Operating Systems: Internals and Design Principles

Chapter 8 Virtual Memory

Ninth Edition
William Stallings

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.

Table 8.1 Virtual Memory Terminology

Hardware and Control Structures

- Two characteristics fundamental to memory management:
  1) All memory references are logical addresses that are dynamically translated into physical addresses at run time. This means that a process may be swapped in and out of main memory such that it occupies different regions of main memory at different times during the course of execution
  2) A process may be broken up into a number of pieces (pages or segments) that don't need to be contiguously located in main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this

Hardware and Control Structures

- Now we come to the breakthrough. If the preceding two characteristics are present, then it is not necessary that all of the pages or all of the segments of a process be in main memory during execution. If the piece (segment or page) that holds the next instruction to be fetched and the piece that holds the next data location to be accessed are in main memory, then at least for a time execution may proceed

Execution of a Process

- Operating system brings into main memory a few pieces (pages or segments) of the program
- **Resident set**
  - Portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory (**memory access fault**)
- Operating system places the process in a blocking state

Execution of a Process

- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O Read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O is complete, which causes the operating system to place the affected process in the Ready state

Implications

More processes may be maintained in main memory

- Because only some of the pieces of any particular process are loaded, there is room for more processes
- This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in a **Ready** state at any particular time

Implications

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 OS and the hardware
- The OS automatically loads pieces of a process into main memory as required

Real and Virtual Memory

- Because a process executes only in main memory, that memory is referred to as real memory
- But a programmer or user perceives a potentially much larger memory–that which is allocated on disk. This latter is referred to as virtual memory
  - Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main 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 not partitioned	
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)	
Internal fragmentation within frames		No internal fragmentation	
No 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 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 list of free holes in main memory	
Processor uses page 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 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

- However, the operating system must be clever about how it manages this scheme. **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 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

Principle of Locality

- The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms. In essence, the operating system tries to guess, based on recent history, which pieces are least likely to be used in the near future
- The principle of locality suggests that a virtual memory scheme may be effective: program and data references within a process tend to cluster. Hence, only a few pieces of a process will be needed over a short period of time
- It is possible to make intelligent guesses about which pieces will be needed in the future, which avoids thrashing

Support Needed for Virtual Memory

For virtual memory to be practical and effective:

- Hardware must support paging and segmentation
- Operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory

Paging

- The term *virtual memory* is usually associated with systems that employ paging, although virtual memory based on segmentation is also used and will be discussed next
- Use of paging to achieve virtual memory was first reported for the Atlas computer
- Each process has its own page table
  - Each page table entry (PTE) 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

A bit is needed in each page table entry to indicate whether the corresponding page is present (P) in main memory (1) or not (0). 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 also includes a modify (M) bit, indicating whether the contents of the corresponding page have been altered since the page was last loaded into main memory. If there has been no change, then it is not necessary to write the page out when it comes time to replace the page in the frame that it currently occupies

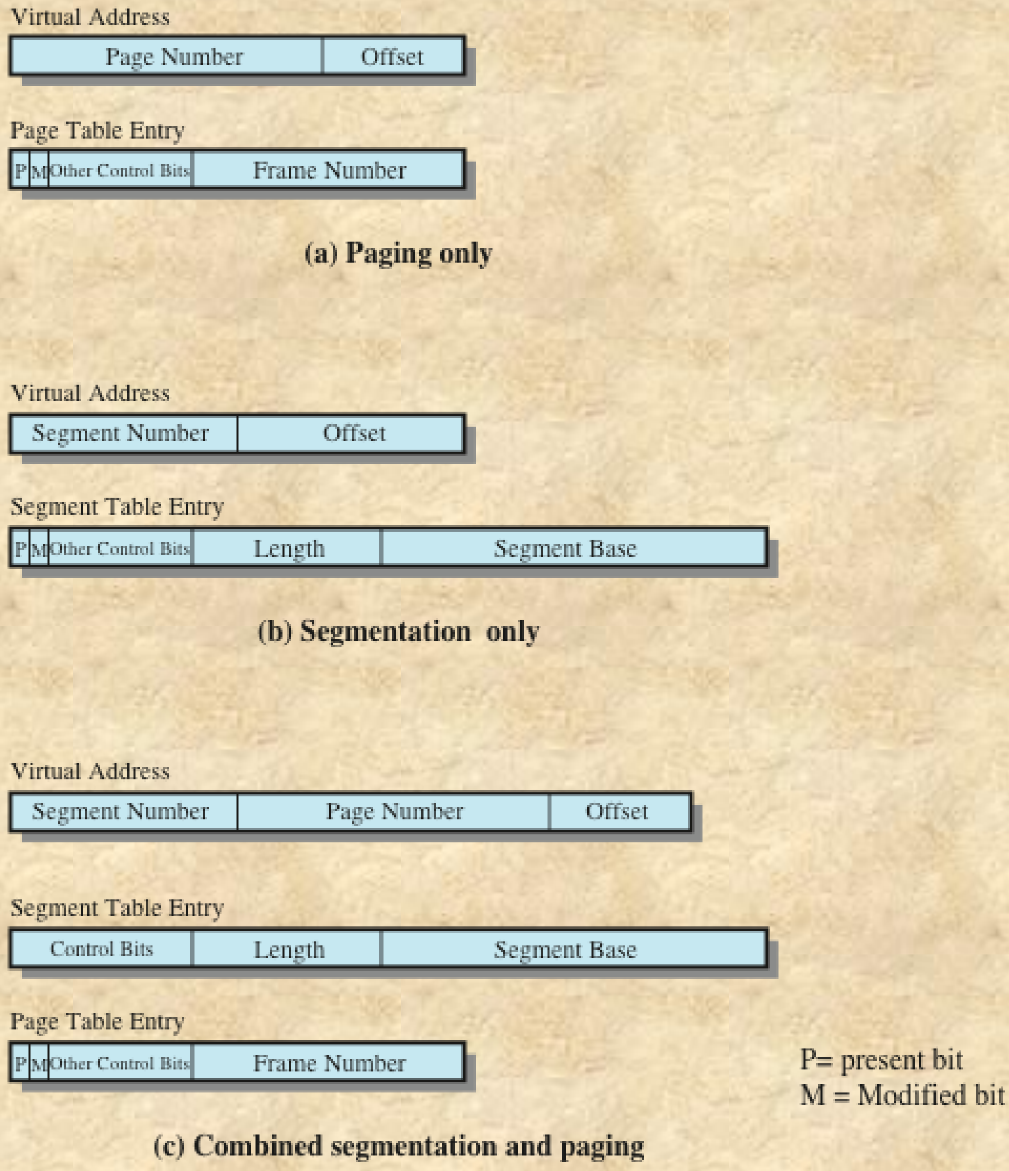


Figure 8.1 Typical Memory Management Formats

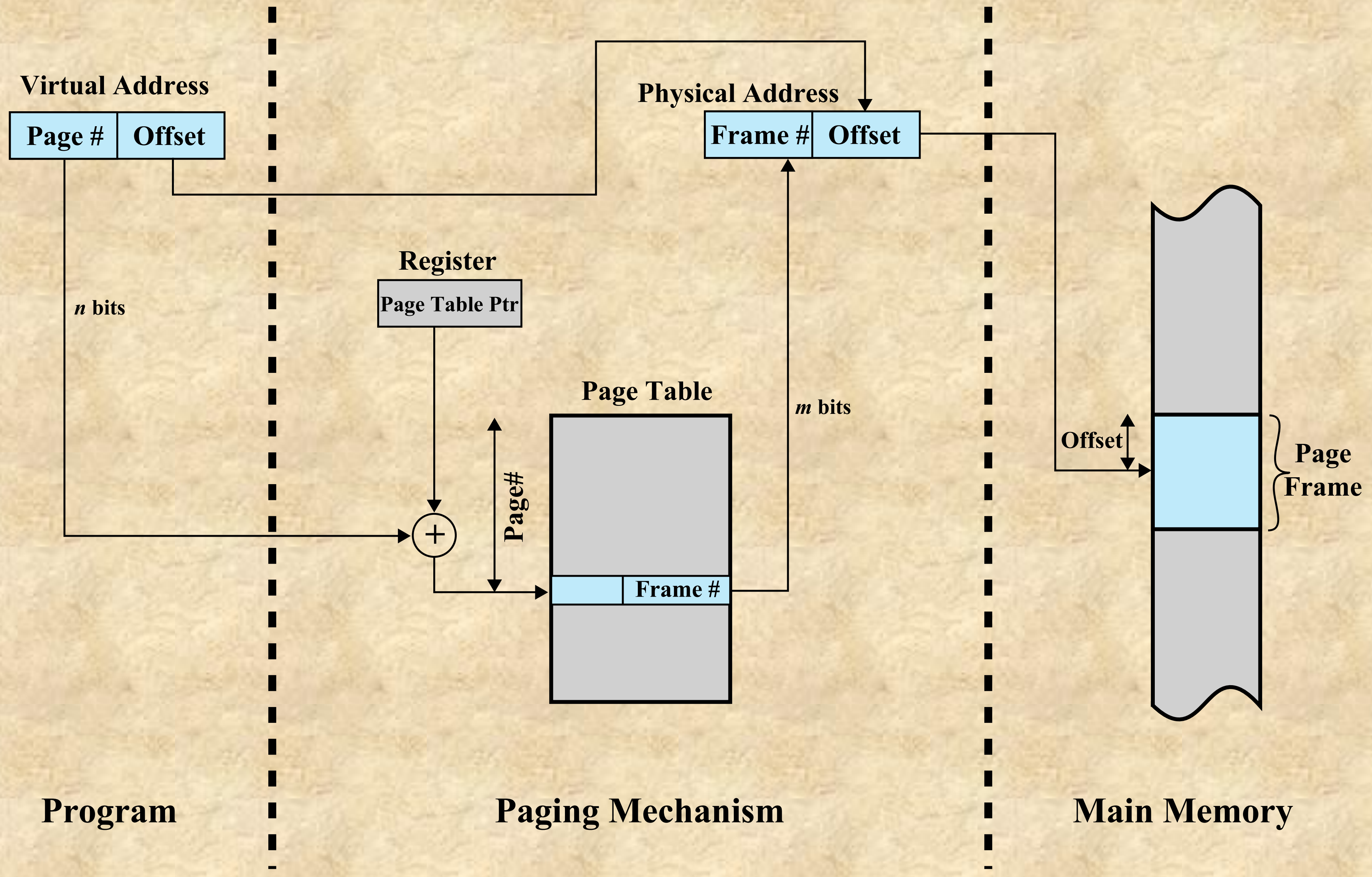


Figure 8.2 Address Translation in a Paging System

Address Translation for Paging

- In Figure 8.2, 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 ($n > m$). Why?

Page Table Structure

- Page tables can take a lot of memory
  - In VAX architecture, each process can have up to 2^31 byte = 2 GB of virtual memory
  - Using 2^9 = 512-byte pages, need 2^22 page table entries per process!
  - Store page tables in virtual memory rather than real memory

32-bit address (4 GB or 2^{32} bytes virtual address space)
4-kbyte (2^{12} bytes) pages
4-byte page table entry

User page table: 2^{20} PTEs * 4 bytes per PTE = 4 MB

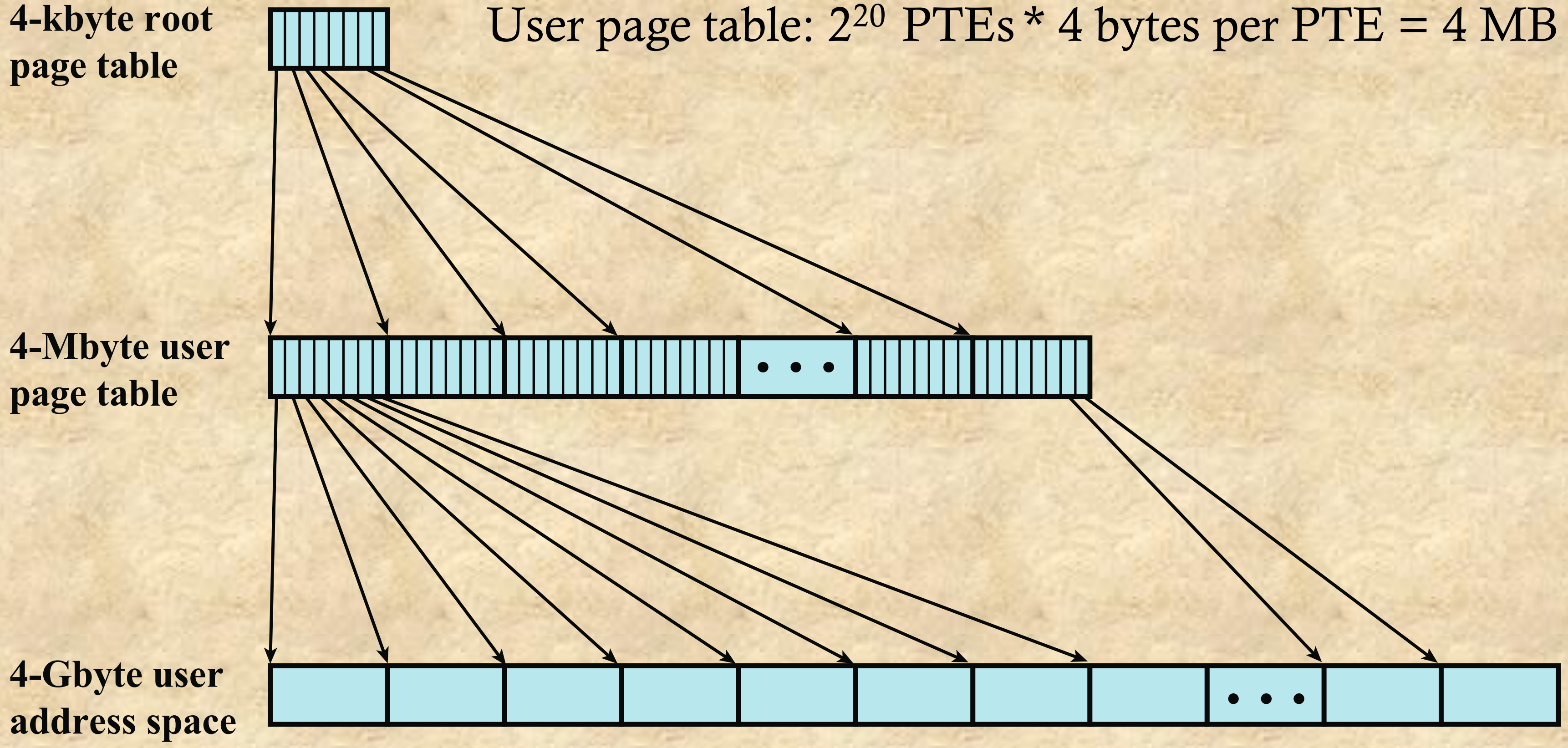


Figure 8.3 A Two-Level Hierarchical Page Table

Address Translation for Two-Level Hierarchical Page Table

- The root page always remains in main memory
- The first 10 bits of a virtual address are used to index into the root page to find a PTE for a page of the user page table. If that page is not in main memory, a page fault occurs. If that page is in main memory, then the next 10 bits of the virtual address index into the user PTE page to find the PTE for the page that is referenced by the virtual address

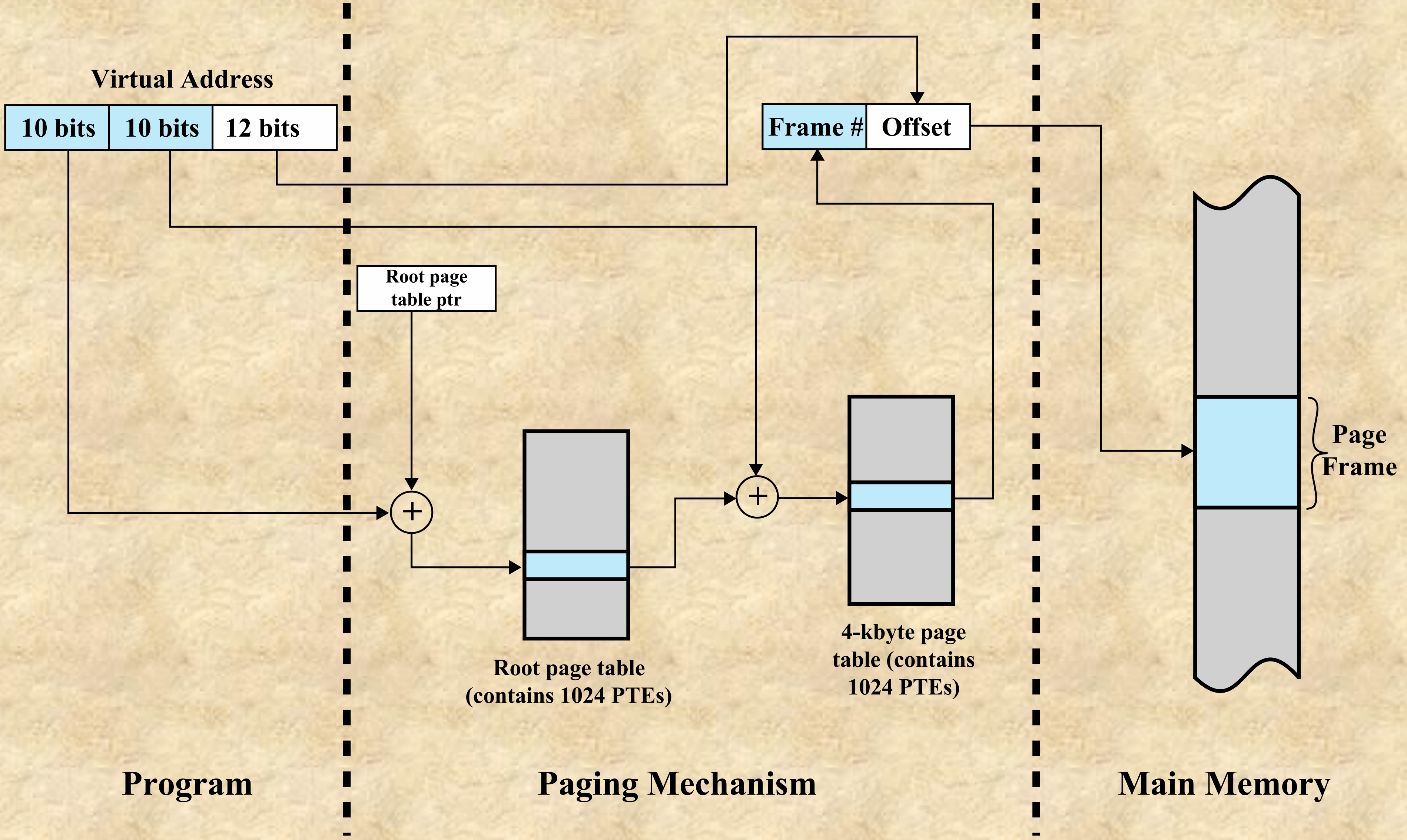
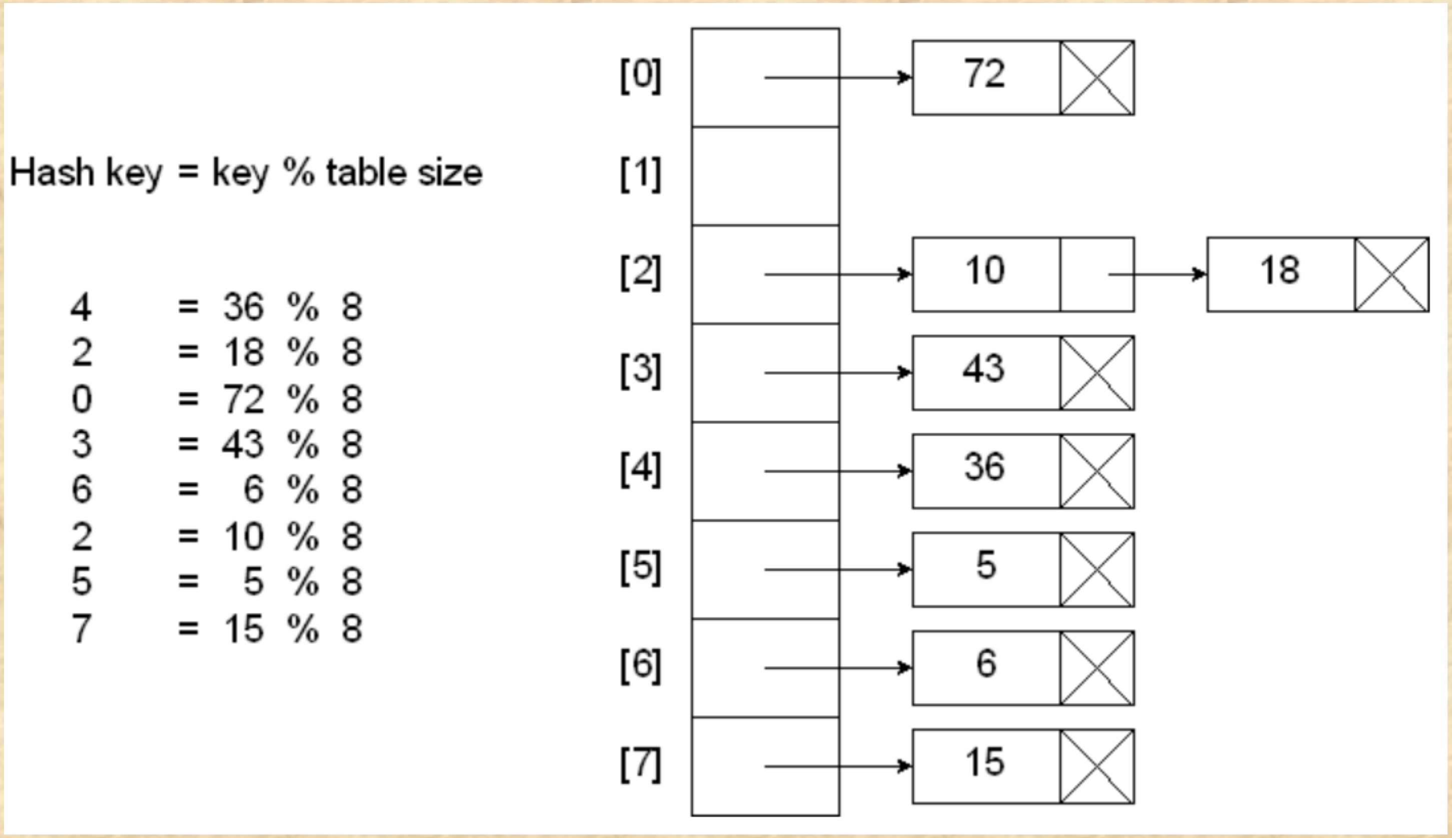


Figure 8.4 Address Translation in a Two-Level Paging System

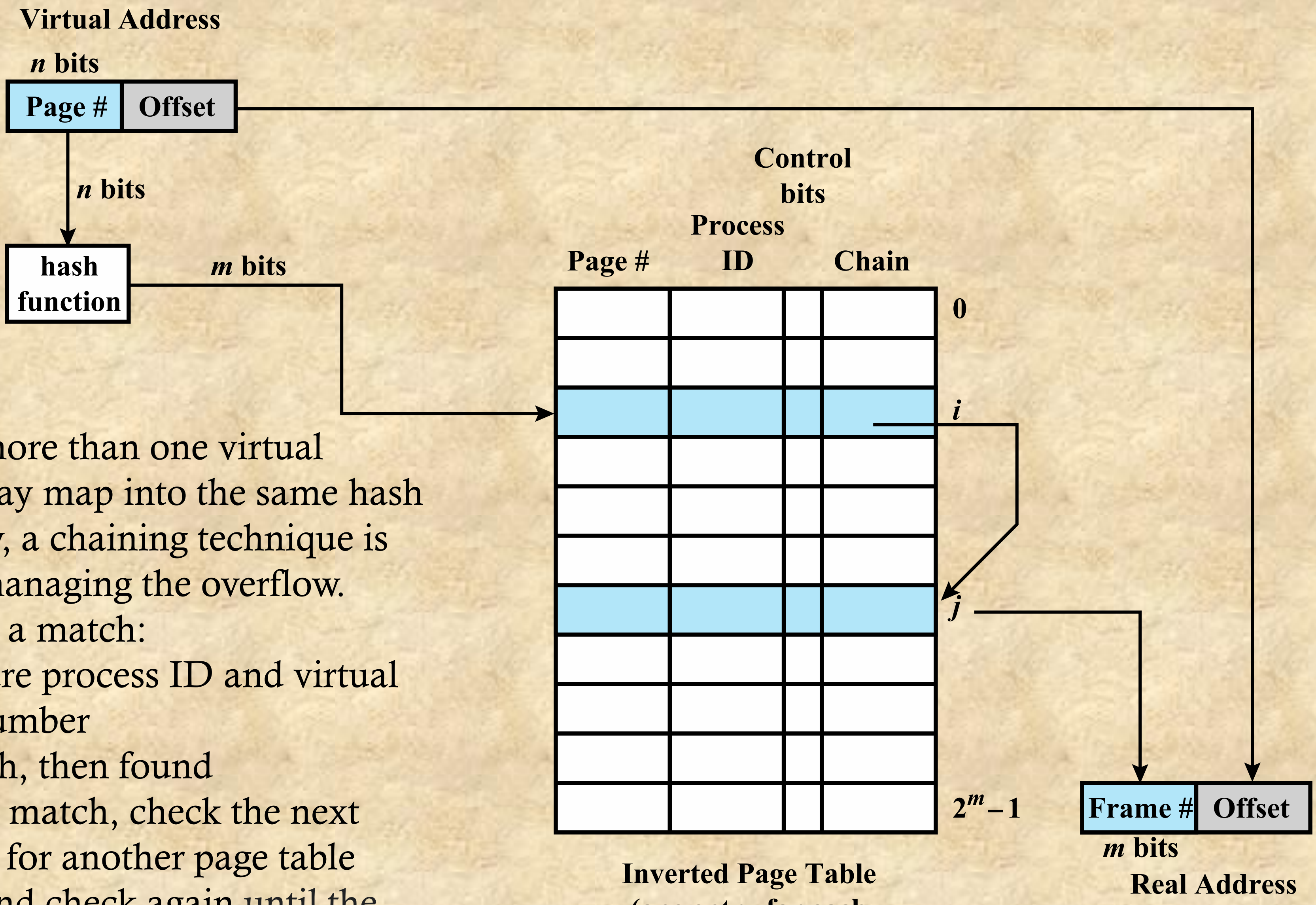
Inverted Page Table

- A drawback of using one or multiple-level page tables is that their size is proportional to that of the virtual address space
- Alternative approach – inverted page table
  - Page number portion of a virtual address (can be together with the process ID) is mapped into a hash value
  - Hash value points to inverted page table
  - One entry for each real memory page frame, rather than per virtual page
- Fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported
- Structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number

Hash Example



http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm



Because more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow.
Search for a match:

- Compare process ID and virtual page number
- if match, then found
- if not a match, check the next pointer for another page table entry and check again until the chain has been exhausted and **a page fault occurs**

Figure 8.5 Inverted Page Table Structure

Inverted Page Table

Each entry in the page table includes:

Page number

- This is the page number portion of the virtual address

Process identifier

- The process that owns this page

Control bits

- Includes flags and protection and locking information

Chain pointer

- The index value of the next entry in the chain

Translation Lookaside Buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses:
  - One to fetch the page table entry
  - One to fetch the data

- To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a ***translation lookaside buffer*** (TLB)
  - This cache functions in the same way as a memory cache and contains those page table entities that have been most recently used

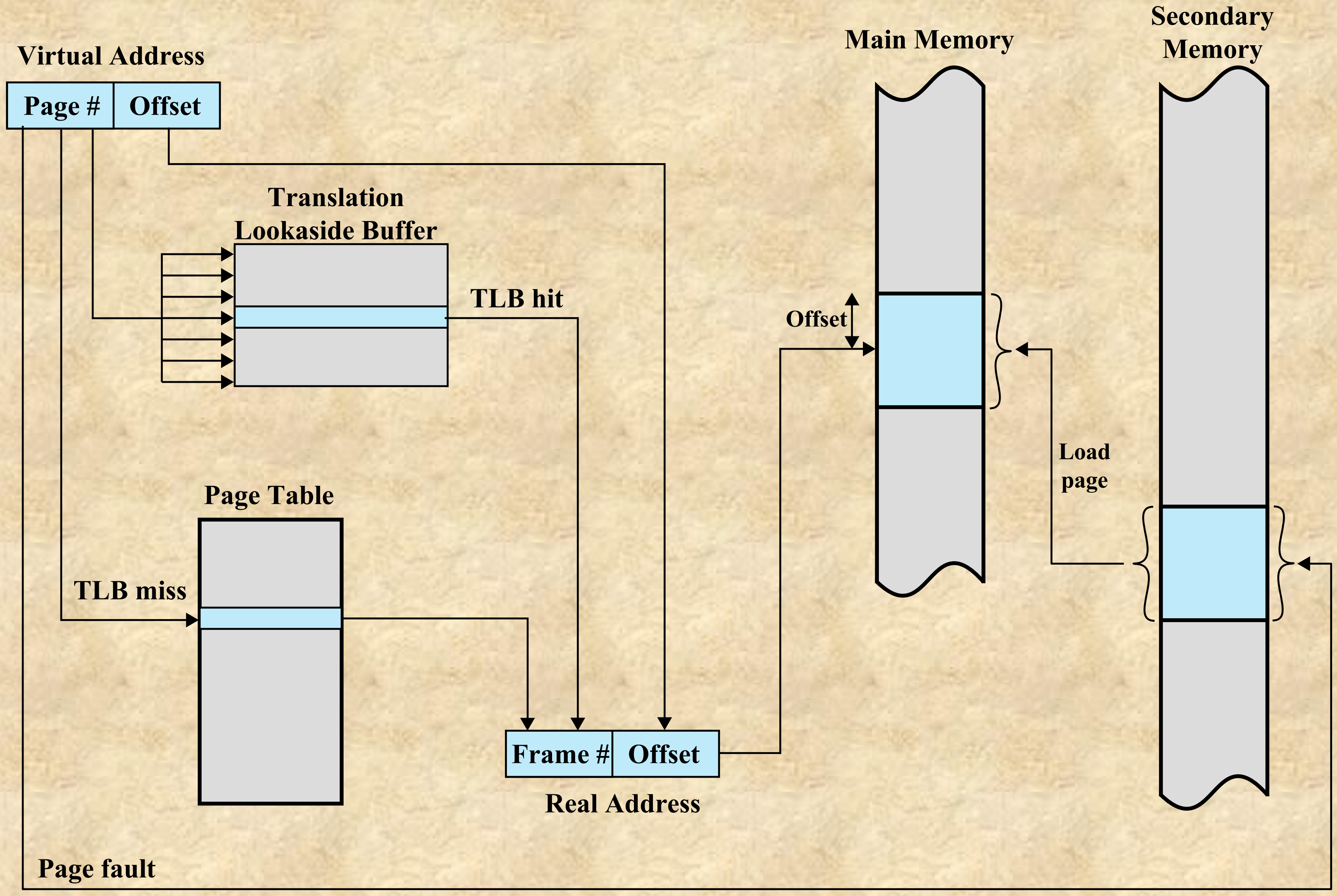


Figure 8.6 Use of a Translation Lookaside Buffer

TLB

- Given a virtual address, the processor will first examine the TLB. If the desired page table entry is present (TLB hit), then the frame number is retrieved and the real address is formed. If the desired page table entry is not found (TLB miss), then the processor uses the page number to index the process page table and examine the corresponding page table entry. If the "present bit" is set, then the page is in main memory, and the processor can retrieve the frame number from the page table entry to form the real address. The processor also updates the TLB to include this new page table entry. Finally, if the present bit is not set, then the desired page is not in main memory and a memory access fault, called a page fault, is issued

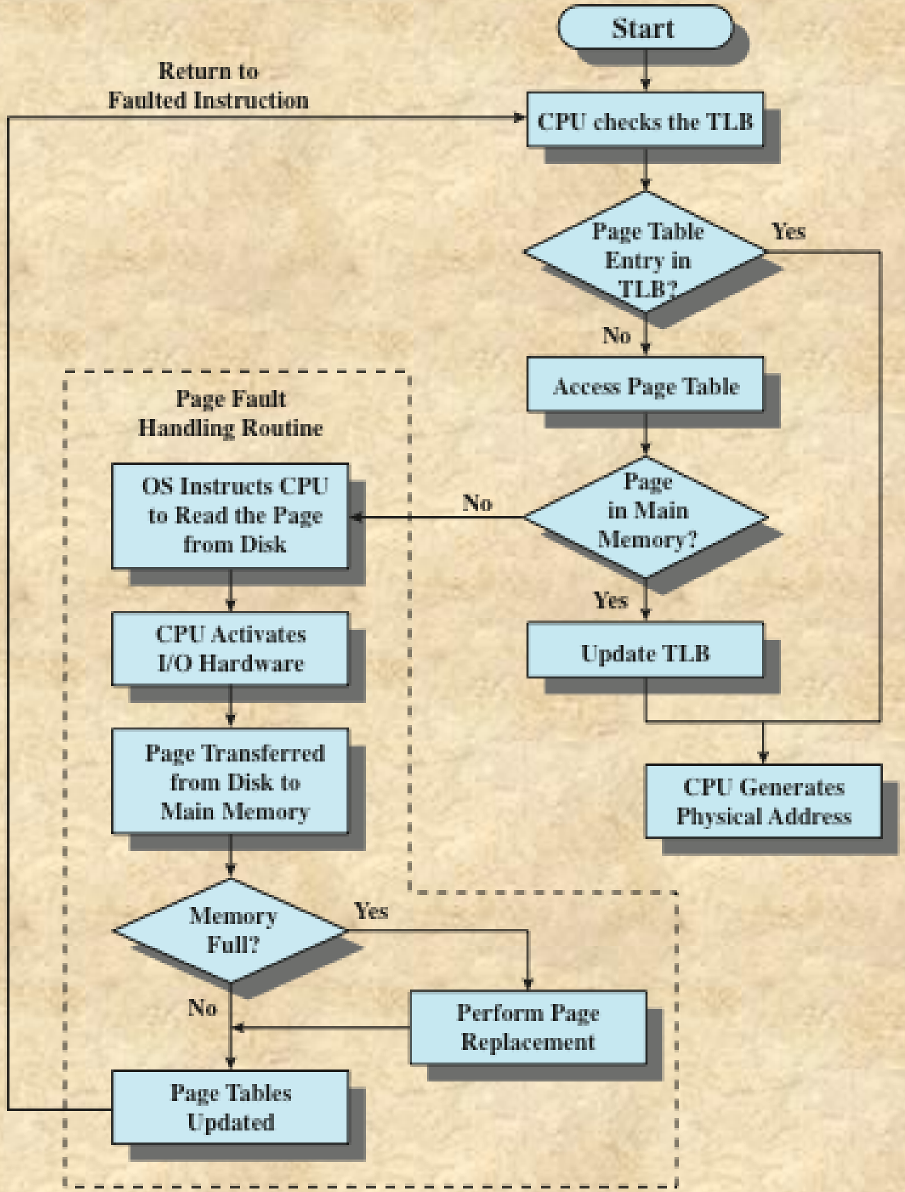

Start
Return to Faulted Instruction
CPU checks the TLB
Page Table Entry in TLB?
Yes
No
Access Page Table
Page Fault Handling Routine
OS Instructs CPU to Read the Page from Disk
No
Page in Main Memory?
Yes
Update TLB
CPU Activates I/O Hardware
Page Transferred from Disk to Main Memory
CPU Generates Physical Address
Memory Full?
Yes
No
Perform Page Replacement
Page Tables Updated

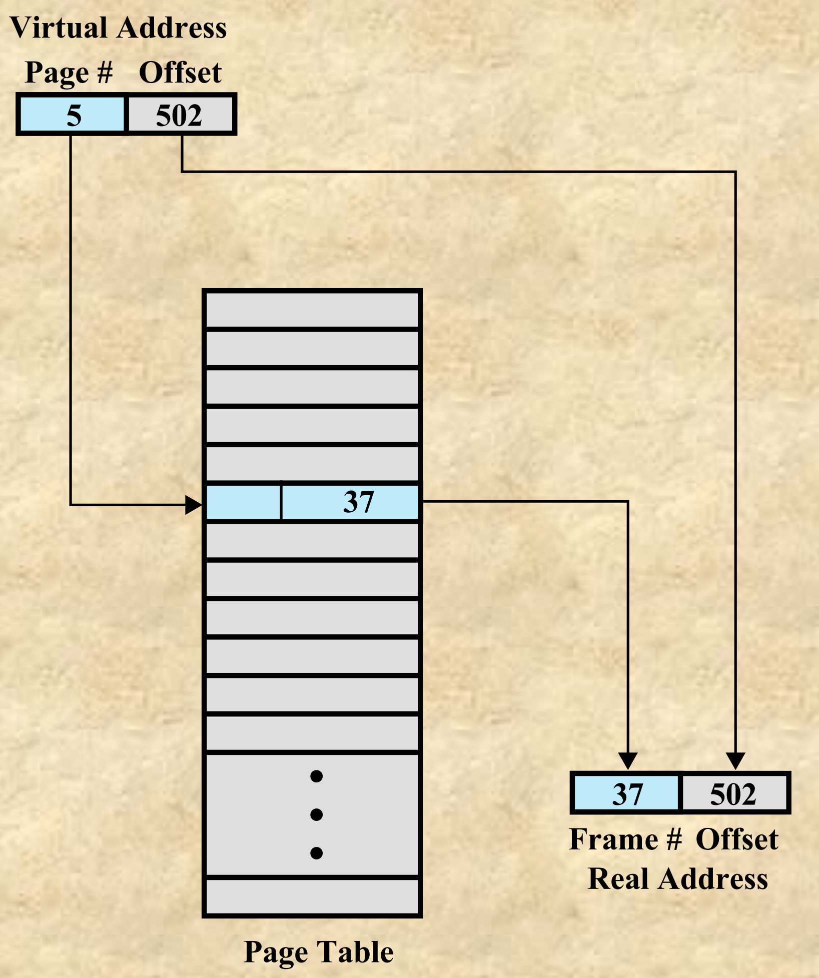
Figure 8.7 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

TLB

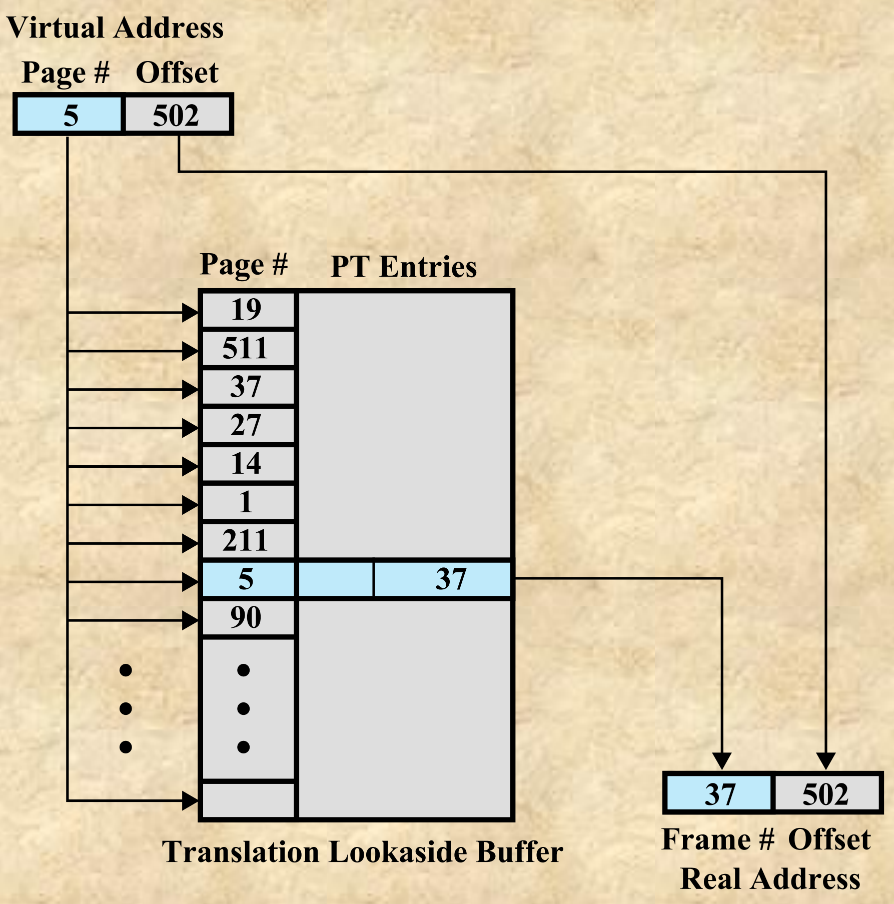
- Figure 8.7 is a flowchart that shows the use of the TLB. The flowchart shows that if the desired page is not in main memory, a page fault interrupt causes the page fault handling routine to be invoked. To keep the flowchart simple, the fact that the operating system may dispatch another process while disk I/O is underway is not shown. **By the principle of locality, most virtual memory references will be to locations in recently used pages. Therefore, most references will involve page table entries in the cache**

Associative Mapping

- The TLB only contains some of the page table entries so we cannot simply index into the TLB based on page number
  - Each TLB entry must include the page number as well as the complete page table entry
- The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number



(a) Direct mapping



(b) Associative mapping

Figure 8.8 Direct Versus Associative Lookup for Page Table Entries

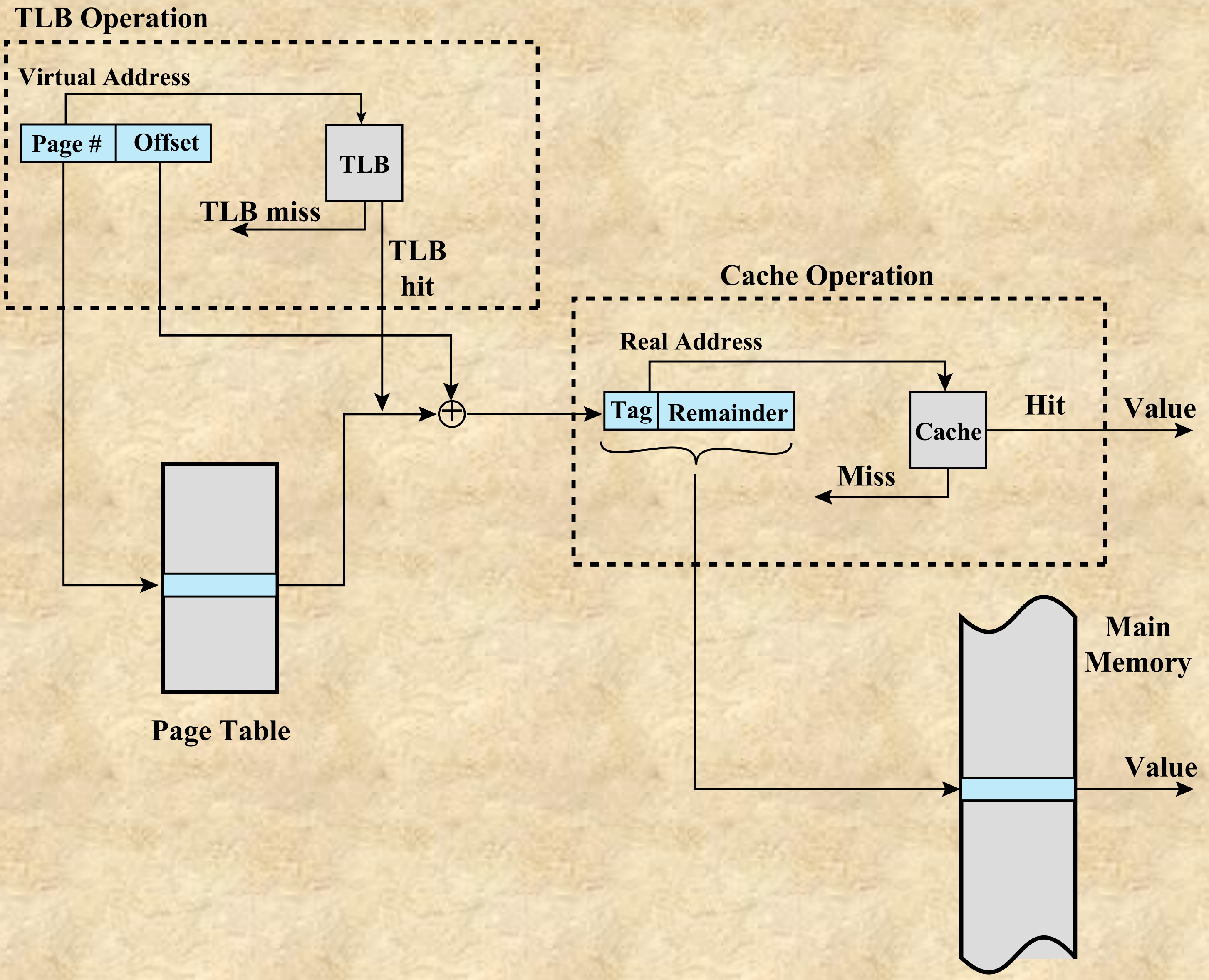


Figure 8.9 Translation Lookaside Buffer and Cache Operation

TLB and Cache

Finally, the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in Figure 8.9. A virtual address will generally be in the form of a page number, offset. First, the memory system consults the TLB to see if the matching page table entry is present. If it is, the real (physical) address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present. If so, it is returned to the CPU. If not, the word is retrieved from main memory

Page Size

- The smaller the page size, the lesser the amount of internal fragmentation
- However, more pages are required per process
  - More pages per process means larger page tables. For large programs in a heavily multiprogrammed environment, some the page tables of active processes are in virtual memory instead of main memory, and there may be a double page fault for a single reference to memory: first to bring in the needed portion of the page table and second to bring in the process page

Page Size

- The physical characteristics of most secondary-memory devices favor a larger page size for more efficient block transfer of data

Table 8.3

Example of Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Segmentation

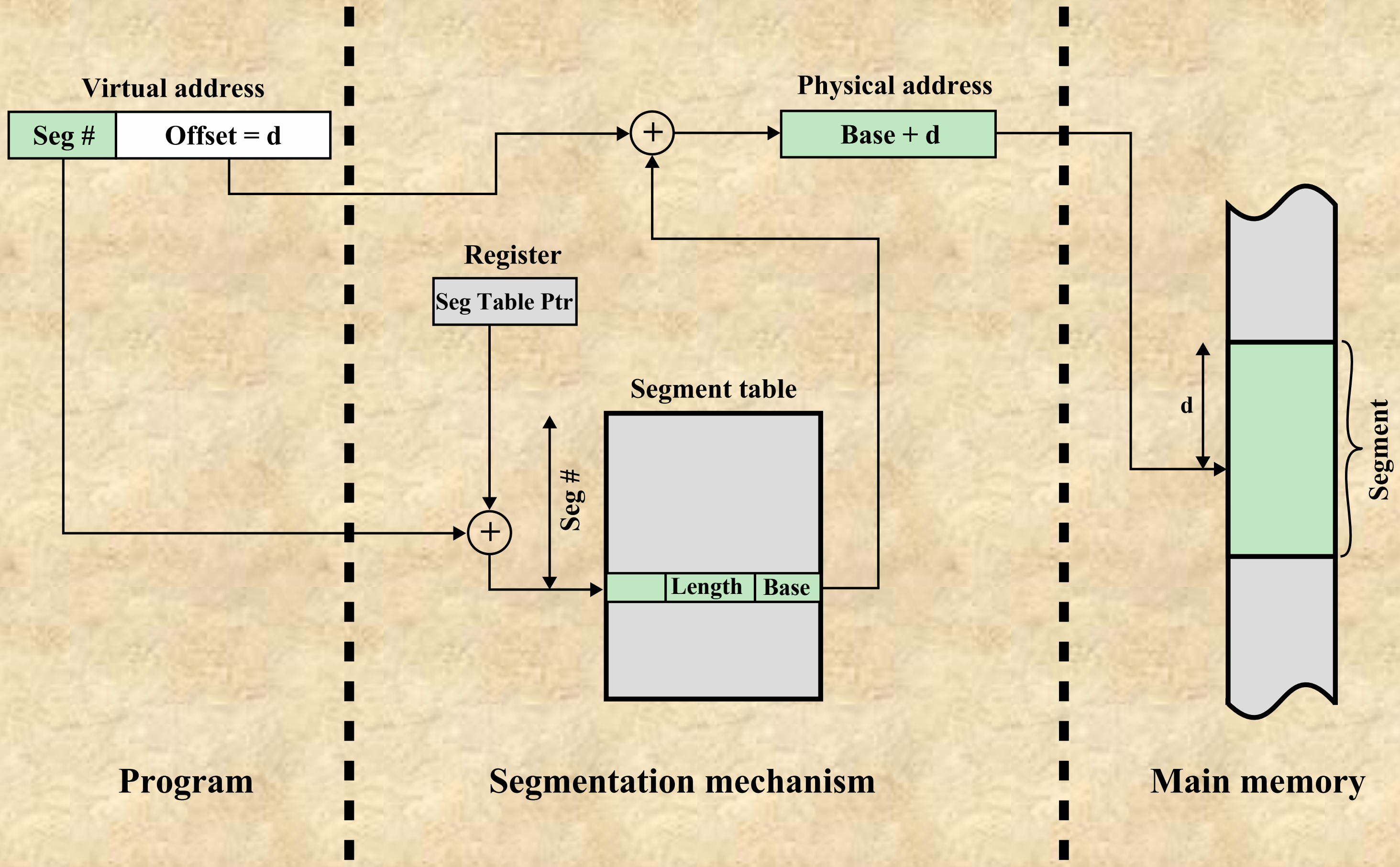
Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments

Advantages:

- Simplifies handling of growing data structures
- Allows programs to be altered and recompiled independently
- Lends itself to sharing data among processes
- Lends itself to protection

Segment Organization

- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment
- A bit is needed to determine if the segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory



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.

Figure 8.11 Address Translation in a Segmentation System

Paging and Segmentation

- Paging
  - Transparent to the programmer, eliminates external fragmentation and thus provides efficient use of main memory
  - The pieces that are moved in and out of main memory are of fixed, equal size. It is possible to develop sophisticated memory management algorithms that exploit the behavior of programs
- Segmentation
  - Visible to the programmer
  - The ability to handle growing data structures, modularity, and support for sharing and protection

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 broken up into a number of fixed-sized pages which are equal in length to a main memory frame

- Segmentation is visible to the programmer
- Paging is transparent to the programmer

Also called a segmented paging system
https://www.javatpoint.com/os-segmented-paging

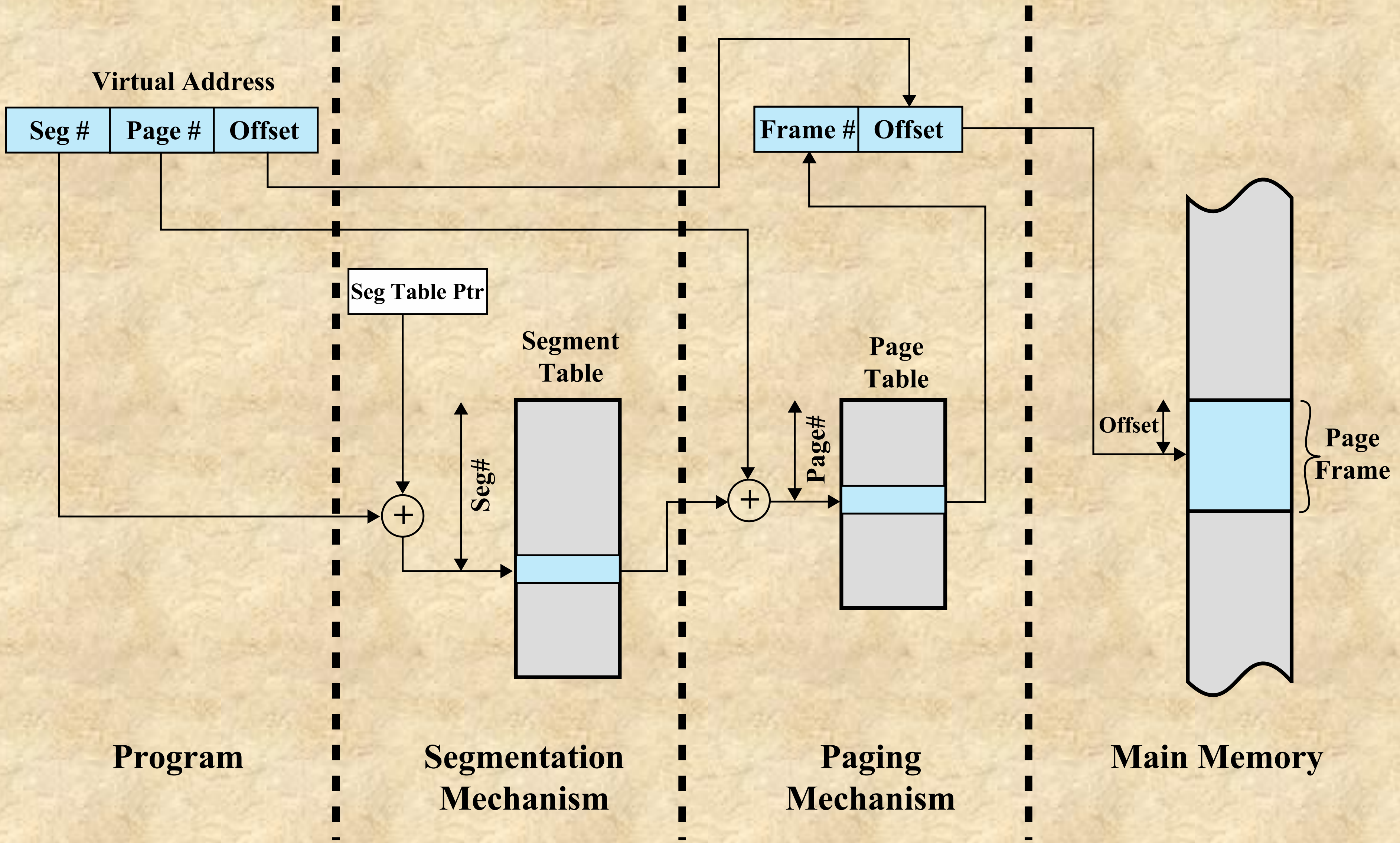
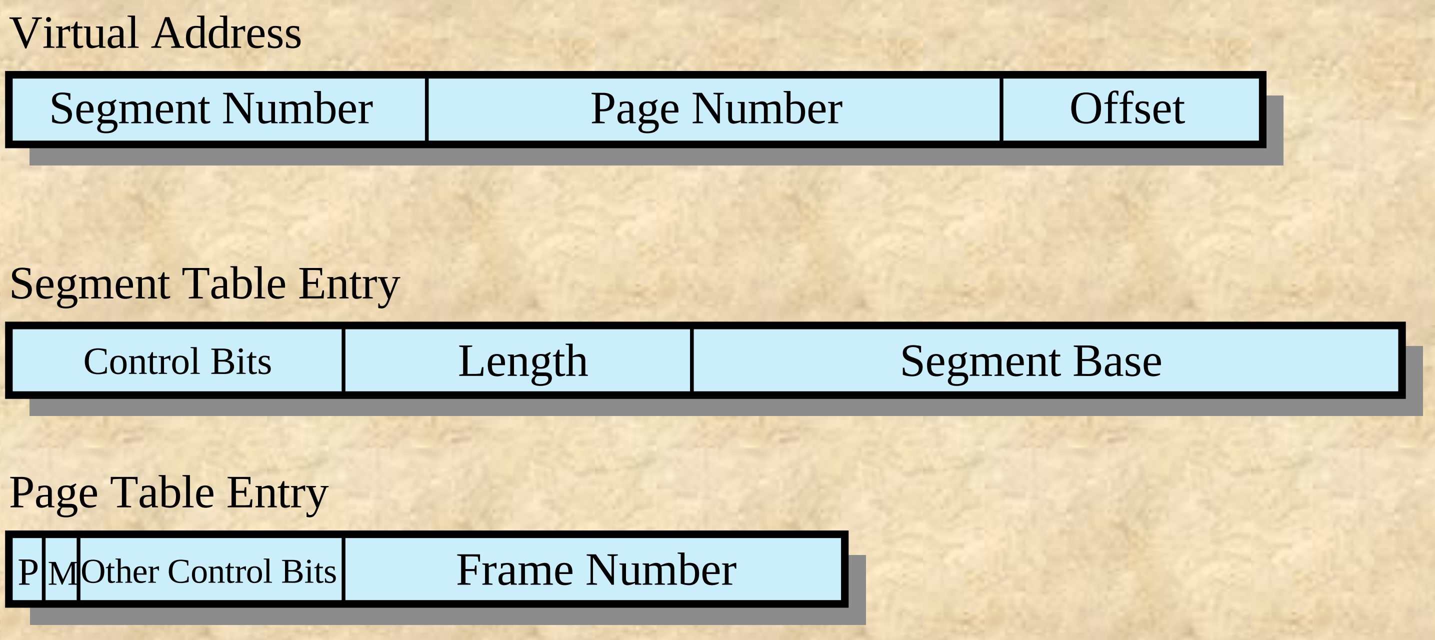


Figure 8.12 Address Translation in a Segmentation/Paging System

Address Translation in Segmented Paging System

- Associated with each process is a segment table and a number of page tables, one per process segment. Now each virtual address consists of a segment number, a page number within that segment, and an offset within that page
- When a particular process is running, a register holds the starting address of the segment table for that process. The processor uses the segment number portion to index into the process segment table to find the page table for that segment. The remainder of the address, page number and offset, regarded as the segment offset, is checked against the limit of the segment. A trap is generated if this limit is violated. Otherwise, the page number portion of the virtual address is used to index the page table and look up the corresponding frame number. Finally, the physical address is obtained by adding the frame address and the offset



P= present bit

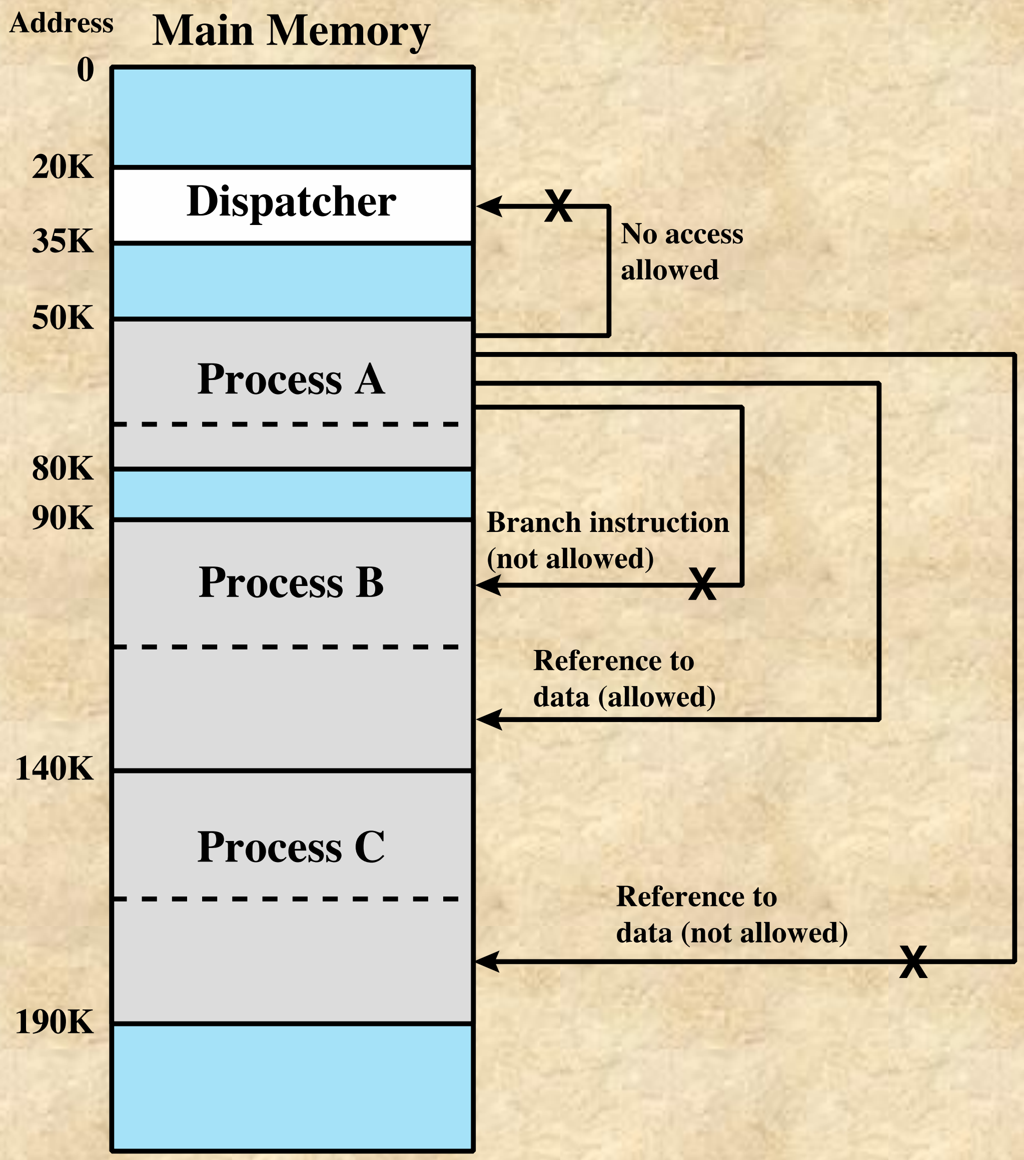
M = Modified bit

(c) Combined segmentation and paging

Figure 8.1 Typical Memory Management Formats

Protection and Sharing

- Segmentation lends itself to the implementation of protection and sharing policies
- Each entry has a base address and length so inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes



Process A (the instruction segment) can access the data segment of Process B, but not the instruction segment of Process B or the data segment of Process C

Figure 8.13 Protection Relationships Between Segments

Operating System Software

- The design of the memory management portion of an operating system depends on three fundamental areas of choice:
  - Whether or not to use virtual memory techniques
  - The use of paging or segmentation or both
  - The algorithms employed for various aspects of memory management

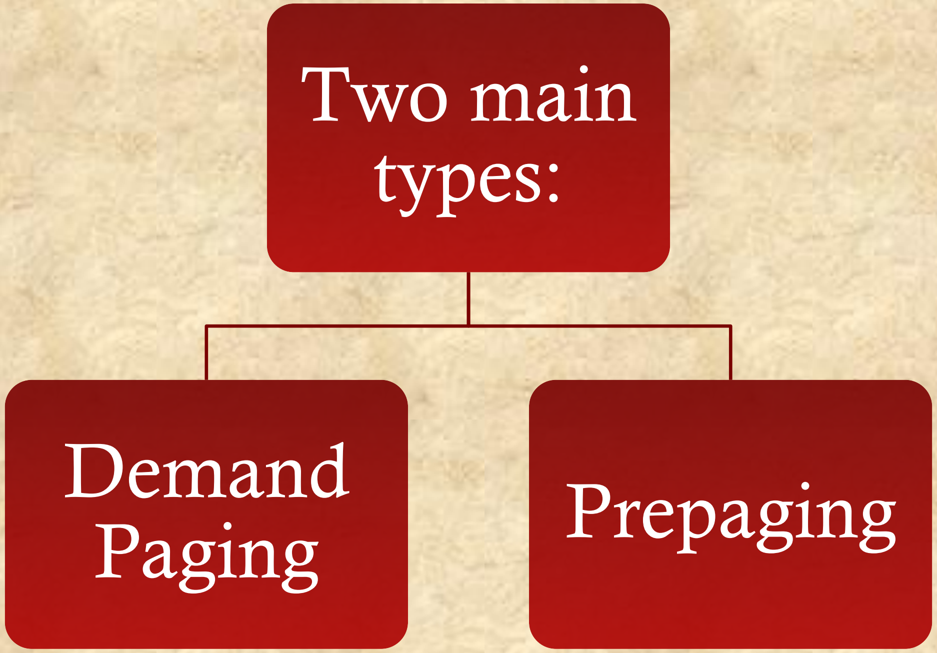
The key issue is to minimize the rate at which page faults occur

Fetch Policy Demand paging Prepaging **Placement Policy** **Replacement Policy** Basic Algorithms Optimal Least recently used (LRU) First-in-first-out (FIFO) Clock Page Buffering	**Resident Set Management** Resident set size Fixed Variable Replacement Scope Global Local **Cleaning Policy** Demand Precleaning **Load Control** Degree of multiprogramming

Table 8.4 Operating System Policies for Virtual Memory

Fetch Policy

- Determines when a page should be brought into memory



Demand Paging

- **Demand Paging**
  - Only brings pages into main memory when a reference is made to a location on the page
  - Many page faults when process is first started
  - Principle of locality suggests that as more and more pages are brought in, most future references will be to pages that have recently been brought in, and page faults should drop to a very low level

Prepaging

- **Prepaging**
  - Pages other than the one demanded by a page fault are brought in
  - Exploits the characteristics of most secondary memory devices
  - If pages of a process are stored contiguously in secondary memory it is more efficient to bring in a number of pages at one time
  - Ineffective if extra pages are not referenced
  - Should not be confused with "swapping"

Placement Policy

- Determines where in real memory a process piece is to reside
- Important design issue in a pure segmentation system (similar to dynamic partitioning, best-fit, first-fit, etc.)
- Paging or combined paging with segmentation placing is irrelevant because hardware performs functions with equal efficiency

Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page must be brought in
    - Objective is that the page that is removed be the page least likely to be referenced in the near future
- The more elaborate the replacement policy the greater the hardware and software overhead to implement it

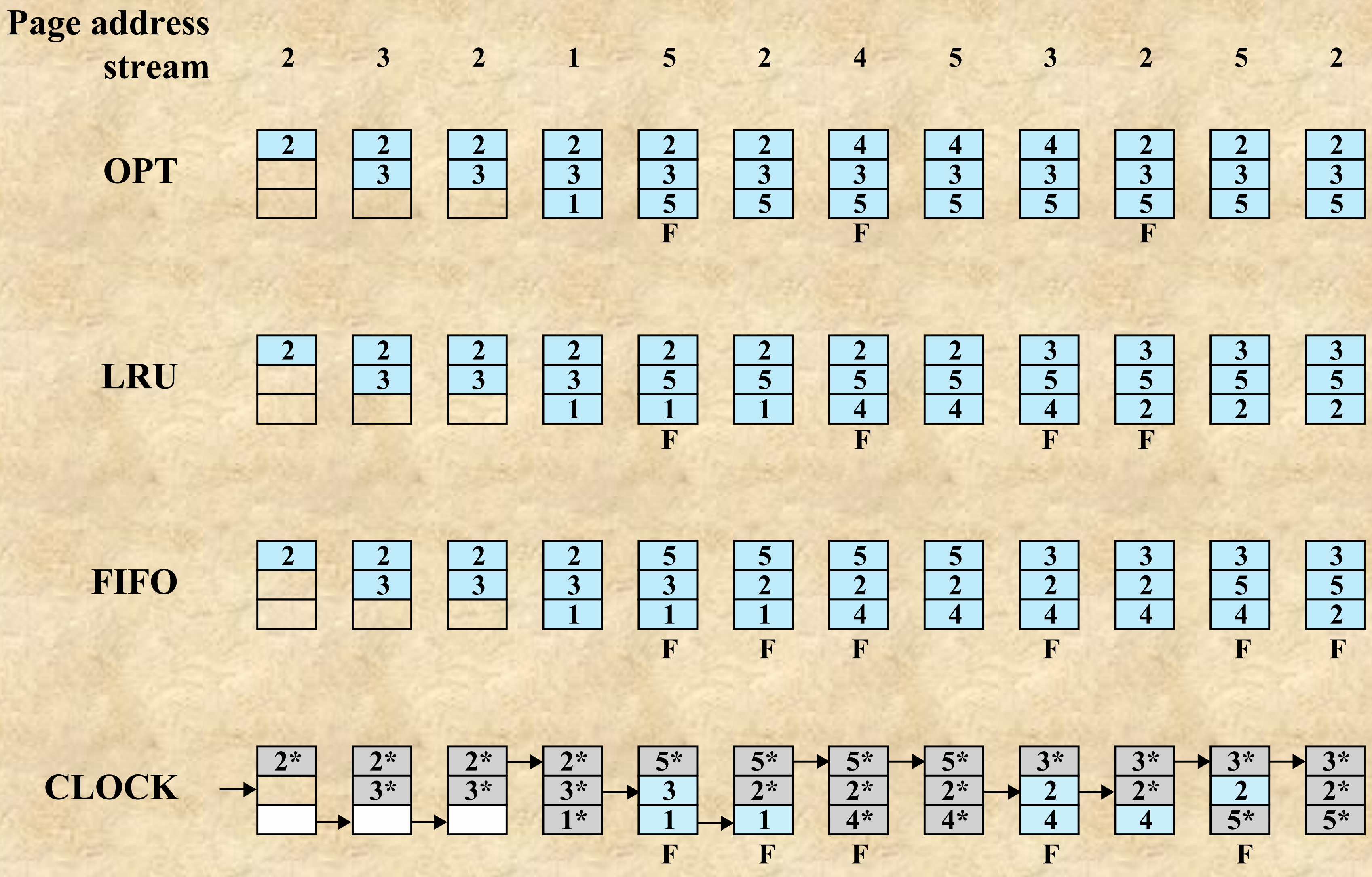
Frame Locking

- When a frame is locked the page currently stored in that frame may not be replaced
  - Kernel of the OS as well as key control structures are held in locked frames
  - I/O buffers and time-critical areas may be locked into main memory frames
  - Locking is achieved by associating a lock bit with each frame

Basic Algorithms

Algorithms used for the selection of a page to replace:

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock



Page address stream
2 3 2 1 5 2 4 5 3 2 5 2
OPT
2 | 2 3 | 2 3 | 2 3 1 | 2 3 5 F | 2 3 5 | 4 3 5 F | 4 3 5 | 4 3 5 | 2 3 5 F | 2 3 5 | 2 3 5
LRU
2 | 2 3 | 2 3 | 2 3 1 | 2 5 1 F | 2 5 1 | 2 5 4 F | 2 5 4 | 3 5 4 F | 3 5 2 F | 3 5 2 | 3 5 2
FIFO
2 | 2 3 | 2 3 | 2 3 1 | 5 3 1 F | 5 2 1 F | 5 2 4 F | 5 2 4 | 3 2 4 F | 3 2 4 | 3 5 4 F | 3 5 2 F
CLOCK
2* | 2* 3* | 2* 3* | 2* 3* 1* | 5* 3 1 F | 5* 2* 1 F | 5* 2* 4* F | 5* 2* 4* | 3* 2 4 F | 3* 2* 4 | 3* 2 5* F | 3* 2* 5*

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

Figure 8.14 Behavior of Four Page-Replacement Algorithms

Optimal

- The optimal policy selects for replacement that page for which the time to the next reference is the longest
  - Results in the fewest number of page faults
  - 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

Least Recently Used (LRU)

- Replaces the page 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
- Difficult to implement
  - One approach is to tag each page with the time of last reference
    - This requires a great deal of overhead

First-in-First-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - Simple replacement policy to implement
- Page that has been in memory the longest is replaced

Clock Policy

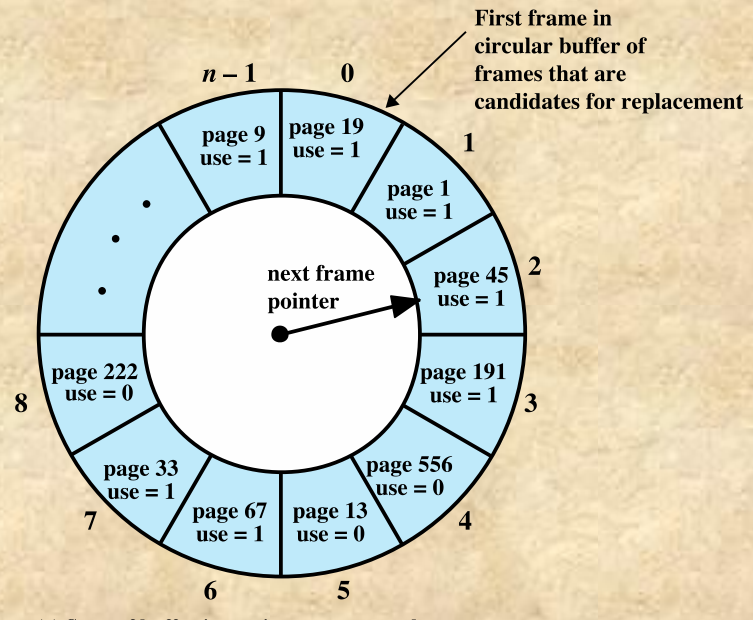
- Requires the association of an additional bit with each frame, referred to as the *use* bit
- When a page is first loaded in memory or subsequently referenced, the use bit is set to 1
- The set of frames is considered to be a circular buffer
- Any frame with a use bit of 1 is passed over by the algorithm
- Page frames visualized as laid out in a circle

Clock Policy

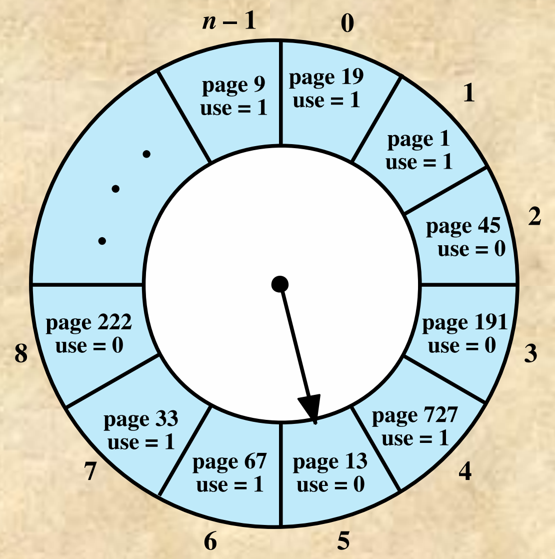
- When a page is replaced, the pointer is set to indicate the next frame in the buffer after the one just updated
- When it comes time to replace a page, the operating system scans the buffer to find a frame with a use bit set to 0
  - Each time it encounters a frame with a use bit of 1, it resets that bit to 0 and continues on
  - If any of the frames in the buffer have a use bit of 0 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 0, and stop at its original position, replacing the page in that frame

We can see that 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. The policy is referred to as a clock policy because we can visualize the page frames as laid out in a circle.

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



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



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

Figure 8.15 Example of Clock Policy Operation

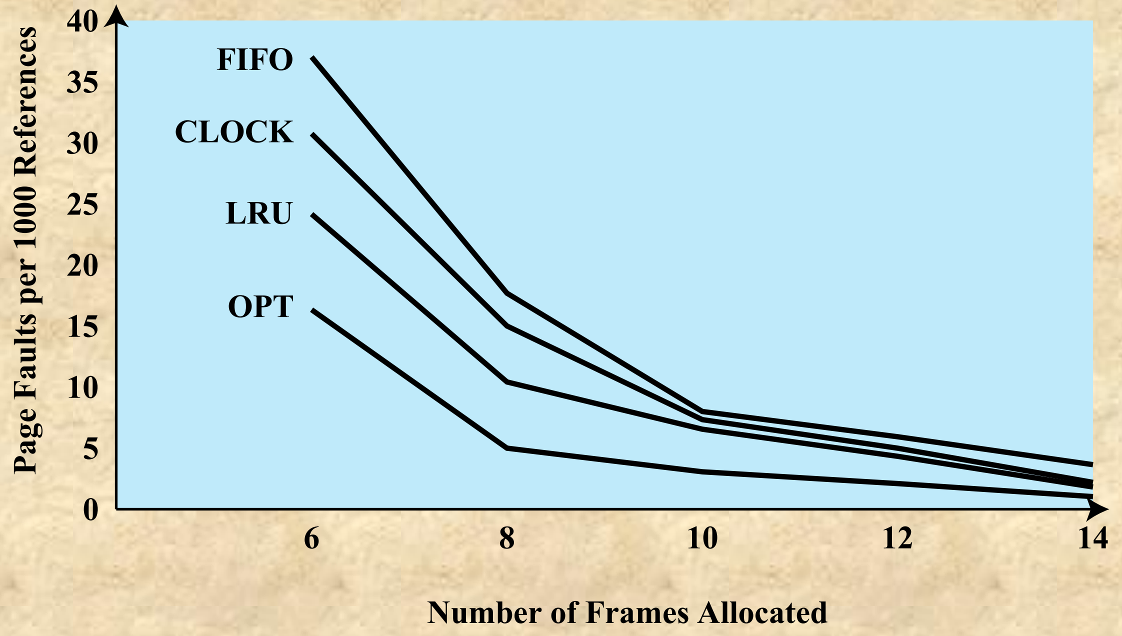


Figure 8.16 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

8.4 Consider the following string of page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Complete a figure similar to Figure 8.15, showing the frame allocation for:

- **a.** FIFO (first-in-first-out)
- **b.** LRU (least recently used)
- **c.** Clock
- **d.** Optimal (assume the page reference string continues with 1, 2, 0, 1, 7, 0, 1)
- **e.** List the total number of page faults and the miss rate for each policy. Count page faults only after all frames have been initialized.

Resident Set Management

- The OS must decide how many pages to bring into main memory
  - The smaller the amount of memory allocated to each process, the more processes can reside in memory
  - Small number of pages loaded increases page faults
  - Beyond a certain size, further allocations of pages will not effect the page fault rate

Resident Set Size

Fixed-allocation

- Gives a process a fixed number of frames in main memory within which to execute
  - When a page fault occurs, one of the pages of that process must be replaced

Variable-allocation

- Allows the number of page frames allocated to a process to be varied over the lifetime of the process

Replacement Scope

- The scope of a replacement strategy can be categorized as ***global*** or ***local***
  - Both types are activated by a page fault when there are no free page frames

Local

- Chooses only among the resident pages of the process that generated the page fault

Global

- Considers all unlocked pages in main memory

	Local Replacement	Global Replacement
Fixed Allocation	•Number of frames allocated to a process is fixed. •Page to be replaced is chosen from among the frames allocated to that process.	•Not possible.
Variable Allocation	•The number of frames allocated to a process may be changed from time to time to maintain the working set of the process. •Page to be replaced is chosen from among the frames allocated to that process.	•Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Table 8.5 Resident Set Management

Fixed Allocation, Local Scope

- Necessary to decide ahead of time the amount of allocation to give a process
- If allocation is too small, there will be a high page fault rate

If allocation is too large, there will be too few programs in main memory

- Increased processor idle time
- Increased time spent in swapping

Variable Allocation, Global Scope

- Easiest to implement and has been adopted in a number of operating systems
  - OS maintains a list of free frames
  - Free frame is added to resident set of process when a page fault occurs
  - If no frames are available, the OS must choose a page currently in memory

Variable Allocation, Local Scope

- When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set
- When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault
- Reevaluate the allocation provided to the process and increase or decrease it to improve overall performance
- Decision to increase or decrease a resident set size is based on the assessment of the likely future demands of active processes

Cleaning Policy

- Concerned with determining when a modified page should be written out to secondary memory

Demand Cleaning

A page is written out to secondary memory only when it has been selected for replacement

Precleaning

Writes modified pages before their page frames are needed.

Allows the writing of pages in batches

Load Control

- Determines the number of processes that will be resident in main memory
  - *Multiprogramming* level
- Critical in effective memory management
- Too few processes, many occasions when all processes will be blocked, and much time will be spent in swapping
- Too many processes will lead to thrashing

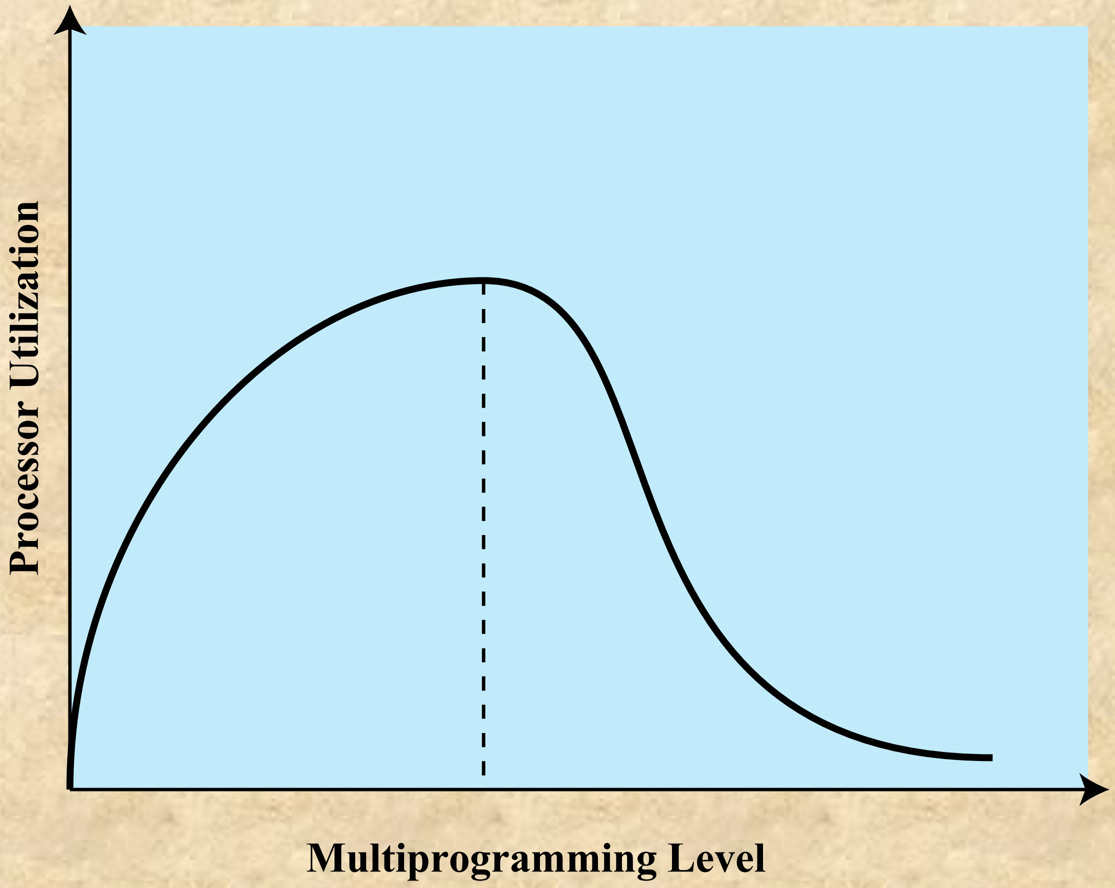


Figure 8.19 Multiprogramming Effects

Process Suspension

- If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be swapped out

Six possibilities exist:

- Lowest-priority process
- Faulting process
- Last process activated
- Process with the smallest resident set
- Largest process
- Process with the largest remaining execution window