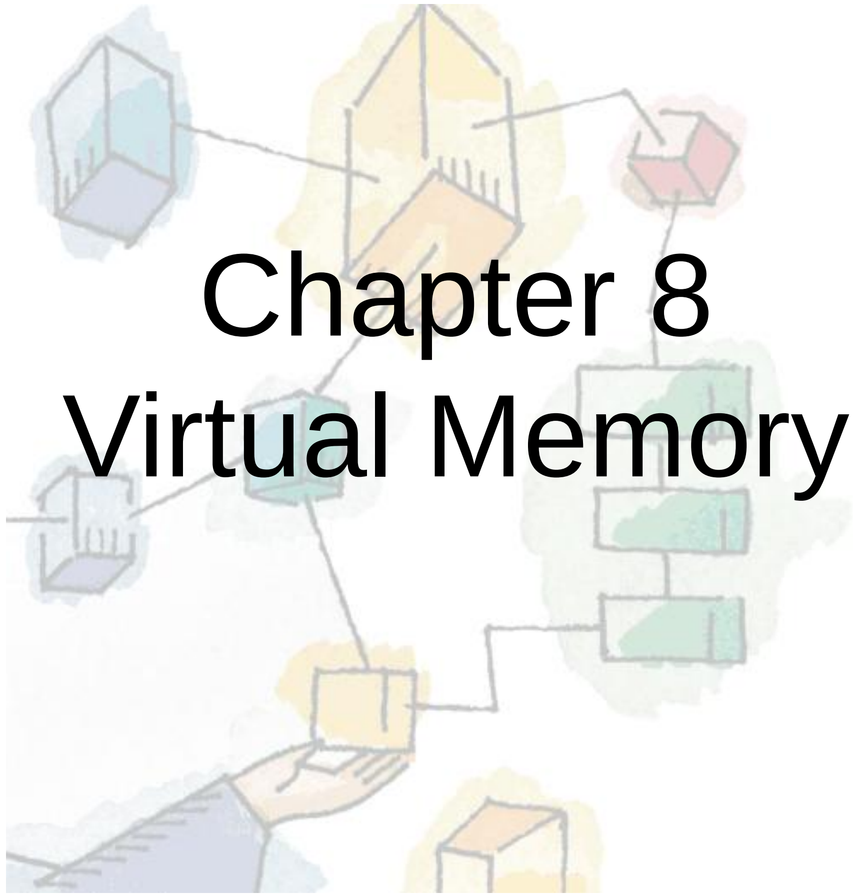


Chapter 8
Virtual Memory

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.

Key points in Memory Management

1. Memory references in a process are logical addresses, dynamically translated into physical addresses at run time
   – A process may be swapped in and out of main memory occupying different regions at different times during execution

2. A process may be broken up into pieces that do not need to located contiguously in main memory

Breakthrough in Memory Management

- **If both** of those 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 next instruction, and the next data location are in memory then execution can proceed
  - at least for some time

Execution of a Process

- Operating system brings into main memory only one or a few pieces of the process
  - Including initial program and data pieces

- **Resident set**
  - Portion of process that is actually in main memory at a given time

Execution of a Process

- As the process executes, it runs smoothly as long as all memory references are within resident set
  - Segment / Page Table is used to translate logical address to physical address

- If CPU encounters a logical address that is not present in main memory,
  - An interrupt is generated
- Current process is put into blocking state and OS gains control

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 complete which causes the operating system to place the affected process in the Ready state

Implications of this new strategy

- More processes may be maintained in main memory
  - Only load in some of the pieces of each process
  - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory

Real and Virtual Memory

- **Real memory**
  - Main memory, the actual RAM
  - Process executes only in main memory
- **Virtual memory**
  - Memory on disk
  - Allows for effective multiprogramming
  - Relieves the user of tight constraints of main memory

Locality and Virtual Memory

- Consider a very large process, with long program and large amount of data

- During a small period of time, a small section of program and some set of data are accessed
- Hence it is wasteful to load entire process when only some part is going to be used before the process is swapped out
- When branching occurs, fault is triggered

Locality and Virtual Memory

- **Advantages of given approach:**
  - More processes maintained in main memory
  - Loading a few pieces will save time

- **Problem??**
  - Thrashing
    - Throw out the piece just before it is required
    - Hence, which pages should be replaced is a concern

Thrashing

- A state in which the system spends most of its time swapping pieces rather than executing instructions.

- To avoid this, the operating system tries to guess which pieces are least likely to be used in the near future.
    - The guess is based on recent history

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

Support Needed for Virtual Memory

- Hardware must support paging and segmentation

- Operating system must be able to manage the movement of pages and/or segments between secondary memory and main memory

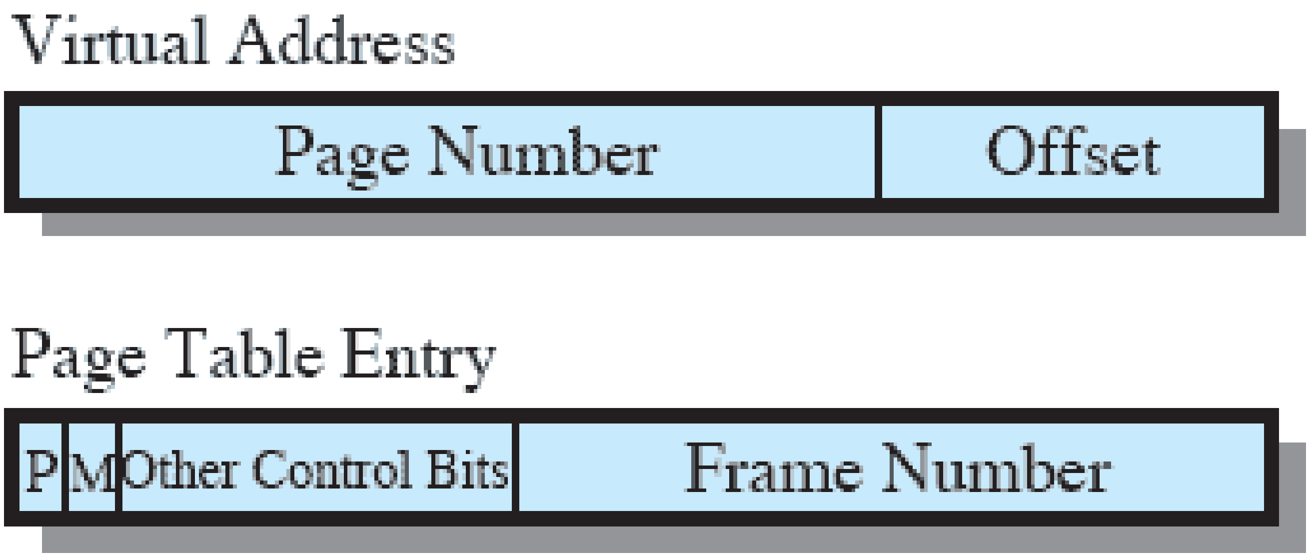
Paging for VM

- Each process has its own page table

- Each page table entry contains the frame number of the corresponding page in main memory

- Two extra bits are needed to indicate:
  - Whether the page is in main memory or not
  - Whether the contents of the page has been altered since it was last loaded

Paging Table



(a) Paging only

Control Bits include bits for protection and sharing

Address Translation

- Page table is usually large in size, hence it can't be kept in registers, so it is kept in main memory

- **Steps:**
  - A register holds starting address of page table for running process
  - Page number field of virtual address is used as an index to find frame number from process page table
  - Frame number is combined with offset to get the real address

Address Translation

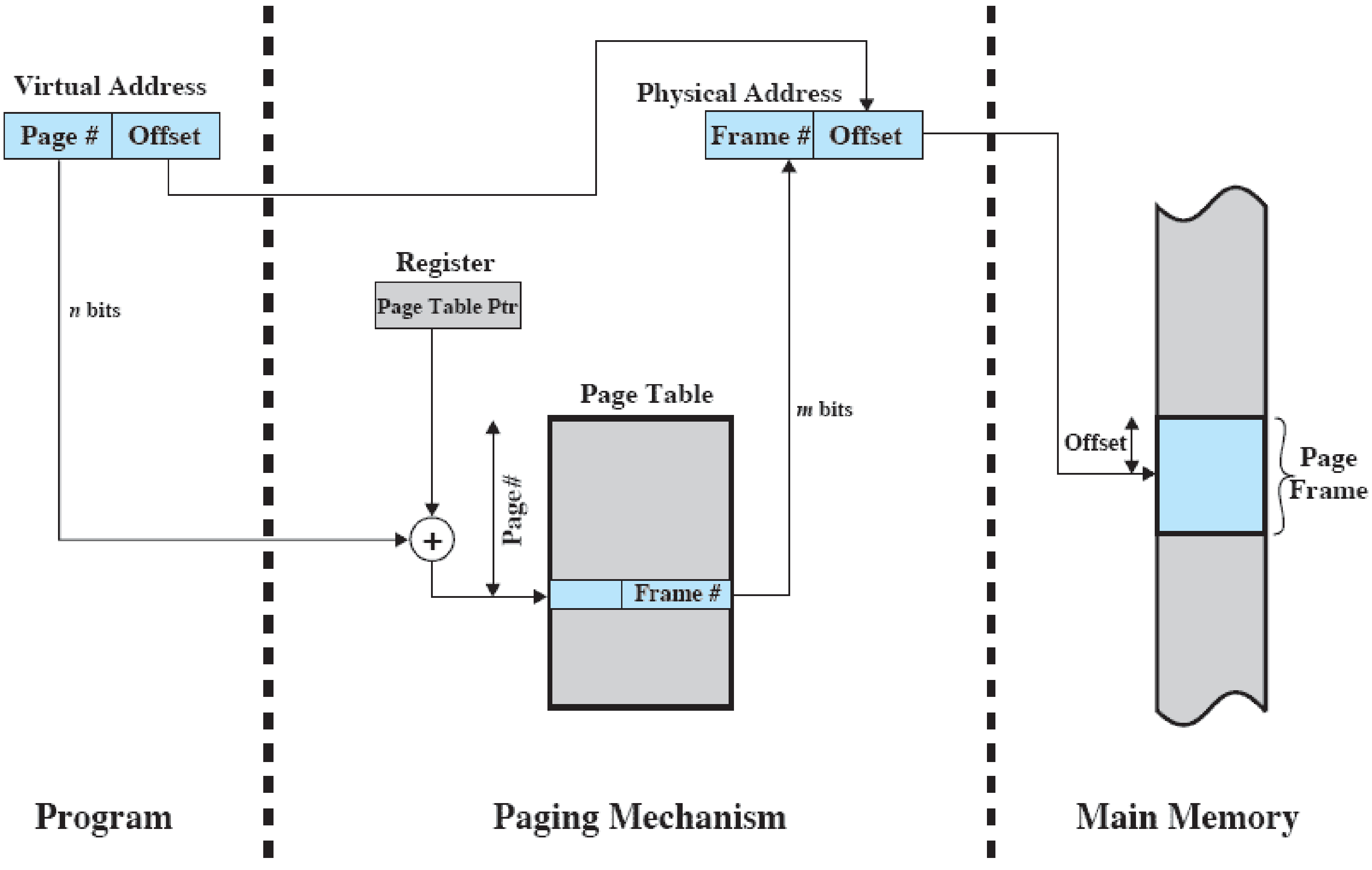


Figure 8.3 Address Translation in a Paging System

Large Page Tables

- A process can occupy huge amount of virtual memory
- E.g. Consider an architecture with 2 GB virtual memory per process, with page size of 512 bytes
  - Page table entries: 2^22 (per process!!)
- Hence page tables are also stored in virtual memory
- When a process is running, part of its page table is in main memory

Two level Scheme

- To maintain page tables properly, a page directory kind of structure can be used

- Page directory can contain X records
  - Each record points to a page table

- Length of page directory : X
- Length of page table: Y

- A process can contain X * Y pages

Two-Level Hierarchical Page Table

- Consider two level example with a 32 bit address: 4 GB address space
- Page size : 4 KB
- Number of PTE: 2^20
- Each entry requires 4 bytes for storage
  - Total space needed: 4 MB (for Page Table!)
- The address can be divided as per following:
  - 10+10+12

Two-Level Hierarchical Page Table

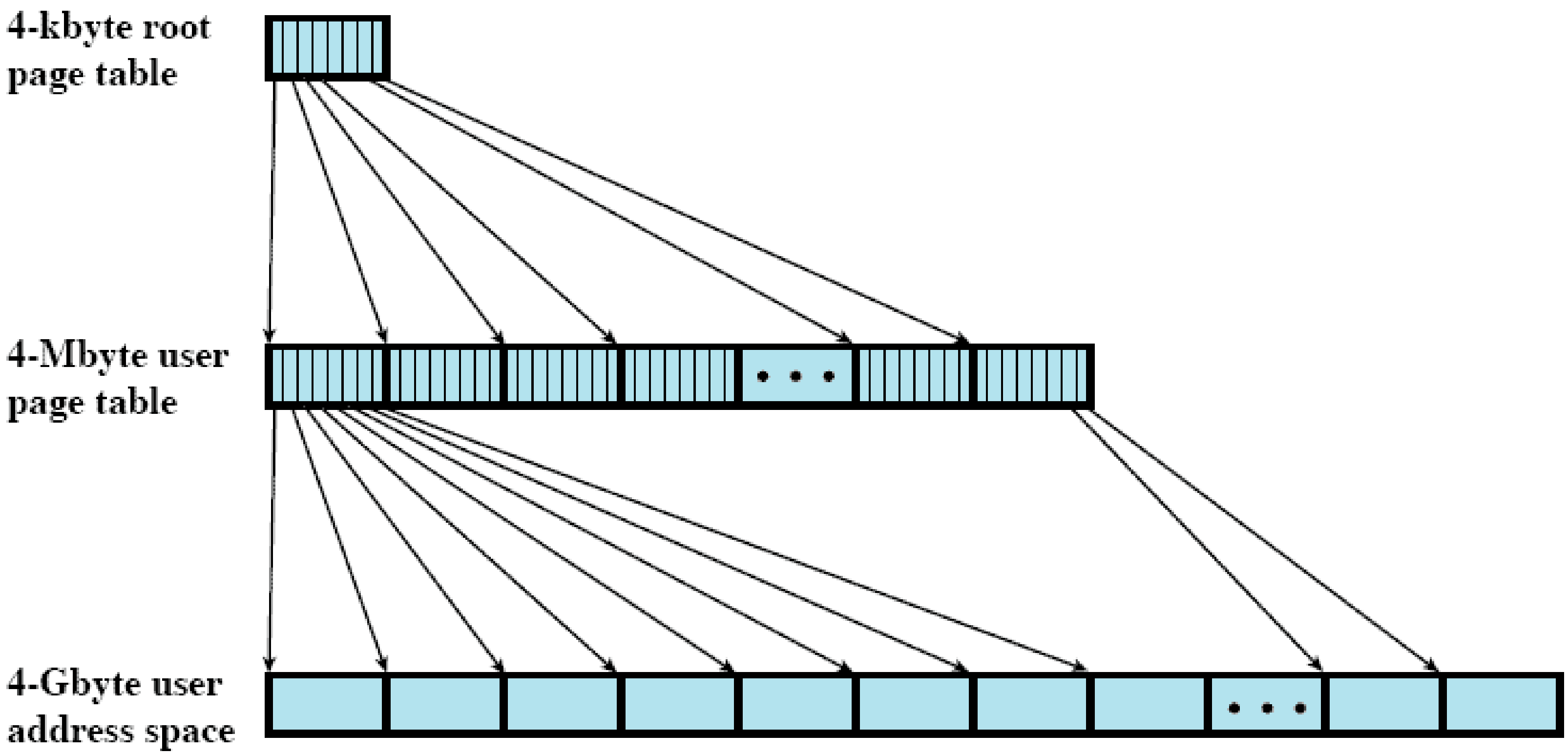


Figure 8.4 A Two-Level Hierarchical Page Table

Address Translation for Hierarchical page table

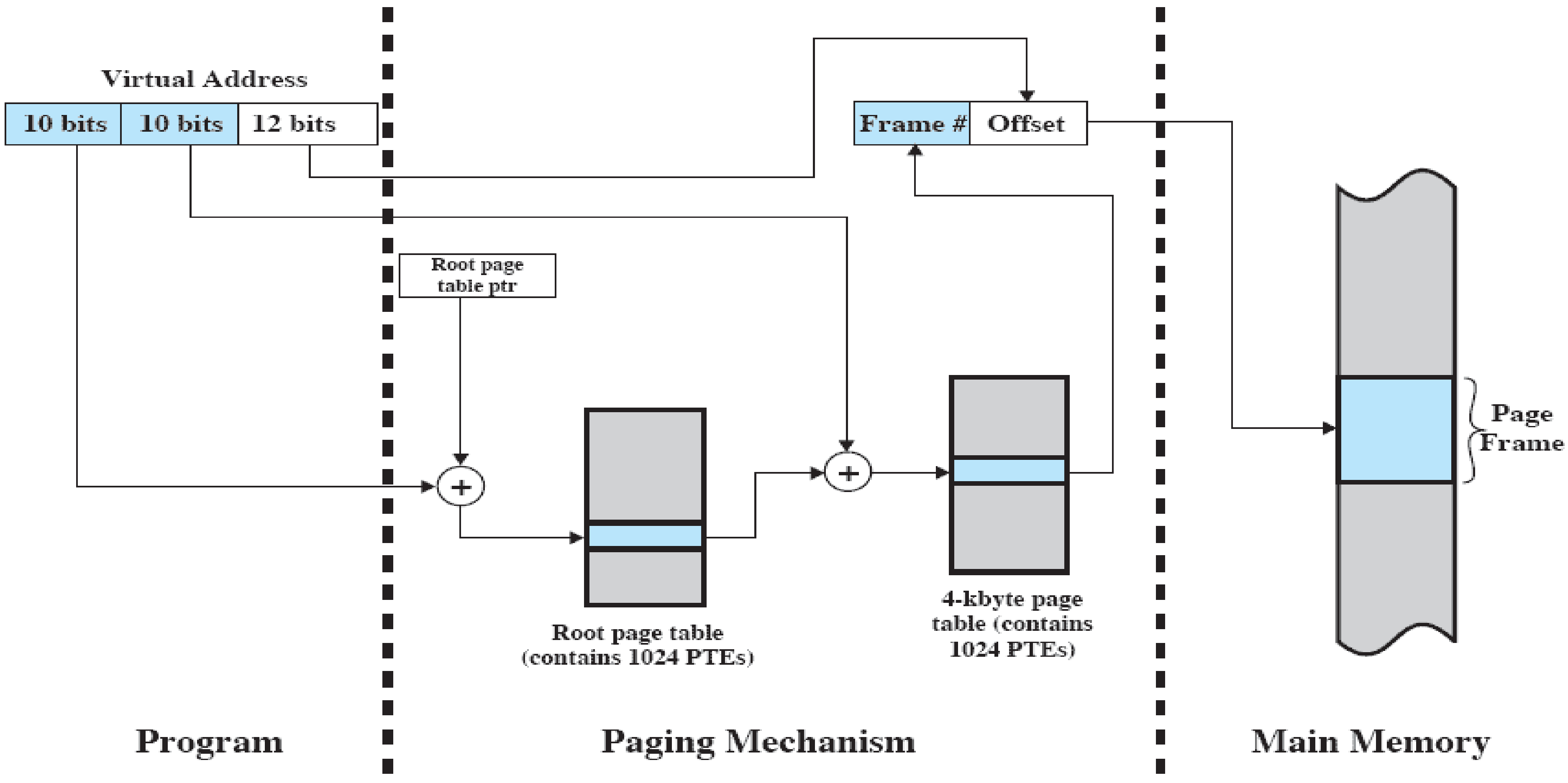


Figure 8.5 Address Translation in a Two-Level Paging System

Page tables grow proportionally

- A drawback of the type of page tables just discussed is that
  - Their size is proportional to that of the virtual address space

- An alternative is Inverted Page Tables

Inverted Page Table

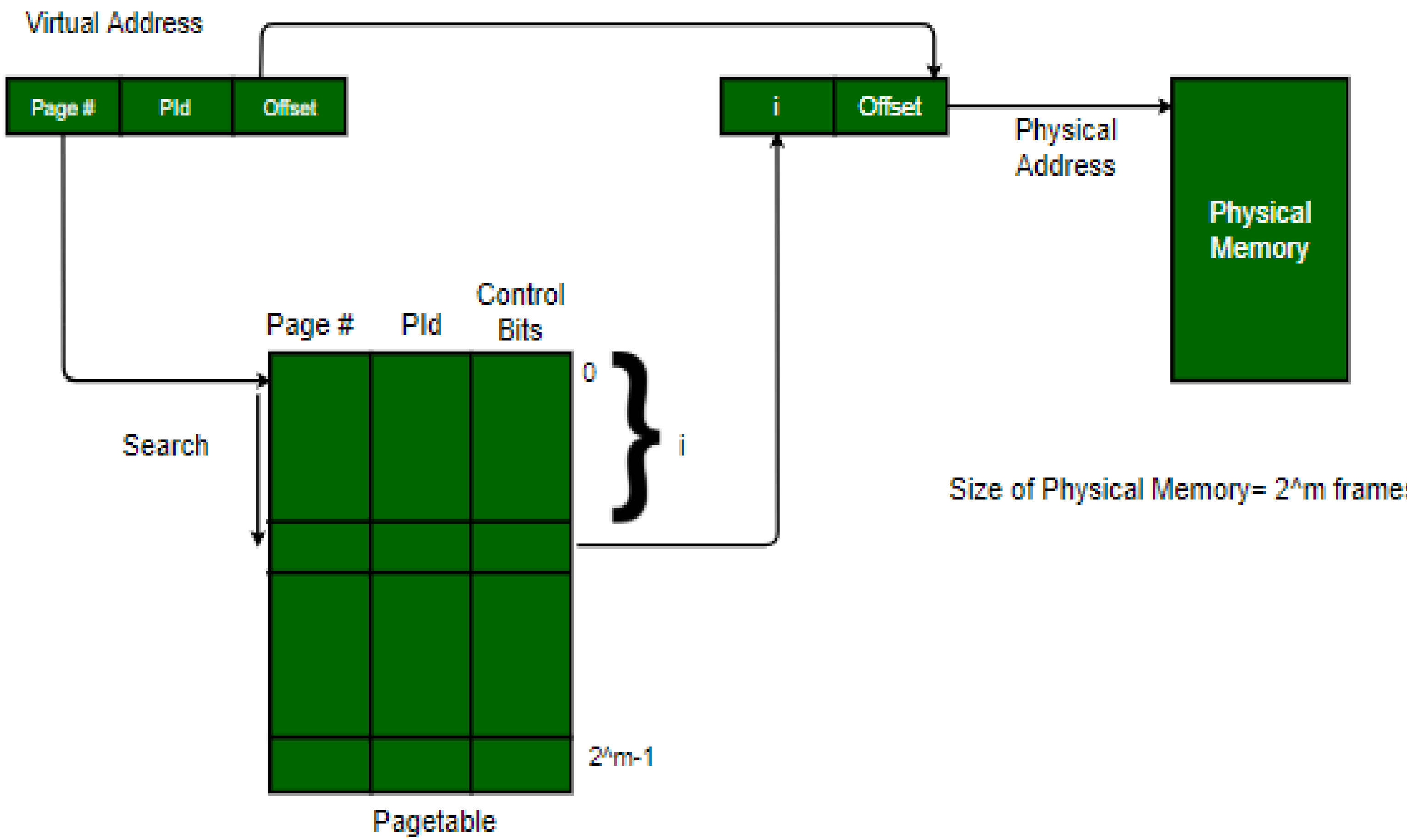
- One page table entry for every frame in main memory

- Hence number of page table entries in inverted page table are fixed
  - Equal to number of frames

- Single table is used to store information for all processes

- **Why called as Inverted?**
  - As the indexing is done based on frame number

Inverted Page Table



Inverted PageTable

Inverted Page Table - Hashing

- Used on PowerPC, UltraSPARC, and IA-64 architecture
- Page number portion of a virtual address is mapped into a hash value
- Hash value points to inverted page table
- Fixed proportion of real memory is required for the tables regardless of the number of processes

Inverted Page Table

Each entry in the page table includes:

- Page number
- Process identifier
  - The process that owns this page.
- Control bits
  - includes flags
- Chain pointer
  - the index value of the next entry in the chain

Inverted Page Table

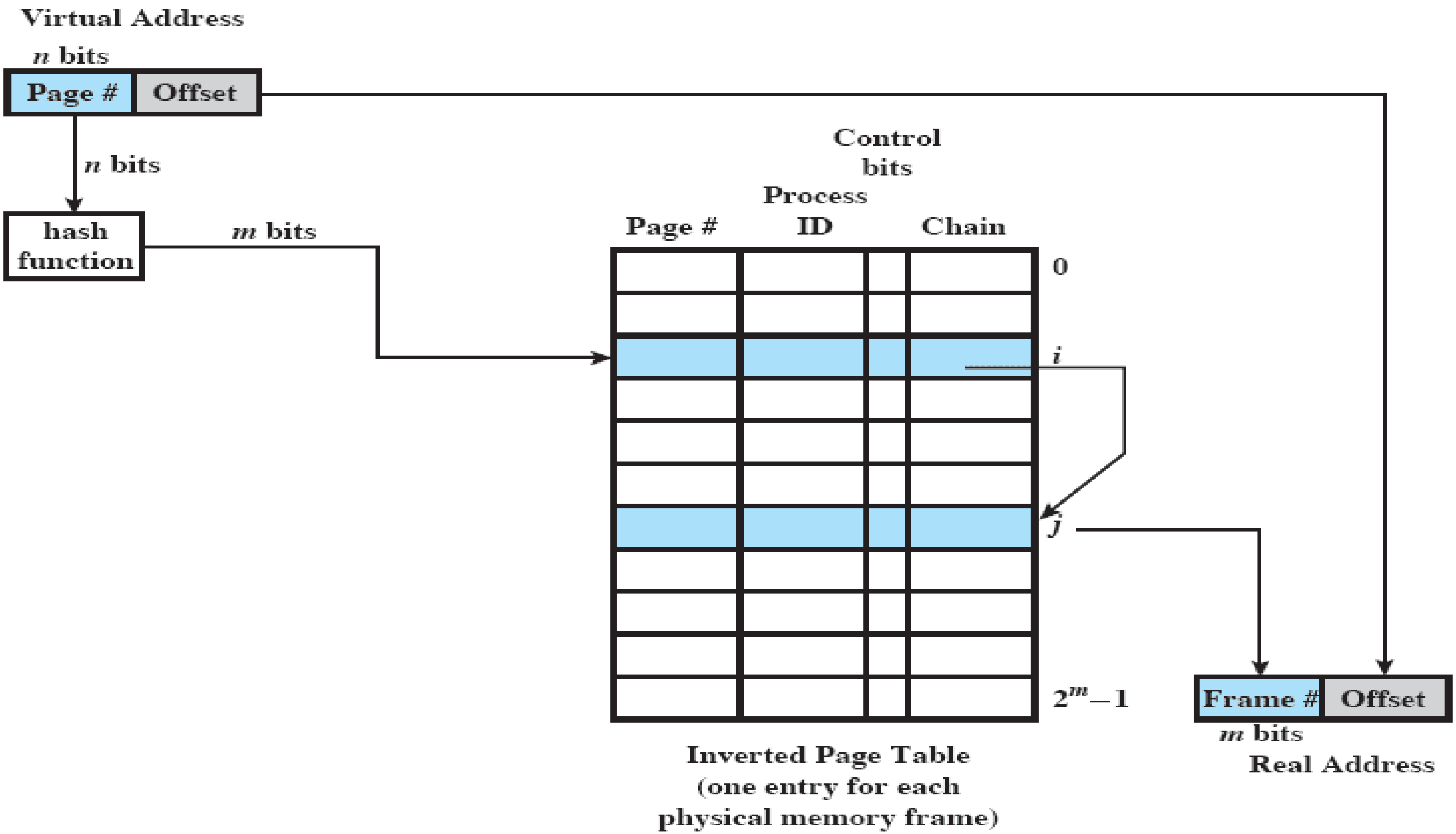


Figure 8.6 Inverted Page Table Structure

Translation Look aside Buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses
  - One to fetch the page table entry
  - One to fetch the desired data
- To overcome this problem a high-speed cache is set up for page table entries
  - Called a Translation Lookaside Buffer (TLB)
  - Contains page table entries that have been most recently used

TLB Operation

- Given a virtual address,
  - processor examines the TLB
- If page table entry is present (TLB hit),
  - the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss),
  - the page number is used to index the process page table

TLB Operation

- First the check is made to see if page is already in main memory (present bit set)
  - If not in main memory a page fault is issued
  - The TLB is updated to include the new page entry

- TLB uses principle of locality

Translation Look aside Buffer

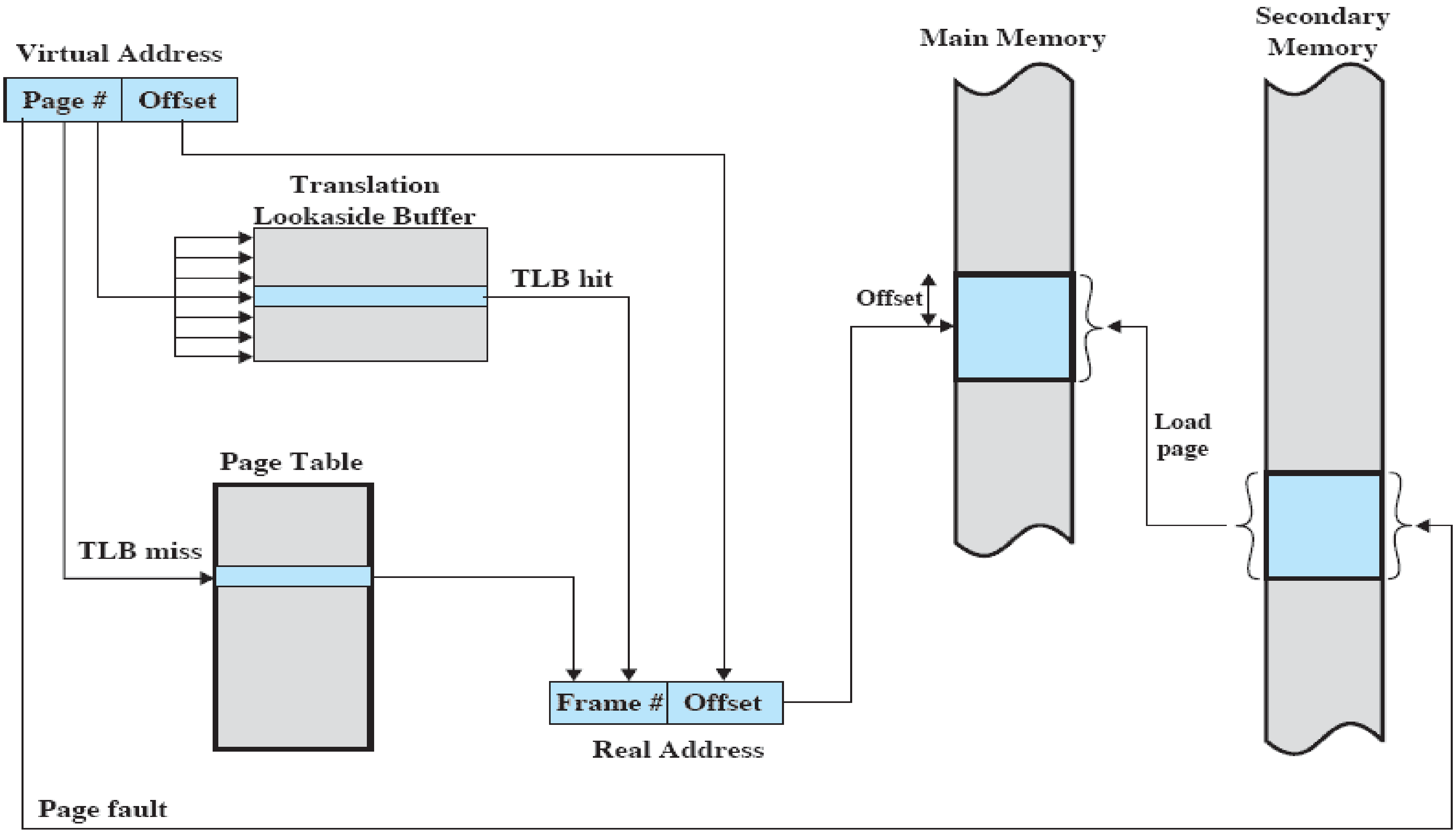


Figure 8.7 Use of a Translation Lookaside Buffer

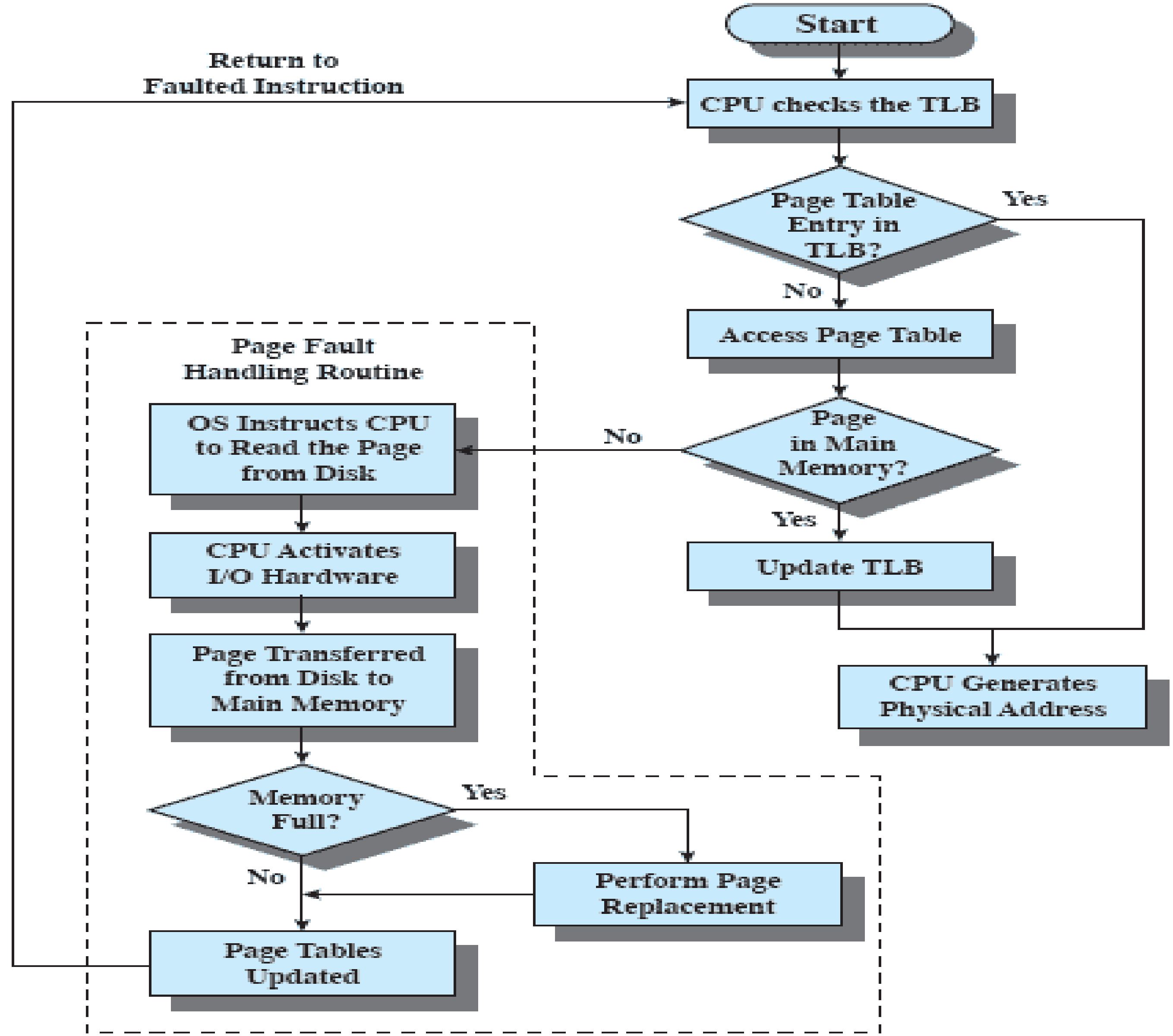


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

Associative Mapping

- As the TLB only contains some of the page table entries we cannot simply index into the TLB based on the page number
  - Each TLB entry must include the page number as well as the complete page table entry

- The process is able to simultaneously query numerous TLB entries to determine if there is a page number match

Translation Look aside Buffer

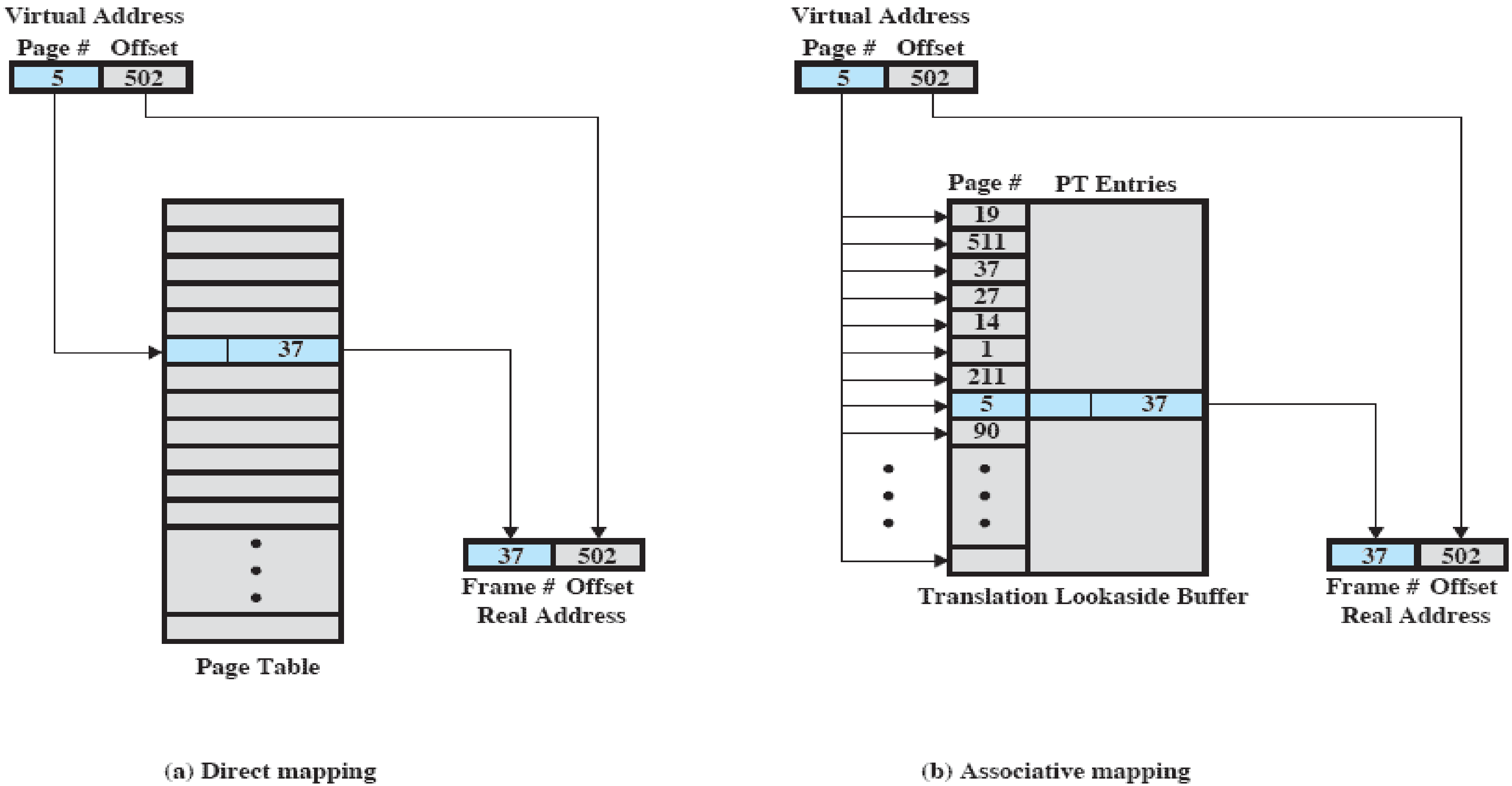


Figure 8.9 Direct Versus Associative Lookup for Page Table Entries

TLB and Cache Operation

- *If the requested block is present in main memory cache*
  - *It can be fetched quickly*

- If TLB gives a hit, real address is generated directly

- If TLB gives a miss and present bit is set, address is generated using page table

- Then a check is done with the MM cache
  - If match found then record is fetched directly

TLB and Cache Operation

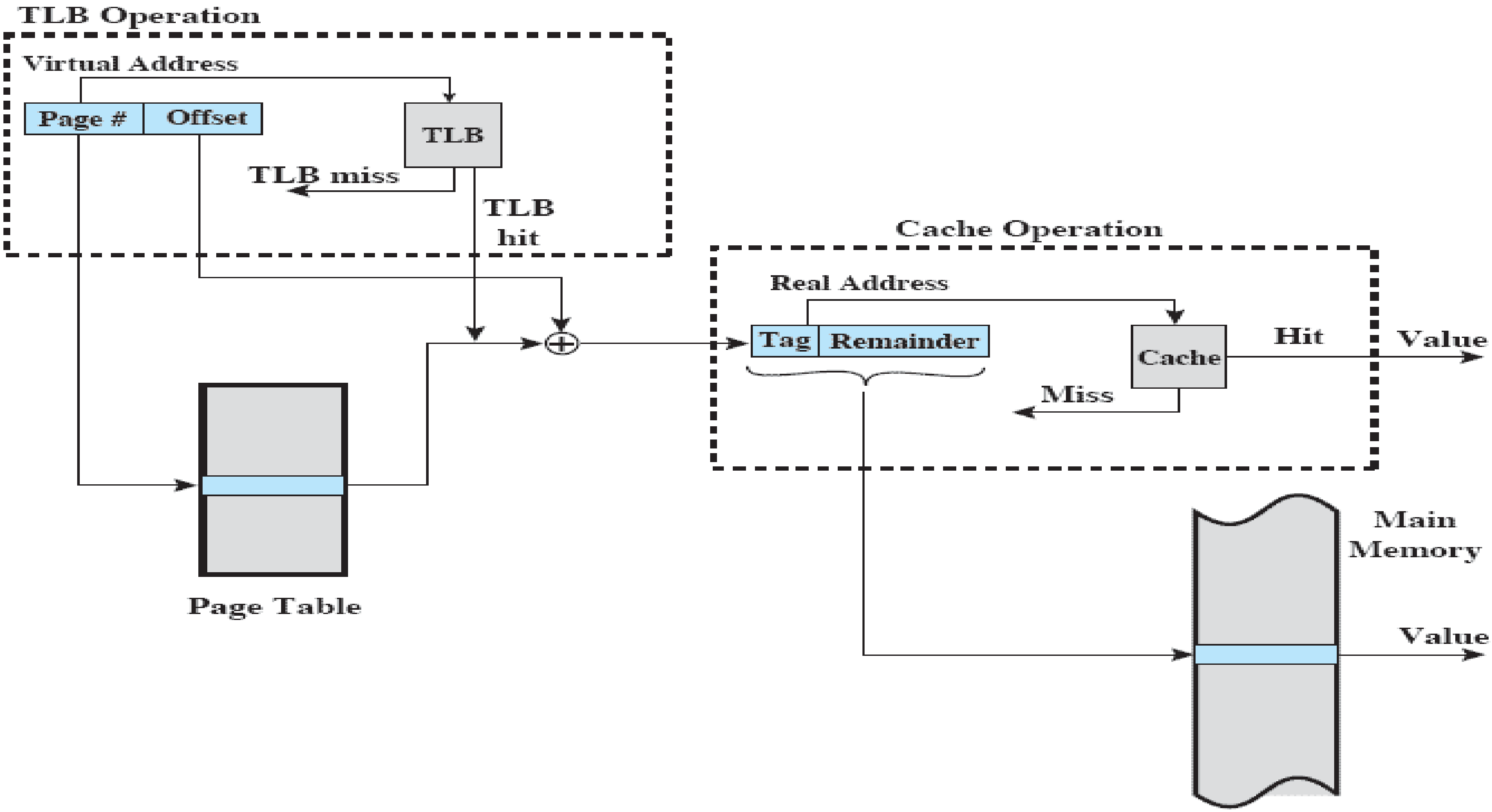


Figure 8.10 Translation Lookaside Buffer and Cache Operation

Spare A Thought!

- Think about the complexity of CPU hardware
- A PTE can be in TLB / MM / SM
- A Record can be in Cache / MM / SM

Page Size

- Decision on page size is very crucial
- **Smaller page size:**
  - Less amount of internal fragmentation
  - But more pages required per process
    - More pages per process means larger page tables
    - Larger page tables means large portion of page tables in virtual memory
- **Larger page size:**
  - Preferable for secondary memory (block data transfer can be used)

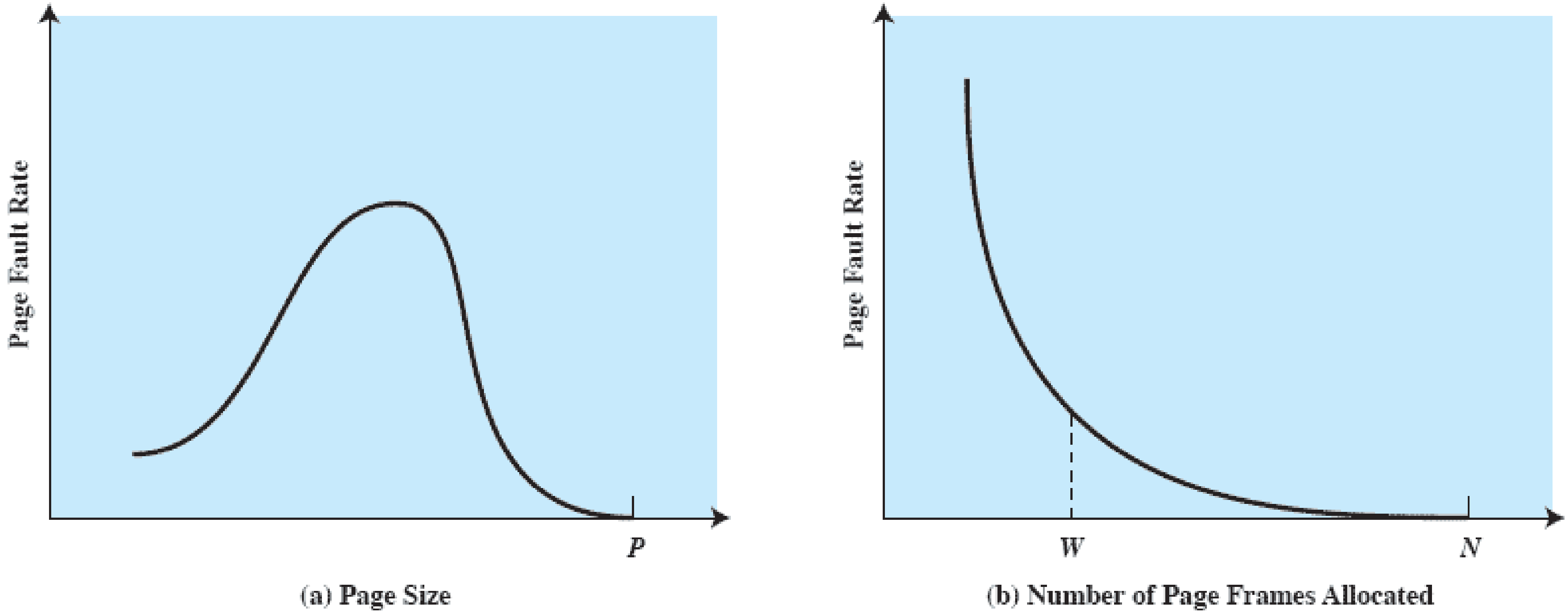
Further complications to Page Size

- Initially large number of pages are available for a process if page size is small
  - Page fault rate will be low
- As the page size is increased
  - Page fault rate will also increase
- When page size becomes equal to size of process
  - Page fault rate will be zero

Page Size

- Page fault rate is also determined by the number of frames allocated to a process
  - For a fixed page size, the fault rate drops as the number of pages maintained in main memory grows

Page Size



P = size of entire process
W = working set size
N = total number of pages in process

Figure 8.11 Typical Paging Behavior of a Program

Example Page Size

Table 8.3 Example Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit word
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbyes 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

Discussion on Page Size

- Page size design relates to
  - Physical memory size
  - Program size

- Main memory is getting larger, application size is also increasing
  - OOP encourages the use of many small modules and their references are scattered over large number of objects for a short period of time
  - Multithreaded applications may result in abrupt changes in instructions executions

Discussion on Page Size

- In both the cases mentioned in previous slide, POL gets weaker
  - Smaller TLB won't be suitable
  - If we opt for larger TLB
    - It will complicate the hardware
  - If we opt for large page size
    - Performance will not be good

Segmentation and VM

- Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments.
  - May be unequal, dynamic size
  - 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

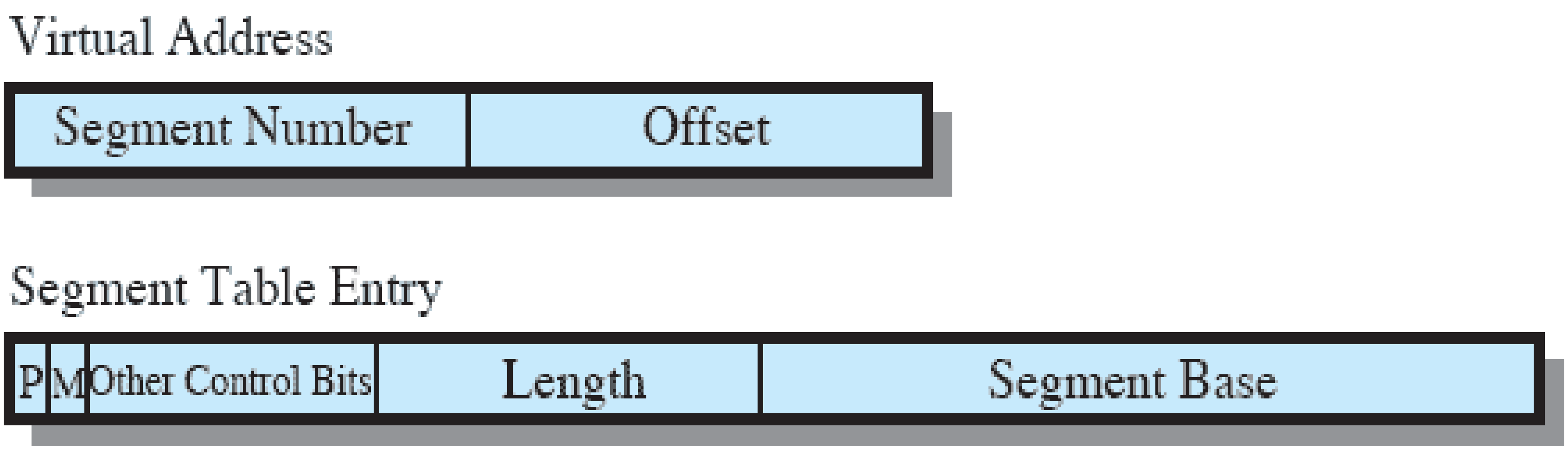
Segment Organization

- Segment table entry contains
  - Starting address corresponding segment in main memory
  - Length of the segment

- A bit is needed to determine if 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

Segment Table Entries



(b) Segmentation only

Address Translation in Segmentation

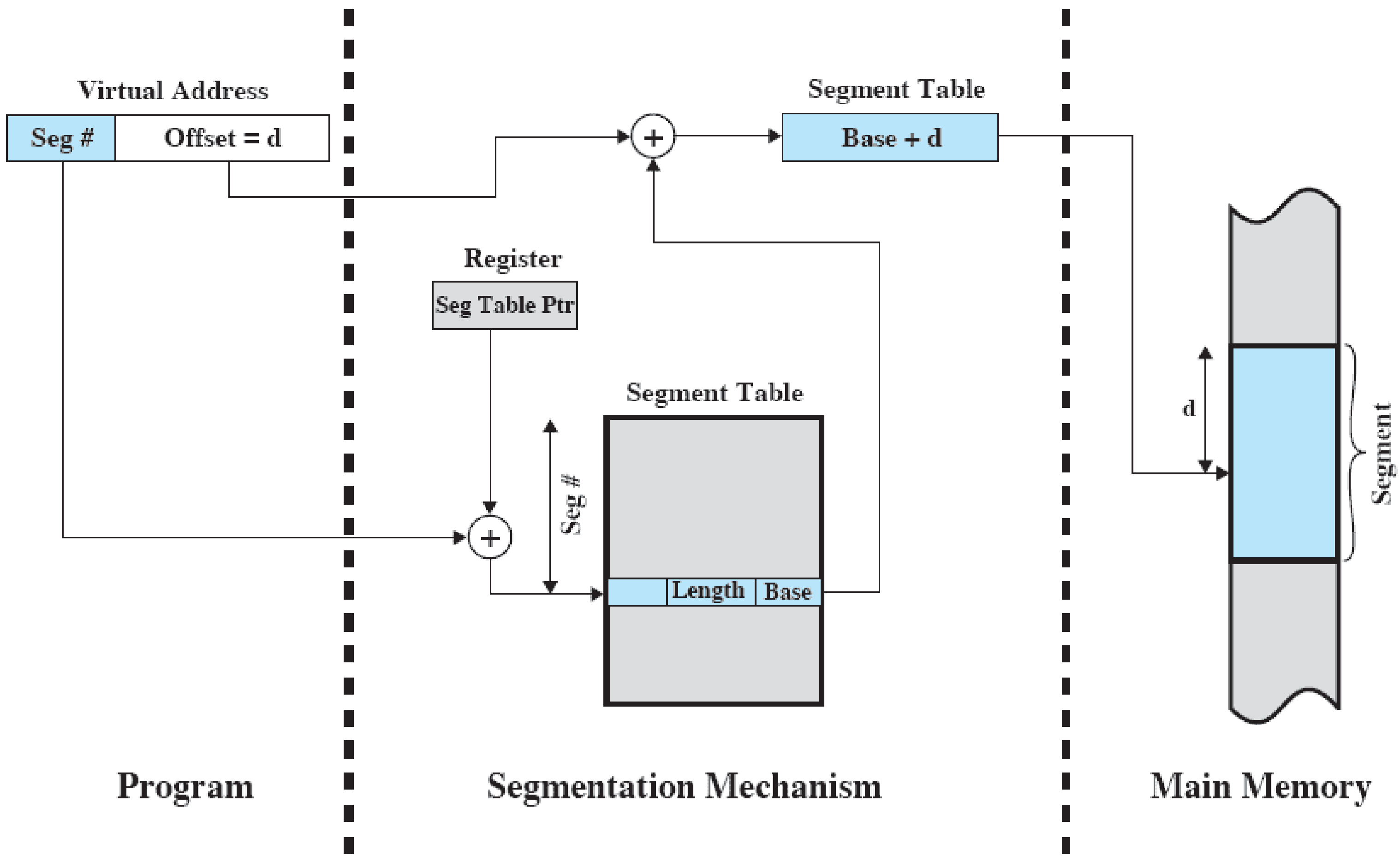


Figure 8.12 Address Translation in a Segmentation System

Combined Paging and Segmentation

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

- Address space is broken into number of segments
- Each segment is broken into fixed-size pages
- Each process has one segment table
- Each segment has a page table

Combined Paging and Segmentation

Virtual Address

Segment Number	Page Number	Offset

Segment Table Entry

Control Bits	Length	Segment Base

Page Table Entry

P	M	Other Control Bits	Frame Number

P= present bit

M = Modified bit

(c) Combined segmentation and paging

Address Translation

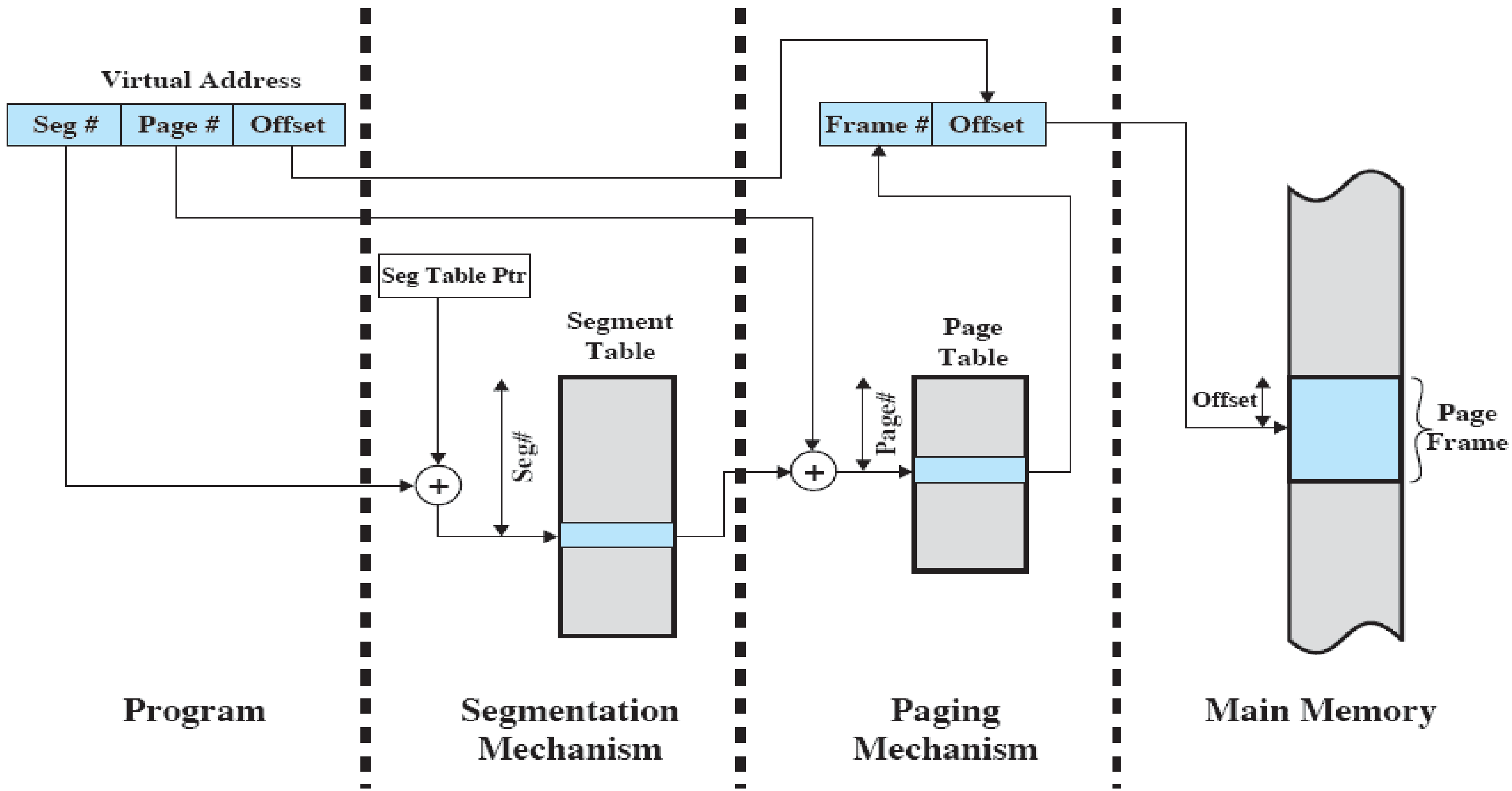


Figure 8.13 Address Translation in a Segmentation/Paging System

Protection and sharing

- Segmentation lends itself to the implementation of protection and sharing policies.
- As each entry has a base address and length, inadvertent memory access can be controlled
- For sharing, segments can be referenced by segment tables of multiple processes

Roadmap

- Hardware and Control Structures
- Operating System Software

Memory Management Decisions

- 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

Key Design Elements

Fetch Policy
- Demand
- 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

- Key aim: Minimise page faults
  - No definitive best policy

Fetch Policy

- Determines when a page should be brought into memory
- **Two main types:**
  - Demand Paging
  - Prepaging

Demand Paging and Prepaging

- **Demand paging**
  - Only brings pages into main memory when a reference is made to a location on the page
  - Many page faults when process first started

- **Prepaging**
  - Brings in more pages than needed
  - More efficient to bring in pages that reside contiguously on the disk
  - Don’t confuse with “swapping”

Placement Policy

- Determines where in real memory a process piece is to reside
- In **pure segmentation**
  - Best, first or next fit can be used
- **Paging** or **combined paging with segmentation**
  - Managed by hardware as placement is irrelevant

Replacement Policy

- When all of the frames in main memory are occupied,
  - and it is necessary to bring in a new page,
    - the replacement policy determines **which** page currently in memory is to be replaced.

But…

- **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

- **Three aspects** to be considered:
  - Number of frames to be allocated to each process
  - Candidate set of pages can be either from the process causing page fault **or** from all processes in main memory
  - Out of the selected set of pages, which page should be replaced

- First and second are part of resident set management and third one is in the scope of replacement policy

Replacement Policy: Frame Locking

- **Frame Locking**

  – If frame is locked, it may not be replaced
    - Kernel of the operating system
    - Key control structures
    - I/O buffers and other critical areas

  – Associate a lock bit with each frame

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

Example

- For example, Consider a page reference 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...

Optimal policy

- Selects for replacement that page for which the time to the next reference is the longest

- But Impossible to have perfect knowledge of future events

- Impossible to implement, used as a reference for comparison

Optimal Policy Example

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
					F		F			F		

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.

LRU Example

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		

First-in, first-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
  - Simplest replacement policy to implement

- Page that has been in memory the longest is replaced
  - But, this page may be needed again very soon if it hasn’t truly fallen out of use

FIFO Example

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F

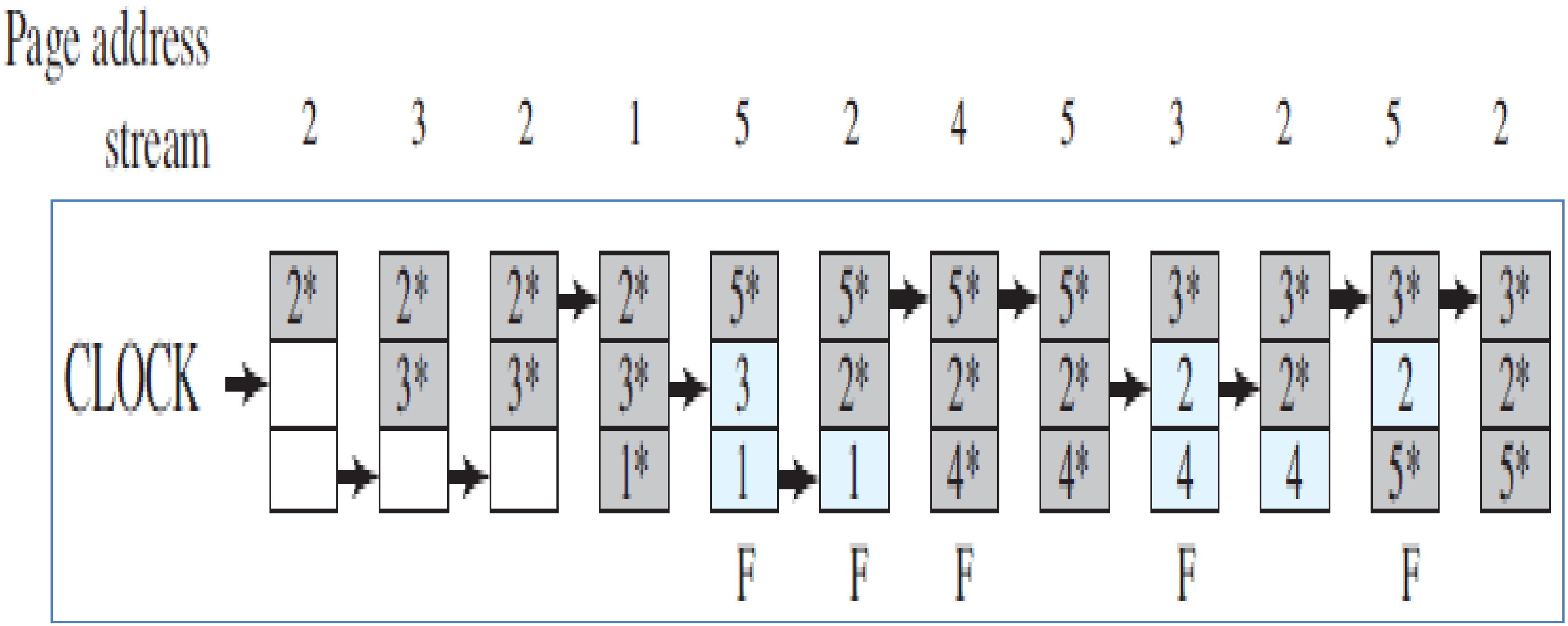
First-in, first-out (FIFO)

- 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.

Clock Policy

- Uses an additional bit called a "use bit"
- When a page is first loaded in memory or referenced, the use bit is set to 1
- When it is time to replace a page, the OS scans the set flipping all 1's to 0
- The first frame encountered with the use bit already set to 0 is replaced.

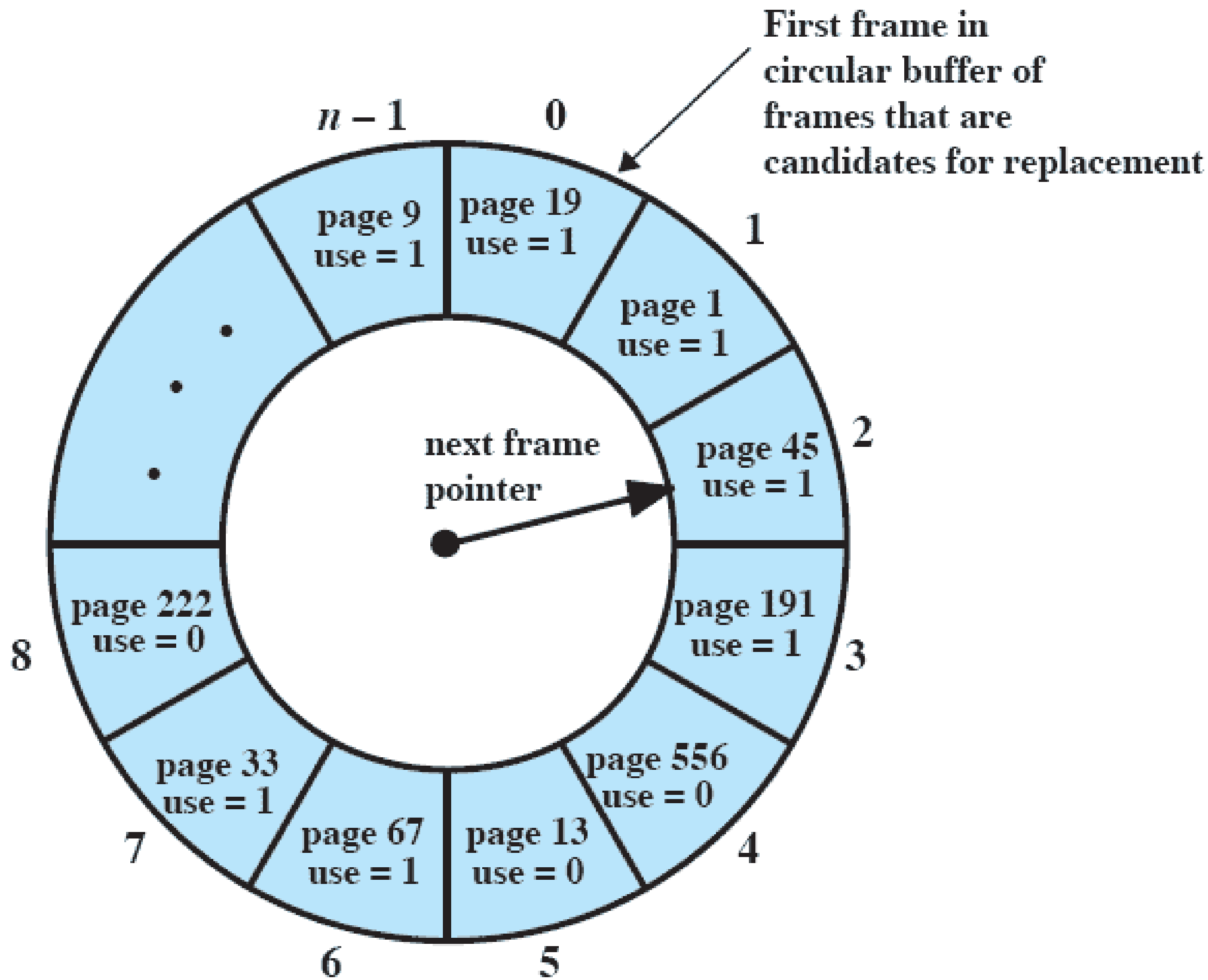
Clock Policy Example



Clock Policy Example

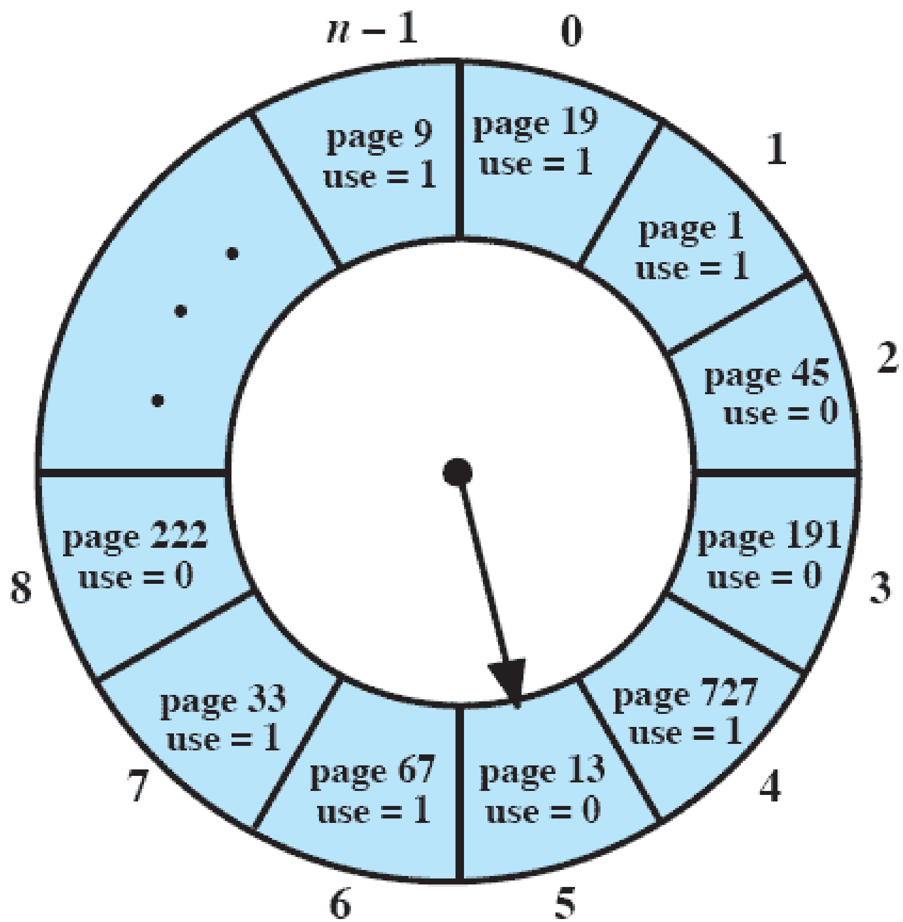
- Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

Clock Policy



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

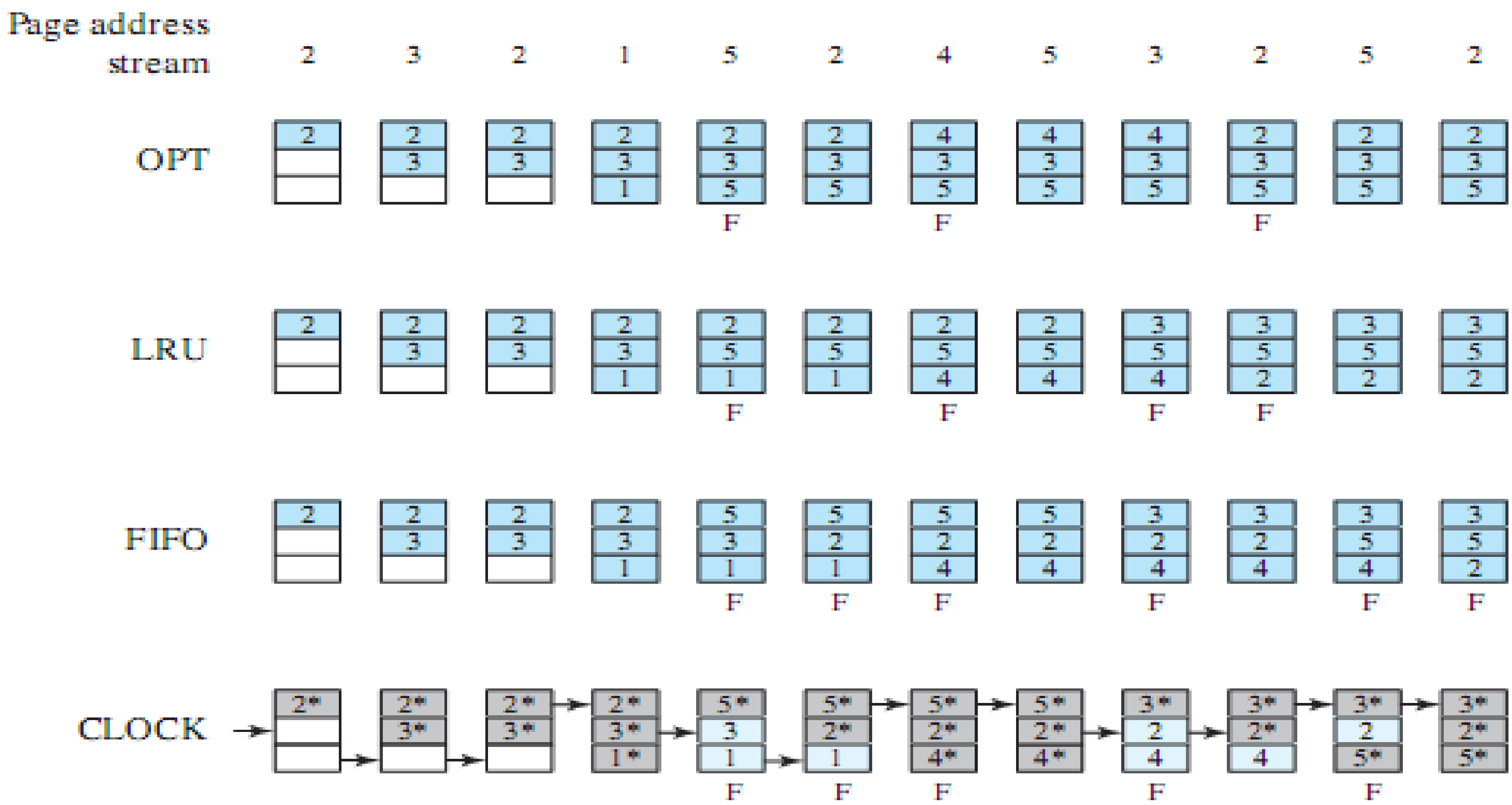
Clock Policy



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

Figure 8.16 Example of Clock Policy Operation

Combined Examples



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

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

Comparison

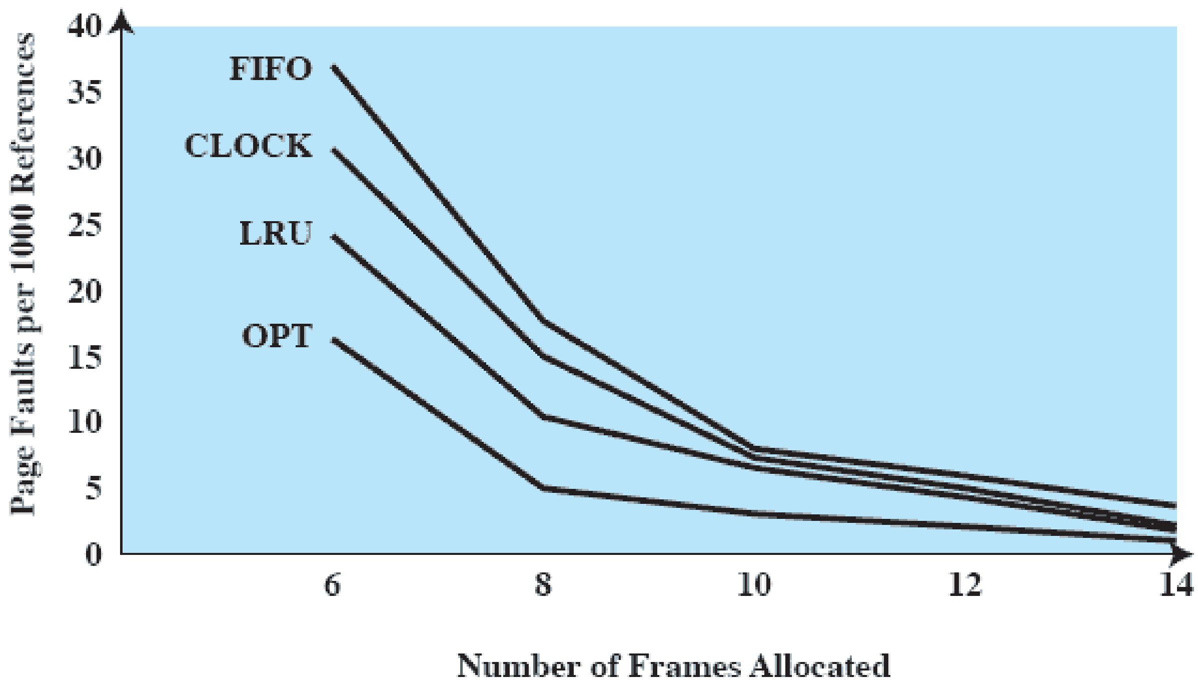


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Performance Improvement of Clock Algorithm

- Use bit and modify bit can have following combinations:
  - u=0, m=0
  - u=1, m=0
  - u=0, m=1
  - u=1, m=1

- Modified approach can use these scenarios for performance improvement

Performance Improvement of Clock Algorithm

- Scan the buffer from current position
  - Don’t modify use bit; look for frame with **u=0, m=0**

- If the step 1 scan fails, scan again
  - Look for a frame with **u=0, m=1**
  - Reset use bit for each bypassed frame

- If step 2 scan fails,
  - Repeat step 1 scan and if required repeat step 2 scan

Performance Improvement of Clock Algorithm

- Find a page that has not been accessed and modified recently
- If this fails, find the page that has been modified, but not accessed recently (POL)

- **Advantages:**
  - Unchanged pages given preference
  - Saves time

Page Buffering

- LRU and Clock policies both involve complexity and overhead
  - – Also, replacing a modified page is more costly than unmodified as it needs to be written to secondary memory

- **Solution: Page buffering**
- Replacement algorithm is simple FIFO
  - – But replaced pages are maintained in two types of lists, within memory

Page Buffering

- Free page list:
  - When a page is replaced and it is not modified, it is appended at tail of this list
- Modified page list:
  - When a page is replaced and it is modified, it is added at tail of this list
- Replaced pages are not removed from memory
  - Just their entries are removed from page table

Page Buffering

- **Advantages:**
  - If the process refers to a page just replaced, it can be fetched quickly
  - Modified pages can be written out in batches
  - **Free and modified page lists act as caches**

Resident Set Management

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

Resident Set Size

- **Fixed-allocation**
  - Gives a process a fixed number of pages within which to execute
  - Decided at load time based on process type

- **Variable-allocation**
  - Number of pages allocated to a process varies over the lifetime of the process
  - Process experiencing higher PFR can be given more frames and less frames for process with less PFR

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.
  - A local replacement policy chooses only among the resident pages of the process that generated the page fault
  - A global replacement policy considers all unlocked pages in main memory

Relationship between RSS and replacement scope

- Fixed allocation, local scope
- Fixed allocation, global scope
- Variable allocation, local scope
- Variable allocation, global scope

Fixed Allocation, Local Scope

- 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 or
  - Increased swapping

Variable Allocation, Global Scope

- Process with high PFR will grow in RSS
- Easiest to implement
  - Adopted by many operating systems

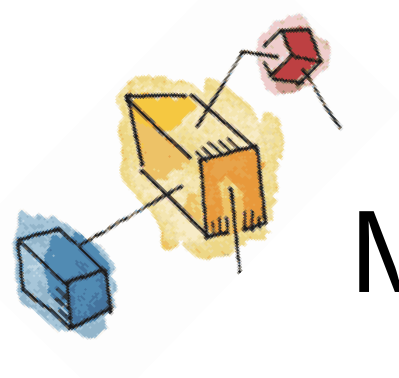
- Operating system keeps list of free frames
- Free frame is added to resident set of process when a page fault occurs
- If no free frame, replacement is performed
  - Therein lies the difficulty … which one to replace.

Variable Allocation, Local Scope

- When new process added, allocate number of page frames based on application type, program request, or other criteria

- When page fault occurs, select page from among the resident set of the process that suffers the fault

- Reevaluate allocation from time to time – makes this policy complex – but better in performance



Resident Set Management Summary

Table 8.5 Resident Set Management

	Local Replacement	Global Replacement
Fixed Allocation	• Number of frames allocated to 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.

Decision on resident set size

- One of the popular method for deciding resident set size is
  - Working set method

- **Working set:**
  - W(t, delta) is the set of pages of the process referenced in last delta time units at a specific time t

Working Set Method

- Consider page reference string as below:
  - At time = t1, 2,6,1,5,7,7,5
  - At time = t2, 3,1,2,5
  - W(t1,7) = {1,2,5,6,7}
  - W(t2,4) = {1,2,3,5}

Working Set Method

- Delta defines the window size
  - Large window size may result in larger working set
  - During a specific period of time, working set may remain same

- **Working set is a function of time**
  - If more pages are referenced during a period of time
    - Working set will grow in size

Working Set Method

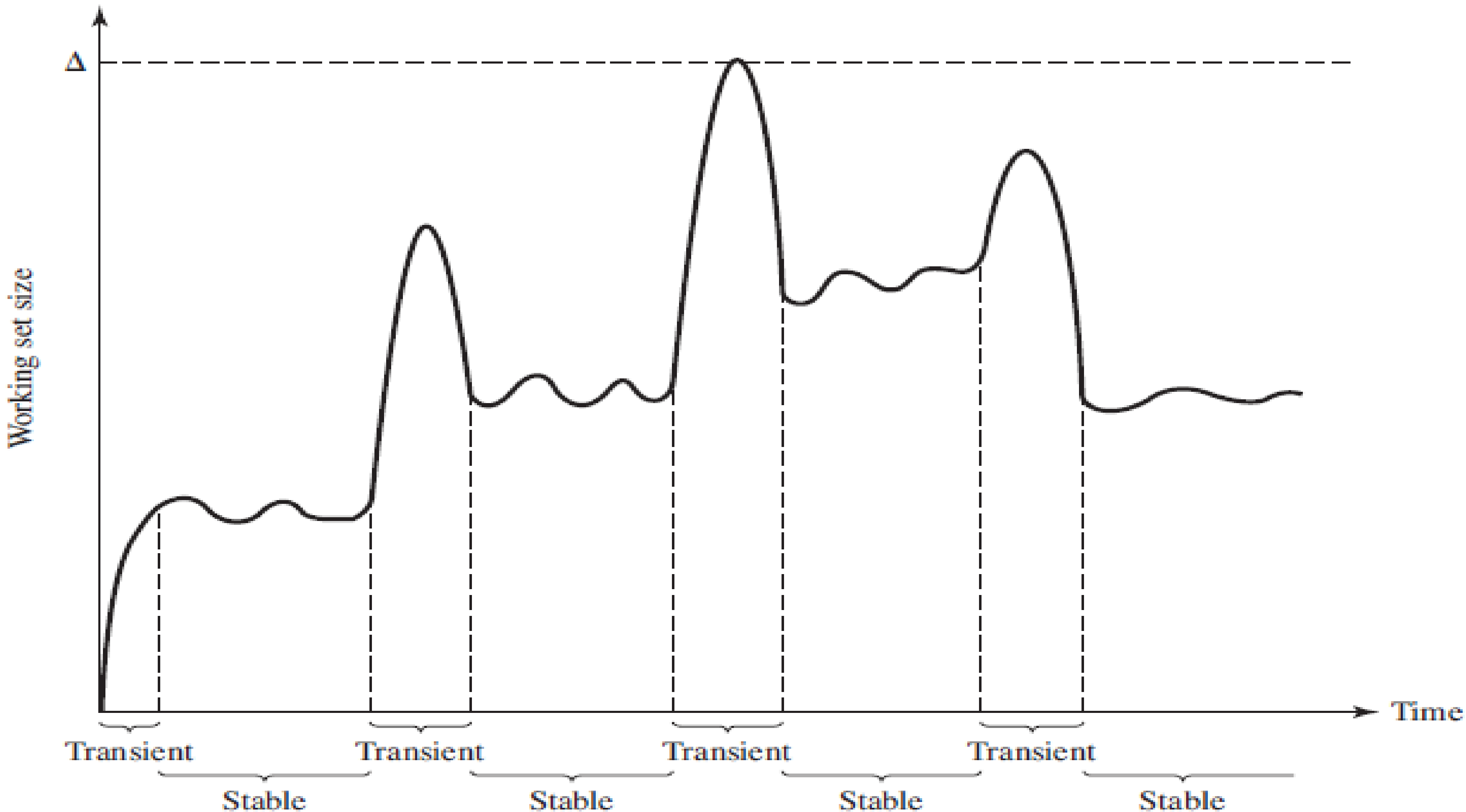


Figure 8.20 **Typical Graph of Working Set Size [MAEK87]**

Working Set Method

- Initially the number of pages referenced is high
- Process should stabilize later due to principle of locality
- But this locality will be shifted to a new locality after some time and working set size increases
- Again it decreases as only the pages of new locality are kept

Applying Working Set Method

- Following scheme can be used to apply working set method:
  - Monitor working set of each process
  - Periodically remove the pages which are not in working set

- A process can only be executed if its working set is present in main memory

Working Set Method: Issues

- Past does not always predict future
- Overhead involved in measurement of working set
- Finding optimal value of delta is difficult

Page Fault Frequency Approach

- Instead of focusing on the pages referenced
  - Focus on the page fault rate of the process
- If page fault rate is above the threshold
  - Resident set size should be increased
- If page fault rate is below the threshold
  - Resident set size should be decreased

Page Fault Frequency Approach

- **PFF Algorithm:**
  - Assign a use bit for each page in MM
  - Use bit is set to 1 when page is accessed
  - At the time of page fault, the time since last page is calculated (T)
    - If T < Threshold (PF occurred quickly)
      - Add a page to resident set of process
    - Else
      - Shrink the resident set of process
        - » Discard pages with use bit=0
        - » Reset use bits

Page Fault Frequency Approach

- **Problem:**
  - This approach does not work well during shift of locality
  - Solution?
    - VSWS (Variable interval Sampled Working Set)

Cleaning Policy

- A cleaning policy is concerned with determining when a modified page should be written out to secondary memory.

- **Demand cleaning**
  - A page is written out only when it has been selected for replacement
  - Two page transfers needed
    - Writing dirty page
    - Reading new page

Cleaning Policy

- **Pre-cleaning**
  - Pages are written out in batches before their frames are needed

Cleaning Policy

- Best approach uses page buffering
  - Decouples cleaning and replacement
- Replaced pages are placed in two lists
  - Modified and unmodified
- Pages in the modified list are periodically written out in batches
- Pages in the unmodified list are either reclaimed if referenced again or lost when its frame is assigned to another page

Load Control

- Determines the number of processes that will be resident in main memory
  - The *multiprogramming* level
- 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

Multiprogramming

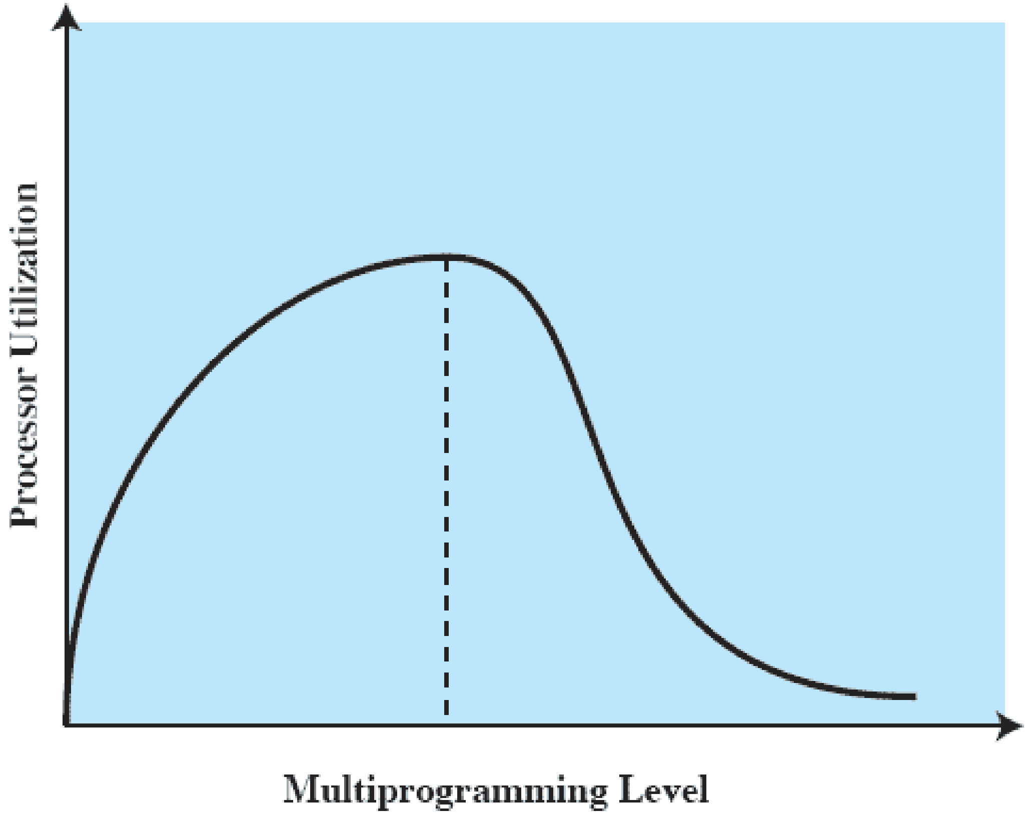


Figure 8.21 Multiprogramming Effects

Process Suspension

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

- Six possibilities exist…

Suspension policies

- Lowest priority process
- Faulting process
  - This process does not have its working set in main memory so it will be blocked anyway
- Process activated long time ago
  - This process is least likely to have its working set resident

Suspension policies cont.

- Process with smallest resident set
  – This process requires the least future effort to reload

- Largest process
  – Obtains the most free frames

- Process with the largest remaining execution window