UTN FRD – Sistemas Operativos
Unidad V Gestión de Memoria

The need for memory management

- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never enough!
- Memory Management, involves swapping blocks of data from secondary storage.
- Memory I/O is slow compared to a CPU
  - The OS must cleverly time the swapping to maximise the CPU’s efficiency

Memory Management

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

Memory Management Requirements

- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation

Requirements: 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 at a different location (relocated)
- Memory references must be translated to the actual physical memory address

Memory Management Terms

Table 7.1 Memory Management Terms

Term	Description
Frame	***Fixed***-length block of main memory.
Page	***Fixed***-length block of data in secondary memory (e.g. on disk).
Segment	***Variable-length*** block of data that resides in secondary memory.

Addressing

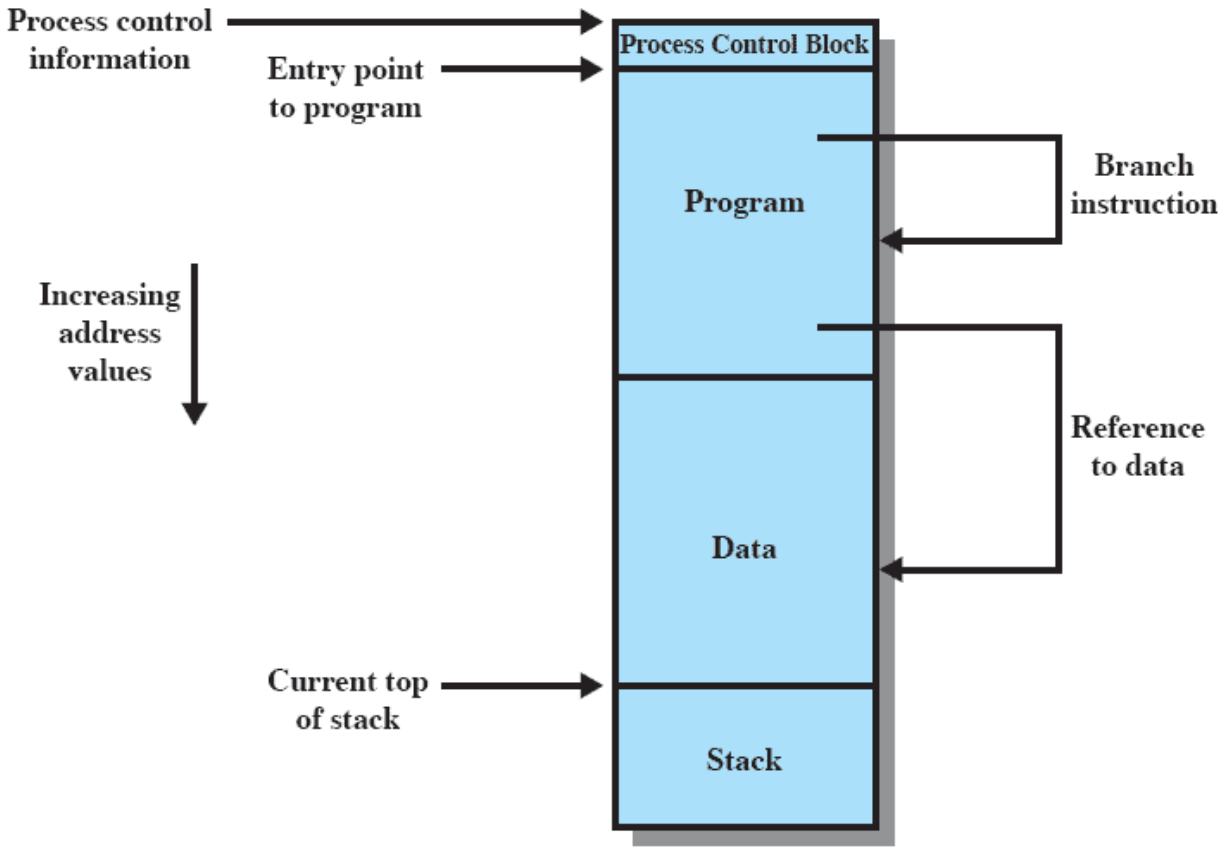


Figure 7.1 Addressing Requirements for a Process

Requirements: Protection

- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time

Requirements: 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

Requirements: 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

Requirements: 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
- Programmer does not know how much space will be available

Partitioning

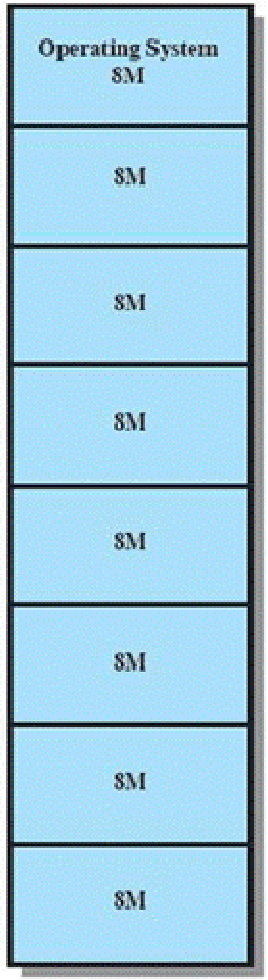
- An early method of managing memory
  - Pre-virtual memory
  - Not used much now
- But, it will clarify the later discussion of virtual memory if we look first at partitioning
  - Virtual Memory has evolved from the partitioning methods

Types of Partitioning

- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation
- Virtual Memory Paging
- Virtual Memory Segmentation

Fixed Partitioning

- Equal-size partitions (see fig 7.3a)
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap a process out of a partition
  - If none are in a ready or running state



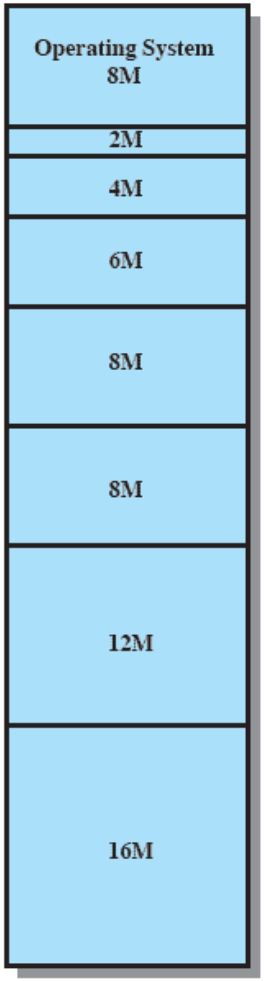
(a) Equal-size partitions

Fixed Partitioning Problems

- A program may not fit in a partition.
  - The programmer must design the program with overlays
- Main memory use is inefficient.
  - Any program, no matter how small, occupies an entire partition.
  - This is results in ***internal fragmentation.***

Solution – Unequal Size Partitions

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

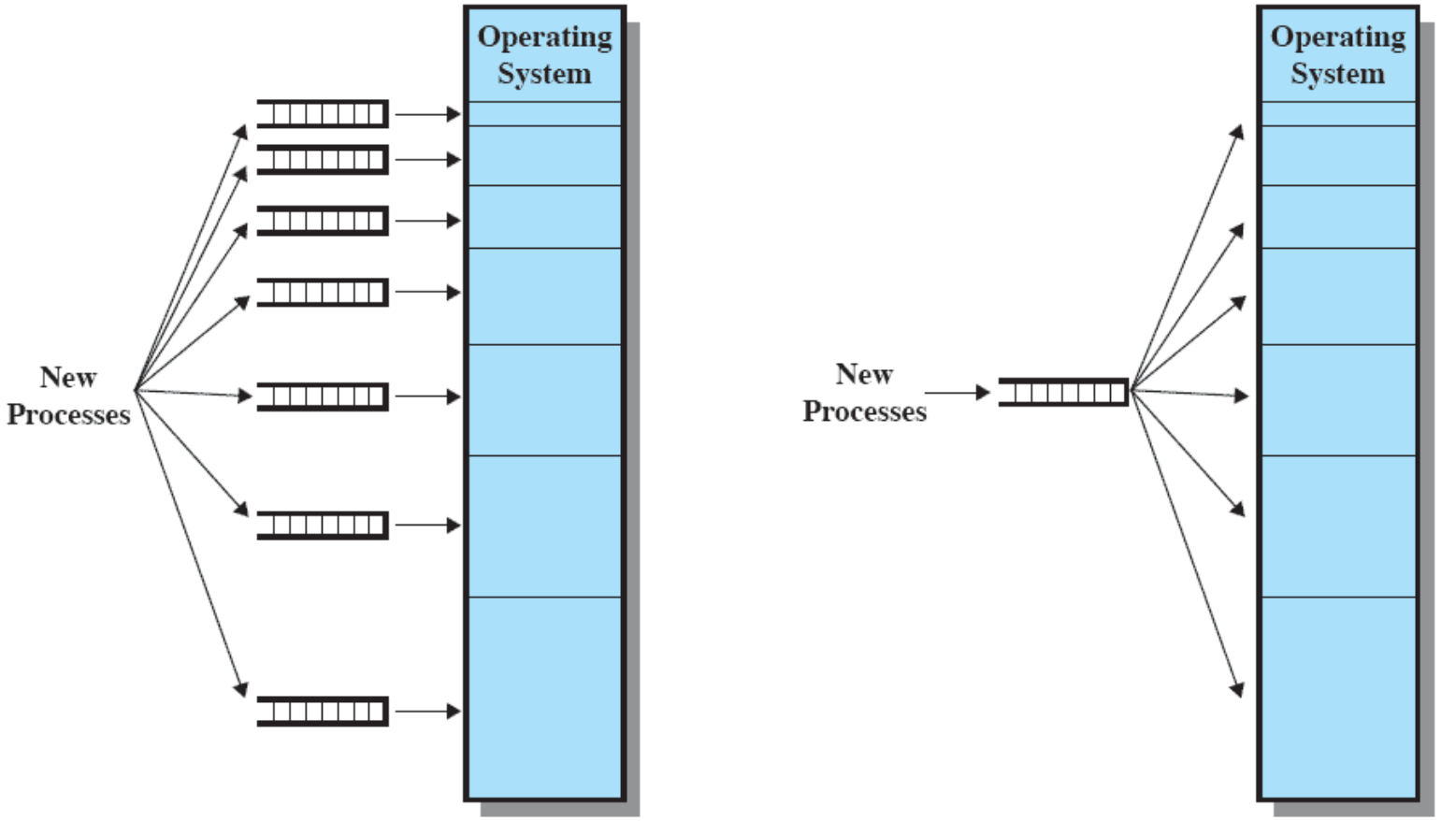


(b) Unequal-size partitions

Placement Algorithm

- Equal-size
  - Placement is trivial (no options)
- Unequal-size
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition

Fixed Partitioning



(a) One process queue per partition

(b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning

Remaining Problems with Fixed Partitions

- The number of active processes is limited by the system
  - I.E limited by the pre-determined number of partitions
- A large number of very small process will not use the space efficiently
  - In either fixed or variable length partition methods

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required

Dynamic Partitioning Example

- ***External Fragmentation***
- Memory external to all processes is fragmented
- Can resolve using ***compaction***
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

Refer to Figure 7.4

Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- 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

Dynamic Partitioning

- 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

Dynamic Partitioning

- 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

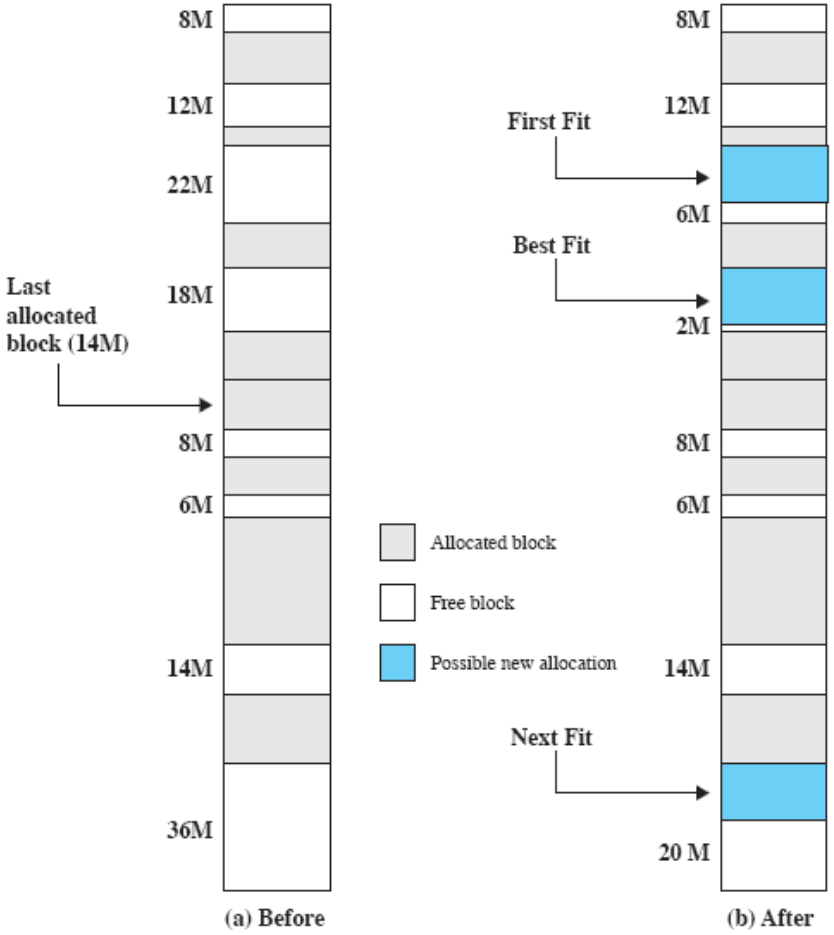


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size *s* where $2^{U-1} < s <= 2^U$
  - entire block is allocated
- Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to *s* is generated

Example of Buddy System

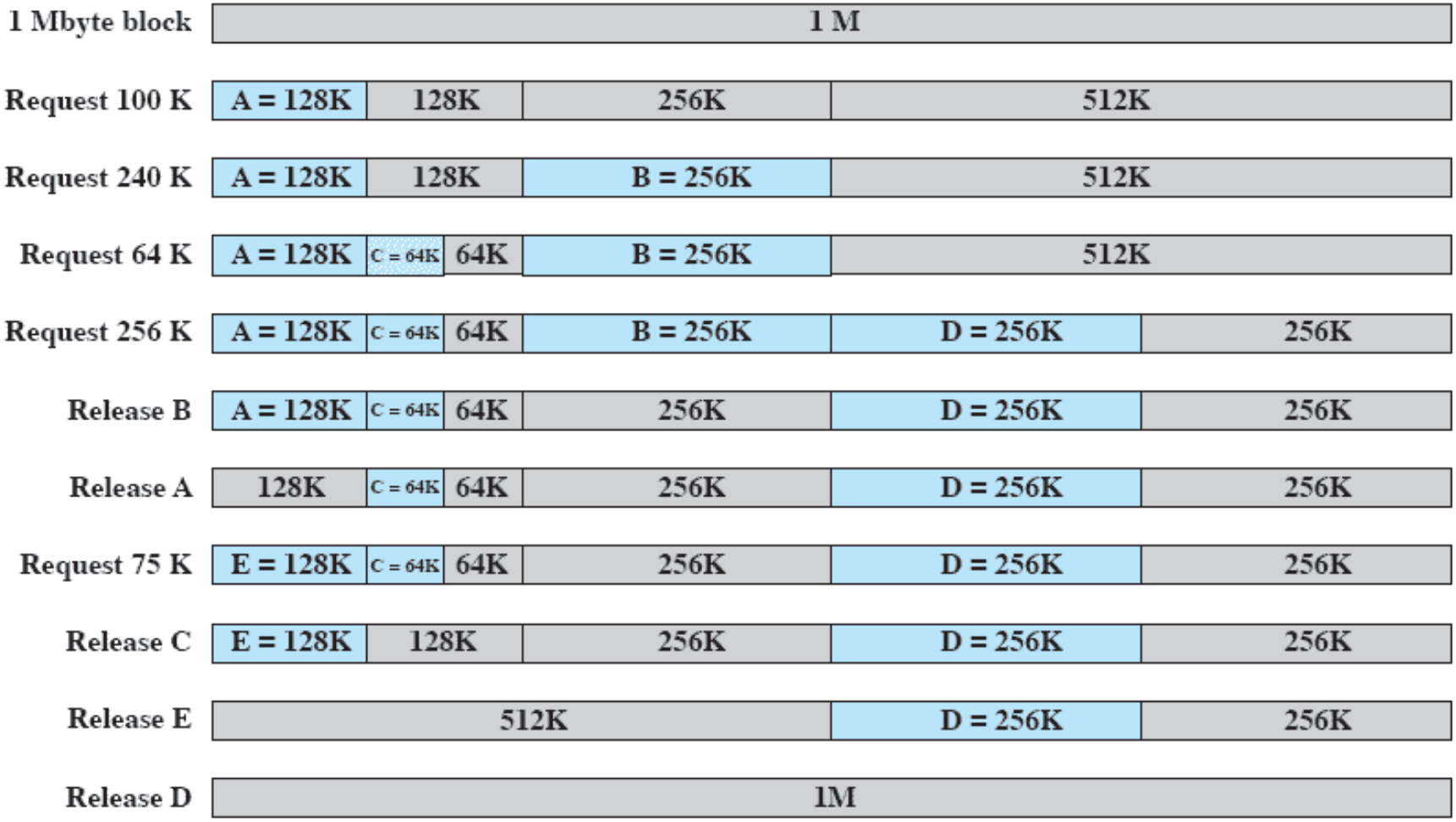


Figure 7.6 Example of Buddy System

Tree Representation of Buddy System

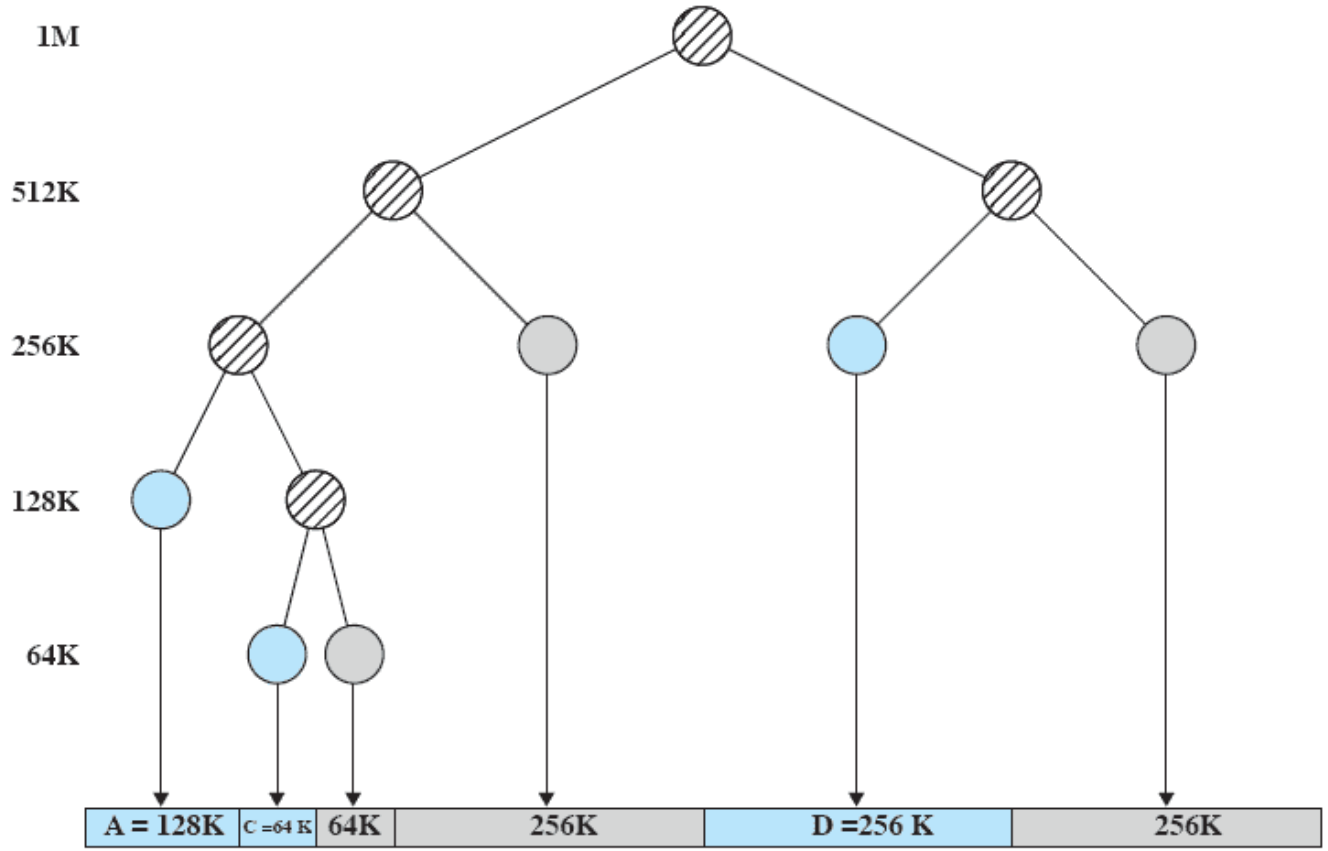


Figure 7.7 Tree Representation of Buddy System

Relocation

- When program loaded into memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution
  - Swapping
  - Compaction

Addresses

- Logical
  - Reference to a memory location independent of the current assignment of data to memory.
- Relative
  - Address expressed as a location relative to some known point.
- Physical or Absolute
  - The absolute address or actual location in main memory.

Relocation

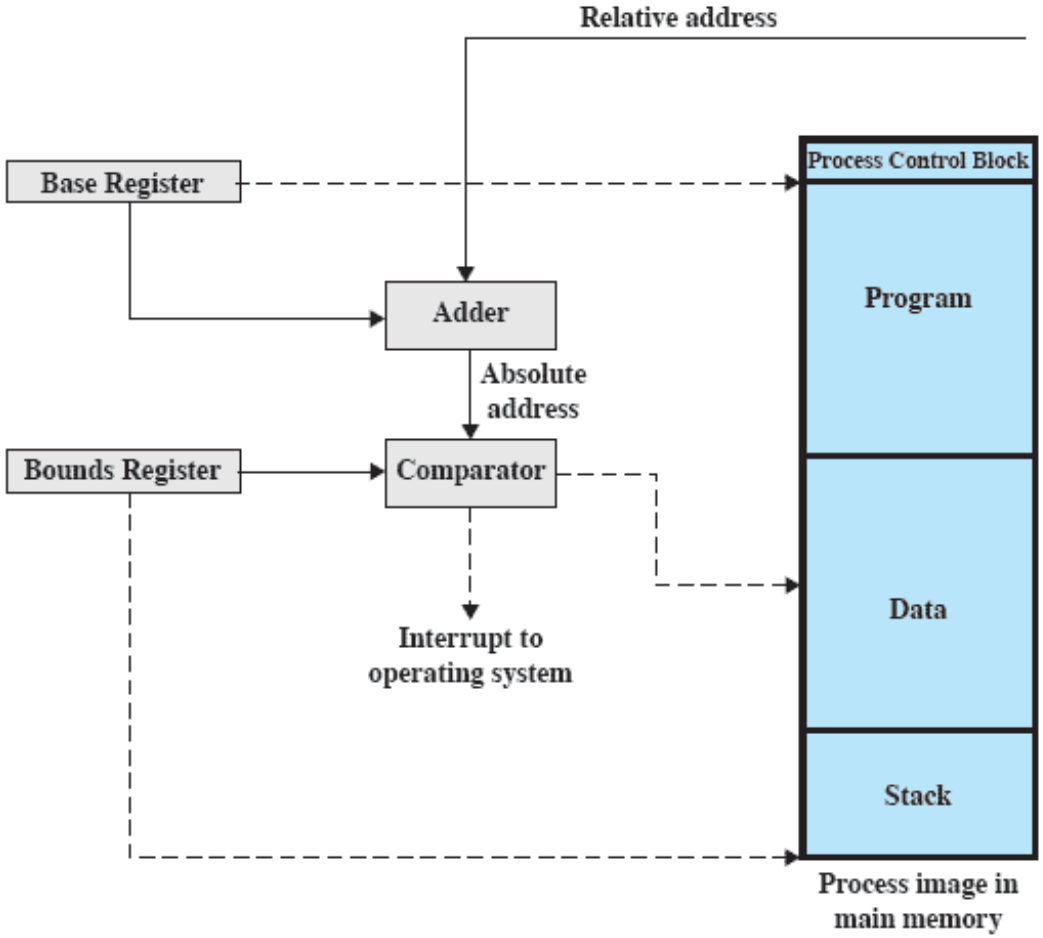


Figure 7.8 Hardware Support for Relocation

Registers Used during Execution

- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in

Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system

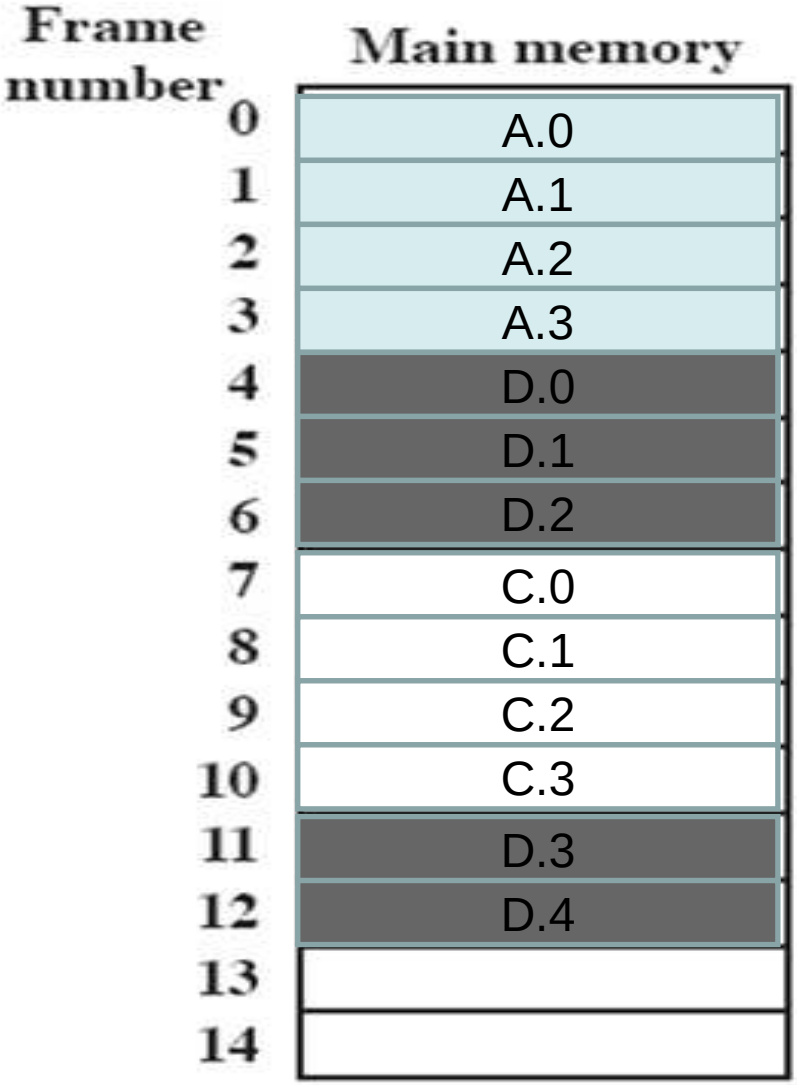
Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called ***pages***
- The chunks of memory are called ***frames***

Paging

- 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

Processes and Frames



Page Table

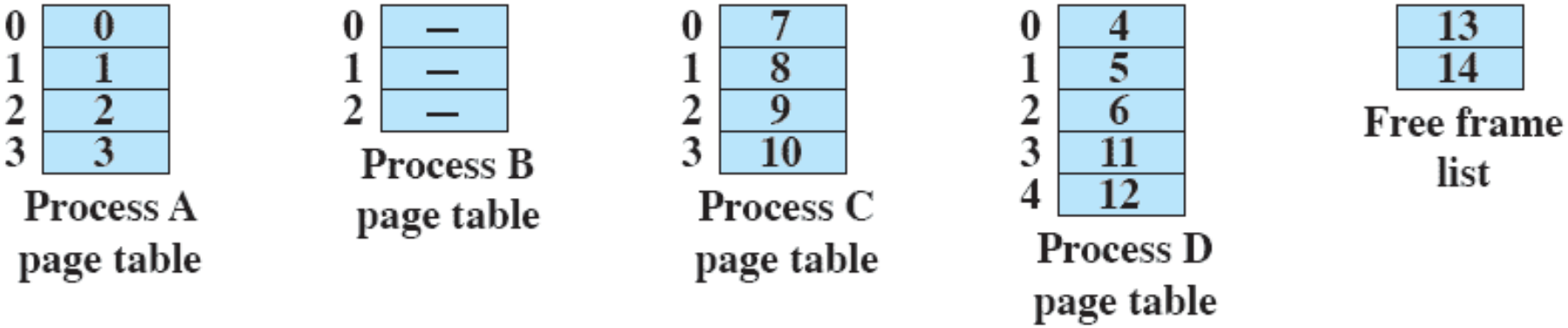


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Segmentation

- A program can be subdivided into segments
  - Segments may vary in length
  - There is a maximum segment length
- Addressing consist of two parts
  - a segment number and
  - an offset
- Segmentation is similar to dynamic partitioning

Logical Addresses

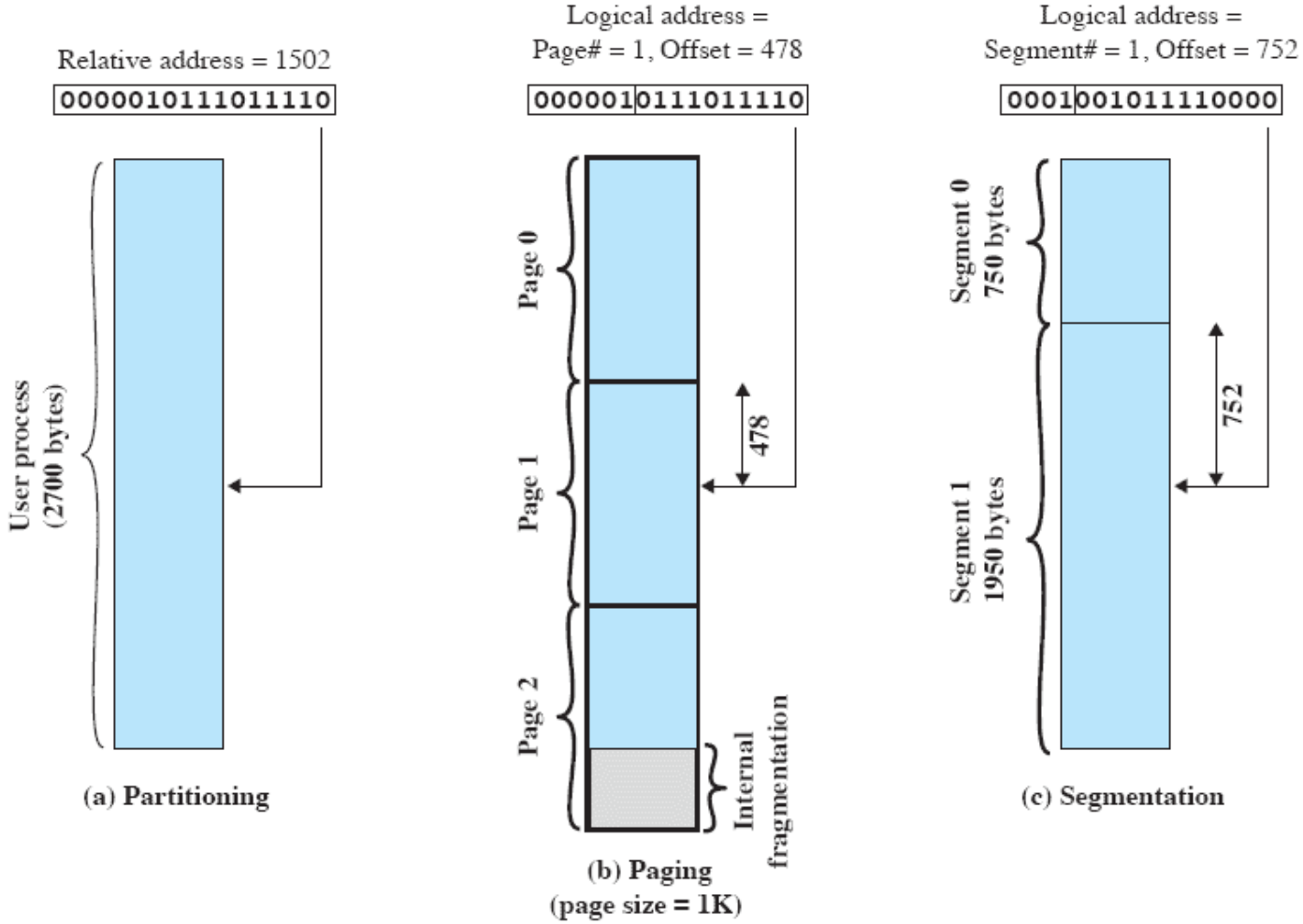
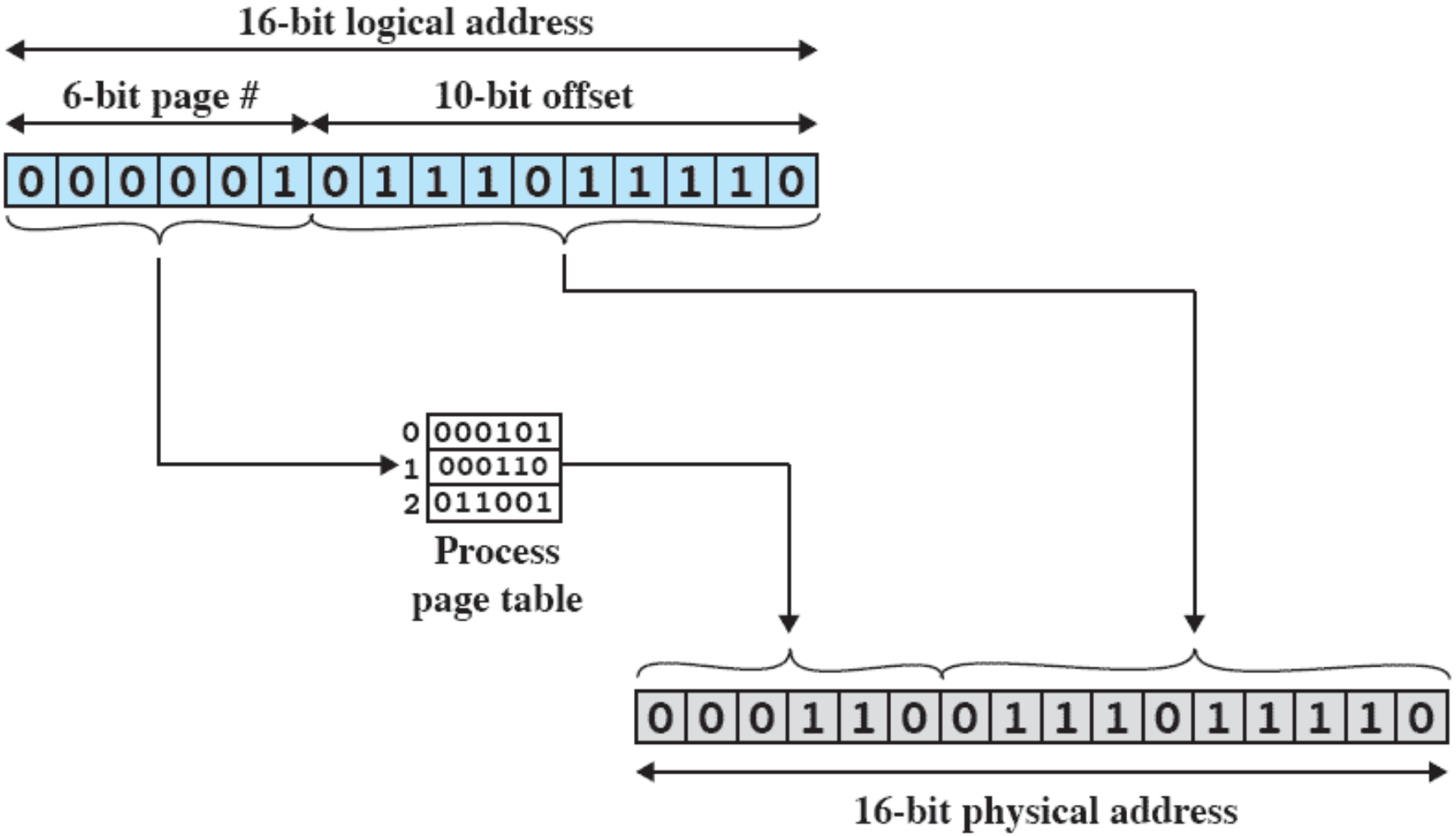


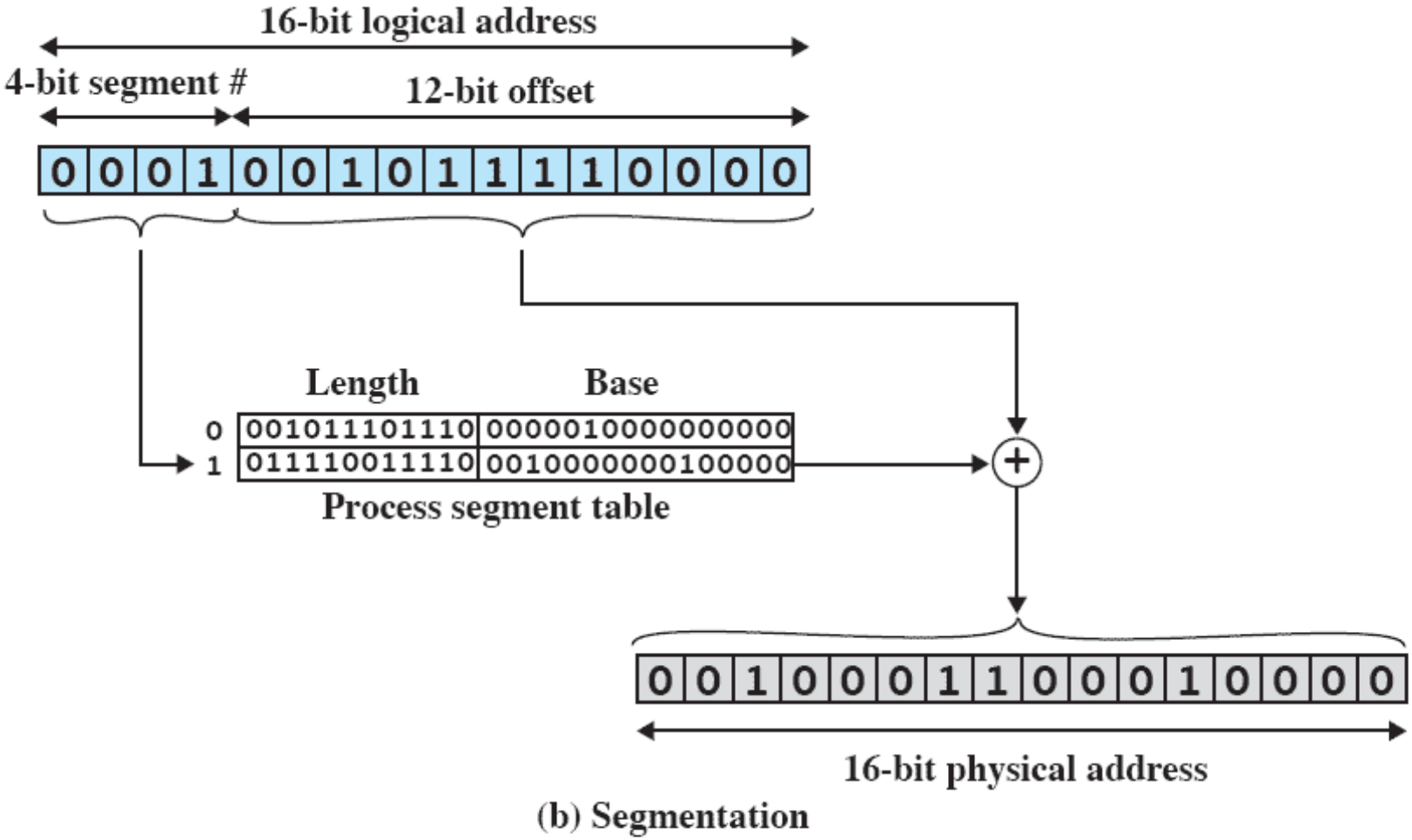
Figure 7.11 Logical Addresses

Paging



(a) Paging

Segmentation



(b) Segmentation

Figure 7.12 Examples of Logical-to-Physical Address Translation

Unidad V Gestión de Memoria
Memoria Virtual

Terminology

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 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 a time

Execution of a Process

- Operating system brings into main memory a few pieces 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
- 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 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
- Virtual memory
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

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

A Processes Performance in VM Environment

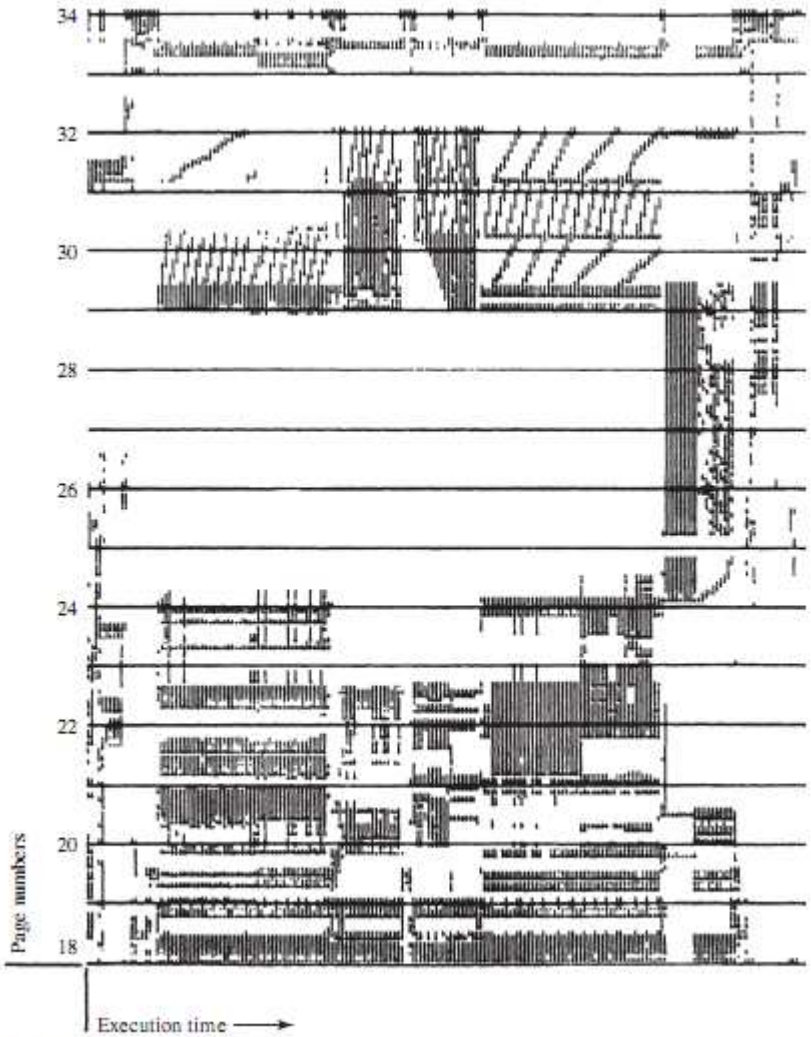


Figure 8.1 Paging Behavior

- Note that during the lifetime of the process, references are confined to a subset of pages.

Support Needed for Virtual Memory

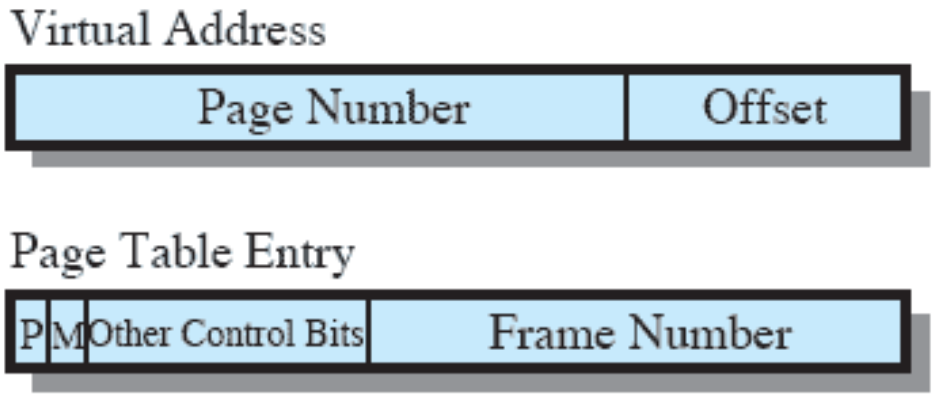
- Hardware must support paging and segmentation
- Operating system must be able to manage the movement of pages and/or segments between secondary memory and main memory

Paging

- 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

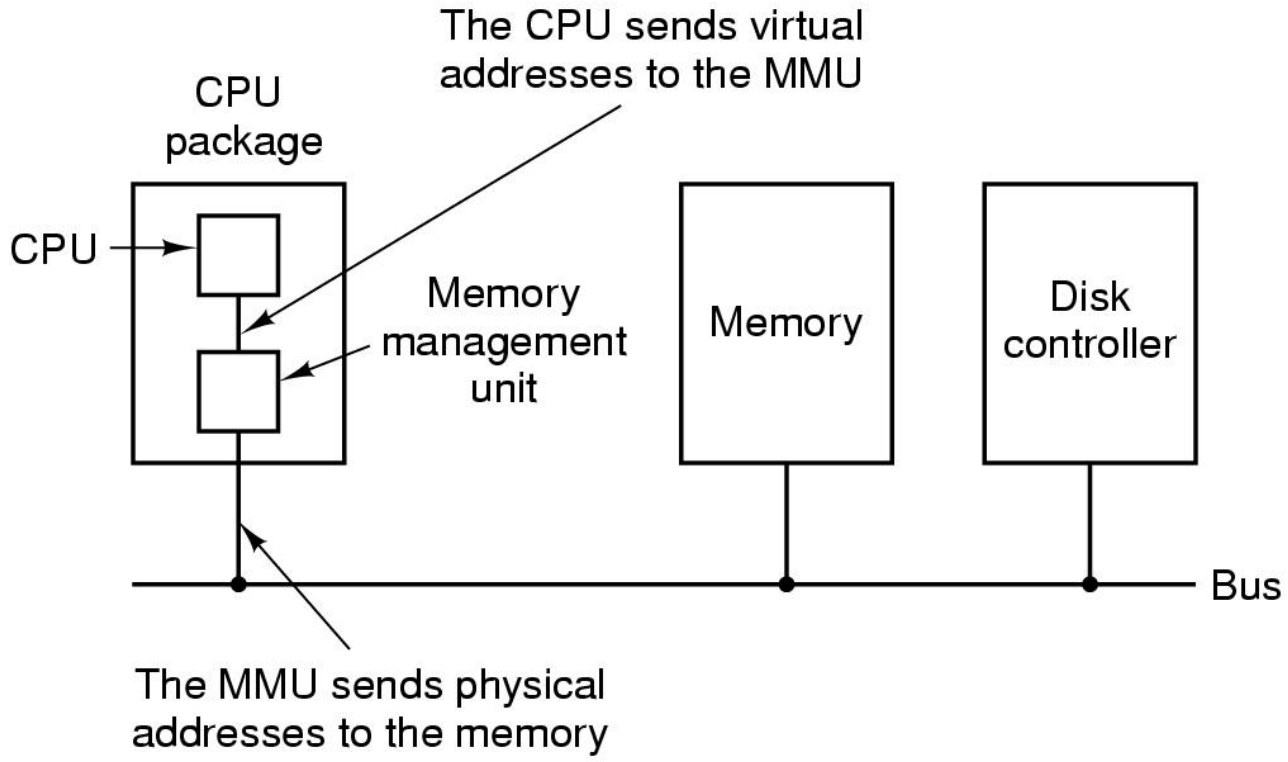
(see next slide)

Paging Table



(a) Paging only

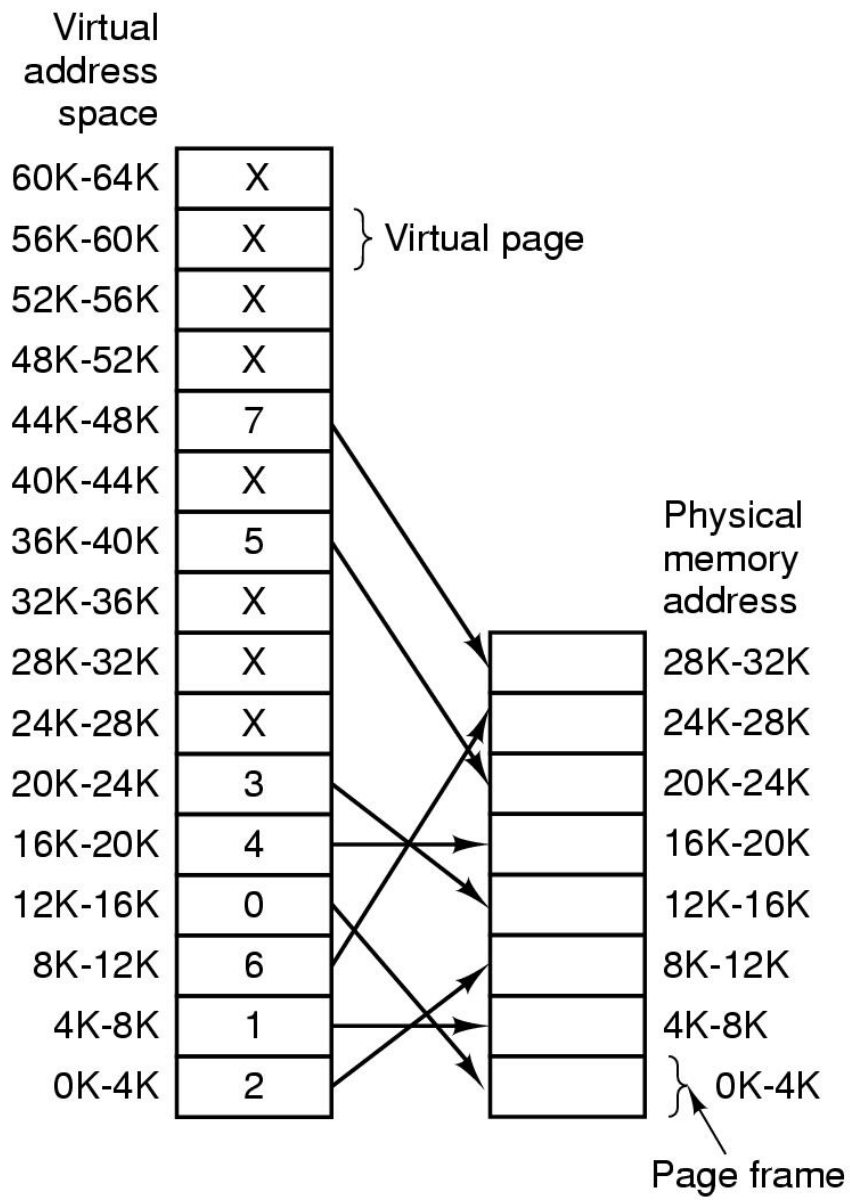
Virtual Memory
Paging (1)



The position and function of the MMU

Paging (2)

The relation between virtual addresses and physical memory addresses given by page table



Address Translation

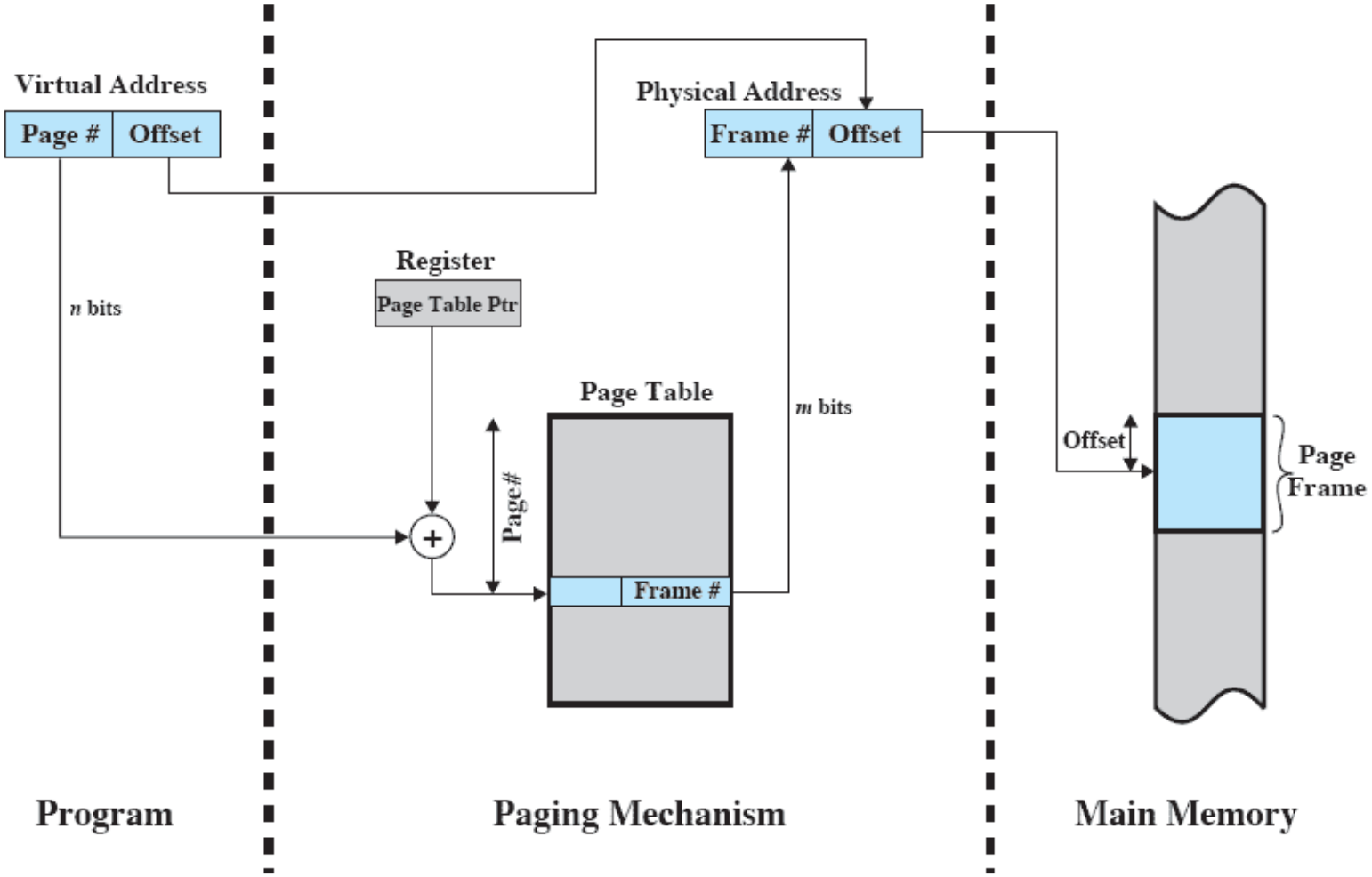


Figure 8.3 Address Translation in a Paging System

Page Tables

- Page tables are also stored in virtual memory
- When a process is running, part of its page table is in main memory

Two-Level Hierarchical Page Table

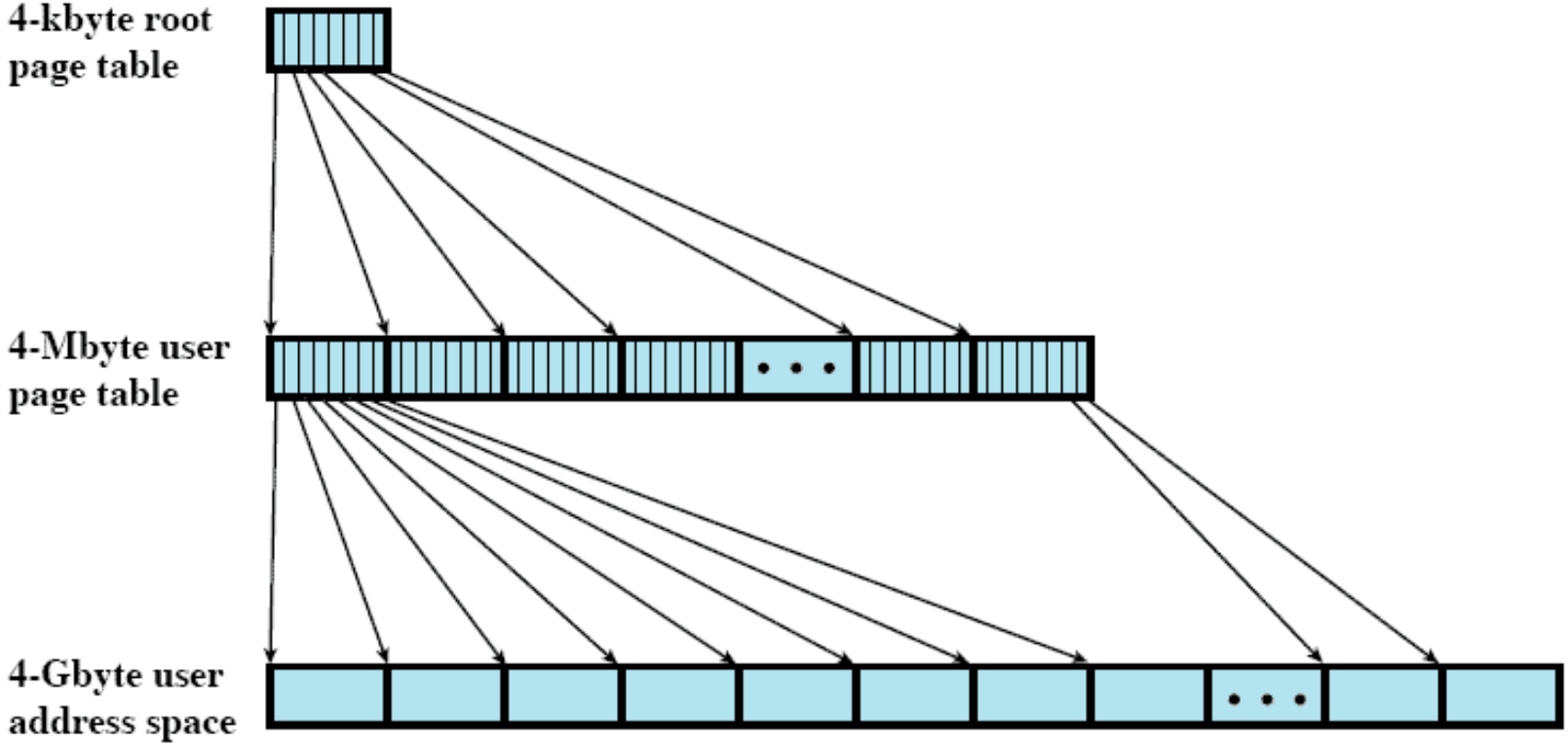


Figure 8.4 A Two-Level Hierarchical Page Table

Address Translation for Hierarchical page table

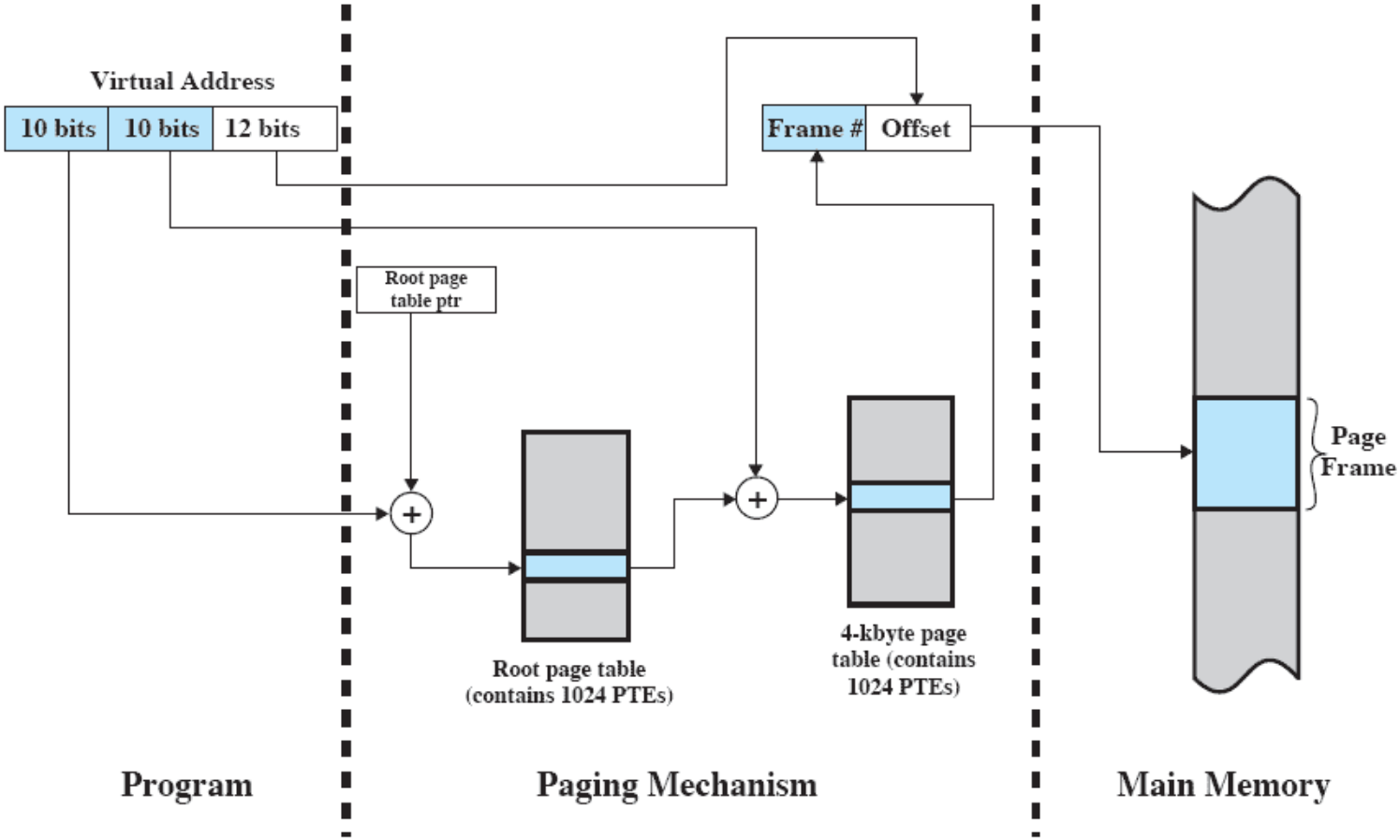


Figure 8.5 Address Translation in a Two-Level Paging System

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

- 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, such as valid, referenced, etc
- Chain pointer
  - the index value of the next entry in the chain.

Inverted Page Table

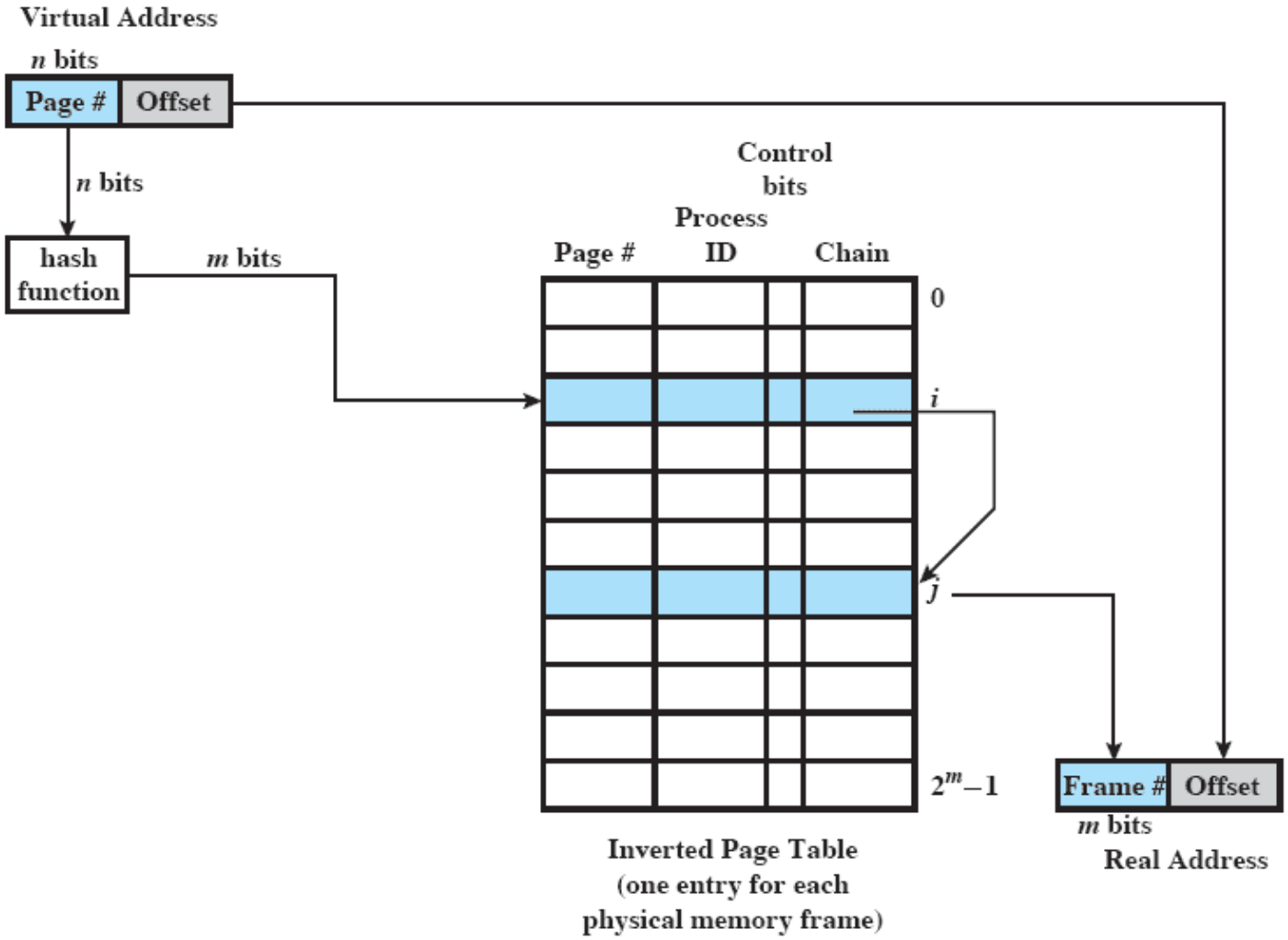


Figure 8.6 Inverted Page Table Structure

Translation Lookaside Buffer

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

Looking into the Process Page Table

- First checks if page is already in main memory
  - If not in main memory a page fault is issued
- The TLB is updated to include the new page entry

Translation Lookaside Buffer

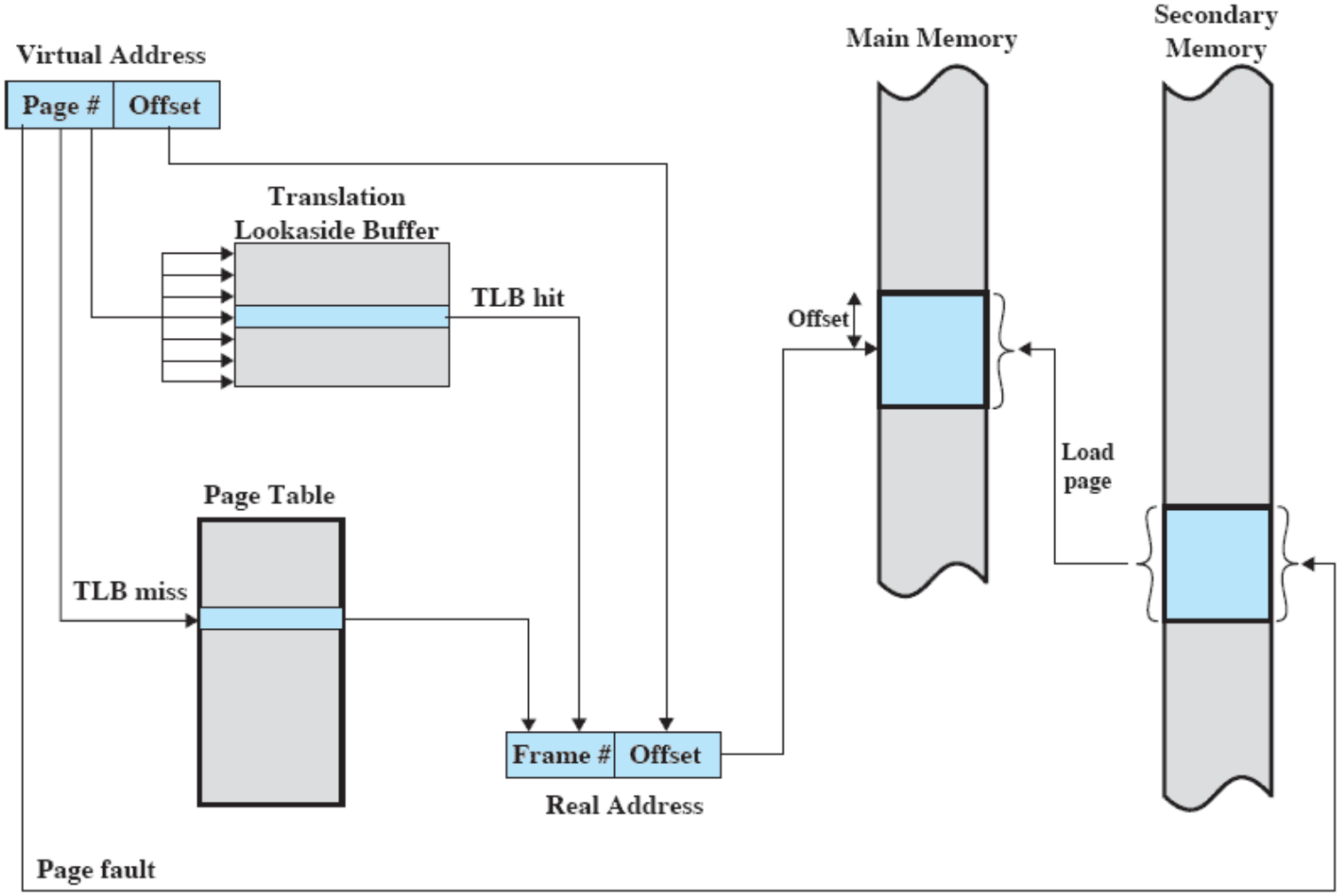


Figure 8.7 Use of a Translation Lookaside Buffer

TLB operation

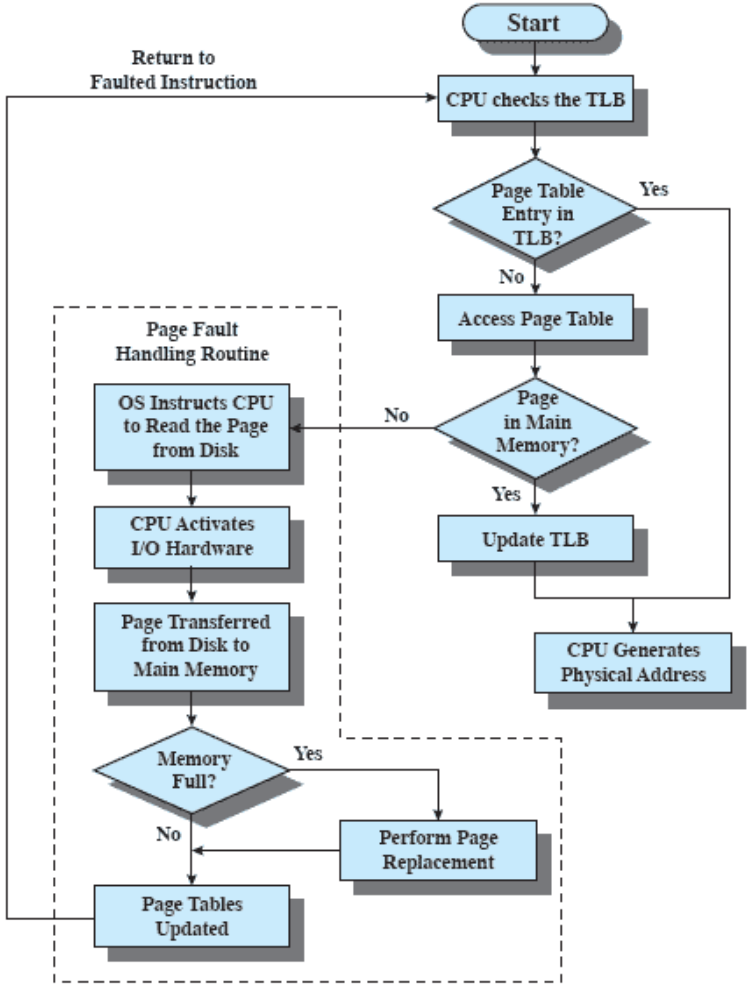


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

Page Size

- Smaller page size, less amount of internal fragmentation
- But Smaller page size, 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

Page Size

- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better

Further complications to Page Size

- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

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

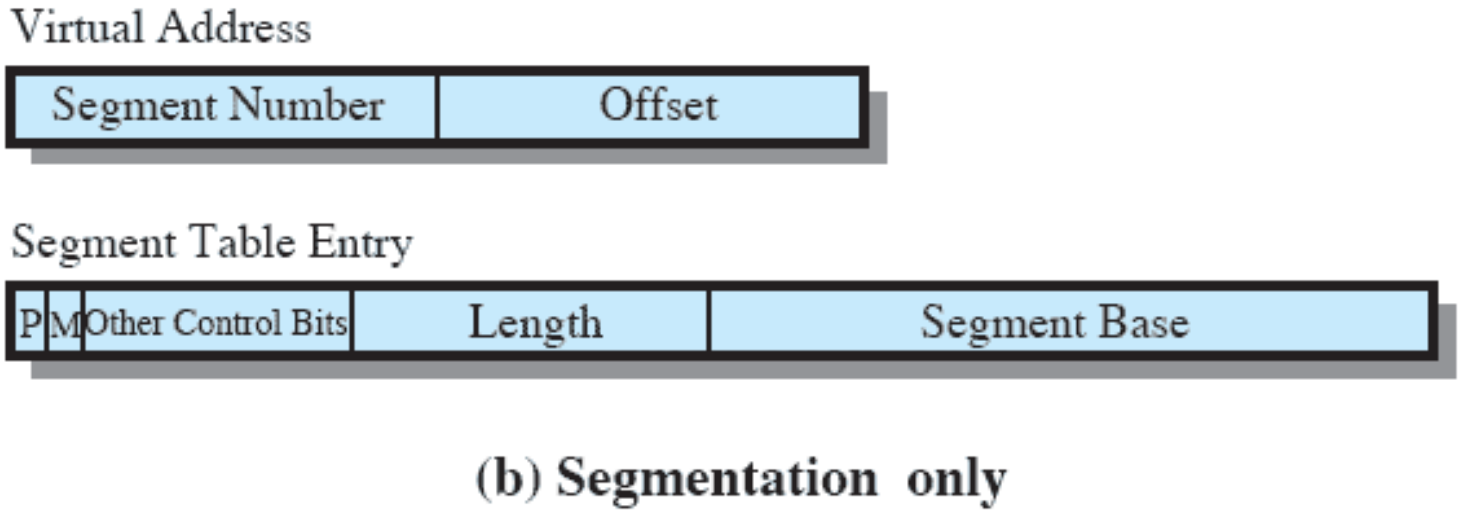
Segmentation

- 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

- Starting address corresponding segment in main memory
- Each entry contains the 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

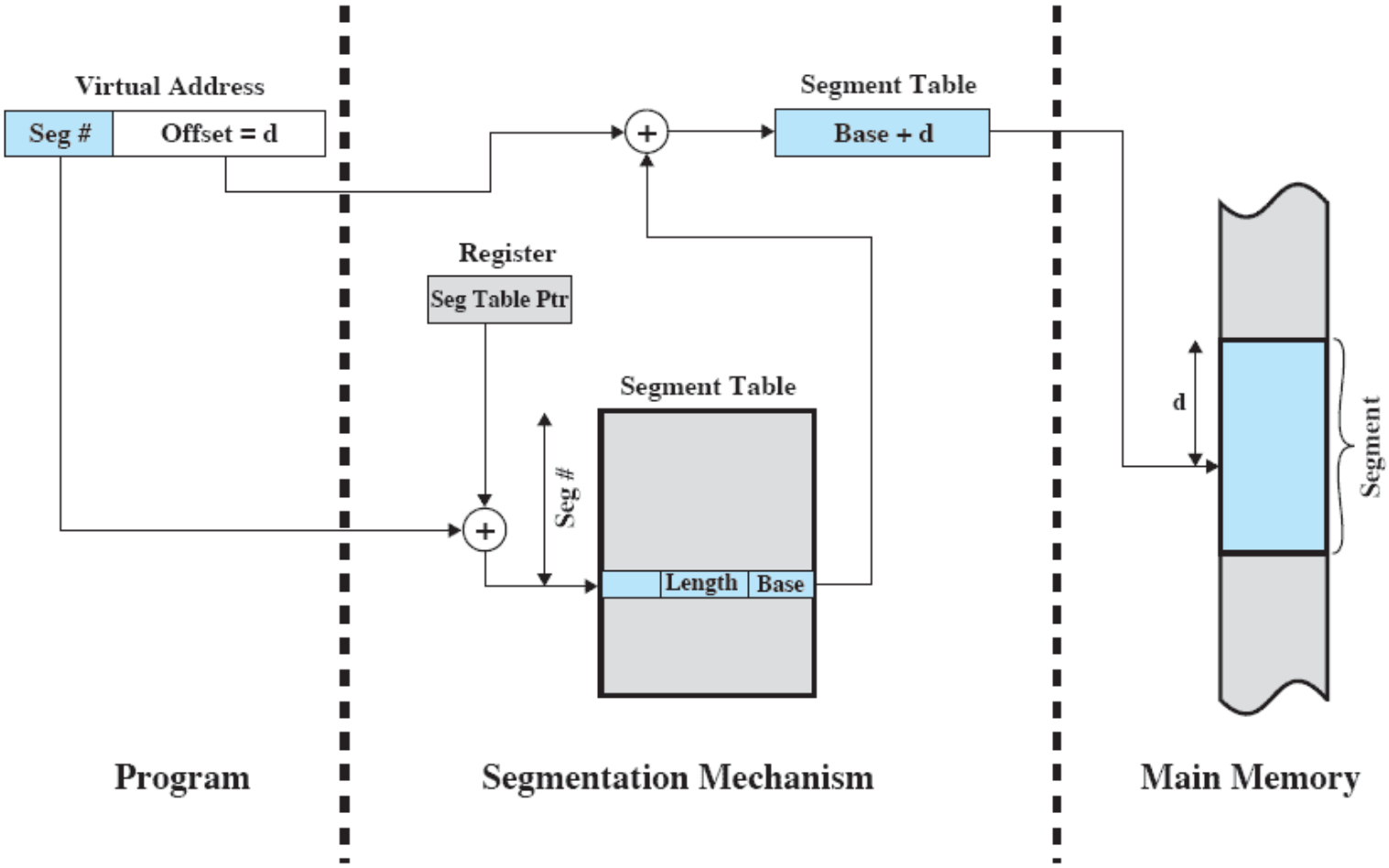
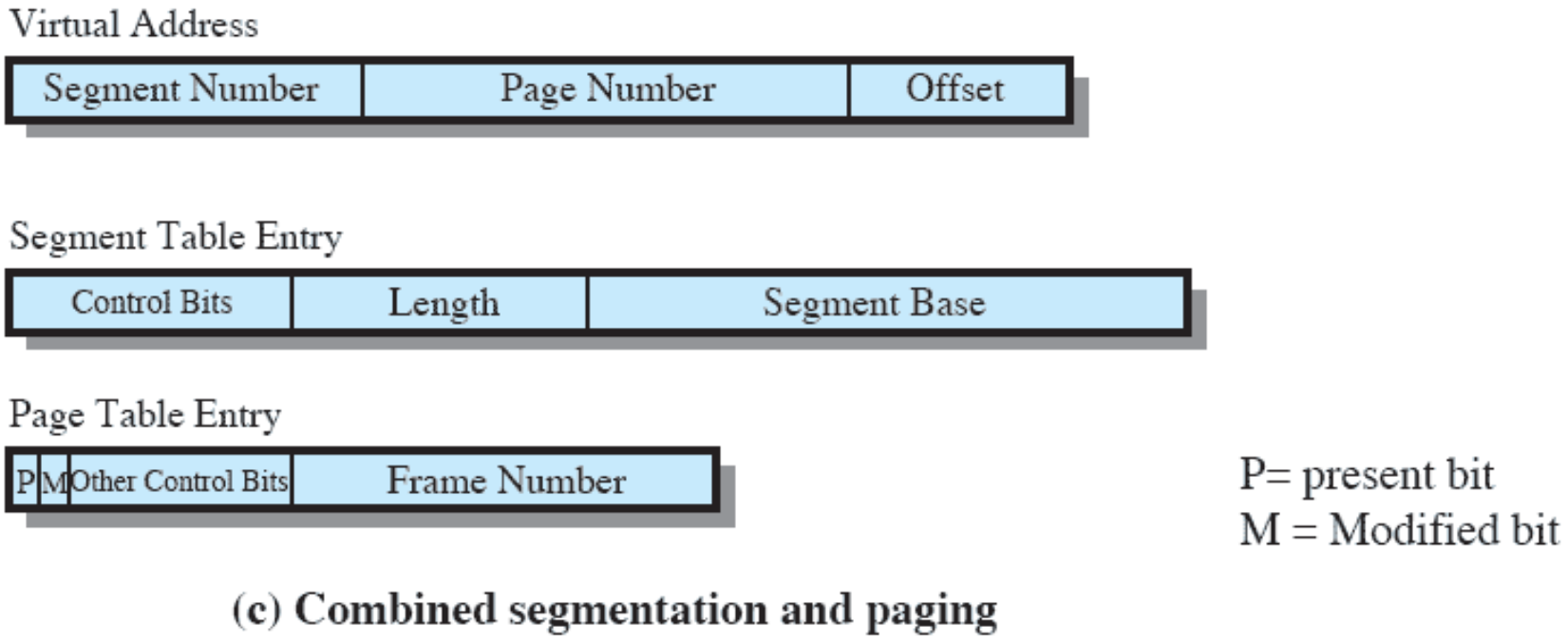


Figure 8.12 Address Translation in a Segmentation System

Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages

Combined Paging and Segmentation



(c) Combined segmentation and paging

Address Translation

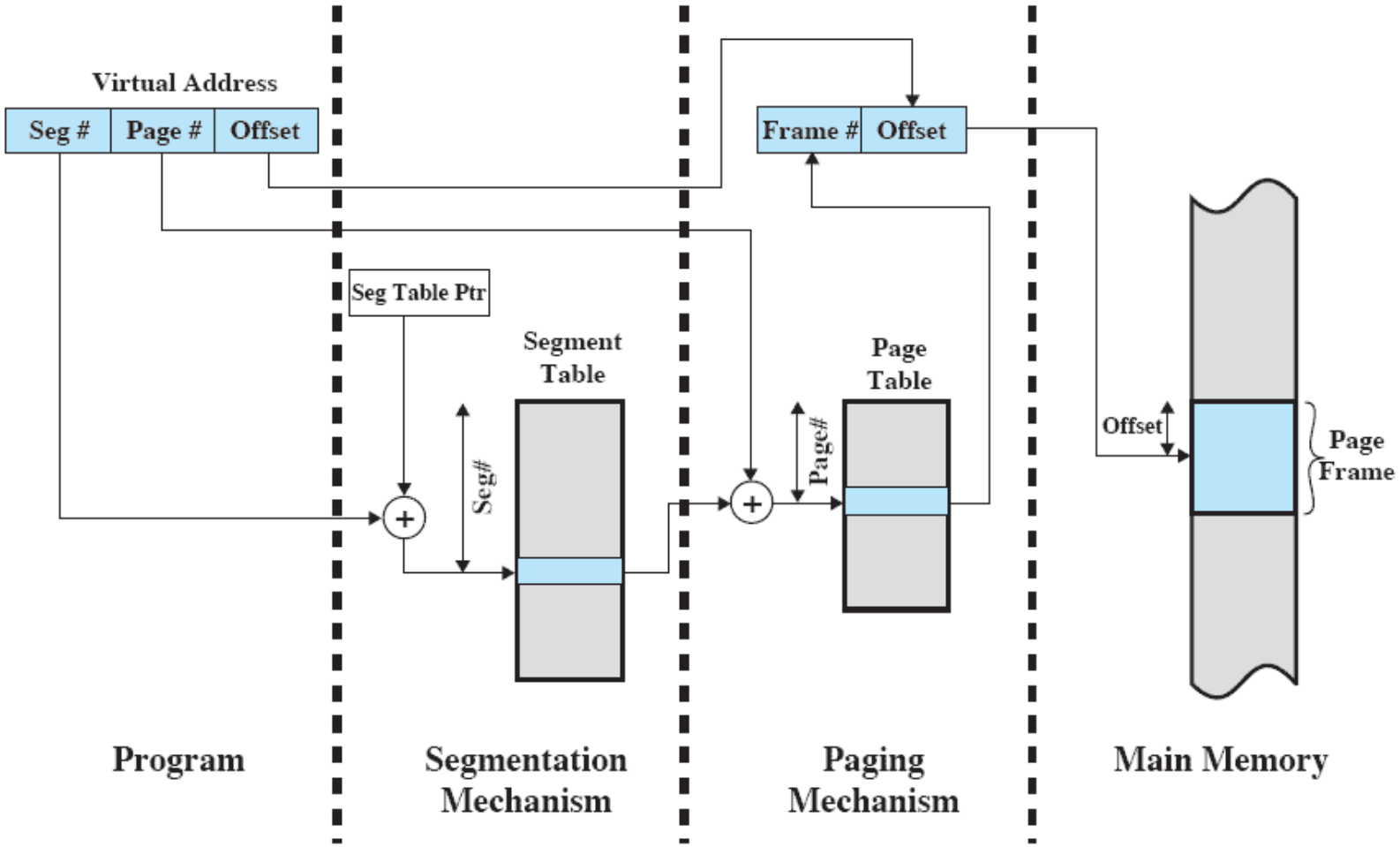


Figure 8.13 Address Translation in a Segmentation/Paging System

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
- Sharing can be achieved by segments referencing multiple processes

Protection Relationships

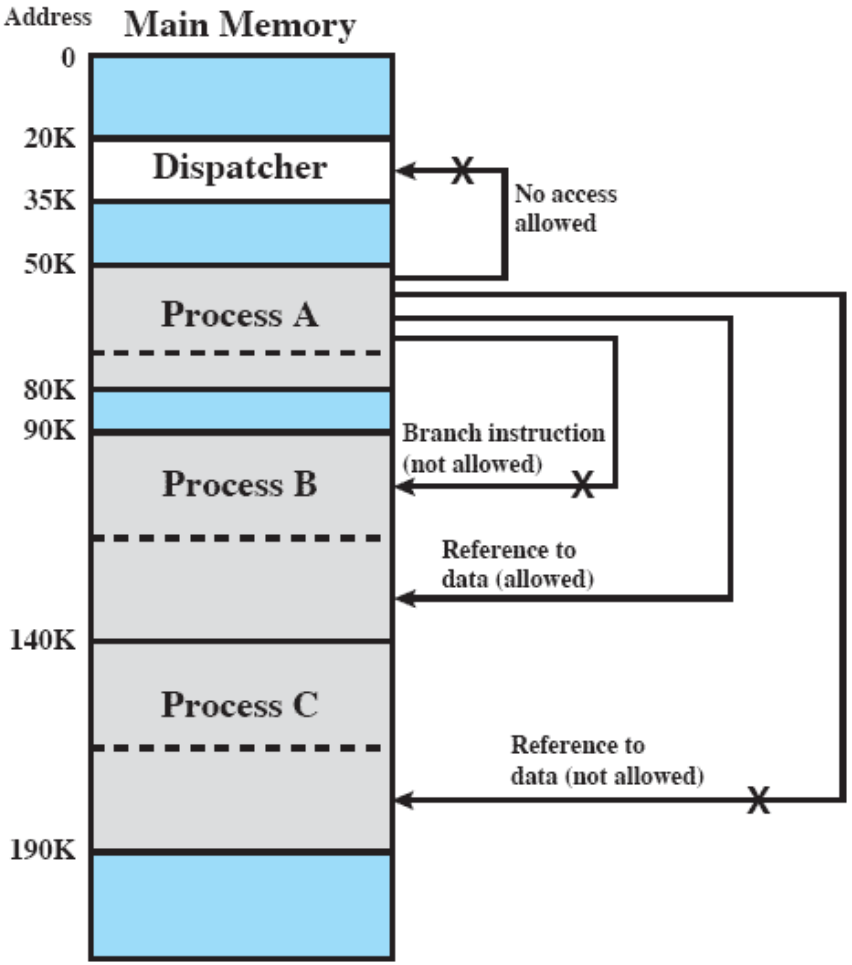


Figure 8.14 Protection Relationships Between Segments

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

Table 8.4 Operating System Policies for Virtual Memory

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
- Important in a segmentation system
- Paging or combined paging with segmentation hardware performs address translation

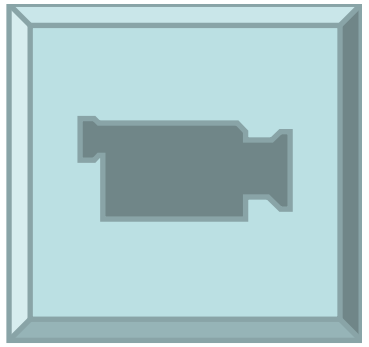
Replacement Policy

- 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: Frame Locking



- Frame Locking
  - If frame is locked, it may not be replaced
  - Kernel of the operating system
  - Key control structures
  - I/O buffers
  - 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
- Examples

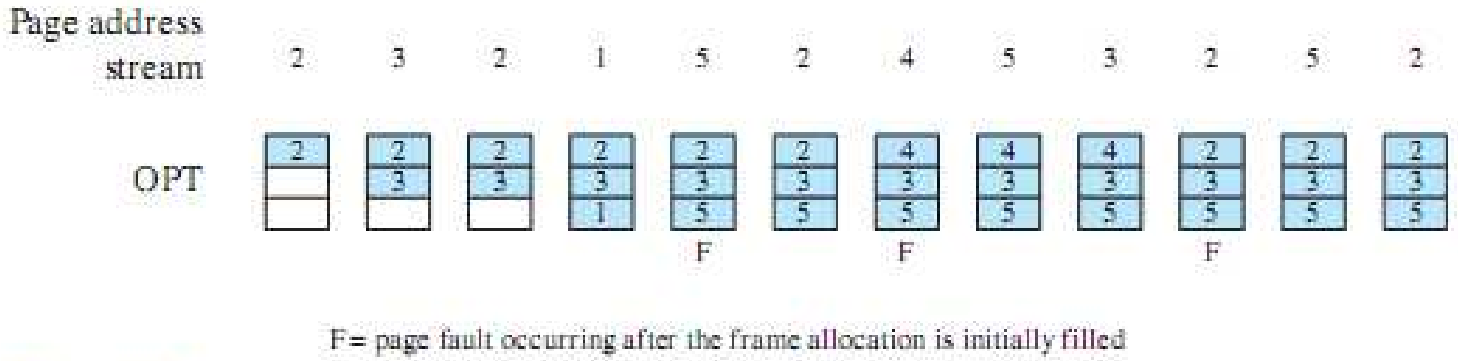
Examples

- 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.

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

Optimal Policy Example



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

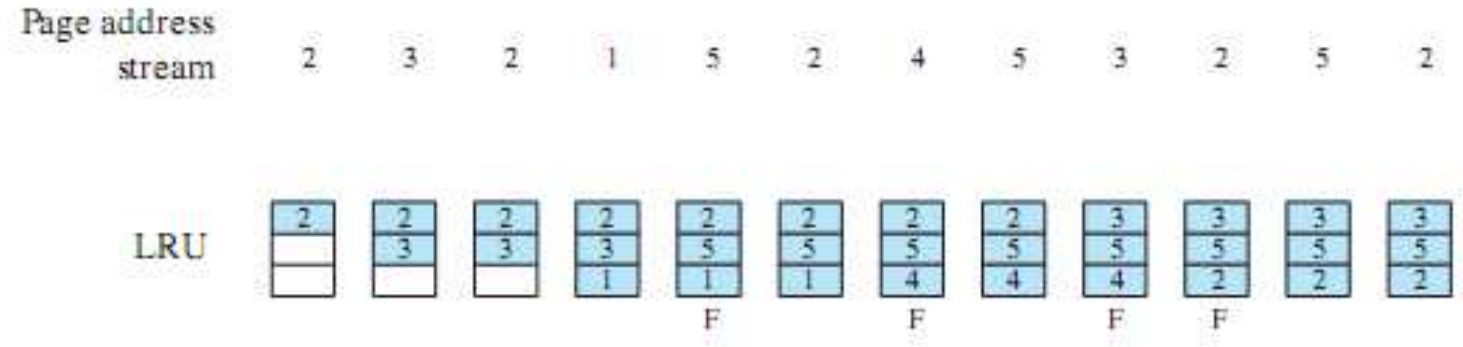
Figure 8.15 **Behavior of Four Page Replacement Algorithms**

- The optimal policy produces three page faults after the frame allocation has been filled.

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



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

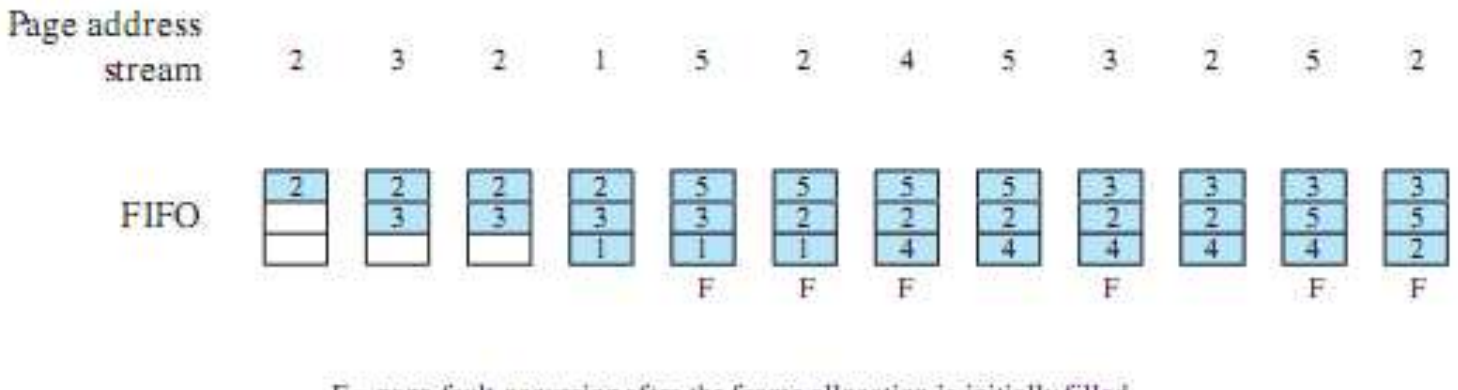
Figure 8.15 **Behavior of Four Page Replacement Algorithms**

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

First-in, first-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - Simplest replacement policy to implement
- Page that has been in memory the longest is replaced
  - But, these pages may be needed again very soon if it hasn’t truly fallen out of use

FIFO Example

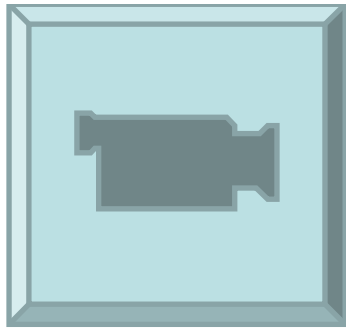


F= page fault occurring after the frame allocation is initially filled

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

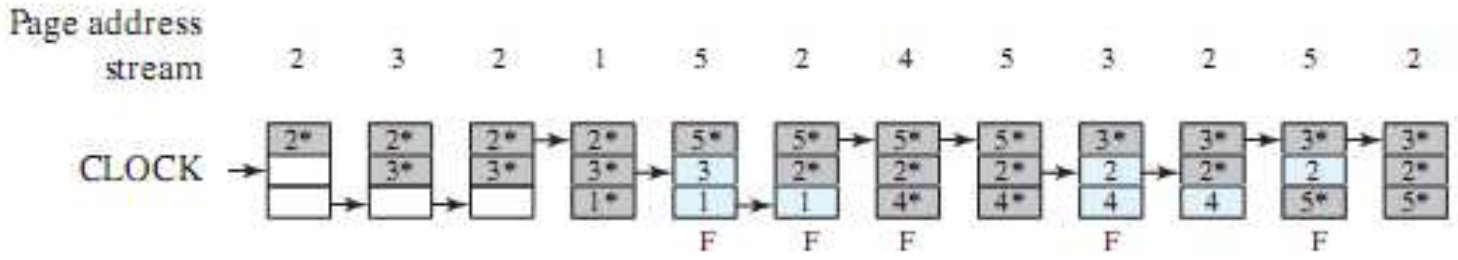
- The FIFO policy results in six page faults.
  - Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

Clock Policy



- Uses and 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

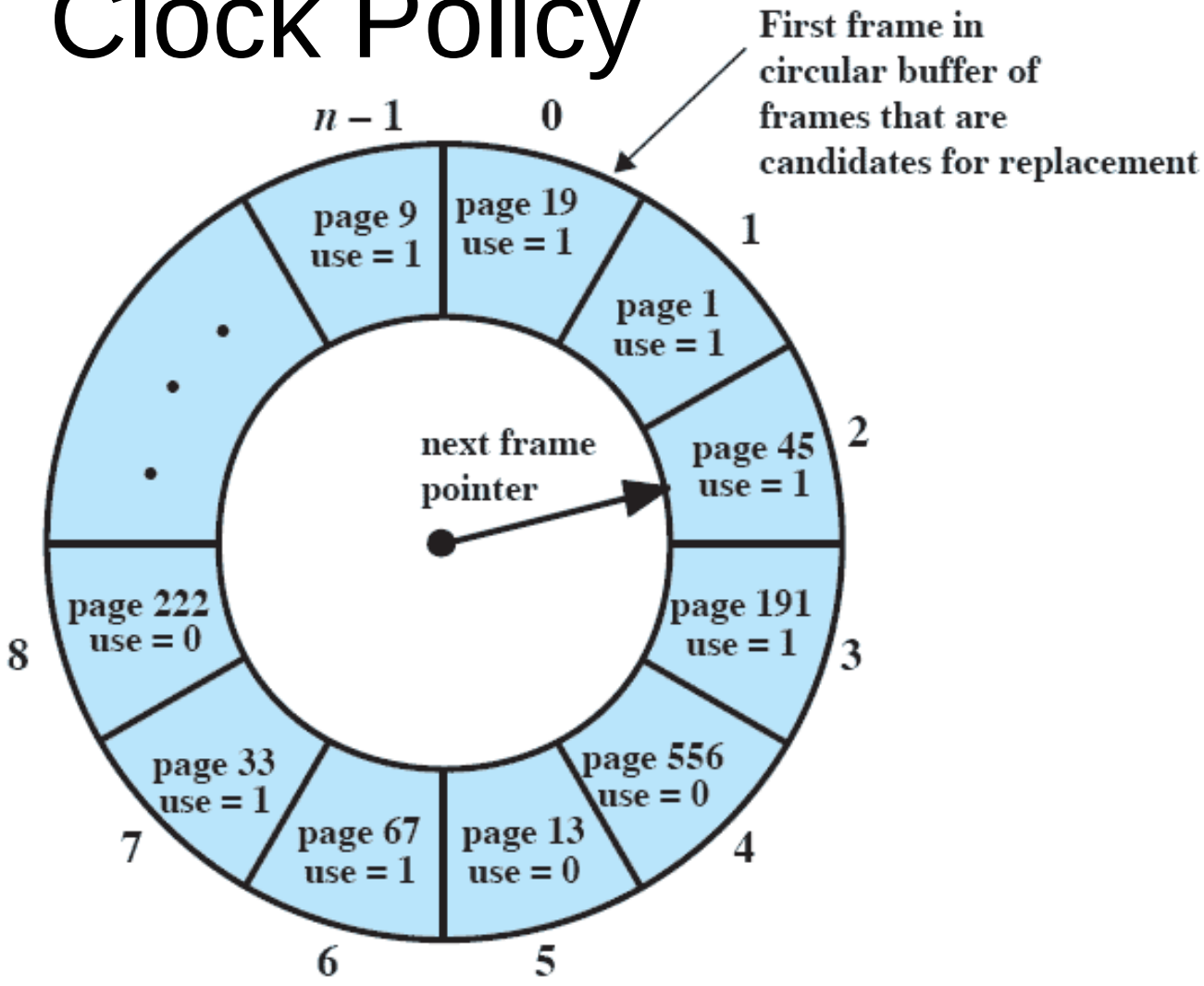


F = page fault occurring after the frame allocation is initially filled

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

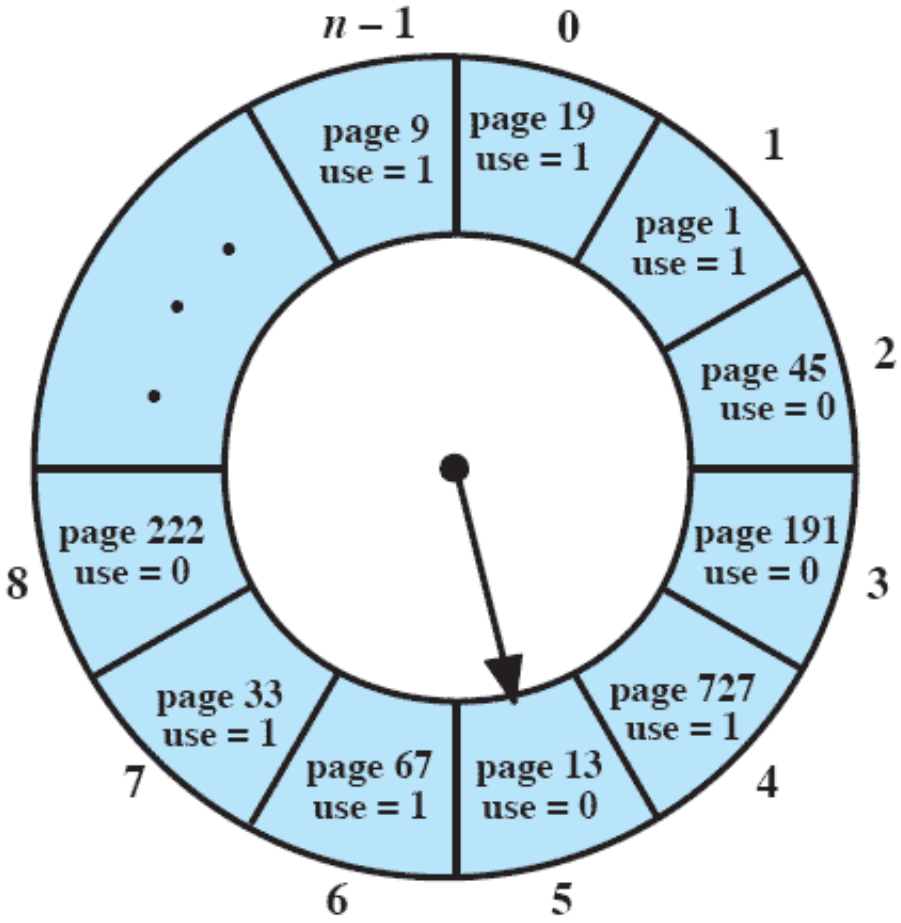
- Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

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

Clock Policy

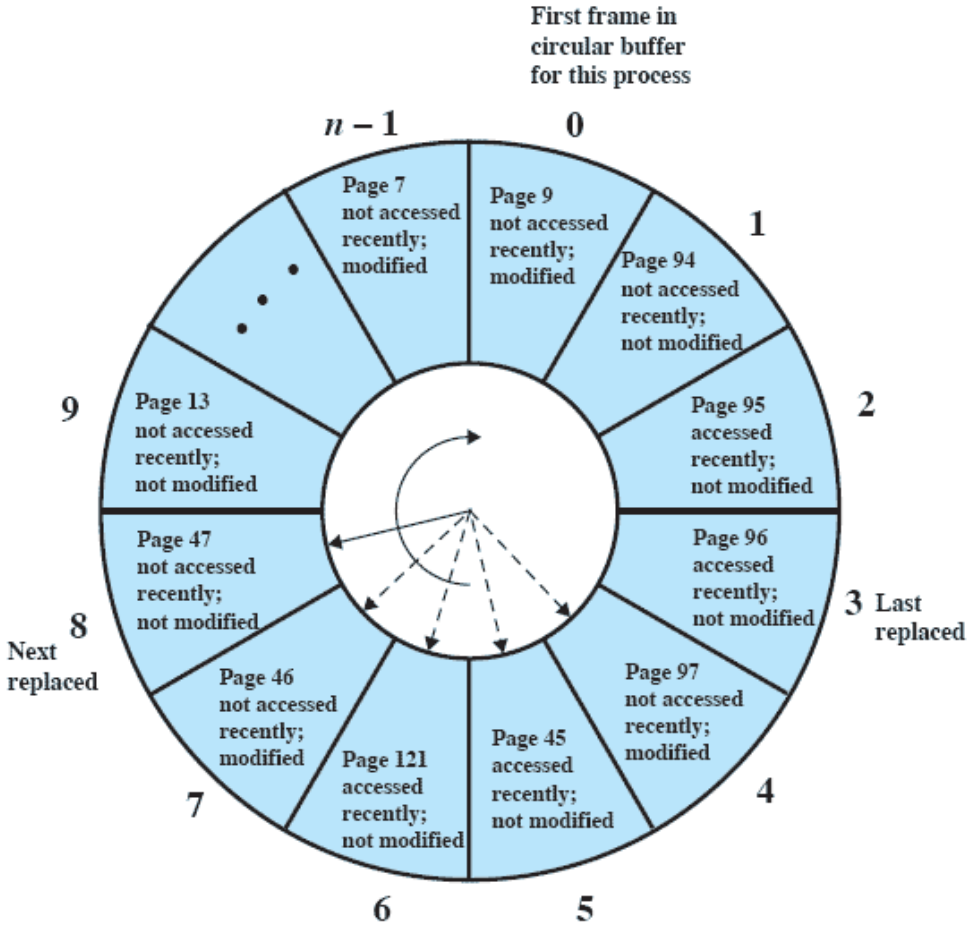
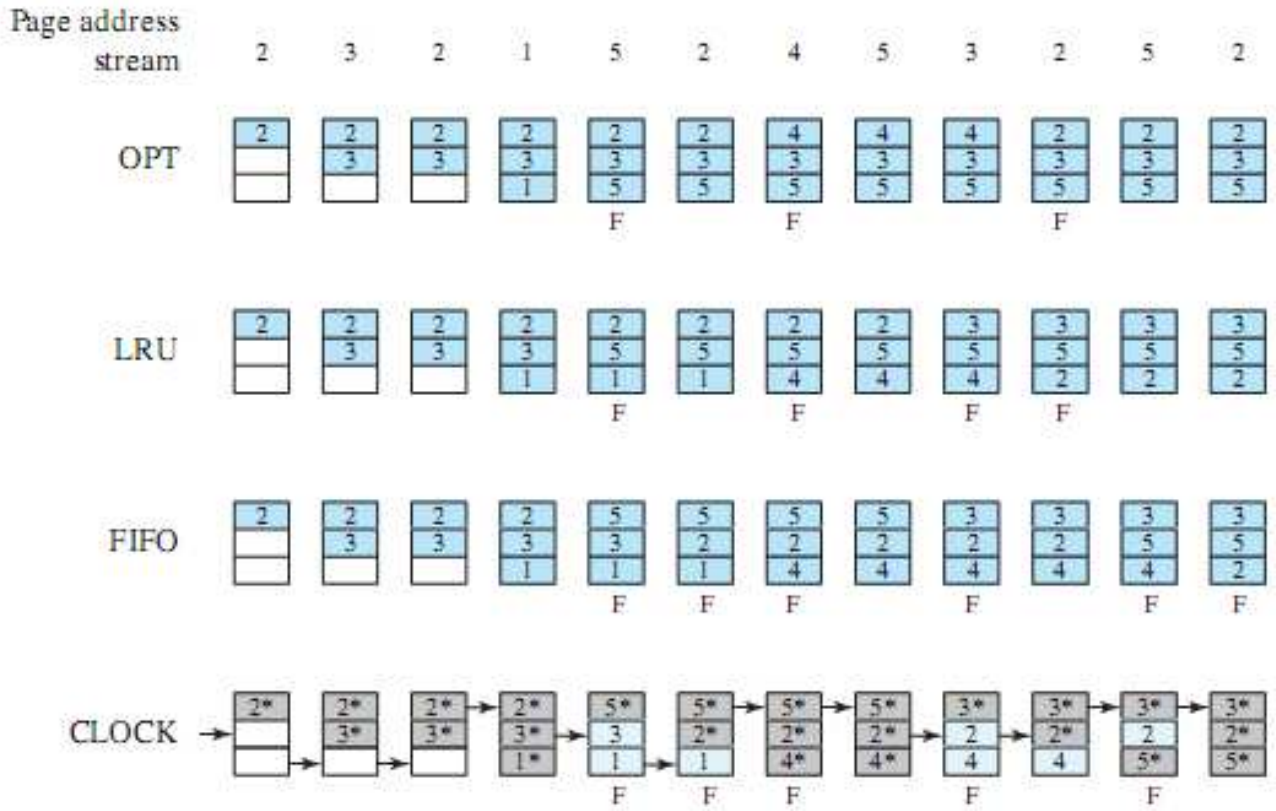


Figure 8.18 The Clock Page-Replacement Algorithm [GOLD89]

Combined Examples



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

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

Comparison

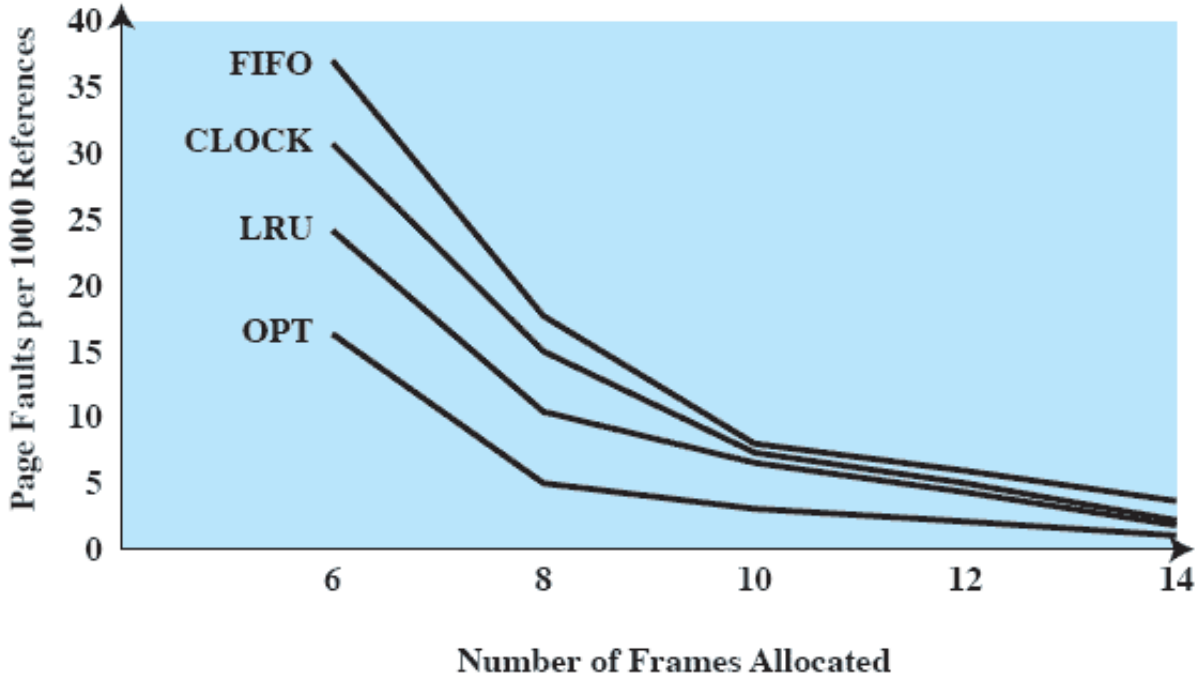


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Page Buffering

- LRU and Clock policies both involve complexity and overhead
  - Also, replacing a modified page is more costly than unmodified as needs written to secondary memory
- Solution: Replaced page is added to one of two lists
  - Free page list if page has not been modified
  - Modified page list

Replacement Policy and Cache Size

- Main memory size is getting larger and the locality of applications is decreasing.
  - So, cache sizes have been increasing
- With large caches, replacement of pages can have a performance impact
  - improve performance by supplementing the page replacement policy with a with a policy for page placement in the page buffer

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 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
  - When a page fault occurs, one of the pages of that process must be replaced
- Variable-allocation
  - Number of pages allocated to a process varies 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.
  - 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

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

- 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, replaces one from another process
  - Therein lies the difficulty … which 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

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.

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
- Precleaning
  - Pages are written out in batches

Cleaning Policy

- Best approach uses page buffering
- 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

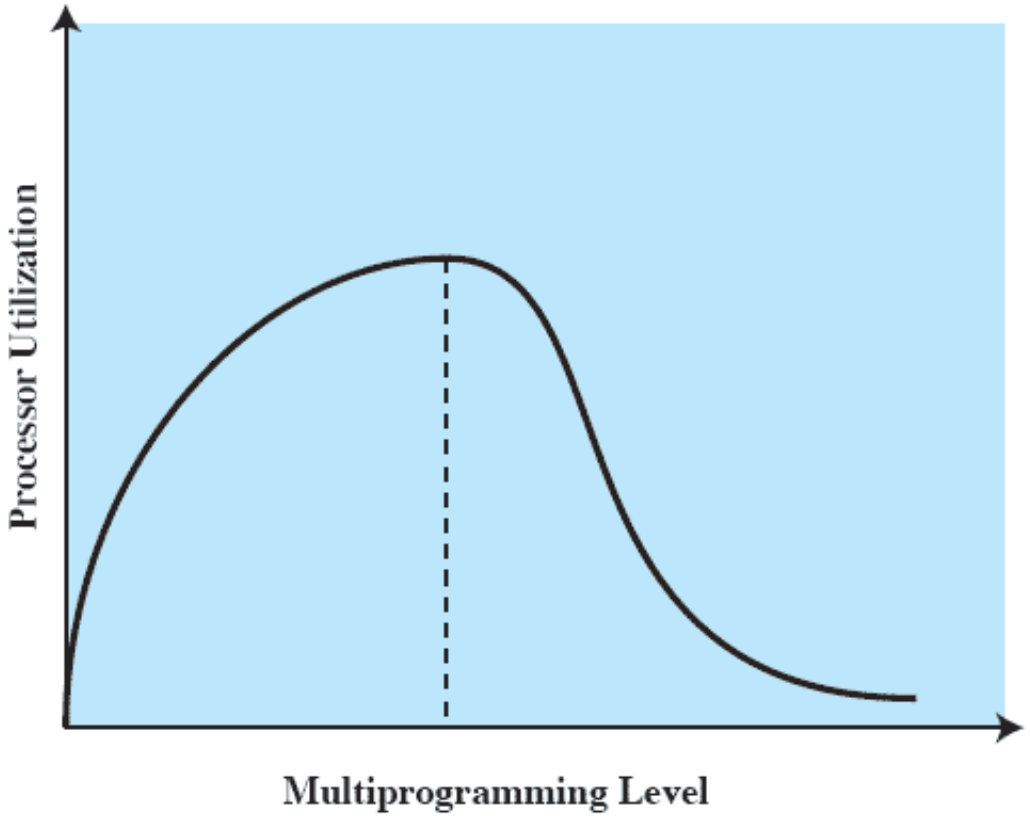


Figure 8.21 Multiprogramming Effects

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
- Last process activated
  - 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