Operating Systems: Memory Management

Memory Management

“Multitasking without memory management is like having a party in a closet”

“Programs expand to fill the memory that holds them.”

Memory Management

- Why talk about memory management?
  - Process isolation
  - Automatic allocation and management
  - Support for modular programming
  - Protection and access control
  - Long term storage

What is memory?

- Physical viewpoint (RAM)
  - Physical chip attached to the motherboard
- Programmer's viewpoint
  - Just large array of words (or bytes)
    - Unique address for each word
  - Is this correct?
    - What if compiler uses a register to store a variable?
    - What if it does not even allocate a variable?
- OS viewpoint: Both are wrong

Memory hierarchy managers

- OS handles a complex memory hierarchy of memory managers
- Heap manager (dynamic memory manager)
    - "High" level
    - Supports basic operations with memory blocks
        - Block allocation, release, changing size of allocated block

Memory hierarchy managers

- Virtual memory manager
  - Lower level than heap manager
  - Manages virtual address spaces
    - Abstracts physical addresses, allowing arbitrary addresses to be assigned to memory cells
    - Allows concurrent processes to be loaded into overlapping memory at different times
    - Lets us separate processes completely from each other
    - Requires kernel and memory manager to find location of memory unit

Memory hierarchy managers

- Virtual memory manager
    - Virtual memory
        - Extension of idea of virtual address space
        - Any memory cell can reside in both main memory and/or hard disk
        - Allows for almost unlimited amounts of memory to be allocated
            - Process allowed 2GB address space on some versions of Windows
                - Though you might never see it taking up 2GB of RAM

Physical view

- Processor does not interact directly with memory or hard disk
  - Works through memory controller and disk controller
- System memory
  - Based on Dynamic RAM, relatively slow, requires periodic refresh
    - 30 to 80, or even 250 cycles of CPU time
- Cache memory
  - Higher speed memory closer to the CPU
  - Usually implemented as static RAM (more expensive, faster)
  - Invisible to programmer

Physical view

- Cache memory
  - Contents cannot be directly read/modified
  - Cache controller
    - Manages cache memory instead of CPU
    - Responsible to accumulate important data and clear out old data
    - Often integrated into CPU
  - L1 cache: Near cpu speed, often read in 2-3 cycles
  - L2 cache: Access times in range of 9-15 cycles
    - Contains data flushed out of L1 cache (victim cache)

Addresses

- Address binding
  - Binding: Mapping from one address space to another
  - Program must be loaded into memory before execution
    - Loading of processes may result in relocation of addresses
    - Link external references to entry points as needed
  - User process may reside in any part in memory

Addresses

- Programs have symbolic addresses in the source programs
  - `int i = 0;`
- Compiler binds symbolic addresses to relocatable addresses
  - Usually assumed to start at 0
- Linkage editor (loader) binds relocatable addresses to absolute addresses

Binding

- Types
  - Compile time binding
    - Binding of absolute addresses by compiler
    - Possible only if compiler knows where program will be in memory
  - Load time binding
    - Needs relocatable code generated by compiler
    - Final binding delayed until code is loaded
    - If change of starting address, have to reload code

Binding

- Types
  - Execution time binding
    - Process may be moved to different addresses during execution
    - Need to delay binding until code is actually being run
    - Requires special hardware

Relocation

- Compiler works with assumed logical address space when creating object module
- Relocation: Adjustment of operand and branch addresses within program to reflect actual address space
- Types:
  - Static relocation
  - Dynamic relocation

Static Relocation

- Static relocation often done through separate linkage editor and loader
  - Starting address not required to be known
  - Absolute physical addresses bound only at time of loading
  - Relocatable physical addresses bound by relocating complete module
  - Program gets relocated twice
    - Once for linking and another for loading

Dynamic Relocation

- Dynamic relocation: Modules kept on disk in relocatable load format
  - Binding of physical addresses delayed to the very last possible moment
    - Every time a storage reference is made
  - Invisible to all but system programmers
  - Forms the basis for virtual memory
  - Permits efficient use of main storage

Dynamic Relocation

- What about when a routine needs to call another routine?
  - Could be out of main memory, have to check
  - If out of memory, load it
    - Unused routine is never actually loaded
  - Can save us on memory requirements for infrequently used code

Memory alignment

- When reading from memory, getting the whole block
  - Not just one (or set of) bytes
- Systems can have unaligned loads and unaligned stores
- Unaligned stores could overwrite data adjacent to target
  - Problem can be avoided by extra overhead
- Bus error vs segmentation fault

Requirements

- Process address space
  - Processes need to run in a private address space
  - User mode
    - Process refers to private stack, data, code areas
  - Kernel mode
    - Kernel data and code areas, uses different private stack
  - Processes may need access to shared address space
    - Explicit requests like shared memory
    - Could be done automatically by kernel to reduce memory usage

Requirements

- Relocation
  - Available main memory must be shared amongst processes
  - Programmer may not know what other programs are resident in memory while their code is executing
  - Processes get swapped in and out to maximize cpu utilization
    - These processes may not be swapped back to the same spot in memory (process got *relocated* to a new part)
  - Of course all memory references need to be resolved to correct addresses

Requirements

- Protection
  - Processes need to be protected against unwanted interference
  - Made harder as relocation because a process's location in memory is unpredictable
  - Impossible to check absolute addresses at compile time
  - Have to deal with dynamic memory allocation through pointers
  - Need to make sure cannot access data or code of OS
  - Need hardware support to protect processes from interfering with one another

Requirements

- Logical organization
  - Memory needs to have some logical organization (usually linearly)
  - Programs written as modules that can be written and compiled independently
- Sharing
  - Need the capability for processes and the OS to share memory as needed/required
  - Could allow each process to access same copy of a program
    - Rather than needing separate copies

Requirements

- Physical Organization
  - Cannot leave programmer with responsibility to manage memory
  - Might not have enough memory for program plus data
  - Programmer does not know how much total space is available.

Memory Management Schemes

- Always have a shortage of main memory
  - Applications grow to fill the memory allocated to them
  - Might need several active process at once

Memory Management Schemes

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

Fixed Partitioning

- Simplest memory management scheme for multiprogrammed systems
- Divide memory into fixed size *partitions*, possibly of different size
    - Fixed at system initialization, cannot be changed during operation

Fixed Partitioning

- Single-Partition Allocation
  - User is provided with a bare machine
  - User has full control of entire memory space
  - Clearly not practical, but has many advantages

Fixed Partitioning

- Single-Partition Allocation
  - Advantages:
    - Maximum flexibility, use memory as you want
    - Maximum simplicity
    - Minimum cost, no special hardware

Fixed Partitioning

- Single-Partition Allocation
  - Disadvantages:
    - No services
    - OS has no control over interrupts
    - No mechanism to process system calls or errors
    - No space to provide multiprogramming

Fixed Partitioning

- Two-Partition Allocation
  - Memory divided into two partitions
    - Resident operating system
    - User memory area
  - OS placed in low memory or high memory depending on location of interrupt vector
  - OS code and data can be protected by hardware
    - base-register, limit register

Multiple-Partition Allocation

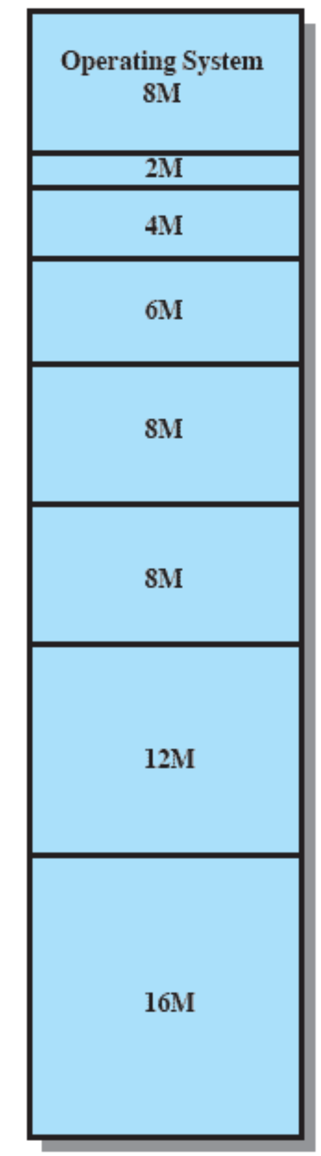
- Multiple-Partition Allocation
  - Necessary for multiprogrammed systems
  - Allocates memory to various processes in wait queue to be brought into memory
- Simplest scheme:
  - Divide memory into a large number of fixed-size partitions
  - One process to each partition
  - Degree of multiprogramming bounded by number of partitions
  - Partitions allocated to processes and released upon termination

Multiple-Partition Allocation

- Problems:
  - Program may not fit in a partition
  - Main memory use is inefficient
    - Any program, no matter how small, occupies entire partition
    - Called *internal fragmentation*

Multiple-Partition Allocation

- Can try and solve it with unequal sized partitions
  - Can fit at least a limited number of larger programs
  - Smaller programs can be placed in smaller partitions
    - Less internal fragmentation



(b) Unequal-size partitions

Placement Algorithm

- How do we determine where processes go?
- For equal-sized, no options so trivial
- For unequal size
    - Want to assign them in such a way as to minimize internal fragmentation
    - Can assign each process to smallest partition within which it can fit
    - Can even have queues for each partition

Placement Algorithm

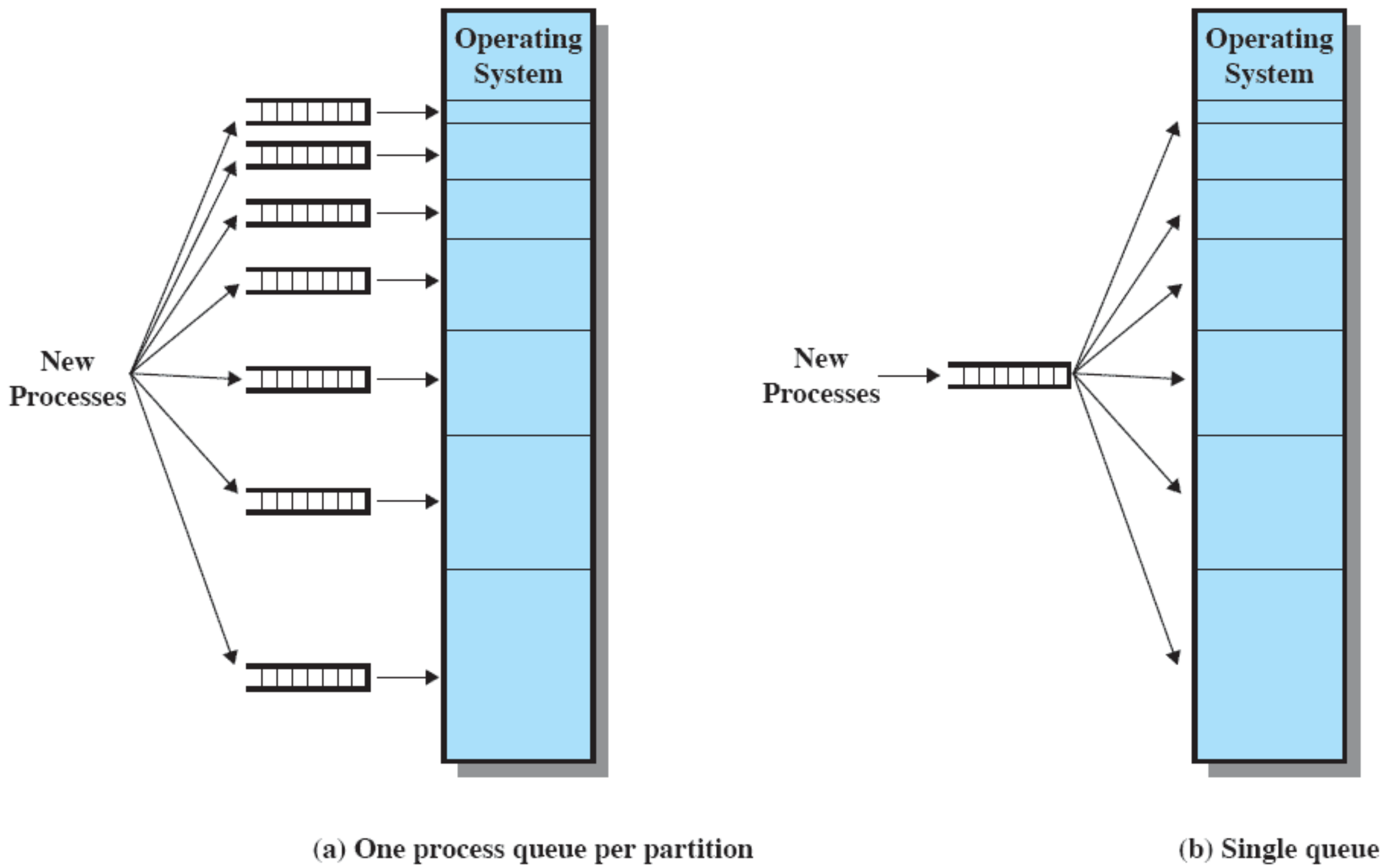


Figure 7.3 Memory Assignment for Fixed Partitioning

Multiple-Fixed Partition

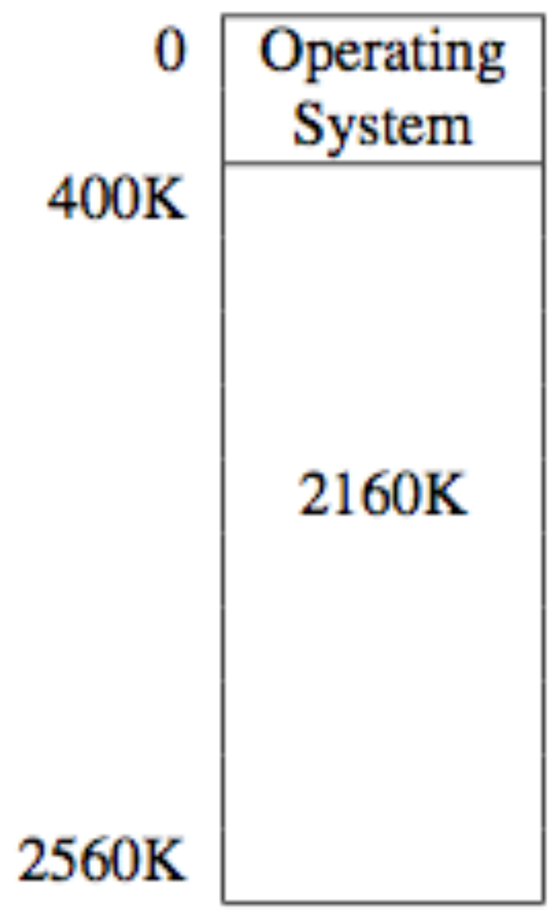
- Still left with some problems
  - Number of active processes limited by system
    - Limited by pre-determined number of partitions
  - Large number of small processes use the space inefficiently
    - In either fixed or variable length partition methods

Dynamic Partitioning

- Variable size partitions
- Basic implementation
  - Keep a table indicating availability of various memory partitions
  - Any large block of available memory is called a *hole*
  - Initially entire memory is identified as one large *hole*
  - When a process arrives, allocation table is searched for a large enough hold
    - If available, that hole is allocated to process

Dynamic Partitioning

- Lets look at an example:



Job Queue

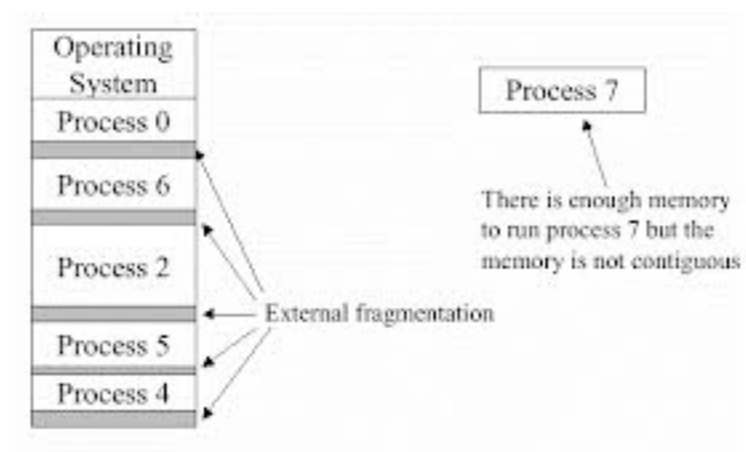
Process	Memory	Time
p_1	600K	10
p_2	1000K	5
p_3	300K	20
p_4	700K	8
p_5	500K	15

Dynamic Partitioning

- Will have holes of various sizes scattered throughout the memory
- Holes can grow when jobs in adjacent holes are terminated
- Holes can also diminish in size if many small jobs are present

External Fragmentation

- Dynamic partitioning often results in fragmentation (External)
  - Division of main memory into small holes not usable by any process
  - Enough total memory exists to satisfy request, but is fragmented across many small holes
  - Caused by mismatch between size of memory request and size of the memory area allocated to satisfy that request
  - Can cause large jobs to starve



Compaction

- One way to handle external fragmentation is with *compaction*
  - Move processes so they are contiguous
  - Shuffle memory contents to place all free memory into one large hole
  - Only possible if system supports dynamic relocation at exec time
  - Very CPU-intensive, can break code that works with pointers

Dynamic Partitioning

- How does OS decide which free block to allocate to a process?
- Many schemes for doing this:
  - Best-fit
  - First-fit
  - Worst-fit
  - Next-fit
  - Buddy's System

Dynamic Partitioning

- Best-fit
  - Allocate the smallest hole that is big enough
  - Entire list of holes needs to be searched
    - Or at least keep list of holes sorted by size
  - Since smallest block is found for process, smallest amount of fragmentation is left
    - Sounds good, actually worst performer overall!
    - Not many people will be able to actually use that small hole
  - Need to do memory compaction more often

Dynamic Partitioning

- First-fit
  - Allocate first hole that is big enough
    - Scan memory from beginning and choose first available block large enough
  - Fastest strategy

Dynamic Partitioning

- Worst-fit
  - Allocate largest available hole
  - Requires sorted order of largest hole to smallest
  - Creates largest fragment possible
  - If a process requiring larger memory arrives later, it cannot be put anywhere
    - Largest hole was already split up and occupied
  - Poor performer overall

Dynamic Partitioning

- Next-fit
  - Modified version of first fit
  - Scans memory from location of the last placement

Allocation

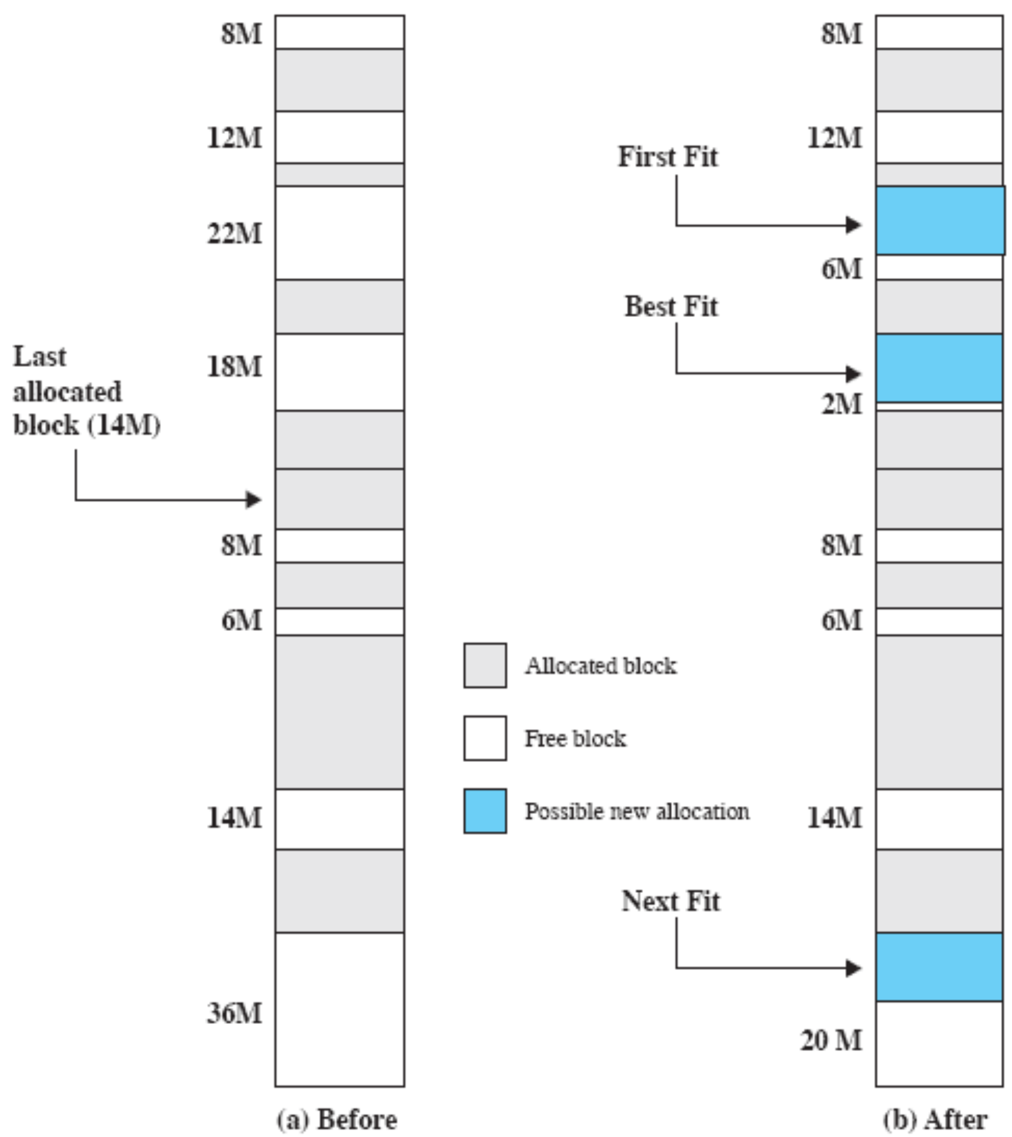


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Buddy System

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

Buddy System

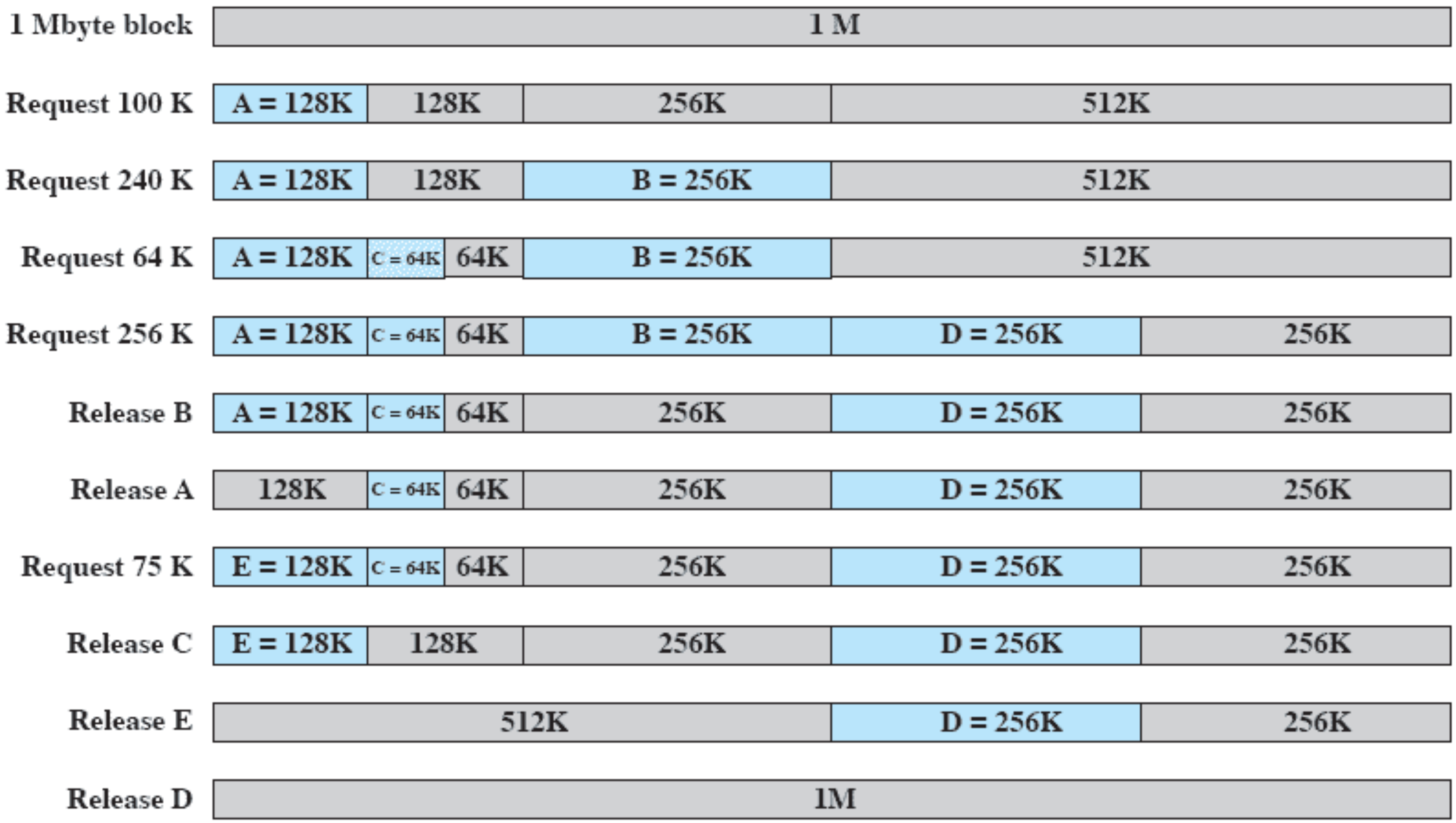


Figure 7.6 Example of Buddy System

Buddy System

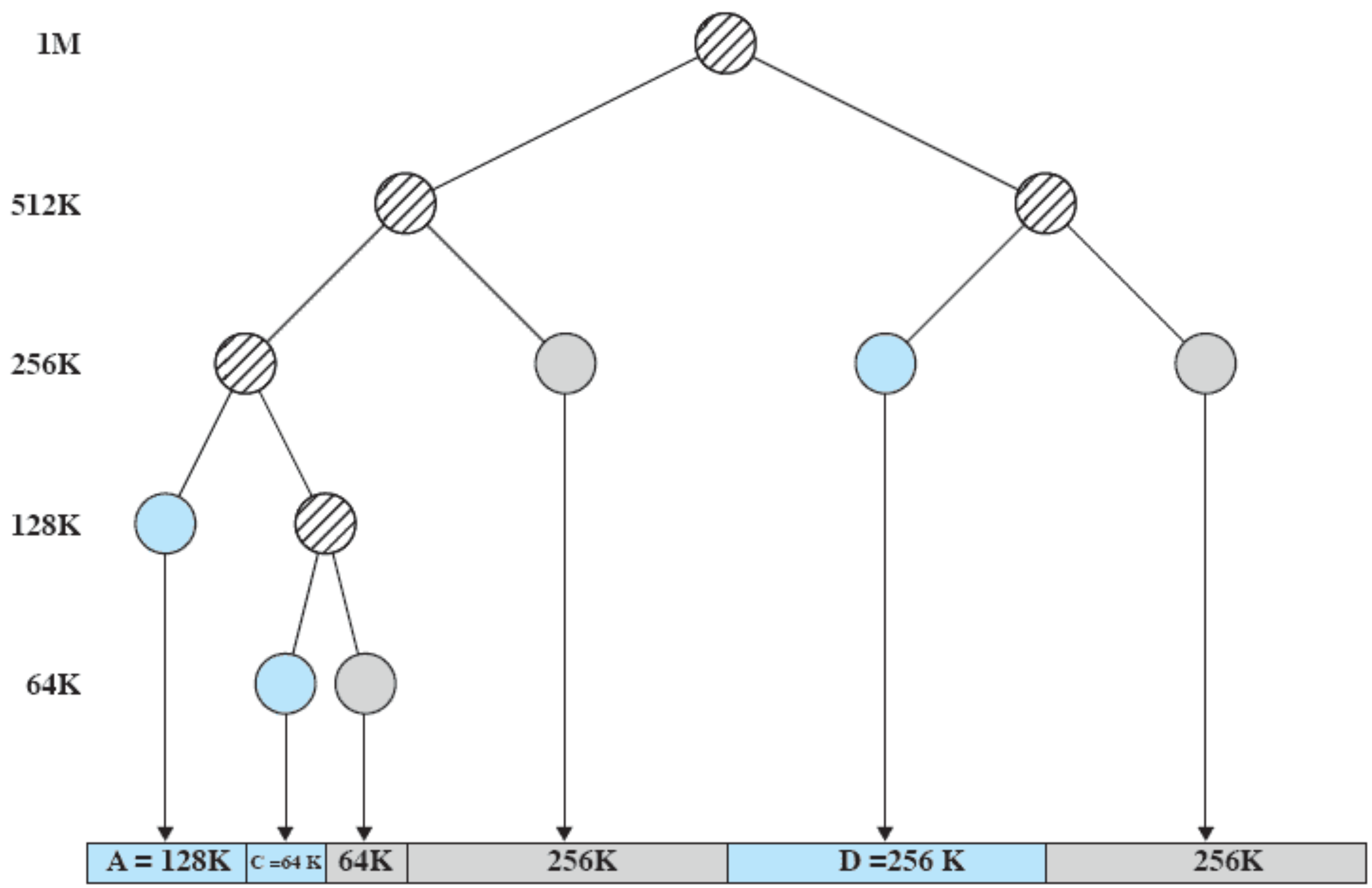


Figure 7.7 Tree Representation of Buddy System

Paging

- So far processes have been contiguous in memory
- Lets let processes memory be noncontiguous
- Paging is one way to do this
    - Partition memory into small and equal fixed-size chunks
    - Divide each process into the same size chunks
    - The chunks of a process are called *pages*
    - The chunks of memory are called *frames*

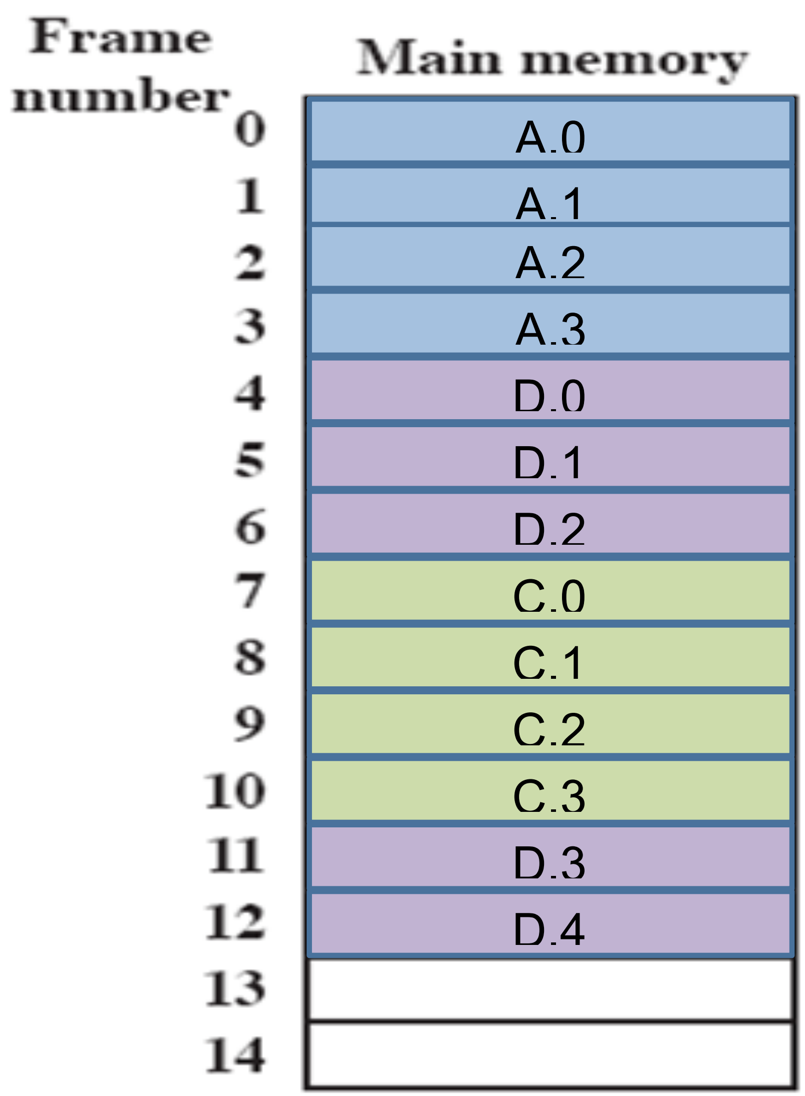
Paging details

- Physical memory divides into *page frames*
  - Frames that can hold pages (size of frames 1k or even 8k bytes)
  - `getpagesize(3C)` can get it on unix system
- Logical memory broken into blocks of same size as *page frame* size
  - Called *pages*
- To execute a process, its pages are loaded into frames

Paging details

- Every address generated by CPU now divided into two parts
  - Page number p
  - Page offset *d*
- Page number used as index into a *page table*
  - This table contains base address of where each page is in memory
  - Page offset defines address of the location within a particular page

Paging



Frame number
Main memory
0
A.0
1
A.1
2
A.2
3
A.3
4
D.0
5
D.1
6
D.2
7
C.0
8
C.1
9
C.2
10
C.3
11
D.3
12
D.4
13
14

Paging

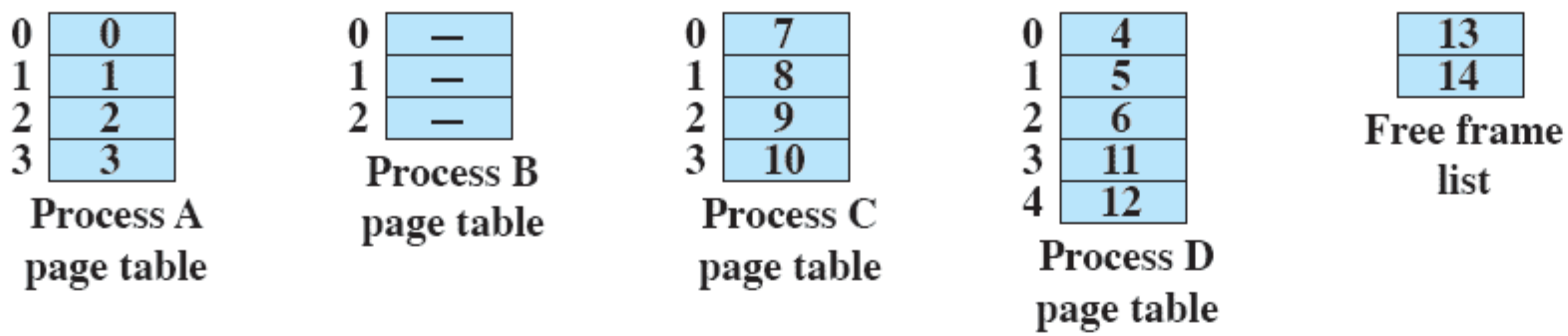


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Paging

- No external fragmentation possible!
- Internal fragmentation is possible
  - Average of half a page per process
- Paging can allow programs larger than main memory to be executed
  - Only bring pages into memory when they are needed (*demand paging*)
- Size of a page has tradeoffs
  - Small page size means more overhead in page table plus more swapping
  - Large page size has more internal fragmentation

Shared pages

- Possible to share common code with paging
    - Easy way would be to simply have same page in two process's tables
- Must be reentrant (pure code)
    - Code cannot modify itself and local data of each user must be kept in separate space (separate data pages for each process)
    - Would have two parts:
        - Permanent part is the instructions that make up the code
        - Temporary part contains memory for local variables for use by code
    - Each execution of permanent part creates a temporary part
        - *activation record* for the code

Shared pages

- For a function to be classed as reentrant:
  - All data is allocated on the stack (no global variables)
  - May not modify its own code
  - Functions don't call any other function that is not reentrant
- Compilers, text editors, windowing systems, unix kernels all use reentrant code

Segmentation

- User often views memory as collection of variable-sized segments
  - Arrays, functions, procedures, main program
- Note no necessary order to these
- Length of each segment could also be different
  - Depending on purpose for each program

Segmentation

- Lets divide our programs into segments
  - Each segment can vary in length
  - Do have maximum segment length
- Address would consider of
  - A segment number (*name*)
  - Segment *offset*
- In some ways similar to dynamic partitioning

Segmentation

- Logical address space is considered to be collection of segments
  - Each with a name and length
- Mapping between logical and physical addresses use a *segment table*
- Each entry in segment table is made up of
  - Segment *base*
  - Segment *limit*
- Segment table can be abstracted as array of *base-limit register pairs*

Segmentation

- Segment name/number s:
  - Used as a index into the segmentation table
- Segment offset d:
  - Added to segment base to produce physical address
  - Must be between 0 and the segment limit
  - Attempting to go past this limit results in trap to OS

Sharing

- Segments represent a semantically defined portion of a program
- Can share these parts, and in fact a bit easier than paging
  - Why?
- Share a particular function between two programs

Fragmentation

- Memory allocation becomes a dynamic storage allocation problem
- Possibility of external fragmentation
    - All blocks of memory left are too small to accommodate a segment
- Compaction can be used whenever needed
    - Segmentation relies on relocatable code
- External fragmentation is dependent on average size of segments

Logical to physical addresses

- Processes see their logical addresses
- Must map logical addresses to physical addresses
- Trivial for partitioning
- For paging or segmentation, must use appropriate table

Logical Addresses

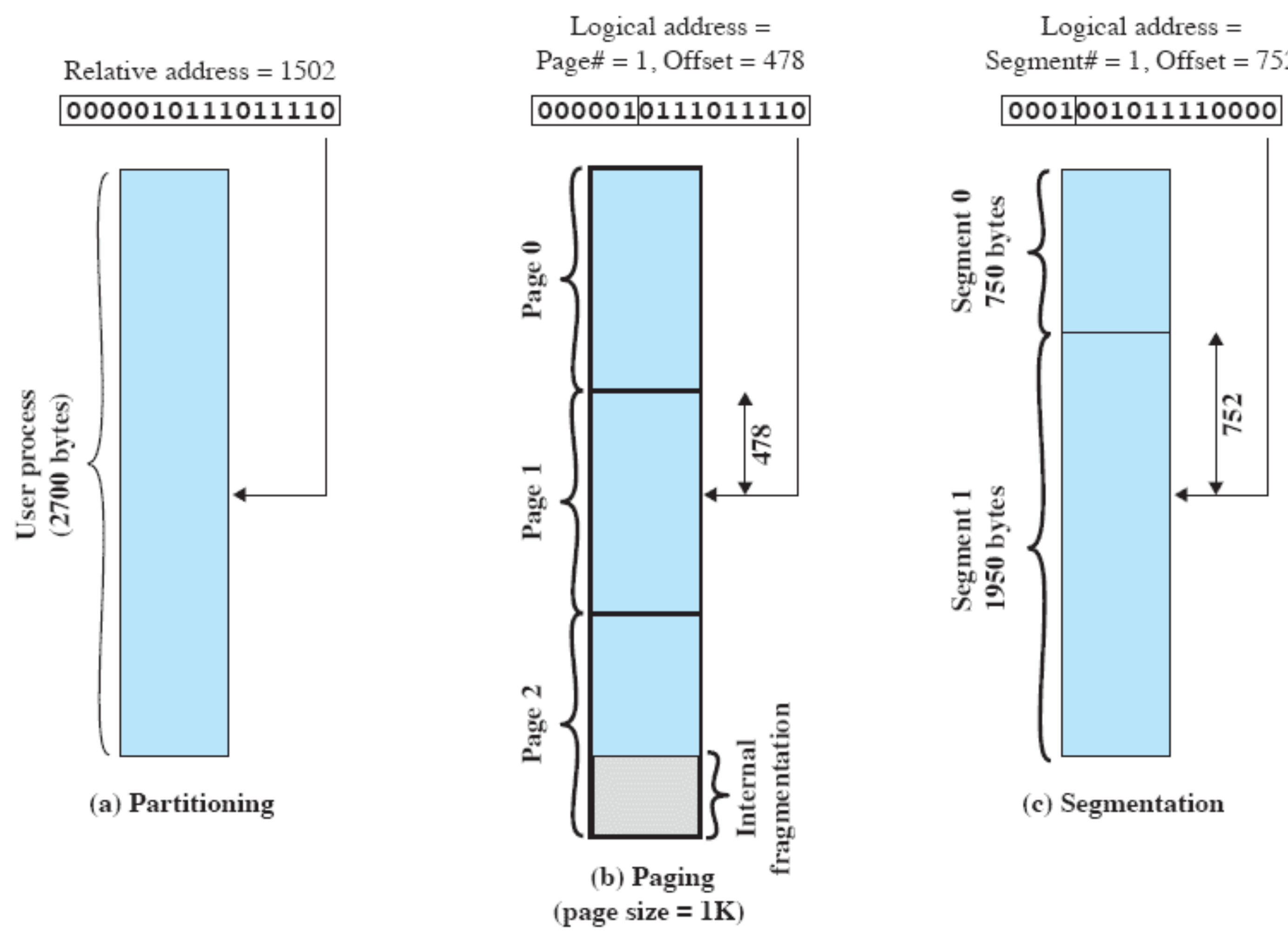
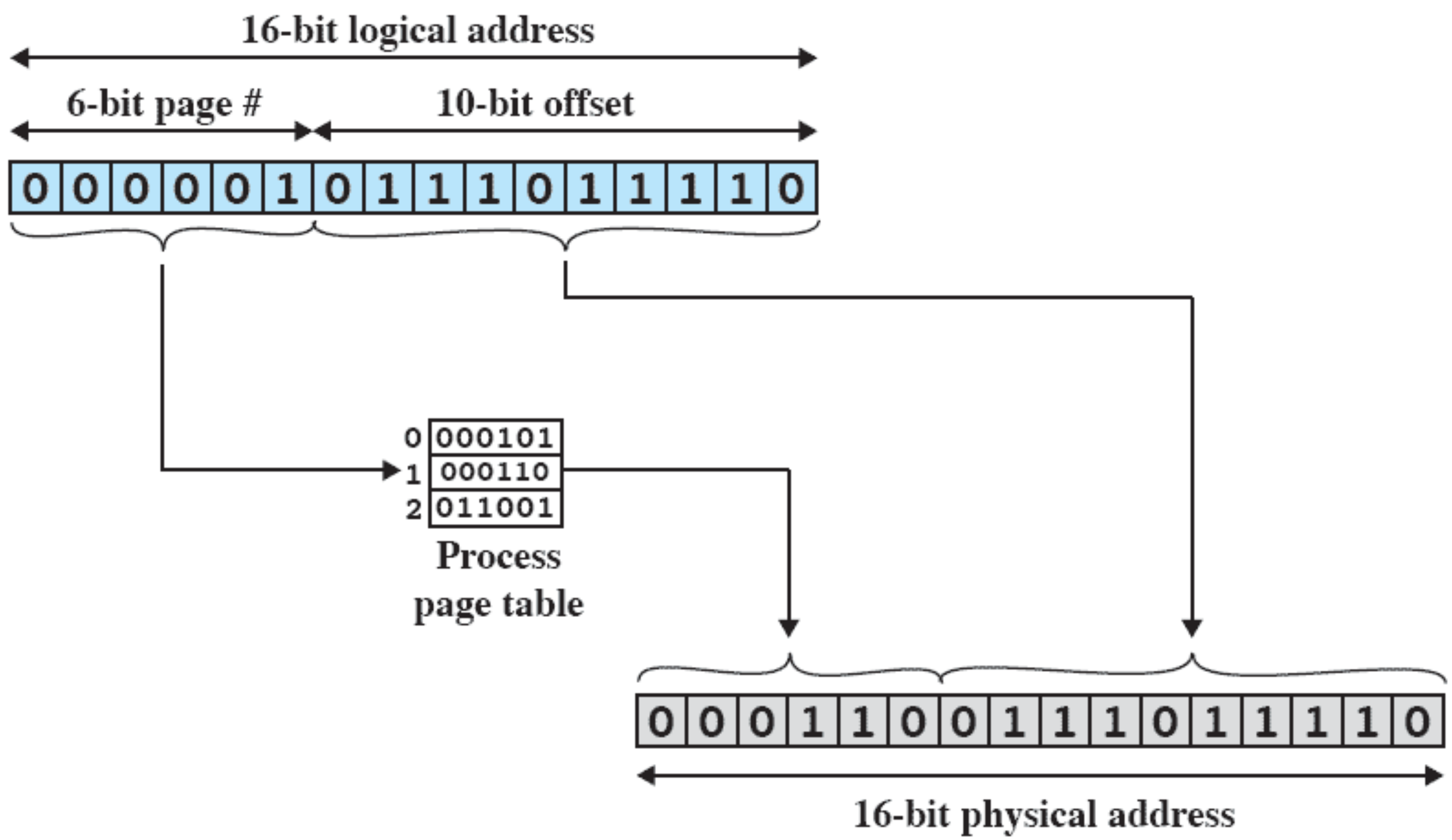


Figure 7.11 Logical Addresses

Logical Addresses



(a) Paging

Logical Addresses

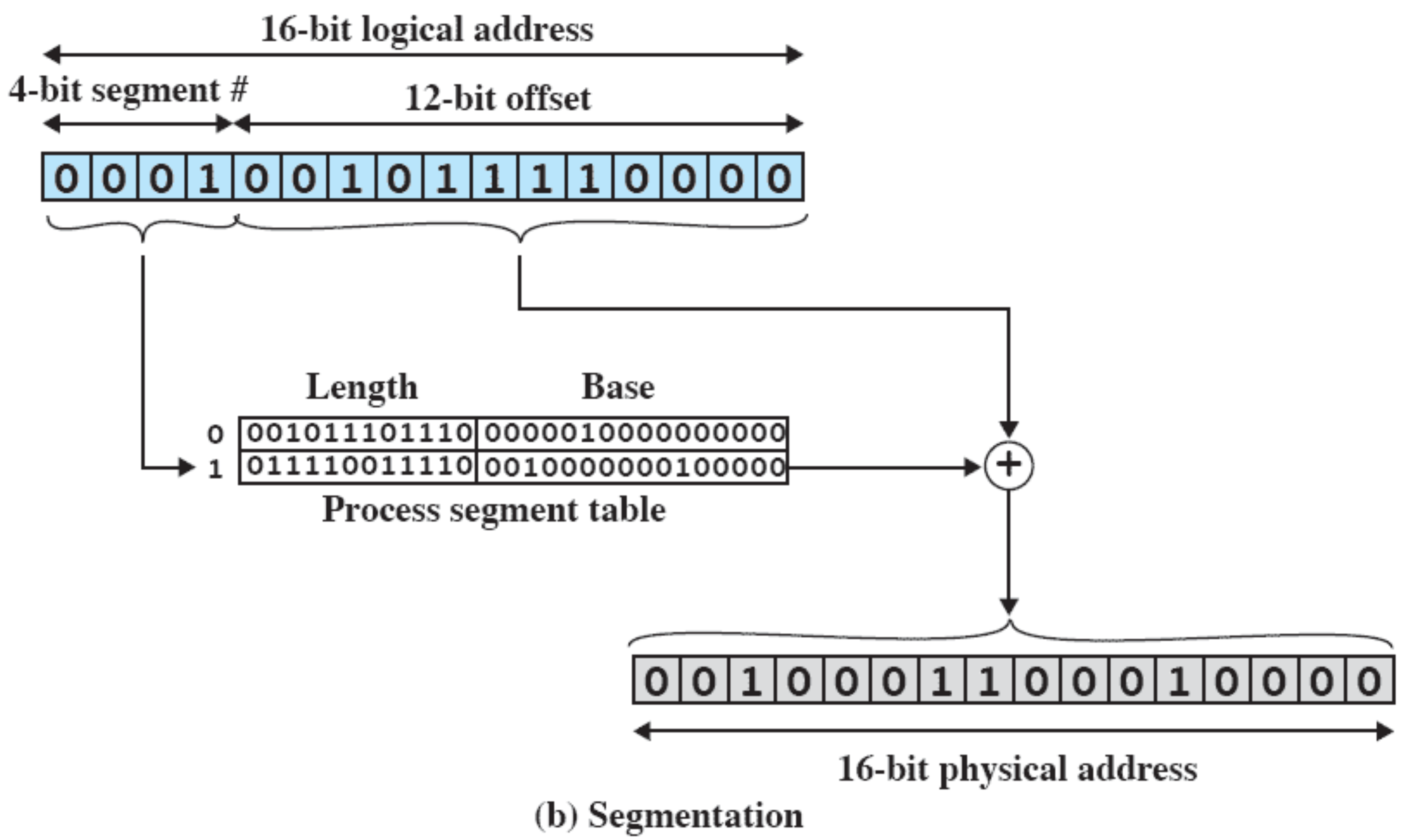
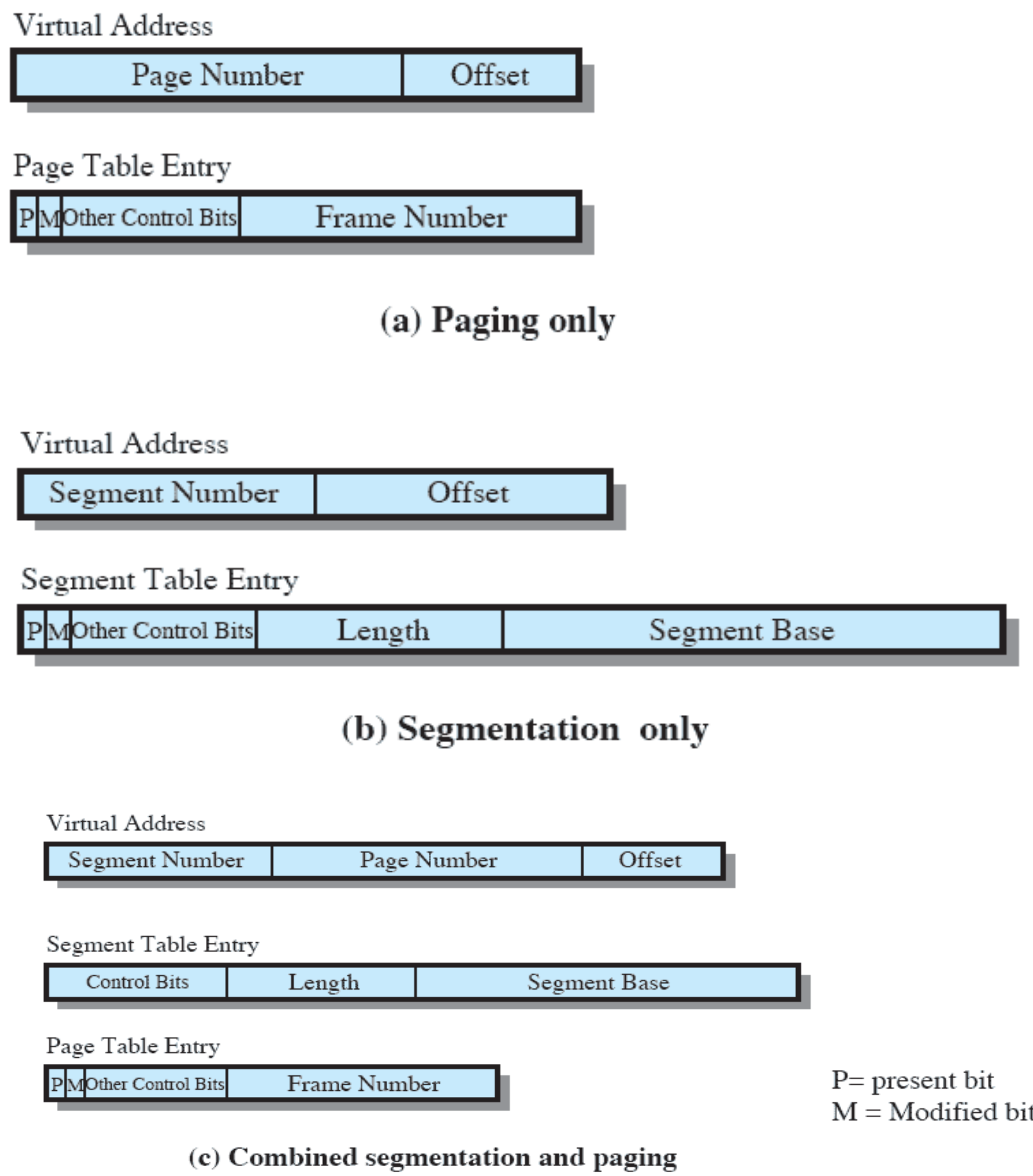


Figure 7.12 Examples of Logical-to-Physical Address Translation

Memory Management Formats



Virtual memory

- Memory management has some key points
  - Memory references are logical addresses dynamically translated into physical addresses at run time
  - A process can be swapped in and out of main memory, occupying different regions at different times during execution
  - A process does not need to be contiguous

Virtual memory

- If all those conditions are present, then it is not necessary that all pages or all segments of a process be in main memory
- As long as next instruction or next data are in memory, execution can proceed
  - At least for a time

Address Translation

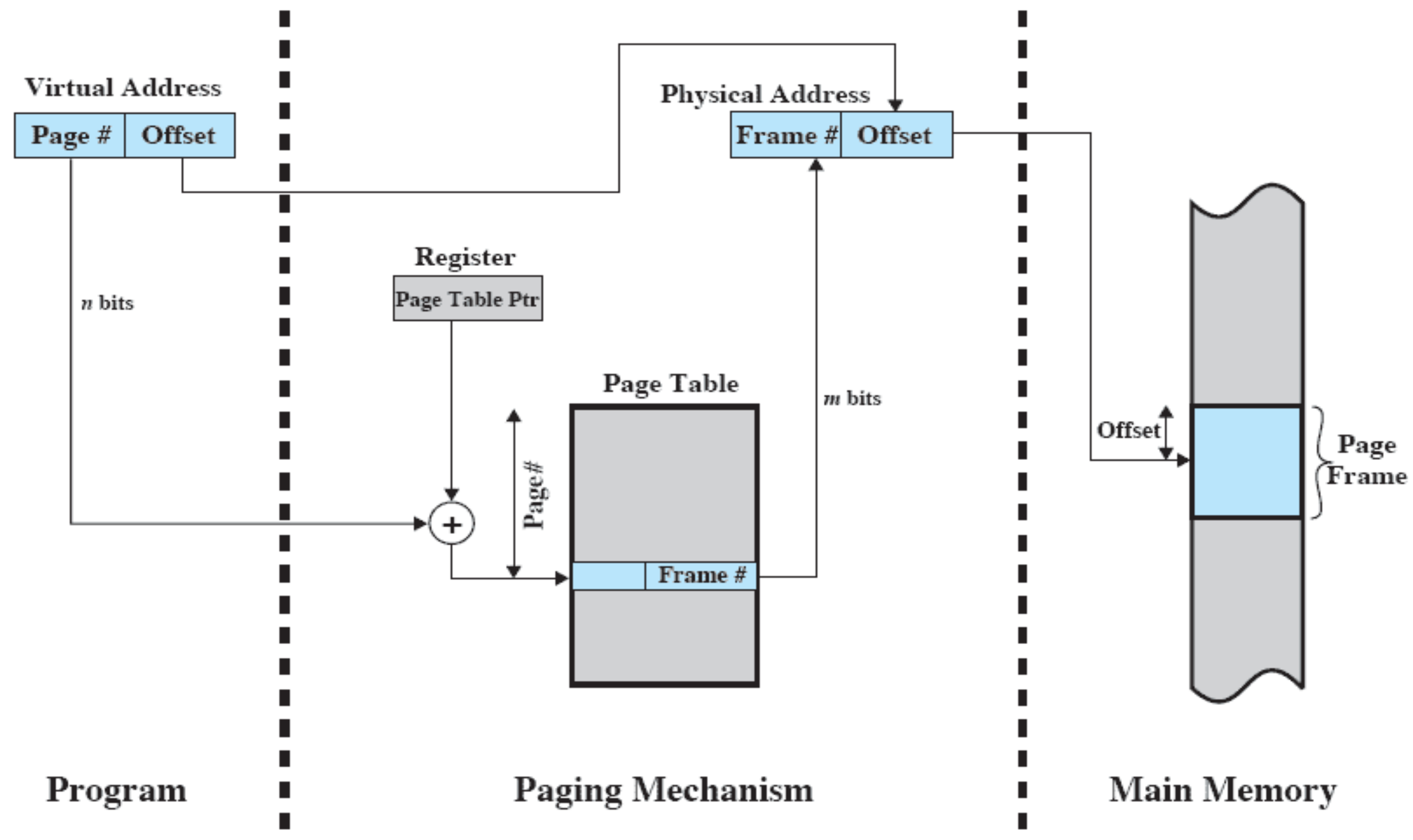


Figure 8.3 Address Translation in a Paging System

Virtual memory

- Process execution in virtual memory:
    - Operating system brings into main memory a few pieces of a program
    - *Resident set*: portion of a process that is in main memory
    - An interrupt is generated when addresses is needed that is not in main memory
        - Operating system places that process in a blocked state

Virtual memory

- 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 this I/O takes place
    - An interrupt is issued when disk I/O completes
    - System then places the process back into Ready state

Implication of virtual memory

- More processes can be maintained in main memory
  - We only load some of the pieces of each
  - With so many processes in main memory, it is likely at least one process will always be in Ready state
- A process may be larger than 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, operating system needs to try and guess which pieces are least likely to be used in the future
  - Guess should probably be based on recent history
  - Why?

Replacement Policy

- Need to reduce thrashing by using a replacement policy
- When all frames in main memory are occupied, need to bring in new page
- Replacement policy determines which page is replaced

Replacement Policy

- Which page is replaced?
  - Page removed should be page least likely to be referenced in future
  - Principle of locality
- Most replacement policies predict future behavior based on past behavior

Replacement Policy

- Frame locking : Additional complication we must allow
  - If frame is locked, it may not be replaced
    - Kernel of operating system
    - Key control structures
    - I/O buffers
    - Would need a lock bit with each frame

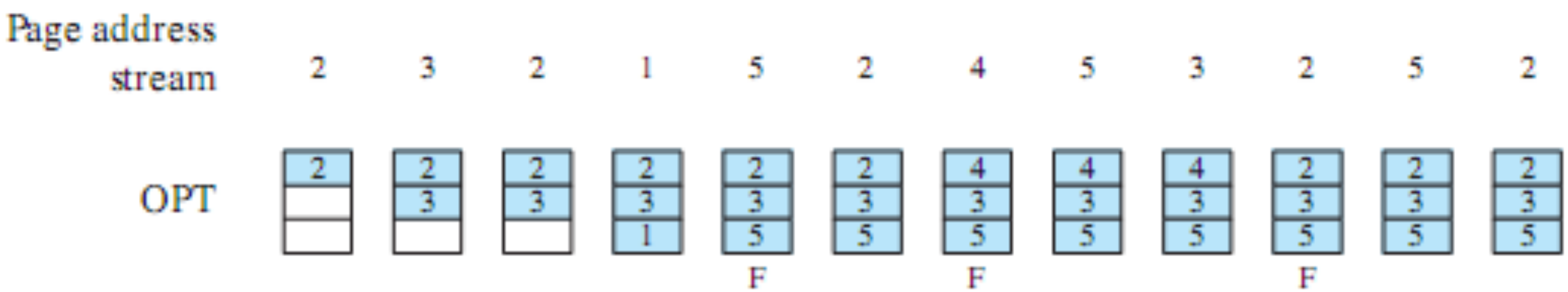
Replacement Policy

- Some basic algorithms used to handle page replacement
  - Optimal
  - Least recently used (LRU)
  - First-in-first-out (FIFO)
  - Clock

Examples

- To go over examples, consider a page address stream formed by executing the program
  - 2 3 2 1 5 2 4 5 3 2 5 2
  - First page reference is 2, second page referenced is 3, and so on
- Optimal policy
  - Selects for replacement page for which time to next reference is the longest
  - Requires perfect knowledge of future events

Optimal Policy



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

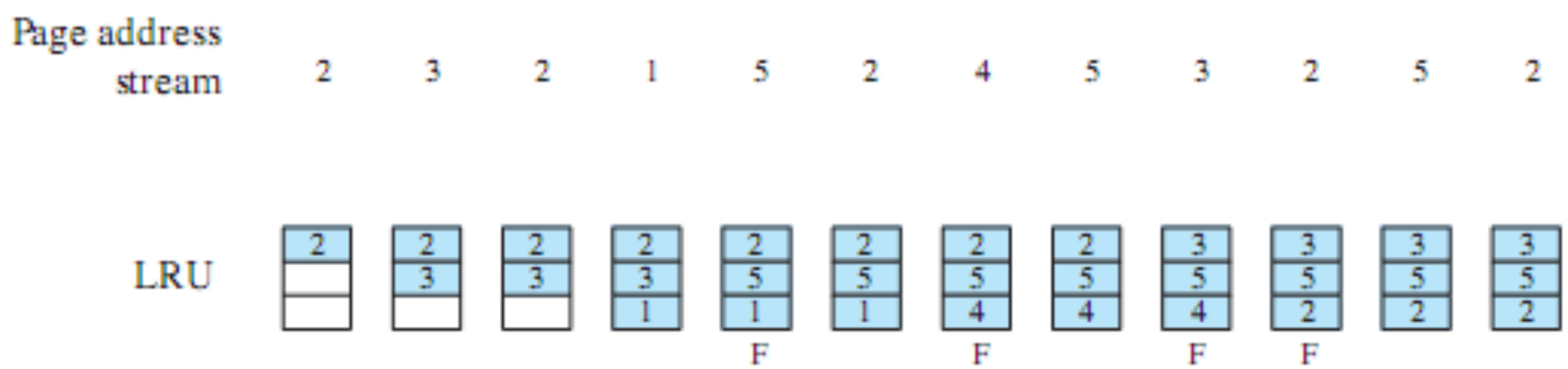
Figure 8.15 Behavior of Four Page Replacement Algorithms

- 3 page faults after frame allocation has been filled

Least Recently Used Policy

- LRU replaces page that has not been referenced for longest time
- By principle of locality, should be page least likely to be referenced
- Expensive to implement
  - One approach is to tag each page with time of last reference
  - Requires expensive overhead each time a page is accessed

LRU Policy



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

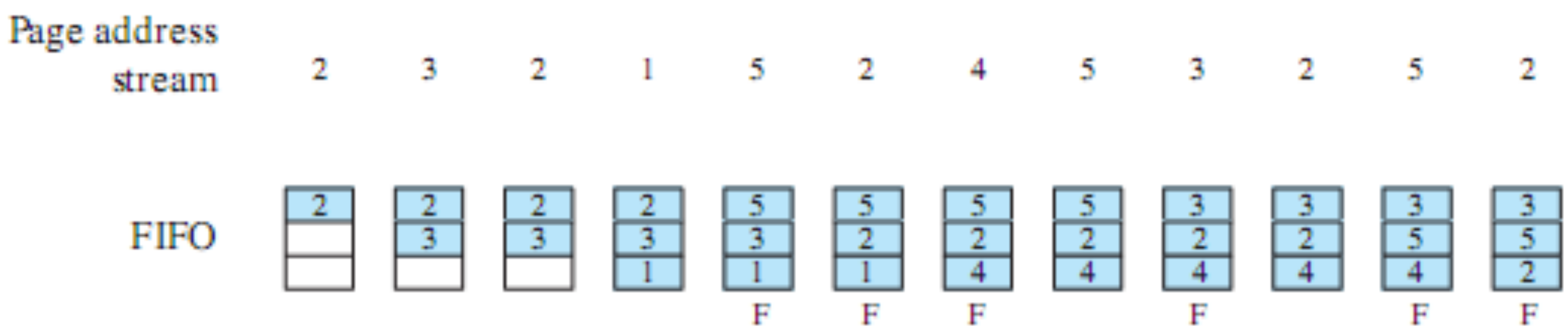
Figure 8.15 Behavior of Four Page Replacement Algorithms

- 4 page faults after frame allocation has been filled

First in, First out

- FIFO treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - Very simple to implement
- Page that has been in memory the longest is replaced
  - These pages may be needed again very soon though
  - For example, if used over and over for a long time
    - Which can be quite likely under some situations

FIFO Policy



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

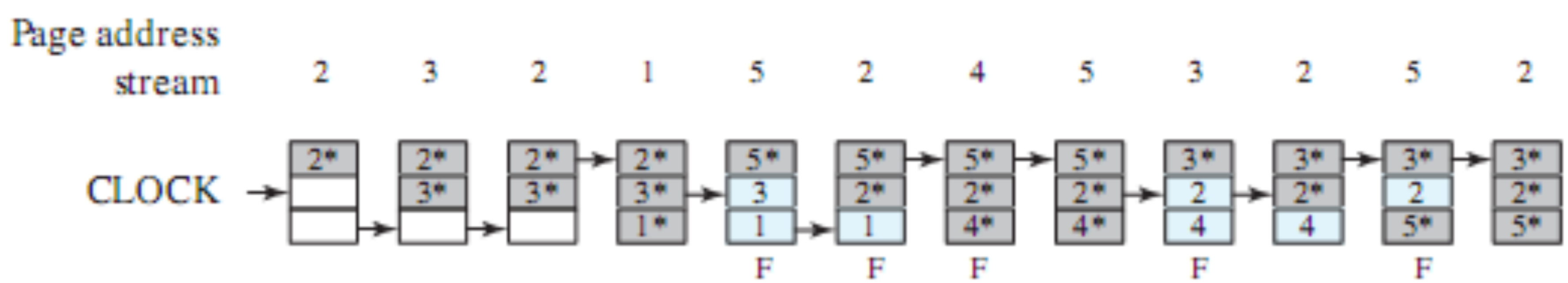
Figure 8.15 Behavior of Four Page Replacement Algorithms

- 6 page faults after frame allocation has been filled
  - LRU recognized that pages 2 and 5 are referenced more frequently

Second-Chance

- Main problem with LRU is that it is expensive to implement
- Lets try and "approximate" LRU
  - Use additional bit called a "use bit"
  - When page is first loaded in memory or referenced, use bit is set to 1
  - When it is time to replace a page, OS scans set flipping all 1's to 0's
  - First frame encountered with use bit already set to 0 is replaced
- Also called Clock policy

Clock Policy

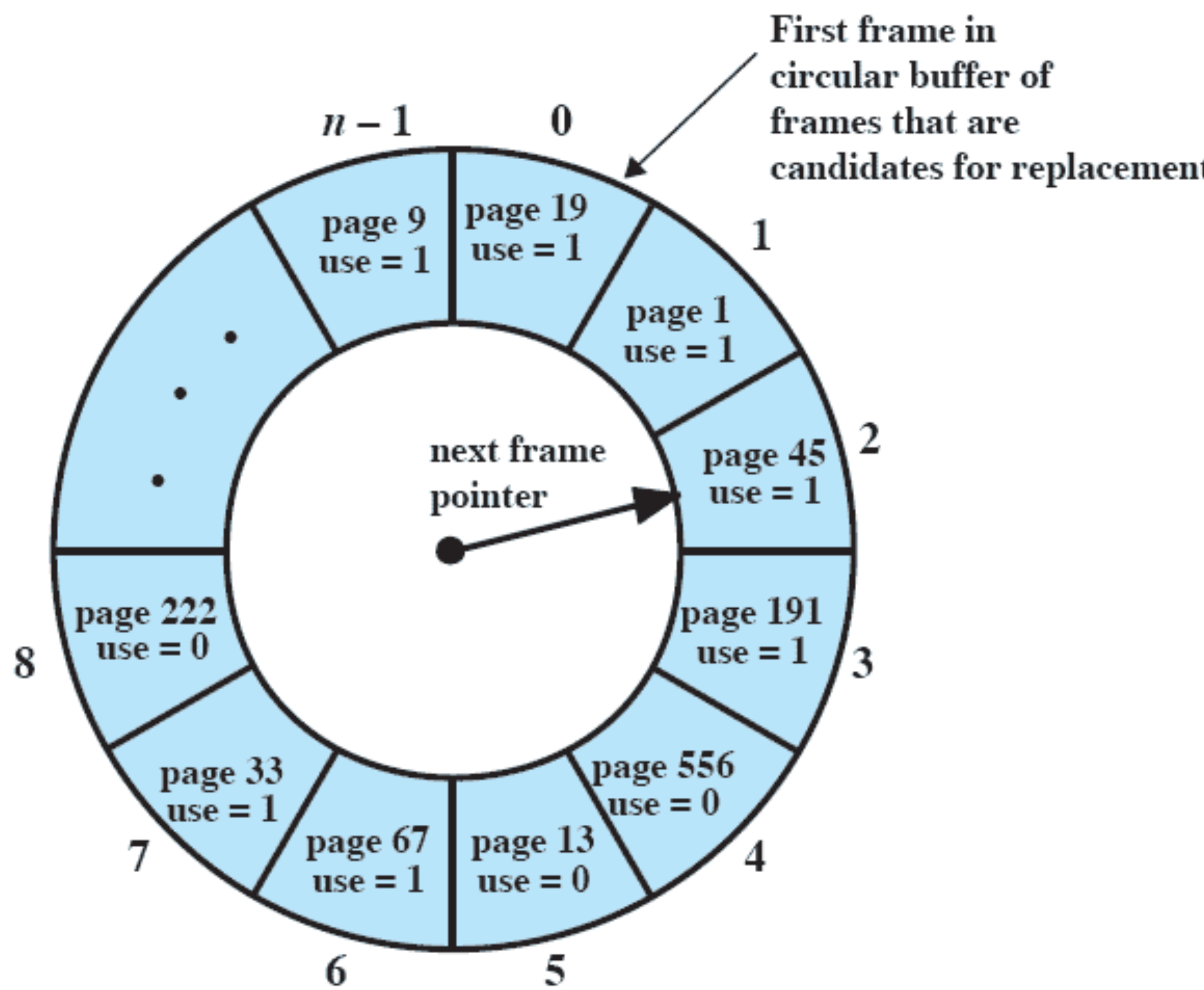


F= page fault occurring after the frame allocation is initially filled

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

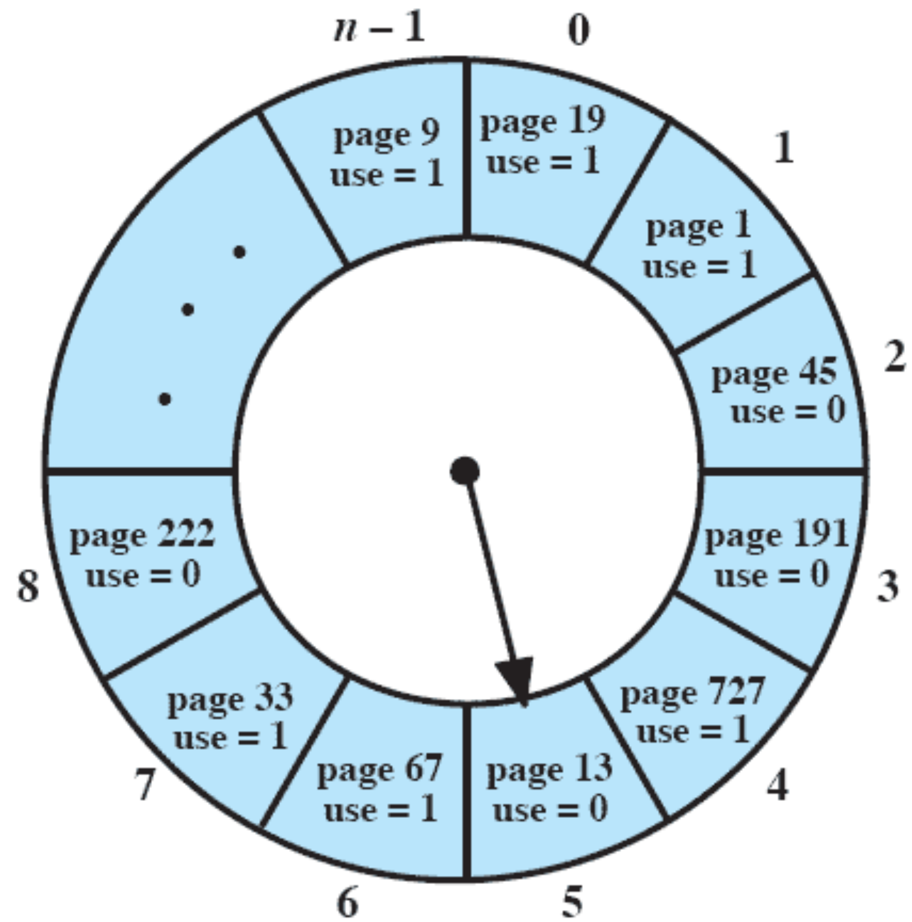
- 5 page faults after frame allocation has been filled
    - Clock policy protected 2 and 5

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

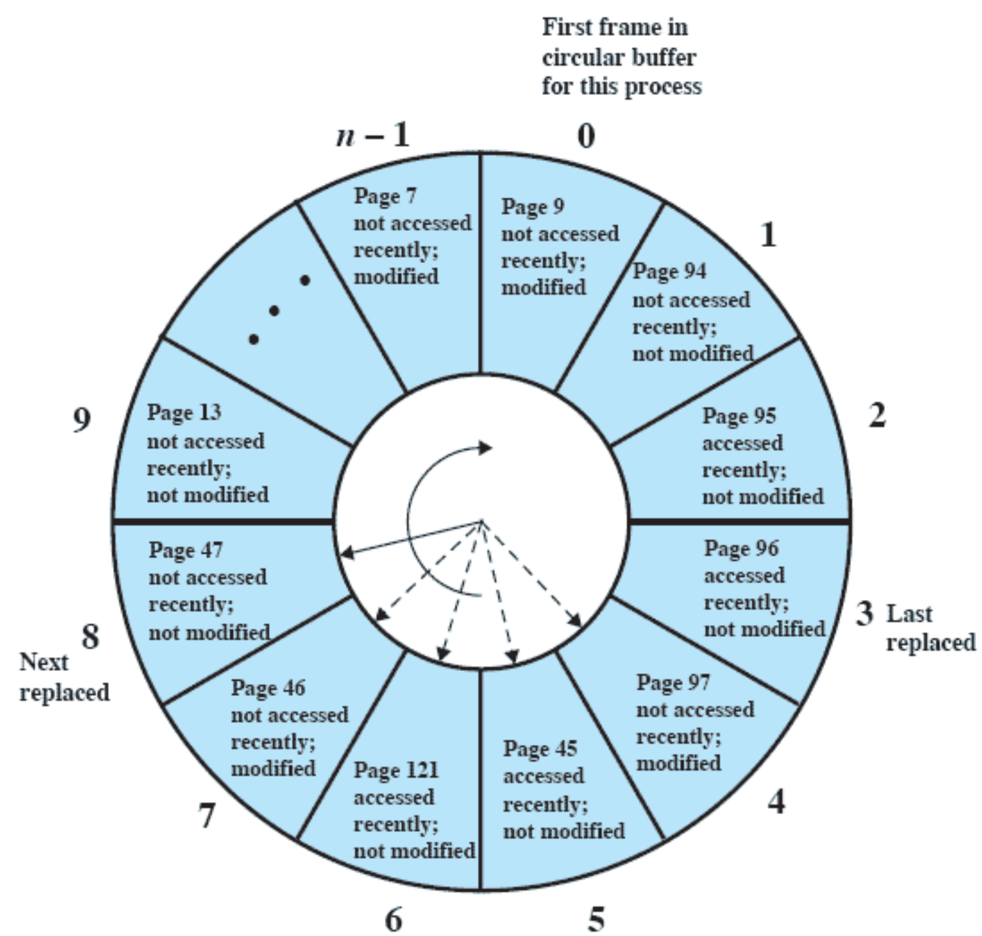


Figure 8.18 The Clock Page-Replacement Algorithm [GOLD89]

Additional-Reference-Bits

- Can try to do something with a bit more accuracy
  - Keep 8-bit byte for each page in page table
- At regular intervals, shift the reference bits by one
- This register contains history of page references for last 8 time periods
- Page with lowest number is the LRU page
- 11000100 used more recently than 01110111
- Numbers not guaranteed to be unique
  - Is this a problem?

Comparison

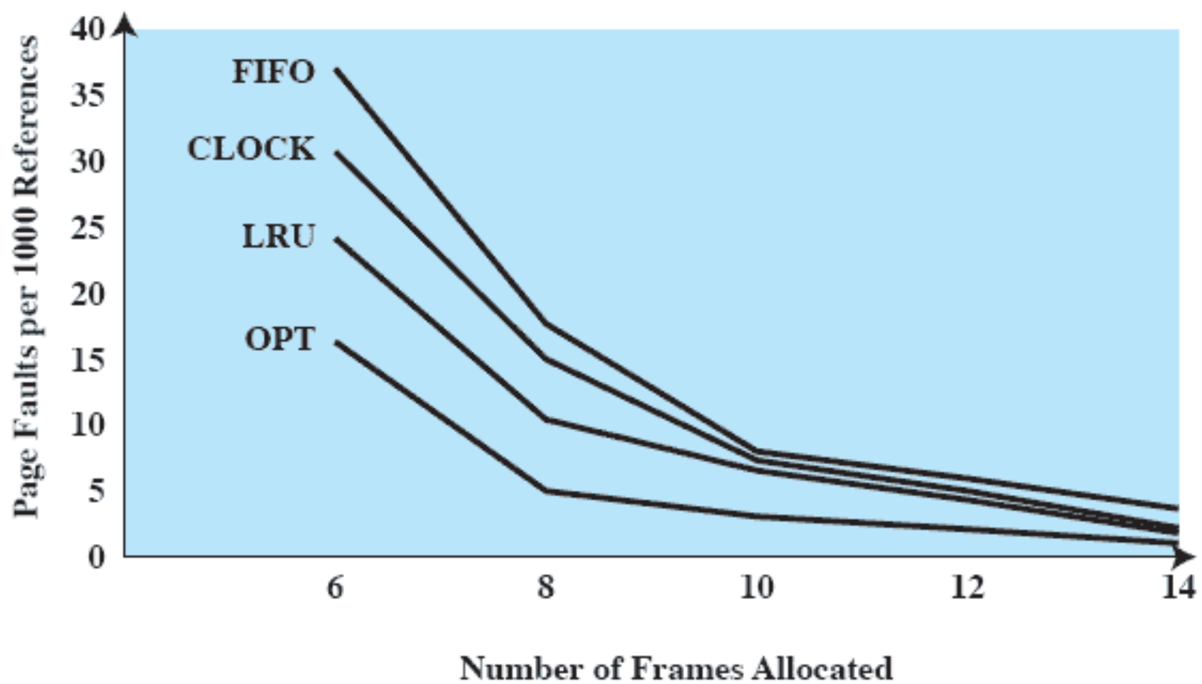


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

Space Concerns

- Often think of simple page table
  - One page table per process
- Is this really practical?
  - Suppose using 32-bit addresses, half for OS, half for user space
  - Number of pages in a user process is 2^32 / pagesize
  - If pagesize is 2^10, 2^22 page table entries per process

Hierarchical Page Tables

- Still only one page table per process, just only load parts being used
- Two-level page table consisting of:
    - Root page table (essentially page directory)
        - Each entry points to a small page table
    - Individual page tßables, each of which points to a portion of total virtual address space

Hierarchical Page Tables

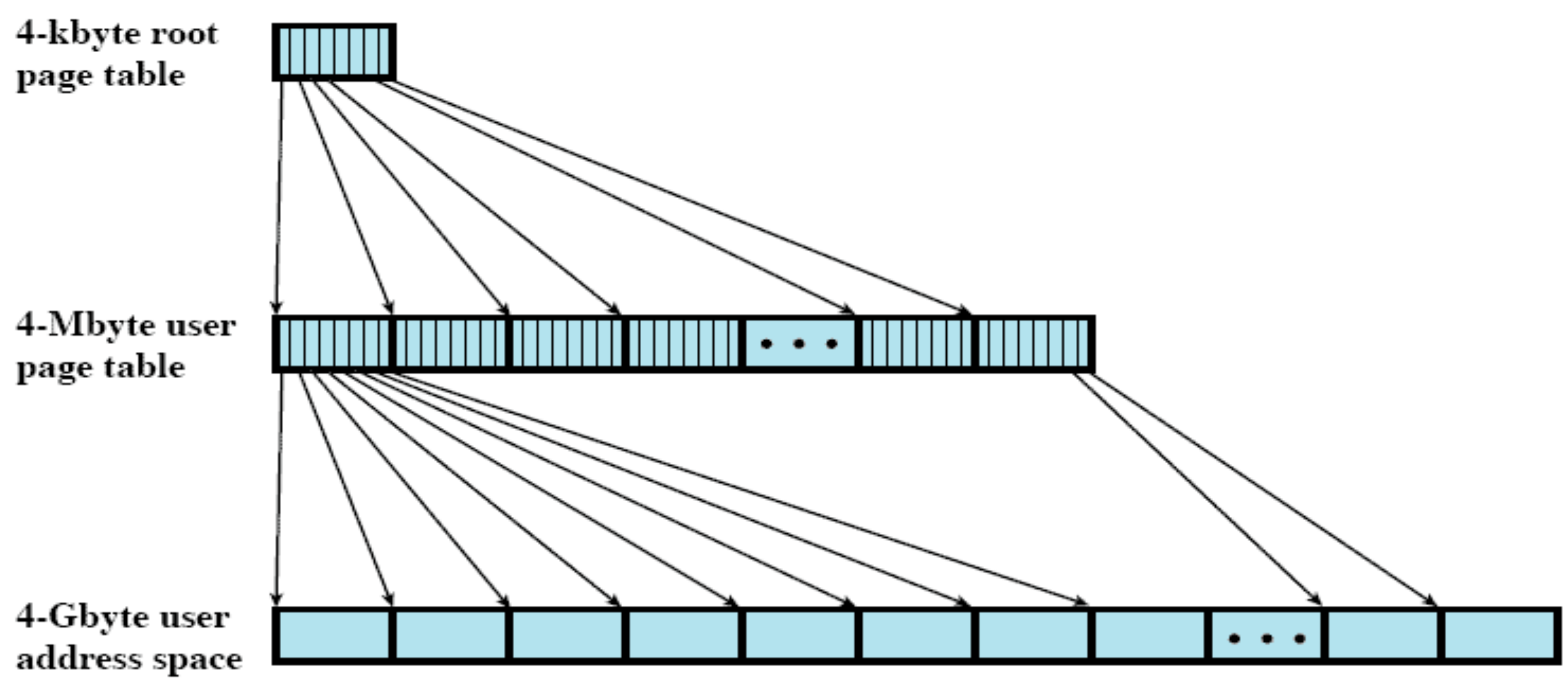


Figure 8.4 A Two-Level Hierarchical Page Table

Hierarchical Page Tables

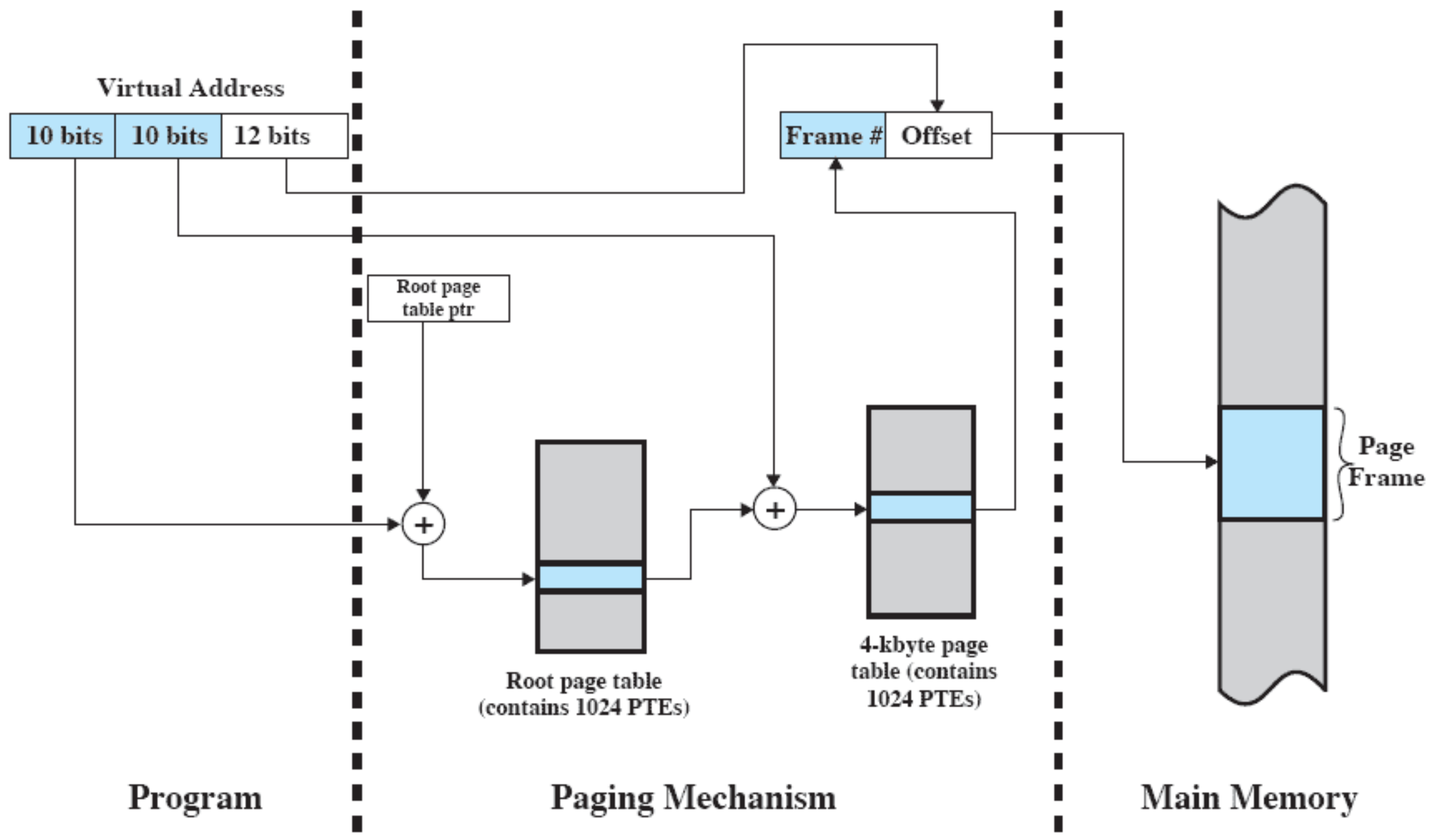


Figure 8.5 Address Translation in a Two-Level Paging System

Translation Lookaside Buffer (TLB)

- Each virtual memory reference can cause two physical memory accesses
  - One to fetch page table entry
  - Another to fetch the data
- Naive implementation would cause this to double execution time
  - To deal with this, special high-speed cache
  - Translation lookaside buffer
- TLB is a cache, so can be implemented with direct or associative caching

Using the TLB

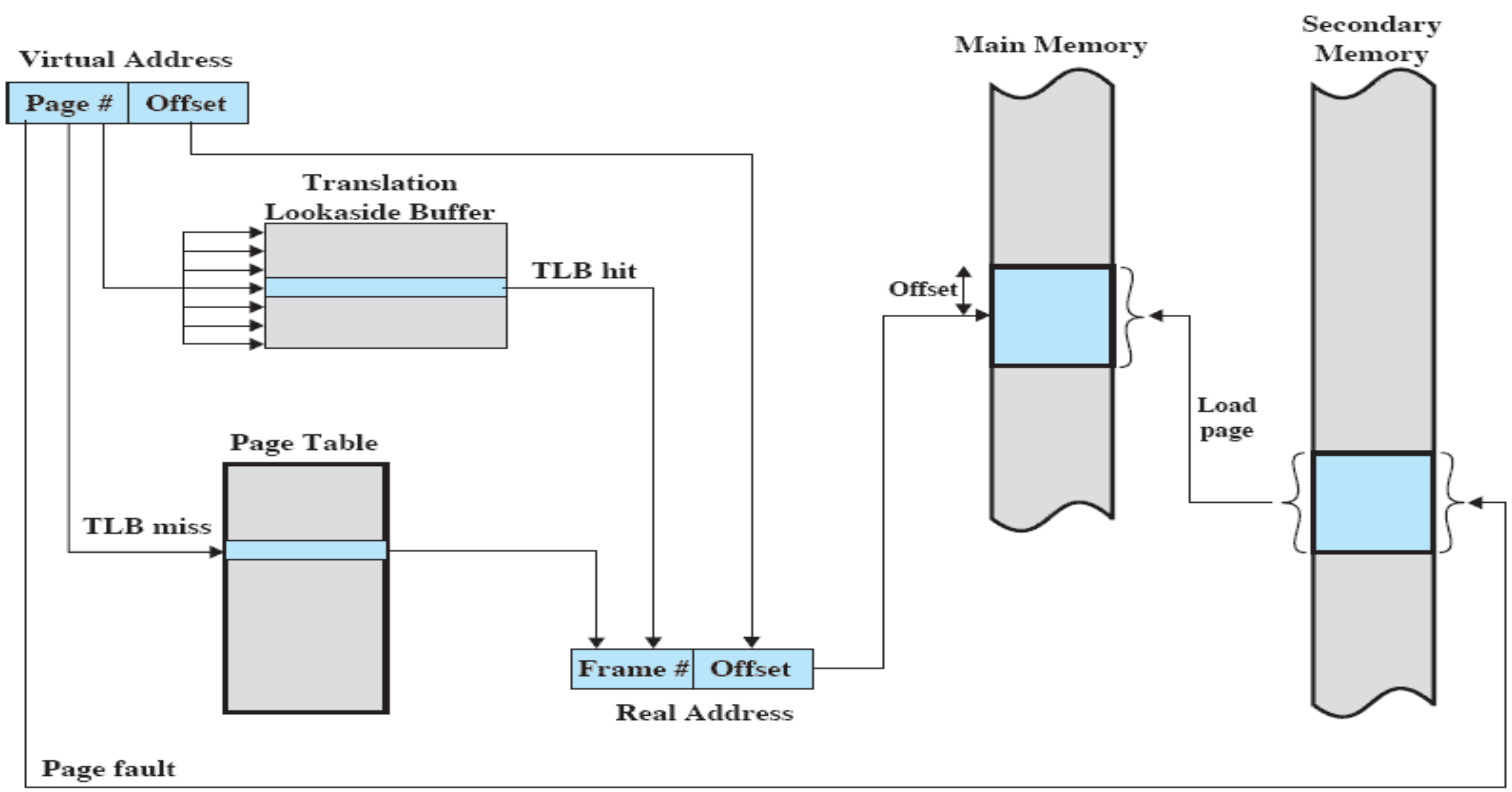


Figure 8.7 Use of a Translation Lookaside Buffer

TLB Operation

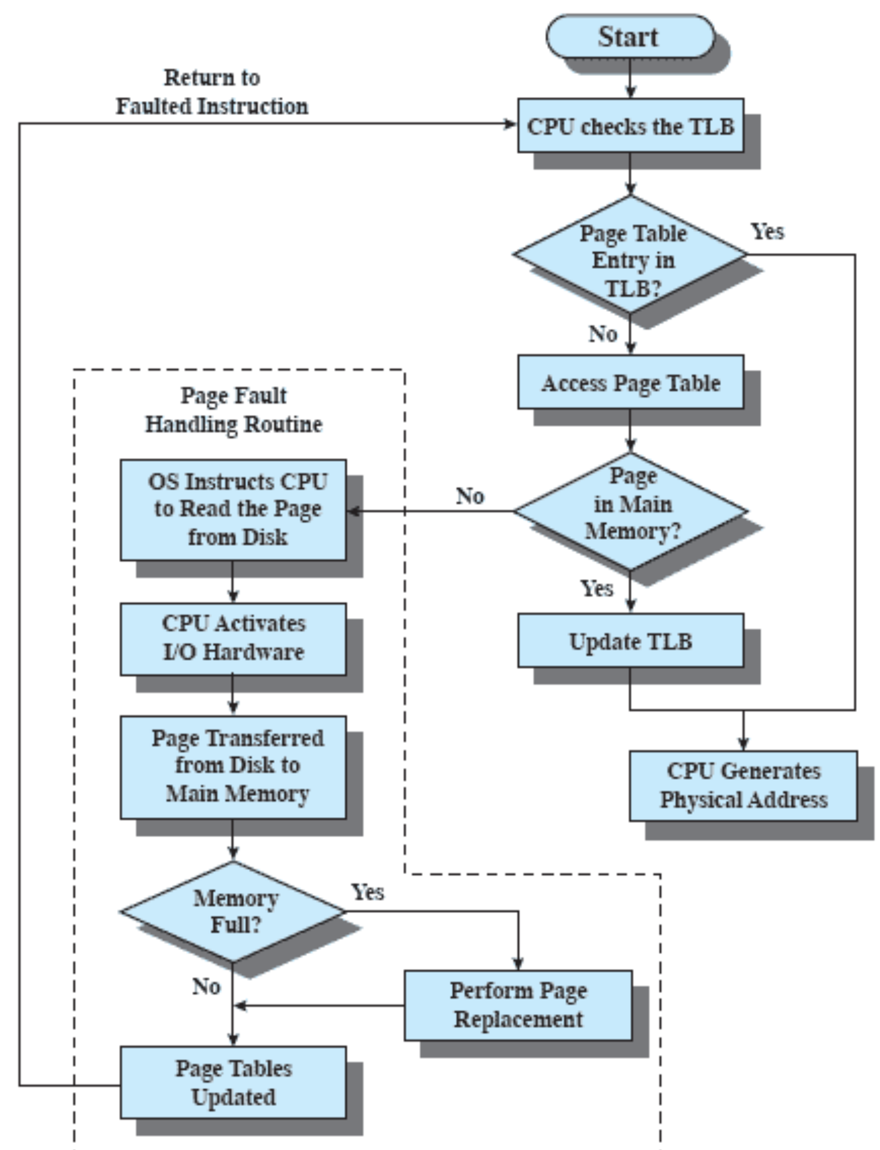


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

Resident set management

- When new process comes into system, OS must decide how many pages to bring into main memory
  - Smaller this is, more processes can reside in memory
  - Larger it is, less page faults (with upper limit at some point)
- This size can be fixed
  - Processes always have the same number of frames allocated to it
  - Page to be replaced is chosen from its resident set
- Varied
  - Allow the max number of frames to a process to vary over time

Segmentation Fault

- When a program tries to access memory it should not have access to
- Generated in two ways:
  - When program tries to read or write memory not allocated to it
  - When writing memory that can only be read
- Generates SIGSEGV signal (usually 11)

Segmentation Fault

- Things that could throw a SIGSEGV:
  - Using uninitialized pointer
  - De-referencing a NULL pointer
  - Trying to access memory program does not have privileges for
    - Accessing array out of bounds
  - Trying to access memory which has already been freed

Bus Error

- When a process tries to access memory that CPU cannot physically address
- Generated when program attempts to use an invalid address
  - Usually caused by alignment issues
- Generates SIGBUS signal (usually 10)

Bus Error

- Things that could throw a SIGBUS:
  - Process tries to access specific physical memory address which is not valid
    - Invalid pointer is dereferenced
  - Unaligned multi-byte access
    - Often multi-byte access must be aligned by bytes
      - Require that longs be accessed at byte 0,4,8,12,16, so on
      - Trying to read byte 3 or 5 would generate a SIGBUS

Paged Segmentation

- Can we combine paging and segmentation?
    - User address space broken up into segments
    - Each segment broken up into number of fixed-size pages
        - Each size of a main memory frame

Paging and Segmentation

- Under paged segmentation, different visibility
  - Segmentation visible to programmer
  - Paging is only visible to OS

Paging and Segmentation

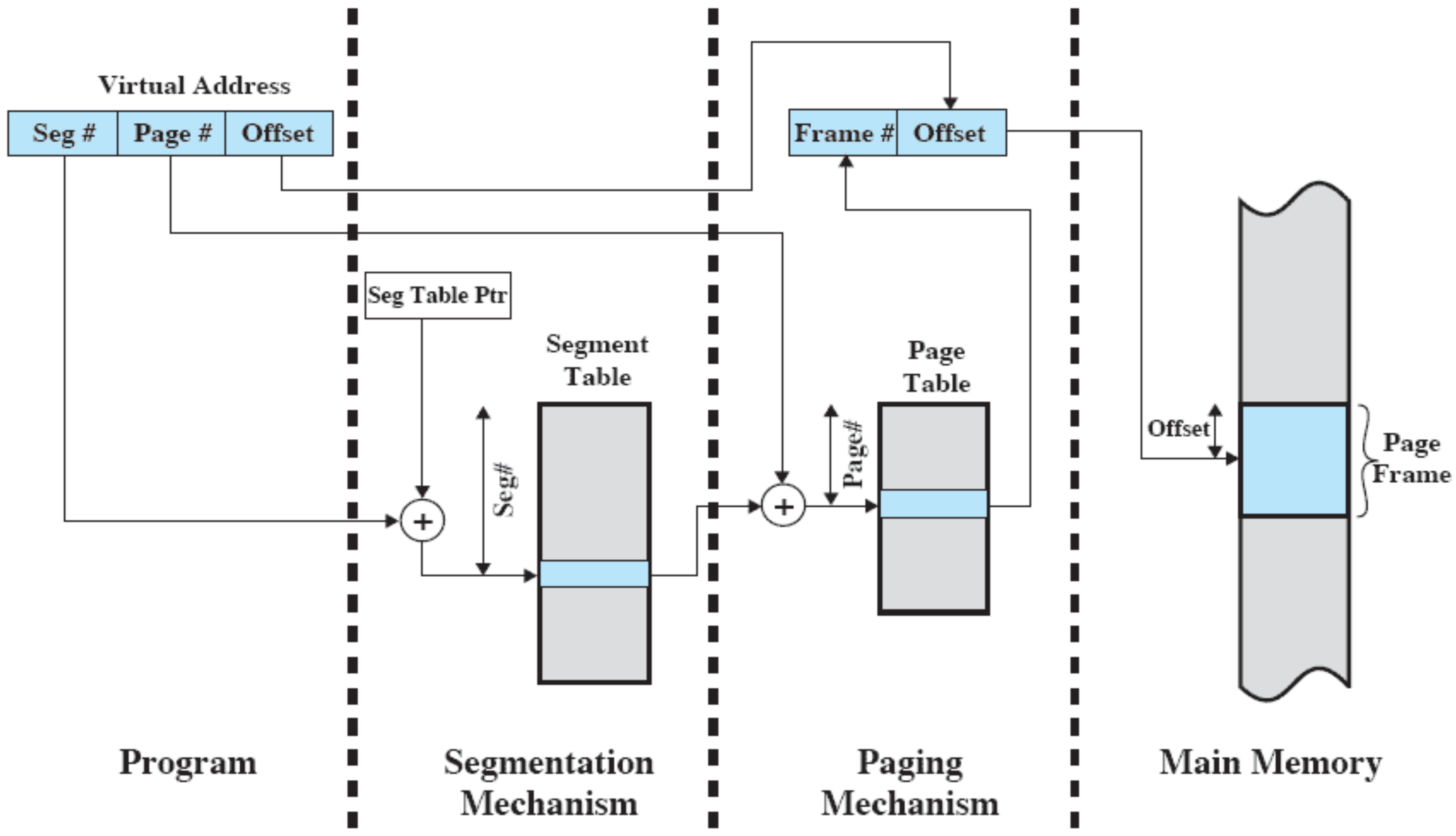


Figure 8.13 Address Translation in a Segmentation/Paging System

Inverted Page Tables

- Lets solve the size of our page tables in another way
  - With just one entry for each frame of physical memory
- A problem!
  - More than one virtual address could map to that location
- Called Inverted Page Table
  - Indexes page table entries by frame number, not by virtual page number

Inverted Page Table

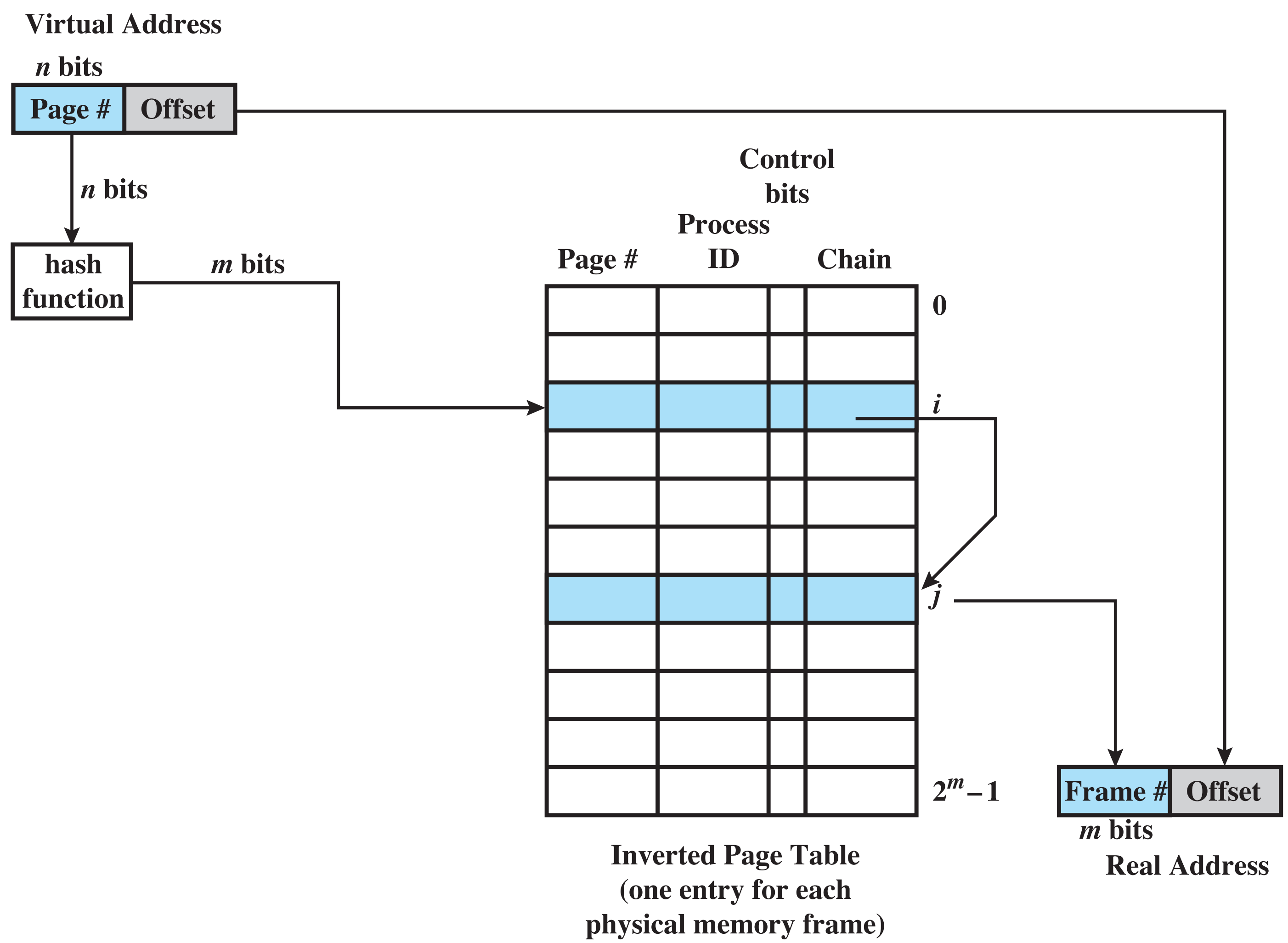


Figure 8.5 Inverted Page Table Structure

Demand Paging

- Demand paging only brings pages into main memory when a reference is made
    - What happens at the start of processes?
    - Can we possibly change this exploiting principle of locality?

Prepaging

- Lets bring in pages other than the one demanded by page fault
- How does this help us? How can we be sure pages will be used?
  - Exploit the characteristics of secondary storage devices
  - If pages of a process are stored contiguously in secondary memory, efficient to bring in a number of pages at one time
  - Ineffective if extra pages are not referenced

End on memory management!

- Any questions?