**5132 – OPERATING SYSTEM**

**MODULE 3**

**CO3 Illustrate memory management schemes.**

Memory management - Different address bindings – compile, link and run time bindings. - Difference between logical address and physical address - Contiguous memory allocation – fixed partition and variable partition – Allocation Strategies - first fit, best fit and worst fit - Define fragmentation – internal and external, and solutions - Paging and paging hardware - Segmentation, advantages of segmentation over paging- Concept of virtual memory - Demand paging - Page-faults and how to handle page faults. - Page replacement algorithms: FIFO, optimal, LRU -Thrashing.

**MEMORY MANAGEMENT**

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

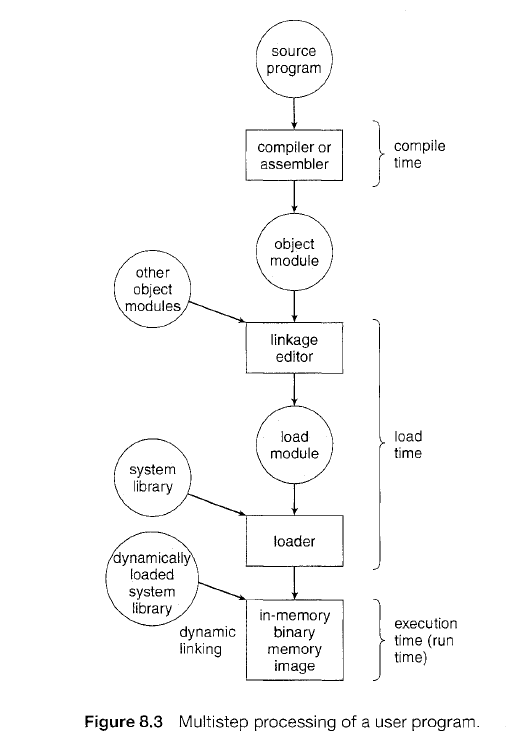
**Address Binding**

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. The processes on the disk that are waiting to be brought into memory for execution form the **input queue.**

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Address Binding refers to the mapping of computer instructions and data to physical memory locations.

In most cases, a user program will go through several steps-some of which may be optional-before being executed (Figure 8.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count).* A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.



**Types of Address Binding :**

Address Binding divided into three types as follows.

**Compile time**. If the compiler is responsible for performing address binding then it is called compile-time address binding. It will be done before loading the program into memory. If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

**Load time.** It will be done after loading the program into memory. This type of address binding will be done by loader. If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

**Execution time**. It will be postponed even after loading the program into memory. The program will be kept on changing the locations in memory until the time of program execution. The dynamic type of address binding done by the processor at the time of program execution. Special hardware must be available for this scheme to work.

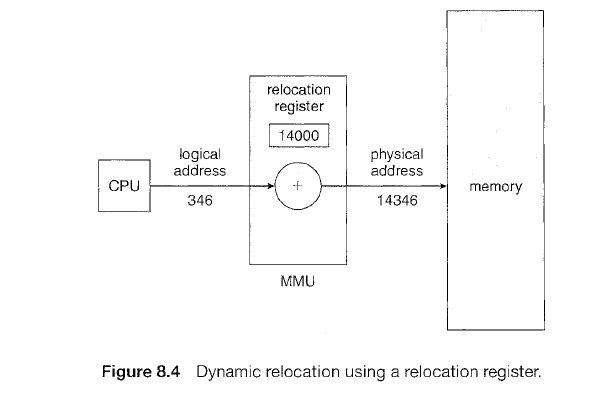
**LOGICAL VERSUS PHYSICAL ADDRESS SPACE**

An address generated by the CPU is commonly referred to as a **logical address** whereas an address seen by the memory unit-that is, the one loaded into the memory address register of the memory-is commonly referred to as a physical address.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address.

The set of all logical addresses generated by a program is a logical address space and the set of all physical addresses corresponding to these logical addresses is a physical address space.

The run-time mapping from logical to physical addresses is done by a hardware device called the Memory Management Unit (MMU).



Base Register or relocation register is the register, acts as a address holder of the base storage location from where the data were stored continuously. The value in the relocation register is *added* to every address generated by a user process at the time the address is sent to memory (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

The user program never sees the *real* physical addresses. The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses.

We now have two different types of addresses: logical addresses (in the range 0 to *max)* and physical addresses (in the range R+ 0 to R+ *max* for a base value R). The user program generates only logical addresses and thinks that the process runs in locations 0 to *max.* However, these logical addresses must be mapped to physical addresses before they are used.

**CONTIGUOUS MEMORY ALLOCATION**

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory.

We need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

In contiguous memory allocation, each process is contained in a single contiguous section of memory. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this scheme, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use.

The method described next is a variable-partition scheme (called MVT). In the scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

Initially, all memory is available for user processes and is considered one large block of available memory a **hole**. Eventually as you will see, memory contains a set of holes of various sizes.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory which the operating system may then fill with another process from the input queue.

At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until finally, the memory requirements of the next process cannot be satisfied -that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage allocation problem** which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The first fit, best fit and worst fit strategies are the ones most commonly used to select a free hole from the set of available holes.

**First fit.** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

**Best fit.** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

**Worst fit.** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

**FRAGMENTATION**

Memory fragmentation can be internal as well as external. In fixed sized partition scheme, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**, unused memory that is internal to a partition.

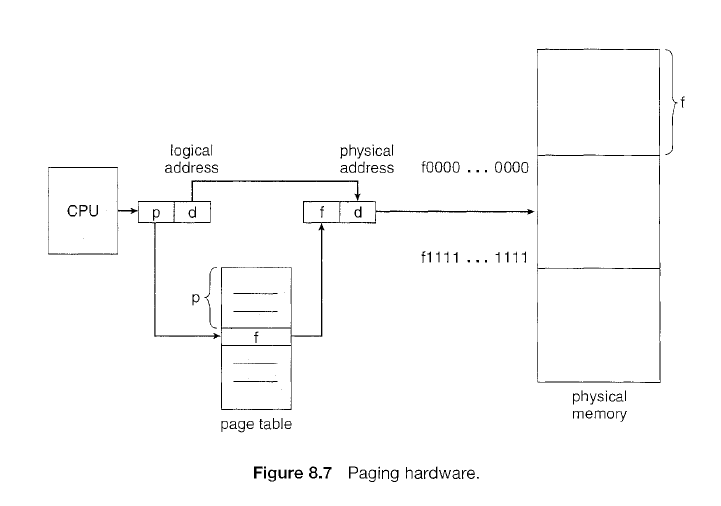
Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50-percent rule.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is possible *only* if relocation is dynamic and is done at execution time. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive. Two complementary techniques are: **paging and segmentation**.

**PAGING**

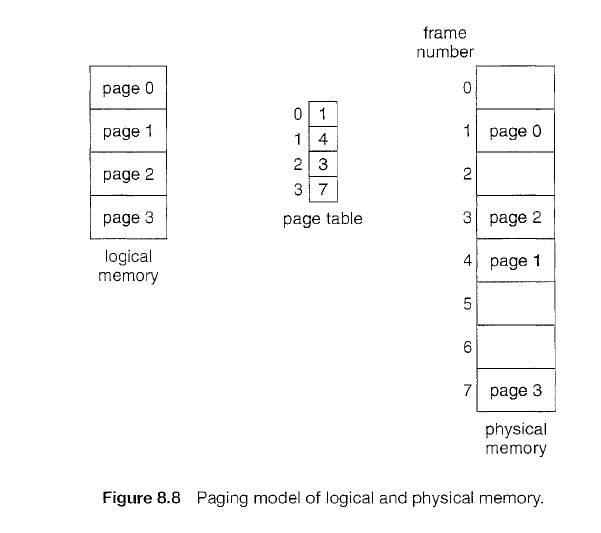
Paging is a memory management scheme that permits physical address space of a process to be non contiguous. Paging avoids external fragmentation and the need for compaction.



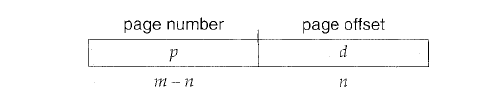
**Basic Method**

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).

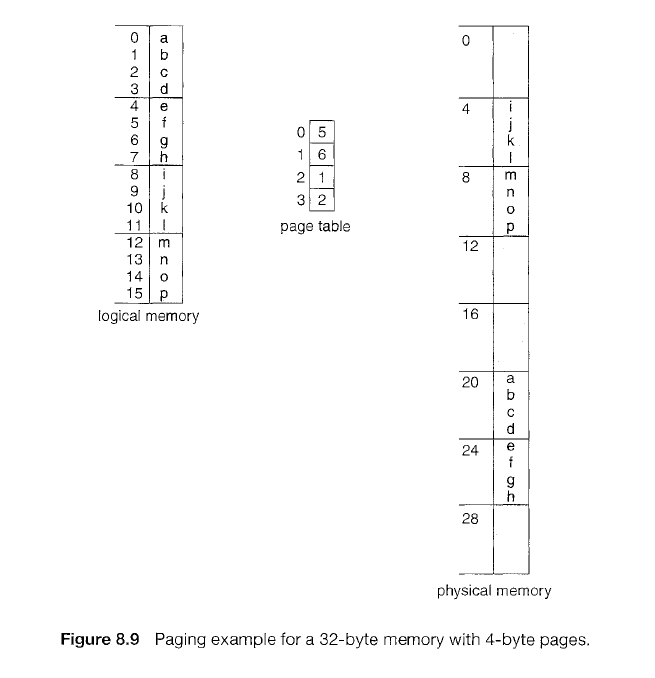
The hardware support for paging is illustrated in Figure 8.7. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 8.8.



The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2m, and a page size is 2n addressing units (bytes or words), then the high-order *m- n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. Thus, the logical address is as follows:



where *p* is an index into the page table and *d* is the displacement within the page.



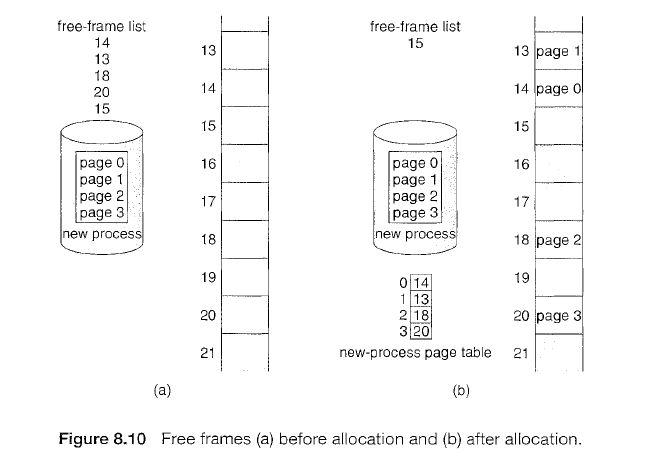
As an example, consider the memory in Figure 8.9. Here, in the logical address, *n=* 2 and *m* = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [ = (5 x 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [ = ( 6 x 4) + 0].

Logical address 13 maps to physical address 9 [ = ( 2 x 4) + 1].

When we use a paging scheme, we have no external fragmentation: *any* free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the *last* frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages (ie, 2048\*35 = 71575) plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need *n* pages plus 1 byte. It would be allocated *n* + 1 frames, resulting in internal fragmentation of almost an entire frame.

This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory. If *n* frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 8.10).



**Paging Hardware**

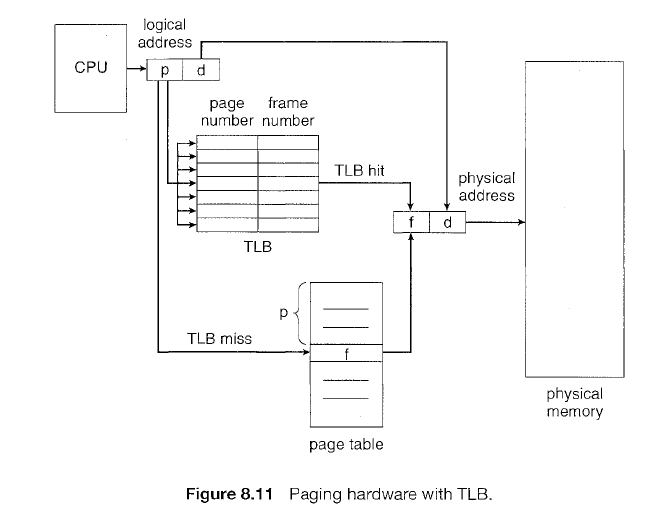
Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values in the process control block.

The use of registers for storing page table is satisfactory if the page table is reasonably small (for example, 256 entries). However, if the page table is very large (for example, 1 million entries), the page table is kept in main memory, and a page table base register (PTBR) points to the page table.

The problem with this approach is the time required to access a user memory location. If we want to access location *i,* we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2.

The standard solution to this problem is to use a special, small, fast lookup hardware cache, called a **translation lookaside buffer (TLB).** The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, (known as a TLB hit) its frame number is immediately available and is used to access memory. If the page number is not in the TLB (known as a TLB miss) a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 8.11). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Several Replacement policies are there for page replacement.



The percentage of times that a particular page number is found in the TLB is called the **Hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. We can find the **effective memory access time**, as:

effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds.

In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

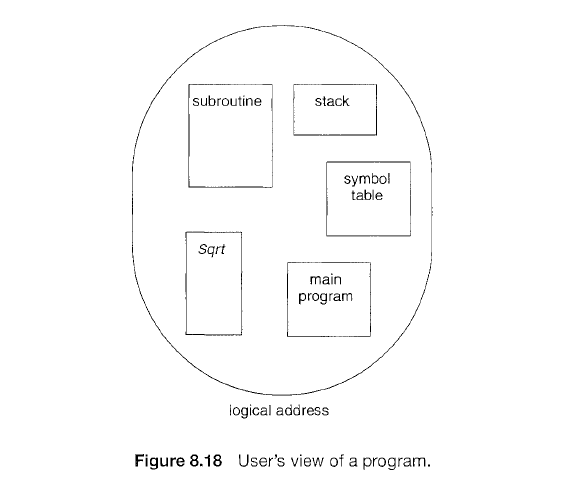
effective access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds.

This increased hit rate produces only a 22 percent slowdown in access time.

**SEGMENTATION**

**Basic Method**

Logical memory can be viewed as a collection of variable-sized segments, with no necessary ordering among segments (Figure 8.18).



Thus a logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

<segment-number, offset>.

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for the following:

The code

Global variables

The heap, from which memory is allocated

The stacks used by each thread

The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

**Hardware**

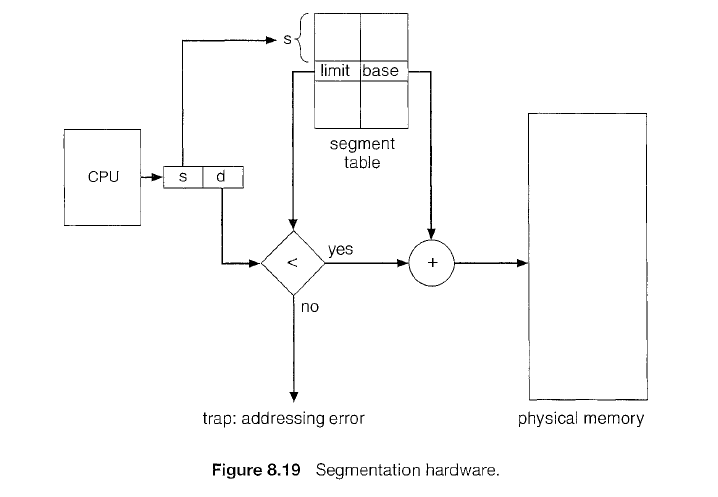
Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a **segment table**. Each entry in the segment table

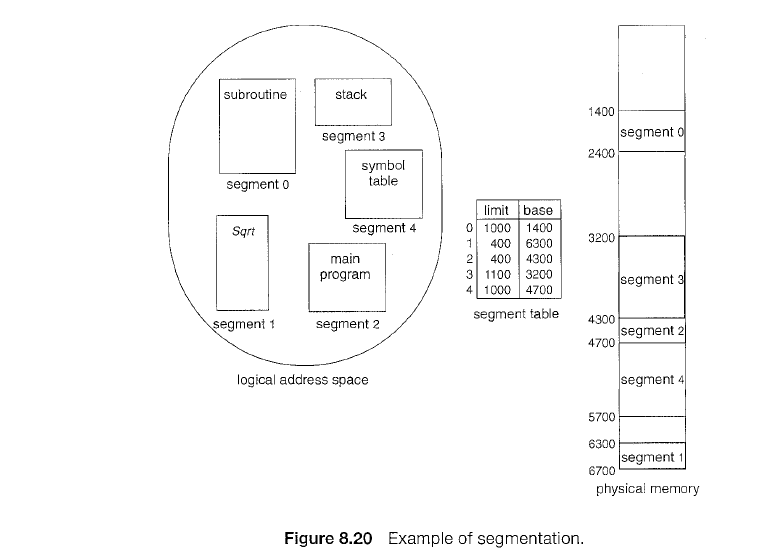
has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.19.

A logical address consists of two parts: a segment number, s, and an offset into that segment, *d.*

The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.



As an example, consider the situation shown in Figure 8.20. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped 

onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

### Advantages of Segmentation

Following are the advantages of segmentation:

1. It is easier to relocate segments compared to the whole address space.
2. Internal fragmentation does not happen.
3. The segment table is smaller in size compared to the page table.
4. It provides protection within the segment.
5. Segment tables use lesser memory than paging
6. Because of the small segment table, the reference to the memory is simple.

**EXERCISES**

1. Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

Given, Logical address space= 64 pages → 2^6

words → offset → 1024 -->2^10

Frames = 32 -->2^5

Logical address = 10 bits + 6 bits = **16 bits**

Physical address = 10-bits + 5-bits = **15 bits**

1. Consider a logical address space of 32 pages with 1,024 words per page, mapped onto a physical memory of 16 frames.

a. How many bits are required in. the logical address?

b. How many bits are required in the physical address?

Given, Logical address space= 32 pages → 2^5

words → offset → 1024 -->2^10

Frames = 16 -->2^4

Logical address = 10 bits + 5 bits = **15 bits**

Physical address = 10-bits + 4-bits = **14 bits**

1. Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

**First fit**

212 K is put in 500 K partition.

417 K is put in 600 K partition.

112 K is put in 288 K partition. (New partition 288 K = 500 K - 212 K)

426 K must wait.

**Best-fit**

212 K is put in 300 K partition.

417 K is put in 500 K partition.

112 K is put in 200 K partition.

426 K is put in 600 K partition.

**Worst-fit**

212 K is put in 600 K partition.

417 K is put in 500 K partition.

112 K is put in 388 K partition. (600 K - 212 K)

426 K must wait.

**In this example Best-fit is the best solution.**

1. Consider the following segment table:

Segment Base Length

0 219 600

1 2300 14

2 90 100

3 1327 580

4 1952 96

What are the physical addresses for the following logical addresses?

a. 0, 430

b. 1, 10

c. 2, 500

d. 3, 400

e. 4, 112

(a) 219 + 430 = 649

(b) 2300 + 10 = 2310

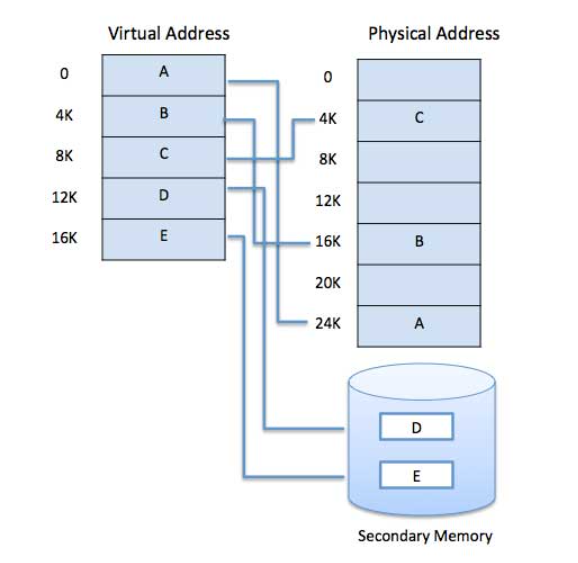
(c) illegal reference; traps to operating system  //is 100>500?  no , its illegal

(d) 1327 + 400 = 1727

(e) illegal reference; traps to operating system // is 96>112? no

**VIRTUAL MEMORY**

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory. In this scheme, user can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.

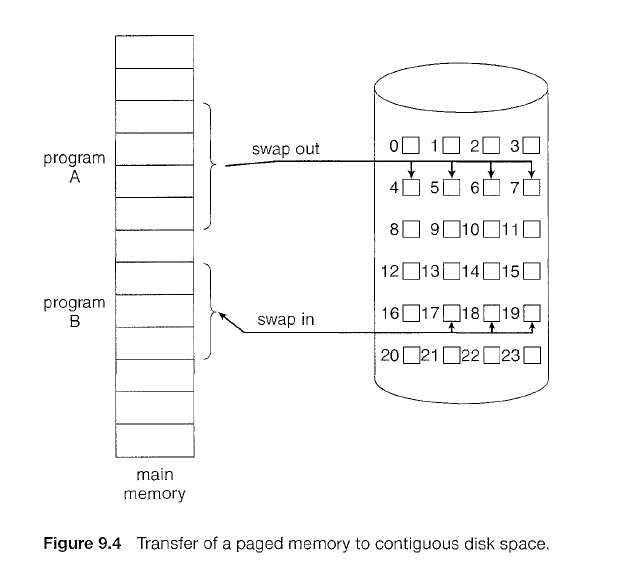


Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available. Virtual memory is commonly implemented by demand paging.

**DEMAND PAGING**

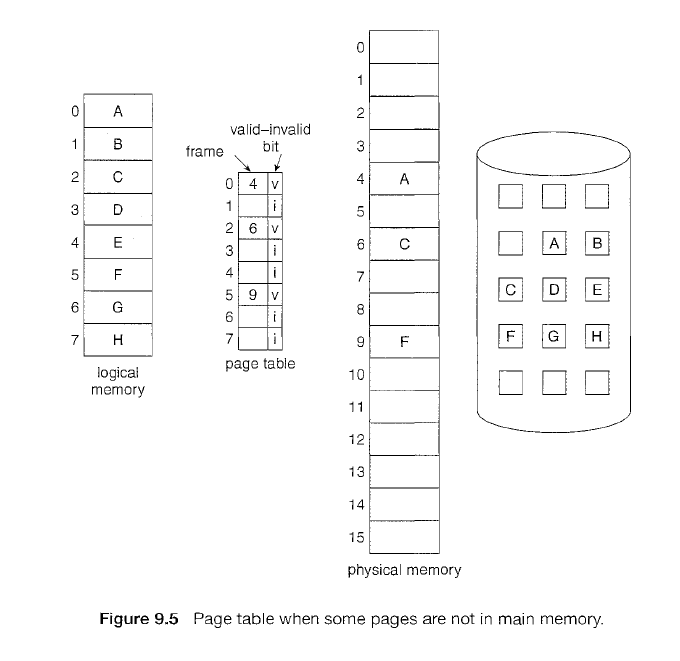
With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping (Figure 9.4) where processes reside in secondary memory (usually a disk).

When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, we use the term *pager,* rather than *swapper,* in connection with demand paging.

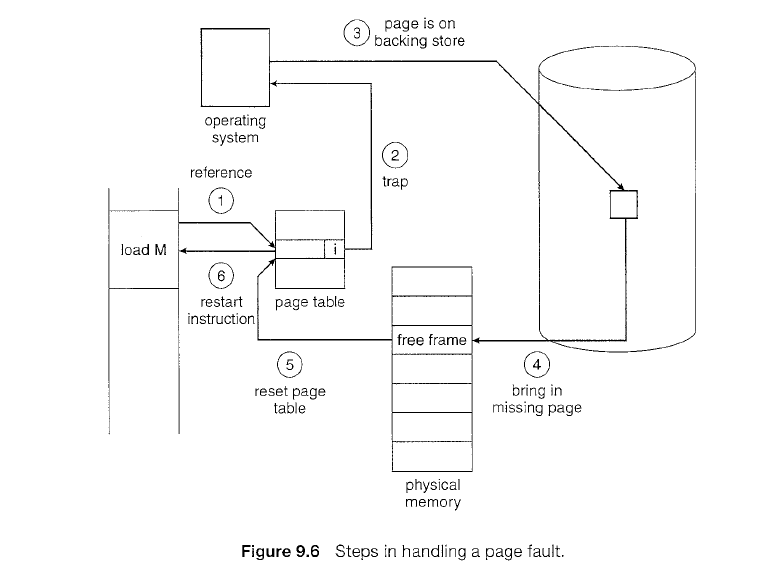


Instead of swapping in a whole process, the pager brings only those pages which are currently needed into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid -invalid bit scheme can be used for this purpose. When this bit is set to “valid”, the associated page is both legal and in memory. If the bit is set to "invalid” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure 9.5.



Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 9.6):



* We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access. If the reference was invalid, we terminate the process.
* We find a free frame (by taking one from the free-frame list, for example).
* We schedule a disk operation to read the desired page into the newly allocated frame.
* When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
* We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

**Performance of Demand Paging**

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the effective access time for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma,* ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let *p* be the probability of a page fault (0 <= *p* <= 1). The effective access time is then

**effective access time= (1 - *p)* x *ma* + *p* x page fault time.**

To compute the effective access time, we must know how much time is needed to service a page fault.

**PAGE REPLACEMENT ALGORITHMS**

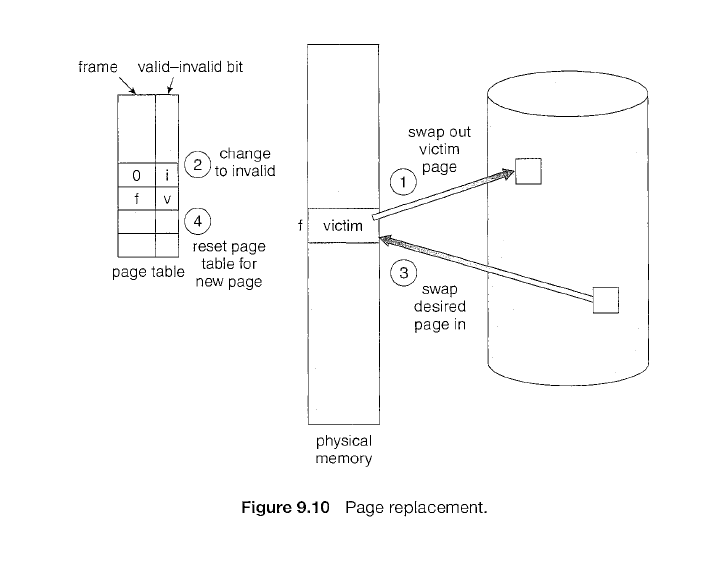
While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are *no* free frames on the free-frame list; all memory is in use. The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. So this option is not the best choice.

The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. The most common solution to this problem is **page replacement**.

**Basic Page Replacement**

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.10). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   1. If there is a free frame, use it.
   2. If there is no free frame, use a page-replacement algorithm to select a **victim frame**
   3. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.



Notice that, if no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.

When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has *not* been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half *if* the page has not been modified.

When page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve this problem is an important task. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**. We can generate reference strings artificially (by using a random-number generator, for example).

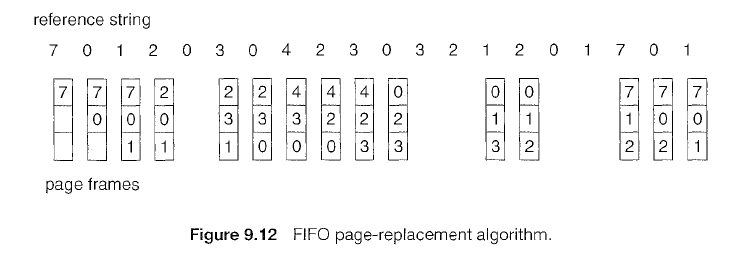
**FIFO PAGE REPLACEMENT**

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

**Example**

Consider the following reference string for a memory with three frames:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



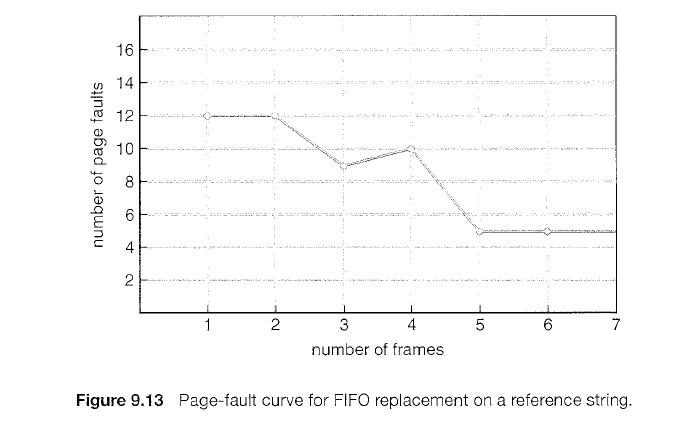
For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 9.12. There are fifteen faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string:

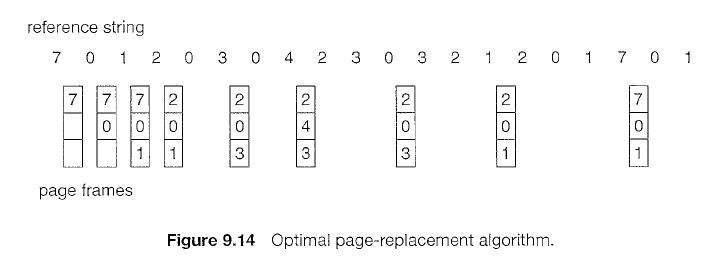
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Figure 9.13 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine). This most unexpected result is known as **Belady’s anomaly**; for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.



**OPTIMAL PAGE REPLACEMENT**

Here, when a page fault occurs, replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.



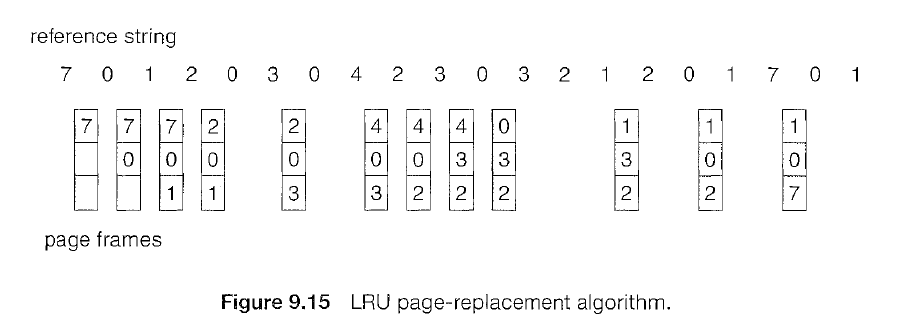
For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.14. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.

The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

**LRU PAGE REPLACEMENT**

In LRU page replacement algorithm, we replace the page that *has not been used* for the longest period of time. LRU replacement associates with each page the time of that page's last use.

The result of applying LRU replacement to our example reference string is shown in Figure 9.15.



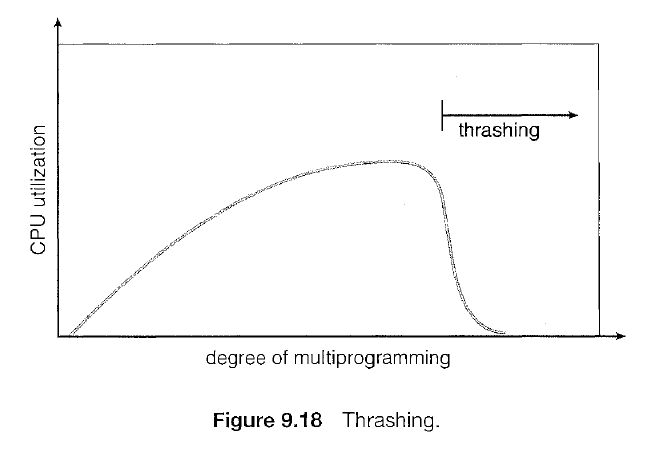
The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.

The LRU policy is often used as a page-replacement algorithm and is considered to be good.

**THRASHING**

If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called **thrashing**. *A* process is thrashing if it is spending more time paging than executing.

This phenomenon is illustrated in Figure 9.18, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.



**Exercices**

1. Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, for three frames. Also find the page fault rate.

* + 1. LRU replacement
    2. FIFO replacement
    3. Optimal replacement