Lecture 06: Memory Management: Virtual Memory

Virtual memory is the memory management strategy in modern operating systems. This lecture discusses the fundamentals of virtual memory under the following captions:

- Early Development of VM Technology
- Questions to Tackle in VM
- Demand Paging
- Page Replacement Algorithms
- Number of Frames to Allocate
- Thrashing
- Other Considerations
- Memory Hierarchy
- Summary and Concluding Remarks

6.1 Early Development of VM Technology

Virtual memory (VM) is the technique used whereby only the required portions of a program are kept in memory. The unused portions are kept in auxiliary storage, and loaded as required. The most significant advantage of this strategy is that a user program is not restricted to physical space of the hardware.

Overlaying

Early VM systems employed the strategy of *overlaying*. In this strategy, only portions of the program that are needed are kept in memory.

Example 1:

Consider a program with three procedures (subprograms): **ProcA**, **ProcB**, and **ProcC**. When **ProcA** is executing it alone occupies memory; when **ProcB** is required, it is overlaid in the space where **ProcA** was… and so on…

Overlaying was controlled by the programmer and therefore became complicated with increased size and complexity of programs.

Dynamic Loading

In *dynamic loading*, only the main portion (segment or page) of the program is loaded at the start. As subprograms are called, they are loaded. Actually, the full page (or segment) table is set up but entries are flagged invalid until needed.

The drawback again was that the technique made too much demand on the programmer. What was needed was to make the OS manage the loading without program intervention. Further refinement led to the following:

- The OS takes care of loading required pages or segments of program code (only currently required parts are loaded). The effect of this is that the user gets the illusion that more space is available to a program than physically exists.
- Whenever the main program refers to a section of code that is not in memory, a *memory fault* occurs. The section must be swapped in. A memory fault may be a page fault or a segment fault, depending on the architecture of the operating system.

6.2 Question to Tackle in VM

In order to be effective, the VM strategy must tackle the following questions:

a. How is a program portion recognized as missing? What is a page (or segment) fault?
b. What happens while the missing portion is brought into memory?
c. Where is the missing portion loaded?
d. How is continuation of program execution effected?
e. How much of a program is held in memory?
f. What happens if there is no room for a missing but needed portion?
g. Is the scheme going to be too complicated or too slow?

Demand paging, the most common implementation of VM will, be examined. The principles established will apply to VM via segmentation also.

6.3 Demand Paging

In *demand paging*, the program resides in auxiliary storage storage. A page is swapped in by the OS only when it is needed. This is done in a manner that is transparent to end user.

6.3.1 Page Fault

A *page fault* occurs when the main program refers to a page that is not in memory. The page must therefore be swapped in. The management of page faults provides the answer to questions a, b, c, d above.

The OS sets up space for the complete page table and flags pages not loaded as invalid. Access to a page not loaded causes a page fault to occur.

When a page fault occurs, the operating system takes over, fetches the missing page, updates the page table and then returns control to the program. Of course, while a page fault is addressed, control is passed to some other process.

When the paging device is through with loading the required page, the operating system is interrupted and the process continues. Figure 6.1 illustrates.

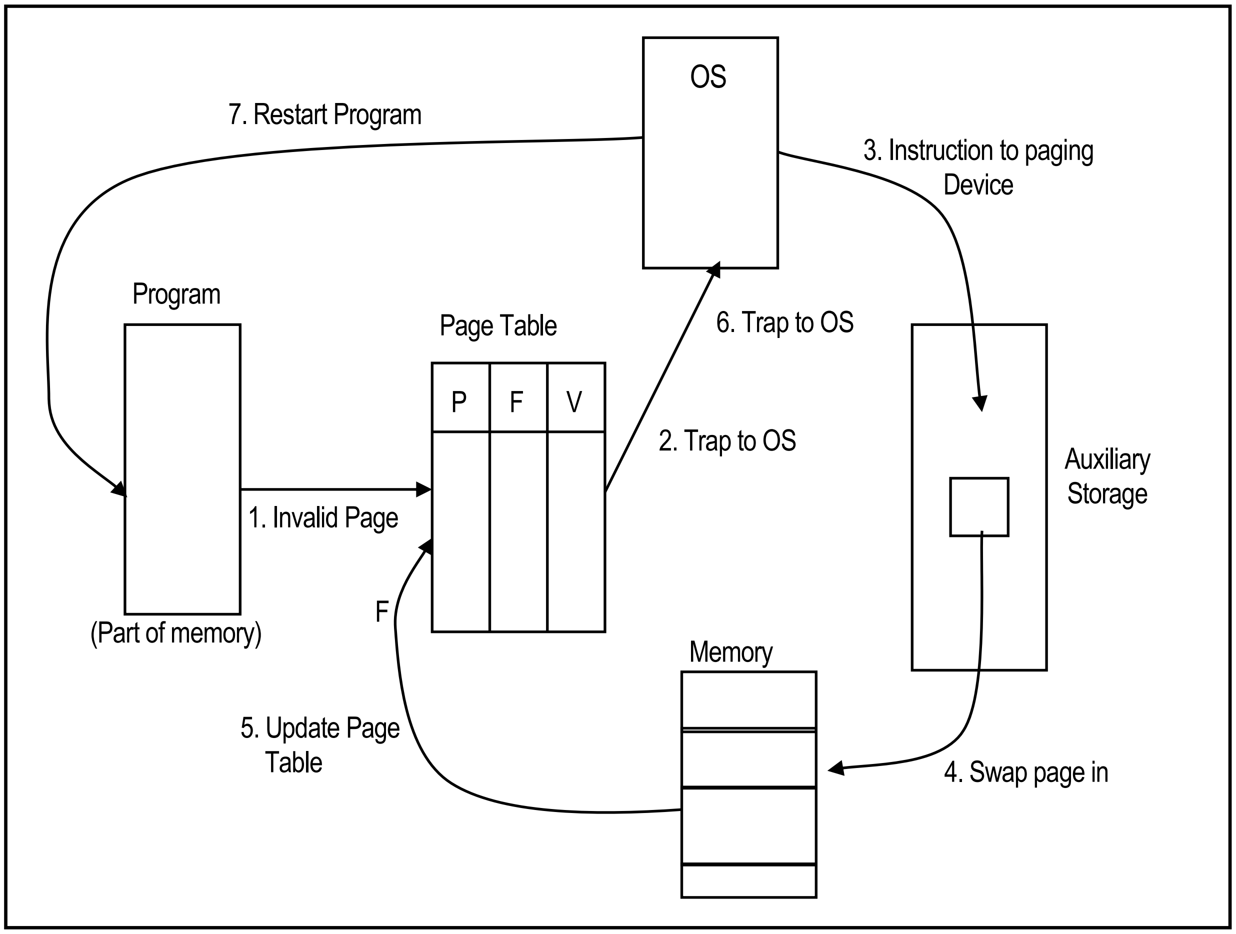
6.3.1 Page Fault (continued)

Page fault service is affected as follows:

1. Find location of the desired page on auxiliary storage.
2. Interrupt the operating system about the missing frame.
3. Find a free frame from auxiliary storage.
4. If no free frame exists, select a "victim frame" for replacement; swap out the "victim frame"; change the page table accordingly; then swap in desired page from auxiliary storage.
5. Update page table.
6. Interrupt the operating system about the update.
7. Restart the process.

Note: If no free frame exists, two page transfers are required to service a page fault (one out and another in). To minimize this overhead, a `dirty bit' may be assigned to each page in the page table. If an address on that page has changed since the last swap-in, its dirty bit is set to `1'; otherwise it's off. When the page is to be replaced, it is swapped out only if its dirty bit is on.

Figure 6.1: Illustrating Implementation of a Page Fault



6.3.2 Resuming the Process

Resuming the process is not difficult. The worst case is to re-fetch the instruction on which the page fault occurred, and execute it.

Since a page fault is obviously an interrupt, which evokes privileged instructions, the house keeping procedures related to interrupts apply here (review section 2.3).

6.3.3 Maintaining Free Space

The maintenance of free frames provides answers to questions c and g above (section 6.2) in the following way:
- A free frame list is maintained.
- Pages are loaded into one of the free frames and the list automatically updated.
- The free frame list may be implemented as a bit map or a free space table (as discussed in lecture 3).
- The list must be constantly updated.

6.3.4 Page Replacement

Page replacement provides the answer to question f above (section 6.2). If the free space list is empty, this there is no free space available. If a page fault occurs at this point, there is no space to hold the missing page. Two alternatives are possible:
- Suspend the process and wait for free frames.
- Override (i.e. replace) a page not currently in use.

Suspending the process is not advisable since it may result in the occurrence of deadlock — several jobs waiting indefinitely on each other (see next lecture). The second alternative is a much wiser, though more complicated alternative. In fact, two options are available:
- Local replacement — replacement of pages confined to the local job.
- Global Replacement — replacement of pages confined to the entire system and not necessary the local job. This would require the maintenance of a separate page table for all jobs.

6.4 Page Replacement Algorithms

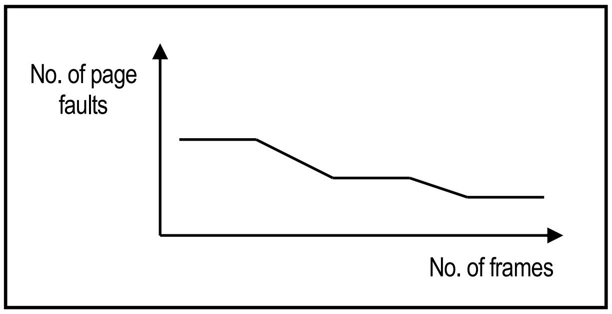
Several page replacement algorithms have been proposed, including FIFO, LRU (least recently used), LFU (least frequently used), SCR (second chance replacement). We will briefly examine each algorithm.
Generally an algorithm with a relatively low page fault rate is preferred to one with a relatively high page fault rate. The algorithms are
- First-in-first-out (FIFO)
- Least frequently used (LFU)
- Least recently used (LRU)
- Second chance replacement (SCR)

6.4.1 FIFO Replacement

The FIFO strategy replaces pages on a first-in-first-out basis. It obviously does not yield the best result, due to a number of reasons:

- The main-program is the most likely segment of code to be swapped out first since it is the most likely segment to be swapped in first. Swapping out a program's main-program would be unfortunate, since it would most likely have to be immediately swapped back in. In a multiprogramming, multitasking system, this would be catastrophic for the operating system.
- No account is taken of the potential use or past use of the page. Therefore page faults could occur frequently.
- The approach suffers from Belady's anomaly: increasing the number of frames for a job does not necessarily result in reduced page faults.

Figure 6.2: Page Fault Curve for FIFO



6.4.2 LFU Replacement

The LFU strategy replaces the page least frequently used. Account is kept of the number of references to pages (frames). The frame with the lowest count is the candidate for replacement. Replacement may be local (i.e., confined the running process), or global (i.e., related to all executing processes as a whole). The page table keeps track of the frequency of usage of each page, as illustrated in figure 6.3.

Figure 6.3: Representation of the Page Table for LFU Strategy

Page	Frame	Frequency	Validity Bit	R/W Bit

LFU suffers from two significant drawbacks:

- A page could develop a high frequency and remain in memory even though it is no longer required.
- A recently brought in page may become the candidate for replacement.

6.4.3 LRU Replacement

The LRU strategy replaces the page least recently used i.e. the page not in use for the longest time. Page table entries could have the time of last usage, as illustrated in figure 6.4.

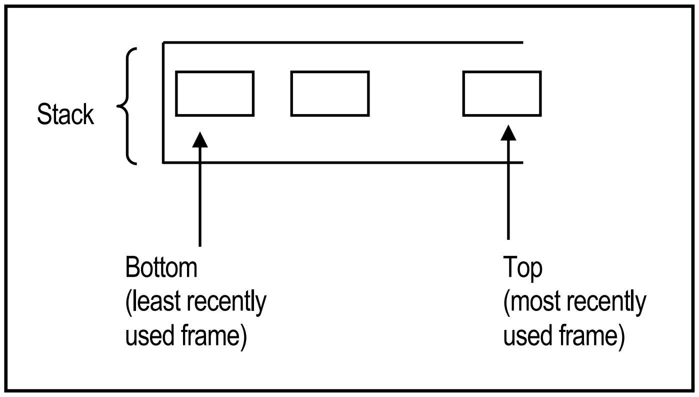
Figure 6.4: Representation of the Page Table for LRU Strategy

Page	Frame	Last Usage Time	Validity Bit	R/W Bit

Time can be tracked from the start of the process, by a counter. Every time a page is referenced, the counter value is copied to the time holder in the page table. When page replacement is required, the page with the lowest time count is replaced.

Alternately, a stack could be used to determine which page should be replaced: As pages are used, the frame number is pushed onto the stack. The frame for replacement will always be at the bottom of the stack (see figure 6.5).

Figure 6.5: Illustrating use of a Stack in LRU



6.4.4 Second Chance Replacement

The SCR algorithm is essentially FIFO with some modification, as explained below:

- In the page table, each page is assigned a reference bit, (that we will denote R).
- When a page is used its R-value is set to 1, otherwise R is 0. When a page P_i is loaded, its load time is noted and R_i is set to 1.
- Each reference to the page table updates R_i values.
- If a page is old and R_i is 1, then R_i is reset to 0 and T_i to the current time. This in effect, gives this page a second chance. It will not be replaced until other pages are either replaced or are given second chances.
- Old pages with $R_i = 0$ are candidates for replacement.

6.4.4 Second Chance Replacement Continued)

In order to facilitate SCR, the page table is adjusted to store the load time and reference bit (R-bit) for each page. This is illustrated in figure 6.6.

Figure 6.6: Representation of the Page Table for SCR Strategy

Page	Frame	Validity Bit	R/W Bit	Load Time	R-Bit

The SCR algorithm can be further enhanced by adding an additional modification bit (M) for each page entry in the page table. If the page has been updated in the last access of the page table, its M-bit is set to 1, otherwise it is set to 0. The page table would then have entries as illustrated in figure 6.7.

Figure 6.7: Representation of the Page Table for Enhanced SCR Strategy

Page	Frame	Validity Bit	R/W Bit	Load Time	R-Bit	M-Bit

By considering the R-bit and the M-bit as ordered pairs, we have the following possibilities:

[0,0]:	Neither currently used nor modified. This is the best candidate for replacement.
[0,1]:	Not recently used but modified. This page will have to be written out before it can be replaced; it is not as good a candidate for replacement as the [0,0] case.
[1,0]:	Recently used but not modified. The page may soon be used again.
[1,1]:	Recently used and modified. This is the worst candidate for replacement.

The SCR algorithm reduces to FIFO if all the page table entries have a [1,1] setting for the R-bit and M-bit. However, this situation, though possible, is highly unlikely to happen.

6.5 Number of Frames to Allocate

We cannot allocate more than the total number of frames available to a process. Thus if **f** is the number of frames allocated to a process and **F** is the total number of frames available, then **f** <= **F**.

Practically, it is not necessary to define an upper limit on frames allocation in a multiprogramming environment.

The minimum number of frames to be allocated to a process depends on the instruction set of the architecture: It is constrained by the maximum number of memory accesses that an instruction will need to execute (assume one page per memory access). To illustrate, for indirect addressing, the minimum allocation is 3 pages (frames).

To allocate frames among contending processes, three alternatives have been used:

- Equal allocation
- Proportional allocation
- Contingency allocation based on the need for, and availability frames

Figure 6.8 provides the formulas used for equal allocation, and proportional allocation respectively. You have no doubt observed that the equal allocation approach is naïve, could be wasteful for small jobs, and exacting for large jobs. Except for a student project, you will not readily find examples of this approach. The proportional allocation is much more pragmatic; the number of frames allocated is proportional to the size of the job.

Figure 6.8: Formulas for Equal Allocation and Proportional Allocation

Equal Allocation

$A_I = M/N$

Where M = Number of frames available
N = Number of processes
A_i = Number of frames allocated to process P_I

Proportional Allocation

$A_I = (S_i/S) * M$

Where A_i = Number of frames allocated to process P_I
S_i = Memory space for P_I
S = Total available memory space
M = Total Number of frames available

Note: In each of these two cases A_i must be adjusted upwards to an integer value.

.

6.5 Number of Frames to Allocate (continued)

Let us now focus for a while on contingency allocation, based on each job's need for frames, and the availability of free frames. This strategy takes into consideration the reality that all jobs will not start together, and their needs will vary. The strategy may be summarized as follows:

- Allocate a basic size to jobs.
- Allocate additional frames to each job based on the behavior of the job.
- Allow high priority jobs to replace frames of lower priority jobs if necessary.
- Replacement may therefore be local or global in the case of lower priority jobs.

The *working set model* (to be discussed in the next section) may be used to dynamically monitor fault rate of jobs in the system.

6.6 Thrashing

Thrashing is the unacceptably high occurrence of page faults. The paging device is therefore kept as busy as it could possibly be (close to 100%).

Thrashing may occur for a given process, a group of processes or the entire system as a whole:

- Thrashing for one process is normally due to inadequate frame allocation for that process.
- Thrashing among several processes is normally due to the paging device trying to satisfy process(es) at the expense of other(s). This could mean that the attempted degree of multiprogramming among these processes is too high. It would also mean that more resources need to be allocated for these processes.
- System-wide thrashing is due to too many jobs being in memory at the same time. The level of multiprogramming needs to be reduced, or more resources need to be provided for the operating system.

Three critical CPU performance factors to watch are the level of *CPU utilization*, the level of *database faults*, and the level of *program faults*. The CPU utilization is a measure of how hard the processor is working over time. The database fault rate is an indication of how much the operating system is working to provide required data to executing processes. The program fault rate is an indication of how hard the operating system is working to provide required program components to executing processes. Each measure is expressed as a percentage. The result of thrashings is degradation in the operating system performance. Thrashing must therefore be avoided, or promptly corrected whenever it occurs.

Example 2:

On earlier versions of the IBM i, the command WRKSYSSTS allows the user to view among other things, the level of page faults occurring. If the page fault rate (database or program) is above a given value (*say 60*), it is not good; it indicates thrashing.

Thrashing may be corrected by one or a combination of the following steps:

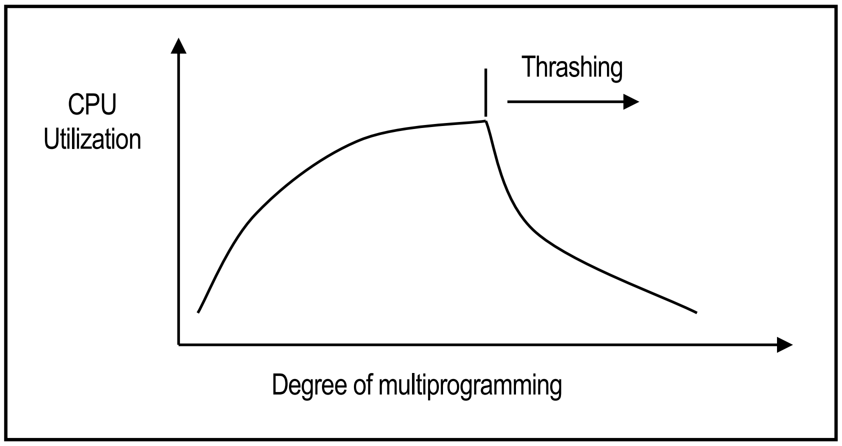
a. Adjust the ratio of memory allocation among *MACHINE, *BASE, *INTER and other pools which might have been created (according to an IBM supplied formula).
b. Reduce and/or increase level of multiprogramming in a pool or some pools by changing the activity level.
c. Reorganize jobs by introduction or withdrawal of subsystems, job queues and pools.

The technique is called performance turning. Modern versions of the OS include expert programs which perform most of the adjustment automatically, but the user is still given the option of adjusting the system environment.

6.6 Thrashing (continued)

Figure 6.9 provides a graphical representation of the utilization of the operating system when thrashing occurs. As the degree of multiprogramming increases, the CPU utilization gradually increases to a dangerously high level (typically above 94%), stays there for a considerable period of time, and then suddenly falls off, not because of a reduction in processing demands, but due to a stalling of the system.

Figure 6.9: Illustrating Thrashing



At its root, thrashing is a memory resource problem, so to reduce it, that root problem must be addressed. Following are strategies that may be employed:

- Reduce the level of multiprogramming. In systems such as IBM i, this can easily be done by simply reducing the activity level in the affected subsystem. Unfortunately, not all systems provide expert users with this flexibility.
- Provide more memory resources for the system. On systems such as the IBM i, this can be done by allocating more memory to the subsystem(s) in question. On many other systems, expert users can install more memory (RAM) chips, but are not allowed to assign memory to internal subsystems.
- Provide each process with the adequate number of frames needed. However, blind pursuit of this strategy runs the risk of reducing the system to paging.
- The working set model is a good compromise to the previous strategy; the approach is based on the locality principle (mentioned in lecture 4). The idea is to have the current page being used, and the most recently used pages of a program, as the working set. These pages must be kept in the page table as they reflect the locality of the program during execution. Cache is one case in point of this concept. Windows is an excellent example of an operating system that has successfully implemented the working set model.

6.7 Other Considerations

The introduction of VM has been huge for memory management. The main challenge is managing the page fault rate. Two additional strategies that have been employed in VM are *pre-paging* and *I/O interlocking*.

6.7.1 Pre-paging

To minimize the high occurrence of page faults, when a new page (locality) is loaded, the operating system could be made to store the actual localities (i.e. page table instance reflecting different localities). When a process is suspended, its entire locality is brought back in memory before resumption. The strategy is referred to as pre-paging.

The cost of pre-paging is that page faults must be carefully evaluated before taking a decision. This is of concern to operating system designers.

6.7.2 I/O Interlock

When demand paging is used, it is sometimes necessary to prevent a page being swapped out. If the page is the object of a pending I/O request it is not desirable to swap out this page. This may be resolved in one of two ways.

- The operating system does not execute I/O to user memory. Instead, it executes I/O to system memory and system memory to user memory. This could be quite expensive.
- I/O lock bits are introduced (one per page) in the page table. A page that is I/O locked cannot be replaced.

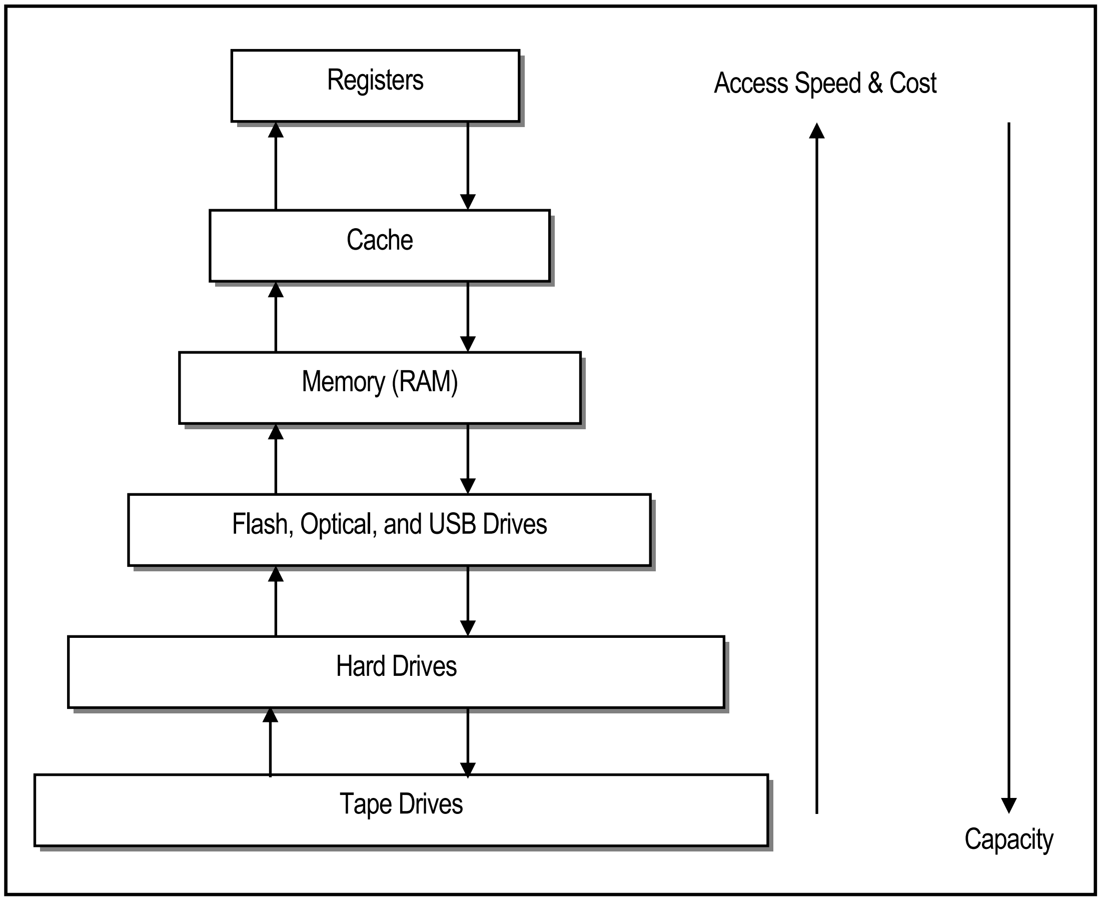
6.8 Memory Hierarchy

By way of review, it is important to remind you of the memory hierarchy in terms of speed of access and capacity of storage. This you would have encountered in a course on *computer organization and/or architecture*. A simplified version is presented in figure 6.10. As you examine the figure, please observe the following:

- As you go down the hierarchy, the average storage capacity increases on each memory unit.
- As you advance up the hierarchy, the average cost of each memory unit increases.
- Magnetic tapes are not a prevalent as they used to be. However, in mainframe environments, they are still used.

6.8 Memory Hierarchy (continued)

Figure 6.10: Memory Hierarchy



6.9 Summary and Concluding Remarks

It is now time to summarize what has been covered in this lecture:

- Virtual memory (VM) is the technique used whereby only the required portions of a program are kept in memory. The unused portions are kept in auxiliary storage, and loaded as required.
- VM is often implemented through demand paging. This technique allows the operating system to load only the required pages of an executing program, and to swap in additional pages as they are required.
- Whenever the processor encounters requires information that is on a page that is not in memory, a page fault occurs. This page fault causes an interrupt of the operating system. The operating system saves the CPU state of the executing process, loads in the missing page, updates the page table, reinstates the CPU state, and then sends control back to the executing process on the process's next instance of CPU attention.
- As the level of multiprogramming increases, the operating may be forced to replace pages that were previously loaded. Four page replacement algorithms are FIFO, LFU, LRU, and SCR. FIFO and LFU are highly inefficient, while LRU and SCR are reasonably efficient, and therefore more widely used.
- The operating system also determines the number of frames to allocate to each executing process. This is typically done via proportional allocation, or the working set model.

6.9 Summary and Concluding Remarks (continued)

- Thrashing is the unacceptably high occurrence of page faults. This is caused by a high degree of multiprogramming. If left unattended, thrashing could bring the system down.
- Trashing can be treated by an appropriate combination of the following strategies: reducing the level of multiprogramming, increasing the system's memory resources, increasing the frame allocation for the process(es) in question, or implementing the working set model.
- Two additional strategies for controlling the page fault rate that have been employed in VM are pre-paging and I/O interlocking.

Each of the last four lectures has made mention of the importance of the operating system allowing processes to peacefully compete for system resources without disrupting the ability of other processes to do the same. In the next lecture, more emphasis is placed on the very important requirement.

6.10 References and/or Recommended Readings

[Bacon & Harris 2003] Bacon, Jean & Tim Harris. 2003. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley. See Chapter 5.

[Davis & Rajkumar 2005] Davis, William S. & T. M. Rajkumar. 2005. *Operating Systems: A Systematic View*, 6th ed. Boston: Addison-Wesley. See Chapter 6.

[Nutt 2004] Nutt, Gary. 2004. *Operating Systems: A Modern Perspective* 3rd ed. Boston: Addison-Wesley. See Chapter 12.

[Silberschatz 2012] Silberschatz, Abraham, Peter B. Galvin, & Greg Gagne. 2012. *Operating Systems Concepts*, 9th Ed. Update. New York: John Wiley & Sons. See Chapter 9.

[Stallings 2005] Stallings, William. 2005. *Operating Systems* 5th ed. Upper Saddle River, New Jersey: Prentice Hall. See Chapter 8.