MCA Part-II

Paper-XIII: Operating System

Topic: VIRTUAL MEMORY

PREPARED BY: DR. KIRAN PANDEY

(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

INTRODUCTION

The need of virtual memory arises from the fact that **physical memory is "limited".** A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory.**

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.

- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below –

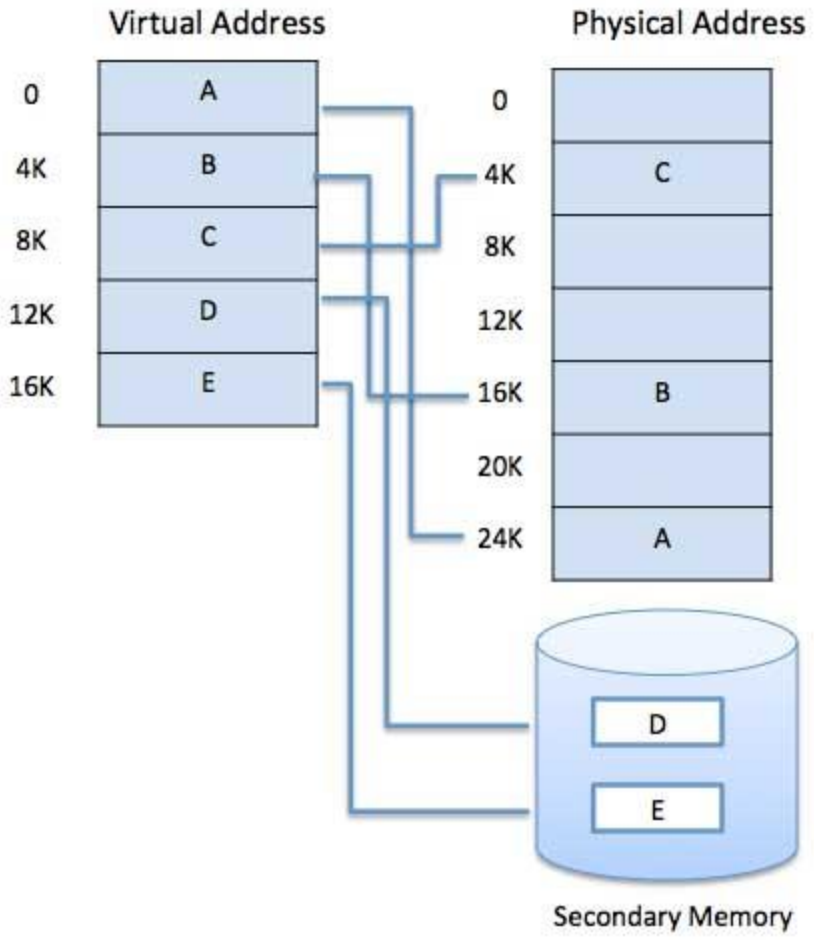


Figure 1: Virtual Memory Concept

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 we are running a program with memory requirements of 6GB on a 32 bit system. 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 6GB 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.

Virtual Memory Management

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.

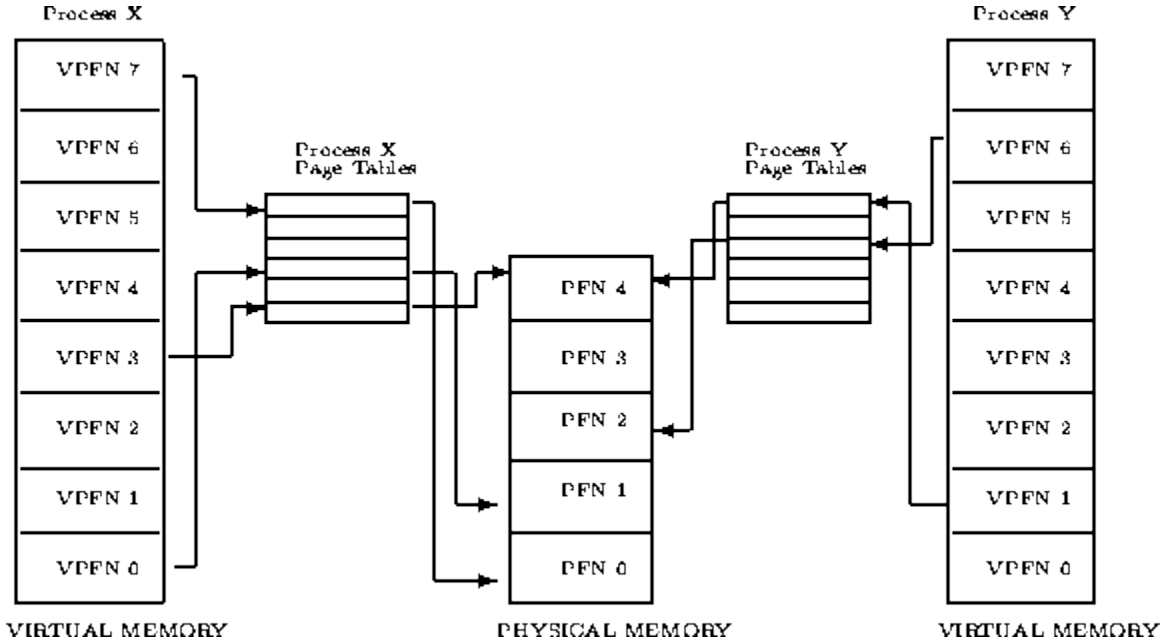


Figure 2: Abstract model of Virtual to Physical address mapping

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 2*.

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.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

DEMAND PAGING

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

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***. Demand paging is shown below in the figure:

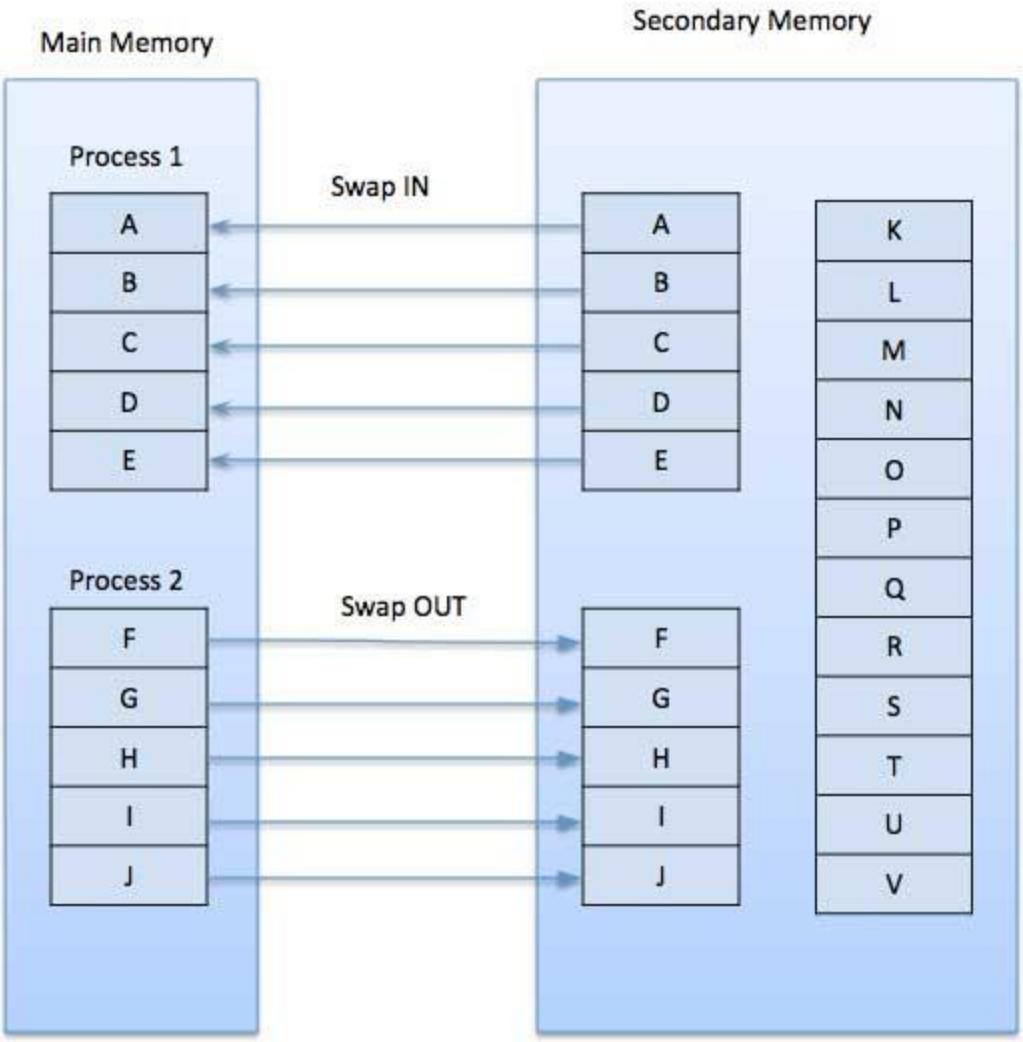


Figure 3 : Demand Paging

While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

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".

Advantages

Following are the advantages of Demand Paging –

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement Algorithms

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

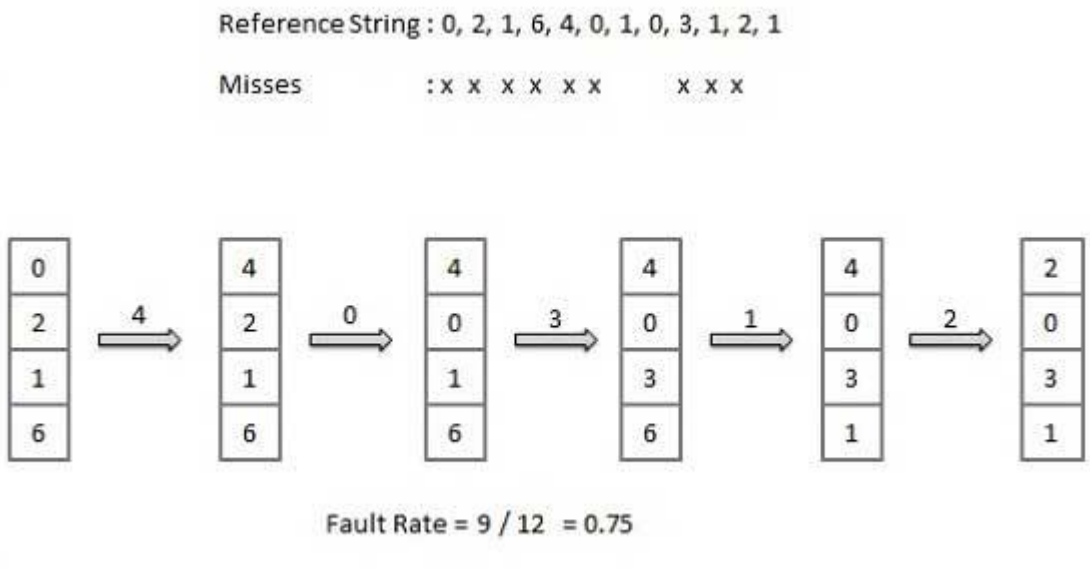
- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses – 123, 215, 600, 1234, 76, 96
- If page size is 100, then the reference string is 1,2,6,12,0,0

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.

a. First In First Out (FIFO) algorithm

- Oldest page in main memory is the one which will be selected for replacement.

- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.



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.

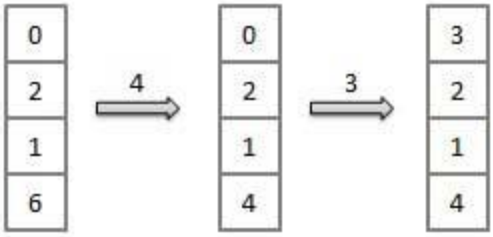
b. Optimal Page algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.

- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



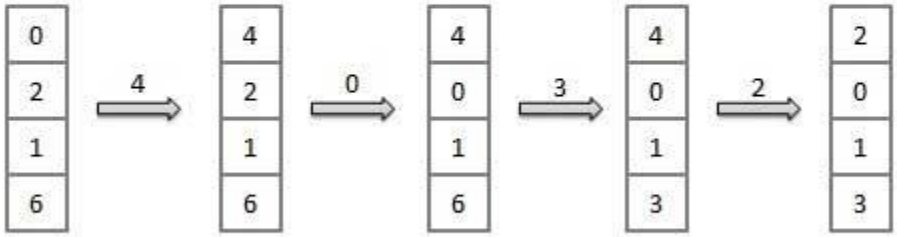
Fault Rate = 6 / 12 = 0.50

c. Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

d. Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.

- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

e. Least frequently Used(LFU) algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

f. Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Thrashing

A process that is spending more time paging than executing is said to be **thrashing**. In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the processes are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.

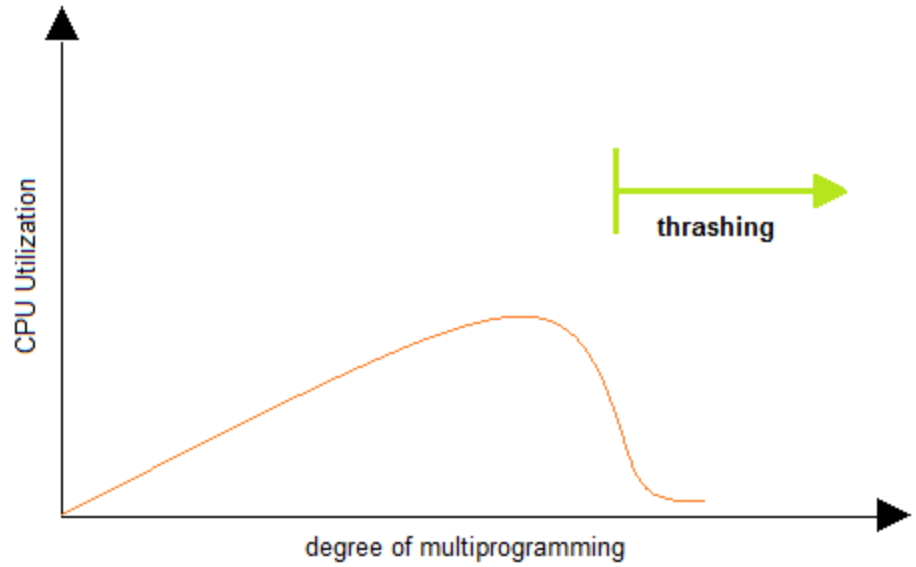


Figure 4: Thrashing

While the transactions in the system are waiting for the paging device, CPU utilization, 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.

The graph in *Figure 4* shows that there is a degree of multiprogramming that is optimal for system performance. CPU utilization 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

processes with as many frames as they really need "right now". There are two techniques for this-*Working-Set Model and Page-Fault Rate.*

a. Working-Set Model

The ***working set model*** is based on the concept of locality, and defines a ***working set window,*** of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set,

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 100 instructions in self-contained loop, we will only fault once (at most) to fetch all 100 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.

Page-Fault Rate

The working-set model is successful, and knowledge of the working set can be useful for pre-paging, but it seems a clumsy way to control thrashing. A strategy that uses the Page-Fault Frequency (PFF) takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds 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.

As with the working-set strategy, we may have to suspend a process. If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.

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

Operating system also uses demand segmentation, which is similar to demand paging. Operating system to uses demand segmentation where there is insufficient hardware available to implement **'Demand Paging'**.

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).

Demand segmentation allows for pages that are often referenced with each other to be brought into memory together, this decreases the number of page faults.