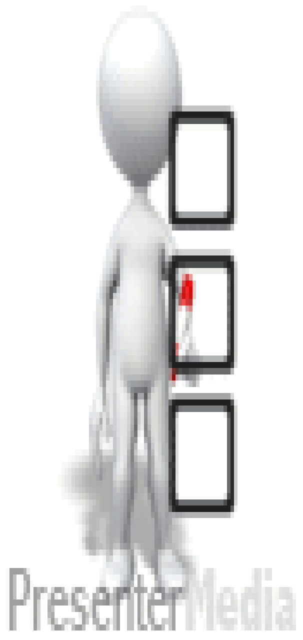
Unit-4

Memory Management

Roadmap

Basic requirements of Memory Management

- Memory Partitioning
- Simple Paging and Segmentation
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- Address translation in Paging
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)



The need for memory management

<u>**Memory Management:-**</u>

Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

- Introduce by pointing out that in a uniprogramming system, main memory is divided into two parts:
  - one part for the operating system (resident monitor, kernel) and
  - one part for the program currently being executed.
- In a multiprogramming system, the “user” part of memory must be further subdivided to accommodate multiple processes.
- Emphasise that memory management is vital in a multiprogramming system.
- If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle.
- Thus memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

Addressing

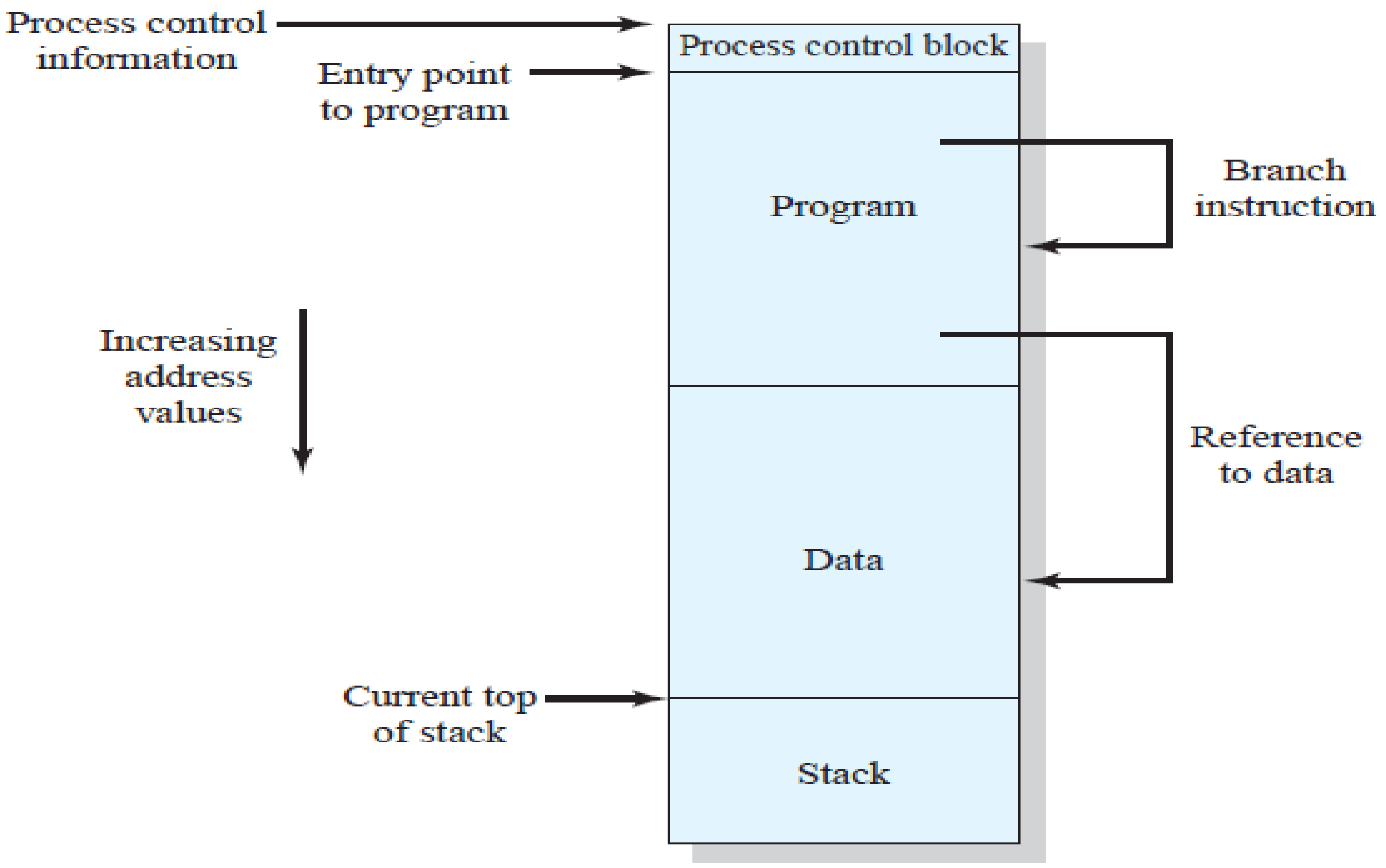


Figure 7.1 **Addressing Requirments for a Process**

The memory management Requirements

- Following are the five requirements that memory management is intended to satisfy:-
  - Relocation
  - Protection
  - Sharing
  - Logical organisation
  - Physical organisation

SRPLP

(1) Relocation:-

- The programmer does not know where the program will be placed in memory when it is executed,
- It may be swapped to disk and return to main memory and when it is next swapped

  back in, it will not place in the same main memory region as before,
- But we may need to **relocate the process to a different area of memory.**

  (relocated).
- Memory references must be translated to the actual physical memory address

(2) Protection:-

- Processes should not be able to reference memory locations in another process without permission.
- In one sense, satisfaction of the relocation requirement increases the difficulty of satisfying the protection requirement.
- The location of a program in main memory is unpredictable so it is impossible to check absolute addresses at compile time.
- The memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)

(3) Sharing

- Allow several processes to access the same portion of memory.
- Better to allow each process access to the same copy of the program rather than have their own separate copy.
- The memory management system must therefore allow controlled access to shared areas of memory without compromising essential protection

(4) Logical Organization:-

- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules among processes
- Segmentation helps here

(5) Physical Organization:-

- Cannot leave the programmer with the responsibility to manage memory.
- Memory available for a program plus its data may be insufficient.
- Overlaying allows various modules to be assigned the same region of Memory but is time consuming to program.
- In a multiprogramming environment, the programmer does not know how much space will be available.
- The task of moving information between the two levels of memory should be a system responsibility.

Memory management Terms

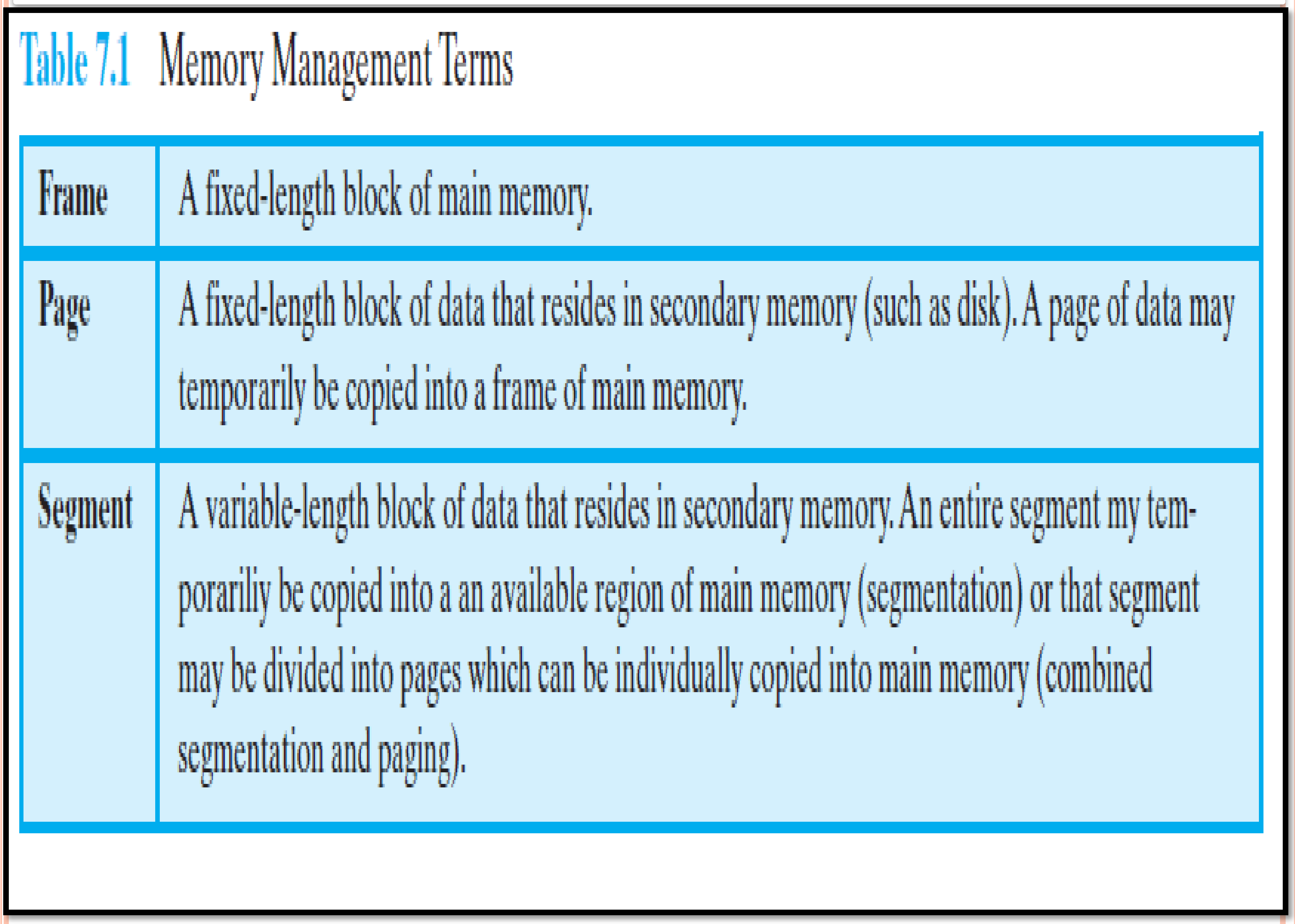
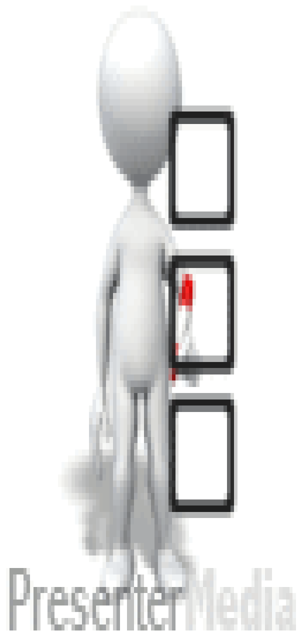


Table 7.1 Memory Management Terms

Frame	A fixed-length block of main memory.
Page	A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.
Segment	A variable-length block of data that resides in secondary memory. An entire segment my temporariliy be copied into a an available region of main memory (segmentation) or that segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

Roadmap

- Basic requirements of Memory Management
- **Memory Partitioning**
- Simple Paging and Segmentation
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- Address translation in Paging
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)



MEMORY PARTITIONING

- The principal operation of memory management is to bring processes into main memory for execution by the processor.
- In almost all modern multiprogramming systems, this involves a sophisticated scheme known as virtual memory.
- Virtual memory is, in turn, based on the use of one or both of two basic techniques: segmentation and paging
- we must prepare the ground by looking at simpler techniques that do not involve virtual memory.
- One of these techniques, is partitioning.
- Two partitioning methods are available:-
  - Fixed Partitioning and
  - Dynamic Partitioning.

(1) Fixed Partitioning

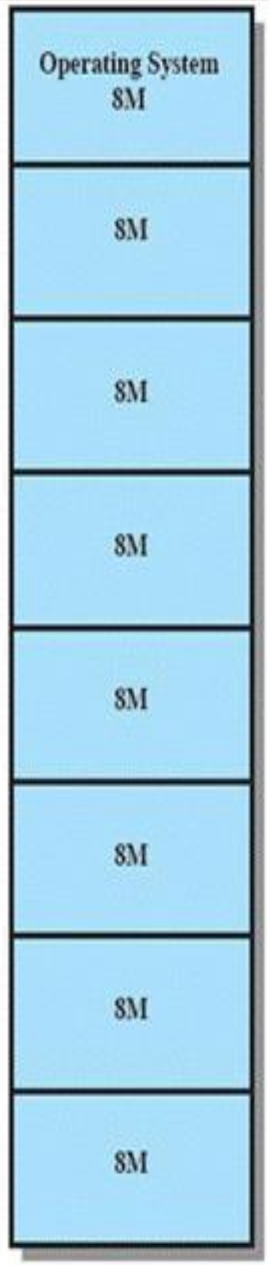
(1) Equal-size partitions (see fig 7.3a)

- The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries.
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
- If all partitions are full and no process is in the Ready or Running state, the operating system can swap a process out of any of the partitions and load in another process

Placement Algorithm Equal-size

- **Equal size:-**
- If all partitions are occupied with processes that are not ready to run, then one of these processes must be swapped out to make room for a new process.
- Which one to swap out is a scheduling decision.

•Process whose size is less than or equal to the partition size can be loaded into any available partition.



(a) Equal-size partitions

Problem with fixed equal size partitions:-

(1)

•A program may not fit in a partition.

•The programmer must design the program with overlays so that only a portion of the program need be in main memory at any one time.

(2)

•Main memory use is inefficient.

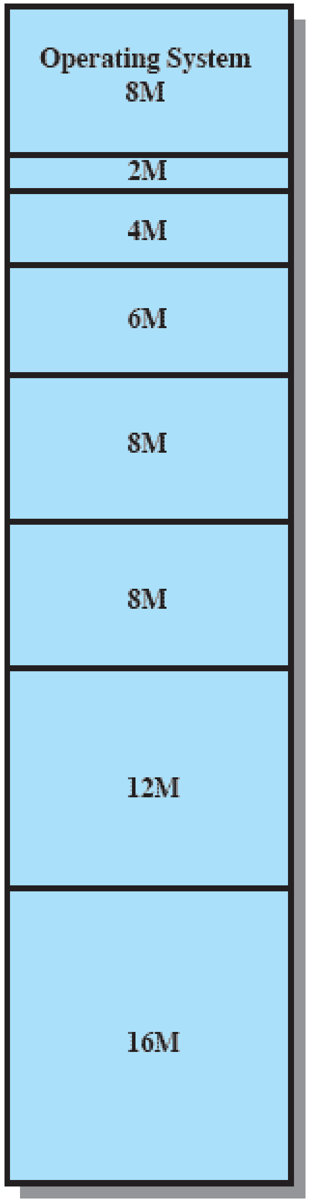
•Any program, no matter how small, occupies an entire partition.

•This results in ***internal fragmentation.***

Internal fragmentation:-

The 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**.

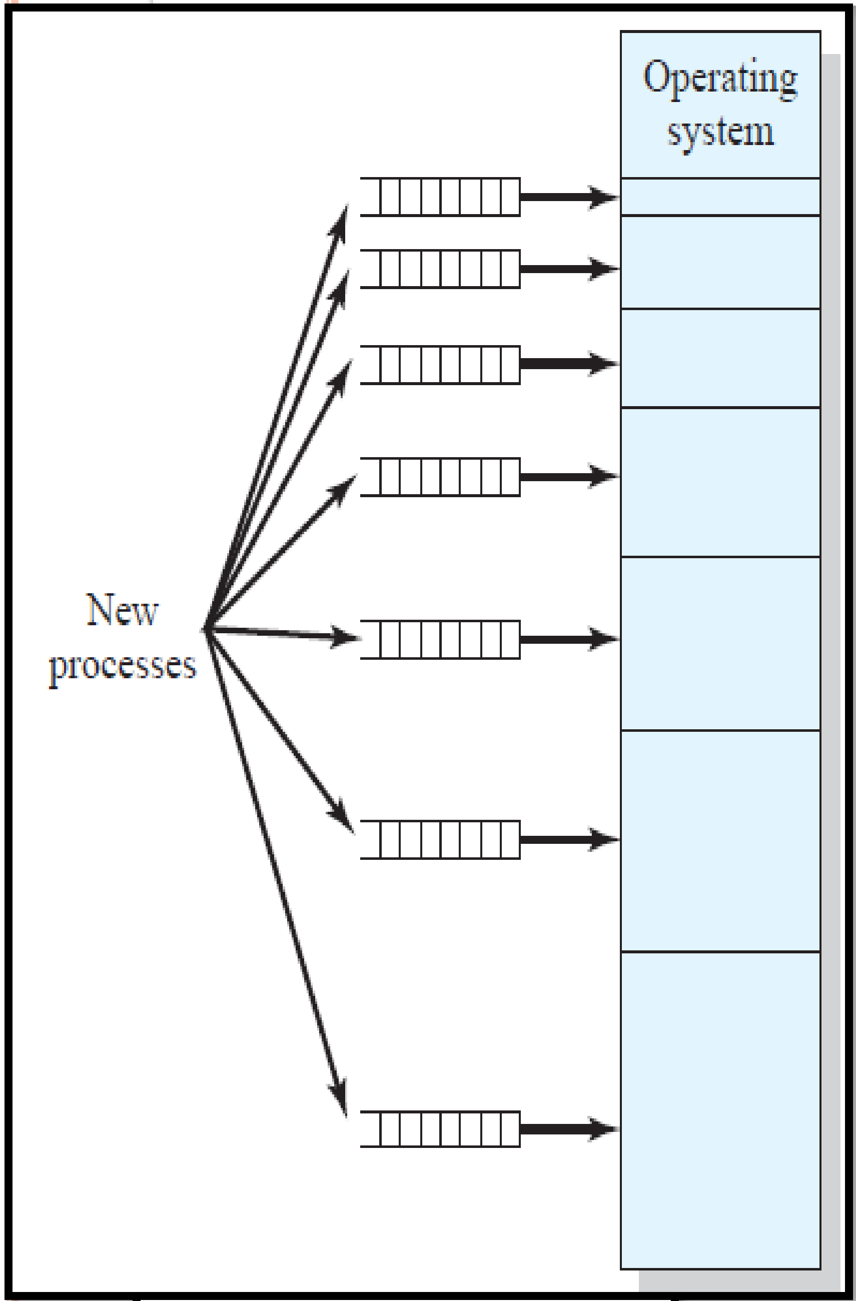
Solution – Unequal Size Partitions



(b) Unequal-size partitions

- Lessens both problems
  - but doesn't solve completely
- In Fig 7.3b,
  - Programs up to 16M can be accommodated without overlays
  - Smaller programs can be placed in smaller partitions, reducing internal fragmentation

PLACEMENT ALGORITHM Unequal size:-



(a) One process queue per partition

•With unequal-size partitions, there are two possible ways to assign processes to partitions.

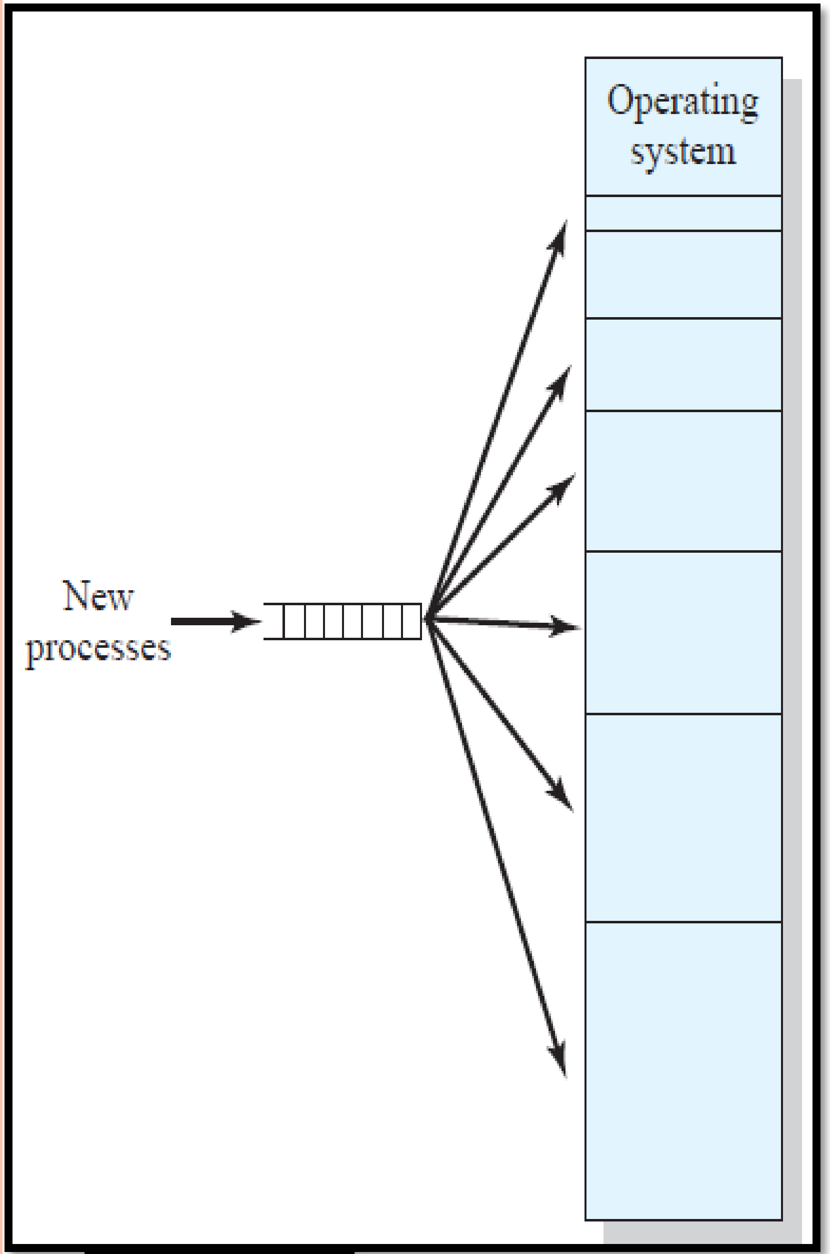
➢The simplest way is to assign each process to the smallest partition within which it will fit.

➢In this case, a scheduling queue is needed for each partition, to hold swapped-out processes destined for that partition (Figure 7.3a).

➢ consider a case in which there are no processes with a size between 12 and 16M at a certain point in time.

➢In that case, the 16M partition will remain unused, even though some smaller process could have been assigned to it.

Placement Algorithm



(b) Single queue

- Thus, a preferable approach would be to employ a single queue for all processes (Figure 7.3b).

- When it is time to load a process into main memory, the smallest available partition that will hold the process is selected.

Remaining Problems with Fixed Partitions

- The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.

- Because partition sizes are preset at system generation time, small jobs will not utilize partition space efficiently.

(2) 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.

- Imagine a system with 64M RAM

1. Initially, main memory is empty, except for the operating system
2. Three processes are loaded in – leaving a ‘hole’ too small for any further process
3. At some point, none of the processes in memory is ready. The operating system swaps out process 2,
4. Which leaves sufficient room to load a new process, process 4 – but that creates another hole
5. 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 for process 2, the operating system swaps process 1 out and swaps process 2 back in leaving yet another hole

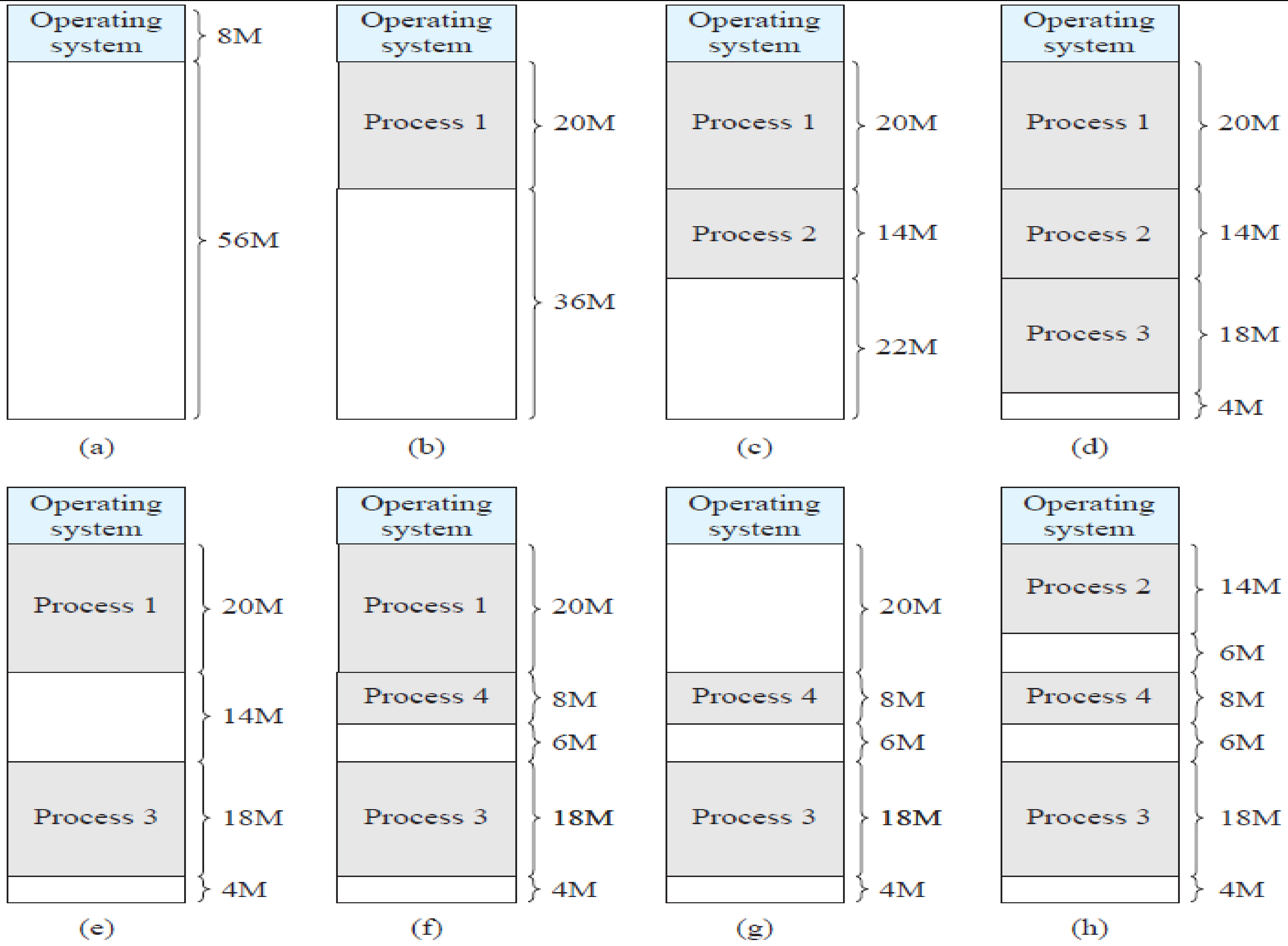


Figure 7.4 The Effect of Dynamic Partitioning

Dynamic Partitioning Example

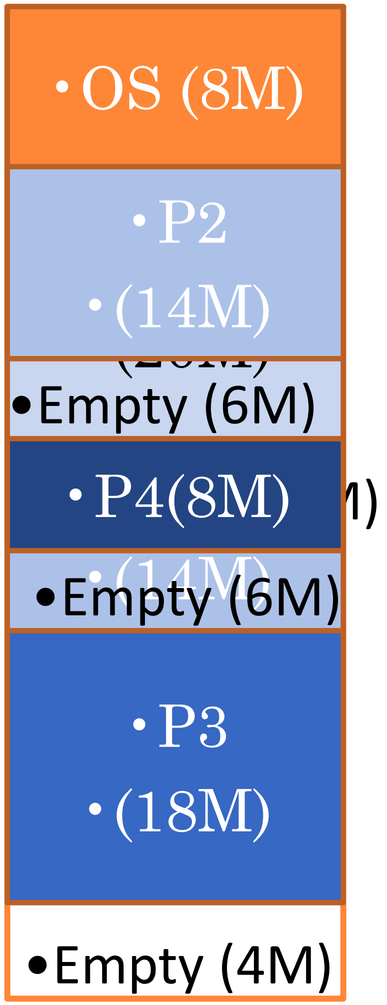


Figure 7.4

- As time goes on, memory becomes more and more fragmented, and memory utilization declines.
- **Problem is:-**
- **External Fragmentation:-**
  - The memory that is external to all partitions becomes increasingly fragmented.
- **Solution** to external fragmentation is
- **Compaction**
- **(**compaction implies the capability of dynamic relocation)**:-**
  - From time to time, the operating system shifts the processes so that they are contiguous and
  - so that all of the free memory is together in one block.

•**Problem :-** A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction

Placement Algorithm for Dynamic Partitioning

- When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.

- Three placement algorithms that might be considered are best-fit, first-fit, and next-fit.
- **Best-fit algorithm**
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often
- **First-fit algorithm**
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block

Placement Algorithm

- **Next-fit**
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

Allocation

•Slide shows Fig 7.5 - an example memory configuration

after a number of placement and swapping-out

operations.

Allocation

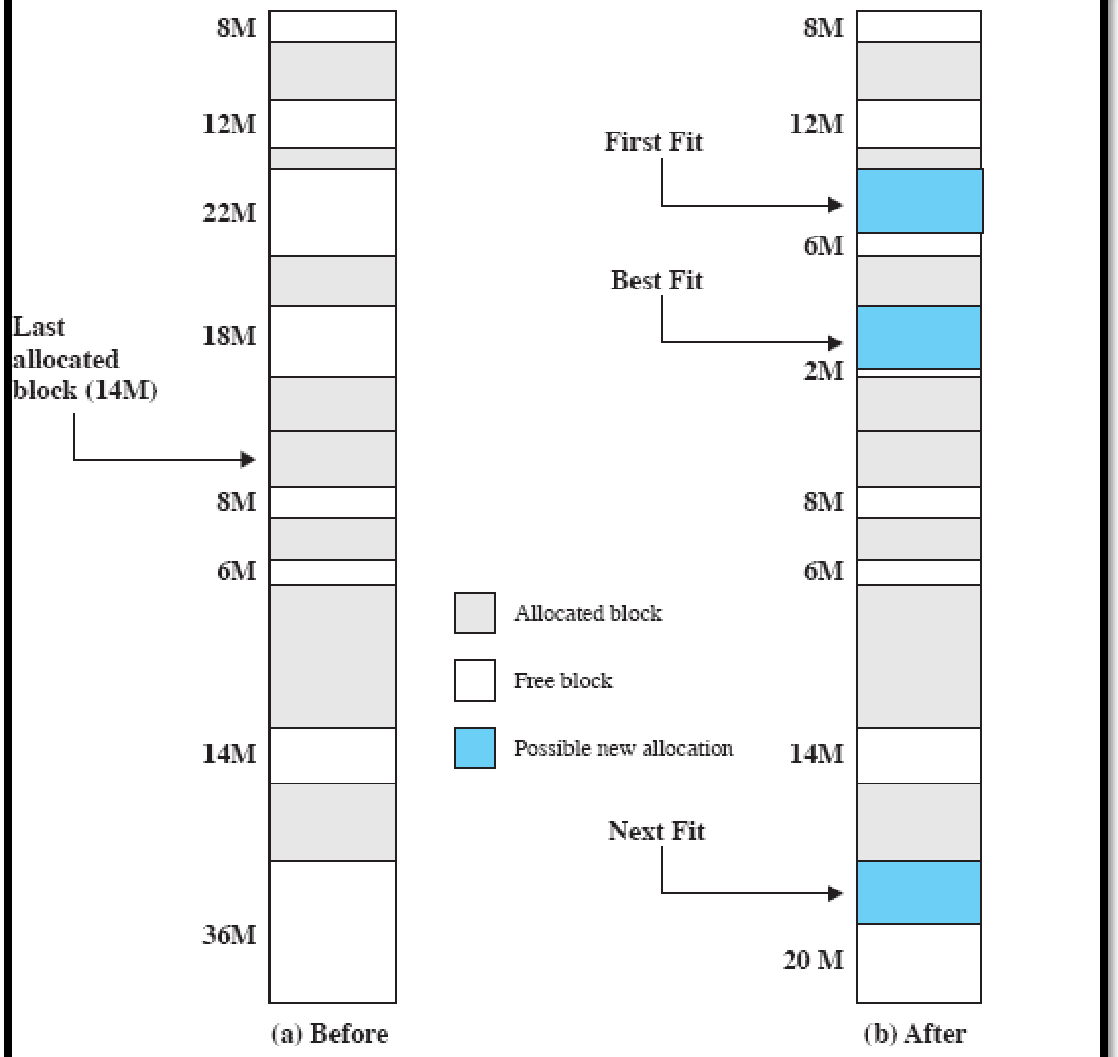
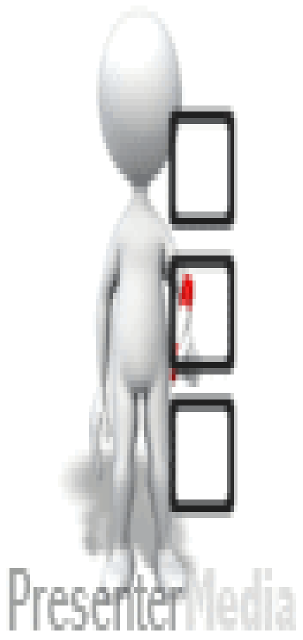


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

- The last block that was used was a 22-Mbyte block from which a 14- Mbyte partition was created.
- Figure 7.5b shows the difference between the best, first, and next-fit placement algorithms in satisfying a 16-Mbyte allocation request.
- **Best-fit** will search the entire list of available blocks and make use of the 18-Mbyte block, leaving a 2-Mbyte fragment.
- **First-fit** results in a 6-Mbyte fragment, and
- **Next-fit** results in a 20-Mbyte fragment.

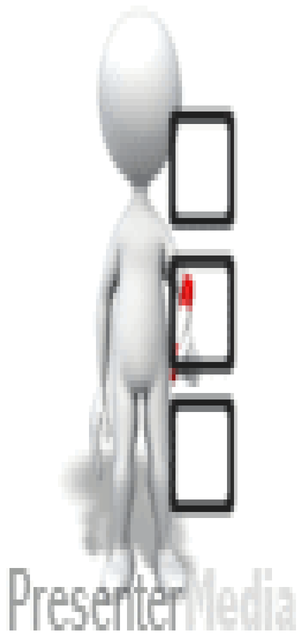
Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Simple Paging and Segmentation
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- Address translation in Paging
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)



Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- **Simple Paging and Segmentation**
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- Address translation in Paging
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)



PAGING

- Partition memory into small equal fixed-size chunks and The chunks of memory are called ***frames***.
- Divide each process into the same size chunks and the chunks of a process are called ***pages***.
- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page

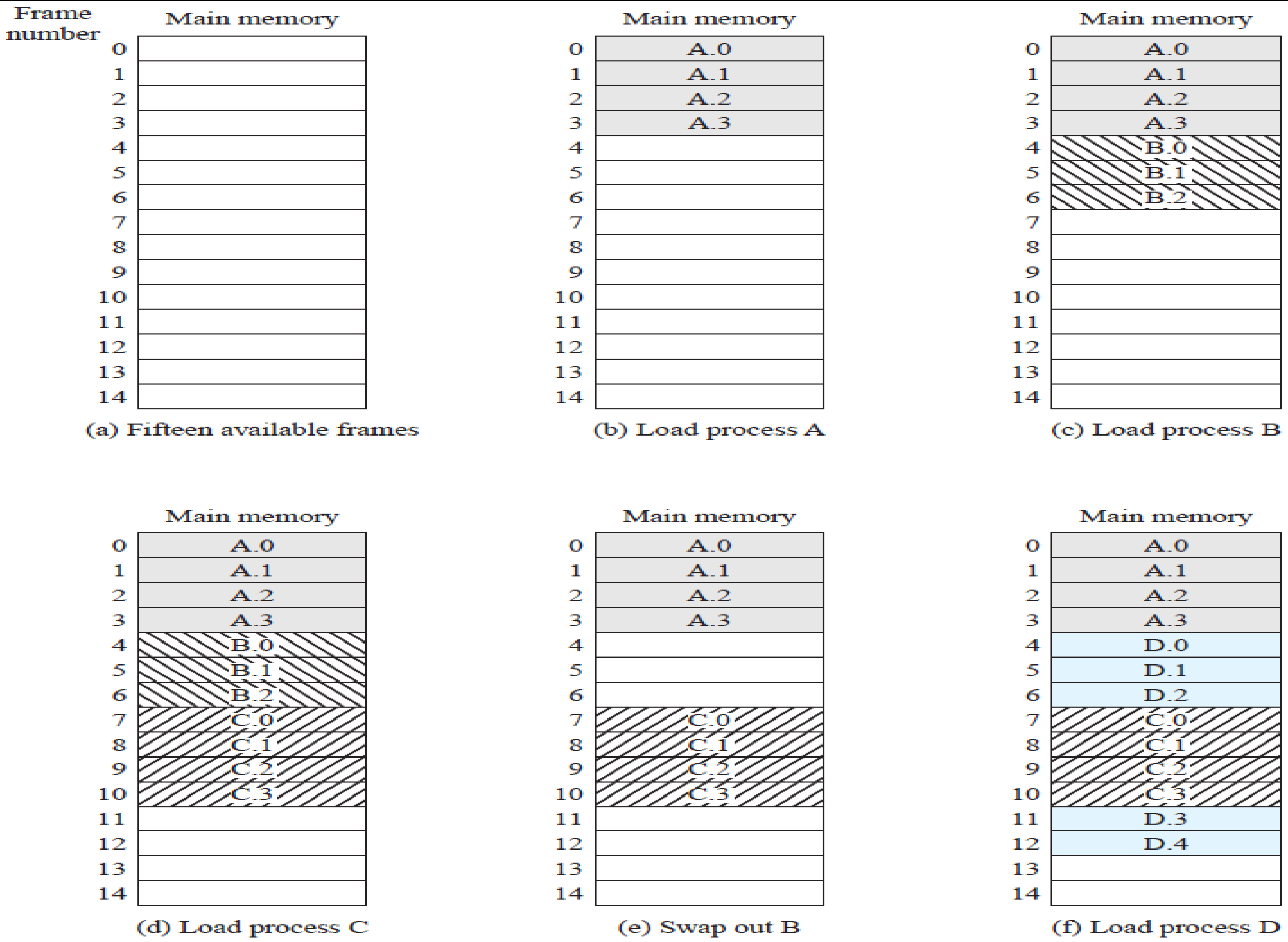


Figure 7.9 **Assignment of Process to Free Frames**

Paging Example

1. System with a number of frames allocated.

2. Process A, stored on disk, consists of four pages.

   When it comes time to load this process, the operating system finds four free frames and loads the four pages of process A into the four frames.

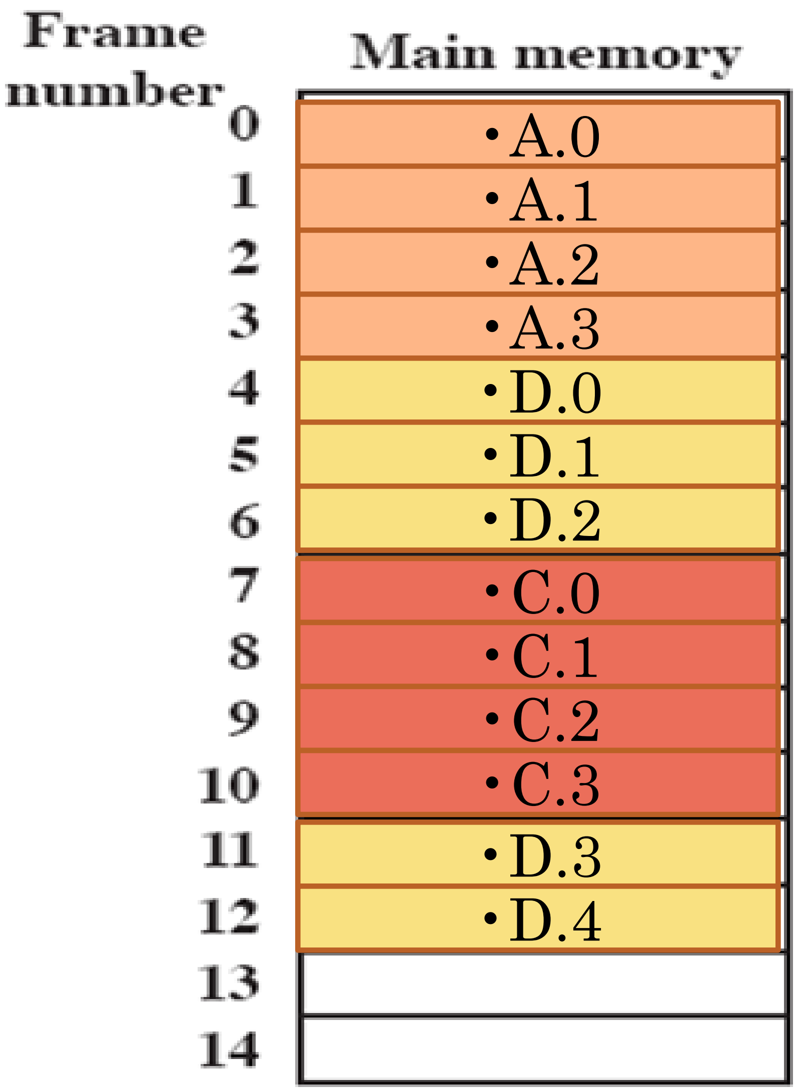
3. Process B, consisting of three pages, and process C, consisting of four pages, are subsequently loaded.

4. Then process B is suspended and is swapped out of main memory.

5. Later, all of the processes in main memory are blocked, and the operating system needs to bring in a new process, process D, which consists of five pages.

- The Operating System loads the pages into the available frames and updates the ***page table***

Processes and Frames



PAGE TABLE

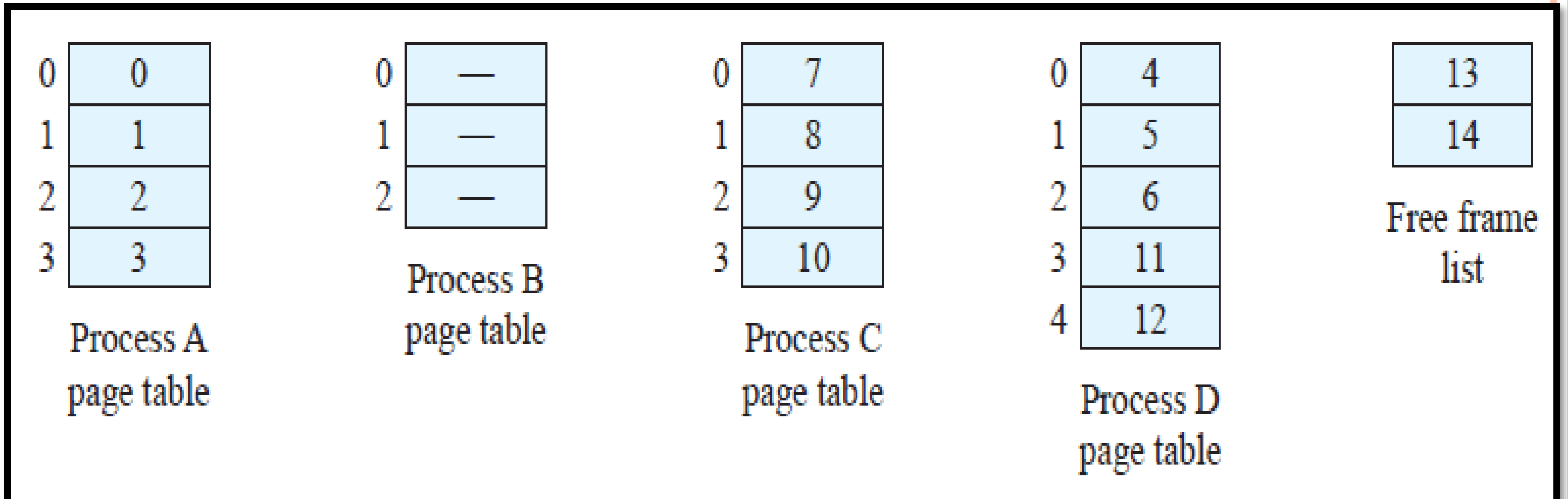


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

- Figure 7.10 shows the various page tables at this time.
- A page table contains one entry for each page of the process, so that the table is easily indexed by the page number (starting at page 0).
- Each page table entry contains the number of the frame in main memory, if any, that holds the corresponding page.
- Operating system maintains a single free-frame list of all frames in main memory that are currently unoccupied and available for pages.

Logical to Physical Address Translation Example

- In our example, we have the logical address 0000010111011110, which is page number 1, offset 478.
- Suppose that this page is residing in main memory frame 6 binary 000110.
- Then the physical address is frame number 6, offset 478 =0001100111011110 (Figure 7.12a).

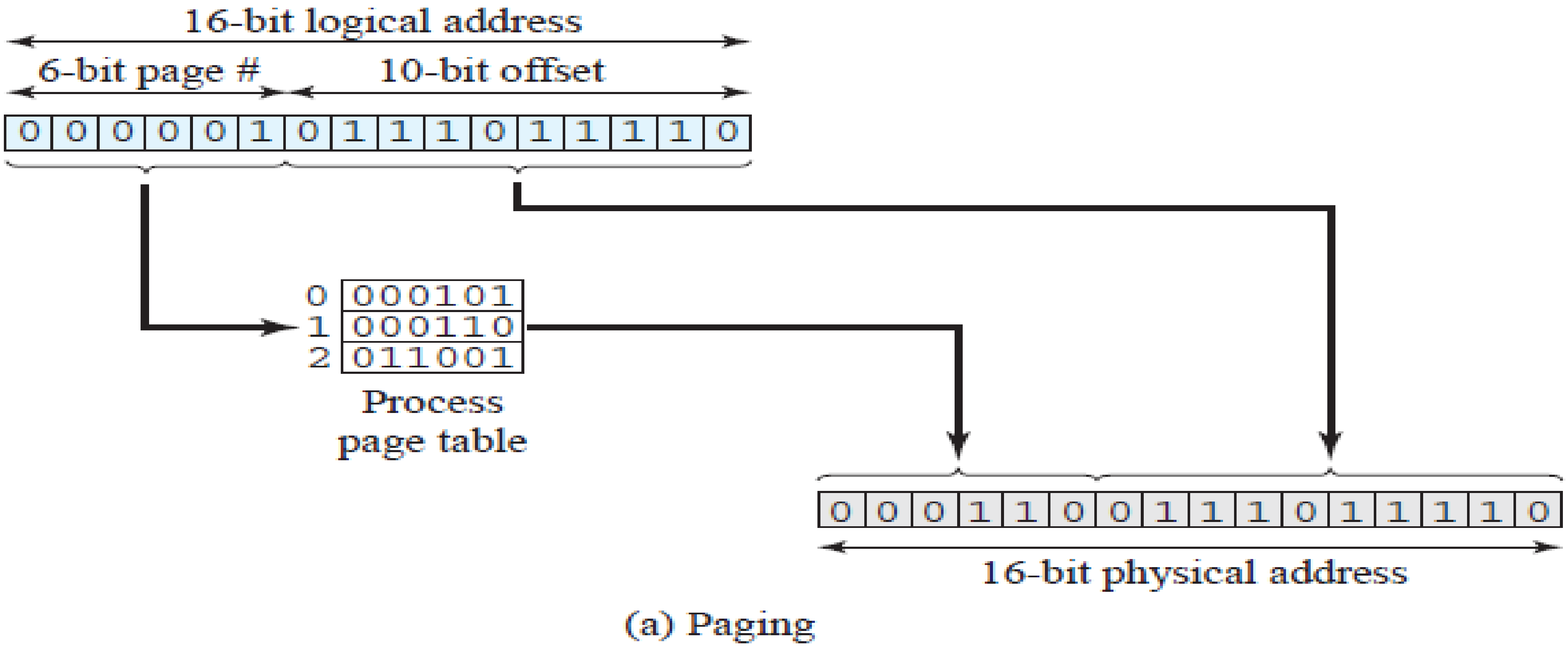


Figure 7.12 Examples of Logical-to Physical Address Translation

SEGMENTATION

- A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments**.
- It is not required that all segments of all programs be of the same length, although there is a maximum segment length.
- As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

•Segmentation is usually visible and is provided as a convenience for organizing programs and data.

• The programmer or compiler will assign programs and data to different segments.

• For purposes of modular programming, the program or data may be further broken down into multiple segments.

Segmentation addressing

•The following steps are needed for address translation:-

(1) Extract the segment number as the leftmost *n bits of the logical address.*

(2) Use the segment number as an index into the process segment table to find the starting physical address of the segment.

(3) Compare the offset, expressed in the rightmost *m bits, to the length of the segment.* If the offset is greater than or equal to the length, the address is invalid.

(4) The desired physical address is the sum of the starting physical address of the segment plus the offset.

Logical to Physical Address Translation Example

- In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752.
- Suppose that this segment is residing in main memory starting at physical address 0010000000100000.
- Then the physical address is
- 0010000000100000 + 001011110000 = 0010001100010000 (Figure 7.12b).

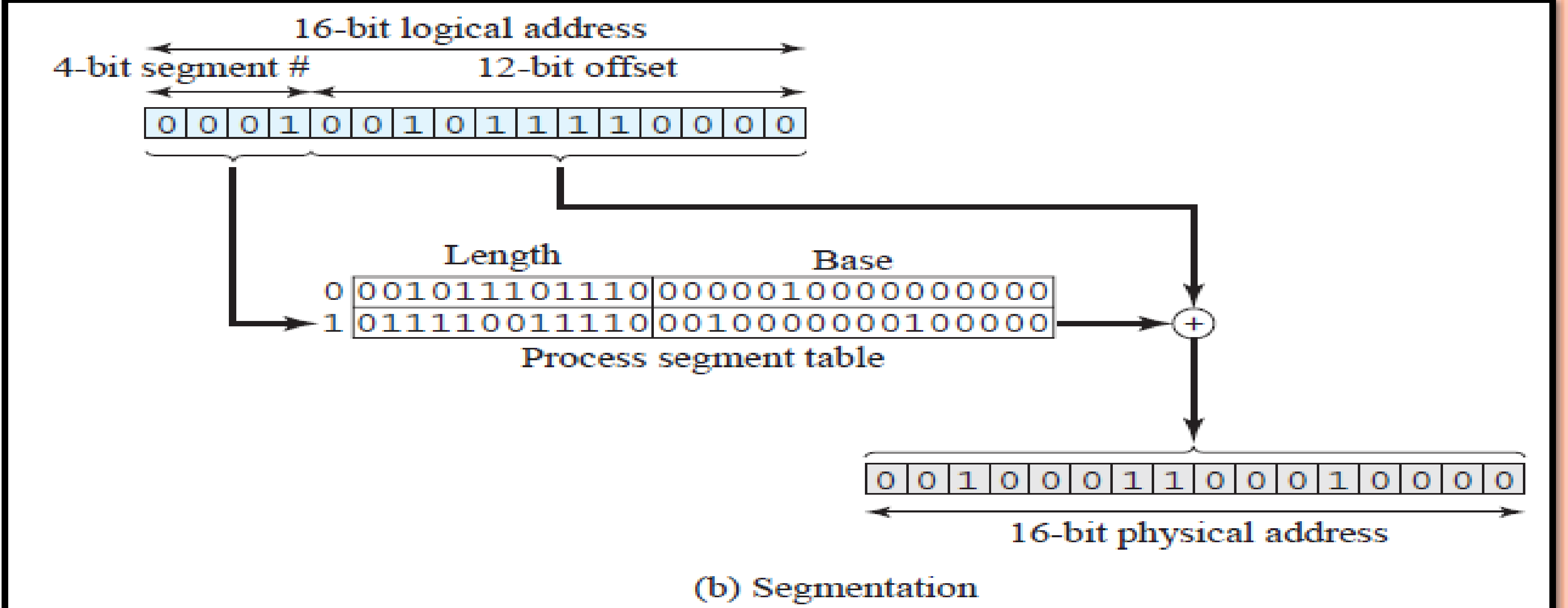


Figure 7.12 Examples of Logical-to Physical Address Translation

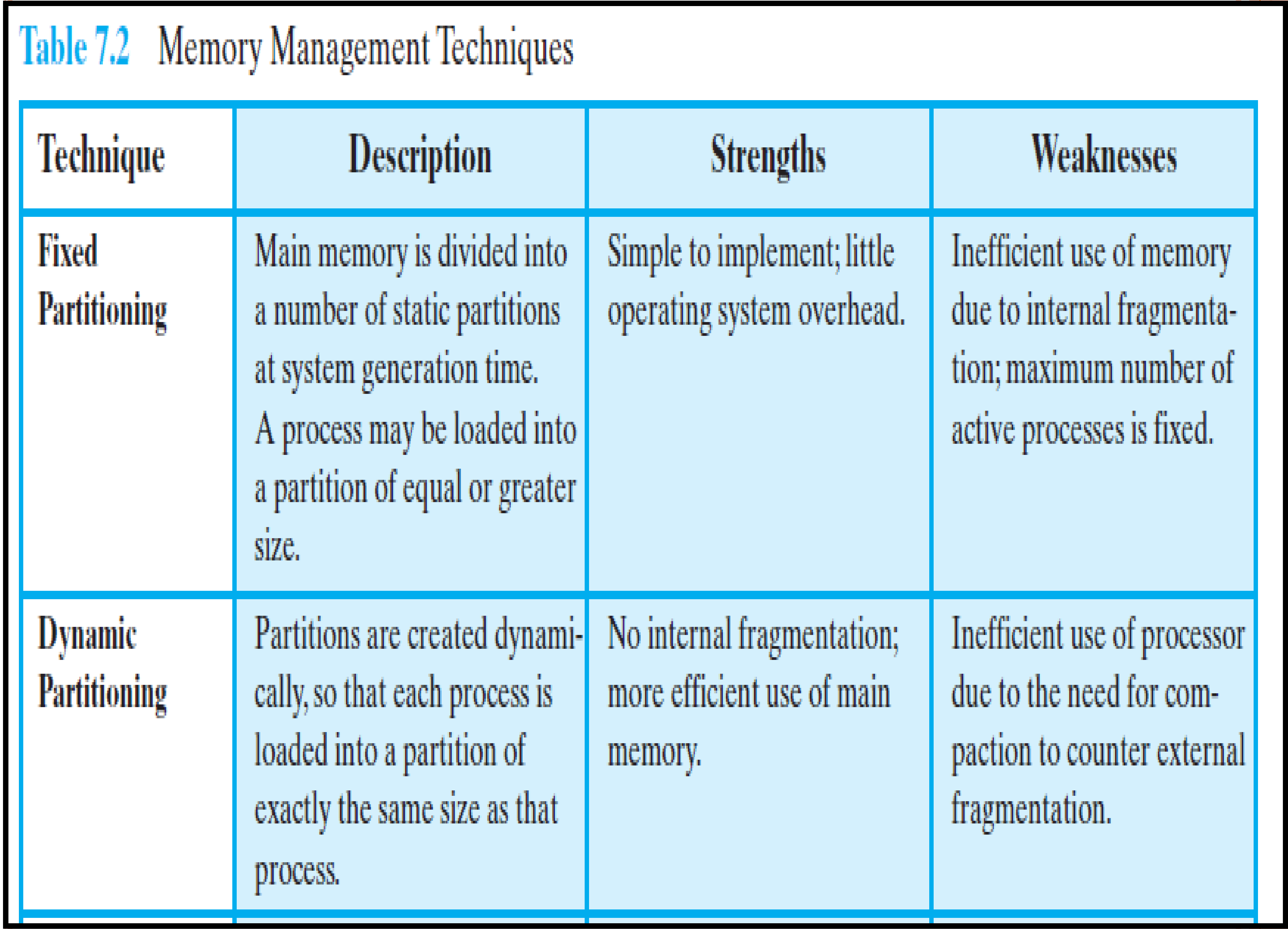


Table 7.2 Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.

Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Buddy System
- Simple Paging and Segmentation
- **Need of Virtual Memory**
- Virtual memory Paging and Segmentation
- Address translation in Paging
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)

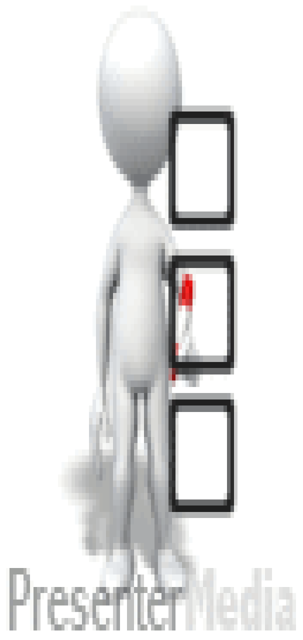


Table 8.1 Virtual Memory Terminology

Virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
Virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
Virtual address space	The virtual storage assigned to a process.
Address space	The range of memory addresses available to a process.
Real address	The address of a storage location in main memory.

Execution of a Process

- The portion of a process that is actually in main memory at any time is defined to be the **resident set of the process**.
- Using the segment or page table, the processor always is able to deter mine whether page or segment is in main memory or not.
- If the processor encounters a logical address that **is not** in main memory,
  - it generates an interrupt indicating a memory access fault.
- The operating system puts the interrupted process in a blocking state and takes control.

Implications of this New Strategy

- There are two implications- both lead to improved system utilization:

1. **More processes may be maintained in main memory.**
   - Because we are only going to load some of the pieces of any particular process, there is room for more processes.

2. **A process may be larger than all of main memory**.
   - If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be loaded separately in some sort of overlay strategy.
   - With virtual memory based on paging or segmentation, that job is left to the operating system and the hardware.
   - The operating system automatically loads pieces of a process into main memory as required.

Real and Virtual Memory

- Real memory
  - Main memory, the actual RAM
- Virtual memory
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

• Table 8.2 summarizes characteristics of paging and segmentation, with and without the use of virtual memory.

Table 8.2 Characteristics of Paging and Segmentation

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames	Main memory partitioned into small fixed-size chunks called frames	Main memory not partitioned	Main memory not partitioned
Program broken into pages by the compiler or memory management system	Program broken into pages by the compiler or memory management system	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)
Internal fragmentation within frames	Internal fragmentation within frames	No internal fragmentation	No internal fragmentation
No external fragmentation	No external fragmentation	External fragmentation	External fragmentation
Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a segment table for each process showing the load address and length of each segment	Operating system must maintain a segment table for each process showing the load address and length of each segment
Operating system must maintain a free frame list	Operating system must maintain a free frame list	Operating system must maintain a list of free holes in main memory	Operating system must maintain a list of free holes in main memory
Processor uses page number, offset to calculate absolute address	Processor uses page number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory frames for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

Thrashing

- In the steady state, practically all of main memory will be occupied with process pieces, so that the processor and operating system have direct access to as many processes as possible.
- Thus, when the operating system brings one piece in, it must throw another out.
- If it throws out a piece just before it is used, then it will just have to go get that piece again almost immediately.
    - Too much of this leads to a condition known as **thrashing**
    - The system spends most of its time swapping pieces rather than executing instructions.

- *Thrashing* is a condition in which excessive paging operations are taking place.
- A system that is *thrashing* can be perceived as either a very slow system or one that has come to a halt.

Principle of Locality

- Program and data references within a process tend to cluster
- Only a few pieces of a process will be needed over a short period of time
- Therefore it is possible to make intelligent guesses about which pieces will be needed in the future.
- This suggests that virtual memory may work efficiently.

A Processes Performance in VM Environment

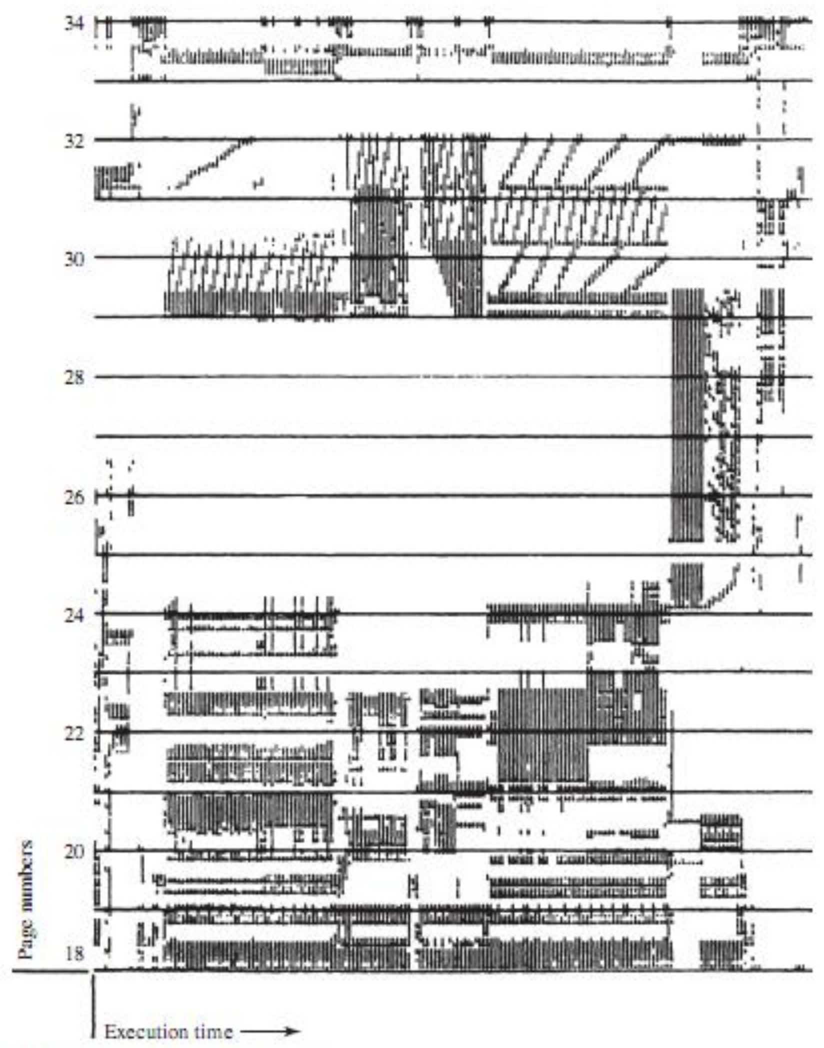


Figure 8.1 Paging Behavior

- One way to confirm the principle of locality is to look at the performance of processes in a virtual memory environment.
- Figure 8.1 is a rather famous diagram that dramatically illustrates the principle of locality.
- Note that during the lifetime of the process, references are confined to a subset of pages.

Support Needed for Virtual Memory

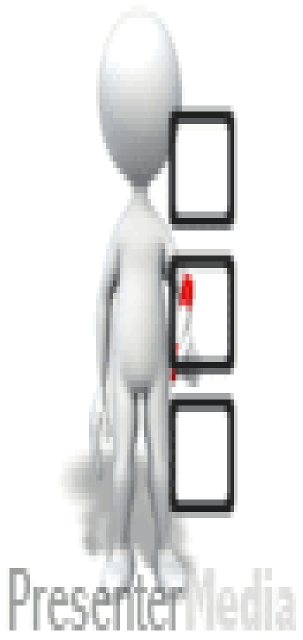
- For virtual memory to be practical and effective, two ingredients are needed.

1) There must be hardware support for the paging and/or segmentation scheme to be employed.

2) The operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory.

Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Buddy System
- Simple Paging and Segmentation
- Need of Virtual Memory
- **Virtual memory Paging and Segmentation**
- Address translation in Paging
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)



Paging

- In the discussion of simple paging, we indicated that each process has its own page table.
  - When all of its pages are loaded into main memory, the page table for a process is created and loaded into main memory.
- Each page table entry contains the frame number of the corresponding page in main memory.
- A page table is also needed for a virtual memory scheme based on paging.
- Again, it is typical to associate a unique page table with each process.
- **(Figure 8.2a – next slide).**
  - The page table entries become more complex because only some of the pages of a process may be in main memory.

Paging Table

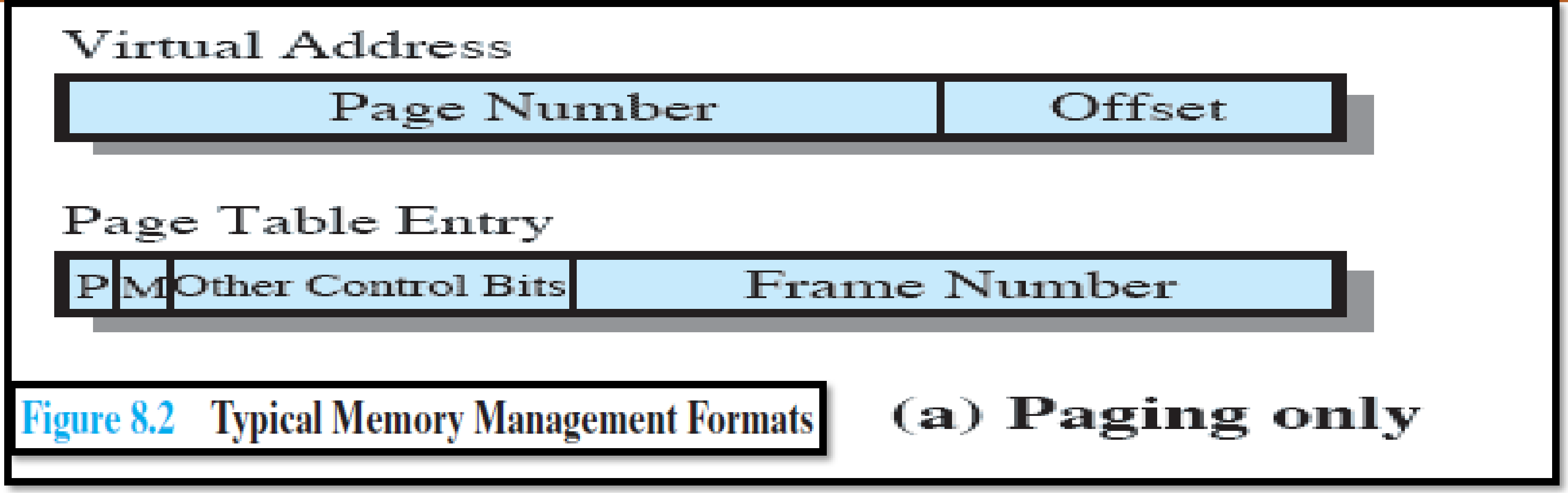


Figure 8.2 Typical Memory Management Formats

•It is typical to associate a unique page table with each process.

• A bit is needed in each page table entry to indicate whether the corresponding page is

• present (P) in main memory or not , If the bit indicates that the page is in memory, then the entry also includes the frame number of that page.

• The page table entry includes a modify (M) bit, indicating whether the contents of the corresponding page have been altered since the page was last loaded

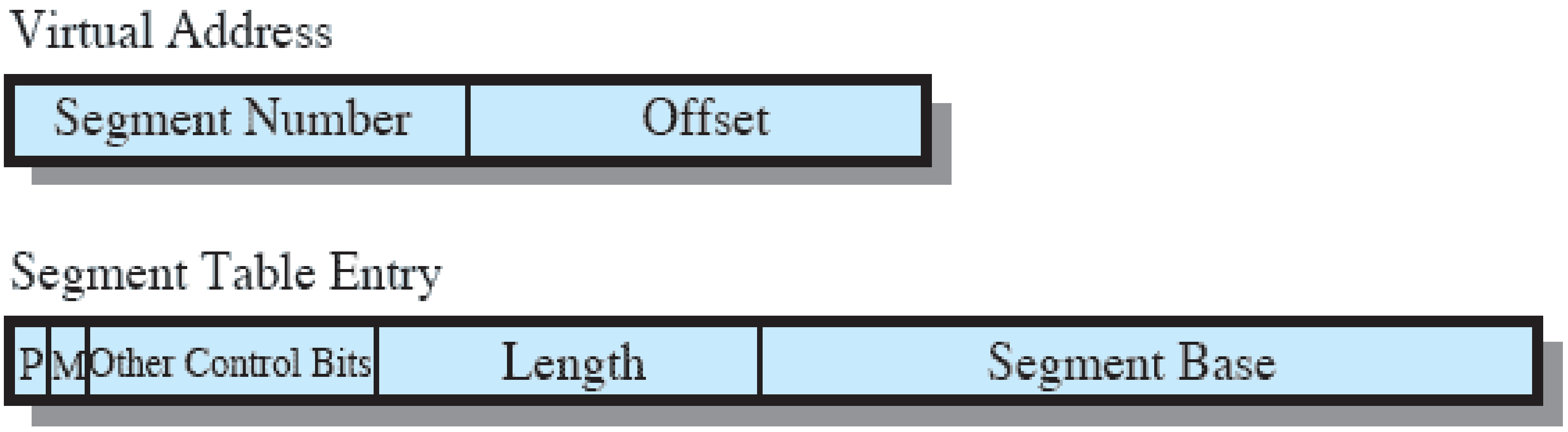
Segmentation

- Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments may be of unequal, indeed dynamic, size.
- Memory references consist of a form of address (segment number, offset).
- This organization has a number of advantages to the programmer over a non segmented address space:

1. **It simplifies the handling of growing data structures**.
   - With segmented virtual memory, the data structure can be assigned its own segment, and the operating system will expand or shrink the segment as needed.
2. **It allows programs to be altered and recompiled independently,** Without requiring the entire set of programs to be reloaded. Again, this is accomplished using multiple segments.
3. **It lends itself to sharing among processes**.
   - A programmer can place a utility program or a useful table of data in a segment that can be referenced by other processes.
4. **It lends itself to protection.**
   - The programmer or system administrator can assign access priviledges in a convenient fashion.

Segment Table Entries

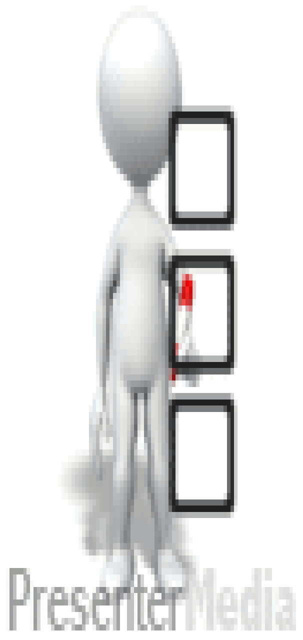
- Each segment table entry contains the starting address of the corresponding segment in main memory, as well as the length of the segment.
- A P bit is needed in each segment table entry to indicate whether the corresponding segment is present in main memory or not.
- If the bit indicates that the segment is in memory, then the entry also includes the starting address and length of that segment.
- A **modify bit** indicates whether the contents of the corresponding segment have been altered.



(b) Segmentation only

Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Buddy System
- Simple Paging and Segmentation
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- **Address translation in Paging**
- Address translation in Segmentation
- Page Replacement Algorithms (FIFO, LRU, Optimal)



ADDRESS TRANSLATION IN A PAGING SYSTEM

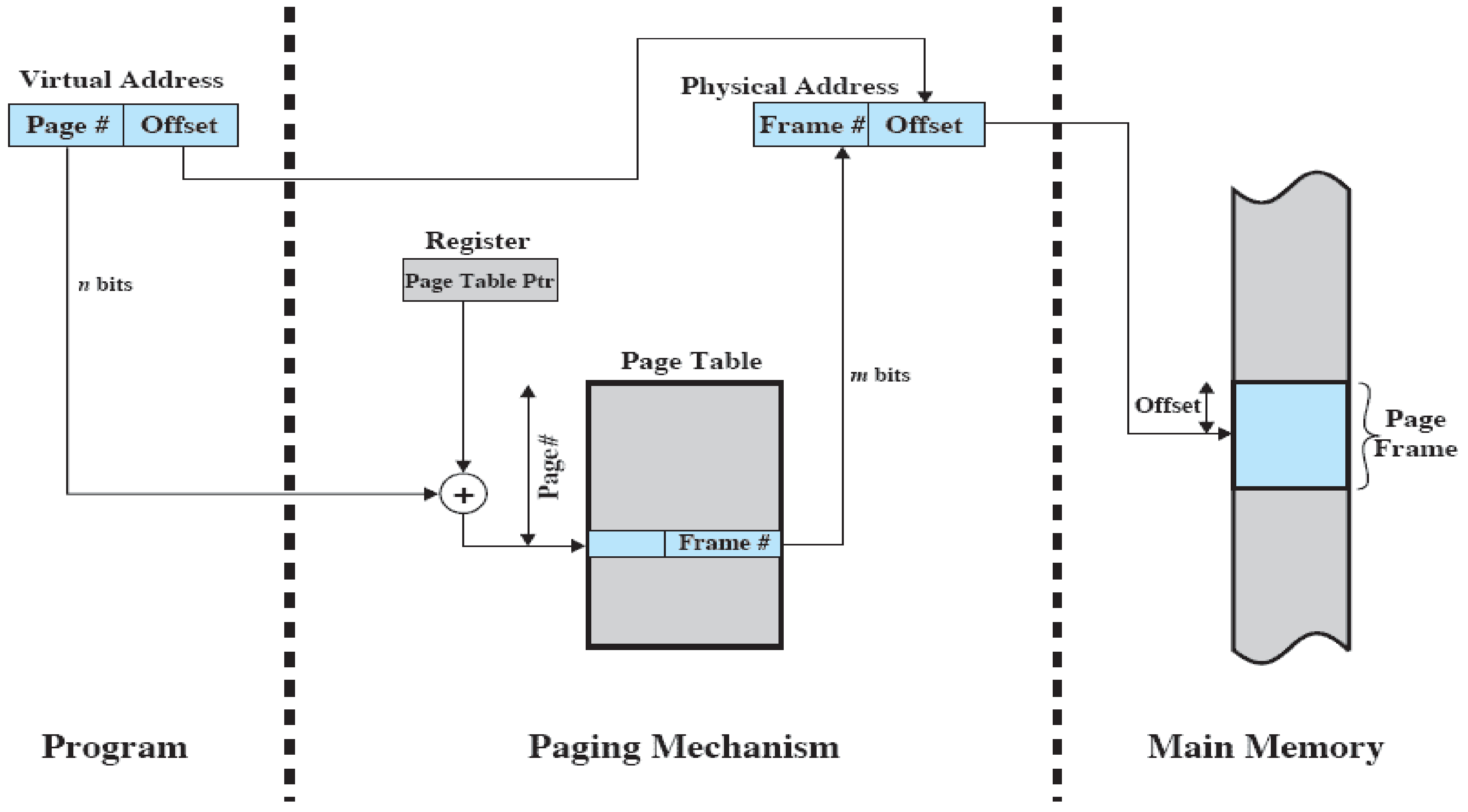


Figure 8.3 Address Translation in a Paging System

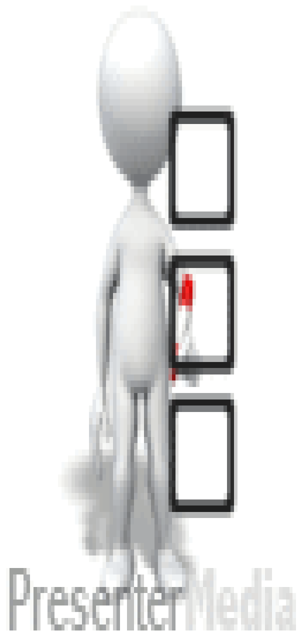
Address Translation In A Paging System

- The basic mechanism for reading a word from memory involves using a page table to translate a virtual/logical address,
  - consisting of page number and offset,
- Into a physical address,
  - consisting of frame number and offset,
- Because the page table is of variable length, depending on the size of the process, we cannot expect to hold it in registers.
  - Instead, it must be in main memory to be accessed.
- Figure 8.3 suggests a hardware implementation.
- When a particular process is running, a register holds the starting address of the page table for that process.
  - The page number of a virtual address is used to index that table and look up the corresponding frame number.
  - This is combined with the offset portion of the virtual address to produce the desired real address.

Typically, the page number field is longer than the frame number field (n > m).

ROADMAP

- Basic requirements of Memory Management
- Memory Partitioning
- Buddy System
- Simple Paging and Segmentation
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- Address translation in Paging
- **Address translation in Segmentation**
- Page Replacement Algorithms (FIFO, LRU, Optimal)



Address Translation in Segmentation

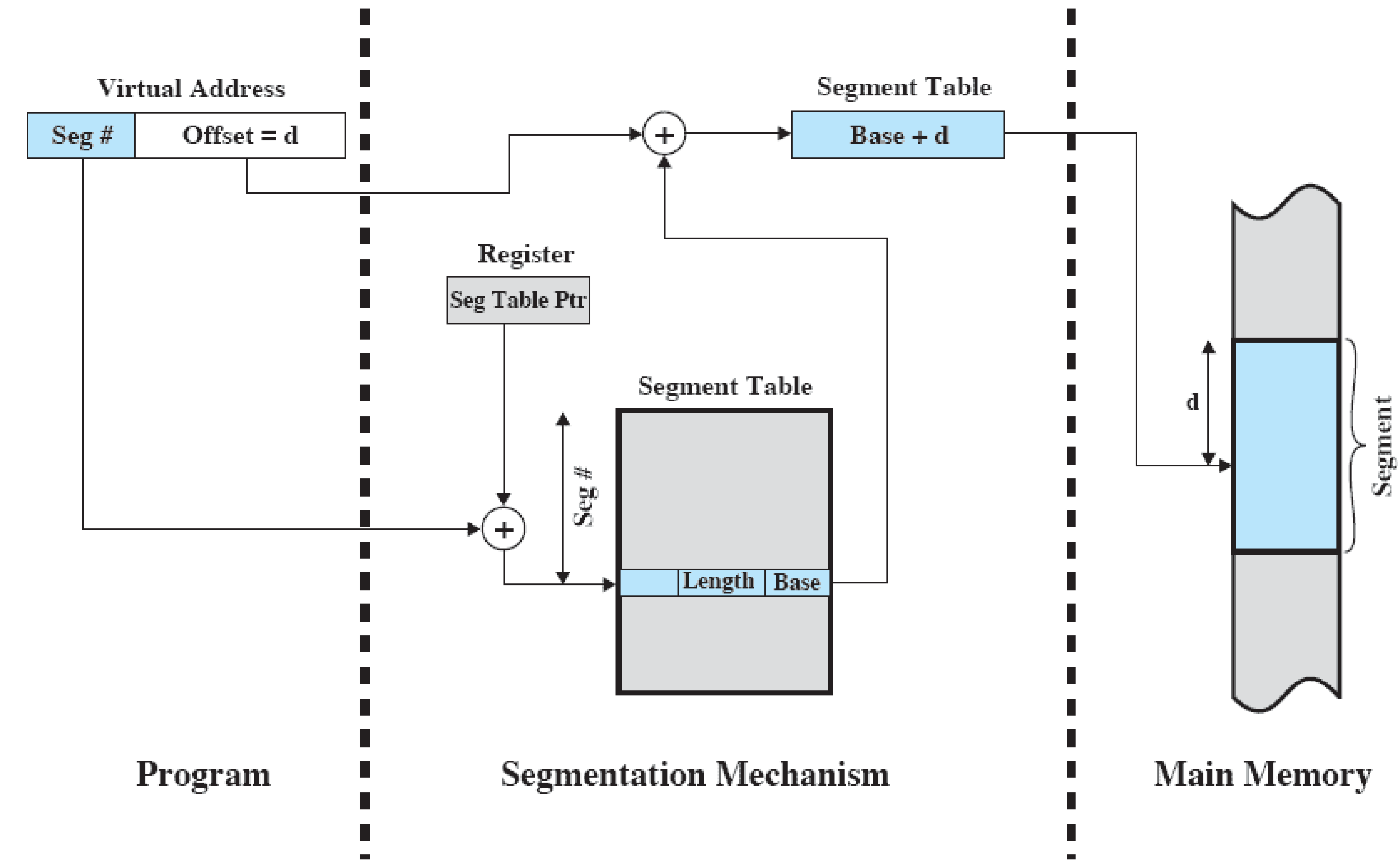


Figure 8.12 Address Translation in a Segmentation System

Address Translation in Segmentation

- Reading a word from memory involves the translation of a virtual, or logical, address, into a physical address, using a segment table.

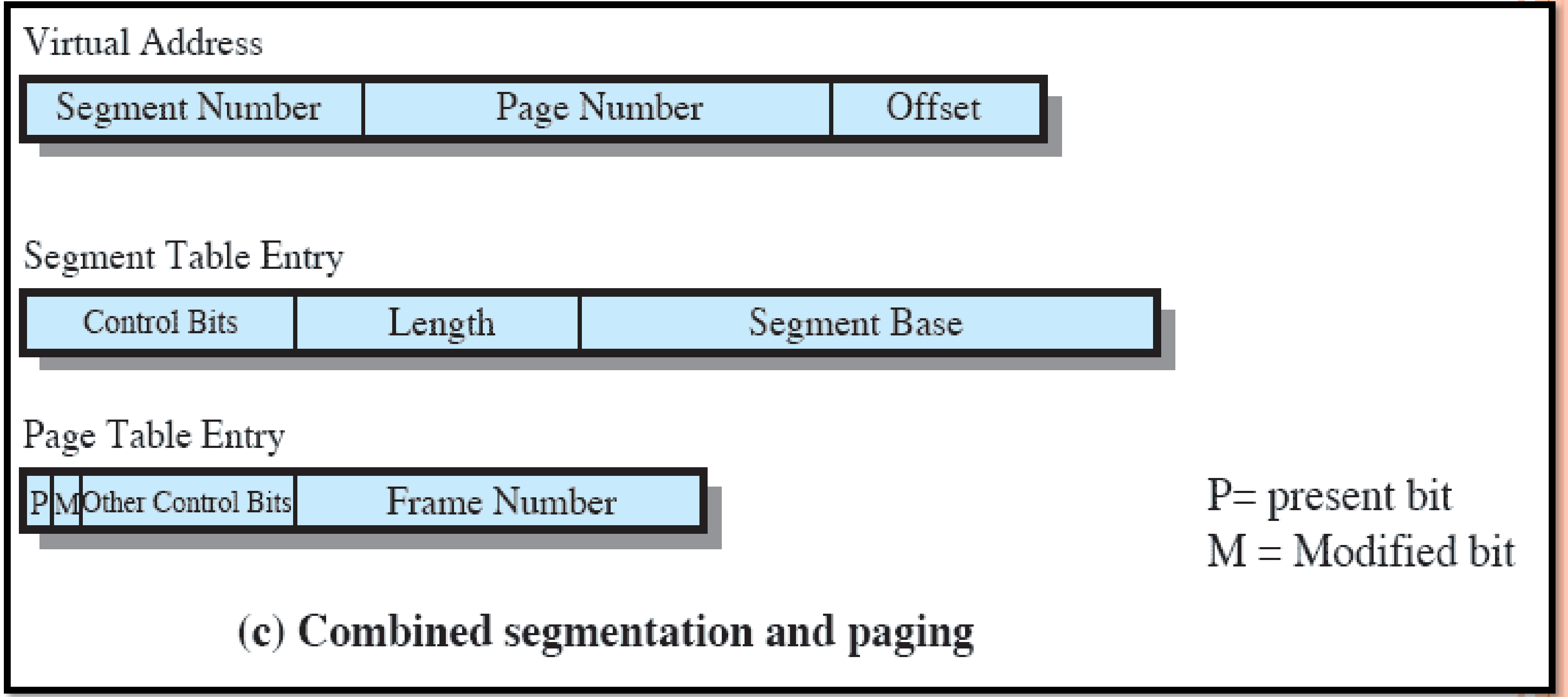
- When a particular process is running, a register holds the starting address of the segment table for that process.

  - The segment number of a virtual address is used to index that table and look up the corresponding main memory address for the start of the segment.
  - This is added to the offset portion of the virtual address to produce the desired real address.

Combined Paging and Segmentation

- In a combined paging/segmentation system, a user's address space is broken up into a number of segments,
- Each segment is, in turn, broken up into a number of fixed-size pages,
  - which are equal in length to a main memory frame.
- If a segment has length less than that of a page, the segment occupies just one page.
- From the programmer's point of view, a logical address still consists of a segment number and a segment offset.
- From the system's point of view, the segment offset is viewed as a page number and page offset for a page within the specified segment.

Combined Paging and Segmentation



(c) Combined segmentation and paging

Combined Paging and Segmentation

- This figure (8.2c) suggests the segment table entry and page table entry formats.
- As before, the segment table entry contains the length of the segment.
- It also contains a base field, which now refers to a page table.
- Other control bits may be used, for purposes of sharing and protection.

- The page table entry is essentially the same as is used in a pure paging system.
  - Each page number is mapped into a corresponding frame number if the page is present in main memory.
  - The modified bit indicates whether this page is modified or not.
  - There may be other control bits dealing with protection or other aspects of memory management.

Address Translation

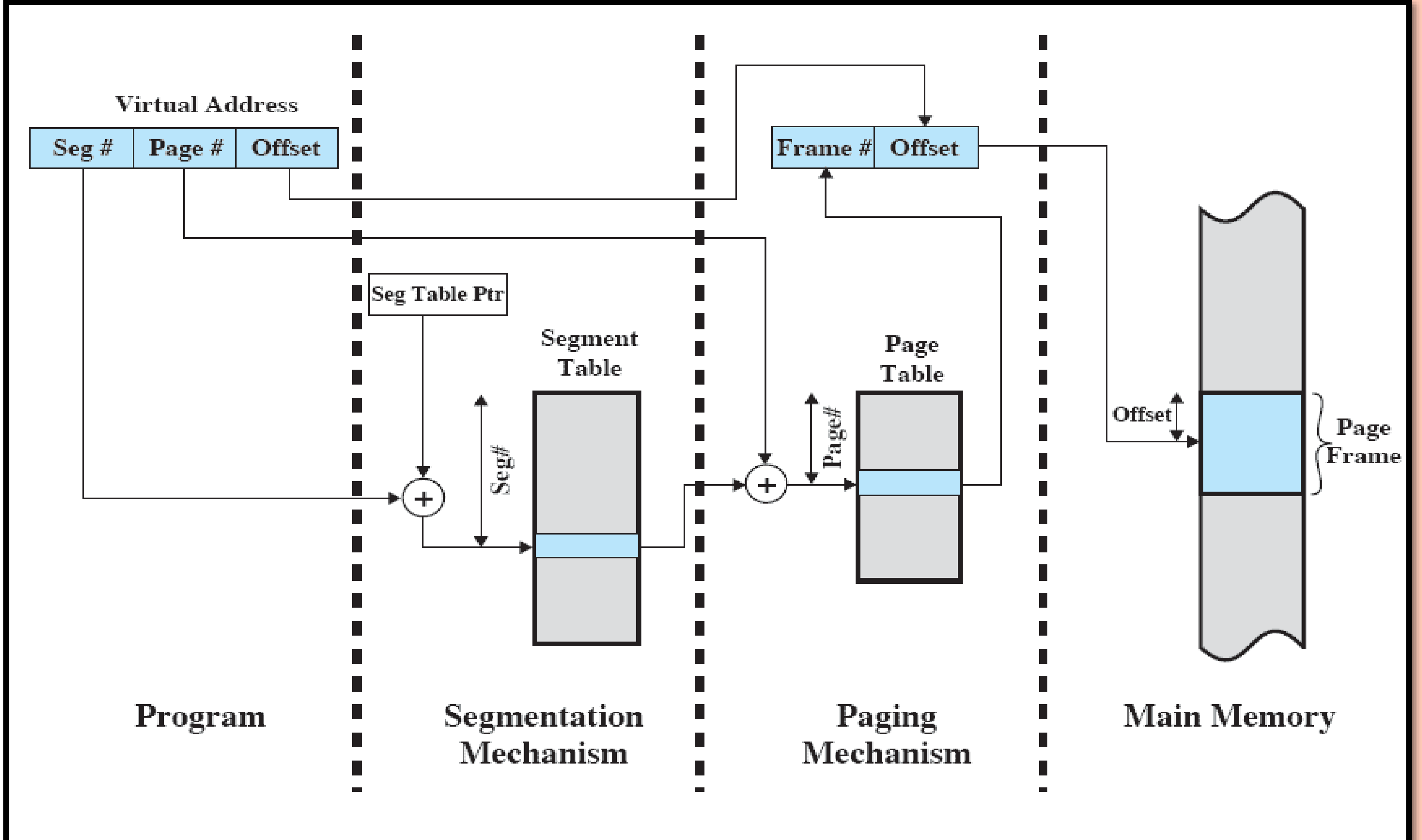


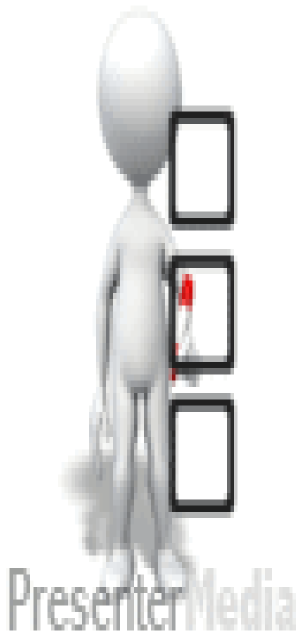
Figure 8.13 Address Translation in a Segmentation/Paging System

Address Translation

- This figure suggests a structure to support combined paging/segmentation (note similarity to Figure 8.5).
- Associated with each process is a segment table and a number of page tables, one per process segment.
- When a particular process is running, a register holds the starting address of the segment table for that process.
- Presented with a virtual address, the processor uses the segment number portion to index into the process segment table to find the page table for that segment.
  - Then the page number portion of the virtual address is used to index the page table and look up the corresponding frame number.
  - This is combined with the offset portion of the virtual address to produce the desired real address.

Roadmap

- Basic requirements of Memory Management
- Memory Partitioning
- Buddy System
- Simple Paging and Segmentation
- Need of Virtual Memory
- Virtual memory Paging and Segmentation
- Address translation in Paging
- Address translation in Segmentation
- **Page Replacement Algorithms (FIFO, LRU, Optimal)**



Replacement Policy

- When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault,
  - the replacement policy determines which page currently in memory is to be replaced.
  - Among the set of pages considered, which particular page should be selected for replacement

- Which page is replaced?
- Page removed should be the page least likely to be referenced in the near future
  - How is that determined?
  - Principal of locality again
- Most policies predict the future behavior on the basis of past behavior

Replacement Policy: Frame Locking

- One restriction on replacement policy needs to be mentioned before looking at various algorithms:
  - Some of the frames in main memory may be locked.
  - When a frame is locked, the page currently stored in that frame may not be replaced.
  - Much of the kernel of the operating system is held on locked frames, as well as key control structures.
  - In addition, I/O buffers and other time-critical areas may be locked into main memory frames.
- Locking is achieved by associating a lock bit with each frame.
- This bit may be kept in a frame table as well as being included in the current page table.

Basic Replacement Algorithms

- There are certain basic algorithms that are used for the selection of a page to replace, they include
  - Optimal
  - Least recently used (LRU)
  - First-in-first-out (FIFO)
  - Clock

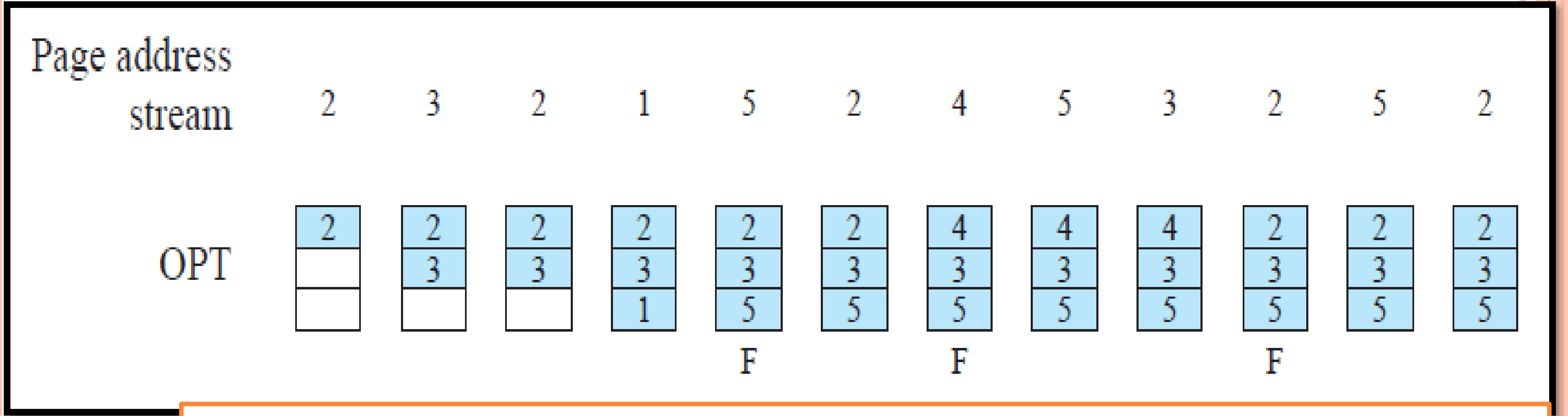
Examples

- An example of the implementation of these policies will use a page address stream formed by executing the program is
  - 2 3 2 1 5 2 4 5 3 2 5 2
- Which means that the first page referenced is 2,
  - the second page referenced is 3,
  - And so on.

(1) Optimal policy

- The optimal policy selects for replacement that page for which the time to the next reference is the longest.
  - This policy results in the fewest number of page faults.
  - **BUT** Clearly, this policy is impossible to implement, because it would require the operating system to have perfect knowledge of future events.
  - However, it does serve as a standard against which to judge real world algorithms.

F = page fault occurring after the frame allocation is initially filled



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

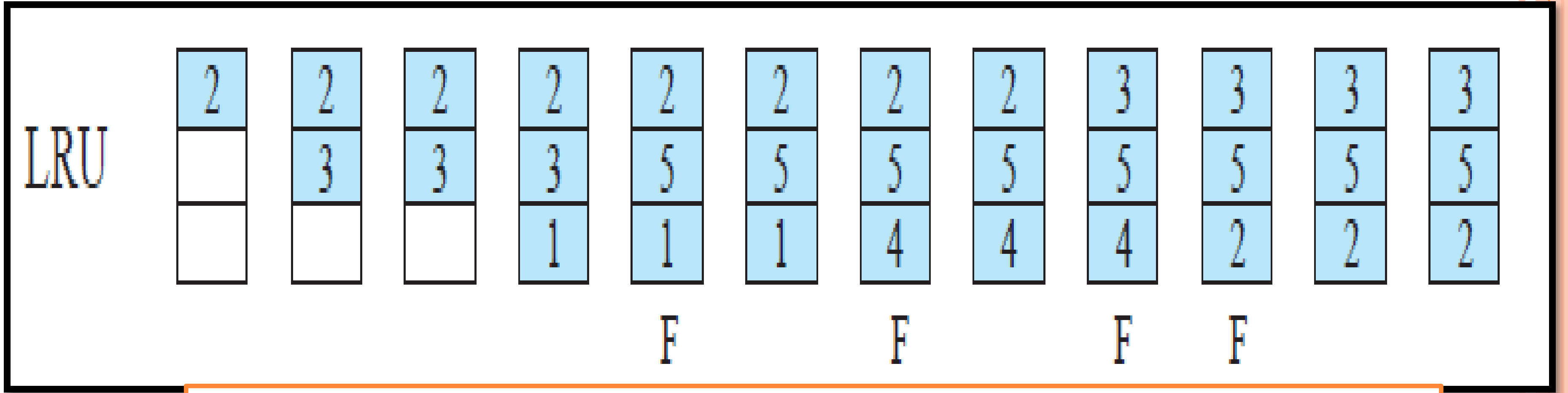
- The optimal policy produces three page faults

Figure 8.15 Behavior of Four Page Replacement Algorithms

(2) Least Recently Used (LRU)

- The least recently used (LRU) policy replaces the page in memory that has not been referenced for the longest time.
- By the principle of locality, this should be the page least likely to be referenced in the near future.
  - And, in fact, the LRU policy does nearly as well as the optimal policy.
- The problem with this approach is the difficulty in implementation.
- One approach would be to tag each page with the time of its last reference;
  - this would have to be done at each memory reference, both instruction and data.
  - Even if the hardware would support such a scheme, the overhead would be tremendous.
  - Alternatively, one could maintain a stack of page references, again an expensive prospect.

LRU Example



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

F = page fault occurring after the frame allocation is initially filled

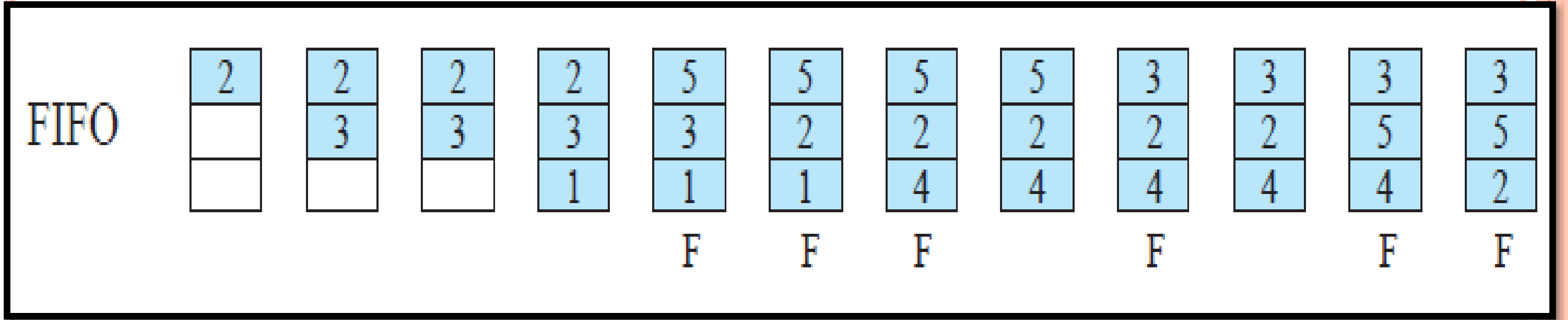
Figure 8.15 **Behavior of Four Page Replacement Algorithms**

- The LRU policy does nearly as well as the optimal policy.
  - In this example, there are four page faults

(3) First-in, first-out (FIFO)

- The first-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style.
- All that is required is a pointer that circles through the page frames of the process.
  - This is one of the simplest page replacement policies to implement.
- The logic behind this choice is that one is replacing the page that has been in memory the longest:
  - A page fetched into memory a long time ago may have now fallen out of use.
  - This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program.
  - Those pages will be repeatedly paged in and out by the FIFO algorithm.

FIFO Example



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

F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms

•Continuing our example in Figure 8.15, the FIFO policy results in six page faults.

•Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

•Although the LRU policy does nearly as well as an optimal policy, it is difficult to implement and imposes significant overhead.

- On the other hand, the FIFO policy is very simple to implement but performs relatively poorly.

(4) Clock Policy

- The simplest form of clock policy requires the association of an additional bit with each frame, referred to as the use bit.
- When a page is first loaded into a frame in memory, the **use bit = 1.**
  - Whenever the page is subsequently referenced, its use bit is set to 1.

The set of frames that are candidates for replacement is considered to be a circular buffer, with which a pointer is associated.

- When a page is replaced, the pointer is set to indicate the next frame in the buffer after the one just updated.
- When it comes time to replace a page, the operating system scans the buffer to find a frame with a **use bit = 0.**
- Each time it encounters a frame with a **use bit = 1**, it resets that bit to zero and continues on.
- If any of the frames in the buffer have a use bit of zero at the beginning of this process, the first such frame encountered is chosen for replacement.
- If all of the frames have a use bit of 1, then the pointer will make one complete cycle through the buffer, setting all the use bits to zero, and stop at its original position, replacing the page in that frame.

This policy is similar to FIFO, except that, in the clock policy, any frame with a use bit of 1 is passed over by the algorithm.

Clock Policy Example

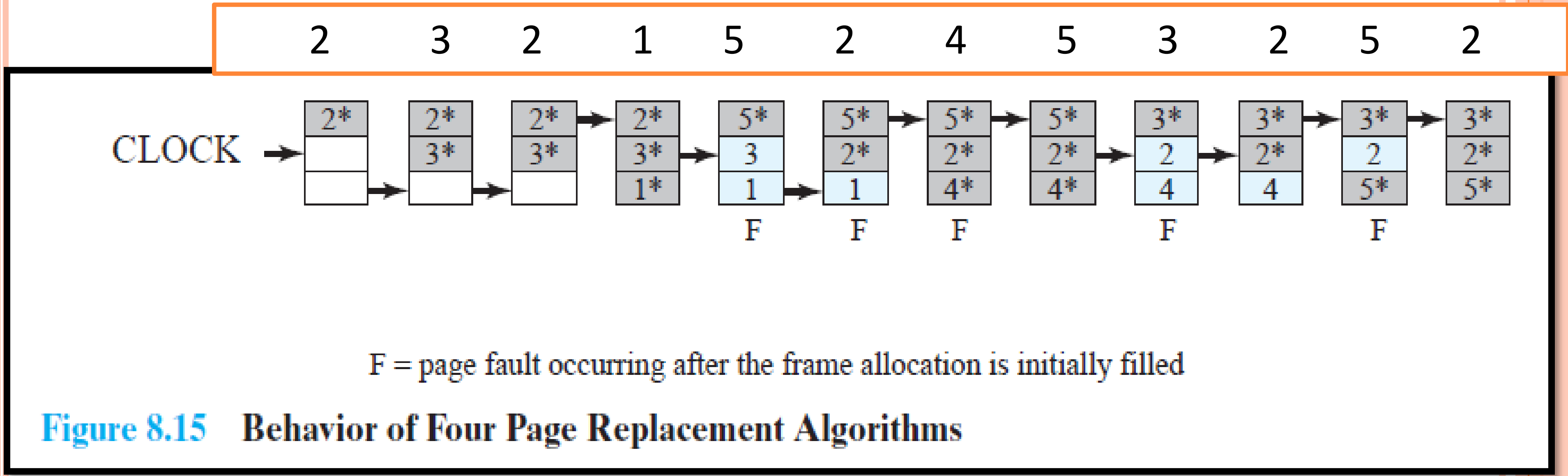
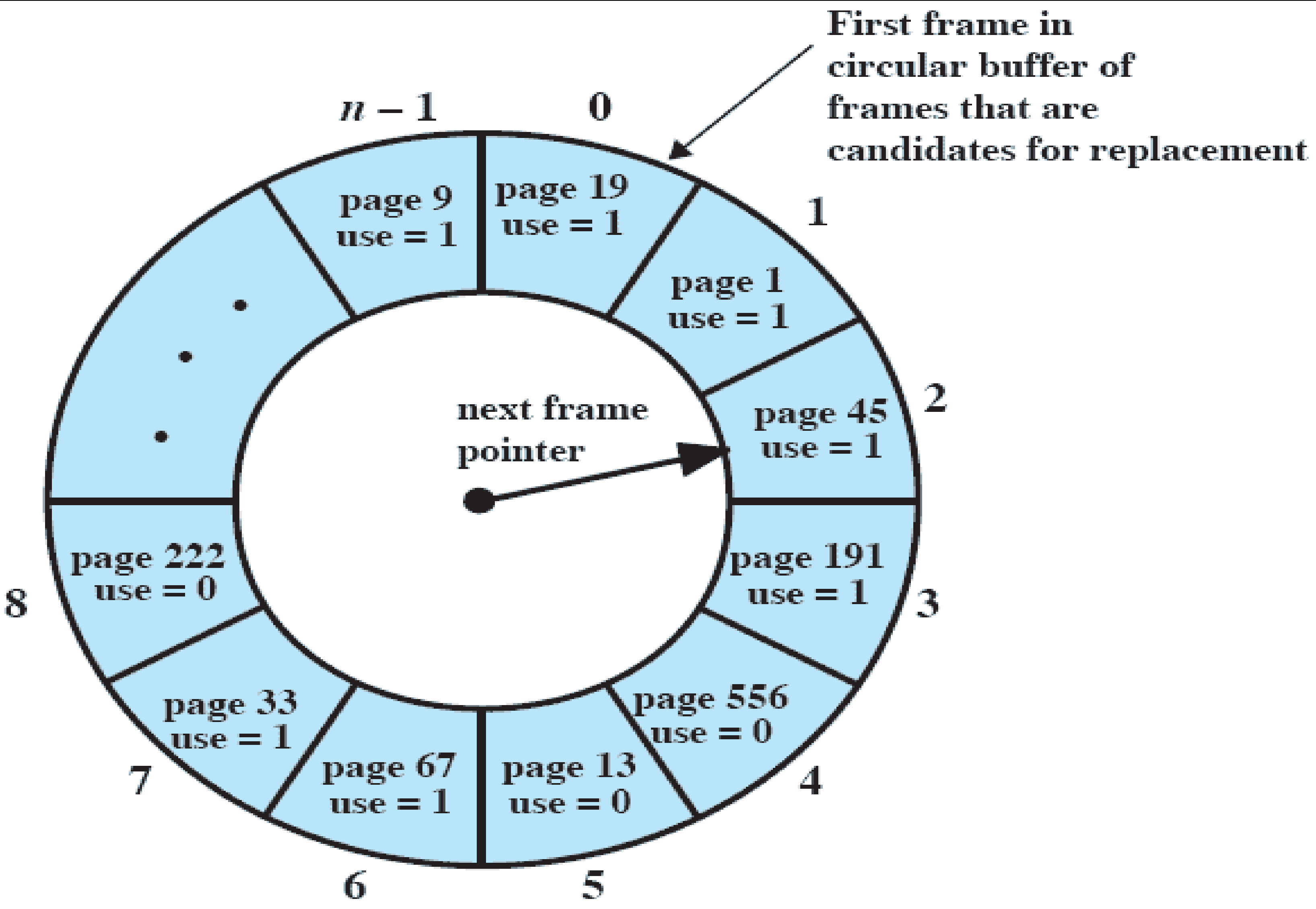

2 3 2 1 5 2 4 5 3 2 5 2
CLOCK
2* 3* 1* 5* 3 1 2* 4* 2 4 5*
F F F F F
F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms

- The presence of an asterisk indicates that the corresponding use bit is equal to 1,
  - and the arrow indicates the current position of the pointer.
- Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

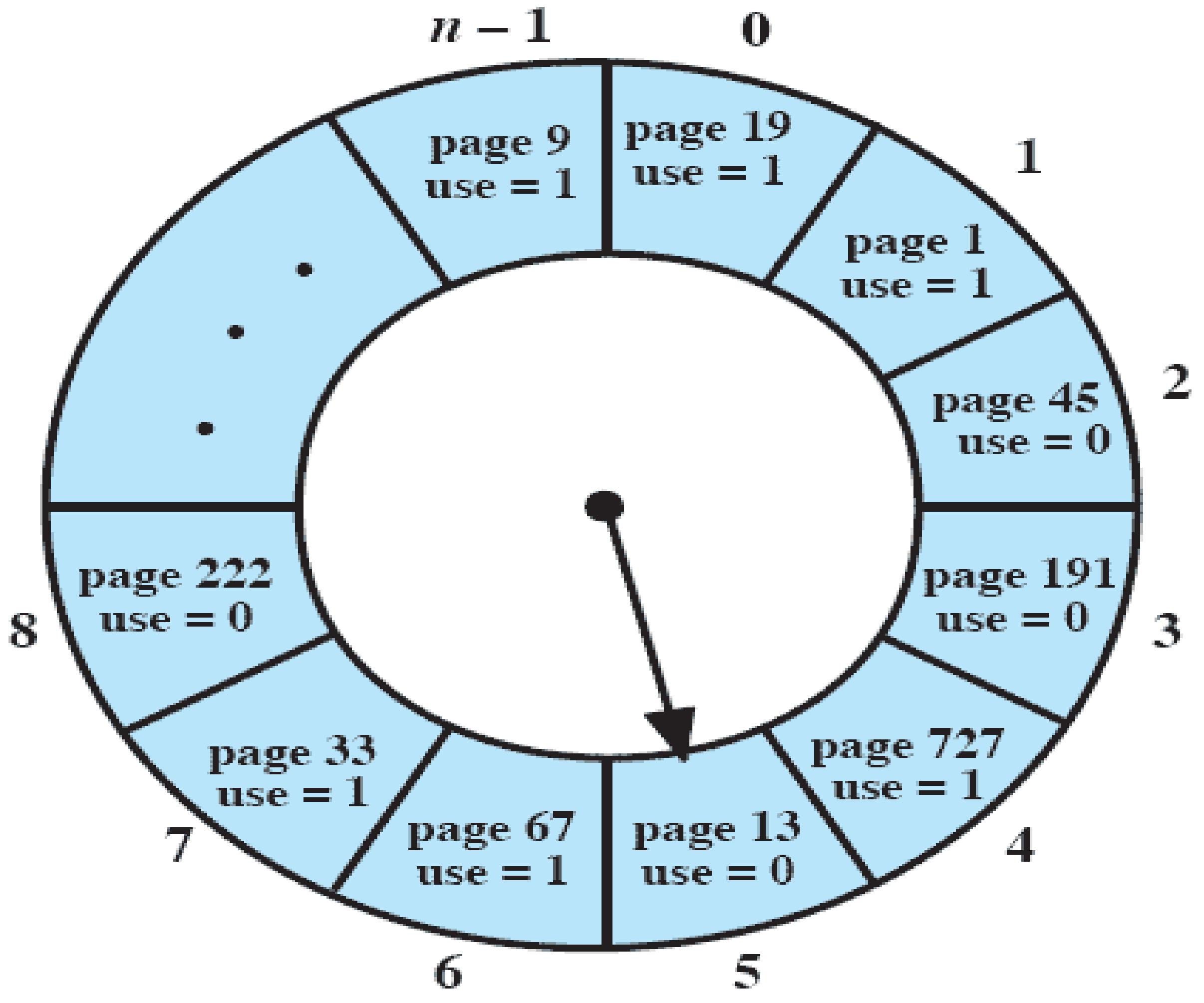
Clock Policy



(a) State of buffer just prior to a page replacement

Figure 8.16 Example of Clock Policy Operation

Clock Policy



(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

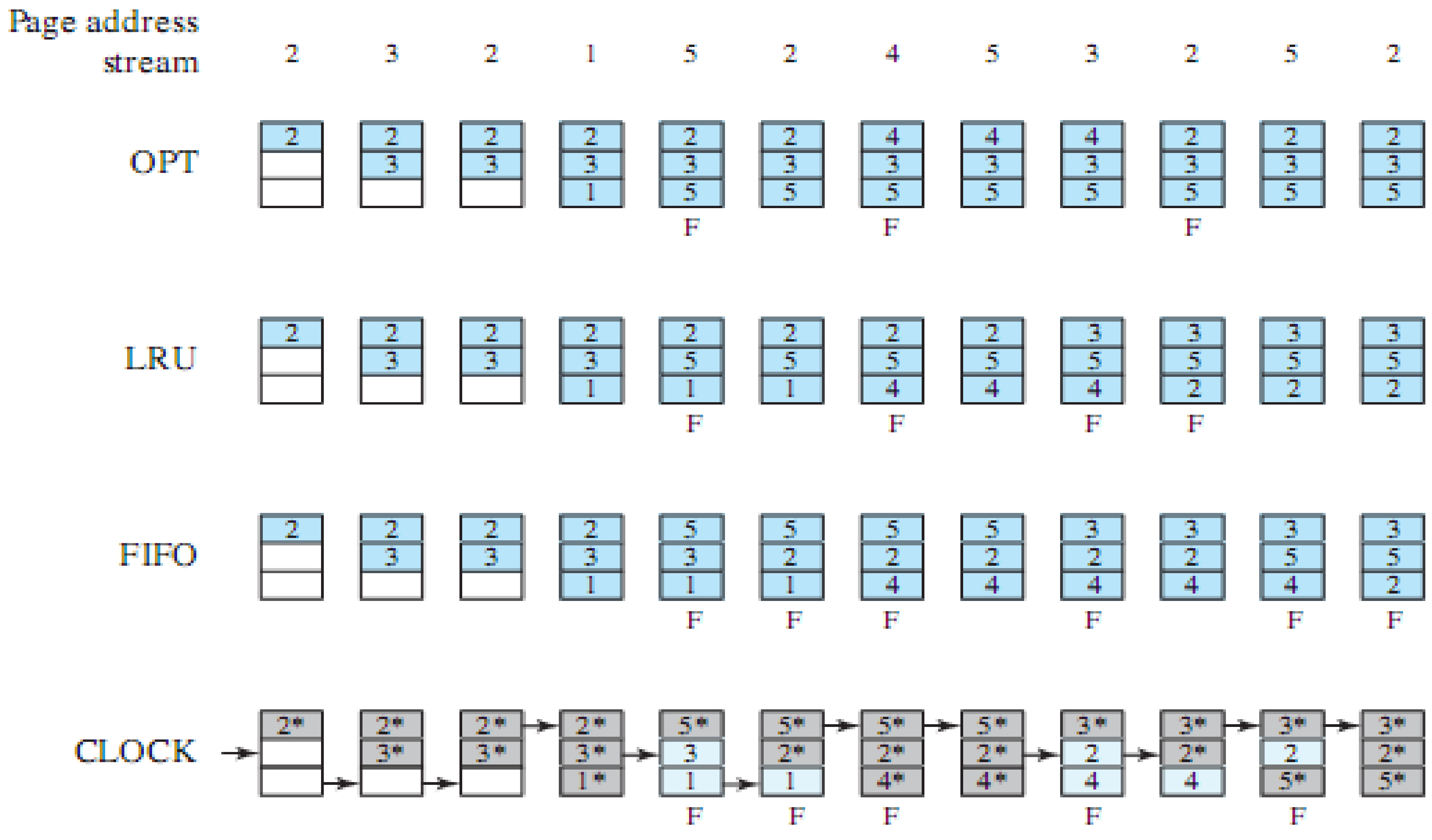
Clock Policy

- This figure (8.16) provides an example of the simple clock policy mechanism.
- A circular buffer of *n* main memory frames is available for page replacement.
- Just prior to the replacement of a page from the buffer with incoming page 727,
  - the next frame pointer points at frame 2, which contains page 45.

The clock policy is now executed.

- Because the use bit for page 45 in frame 2 is equal to 1, this page is not replaced.
- Instead, the use bit is set to zero and the pointer advances.
- Similarly, page 191 in frame 3 is not replaced; its use bit is set to zero and the pointer advances.
- In the next frame, frame 4, the use bit is set to 0.Therefore, page 556 is replaced with page 727.
- The use bit is set to 1 for this frame and the pointer advances to frame 5, completing the page replacement procedure.

COMBINED EXAMPLES



F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms

Comparison

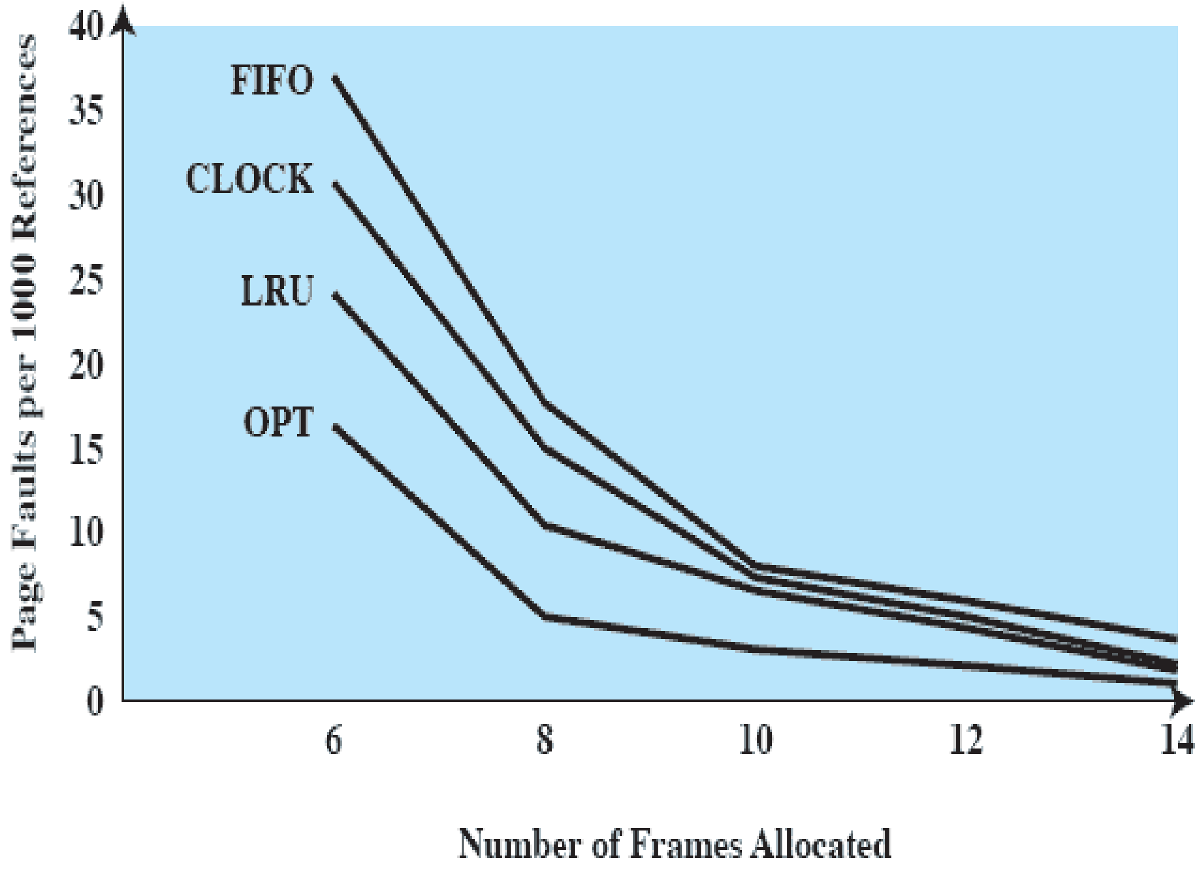


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

UNIT – 4

COMPLETED