MODULE 3

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: -

To increase in performance, keep several processes in memory—that is, we must share memory.

Memory is central to the operation of a modern computer system.

Memory consists of a *large array of bytes*, each with its own *address*. The CPU fetches instructions from memory according to the value of the program counter.

A typical instruction-execution cycle, for example,

first fetches an instruction from memory.

The instruction is then decoded and may cause operands

to be fetched from memory.

After the instruction has been executed on the operands, results may be stored back in memory.

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. Registers that are built into the CPU are generally accessible within *one cycle* of the CPU clock. The same cannot be said of main memory, which is accessed via a transaction on the *memory bus*. Completing a memory access may take *many cycles* of the CPU clock.

Each process has a *separate* memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.

To separate memory spaces, we need the ability to determine the range of *legal addresses* that the process may access and to ensure that the process can access only these legal addresses.

We can provide this protection by using two registers, usually a ***base*** and a ***limit***.

The ***base register*** holds the smallest legal physical memory address; the ***limit register*** specifies the size of the range.

For example, if the *base* register holds *300040* and the *limit* register is *120900*, then the program can legally access all addresses from *300040* through *420939* (inclusive).

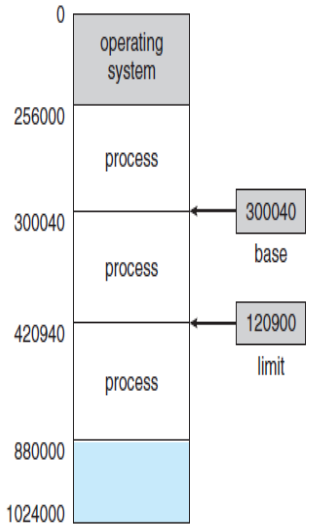


Figure 8.1 A base and a limit register define a logical address space.

Protection of memory space is accomplished by having the *CPU* hardware *compare* every address generated in user mode with the *registers*. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a *trap* to the operating system, which treats the attempt as a *fatal error* (Figure 8.2).

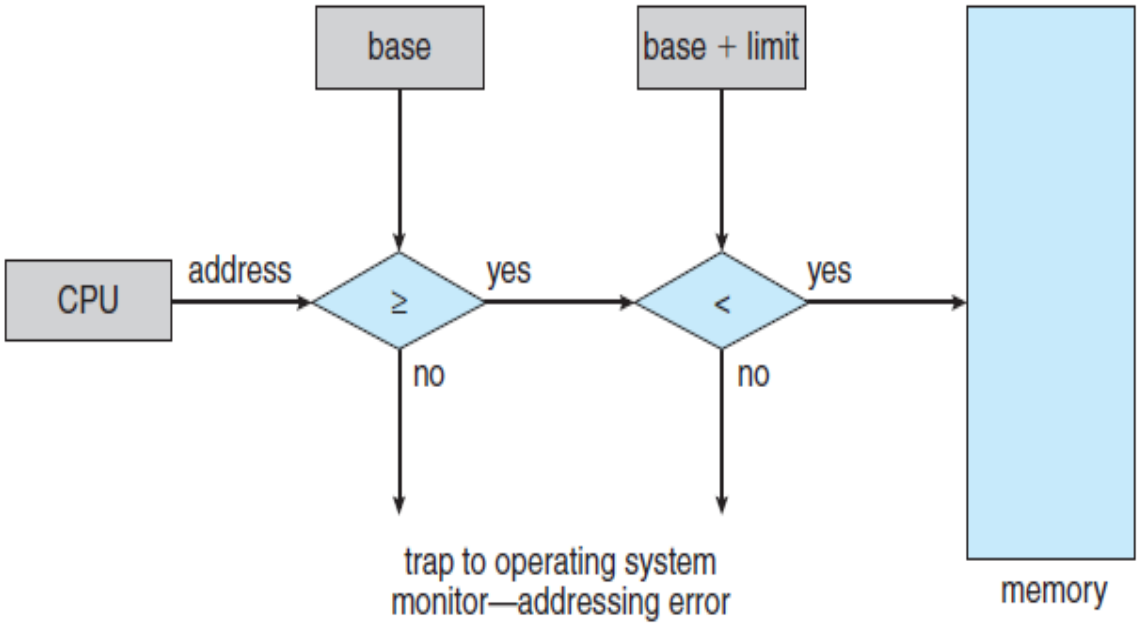


Figure 8.2 Hardware address protection with base and limit registers.

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Logical versus Physical Addresses: -

An address generated by the *CPU* is commonly referred to as a **logical address**.

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 set of all logical addresses generated by a program is a *logical address space*.

The set of all physical addresses corresponding to these logical addresses is a *physical address space*.

The run-time *mapping* from *virtual (logical) to physical addresses* is done by a hardware device called the *memory-management unit (MMU).*

Many different methods are used to accomplish such mapping.

The mapping with a simple MMU scheme that is a generalization of the *base-register* scheme is shown in fig

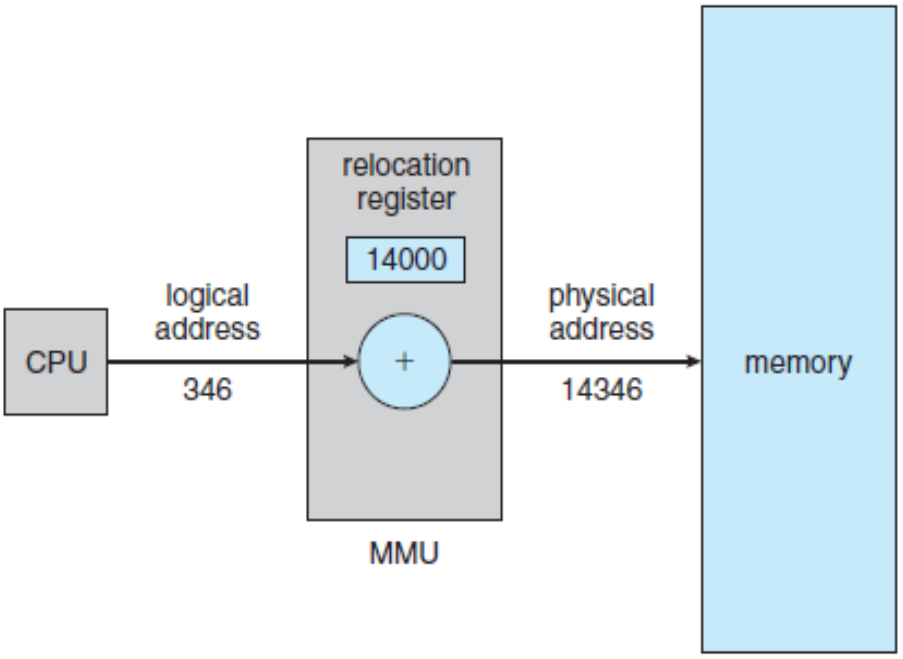


Figure 8.4 Dynamic relocation using a relocation register.

The base register is now called a *relocation register*. 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.

That is these *logical addresses must be mapped to physical addresses before they are used.*

Address Binding:

The Association of program instruction and data to the actual physical memory locations is called the Address Binding.

Types of Address Binding:

Address Binding divided into three types as follows.

1. Compile-time Address Binding
2. Load time Address Binding
3. Execution time Address Binding

Compile-time Address Binding:

- 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.
- The compiler requires interacts with an OS memory manager to perform compile-time address binding.

Load time Address Binding:

- It will be done after loading the program into memory.
- This type of address binding will be done by the OS memory manager i.e loader.

Execution time or dynamic Address Binding:

- 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.

Contiguous Memory Allocation: -

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.

In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.

Each partition may contain exactly one process. Thus, the degree of *multiprogramming* is bound by the *number of partitions*.

In this *multiple partition* method, 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.

In the **variable-partition** 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*.

At any given time, then, we have a list of available block sizes and an input queue. 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.

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. This procedure is a particular instance of the general *dynamic storage allocation problem*,

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, which may be more useful than the smaller leftover hole from a best-fit approach.

1. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of 12KB, 10KB, 9KB for *first fit*? Now repeat the question for *best fit, and worst fit*.

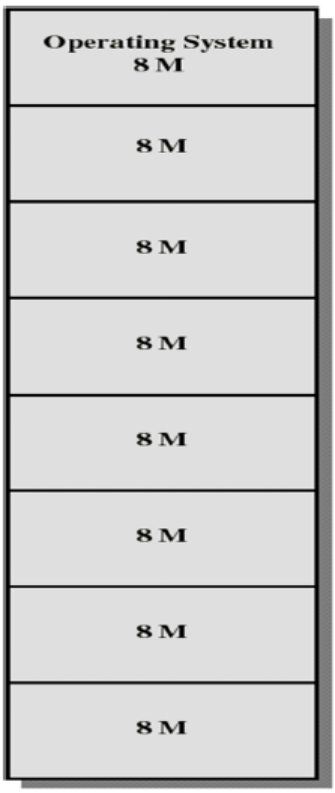
10 10

Fragmentation: -

Consider *a fixed-size partition of equal size*. In this case *main memory utilization* is extremely *inefficient.* Any program, that is small, occupies an entire partition.

In this example; there may be a program whose length is less than 2Mbytes. Yet it occupies an 8Mbyte partition whenever it is swapped in.

This phenomenon, in which there is a *wasted space internal to a partition* due to the fact that *the block of data loaded is smaller than the partition*, is referred to as **internal fragmentation**.



Equal-size partitions

To overcome some of the difficulties with fixed partitioning, an approach is known as *dynamic partitioning*.

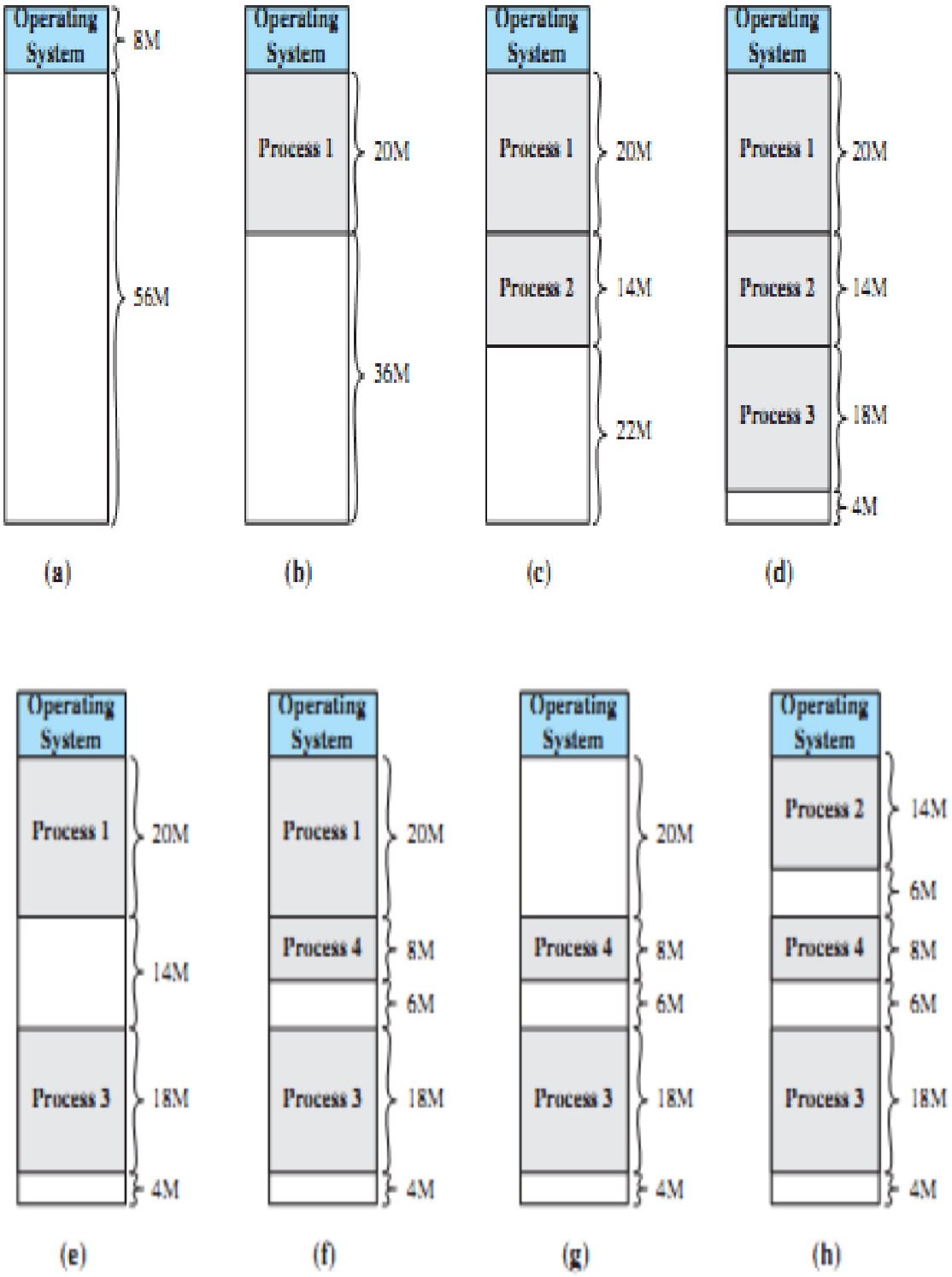
With dynamic partitioning, the partitions are of *variable length and number*. When a process is brought into main memory, it is allocated exactly *as much memory as it requires and no more.*

Consider an example using 64Mbytes of main memory is shown in fig. Initially main memory is empty except for OS in fig 2.8(a).

The first three processes are loaded in, starting where the OS ends and occupying just enough spaces for each process fig 2.8 (b), (c) and (d).

This leaves a *hole* at the end of memory that is too small for a fourth process.

At some point none of the processes in memory is ready. The OS swaps out process 2 (fig 2.8 (e)), which leaves sufficient room to load a new process 4 (fig 2.8(f)). Because process 4 is smaller than process 2, another hole is created.



Later a point is reached at which none of the processes in main memory is ready, but process 2 in the ready-suspend state is available.

Because there is insufficient room in memory process 2, the OS swaps process 1 out (fig 2.8(g)) and swaps process 2 back in (fig. 2.8(h)).

In the above example, it leads to a situation in which there are *a lot of small holes* in memory. As time goes on memory becomes more and more fragmented and memory utilization declines.
This phenomenon is referred to as **external fragmentation**, indicating that the memory that is external to all partitions becomes increasingly fragmented.
That is when enough total memory space exists to satisfy a request, but it is not contiguous, storage is fragmented into a large number of holes. This wasted space not allocated to any partition is called external fragmentation.

One technique for overcoming external fragmentation is **compaction**.
From time to time, the OS shifts the processes so that they are contiguous and so that all of free memory is together in one block.
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.*

Another possible solution to the *external-fragmentation* problem is to permit the logical address space of the processes to be *noncontiguous,* thus allowing a process to be allocated physical memory wherever such memory is available.
Two complementary techniques achieve this solution: ***segmentation*** and ***paging*.**

Segmentation: -

The user's view of memory is not the same as the actual physical memory. The hardware could provide a memory mechanism that *mapped* the programmer's view to the actual physical memory. *Segmentation* provides such a mechanism.

In programmers view memory as a collection of *variable-sized segments,* with no necessary ordering among the segments shown in fig 8.7
When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions.
It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name.
The programmer talks about "the stack," "the math library," and "the main program" without caring *what addresses in memory* these elements occupy.

Elements within a segment are identified by their *offset* from the beginning of the segment: the first statement of the program

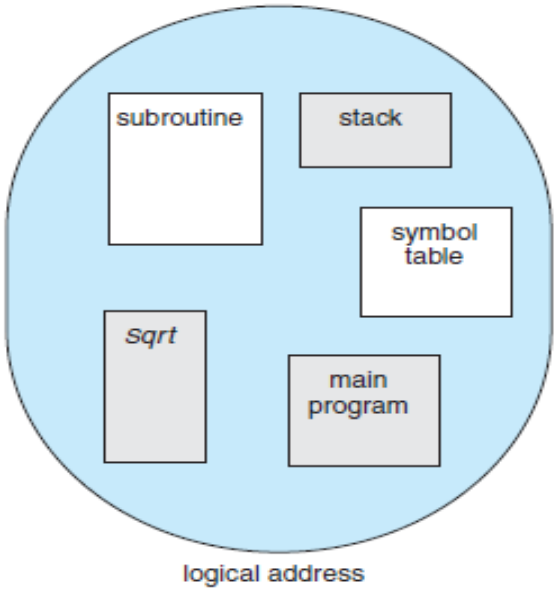


Figure 8.7 Programmer's view of a program.

Segmentation is a memory-management scheme that supports this programmer view of memory.

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 programmer therefore specifies each address by two quantities: a ***segment name*** and an ***offset.***

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*>.

Segmentation Hardware: -

To map two-dimensional user-defined addresses into one-dimensional physical addresses and 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.

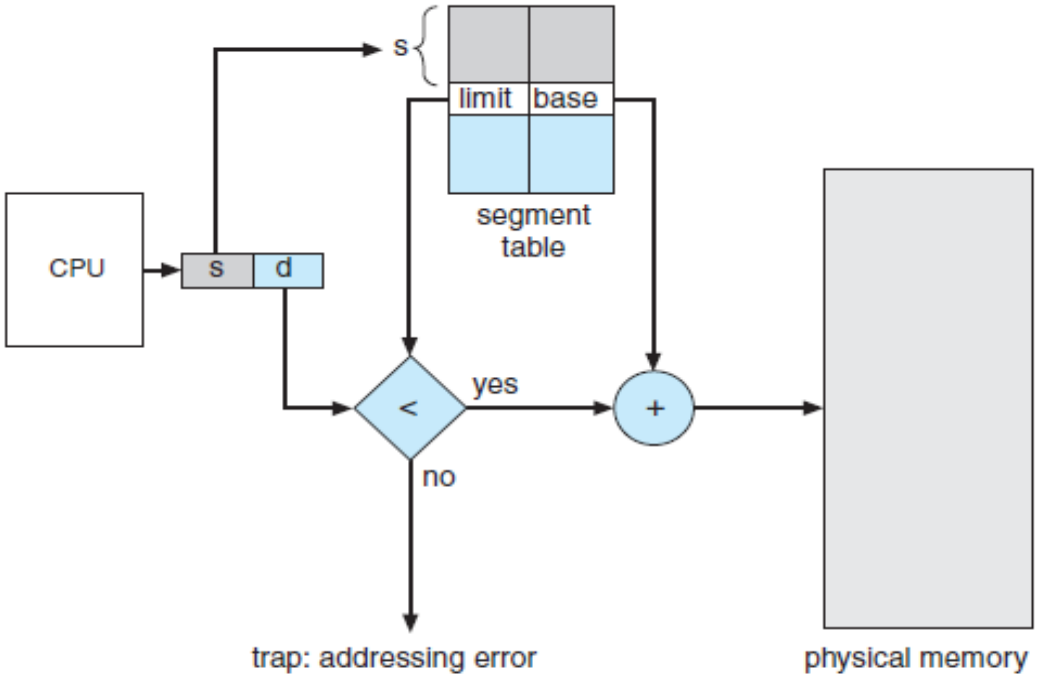


Figure 8.8 Segmentation hardware.

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.

Consider the situation shown in Figure 8.9. 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 to3200 (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 1,000 bytes long byte.

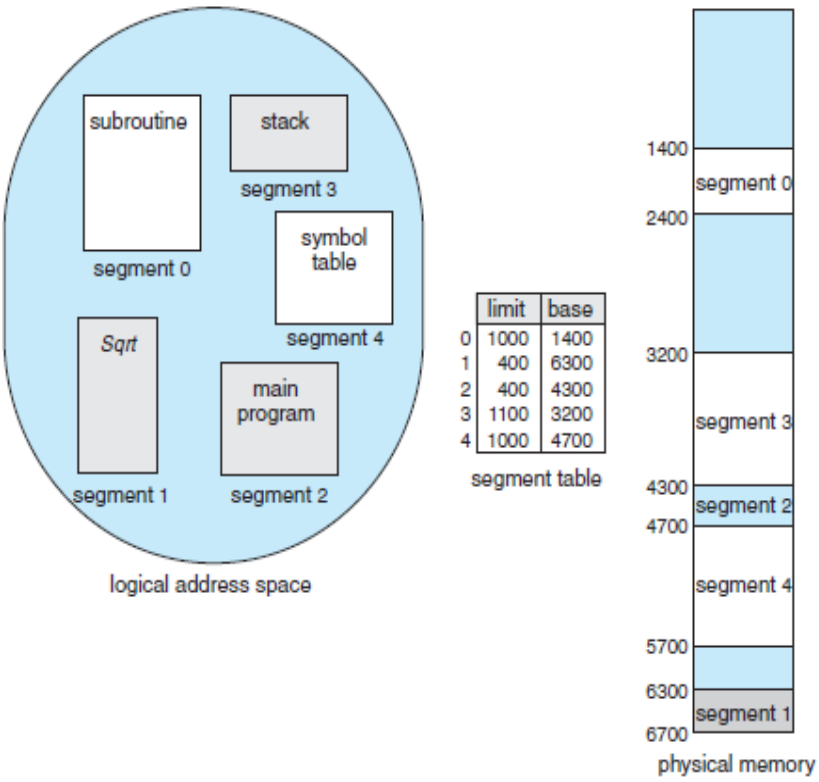


Figure 8.9 Example of segmentation.

Paging: -

Paging is another memory-management scheme that avoids external fragmentation.

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 backing store is divided into fixed-sized blocks that are the same size as the memory frames

The hardware support for paging is illustrated in Figure 8.10

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.

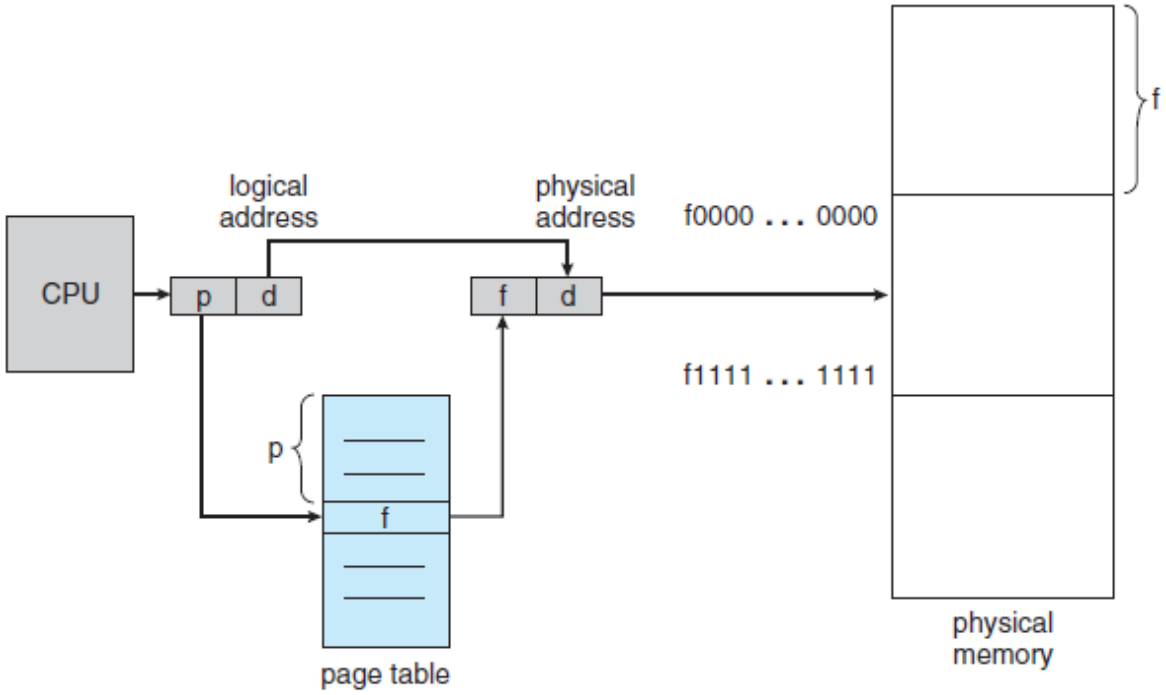


Figure 8.10 Paging hardware.

The paging model of memory is shown in Figure 8.11

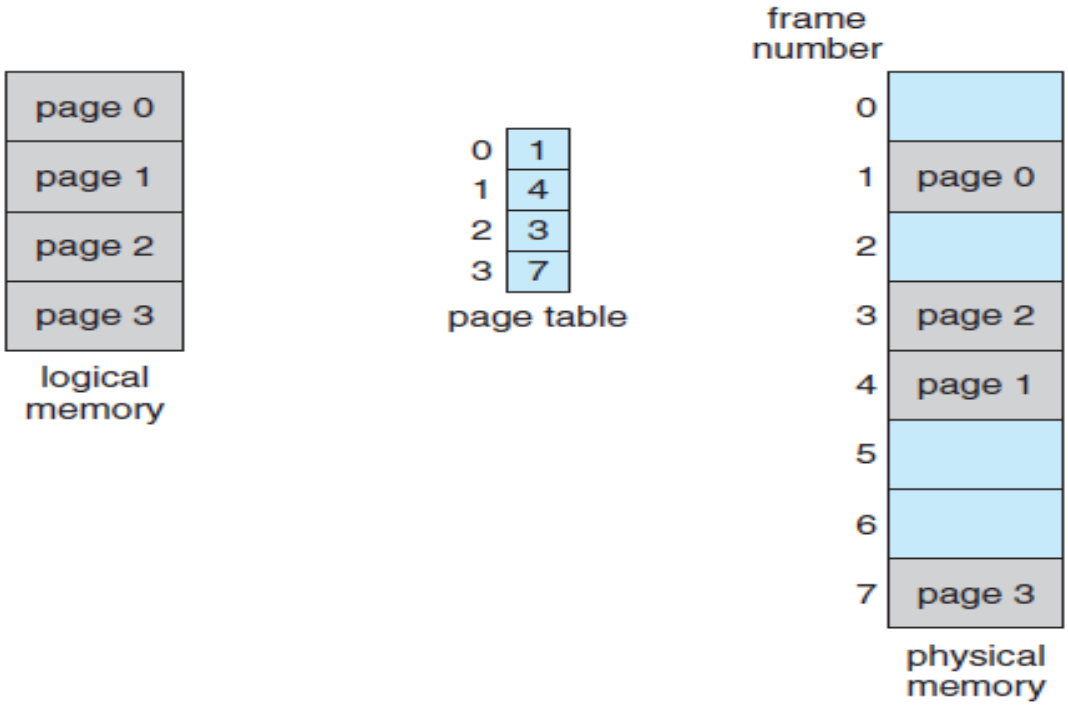


Figure 8.11 Paging model of logical and physical memory.

The page size (like the frame size) is defined by the hardware.

The *size of a page is a power of 2*, varying between 512 bytes and 1 GB 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.*

When a process arrives in the system to be executed, its size, expressed in *pages*. 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 as shown in figure 8.13

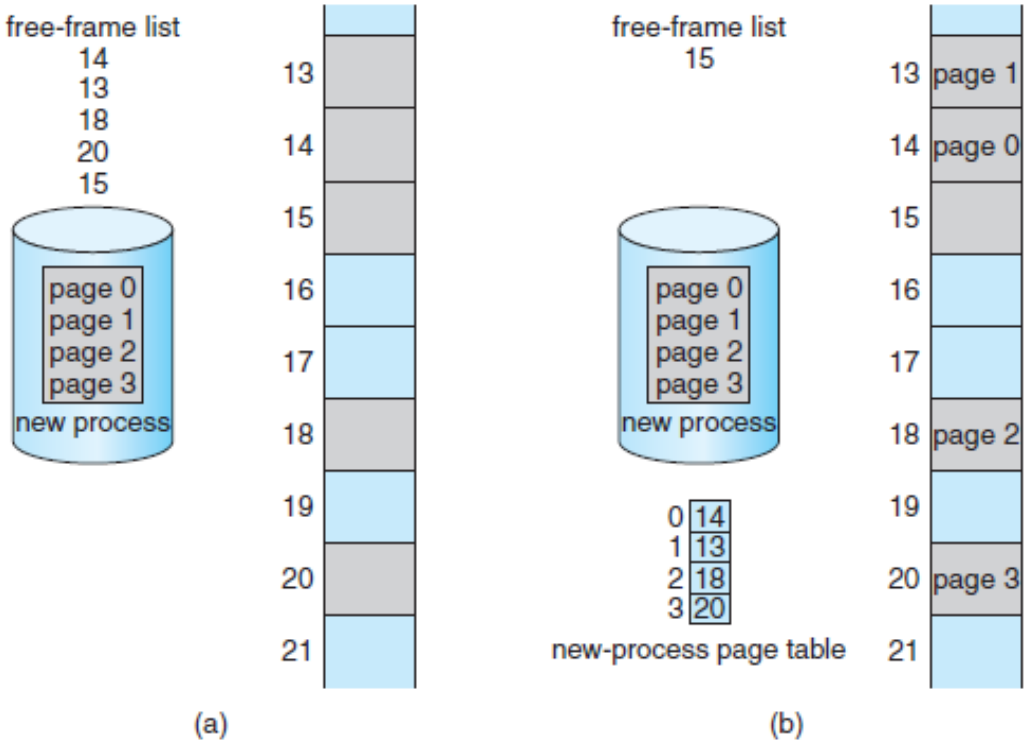


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

Virtual Memory: -

Virtual memory is a technique that *allows the execution of processes that are not completely in memory.*

One major advantage of this scheme is that *programs can be larger than physical memory.*

Virtual memory also allows processes to share files easily and to implement shared memory. The entire program is needed, it may not all be needed at the same time.

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large *virtual* address space, simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM.

Demand Paging: -

Consider how an executable program might be loaded from disk into memory.

One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially need the entire program in memory.

Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user.

An alternative strategy is to

load pages only as they are needed.

This technique is known as *demand paging* and is commonly used in virtual memory systems. With demand-paged virtual memory, *pages are loaded only 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, though, we use a *lazy swapper*.

A lazy swapper never swaps a page into memory unless that page will be needed.

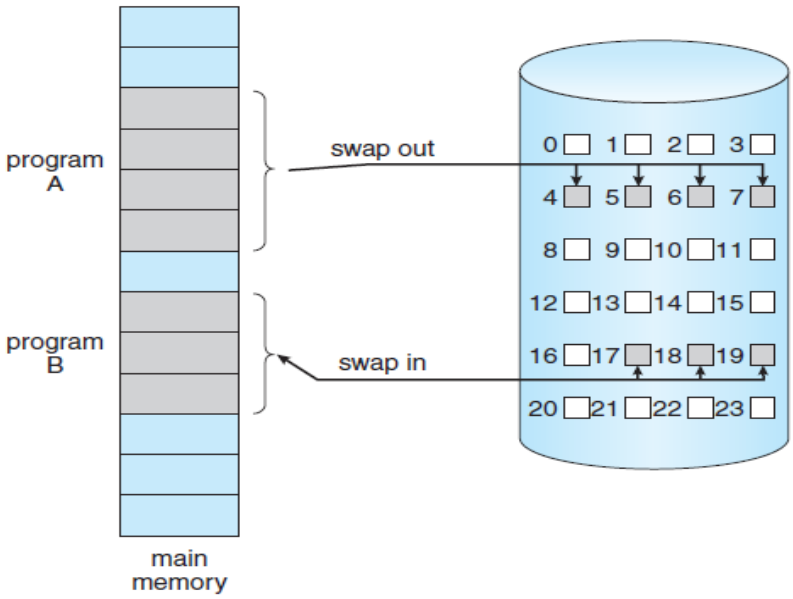


Figure 9.4 Transfer of a paged memory to contiguous disk space.

When a process is to be swapped in, the pager (swapper) guesses which pages will be used before the process is swapped out again.

Instead of swapping in a whole process, the pager brings only those pages 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 addressspace 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.

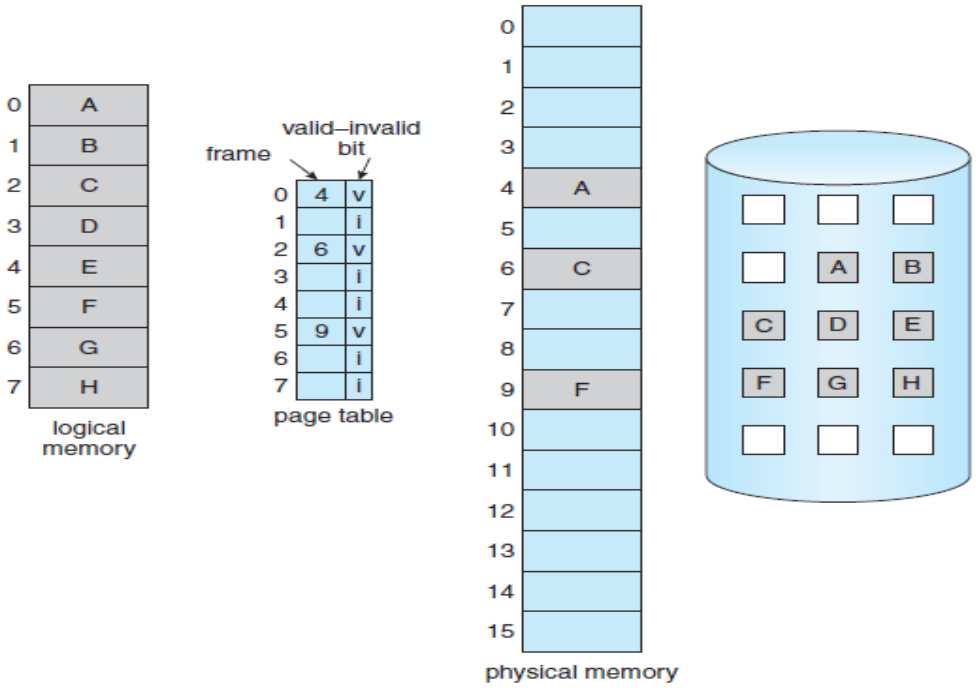


Figure 9.5 Page table when some pages are not in main memory.

If the process executes and accesses pages that are *memory resident,* execution proceeds normally.

But if the process tries to access a page that was not brought into memory, access to a page marked invalid causes a page fault.

The procedure for handling this page fault is in Figure 9.6:

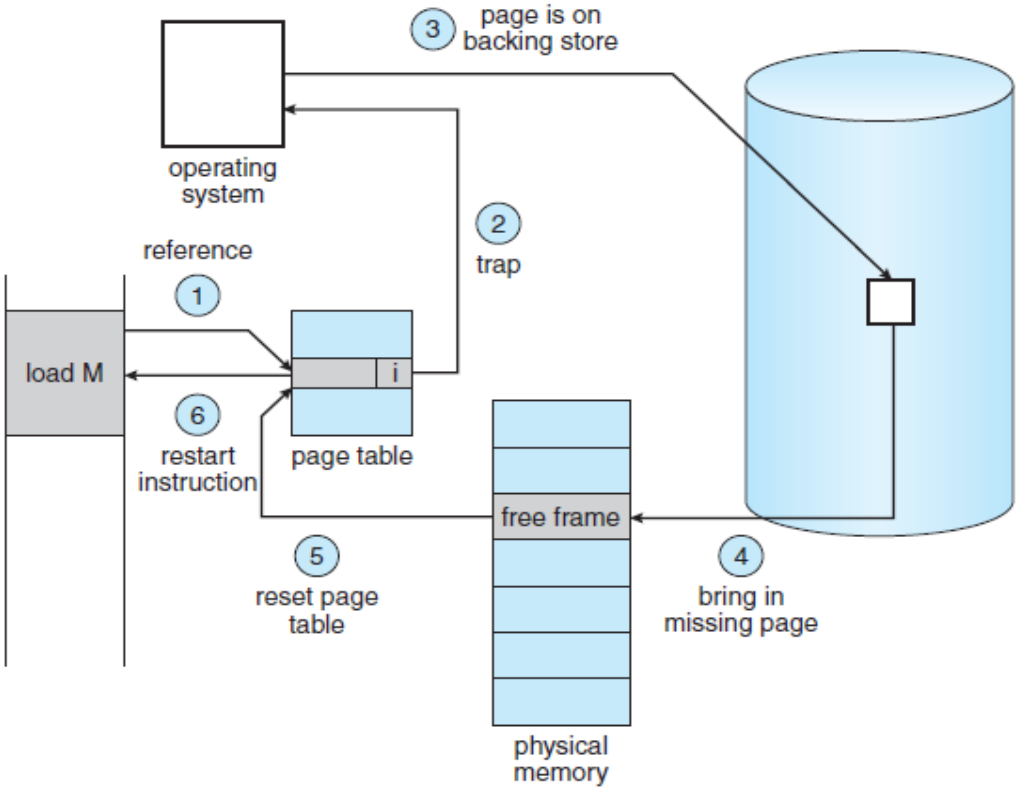


Figure 9.6 Steps in handling a page fault.

1. 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.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. 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.
6. 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.

Pure Demand Paging: -

When starting execution of a process with no pages in memory, 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 schema is *pure demand paging*.

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 (disk space) and changing the page table (and all other tables) to indicate that the page is no longer in memory as shown in fig 9.10. We can now use the freed frame to hold the page for which the process faulted

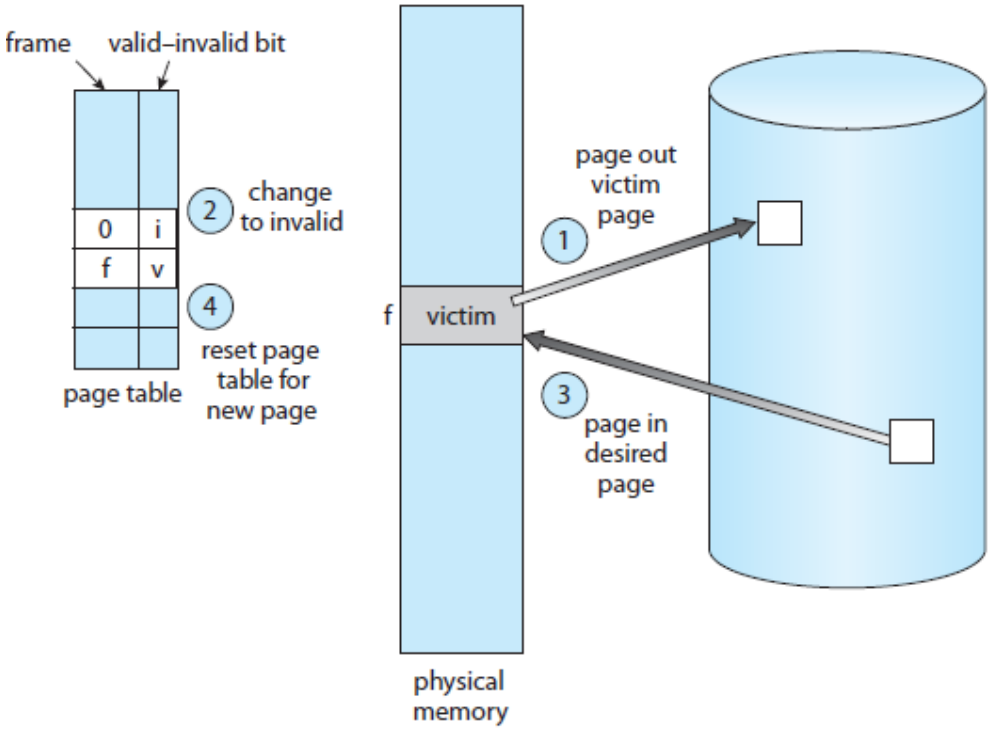


Figure 9.10 Page replacement.

There are many different page-replacement algorithms.

Every operating system probably has its own replacement scheme.

FIFO Page Replacement Algorithm: -

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.

Consider the reference string

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

for a memory with three frames.

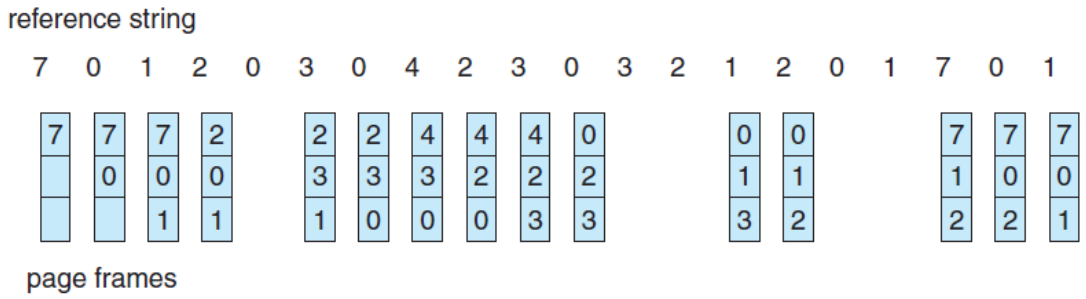


Figure 9.12 FIFO page-replacement algorithm.

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, since it is now first in line. 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

Consider the following reference string:

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

(i). for a memory with four frames and (ii). three frames

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 Algorithm: -

The algorithm that has *the lowest page-fault rate* of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN.

It is simply this:

Replace the page that will not be used for the longest period of time.

Consider the reference string

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

for a memory with three frames.

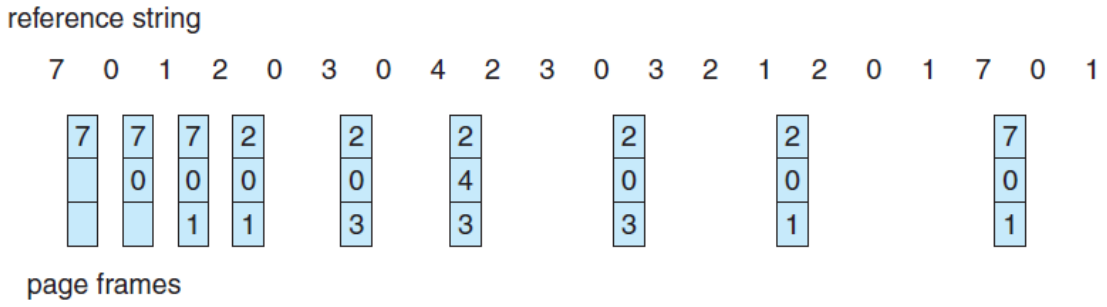


Figure 9.14 Optimal page-replacement algorithm.

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.

With only *nine page faults*, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

LRU Page Replacement: -

The key distinction between the FIFO and OPT algorithms is that the FIFO algorithm uses the *time when a page was brought into memory*, whereas the OPT algorithm uses *the time when a page is to be used*.

Least recently used (LRU) algorithm 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

Consider the reference string

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

for a memory with three frames.

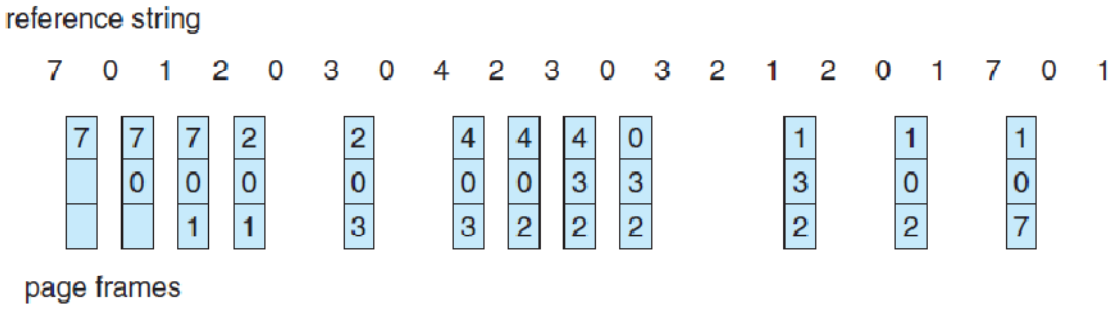


Figure 9.15 LRU page-replacement algorithm.

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. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

Thrashing: -

Thrashing is a condition or a situation when the system is spending a major portion of its time servicing the page faults, but the actual processing done is very negligible.

Thrashing has an impact on the operating system's execution performance.

The basic concept involved is that if a process is allocated too few frames, then there will be too many and too frequent page faults. As a result, no valuable work would be done by the CPU, and the CPU utilization would fall drastically.

The long-term scheduler would then try to improve the CPU utilization by loading some more processes into the memory, thereby increasing the degree of multiprogramming. Unfortunately, this would result in a further decrease in the CPU utilization, triggering a chained reaction of higher page faults followed by an increase in the degree of multiprogramming, called thrashing.