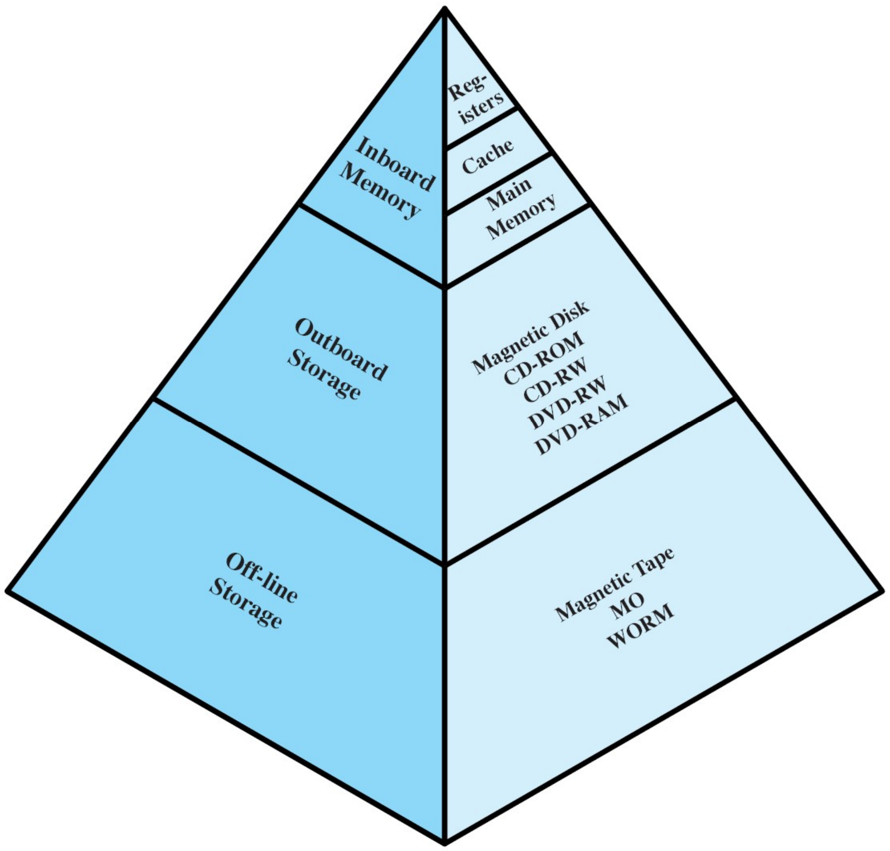
Operating Systems

7. Memory Management

Memory Management

- Ideally programmers want memory that is
  - Large
  - Fast
  - Non volatile
- Memory hierarchy
  - Small amount of fast, expensive memory – cache
  - Some medium-speed, medium price main memory
  - Gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy



Requirements for Memory Management Unit (MMU)

- Relocation
  - Change the physical placement of a process
- Protection
  - Restrict processes access to physical memory
- Sharing
  - Allow processes to share memory location
- Logical Organization
  - Support organization of computer programs etc.
- Physical Organization
  - Efficient utilization of hardware
  - Memory hierarchy

Background

Program

- Typically resides on disk
- Must be brought into memory to be executed

Address binding of a program

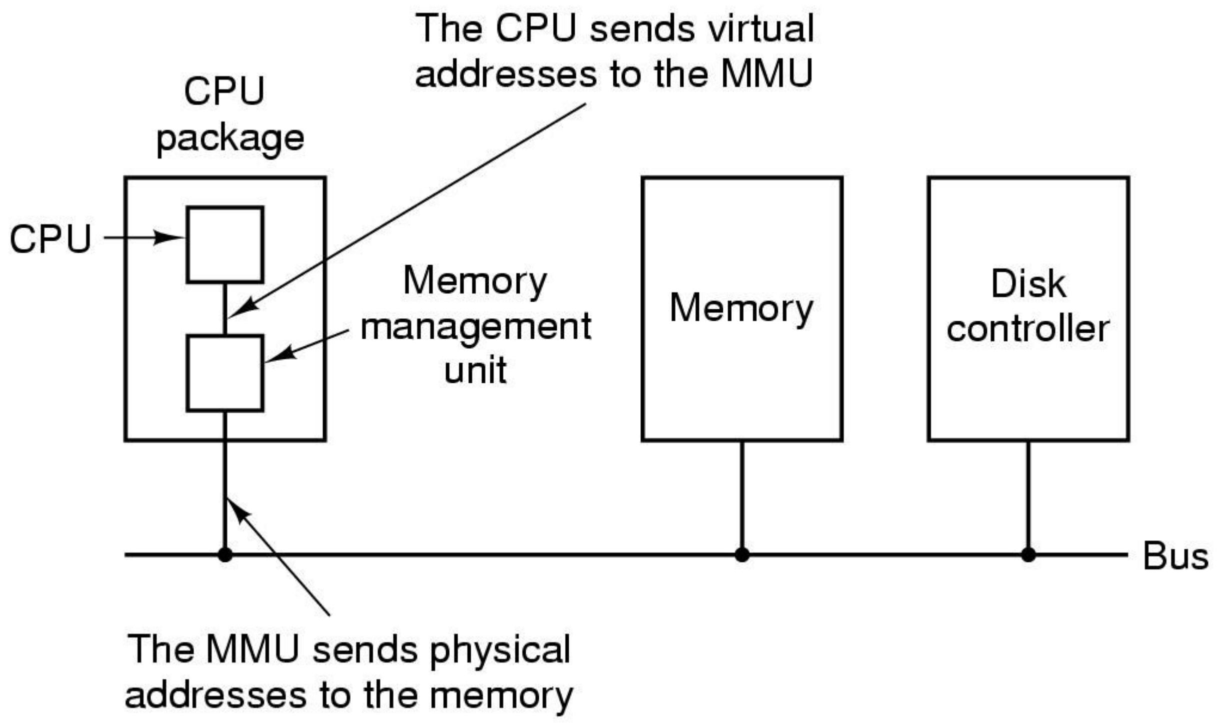
- Compile time
  - Program must always be loaded to the same memory location
- Load time
  - Memory location determined when program is loaded
- Run time/ Execution time
  - No absolute memory locations
  - Memory location determined by memory management unit (MMU)
    - ➢ Swapping of processes and relocation

Background

Memory

- Can be subdivided to accommodate multiple processes
- Needs to be allocated efficiently to pack as many processes into memory as possible

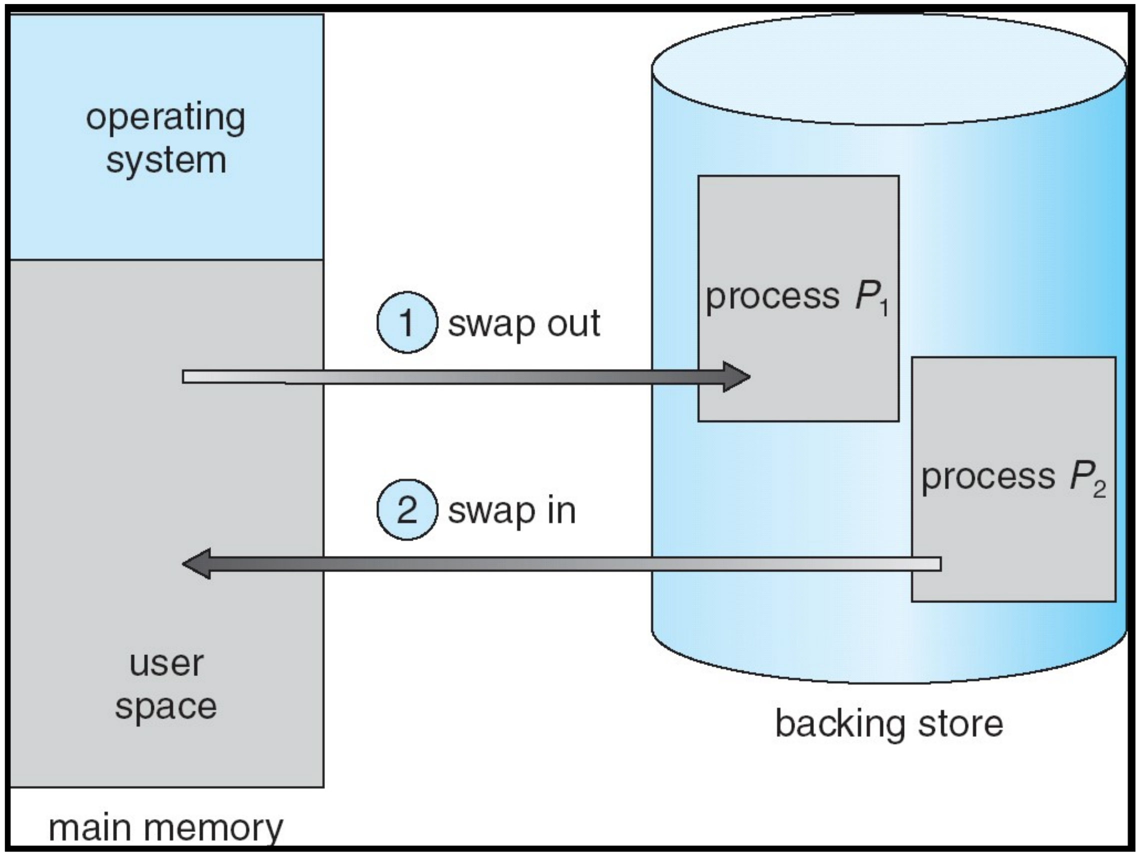
Position and Function of the MMU


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

- Logical address: Generated by the CPU
  - Also referred to as virtual address or relative address
- Physical address: Address seen by the memory unit
  - Also referred to as absolute address

Sharing Memory Through Swapping

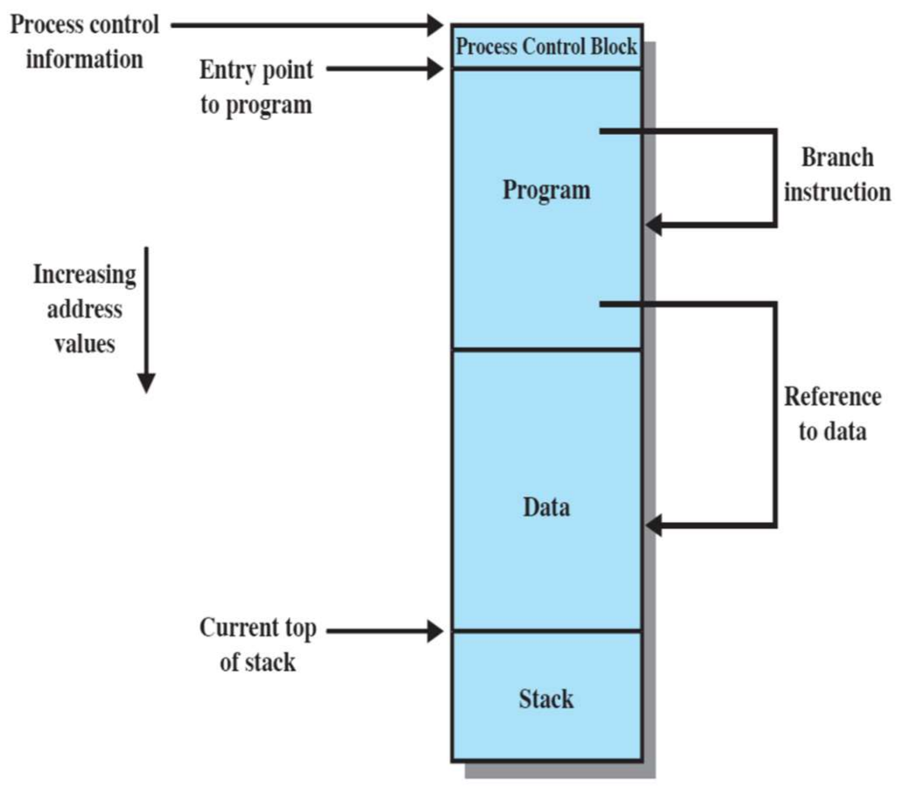
- A process can be swapped out of memory to a backing store
  - Swap device
- Later brought back (swap-in) into memory for continued execution



Relocation and Protection

Relocation

- Swapping
  - Program must be relocated
- Cannot be sure about the location in memory
- Addressing
  - No absolute addressing feasible
  - For example, locations of variables, code routines

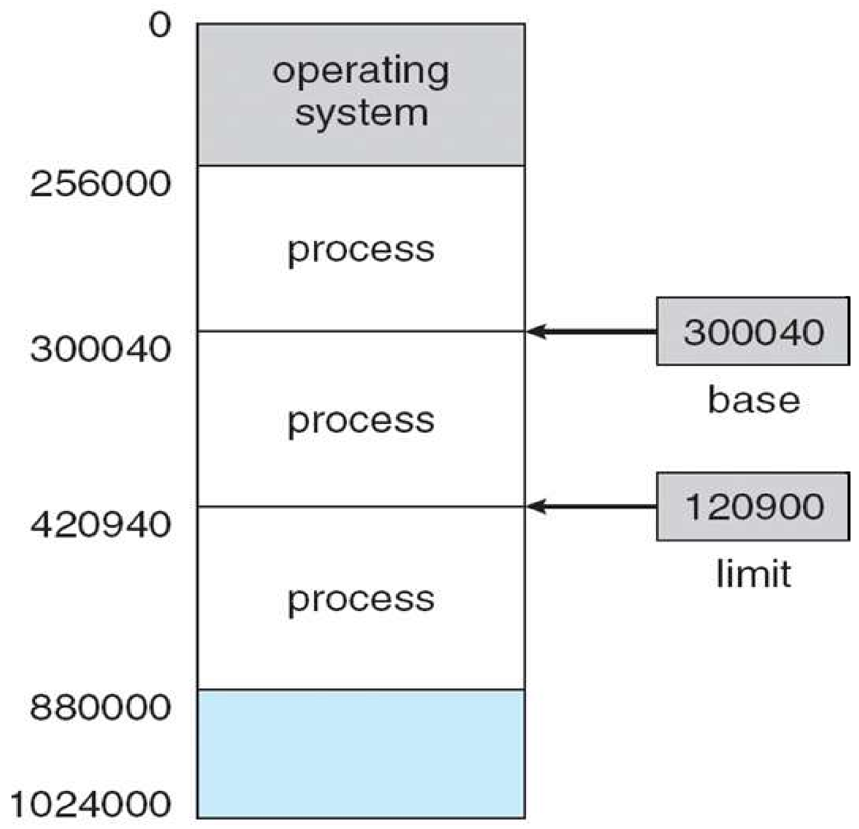


Protection

- Must keep a program out of other processes' partitions

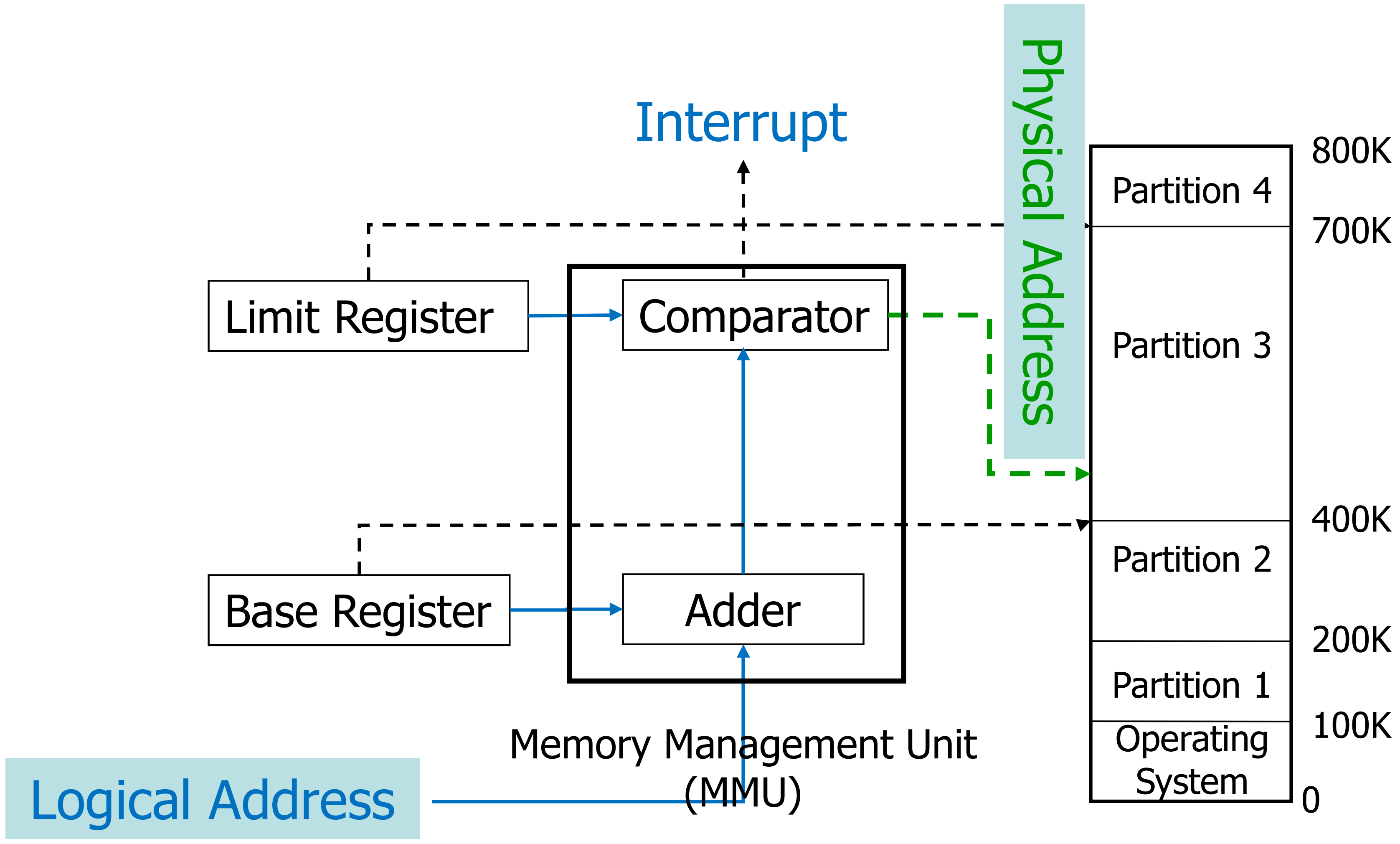
Hardware Support For Relocation and Protection

- Base, bounds (limit) registers
  - Define the range which a process can access
  - Set when the process is executing



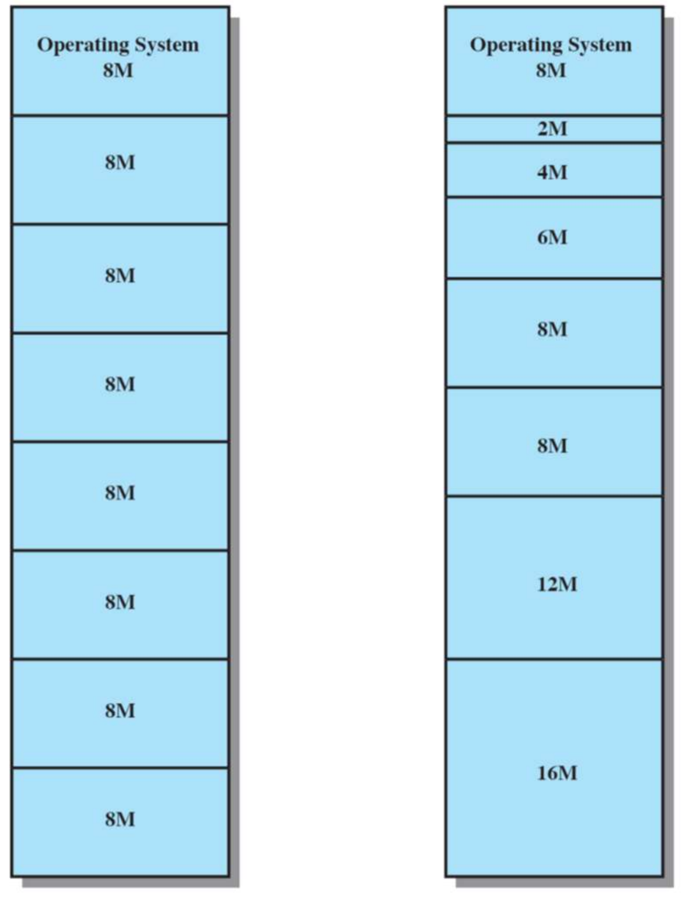
Hardware Support For Relocation and Protection

Addition and comparison has to be performed on every address



Contiguous Allocation of Memory: Fixed Partitioning

- Any program, no matter how small, occupies an entire partition
- Causes internal fragmentation
  - Allocated memory can be larger than requested memory
  - Difference internal to a partition, but not being used by the process

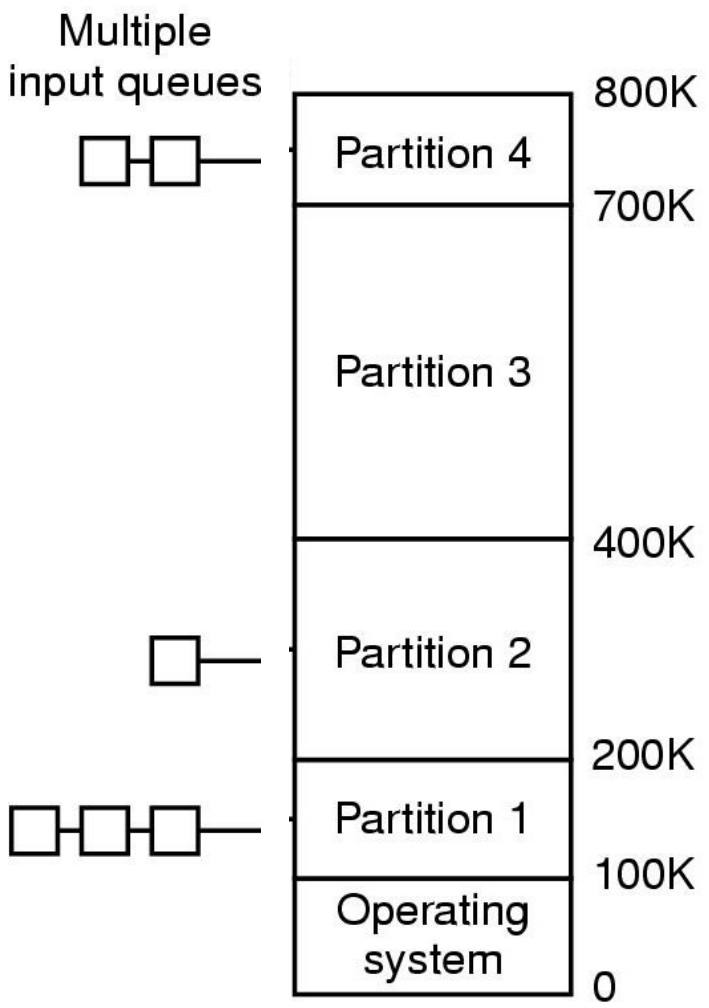


(a) Equal-size partitions (b) Unequal-size partitions

Fixed Partitioning

Multiple Queues

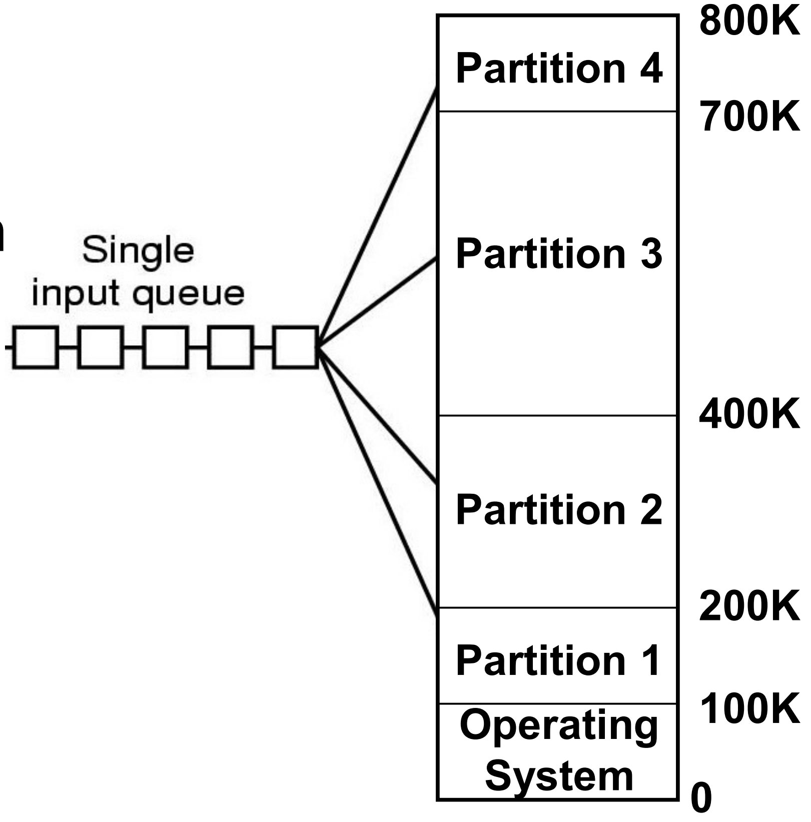
- Small processes have to wait, even though plenty of memory is free
  - When a large partition is empty
  - Queues for small partition is full
- Alternative: Maintain a single Queue



Fixed Partitioning

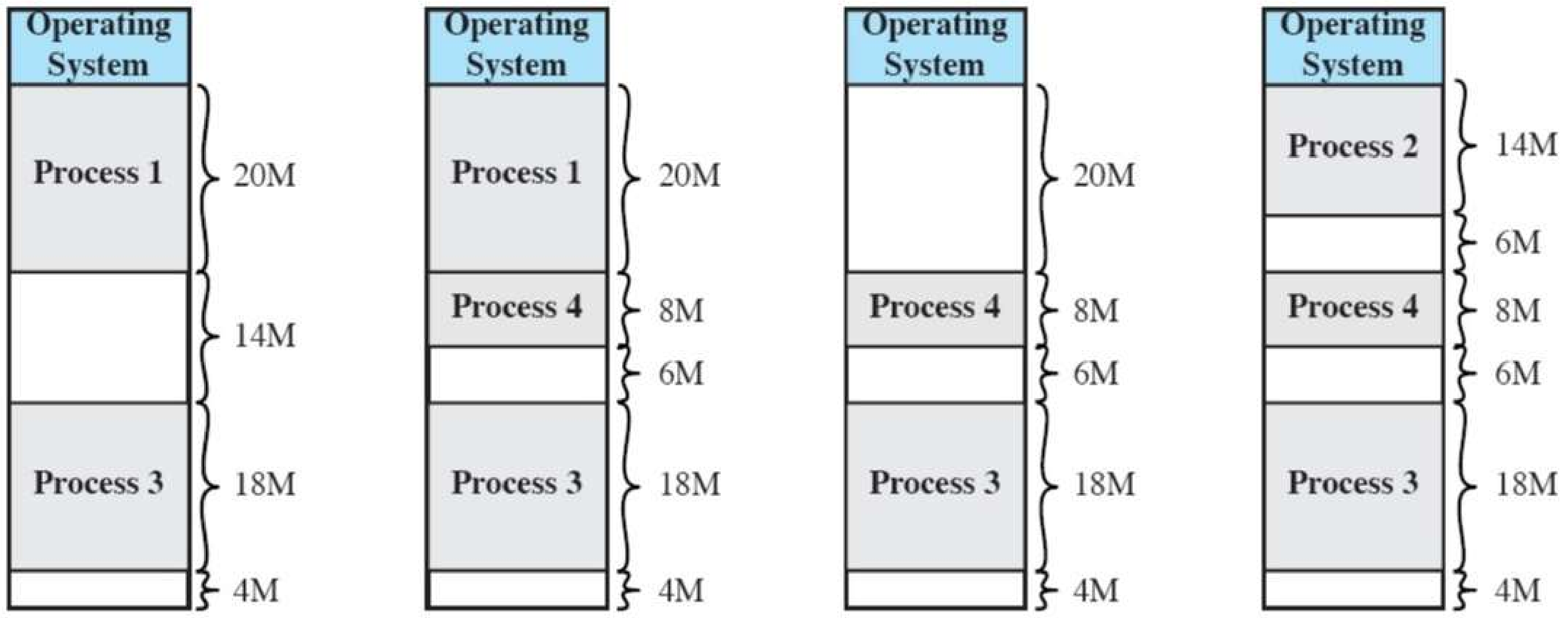
Single Queue

- Whenever a partition becomes free, a process is selected
  - Closest to the front of the queue
  - Smaller than the partition size
- Undesirable to waste a large partition for smaller process
  - Search the queue, find the largest process that fits it
- Unfair for smaller processes
  - A process may not be skipped more than k times



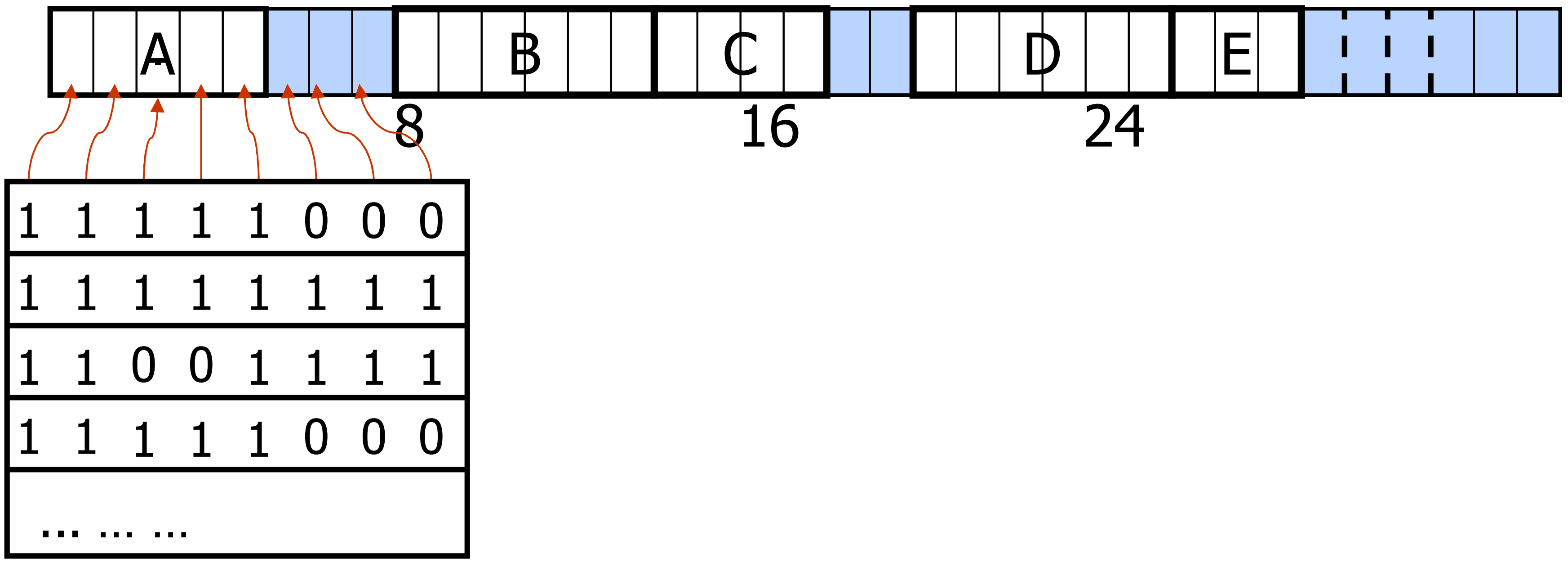
Contiguous Allocation: Dynamic Partitioning

- Process is allocated exactly as much memory as required
- Eventually holes in memory: External fragmentation
  - Total memory space exists to satisfy a request
  - But it is not contiguous
- Must use compaction to shift processes (defragmentation)



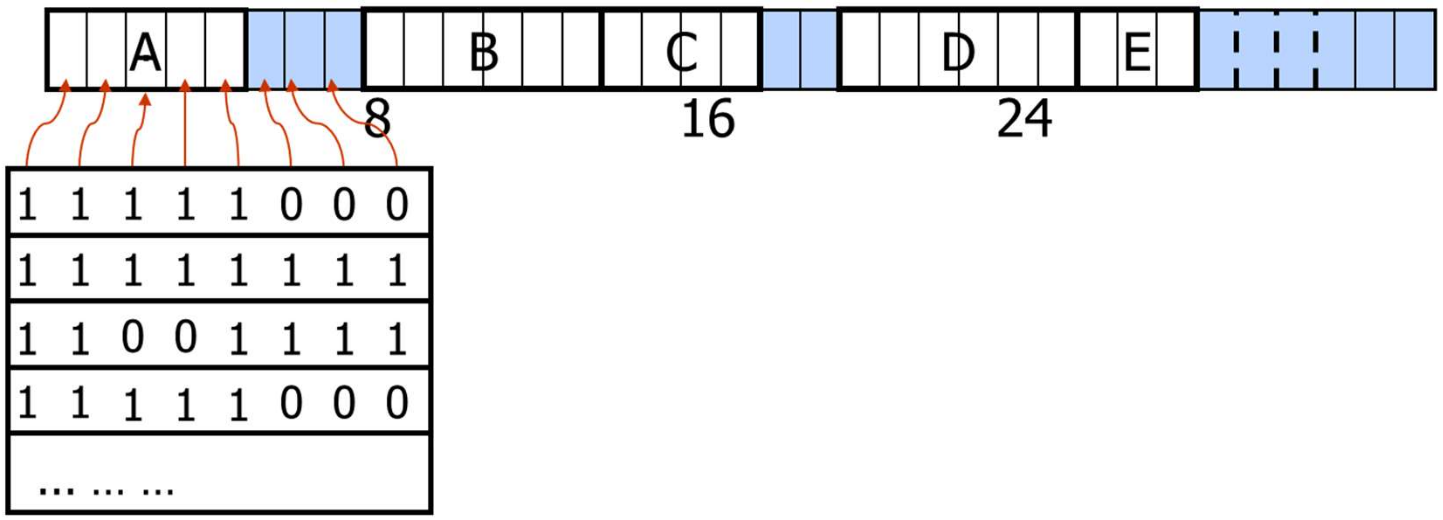
Memory Management With Bitmaps

- Memory is divided up into allocation units
  - Few words
  - Or several kilobytes



Memory Management With Bitmaps

- Size of allocation unit is important
- Smaller allocation unit
  - Larger bitmap required
- Larger allocation unit
  - Smaller bitmap required
  - More memory will be wasted
    - ➢ If the process size is not an exact multiple of the allocation unit

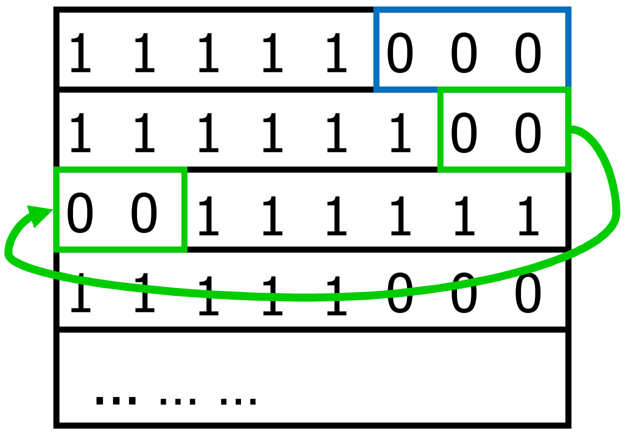


Memory Management With Bitmaps

- To bring a k unit process in memory
- Search for a k run consecutive 0 bits in the map
- Search can be slow
- Since, k run may cross word boundaries

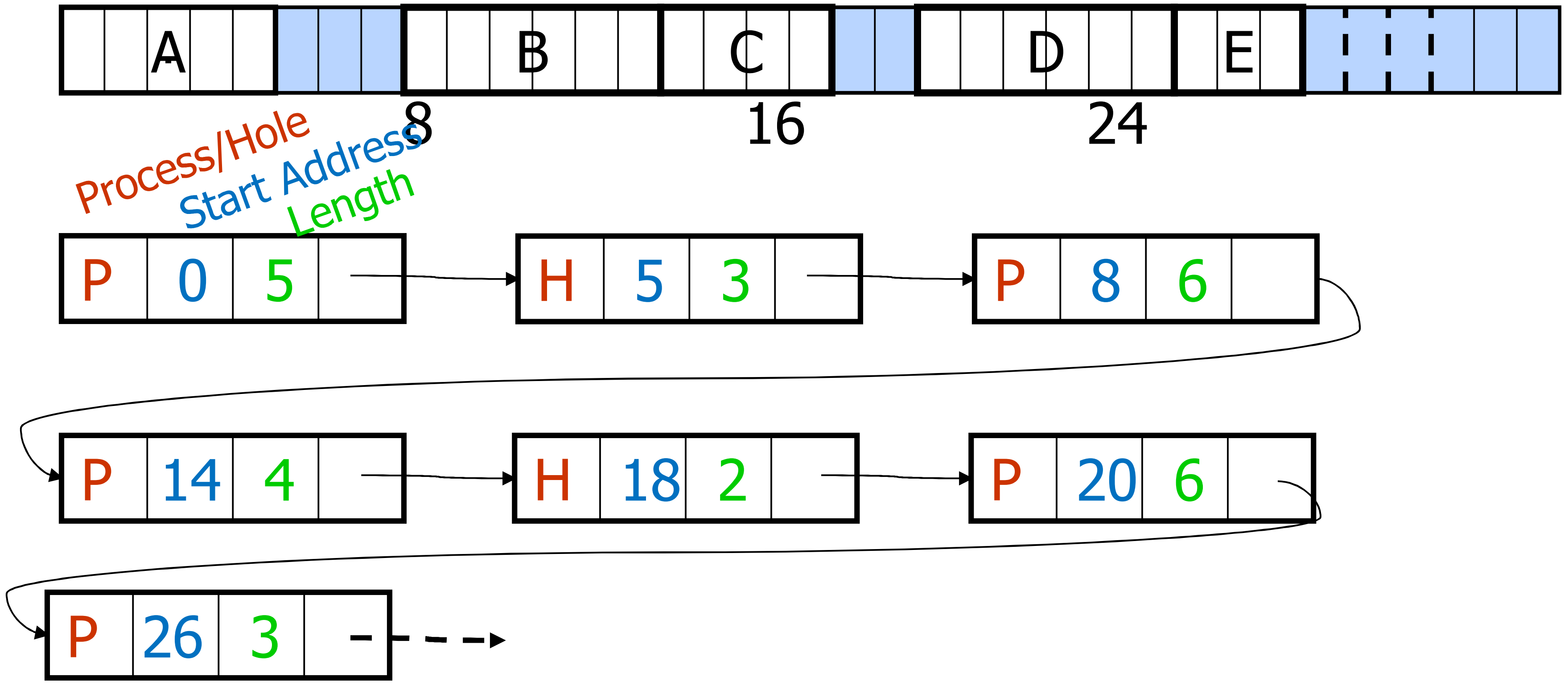
Find run of length = 3

Find run of length = 4



Memory Management with Linked Lists

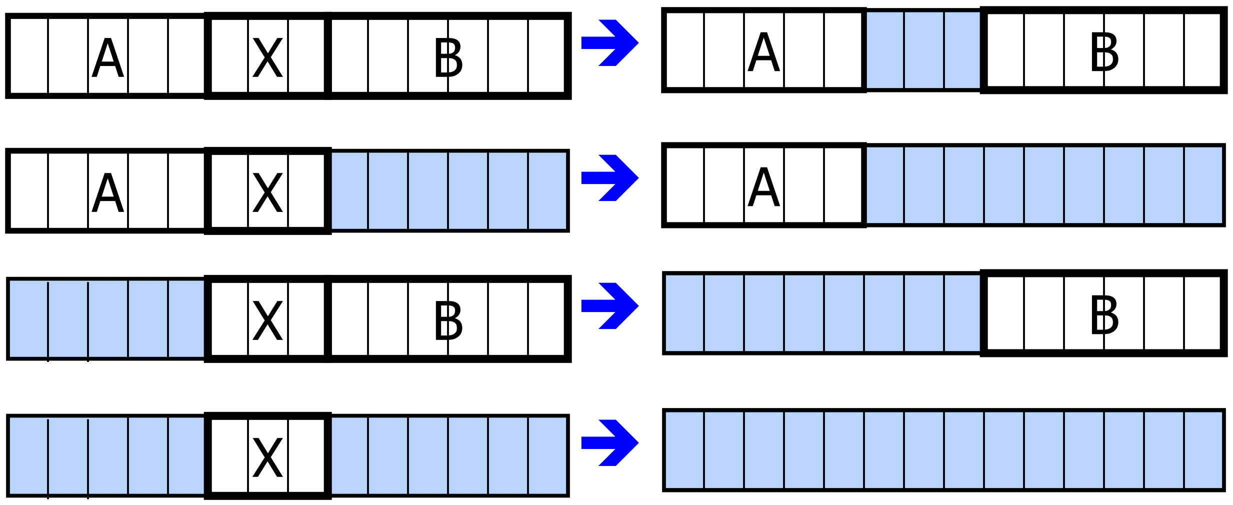
- Linked list of memory segments
  - Free segments → Holes
  - Allocated segments → Processes



Memory Management with Linked Lists

- Segment list is sorted by addresses
- Sorting helps in updating the list, when a process is swapped out or exits
- A process usually has two neighbors

Updating the list requires



Replace P by H

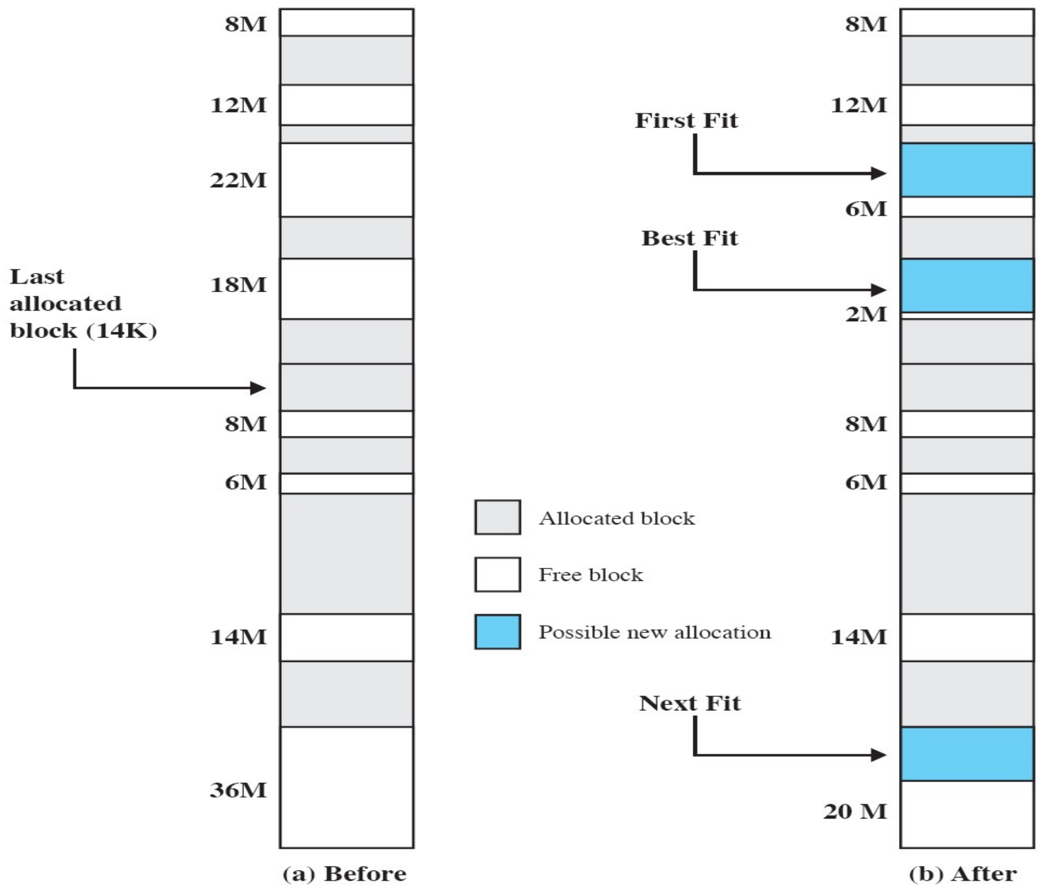
Two entries are merged

Two entries are merged

Three entries are merged

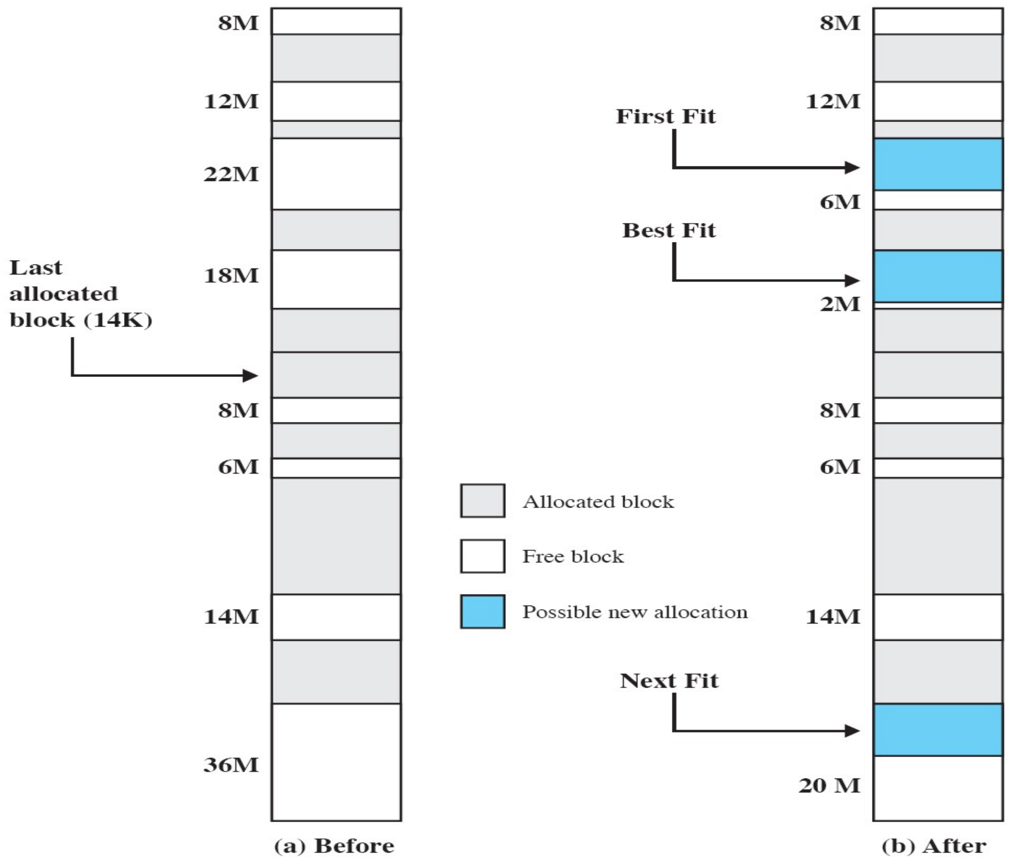
Dynamic Partitioning: Placement Algorithms

- First-fit: Use the first block that is big enough
  - Simple and fast



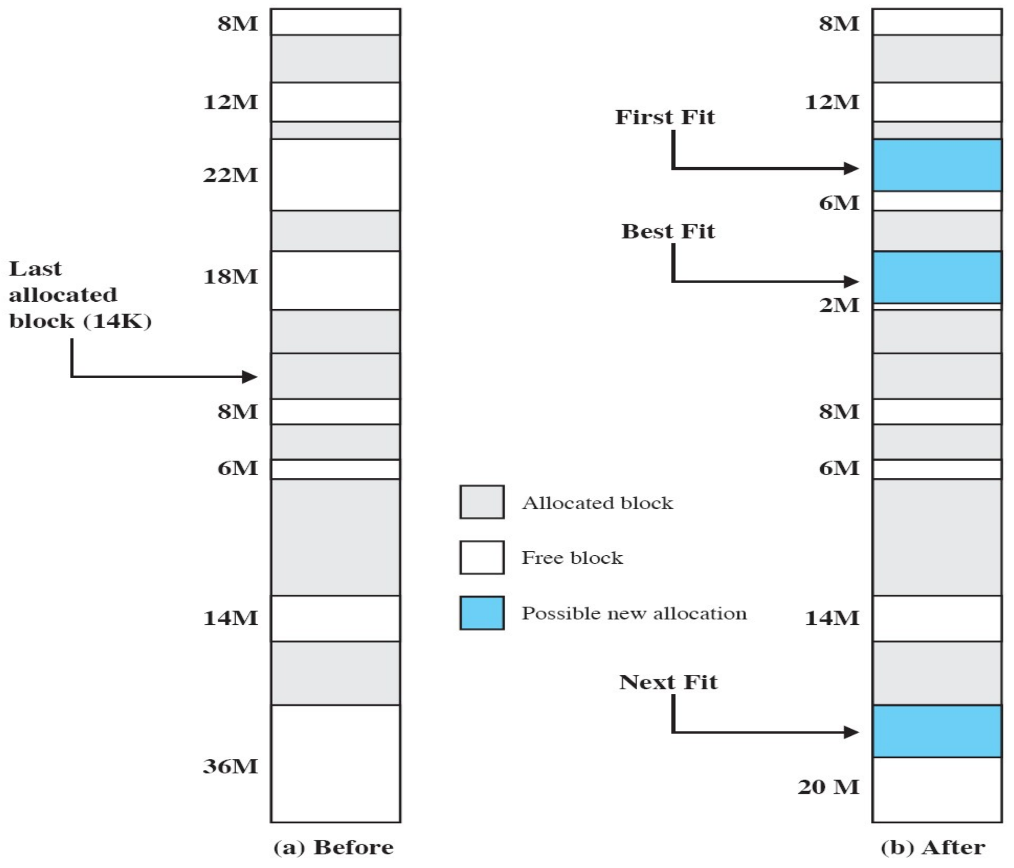
Dynamic Partitioning: Placement Algorithms

- Next-fit: Use the next block that is big enough
  - Tends to eat-up the large block at the end of the memory
  - Gives no better performance than First-fit



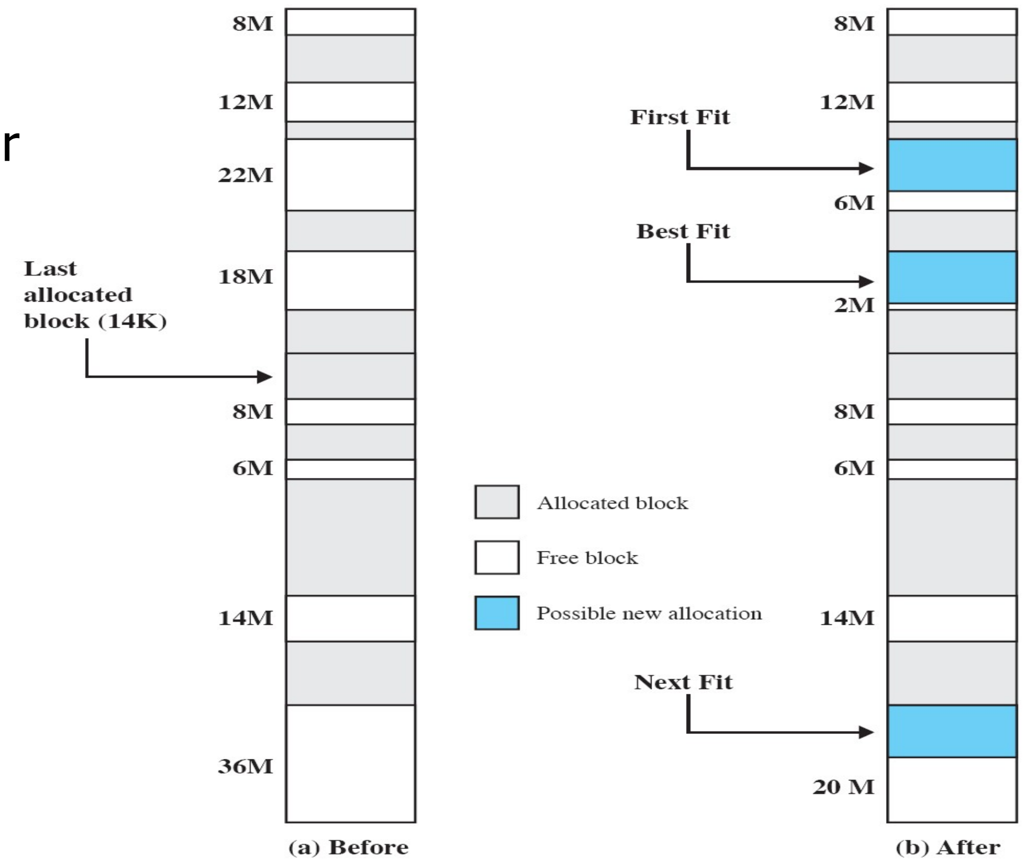
Dynamic Partitioning: Placement Algorithms

- Best-fit: Use the smallest block that is big enough
  - Must search entire list (unless free blocks are ordered by size)
  - Produces the smallest leftover hole
    - Tends to fill up memory, with tiny useless holes
    - First Fit generates larger holes on the average



Dynamic Partitioning: Placement Algorithms

- Worst-fit: Use the largest block
  - Must also search entire list
  - Produces the largest leftover hole…
  - … but eats-up big blocks



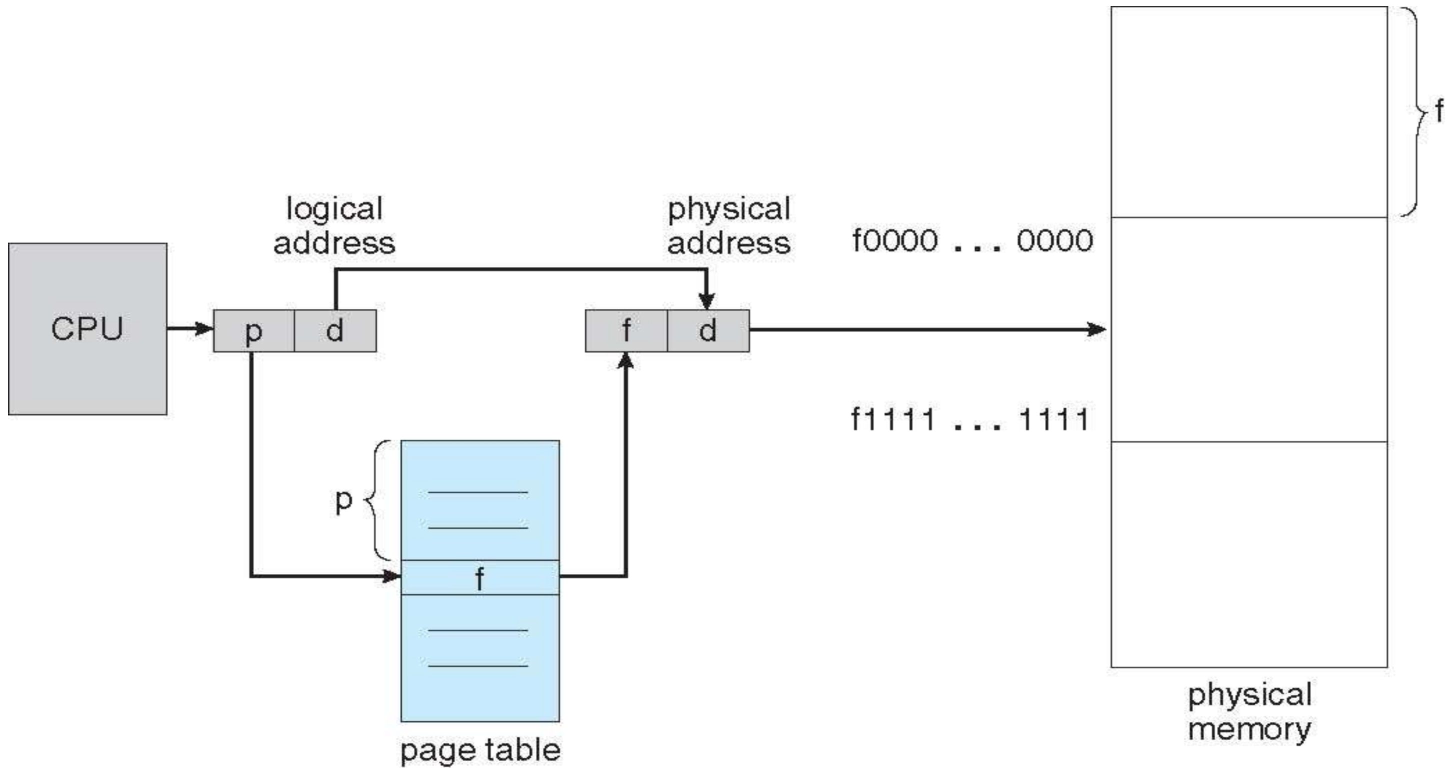
Memory Management with Linked Lists

- Search time of all four placement algorithms can be improved by
  - Keeping separate lists for Process and Holes
  - While allocating memory only have to search the lists of holes
  - The Hole list can be kept sorted by size
    - ➢ As soon as a hole that fits is found, no more searching is required

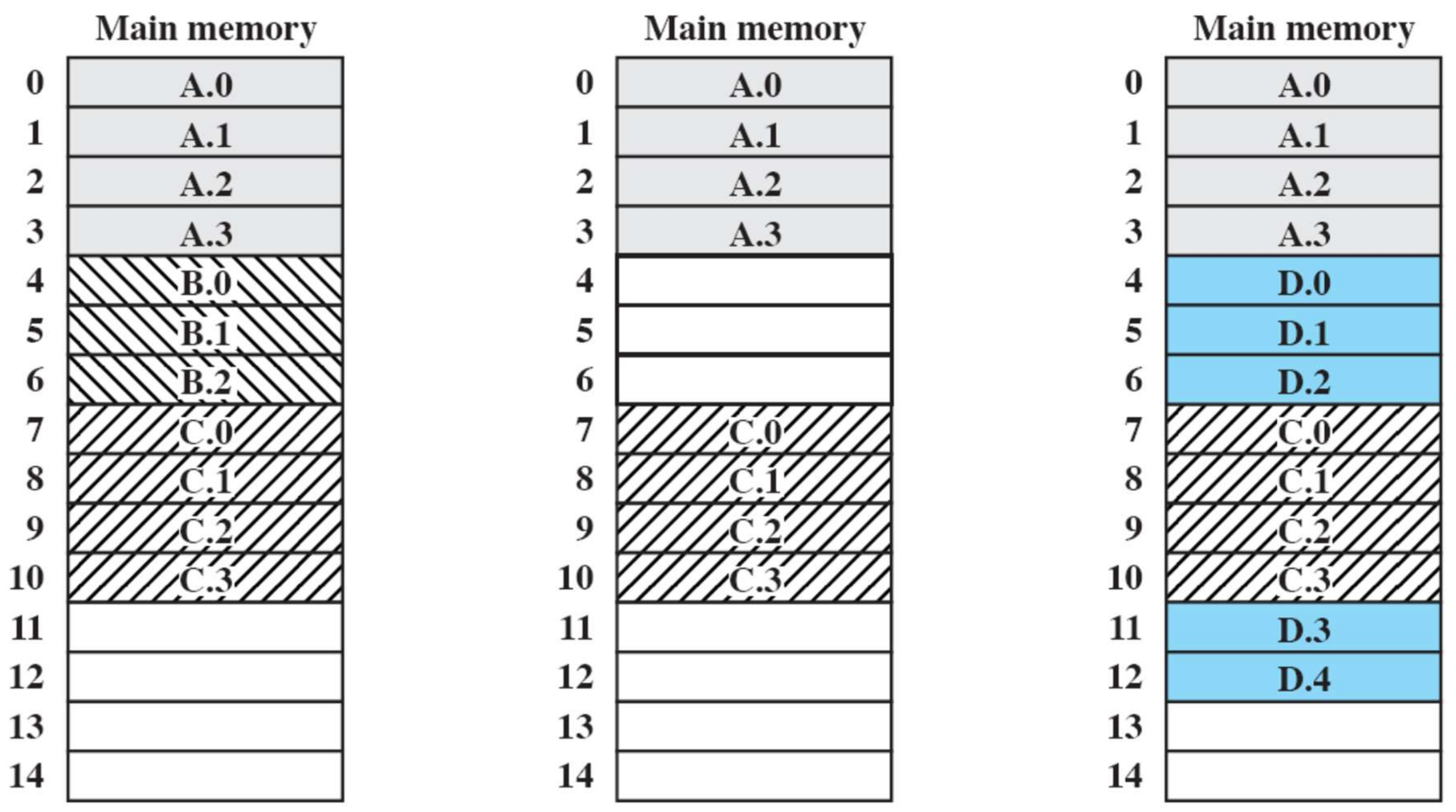
- Drawback
  - Problem while de-allocating memory
  - A node from the process list has to be inserted in the hole list

To Avoid External Fragmentation: Paging

- Partition memory into small equal-size chunks (frames)
- Divide each process into the same size chunks (pages)
- OS maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address = (page number, offset within page)



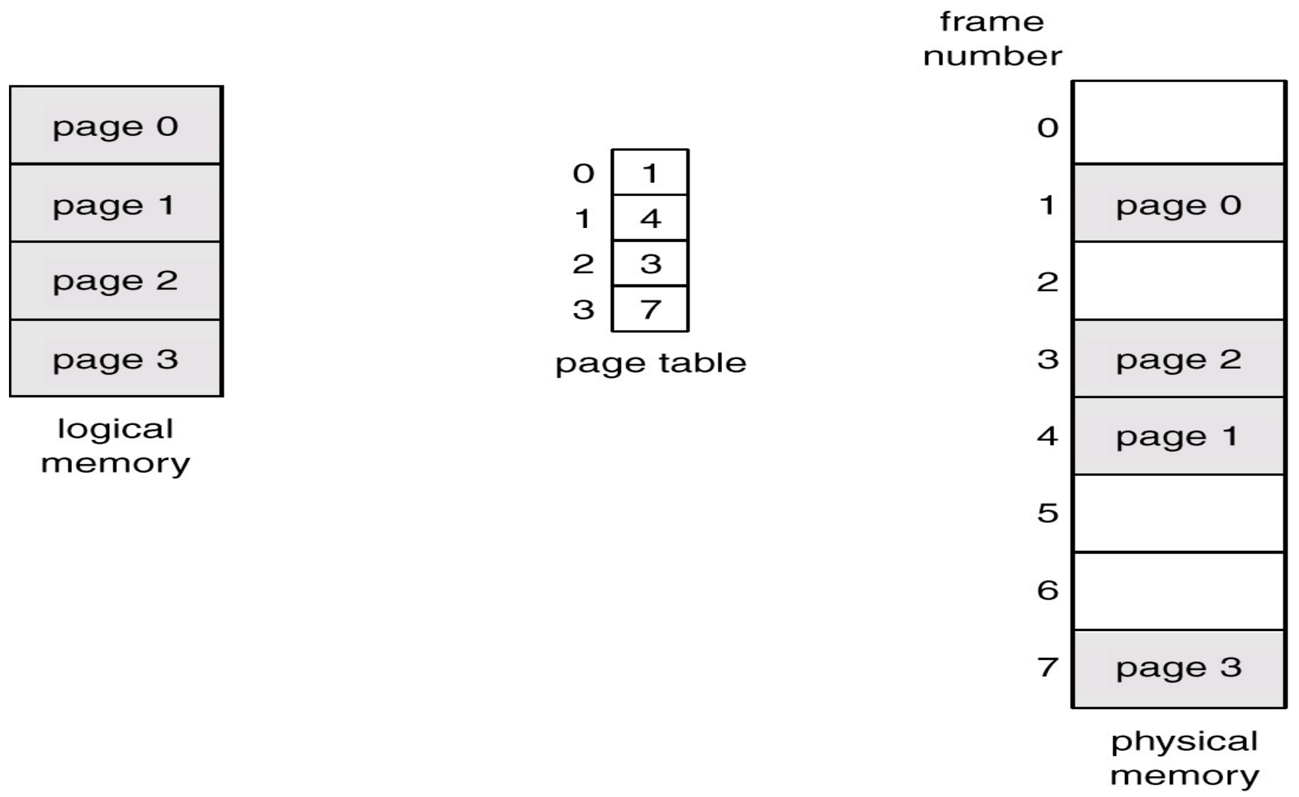
Page Example: Allocation of Frames



(d) Load Process C (e) Swap out B (f) Load Process D

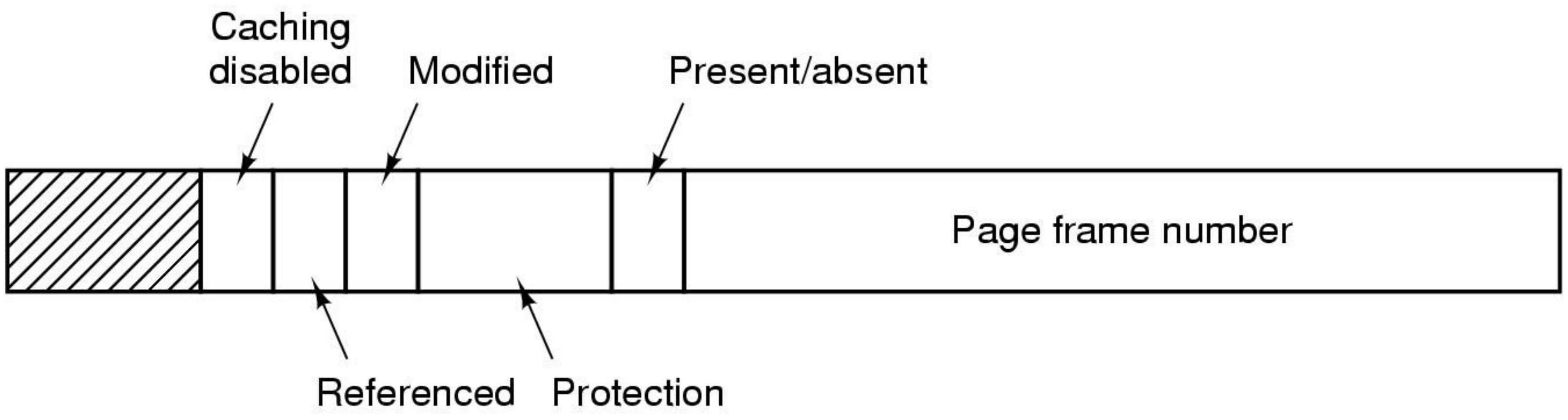
Paging Example

- Pages of a process form the logical memory
- Maintain for each process a page table



- Question: Do we avoid fragmentation completely?

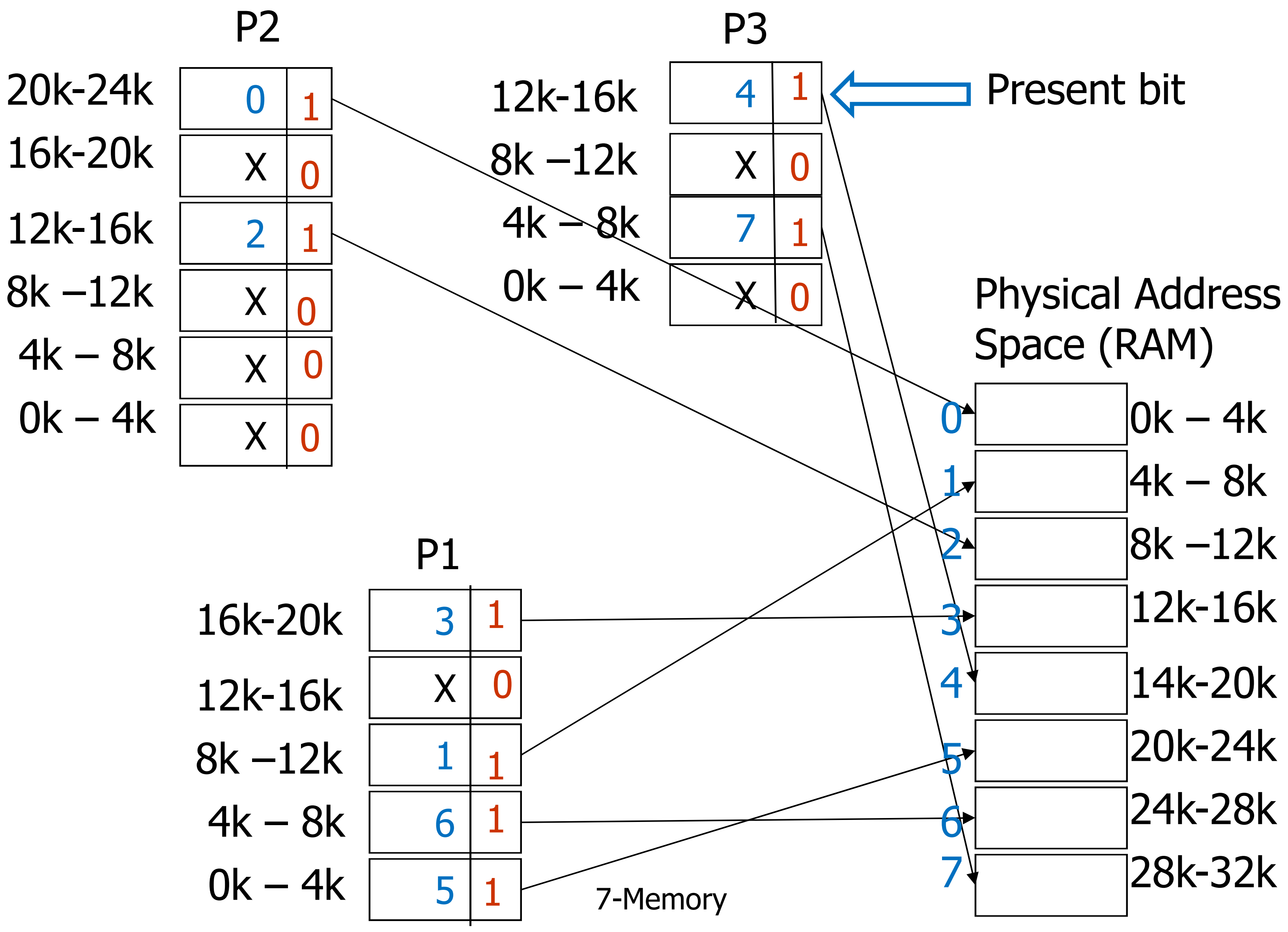
Typical Page Table Entry



Exact layout is hardware dependent

- Present/Valid: value valid to be used, otherwise page fault
- Protection: defines accesses which are permitted
- Referenced: help to decide which frames can be reclaimed
- Caching: Needed if value is read from I/O device
- Modified: Needed if page frames are reclaimed
  - Dirty: Needs to be written to disk first

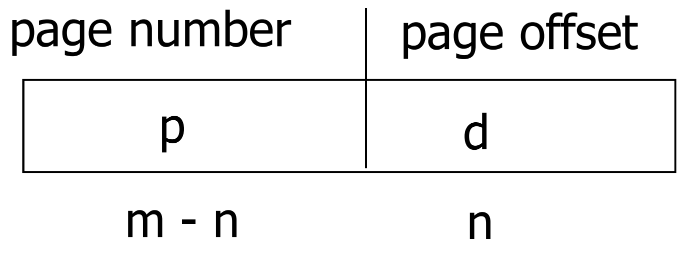
Paging Example



Address Translation Scheme

Address generated by CPU is divided into two parts

- Page number (p) – used as an index into a page table which contains base address of each page in physical memory
- Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

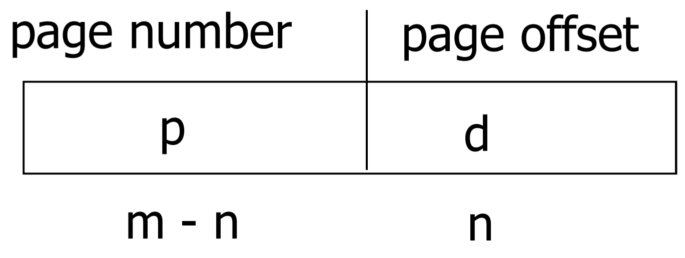


- For given logical address space 2^m and page size 2^n

Address Translation Scheme

- Size of logical address space = 2^m
  - # of pages (2^{m-n})× page size (2^n)

- Example: 16 X 4096
  - $2^m = 2^4 \times 2^{12}$
  - $2^m = 2^{16}$
  - m = 16 bits

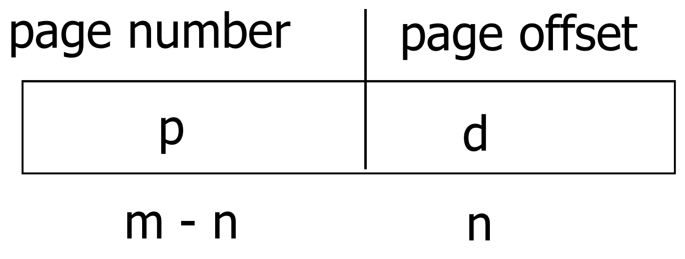


Address Translation Scheme: Example

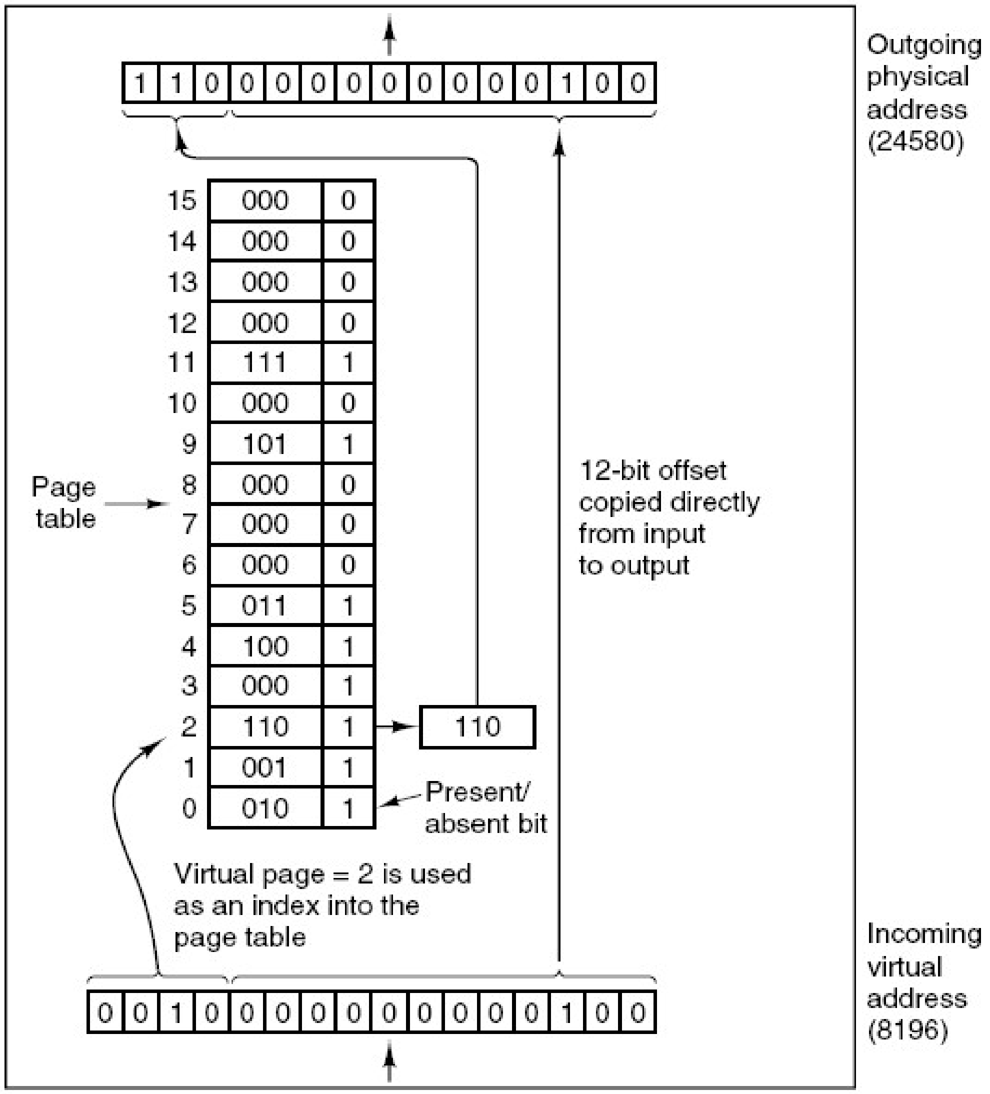
Given m=16 and n=12

- Page number = m-n bits = Leftmost 4 bits
  - 4 bit page number can represent 16 pages
- Page offset = n bits = Rightmost 12 bits
  - 12 bit offset can address all 4096 bytes within a page (Frame size) !!

- Consider logical address 8196 = 0010000000000100
  - Page number is 2 and offset is 4



Address Translation Scheme: Example

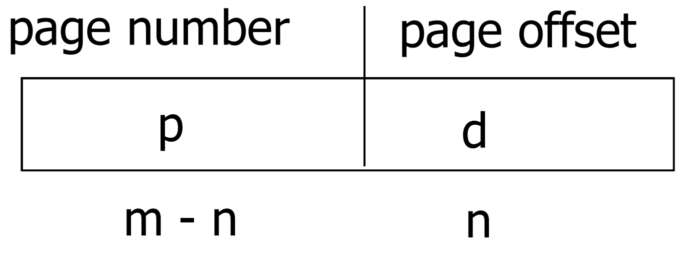


Address Translation Scheme: Example

- Consider an address of m = 16 and n = 10 bits
  - Leftmost m-n bits are page number
  - Rightmost n bits are the offset

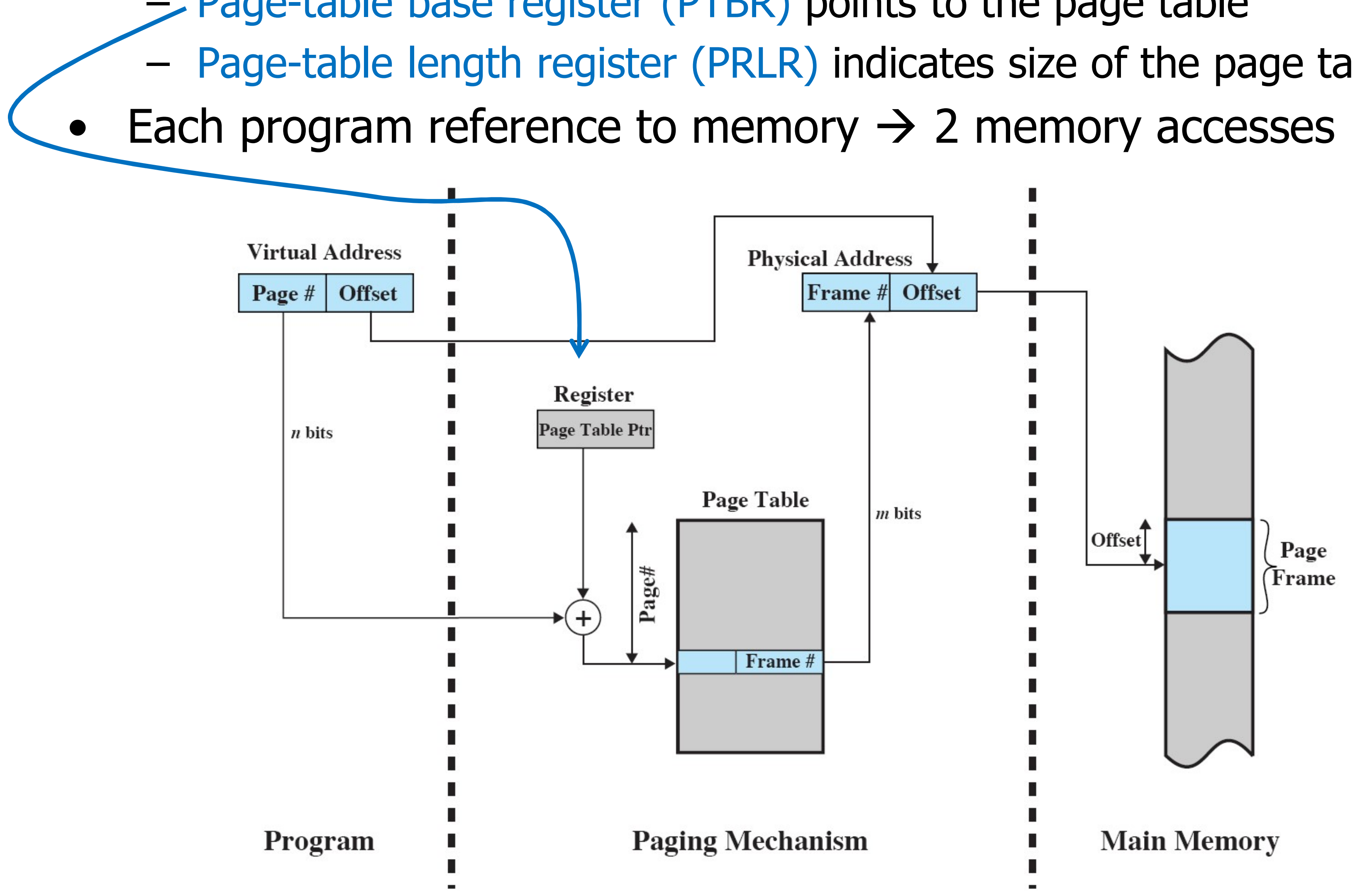
- Given a logical address 0000010111011110
  - Page number is 1 and offset 478

- Suppose this page is residing in main memory frame 6 (000110)
  - Physical address is frame number 6, offset 478



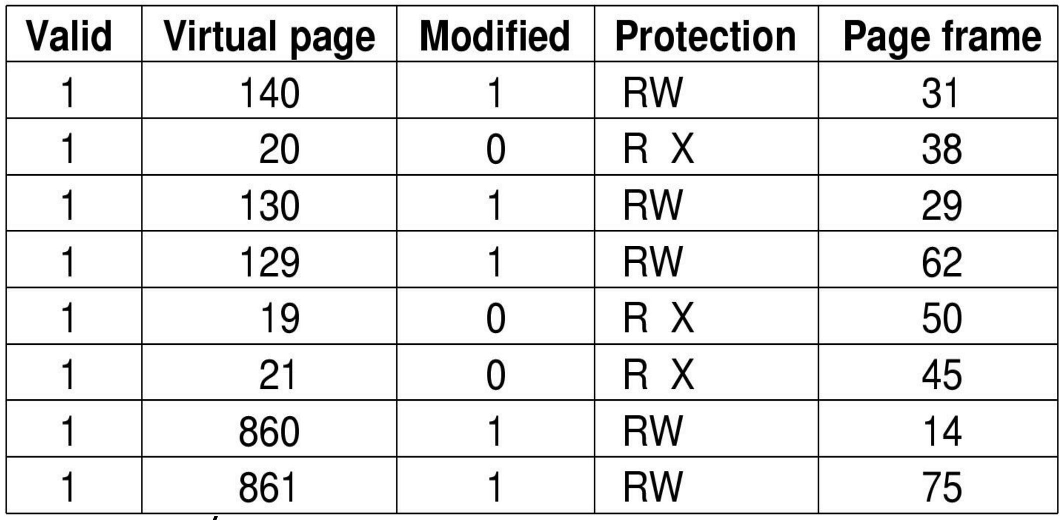
Implementation of Page Table – Main Memory

- Page table is kept in main memory
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PRLR) indicates size of the page table
- Each program reference to memory → 2 memory accesses



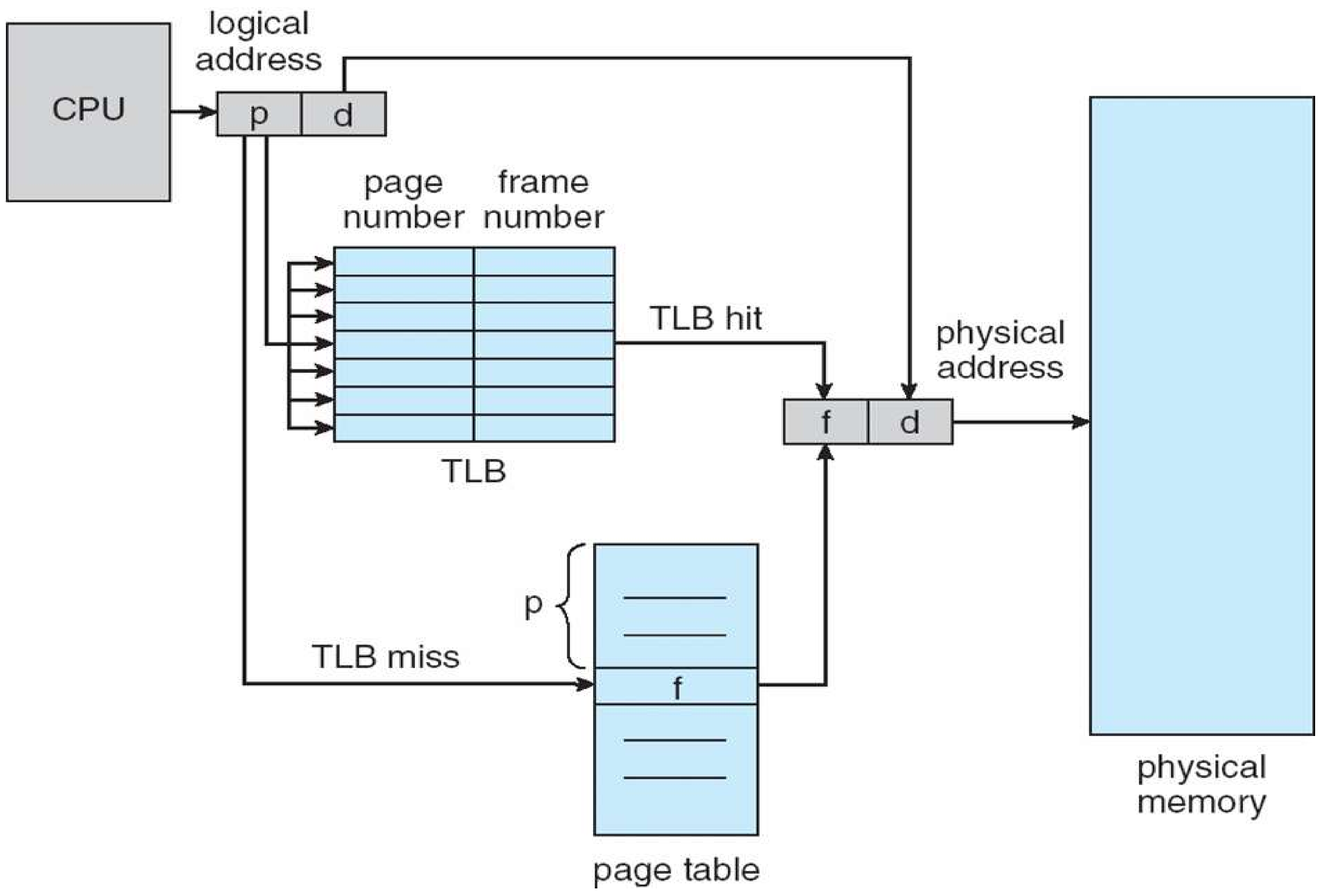
Implementation of Page Table – Associative Memory

- Translation Lookaside Buffers (TLBs)
  - Special fast-lookup hardware cache
  - Parallel search (cache for page table)
  - Resides usually inside the MMU
- Address translation (P, O)
  - If P is in associative register (hit)
  - Get frame # from TLB
  - Else get frame # from page table in m



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Paging Hardware With TLB



Access Time With Cache Registers

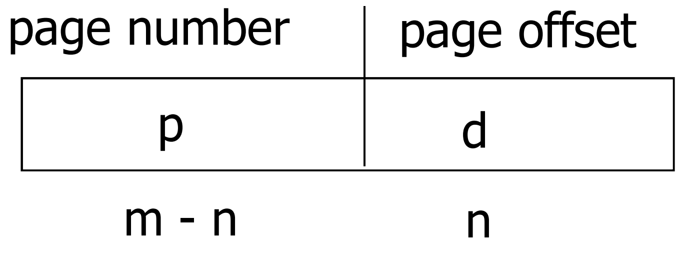
- Associative Lookup = ε time units (fraction of microsecond)
  - Can be < 10% of memory access time
- **Hit ratio** (= α)
  - Percentage of times a page number is found in associative registers
- Effective Access Time = $(1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$
  - Assume memory cycle time is 1 microsecond
- Example:
  - Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - Effective access time = 0.80 * 120 + 0.20 * 220 = 140 ns

Single-Level Page Table Limitation

- Consider a system with
  - 32-bit logical address space
  - 4KB frame size

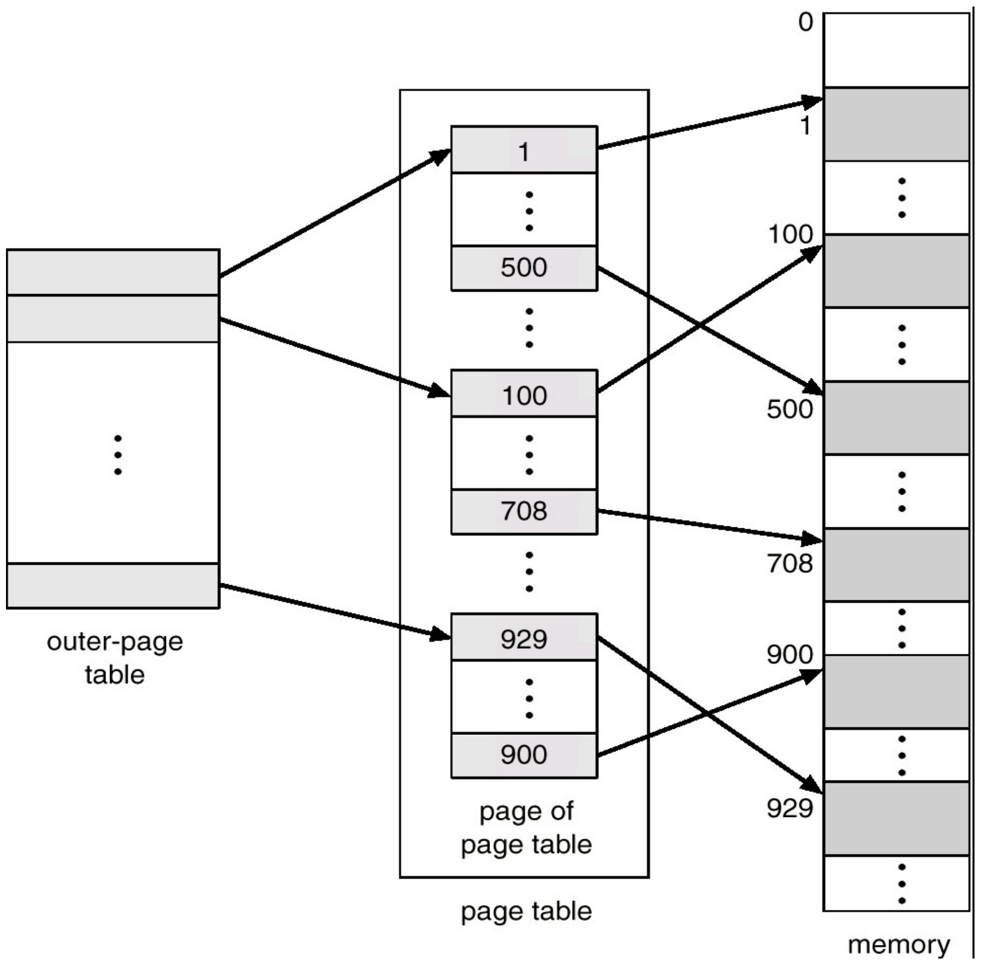
- How many entries will be in the (single-level) page table?
  - 2^{20} (1 million) entries

- Each entry of page table consists of 4 bytes
  - What will be the size of (single-level) page table?
    - ➢ 4MB



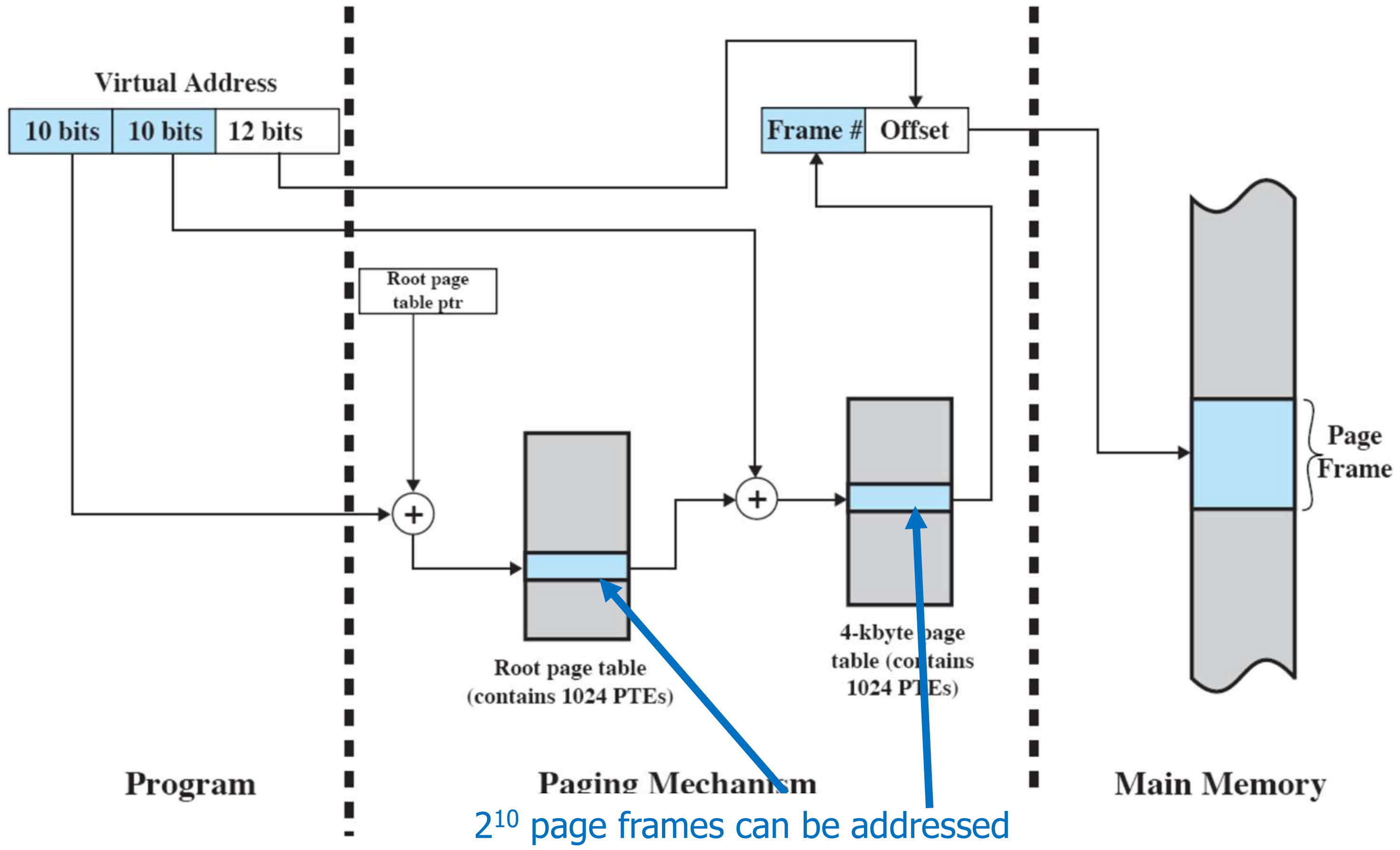
Two-Level Page-Table Scheme

- Page-table may be large, i.e., occupy several pages/frames itself



Address Translation in Two-Level Memory

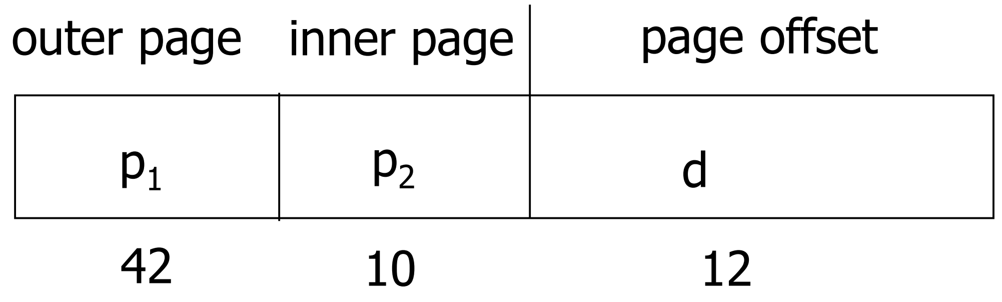
- Assume byte level addressing
  - 2^{32} bytes = 2^{22}kbytes = 2^{12}Mbytes = 4Gbytes can be supported



2^{10} page frames can be addressed

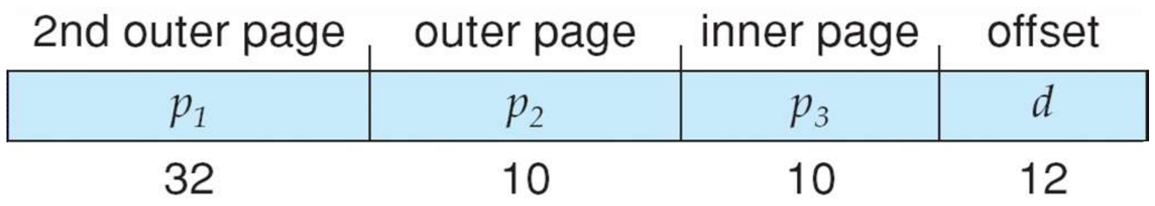
Multi-Level Page Table

- Even two-level paging scheme might not be sufficient
- Consider a system with 64-bit logical address space
  - Frame size is 4 KB (2^{12})
- How many entries in single-level page table?
  - Page table has 2^{52} entries
- How many entries in two-level page table



Multi-Level Page Table

- How many entries in three level page table?

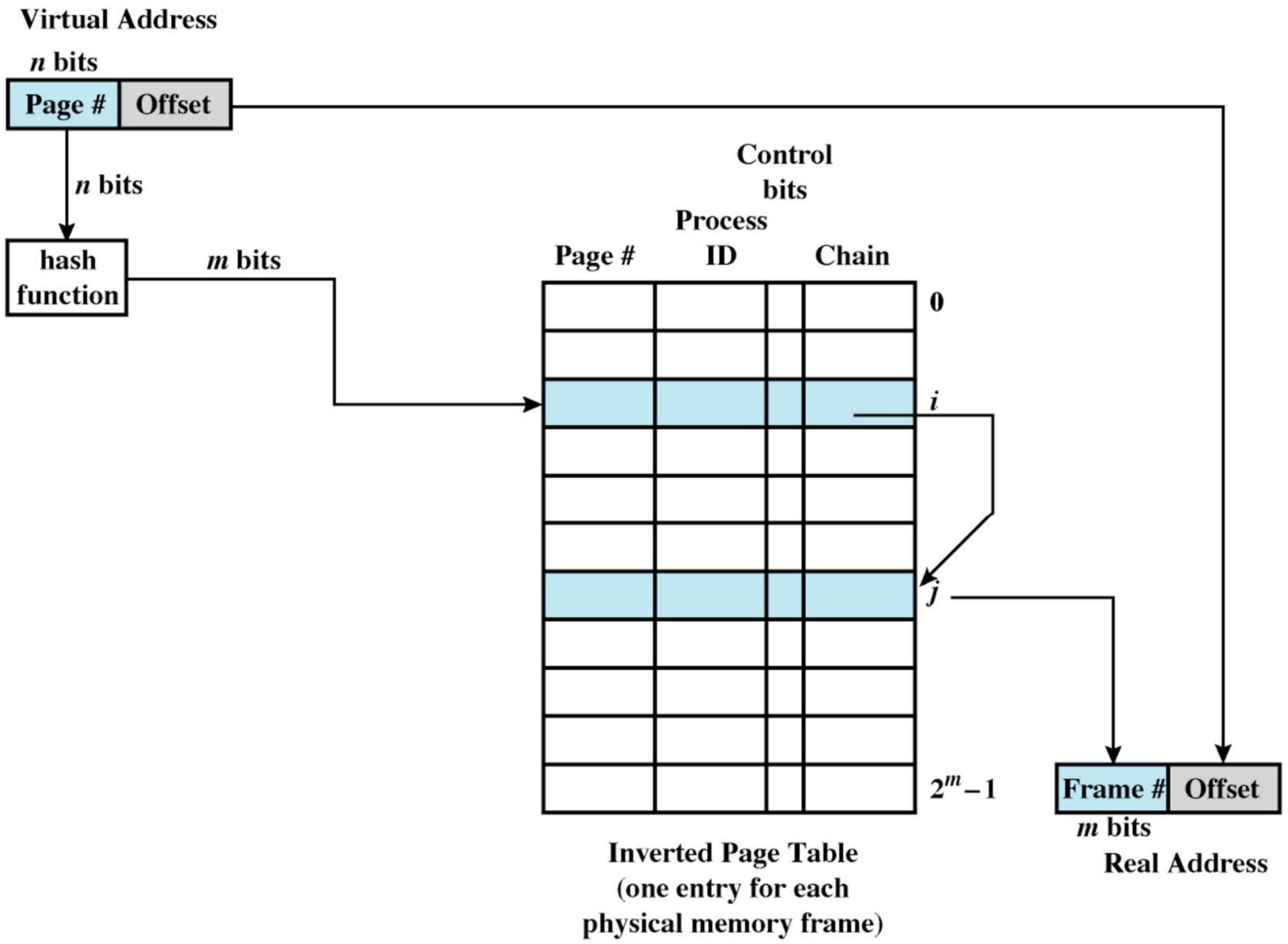


- How many levels are require such that each page table fits within a single frame?

Inverted Page Table

- One entry for each page frame of real memory
- Entry consists of
  - Virtual address of the page stored in that real memory location,
  - With information about the process that owns that page
- Decreases memory needed to store each page table
- But increase in time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Example



Chapter Overview

- Background
- Memory allocation schemes
  - Fixed partition
  - Dynamic partition
  - Paging
- Virtual memory

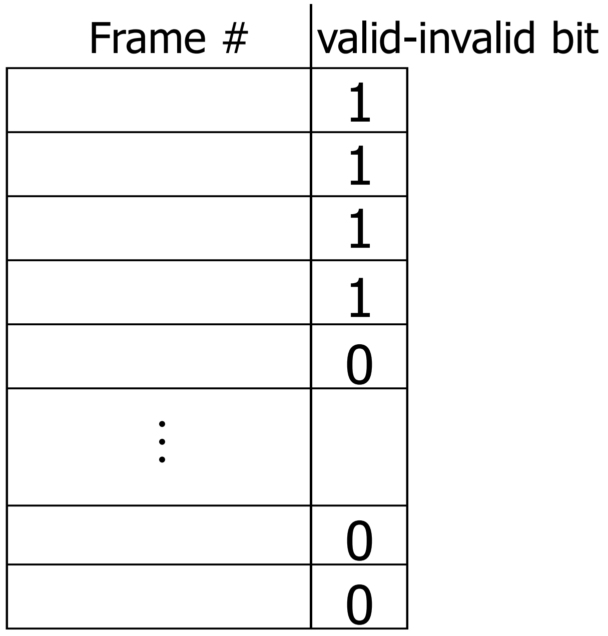
Execution of a Program: Virtual Memory Concept

Main memory = cache of the disk space

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- Whenever an address is needed that is not in main memory
  - Generation of a page-fault interrupt
  - OS places the process in blocking state and issues a disk IO request
  - Another process is dispatched

Valid-Invalid (Present/Absent) Bit

- With each page table entry a valid–invalid bit is associated
  - 1 → in-memory
  - 0 → not-in-memory (Initially 0)



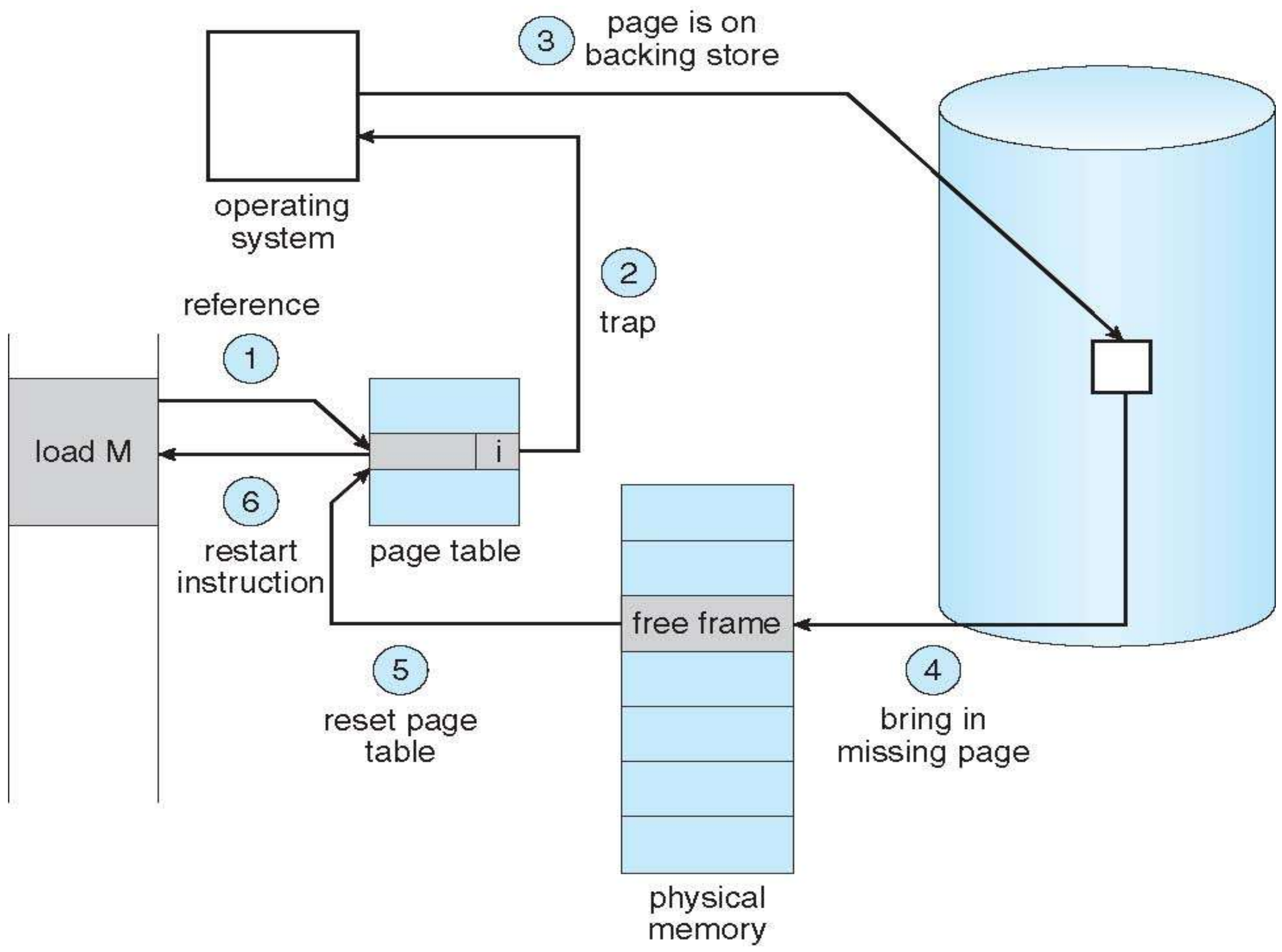
Page table

- During address translation, if valid-invalid bit in page table entry is 0
  - Page fault interrupt to OS

Page Fault and Address Translation

In response to page-fault, paging hardware issue trap to OS

- Check validity of memory reference
- Get empty frame (swap out that page?)
- Swap in page into frame
- Reset tables, validation bit
- Restart instruction



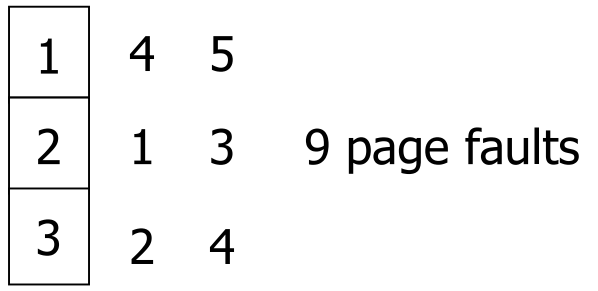
If There is No Free Frame?

Page replacement

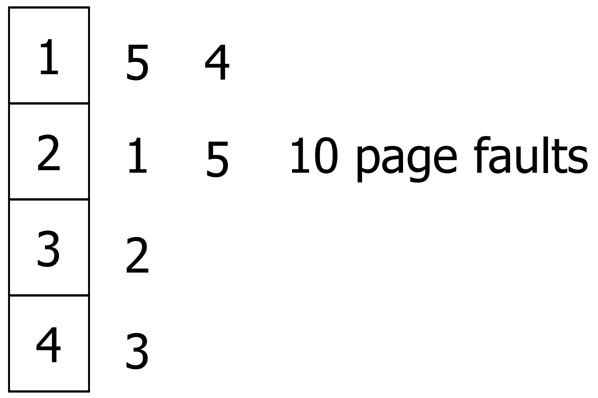
- Goal: Algorithm which will result in minimum number of page faults
- Page fault forces choice
  - Which page must be removed
  - Make room for incoming page
- Modified page must first be saved
  - Unmodified just overwritten (use dirty bit to optimize writes to disk)
- Better not to choose an often used page
  - Will probably need to be brought back in soon

First-In-First-Out (FIFO) Replacement Algorithm

- Can be implemented using a circular buffer
- Example: Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 frames



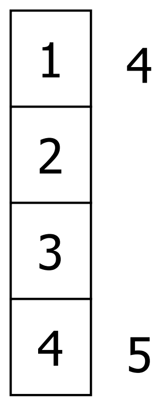
  - 4 frames



- Belady’s Anomaly: More frames, sometimes more page faults
  - Problem: replaces pages that will be needed soon

Optimal Replacement Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

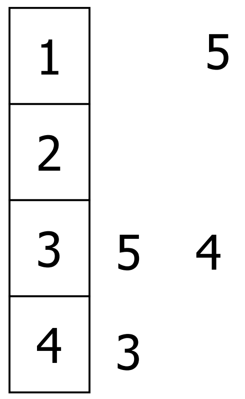
- How can we know this information?
- Standard for measuring how well other algorithms perform

Least Recently Used (LRU) Replacement Algorithm

- Idea: Replace 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
- Implementation
  - Tag each page with the time of last reference
  - Use a stack
- Problem
  - High overhead
  - OS kernel involvement at every memory reference!!!
    - ➢ If hardware support not available

LRU Algorithm

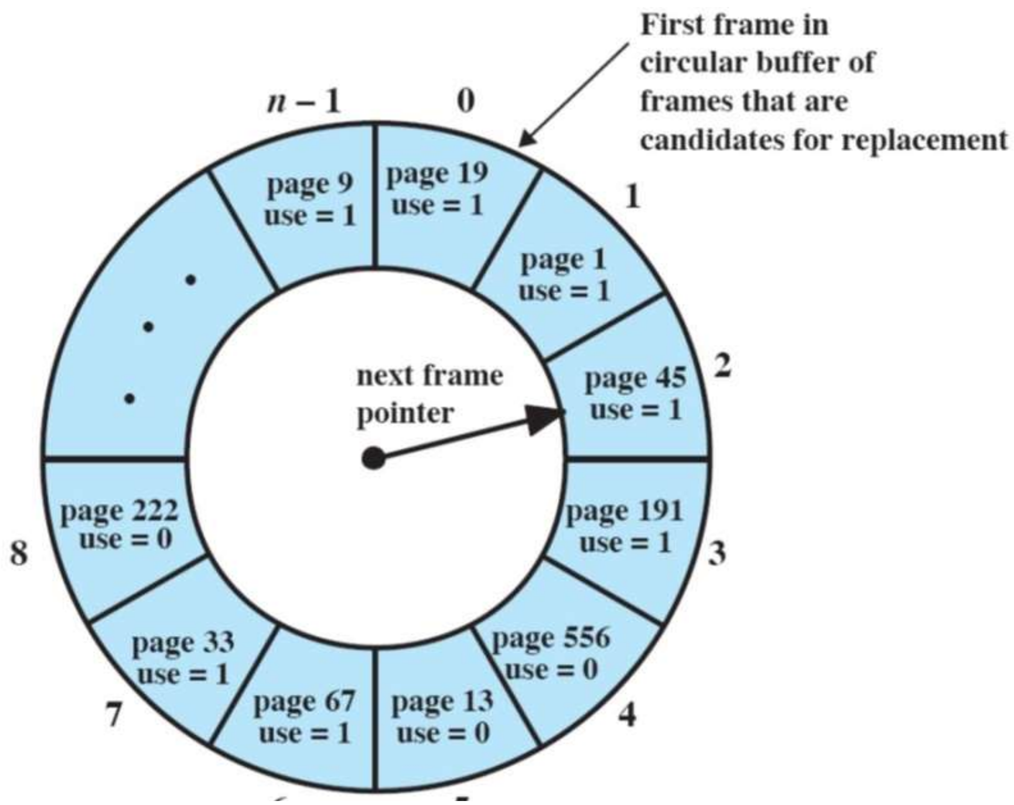
- 4 frame example
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

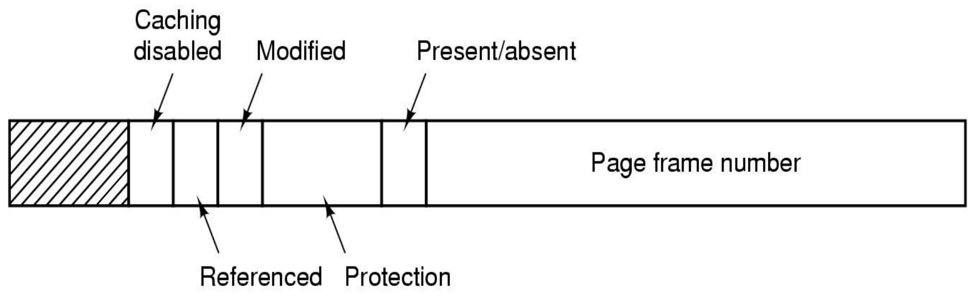


8 page faults

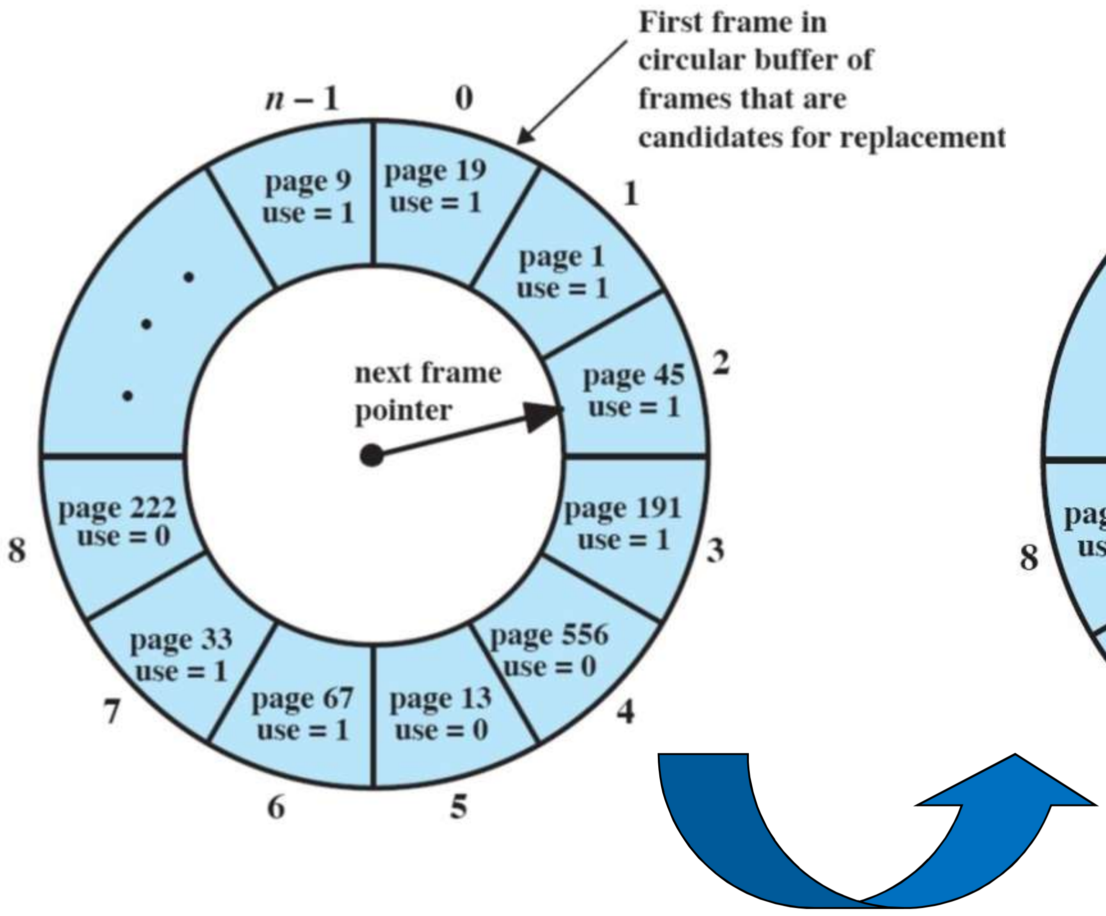
LRU Approximations: Clock/Second Chance

- Uses reference (use) bit
  - Initially 0
  - When page is referenced, set to 1 by HW
- To replace a page
  - The first frame encountered with use bit 0 is replaced
  - During the search for replacement, each use bit set to 1 is changed to 0 by OS
- Note: if all bits set → FIFO

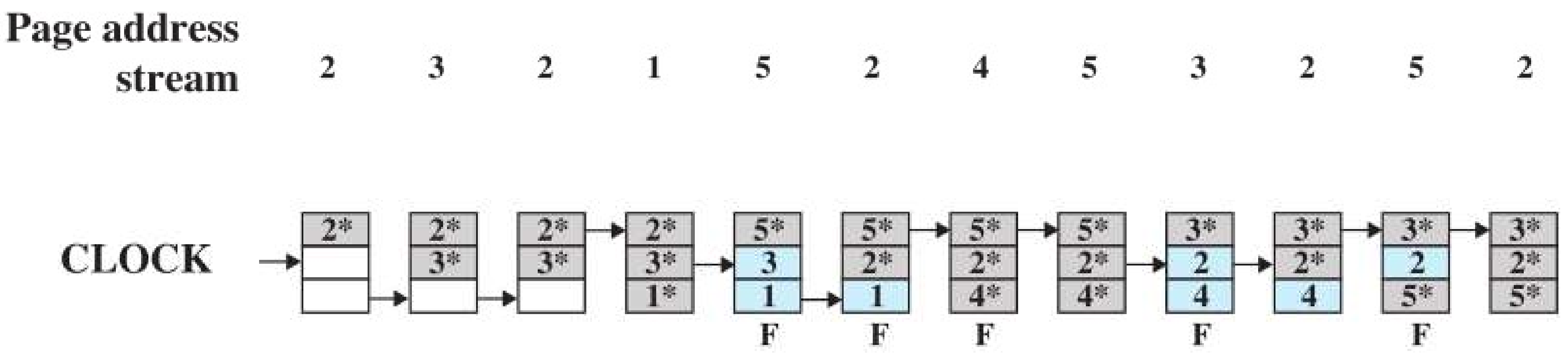




Example Clock Policy

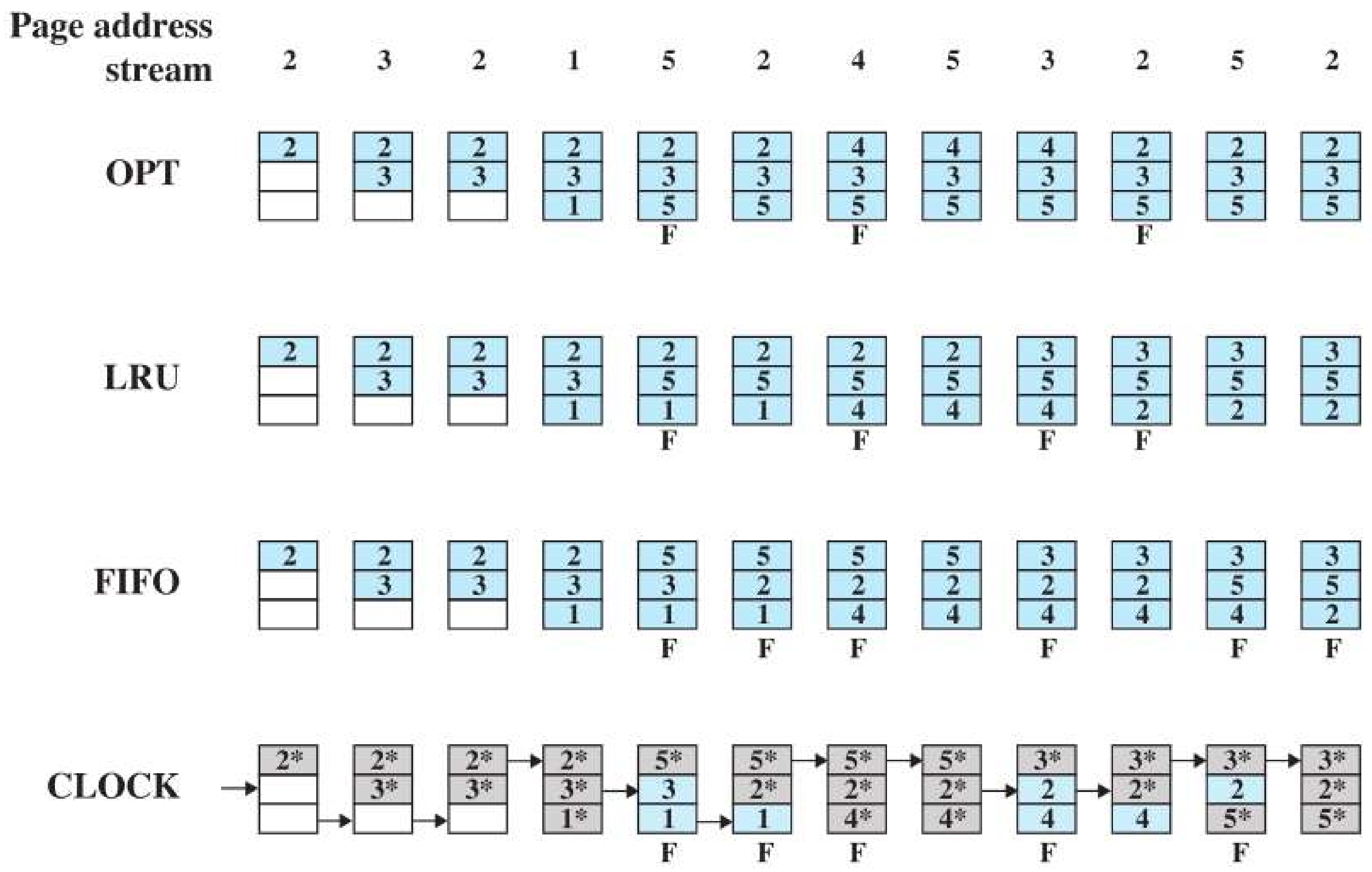


Example Clock Policy



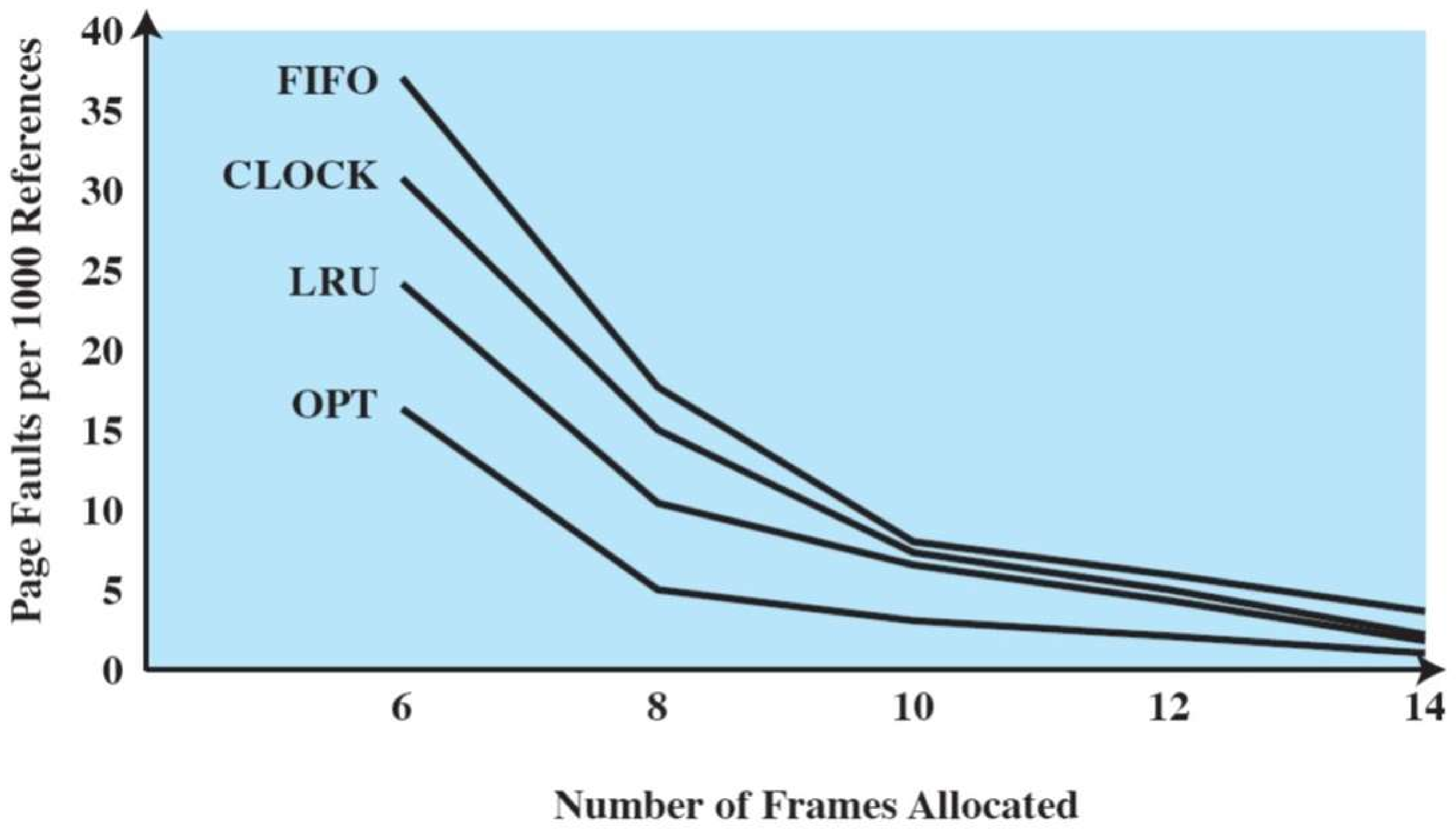
F = page fault occurring after the frame allocation is initially filled

Comparison of Basic Replacement Algorithms



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

Comparison of Basic Replacement Algorithms



Enhancement of Clock/Second Chance

- Clock algorithm can be made more powerful by using more bits
  - Use reference bit (R) and modify bit (M) as an ordered pair

- With R and M bits, the following four classes are possible
  - Class 0: Neither recently used nor modified (R=0, M=0)
    - ➢ Best page to replace
  - Class 1: Not recently used but modified (R=0, M=1)
    - ➢ The page will need to be written out before replacement
  - Class 2: Recently used but clean (R=1, M=0)
    - ➢ Probably page will be used again soon
  - Class 3: Recently used and modified (R=1, M=1)
    - ➢ Probably page will be need soon and disk I/O is required to write page

- Replacement strategy
  - Replace the first page encountered in the lowest non empty class

Enhancement of Clock/Second Chance

- When a process starts
  - All its page entries are marked as not referenced and not modified
- When a page is referenced
  - The R (reference) bit is set
- When the page is later written to
  - The M (modified) bit is set
- Difference to clock algorithm
  - Preference are given to pages that have been modified to reduce I/O

Any Question So Far?