question (for demand paging, there is no placement question).

2.7 COMBINED SYSTEMS

The combined systems are of two types. They are:

- Segmented Paging
- Paged Segmentation

2.7.1 Segmented Paging

In a pure paging scheme, the user thinks in terms of a contiguous linear address space, and internal fragmentation is a problem when portions of that address space include conservatively large estimates of the size of dynamic structures. Segmented paging is a scheme in which logically distinct (e.g., dynamically-sized) portions of the address space are deliberately given virtual addresses a LONG way apart, so we never have to worry about things bumping into each other, and the page table is implemented in such a way that the big unused sections don't cost us much. Basically the only page table organisation that doesn't work well with segmented paging is a single linear array. Trees, inverted (hash) tables, and linked lists work fine. If we always start segments at multiples of some large power of two, we can think of the high-order bits of the virtual address as specifying a segment, and the low-order bits as specifying an offset within the segment. If we have tree-structured page tables in which we use k bits to select a child of the root, and we always start segments at some multiple of 2^(word size-k), then the top level of the tree looks very much like a segment table. The only difference is that its entries are selected by the high-order address bits, rather than by some explicit architecturally visible mechanism like segment registers. Basically all modern operating systems on page-based machines use segmented paging.

2.7.2 Paged Segmentation

In a pure segmentation scheme, we still have to do dynamic space management to allocate physical memory to segments. This leads to external fragmentation and forces us to think about things like compaction. It also doesn't lend itself to virtual memory. To address these problems, we can page the segments of a segmented machine. This is paged segmentation. Instead of containing base/bound pairs, the segment table entries of a machine with paged segmentation indicate how to find the page table for the segment. In MULTICS, there was a separate page table for each segment. The segment offset was interpreted as consisting of a page number and a page offset. The base address in the segment table entry is added to the segment offset to produce a "linear address" that is then partitioned into a page number and page offset, and looked up in the page table in the normal way. Note that in a machine with pure segmentation, given a fast way to find base/bound pairs (e.g., segment registers), there is no need for a TLB. Once we go to paged segmentation, we need a TLB. The difference between segmented paging and paged segmentation lies in the user's programming model, and in the addressing modes of the CPU. On a segmented architecture, the user generally specifies addresses using an effective address that includes a segment register specification. On a paged architecture, there are no segment registers. In practical terms, managing segment registers (loading them with appropriate values at appropriate times) is a bit of a nuisance to the assembly language

programmer or compiler writer. On the other hand, since it only takes a few bits to indicate a segment register, while the base address in the segment table entry can have many bits, segments provide a means of expanding the virtual address space beyond 2^(word size). We can't do segmented paging on a machine with 16-bit addresses. It's beginning to get problematical on machines with 32-bit addresses. We certainly can't build a MULTICS-style single level store, in which every file is a segment that can be accessed with ordinary loads and stores, on a 32-bit machine. Segmented architectures provide a way to get the effect we want (lots of logically separate segments that can grow without practical bound) without requiring that we buy into very large addresses. As 64-bit architectures become more common, it is possible that segmentation will become less popular. One might think that paged segmentation has an additional advantage over segmented paging: protection information is logically associated with a segment, and could perhaps be specified in the segment table and then left out of the page table. Unfortunately, protection bits are used for lots of purposes other than simply making sure we cannot write your code or execute your data.

☞ Check Your Progress 1

1) What are the steps that are followed by the Operating system in order to handle the page fault?

……

……

2) What is demand paging?

……

……

3) How can you implement the virtual memory?

……

……

4) When do the following occurs?

   i) A Page Fault
   ii) Thrashing

……

……

5) What should be the features of the page swap algorithm?

……

……

6) What is a working set and what happens when the working set is very different from the set of pages that are physically resident in memory?

……

……

2.8 SUMMARY

With previous schemes, all the code and data of a program have to be in main memory when the program is running. With virtual memory, only some of the code and data have to be in main memory: the parts needed by the program now. The other parts are loaded into memory when the program needs them without the program having to be aware of this. The size of a program (including its data) can thus exceed the amount of available main memory.

There are two main approaches to virtual memory: paging and segmentation. Both approaches rely on the separation of the concepts virtual address and physical address. Addresses generated by programs are virtual addresses. The actual memory cells have physical addresses. A piece of hardware called a memory management unit (MMU) translates virtual addresses to physical addresses at run-time.

In this unit we have discussed the concept of Virtual memory, its advantages, demand paging, demand segmentation, Page replacement algorithms and combined systems.

2.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) The list of steps that are followed by the operating system in handling a page fault:

   a) If a process refers to a page which is not in the physical memory then an internal table kept with a process control block is checked to verify whether a memory reference to a page was valid or invalid.

   b) If the memory reference to a page was valid, but the page is missing, the process of bringing a page into the physical memory starts.

   c) Free memory location is identified to bring a missing page.

   d) By reading a disk, the desired page is brought back into the free memory location.

   e) Once the page is in the physical memory, the internal table kept with the process and page map table is updated to indicate that the page is now in memory.

   f) Restart the instruction that was interrupted due to the missing page.

2) In demand paging pages are loaded only on demand, not in advance. It is similar to paging system with swapping feature. Rather than swapping the entire program in memory, only those pages are swapped which are required currently by the system.

3) Virtual memory can be implemented as an extension of paged or segmented memory management scheme, also called **demand** paging or **demand segmentation** respectively.

4) A Page Fault occurs when a process accesses an address in a page which is not currently resident in memory. Thrashing occurs when the incidence of page faults becomes so high that the system spends all its time swapping pages rather than doing useful work. This happens if the number of page frames allocated is insufficient to contain the working set for the process either because there is inadequate physical memory or because the program is very badly organised.

5) A page swap algorithm should

   a) Impose the minimum overhead in operation

   b) Not assume any particular hardware assistance

   c) Try to prevent swapping out pages which are currently referenced (since they may still be in the Working Set)

   d) Try to avoid swapping modified pages (to avoid the overhead of rewriting pages from memory to swap area).

6) The working Set is the set of pages that a process accesses over any given (short) space of time. If the Working Set is very different from the set of pages that are physically resident in memory, there will be an excessive number of page faults and thrashing will occur.

2.10 FURTHER READINGS

1) Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Wiley and Sons (Asia) Publications, New Delhi.

2) H.M.Deitel, *Operating Systems*, Pearson Education Asia, New Delhi.

3) Andrew S.Tanenbaum, *Operating Systems- Design and Implementation*, Prentice Hall of India Pvt. Ltd., New Delhi.

4) Achyut S. Godbole, *Operating Systems*, Second Edition, TMGH, 2005, New Delhi.

5) Milan Milenkovic, *Operating Systems*, TMGH, 2000, New Delhi.

6) D. M. Dhamdhere, *Operating Systems – A Concept Based Approach*, TMGH, 2002, New Delhi.

7) William Stallings, *Operating Systems*, Pearson Education, 2003, New Delhi.