Operating Systems & Systems Programming
Module 5
Memory Management

Dr. Vikash

Jaypee Institute of Information Technology, Noida

Overview

Objectives

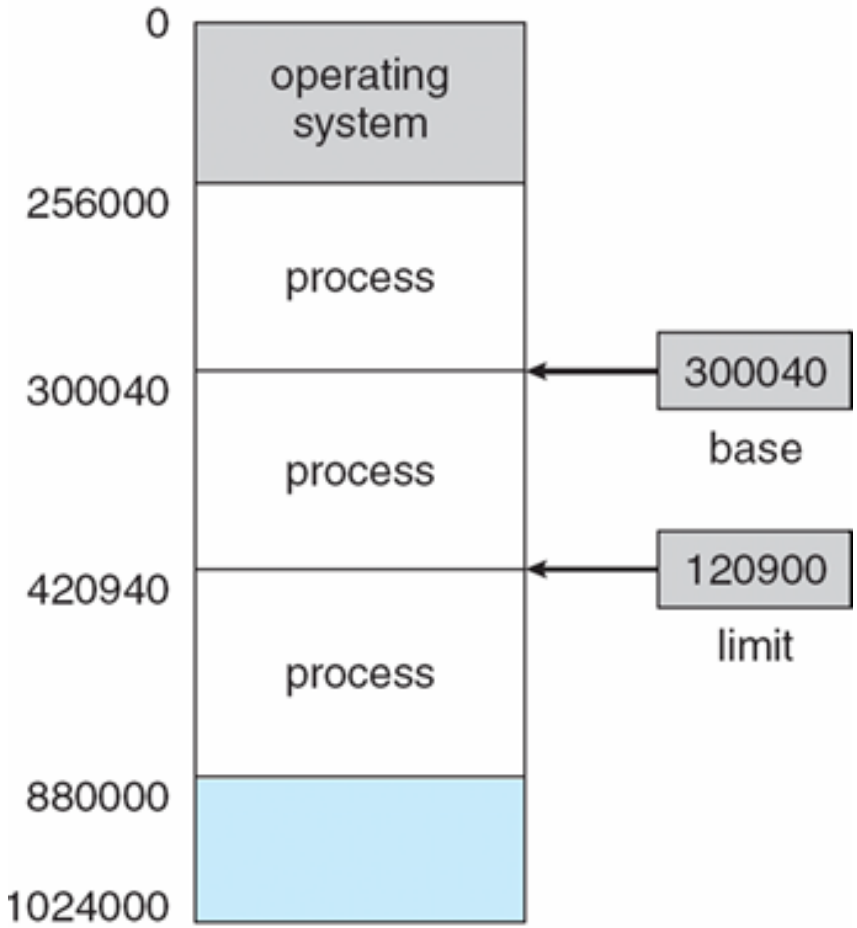
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Translating Logical Address into Physical Address

CPU always generates a logical address.

A physical address is needed to access the main memory.

Following steps are followed to translate logical address into physical address-

Step 1.

- The translation scheme uses two registers that are under the control of operating system.
- During context switching, the values corresponding to the process being loaded are set in the registers.
- Relocation Register stores the base address or starting address of the process in the main memory.
- Limit Register stores the size or length of the process.

Translating Logical Address into Physical Address

Step 2.

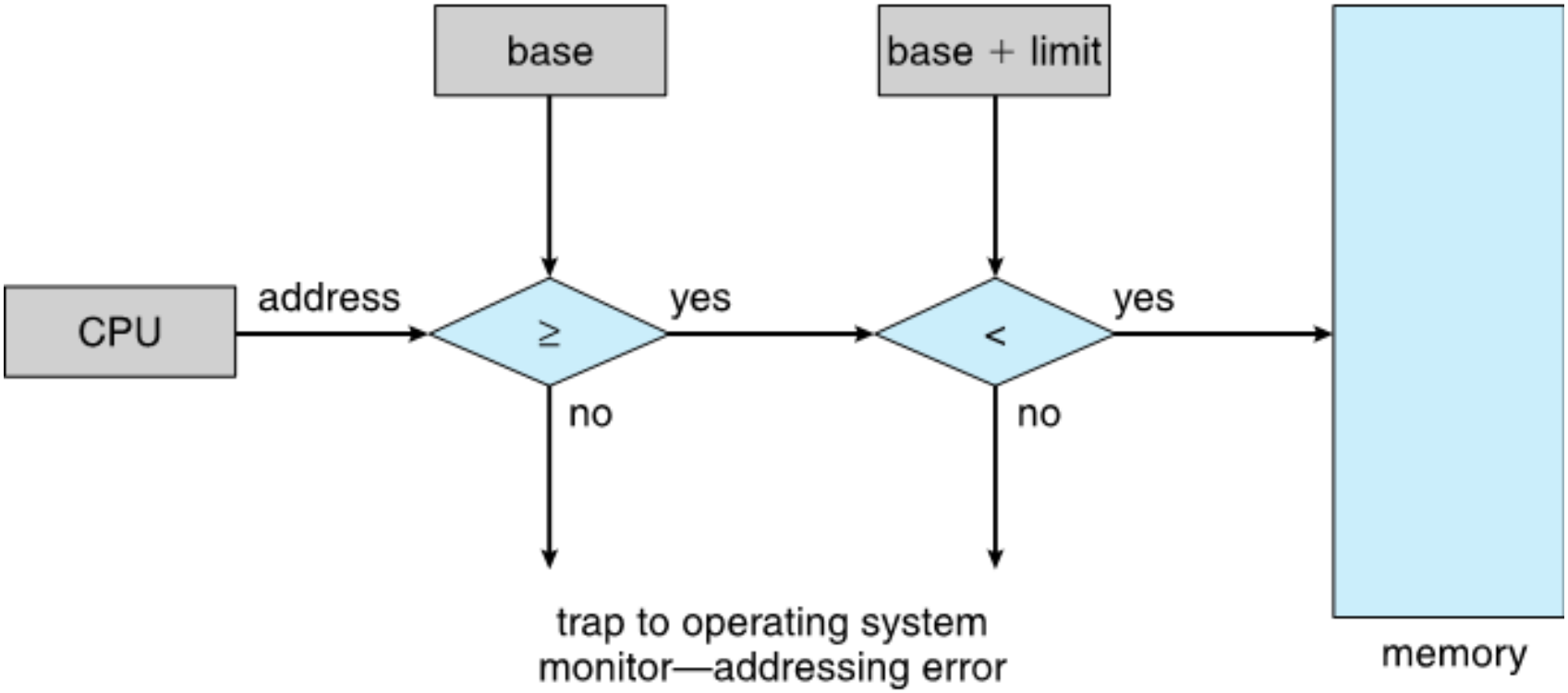
- CPU generates a logical address containing the address of the instruction that it wants to read.

Step 3. The logical address generated by the CPU is compared with the limit of the process.

Now, two cases are possible-

1. Generated Address $\geq$ Limit
   - If address is found to be greater than or equal to the limit, a trap is generated.
   - This helps to prevent unauthorized access.
2. Generated Address $<$ Limit
   - The address must always lie in the range [0, limit-1].
   - If address is found to be smaller than the limit, then the request is treated as a valid request.
   - Then, generated address is added with the base address of the process.
   - The result obtained after addition is the address of the memory location storing the required word.

Translating Logical Address into Physical Address



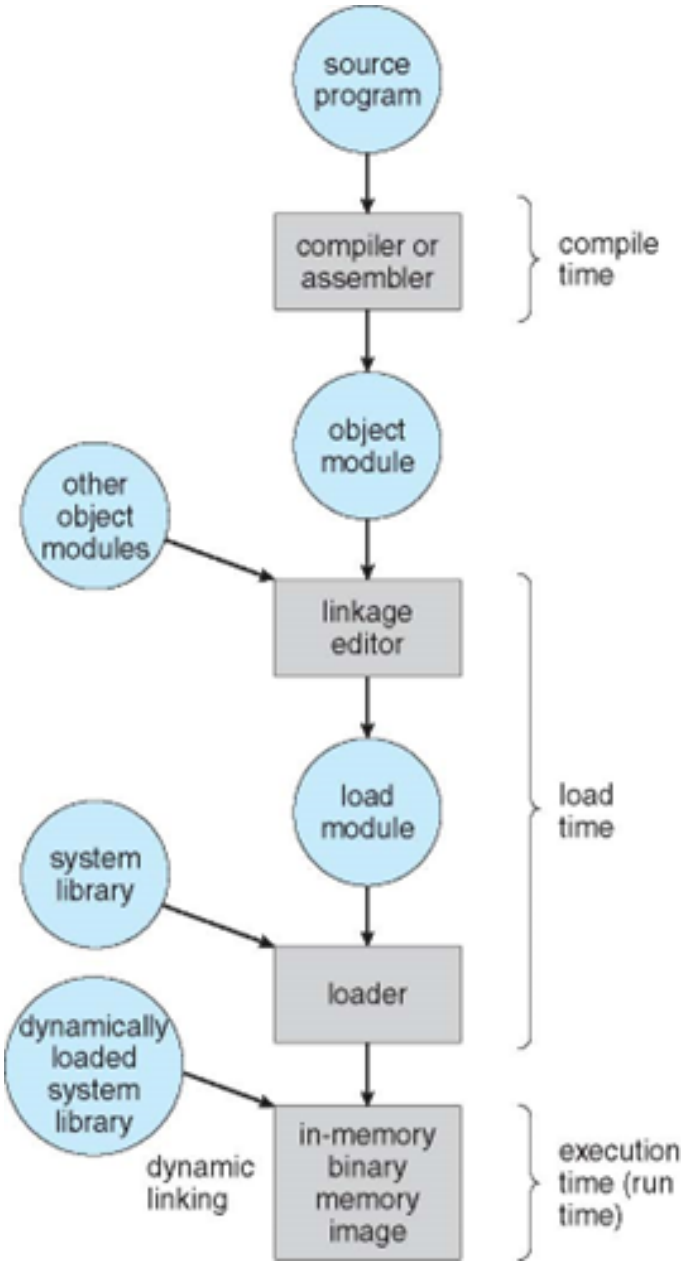
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be ?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses bind to relocatable addresses i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses i.e. 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another [Need hardware support for address maps (e.g., base and limit registers)]

Multistep Processing of a User Program



Logical vs. Physical Address Space

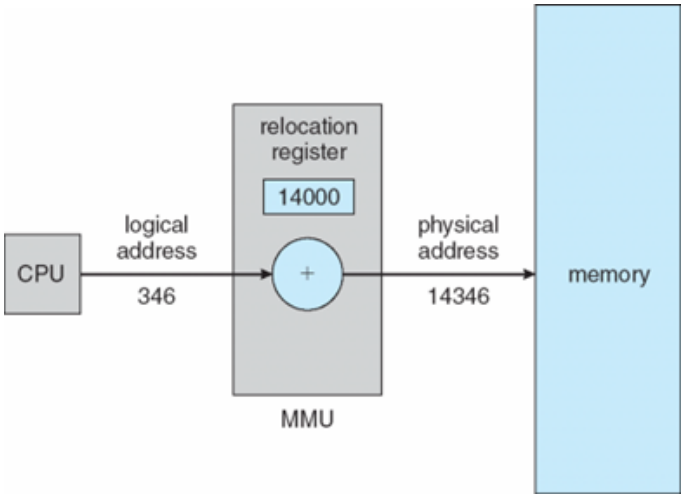
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



Dynamic Linking

- **Static linking** - system libraries and program code combined by the loader into the binary program image
- Dynamic linking - linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

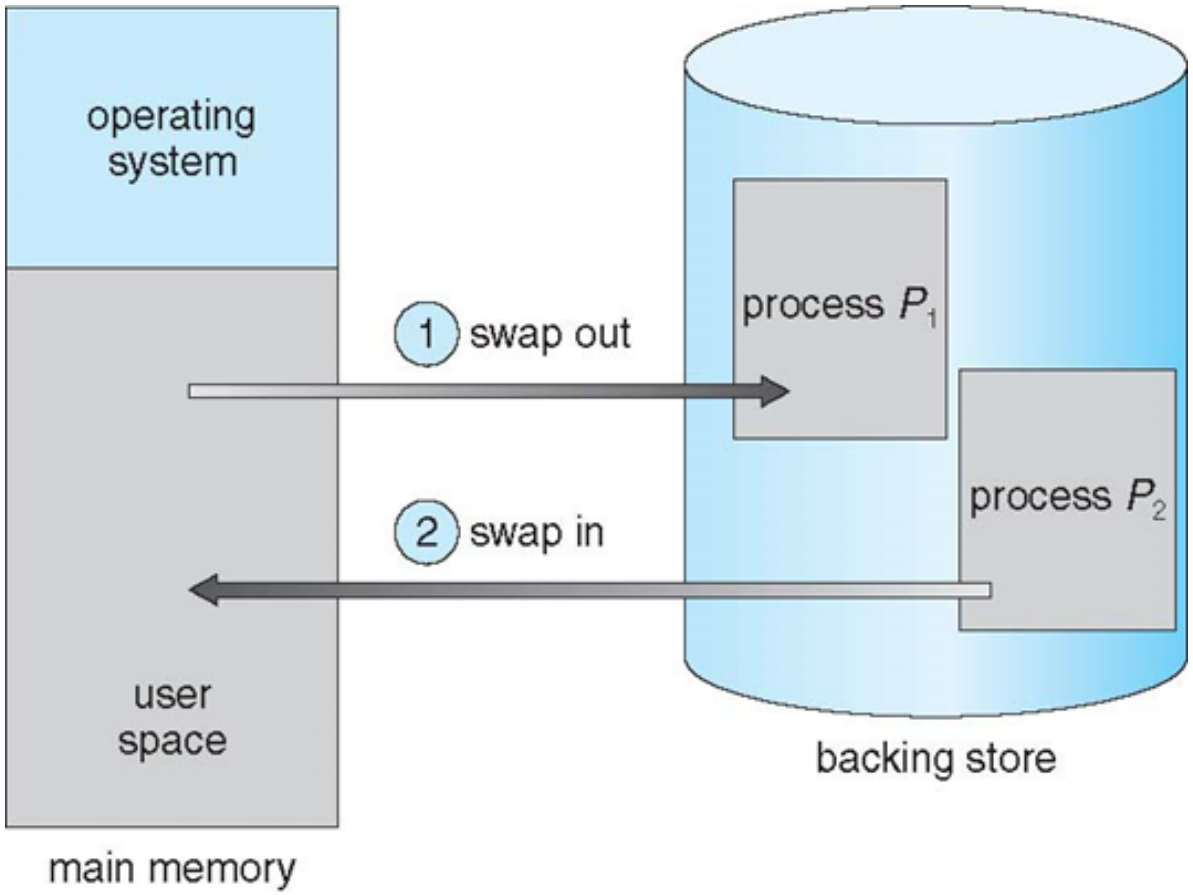
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** - fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** - swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

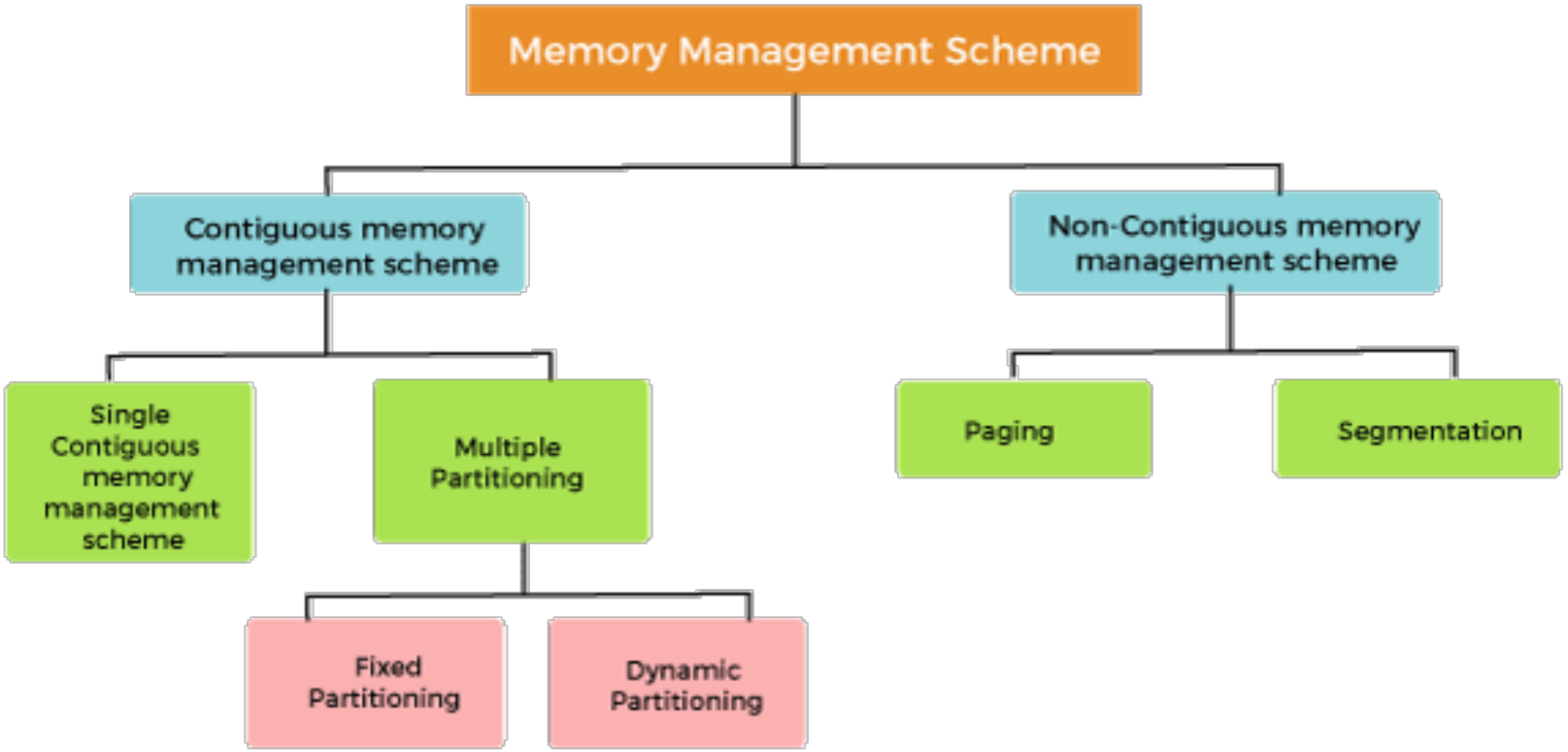
Swapping

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Memory Management Techniques



Classification of memory management schemes

Memory Management Techniques

Technique	Description	Strengths	Weakness
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead	Inefficient use of memory due to internal fragmentation, maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory	Inefficient use of processor due to the need for compaction to consider external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available not necessary contiguous frames.	No external fragmentation	A small amount of internal fragmentation
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partition that need to be contiguous.	No internal fragmentation improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Non-resident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all the segments of a process non- resident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming' large virtual address space; protection and sharing support	Overhead of complex memory management

Continuous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions: Fixed and Dynamic**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses each logical address must be less than the limit register
  - MMU maps logical address dynamically
  - Can then allow actions such as kernel code being **transient** and kernel changing size

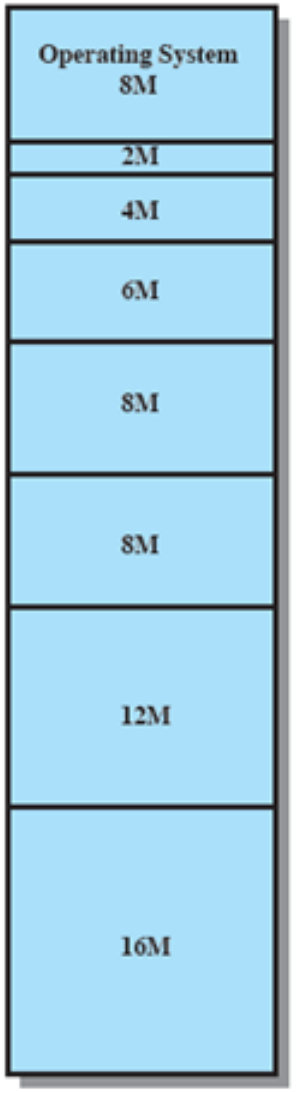
Fixed Partitioning

- **Equal-size partitions**: 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 out a process if all partitions are full and no process is in the Ready or Running state
- A program may be too big to fit in a partition, program needs to be designed with the use of **overlays**.
- **Disadvantage:**
  - Main memory utilization is inefficient any program, regardless of size, occupies an entire partition
  - The number of partitions specified at system generation time limits the number of active processes in the system
  - **internal fragmentation**: wasted space due to the block of data loaded being smaller than the partition

(a) Equal-size partitions

Unequal Size of Fixed Partition

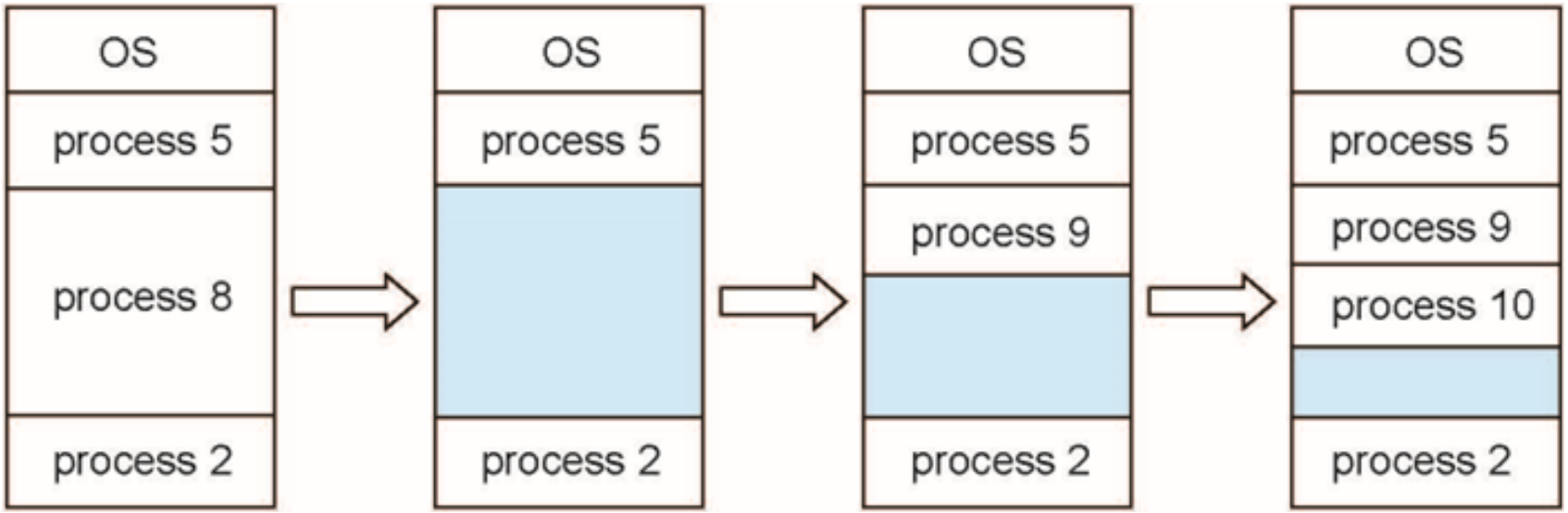
- Using unequal size partitions helps lessen the problems
  - programs up to 16M can be accommodated without overlays
  - partitions smaller than 8M allow smaller programs to be accommodated with less internal fragmentation



(b) Unequal-size partitions

Multiple Partition Allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process needs)
- **Hole** - block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocation partitions b) free partitions (hole)



Dynamic Memory Allocation

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBMs mainframe operating system, OS/MVT
- **Disadvantage:**
  - **External Fragmentation:** memory becomes more and more fragmented, memory utilization declines
  - **Compaction:** technique for overcoming external fragmentation
  - OS shifts processes so that they are contiguous free memory is together in one block.
  - time consuming and wastes CPU time

Dynamic Memory Allocation

How to satisfy a request of size n from a list of free holes ?

- **First-fit:** Allocate the first hole that is big enough
- **Best-fit:** Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the largest hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Dynamic Memory Allocation

- Point 1: For dynamic partitioning
  - Worst Fit Algorithm works best.
  - This is because space left after allocation inside the partition is of large size.
  - There is a high probability that this space might suit the requirement of arriving processes.
- Point 2: For dynamic partitioning,
  - Best Fit Algorithm works worst.
  - This is because space left after allocation inside the partition is of very small size.
  - There is a low probability that this space might suit the requirement of arriving processes.

Exercise Problem 1

Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB. These partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order.

Perform the allocation of processes using-

1. First Fit Algorithm
2. Best Fit Algorithm
3. Worst Fit Algorithm

Exercise Problem 2

- Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?
- First-fit:
  - 212K is put in 500K partition
  - 417K is put in 600K partition
  - 112K is put in 288K partition (new partition 288K = 500K - 212K)
  - 426K must wait
- Best-fit:
  - 212K is put in 300K partition
  - 417K is put in 500K partition
  - 112K is put in 200K partition
  - 426K is put in 600K partition
- Worst-fit:
  - 212K is put in 600K partition
  - 417K is put in 500K partition
  - 112K is put in 388K partition
  - 426K must wait

Exercise Problem 2

Consider the requests from processes in given order 300K, 25K, 125K and 50K. Let there be two blocks of memory available of size 150K followed by a block size 350K. Which of the following partition allocation schemes can satisfy above requests?

A) Best fit but not first fit.

B) First fit but not best fit.

C) Both First fit & Best fit.

D) neither first fit nor best fit.

Buddy System – Memory Allocation Algorithm

- Static partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition size and process size.
- A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction.
- An interesting compromise of fixed and dynamic partitioning is the buddy system.

What is Buddies ?

- The buddy system(binary) allows a single allocation block to be split, to form two blocks half the size of parent block. These two blocks are known as buddies.
- Part of definition of 'buddy' is that the buddy of block B must be the same size as B, and must be adjacent in memory (so that it is possible to merge them later).
- The other important property of buddies, stems from the fact that in buddy system, every block is an address in memory which is exactly divisible by its size.
- So, all the 16-byte blocks are at addresses which are multiples of 16; all the 64 K blocks are at addresses which are multiples of 64 K... and so on.1

Types of BUDDY system

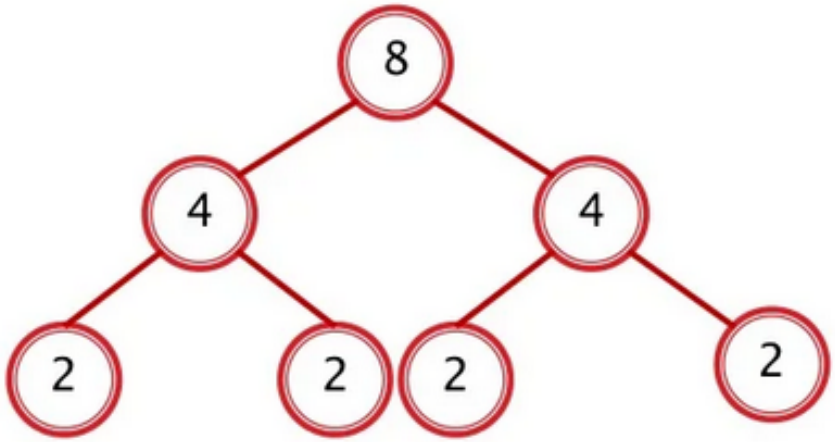
There are a number of buddy system, proposed by many researcher, which are capable to reduce the execution time and increase memory utilization.

- Binary Buddy System
- Fibonacci Buddy System
- Weighted Buddy System
- Tertiary Buddy System

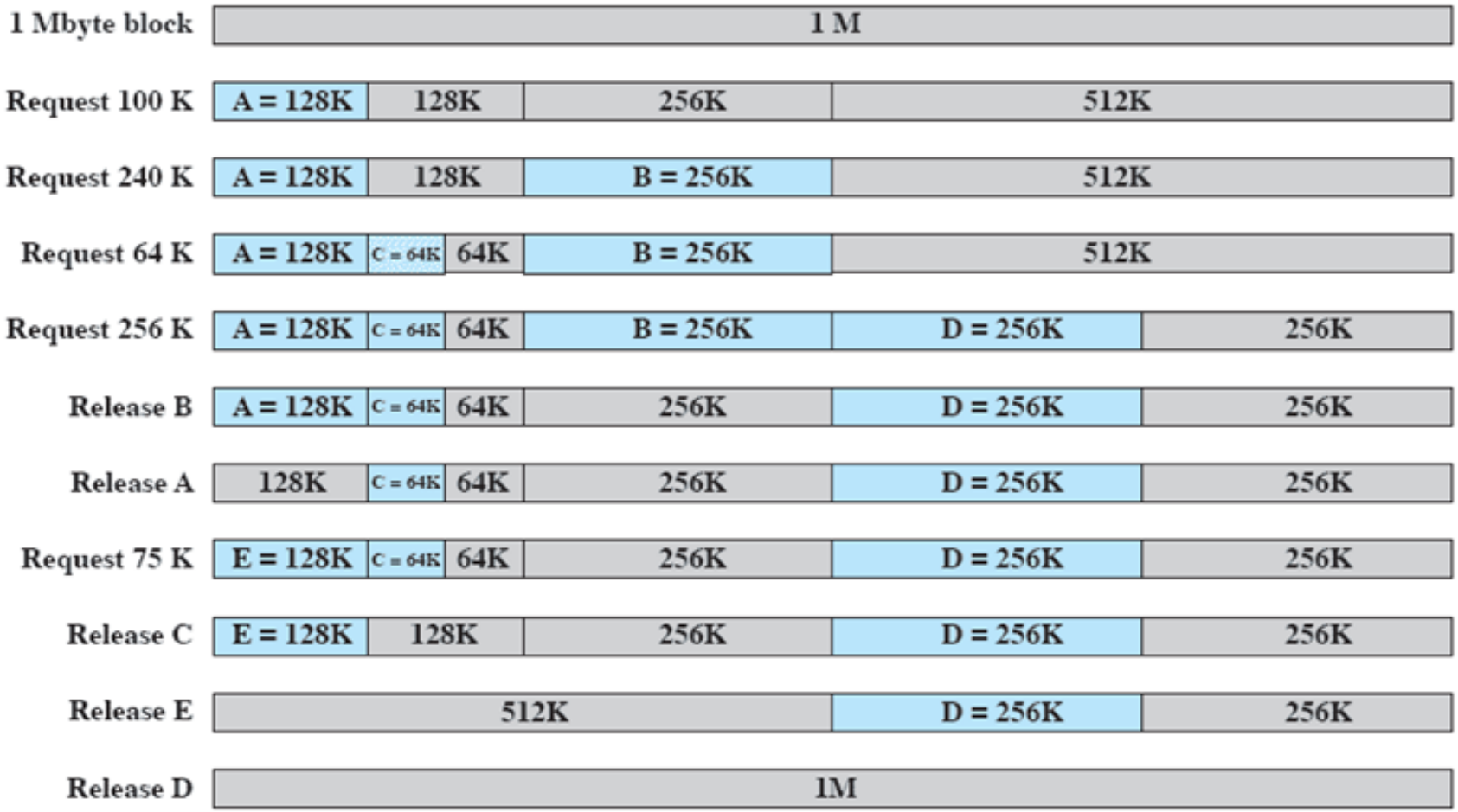
Binary Buddy System

- In binary buddy system the memory block of 2^m is into two equal parts of 2^{m-1}.
- It satisfied the following recurrence relation

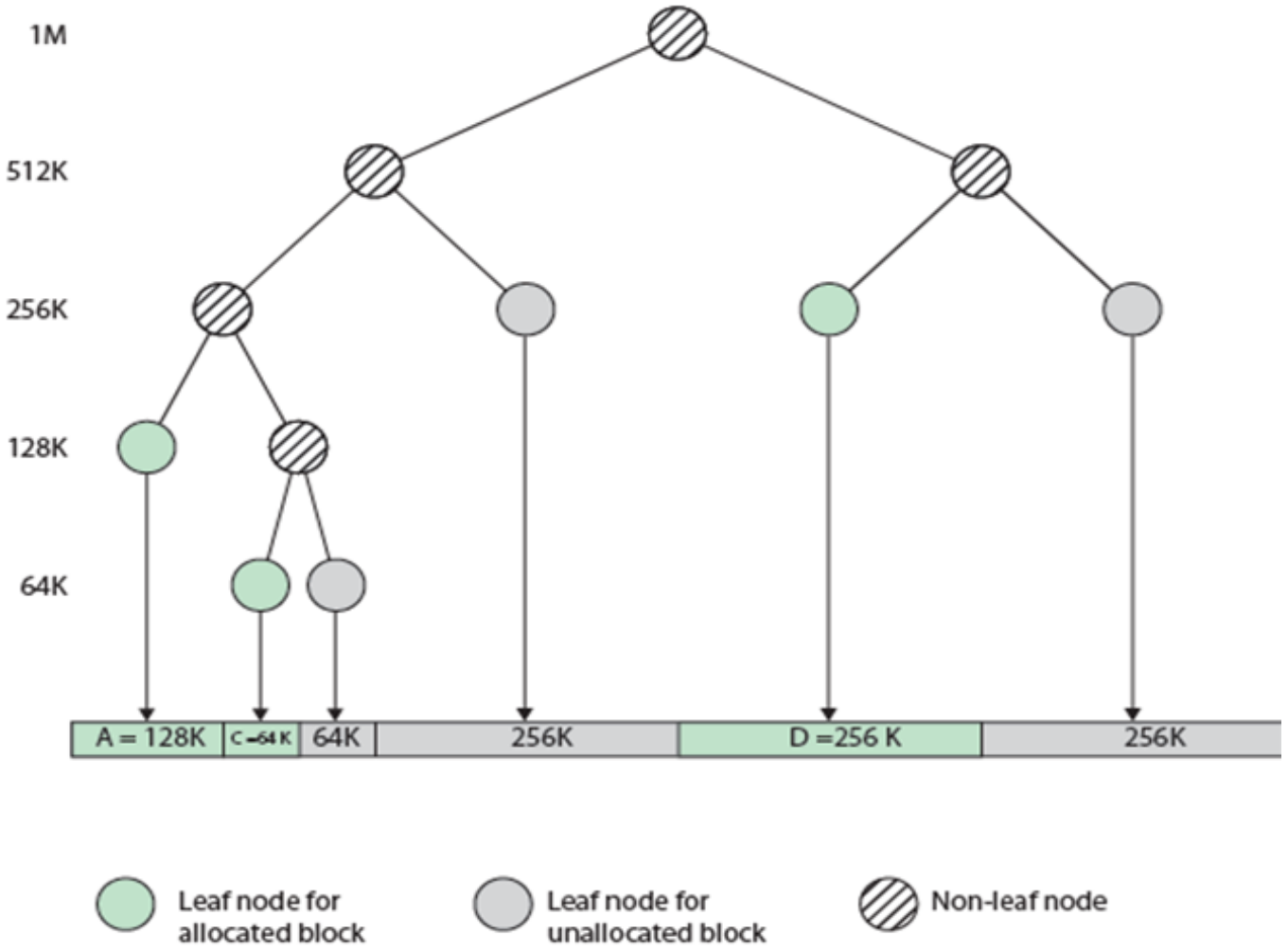
$$L_i = L_{i-1} + L_{i-1}$$

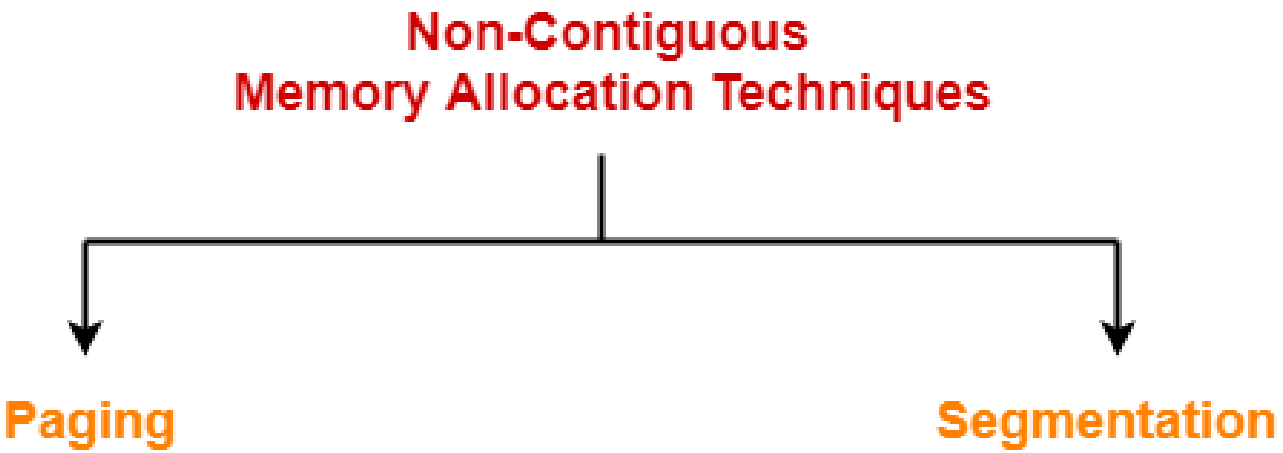


Binary Buddy System



Tree representation of buddy system



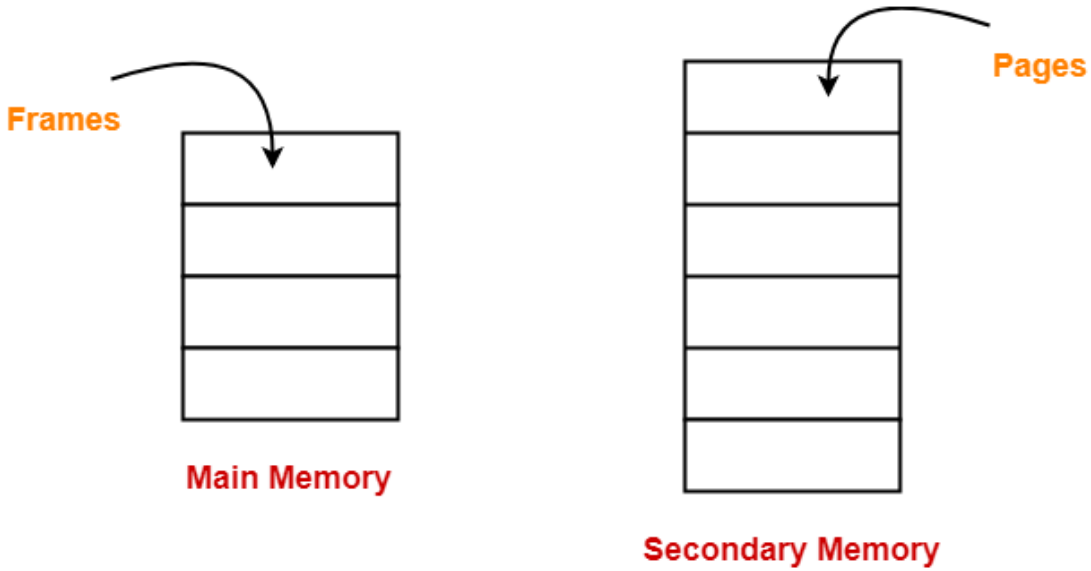

Non-Contiguous
Memory Allocation Techniques
Paging
Segmentation

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Paging

- Paging is a fixed size partitioning scheme.
- In paging, secondary memory and main memory are divided into equal fixed size partitions.
- The partitions of secondary memory are called as **pages**.
- The partitions of main memory are called as **frames**.



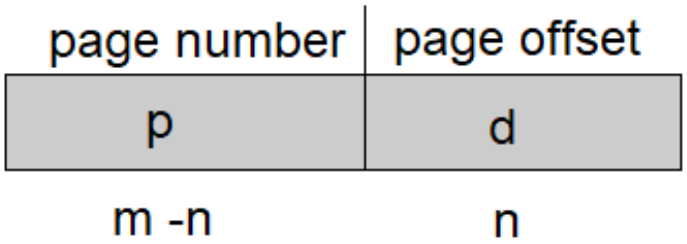
- Each process is divided into parts where size of each part is same as page size.
- The size of the last part may be less than the page size.
- The pages of process are stored in the frames of main memory depending upon their availability.

Fragmentation

- **External Fragmentation** - total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** - allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, 0.5 N blocks lost to fragmentation
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem (Latch job in memory while it is involved in I/O)
- Now consider that backing store has same fragmentation problems

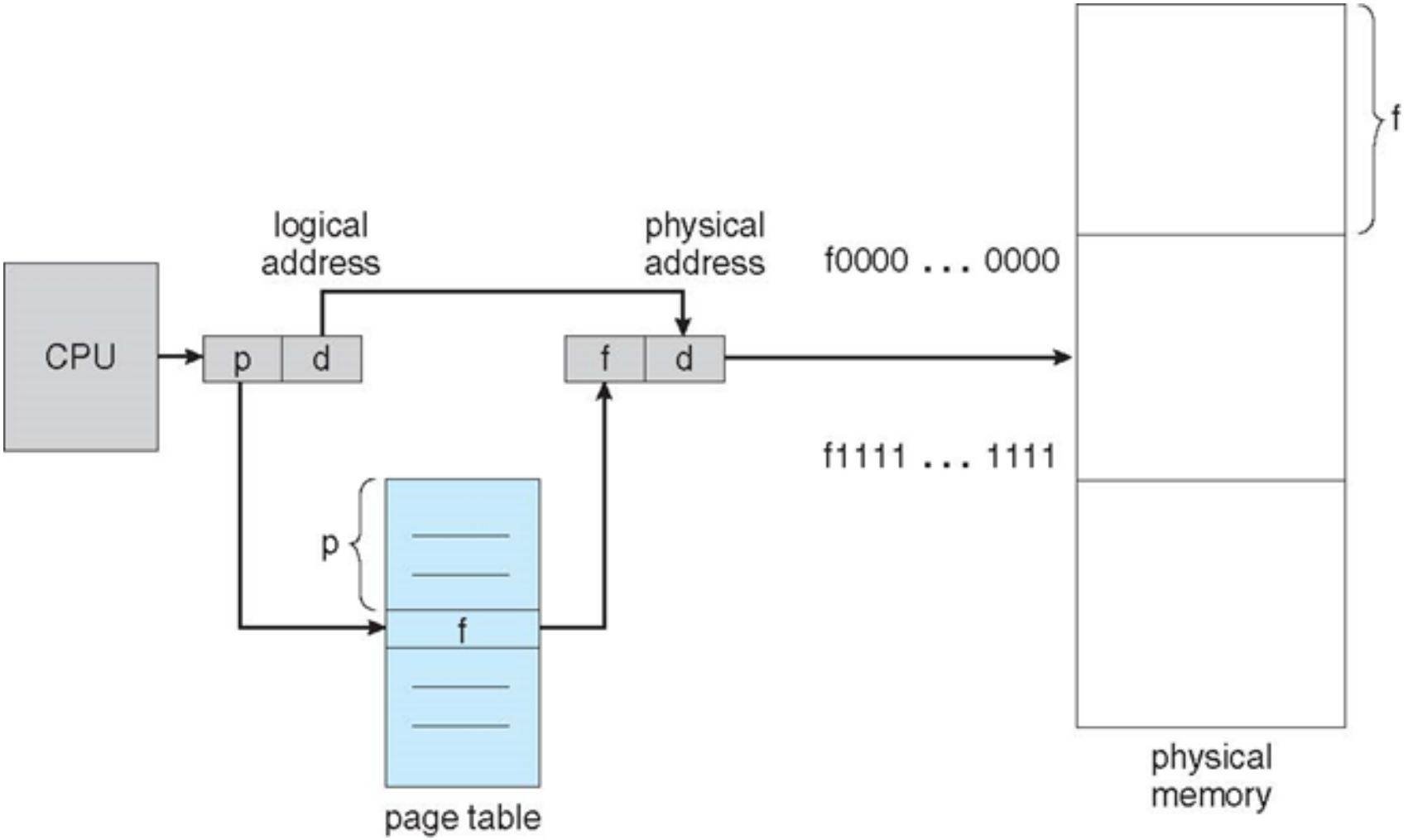
Address Translation Scheme

- Address generated by CPU is divided into:
  - **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

Paging Hardware



Advantage & Disadvantage

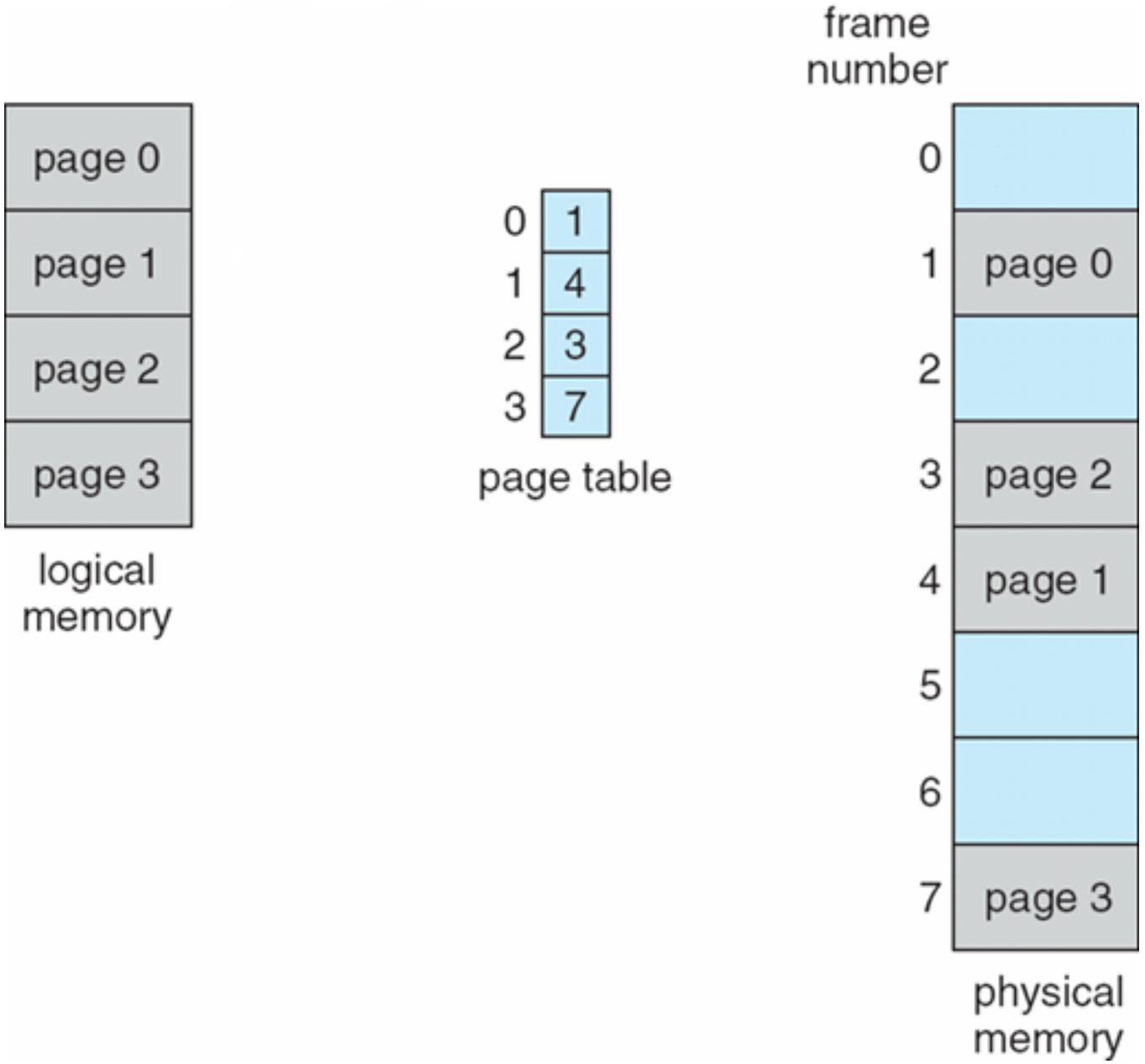
Advantage

- It allows to store parts of a single process in a non-contiguous fashion.
- It solves the problem of external fragmentation.

Disadvantage

- It suffers from internal fragmentation.
- There is an overhead of maintaining a page table for each process.
- The time taken to fetch the instruction increases since now two memory accesses are required.

Paging Model of Logical and Physical Memory



Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory or translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Page Table Entry

- A page table entry contains several information about the page.
- The information contained in the page table entry varies from operating system to operating system.
- The most important information in a page table entry is frame number.

Figure: Page Table Entry Format

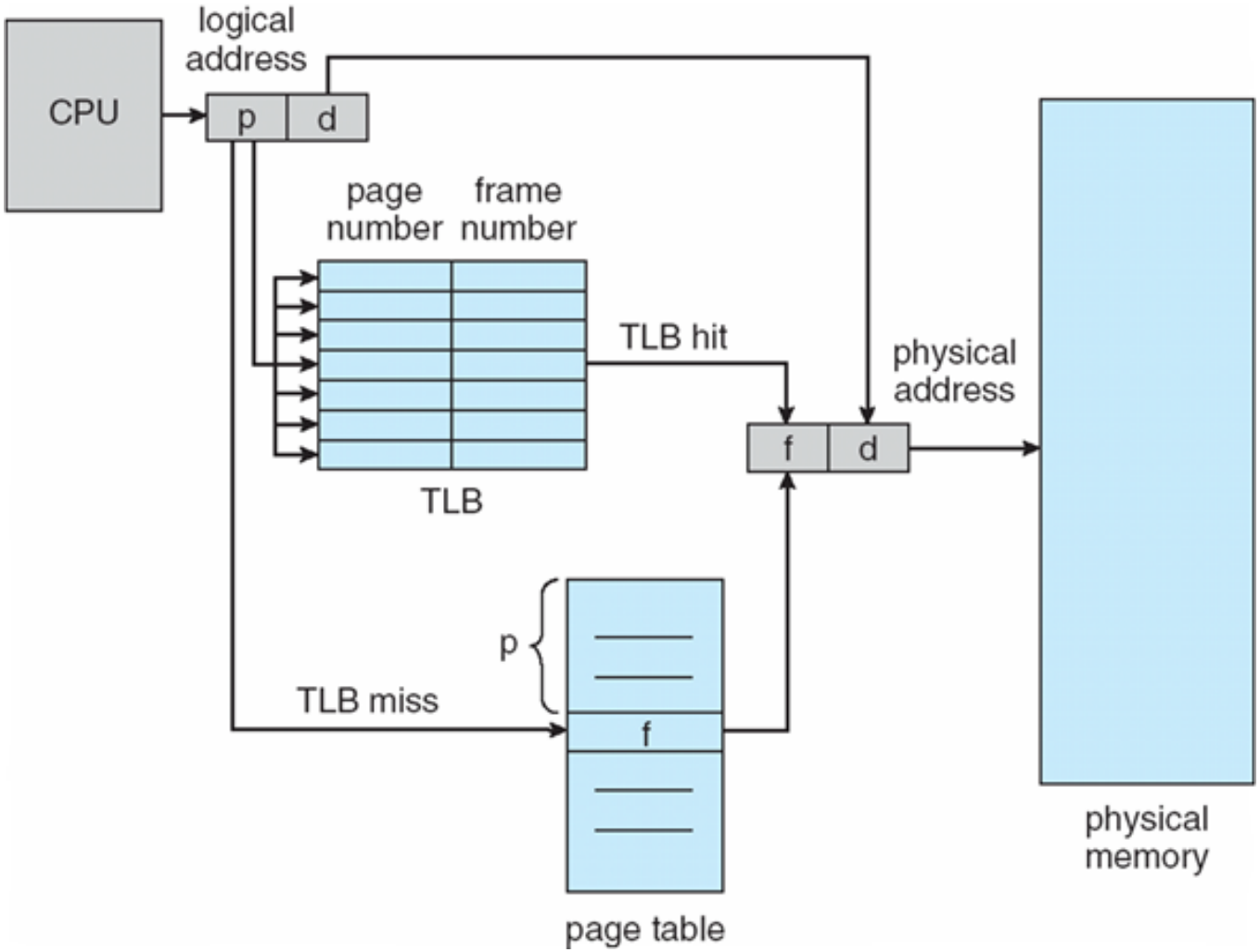
Associative Memory

- Associative memory parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

- Associative Lookup $= \eta$ time unit
  - Can be $< 10\%$ of memory access time
- Hit ratio $= \alpha$
  - Hit ratio - percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Effective Access Time

EAT = TLB hit $\times$ (TLB access time + memory access time) + TLB miss (TLB access time+page table access time + memory access time)

$$EAT = \alpha \times (t + m) + (1 - \alpha)(t + m + m)$$

$$EAT = \alpha \times (t + m) + (1 - \alpha)(t + 2m)$$

Important Formulas

For Main Memory

- Physical Address Space = Size of main memory
- Size of main memory = Total number of frames x Page size
- Frame size = Page size
- If number of frames in main memory = 2^X, then number of bits in frame number = X bits
- If Page size = 2^X Bytes, then number of bits in page offset = X bits
- If size of main memory = 2^X Bytes, then number of bits in physical address = X bits

Optimal Page Size = $\sqrt{2 \times Process\_size \times Page\_table\_entry\_size}$

Important Formulas

For Process

- Virtual Address Space = Size of process
- Number of pages the process is divided = Process size / Page size
- If process size = 2^X bytes, then number of bits in virtual address space = X bits

For Page Table

- Size of page table = Number of entries in page table x Page table entry size
- Number of entries in pages table = Number of pages the process is divided
- Page table entry size = Number of bits in frame number + Number of bits used for optional fields if any

Practice Problem

A paging scheme uses a Translation Lookaside buffer (TLB). A TLB access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page fault ?

(a) 54

(b) 60

(c) 65

(d) 75

Practice Problem

A paging scheme uses a Translation Lookaside buffer (TLB). The effective memory access takes 160 ns and a main memory access takes 100 ns. What is the TLB access time (in ns) if the TLB hit ratio is 60% and there is no page fault ?

(a) 54

(b) 60

(c) 20

(d) 75

Practice Problem

Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in physical memory. It takes 10 milliseconds to search TLB and 80 milliseconds to access the physical memory. If the TLB hit ratio is 0.6, the effective memory access time(in milliseconds) is _____.

(a) 120

(b) 122

(c) 126

(d) 130

Practice Problems

1. Calculate the size of memory if its address consists of 22 bits and the memory is 2-byte addressable.
2. Calculate the number of bits required in the address for memory having size of 16 GB. Assume the memory is 4-byte addressable.
3. Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is _____.
4. Consider a machine with 64 MB physical memory and a 32 bit virtual address space. If the page size is 4 KB, what is the approximate size of the page table ?

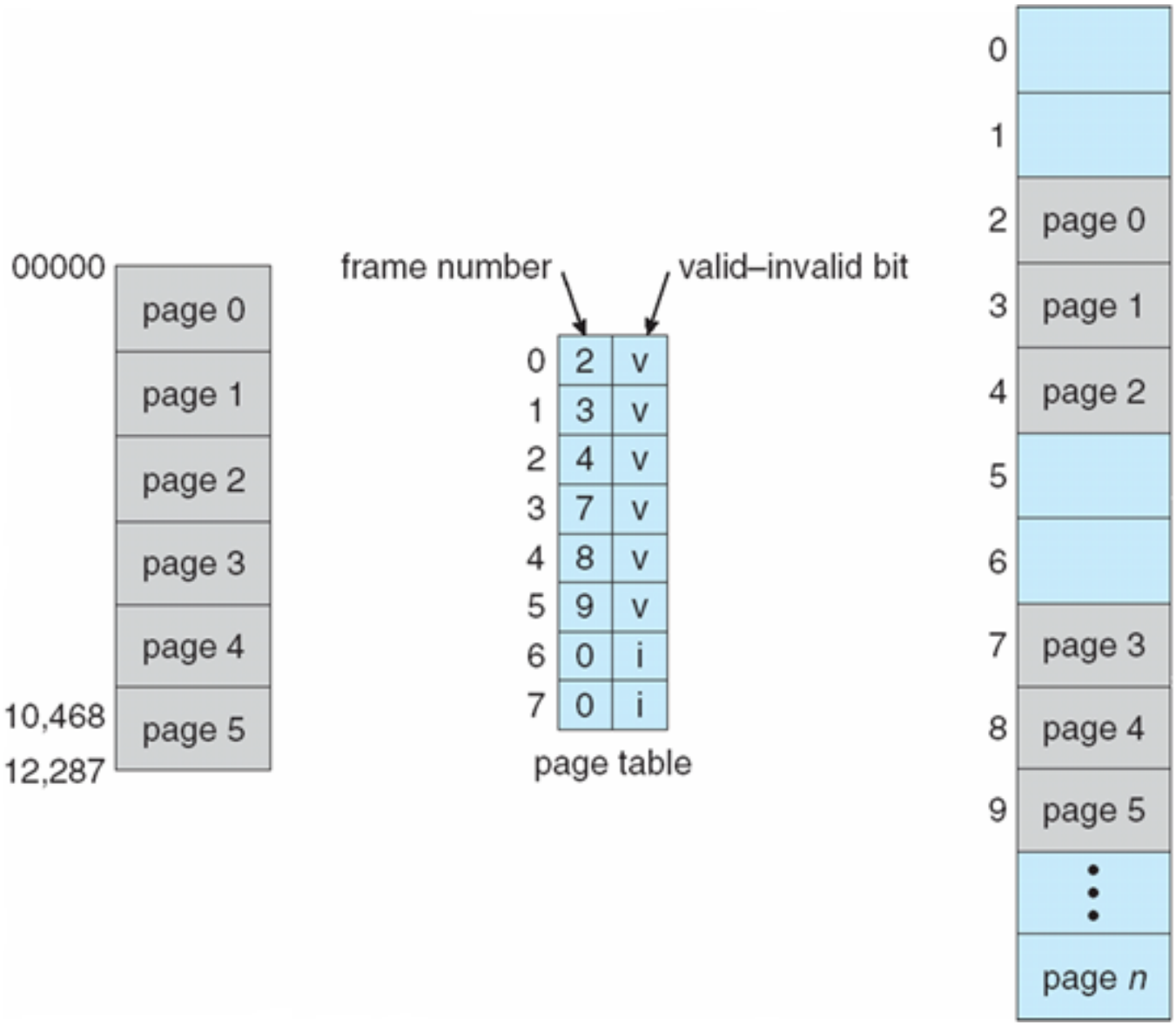
Practice Problems

1. In a virtual memory system, size of virtual address is 32-bit, size of physical address is 30-bit, page size is 4 Kbyte and size of each page table entry is 32-bit. The main memory is byte addressable. Which one of the following is the maximum number of bits that can be used for storing protection and other information in each page table entry ?
   a. 2
   b. 10
   c. 12
   d. 14

Practice Problems

1. Consider a single level paging scheme. The virtual address space is 4 MB and page size is 4 KB. What is the maximum page table entry size possible such that the entire page table fits well in one page?
2. Consider a single level paging scheme. The virtual address space is 4 GB and page size is 128 KB. What is the maximum page table entry size possible such that the entire page table fits well in one page?
3. Consider a single level paging scheme. The virtual address space is 128 TB and page size is 32 MB. What is the maximum page table entry size possible such that the entire page table fits well in one page?
4. Consider a single level paging scheme. The virtual address space is 256 MB and page table entry size is 4 bytes. What is the minimum page size possible such that the entire page table fits well in one page?
5. Consider a single level paging scheme. The virtual address space is 512 KB and page table entry size is 2 bytes. What is the minimum page size possible such that the entire page table fits well in one page?
6. Consider a single level paging scheme. The virtual address space is 16 GB and page table entry size is 4 bytes. What is the minimum page size possible such that the entire page table fits well in one page?

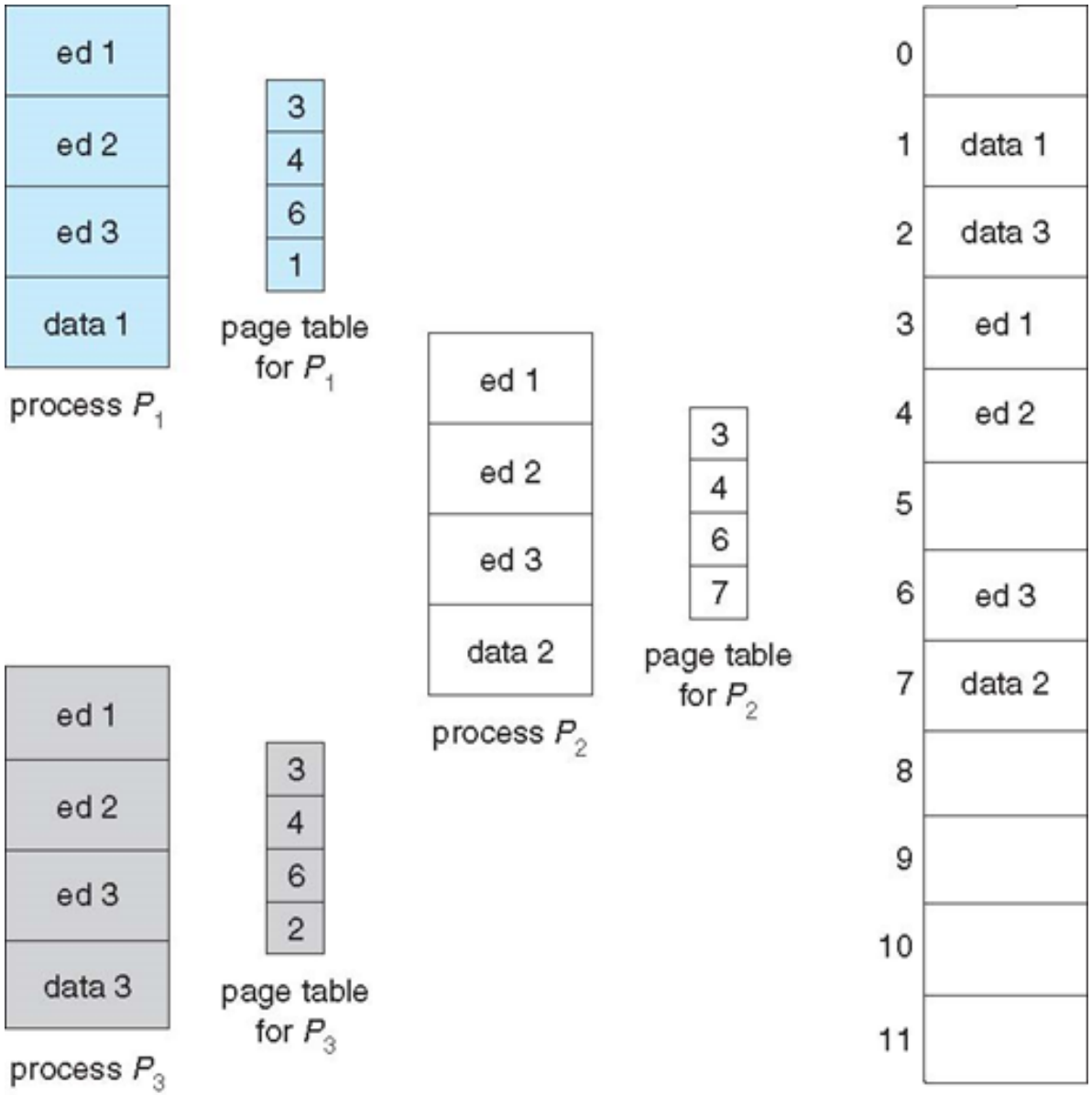
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Multilevel Paging

Multilevel paging is a paging scheme where there exists a hierarchy of page tables.

Need

The size of page table is greater than the frame size. As a result, the page table can not be stored in a single frame in main memory.

Working

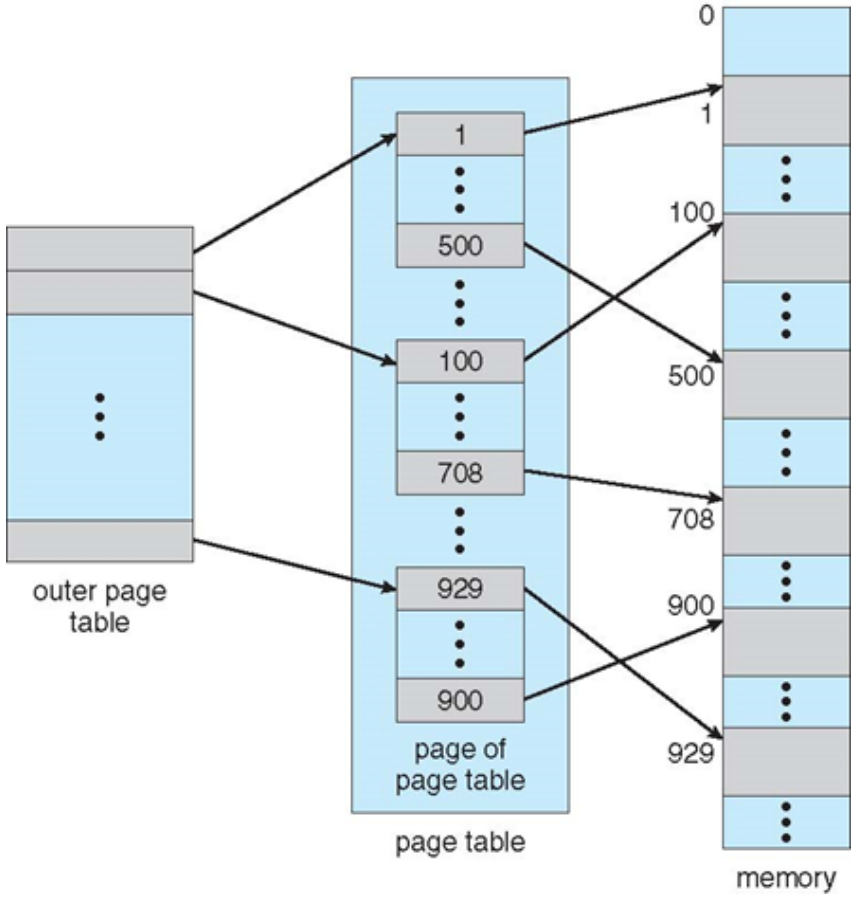
- The page table having size greater than the frame size is divided into several parts.
- The size of each part is same as frame size except possibly the last part.
- The pages of page table are then stored in different frames of the main memory.
- To keep track of the frames storing the pages of the divided page table, another page table is maintained.
- As a result, the hierarchy of page tables get generated.
- Multilevel paging is done till the level is reached where the entire page table can be stored in a single frame.

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32}/2^{12}$)
  - If each entry is 4 bytes $\rightarrow$ 4 MB of physical address space / memory for page table alone
- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

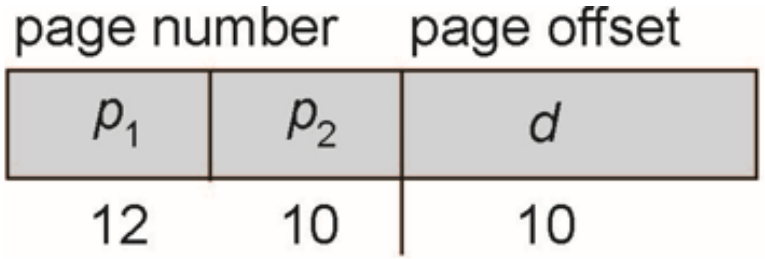
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



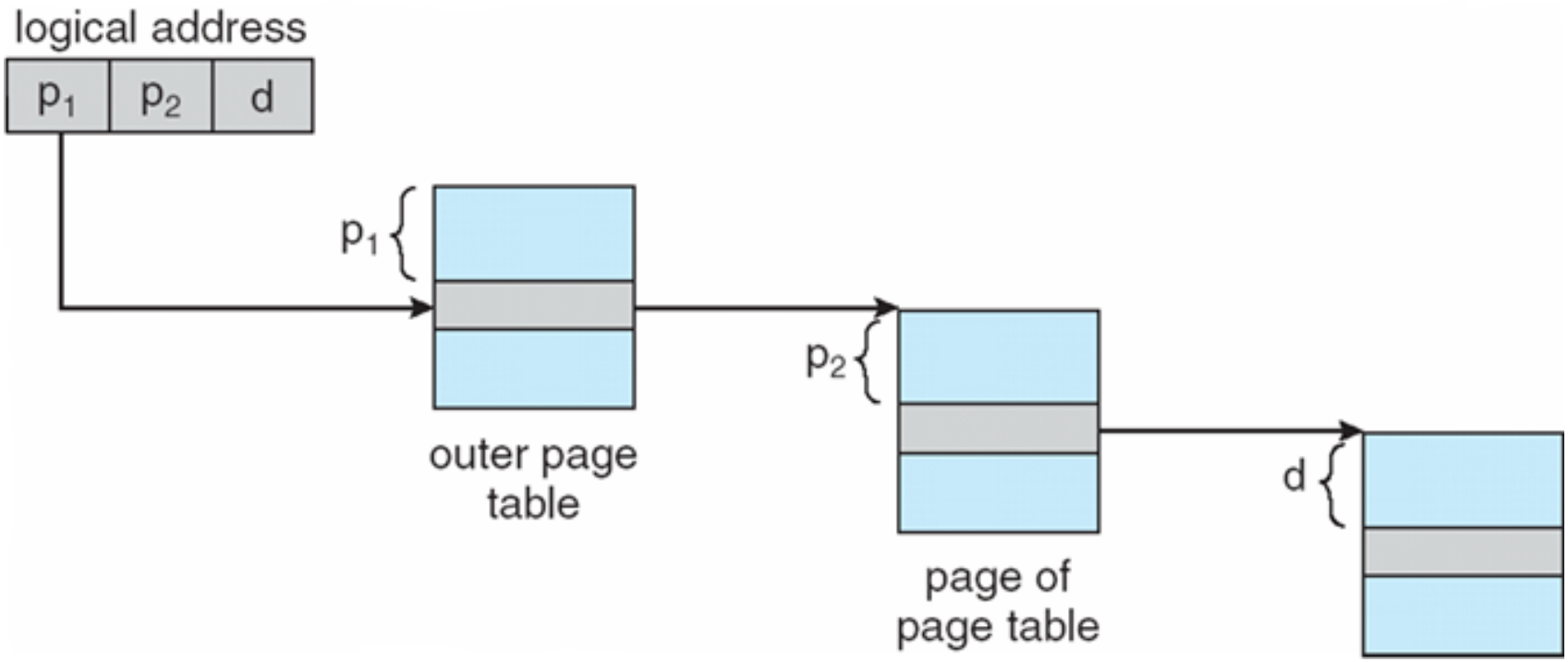
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



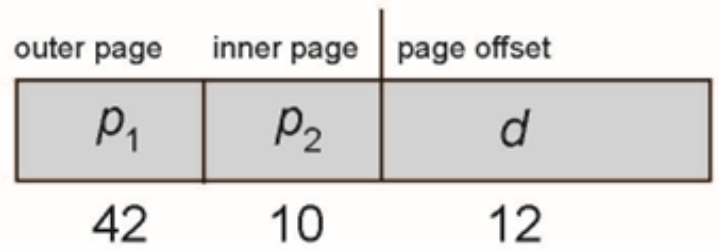
- where p_1 is an index into the outer page table, and p2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



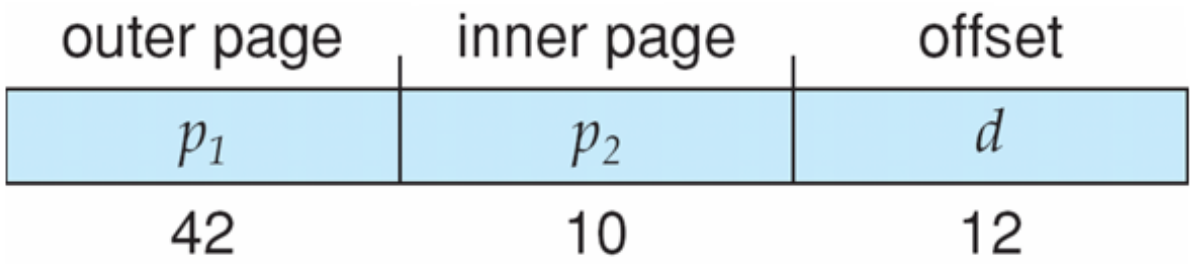
64-bit Logical Address Space

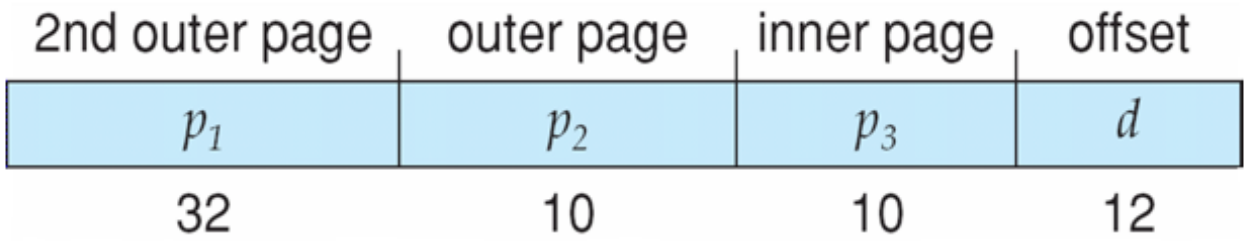
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 210 4-byte entries
  - Address would look like



  - Outer page table has 242 entries or 244 bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 234 bytes in size (and possibly 4 memory access to get to one physical memory location)

Three-level Paging Scheme





Practice Problem 1

Consider a system using multilevel paging scheme. The page size is 1 GB. The memory is byte addressable and virtual address is 72 bits long. The page table entry size is 4 bytes.

Find-

1. How many levels of page table will be required?
2. Give the divided physical address and virtual address.

Practice Problem 2

Consider a system using multilevel paging scheme. The page size is 1 MB. The memory is byte addressable and virtual address is 64 bits long. The page table entry size is 4 bytes.

Find-

1. How many levels of page table will be required?
2. Give the divided physical address and virtual address.

Practice Problem 3

Consider a system using multilevel paging scheme. The page size is 256 MB. The memory is byte addressable and virtual address is 72 bits long. The page table entry size is 4 bytes.

Find-

1. How many levels of page table will be required?
2. Give the divided physical address and virtual address.

Practice Problem 4

Consider a system using multilevel paging scheme. The page size is 16 MB. The memory is byte addressable and virtual address is 72 bits long. The page table entry size is 4 bytes.

Find-

1. How many levels of page table will be required?
2. Give the divided physical address and virtual address.

Practice Problem 5

Consider a single level paging scheme. The page size is 4 KB and page table entry size is 4 bytes. The size of page table is 4 KB. Give the division of virtual address space.

Practice Problem 6

Consider a two level paging scheme. The page size is 4 KB and page table entry size is 4 bytes. The size of outer page table is 4 KB. Give the division of virtual address space.

Practice Problem 7

Consider a three level paging scheme. The page size is 4 KB and page table entry size is 4 bytes. The size of outermost page table is 4 KB. Give the division of virtual address space.

Practice Problem 8

Consider a single level paging scheme with a TLB. Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is _____ msec.

Practice Problem 9

Consider a two level paging scheme with a TLB. Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is _____ msec.

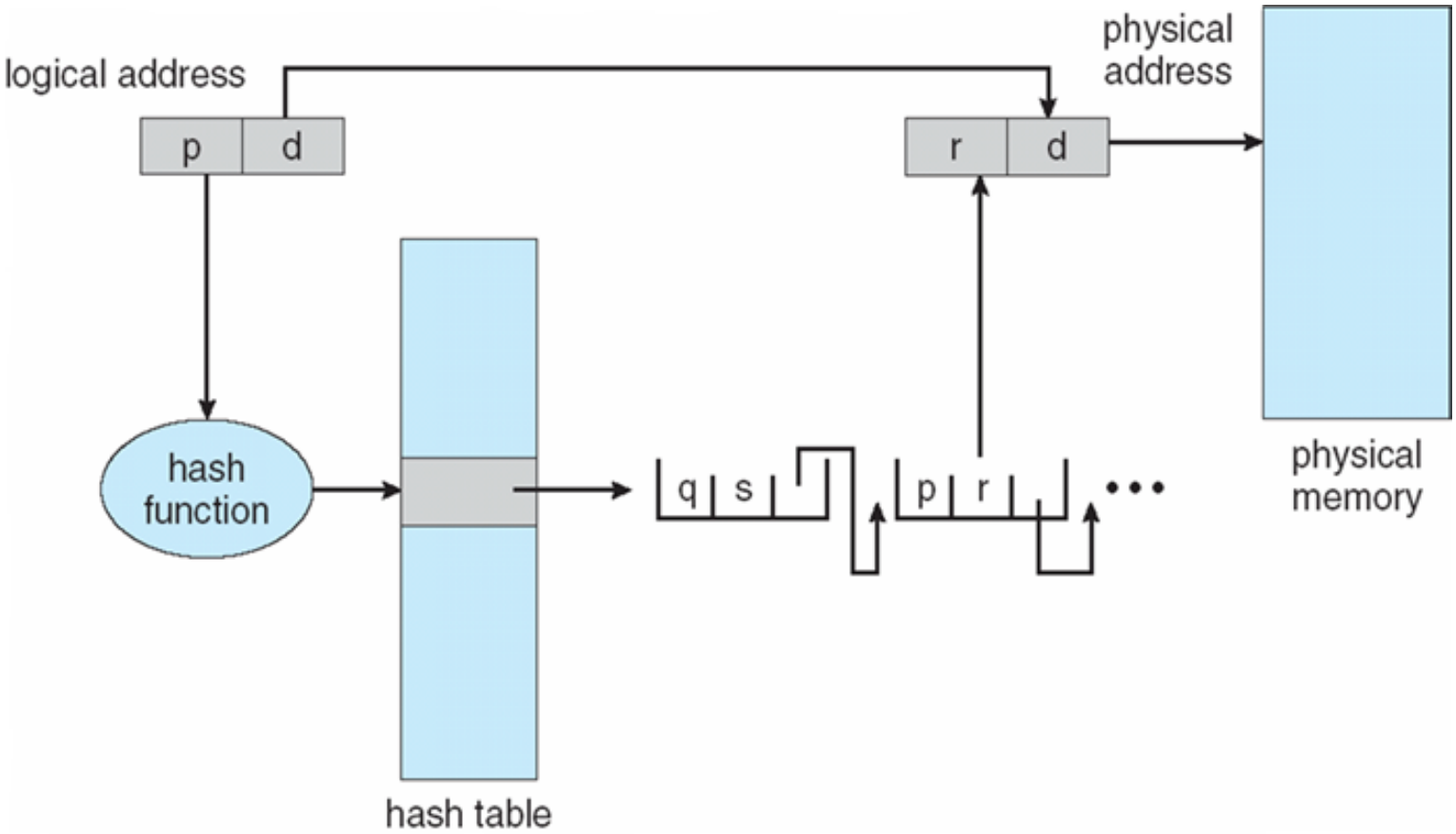
Practice Problem 10

Consider a three level paging scheme with a TLB. Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is _____ msec.

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  (1) the virtual page number
  (2) the value of the mapped page frame
  (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

Hashed Page Table



Hashed Page Tables

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the 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 increases 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
  - TLB can accelerate access
- But how to implement shared memory ?
  - One mapping of a virtual address to the shared physical address

Inverted Page Table

An inverted page table is one of the techniques to structure a page table, where the table is indexed by the actual frame number in the physical memory.

Need

Most of the Operating Systems use a separate page table for each process as in normal paging. In normal paging if there are 100 process then 100 will be the page tables in main memory. Sometimes when a process size is very large then its page table size also increases considerably. Through inverted page table, the overhead of storing an individual page table for each process is removed. A global page table is used which is utilized by all processes.

Inverted Page Table

Example

If A process size is 2 GB with Page size = 512 Bytes and page table entry size is 4 Bytes, then

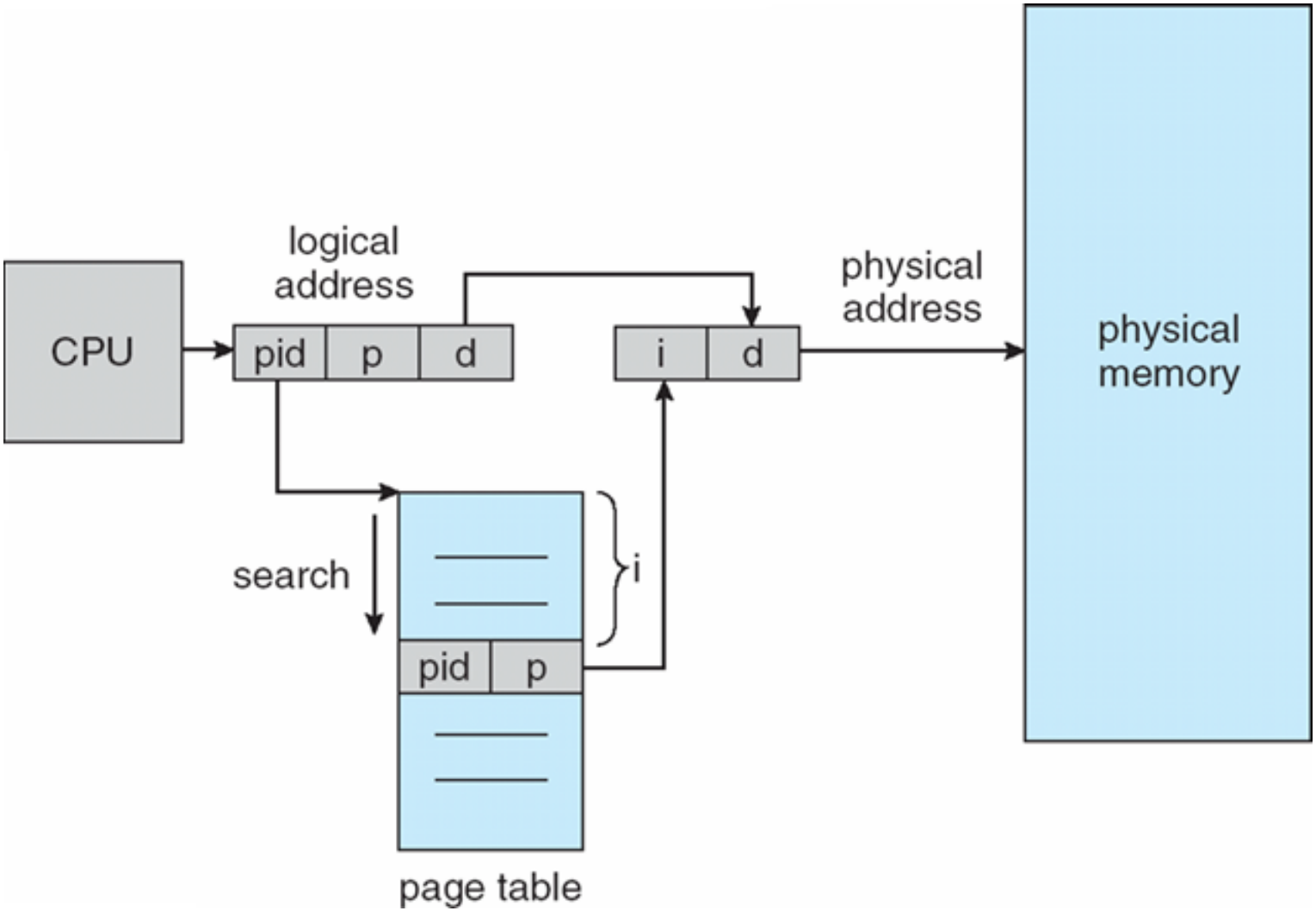
- Number of pages in the process = 2 GB / 512 B = 2^22
- Page Table Size = $2^{22} \times 4$ bytes = $2^{22} \times 2^2 = 2^{24}$

Working

In inverted page table indexing is done with frame numbers instead of the logical page number. Each entry in the page table contains the following fields.

- **Frame No:** It specifies the Frame where the page no is actually present in main memory
- **Page number:** It specifies the page number which is required.
- **Process id:** An inverted page table contains pages of all the processes in execution. So page No may be same of different process but Process Id of each process is unique. Through Process ID we get the desired page of that process.
- **Control bits:** These bits are used to store paging table entry information. These include the valid/invalid bit, protection, dirty bit, reference bits, and other information bit.
- **Chained pointer:** It is possible when two or more processes share some part of main memory. In simple words, when two or more logical pages need to map to same Page Table Entry then a chaining pointer is used.

Inverted Page Table Architecture



Practice Problem 1

A computer system, using inverted page table where logical address space is 16 MB and physical address space 8 GB, and page size are 4 KB. memory is byte addressable and page entry size is 8 bytes. What is the page table size ?

a. 16 KB
b. 128 KB
c. 16 MB
d. 128 MB

Segmentation

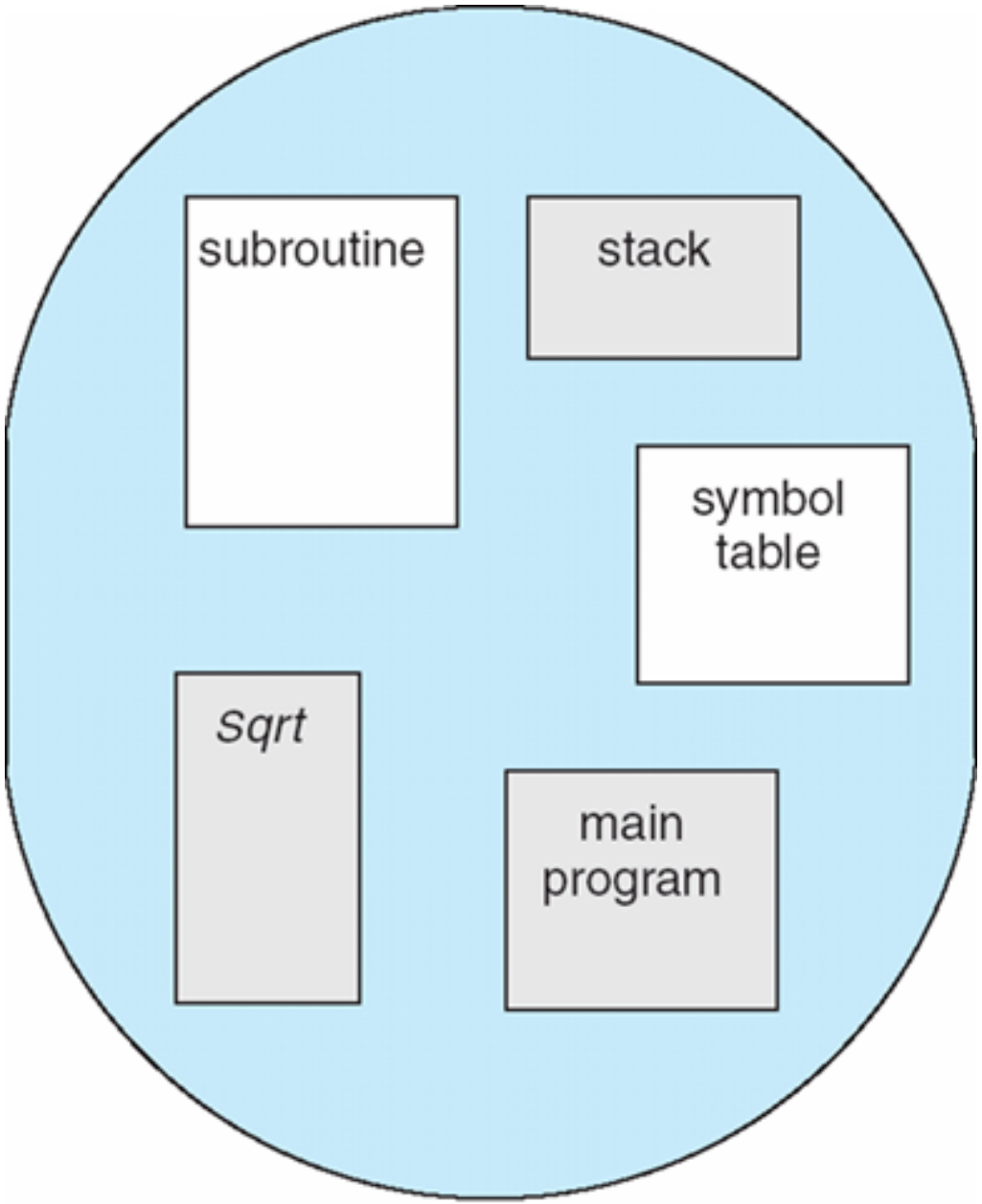
Definition

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. Segmentation gives users view of the process which paging does not give. Here the users view is mapped to physical memory.

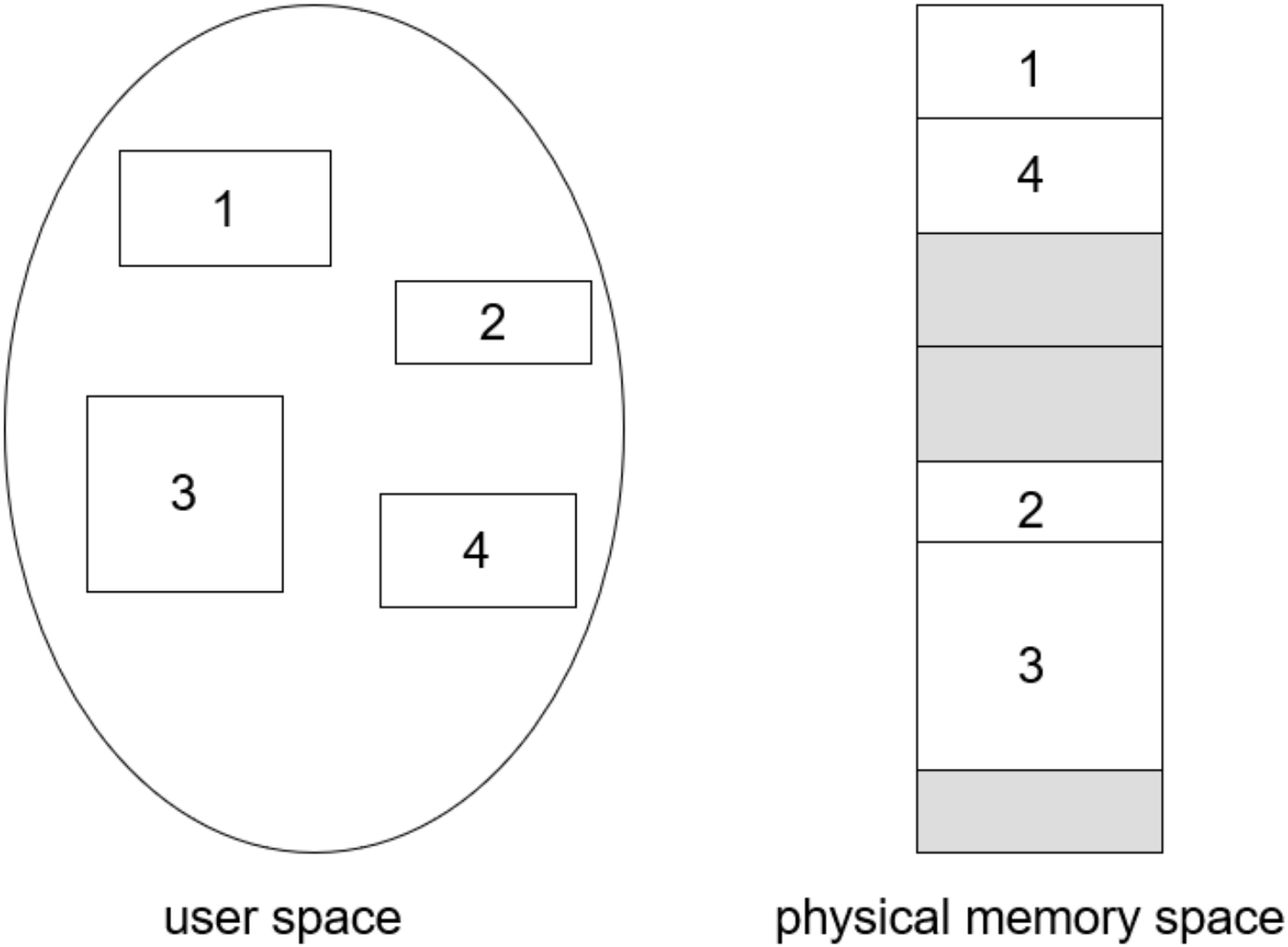
There are types of segment:

1. **Simple segmentation :-** Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.
2. **Virtual memory segmentation :-** Each process is divided into a number of segments, not all of which are resident at any one point in time.

User's View of a Program



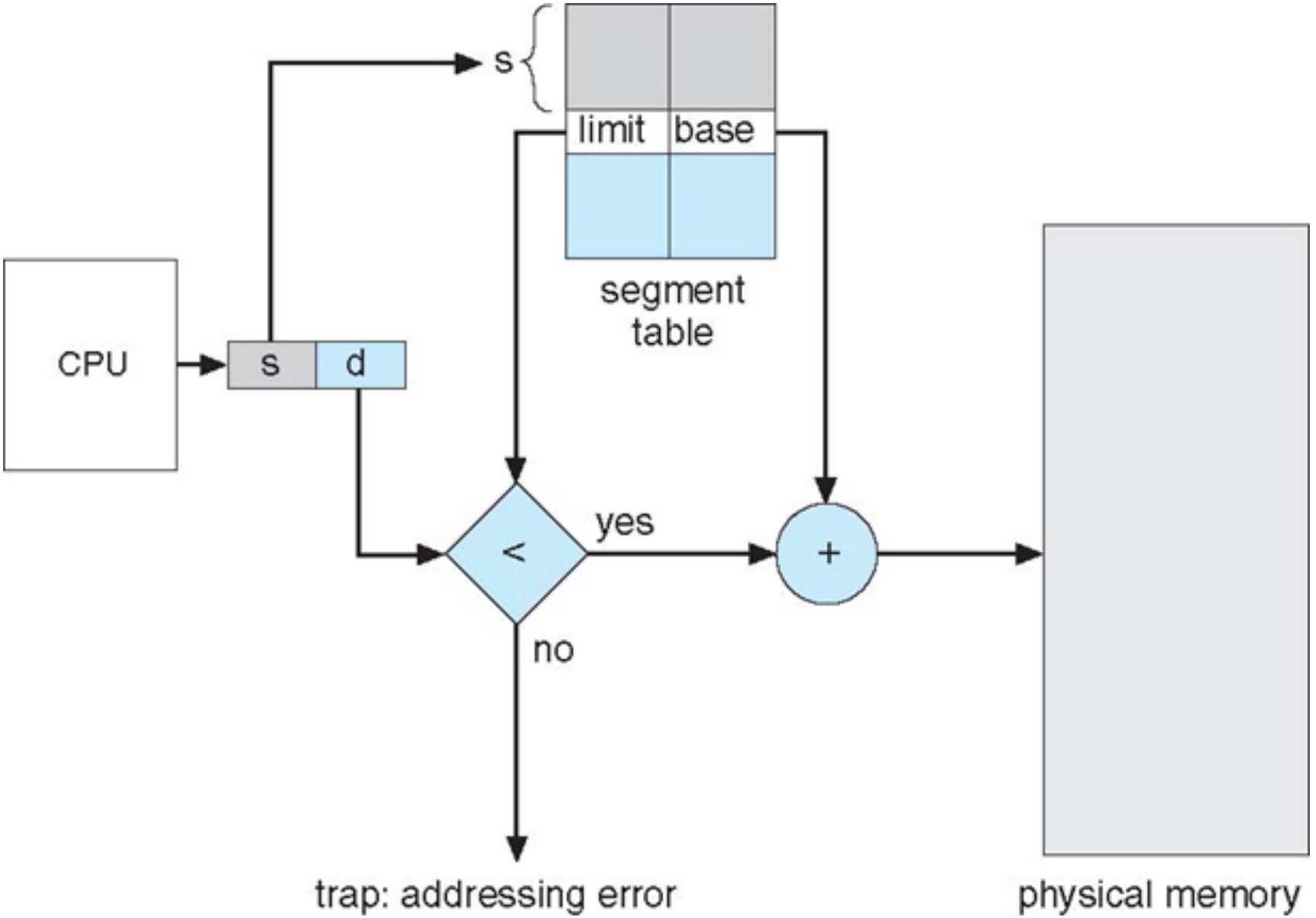
Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple: <segment-number, offset>
- **Segment table** - maps two-dimensional physical addresses; each table entry has:
  - **base** - contains the starting physical address where the segments reside in memory
  - **limit** - specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment tables location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
- Protection bit
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



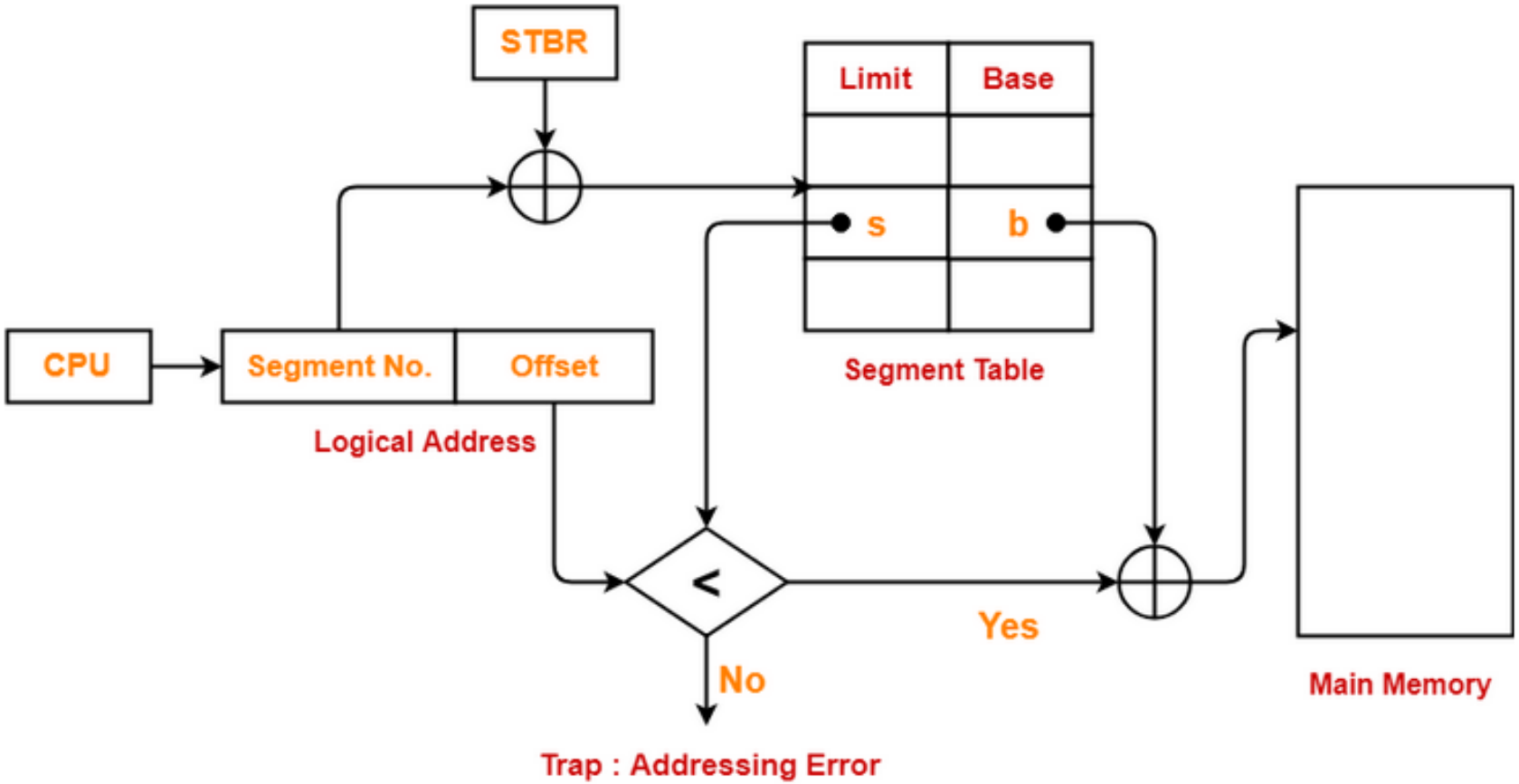
Segment Table

- Segment table is a table that stores the information about each segment of the process.
- It has two columns.
- First column stores the size or length of the segment.
- Second column stores the base address or starting address of the segment in the main memory.
- Segment table is stored as a separate segment in the main memory.
- Segment table base register (STBR) stores the base address of the segment table.
- Limit indicates the length or size of the segment.
- Base indicates the base address or starting address of the segment in the main memory.

	Limit	Base
Seg-0	1500	1500
Seg-1	500	6300
Seg-2	400	4300
Seg-3	1100	3200
Seg-4	1200	4700

Segment Table

Translating Logical Address into Physical Address



Translating Logical Address into Physical Address

Step 1 CPU generates a logical address consisting of two parts-

- Segment Number specifies the specific segment of the process from which CPU wants to read the data.
- Segment Offset specifies the specific word in the segment that CPU wants to read.

Step 2 For the generated segment number, corresponding entry is located in the segment table. Then, segment offset is compared with the limit (size) of the segment.

Now, two cases are possible-

C 1 **Segment Offset** $>=$ **Limit:** If segment offset is found to be greater than or equal to the limit, a trap is generated.

C 2 **Segment Offset** $<$ **Limit:**

- If segment offset is found to be smaller than the limit, then request is treated as a valid request.
- The segment offset must always lie in the range [0, limit-1],
- Then, segment offset is added with the base address of the segment.
- The result obtained after addition is the address of the memory location storing the required word.

Advantage & Disadvantage

Advantage

- It allows to divide the program into modules which provides better visualization.
- Segment table consumes less space as compared to Page Table in paging.
- It solves the problem of internal fragmentation.

Disadvantage

- There is an overhead of maintaining a segment table for each process.
- The time taken to fetch the instruction increases since now two memory accesses are required.
- Segments of unequal size are not suited for swapping.
- It suffers from external fragmentation as the free space gets broken down into smaller pieces with the processes being loaded and removed from the main memory.

Practice Problem

Consider the following segment table-

Segment No.	Base	Length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

Which of the following logical address will produce trap addressing error?

1. 0, 430
2. 1, 11
3. 2, 100
4. 3, 425
5. 4, 95

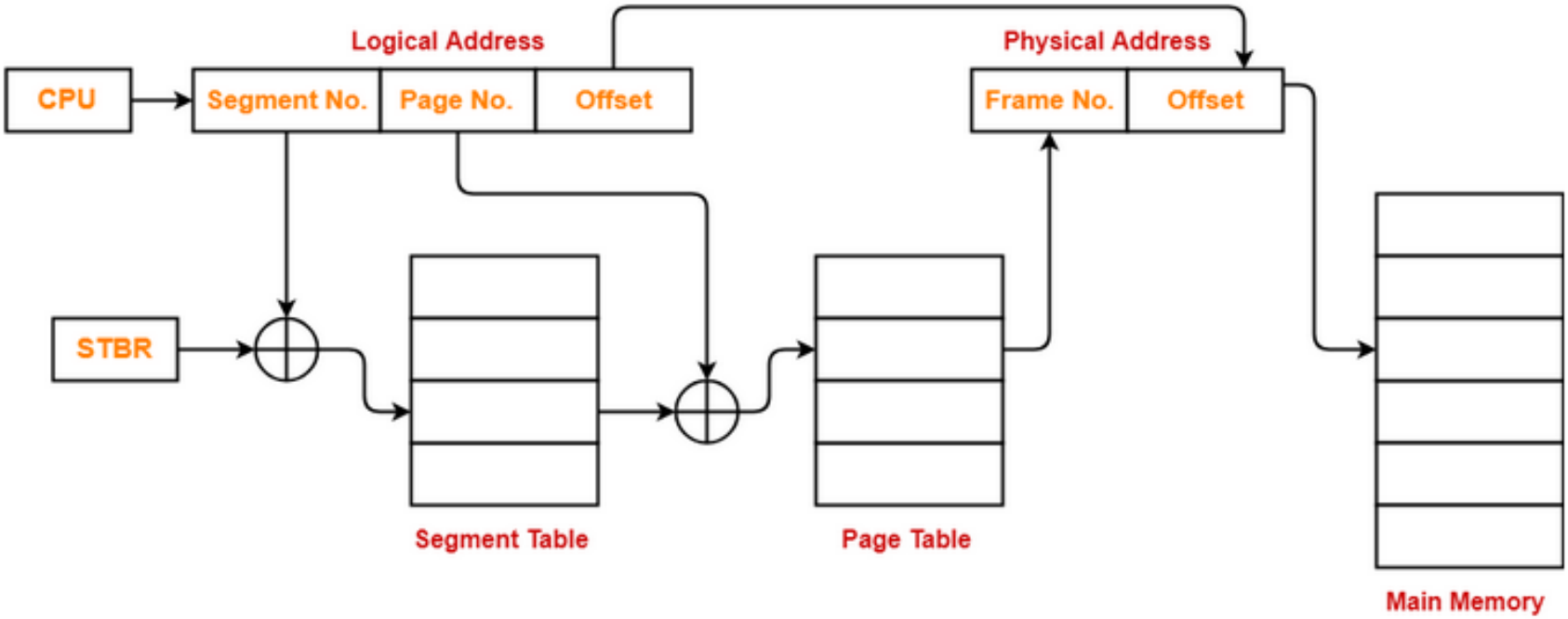
Segmented Paging

Segmented paging is a scheme that implements the combination of segmentation and paging.

Working

- Process is first divided into segments and then each segment is divided into pages.
- These pages are then stored in the frames of main memory.
- A page table exists for each segment that keeps track of the frames storing the pages of that segment.
- Each page table occupies one frame in the main memory.
- Number of entries in the page table of a segment = Number of pages that segment is divided.
- A segment table exists that keeps track of the frames storing the page tables of segments.
- Number of entries in the segment table of a process = Number of segments that process is divided.
- The base address of the segment table is stored in the segment table base register.

Translating Logical Address into Physical Address



Translating Logical Address into Physical Address

Step 1 CPU generates a logical address consisting of three parts-

- **Segment Number:** specifies the specific segment from which CPU wants to reads the data.
- **Page Number:** specifies the specific page of that segment from which CPU wants to read the data.
- **Page Offset:** specifies the specific word on that page that CPU wants to read.

Step 2 For the generated segment number, corresponding entry is located in the segment table. Segment table provides the frame number of the frame storing the page table of the referred segment. The frame containing the page table is located.

Step 3 For the generated page number, corresponding entry is located in the page table. Page table provides the frame number of the frame storing the required page of the referred segment. The frame containing the required page is located.

Step 4 The frame number combined with the page offset forms the required physical address. For the generated page offset, corresponding word is located in the page and read.

Practice Problem 1

A certain computer system has the segmented paging architecture for virtual memory. The memory is byte addressable. Both virtual and physical address spaces contain 216 bytes each. The virtual address space is divided into 8 non-overlapping equal size segments. The memory management unit (MMU) has a hardware segment table, each entry of which contains the physical address of the page table for the segment. Page tables are stored in the main memory and consists of 2 byte page table entries. What is the minimum page size in bytes so that the page table for a segment requires at most one page to store it ?

Practice Problem 2

A certain computer system has the segmented paging architecture for virtual memory. The memory is byte addressable. Both virtual and physical address spaces contain 216 bytes each. The virtual address space is divided into 8 non-overlapping equal size segments. The memory management unit (MMU) has a hardware segment table, each entry of which contains the physical address of the page table for the segment. Page tables are stored in the main memory and consists of 2 byte page table entries.. Assume that each page table entry contains (besides other information) 1 valid bit, 3 bits for page protection and 1 dirty bit. How many bits are available in page table entry for storing the aging information for the page? Assume that page size is 512 bytes.

Virtual Memory : Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running $\rightarrow$ more programs run at the same time (Increased CPU utilization and throughput with no increase in response time or turnaround time)
  - Less I/O needed to load or swap programs into memory -¿ each user program runs faster

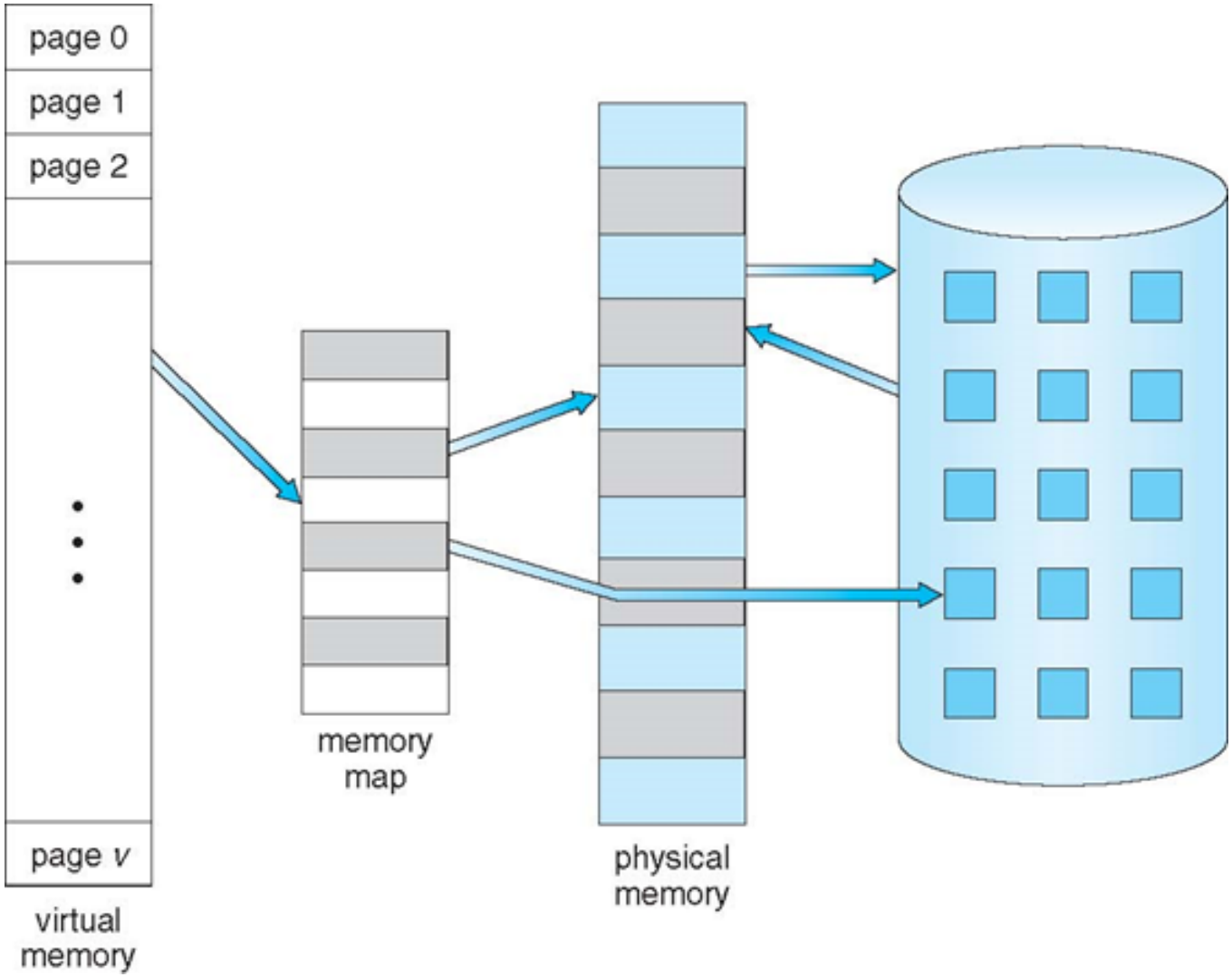
Virtual Memory : Background

- **Virtual memory** - separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- **Virtual address space** - logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames. MMU must map logical to physical

Virtual memory can be implemented via

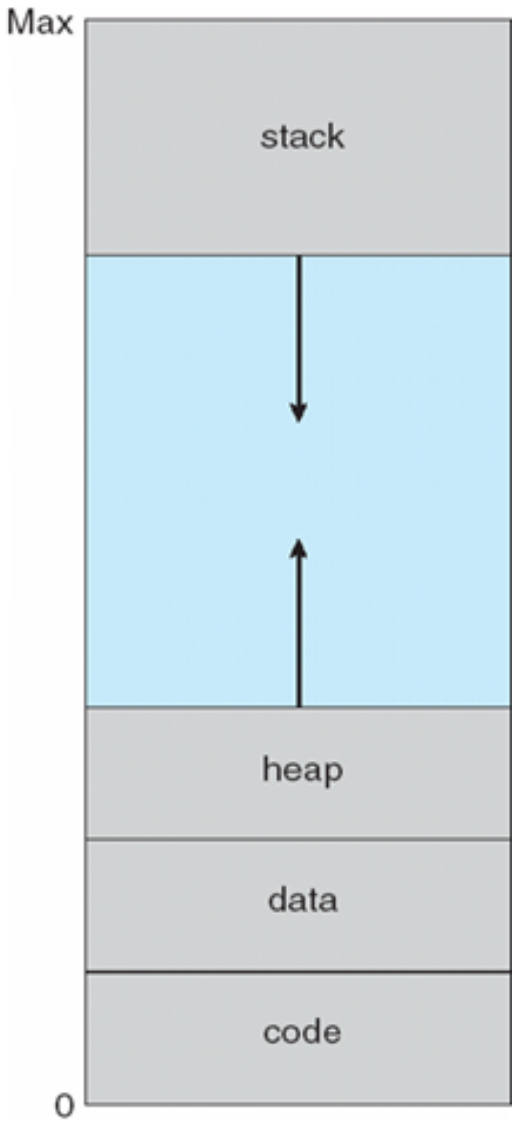
- Demand paging
- Demand segmentation

Virtual Memory That is Larger Than Physical Memory

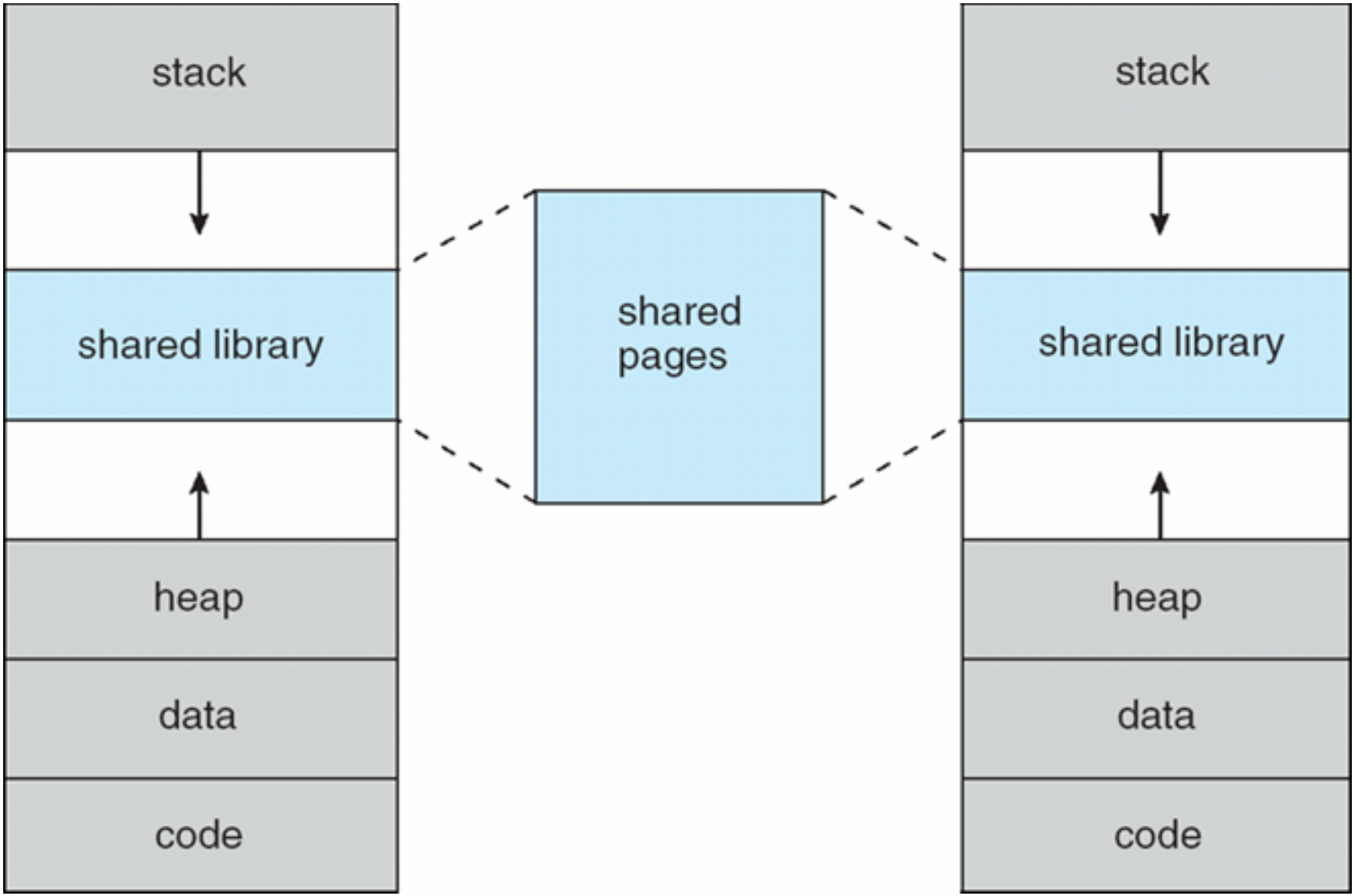


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow down while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole (No physical memory needed until heap or stack grows to a given new page)
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during *fork()*, speeding process creation

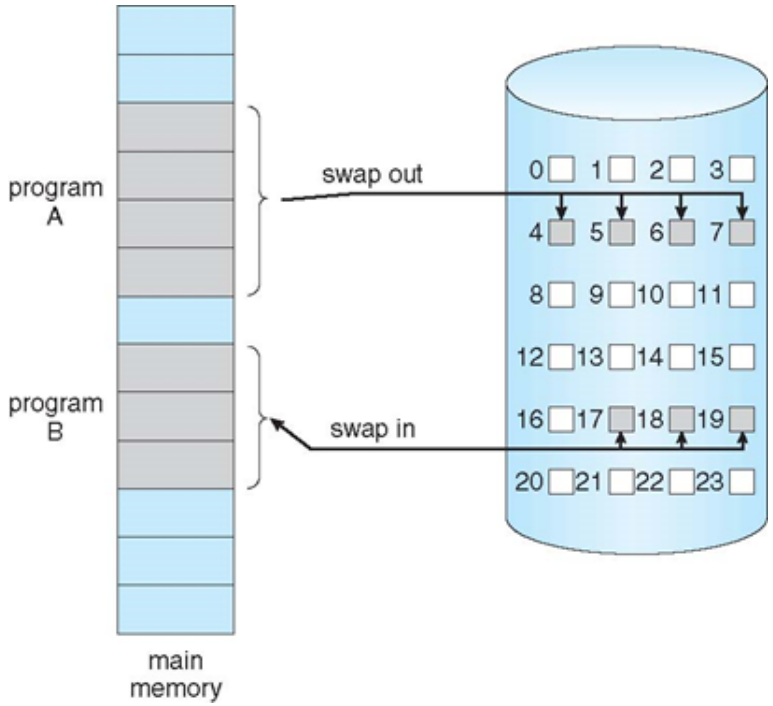


Shared Library Using Virtual Memory



Virtual-address Space

- Could bring entire process into memory at load time Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed → reference to it
  - invalid reference → abort
  - not-in-memory → bring to memory
- **Lazy swapper** - never swaps a page into memory unless page will be needed
- Swapper that deals with pages is a **pager**



Basic Concepts

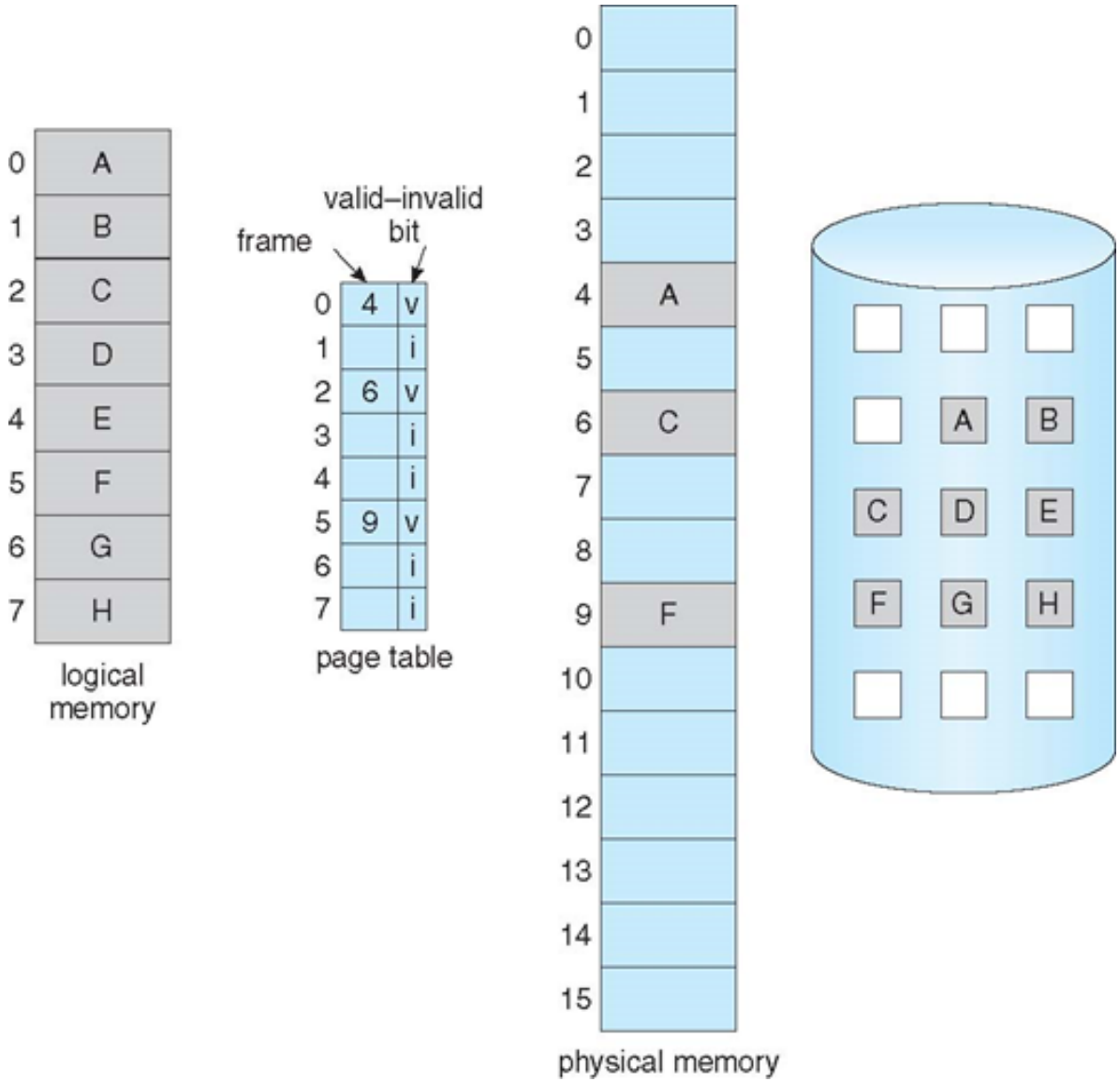
- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
  - Without changing program behavior
  - Without programmer needing to change code

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (v→in-memory - **memory resident**, i→not in-memory)
- Initially validinvalid bit is set to **i** on all entries
- Example of a page table snapshot:

- During MMU address translation, if validinvalid bit in page table entry is **i** → page fault

Page Table When Some Pages Are Not in Main Memo



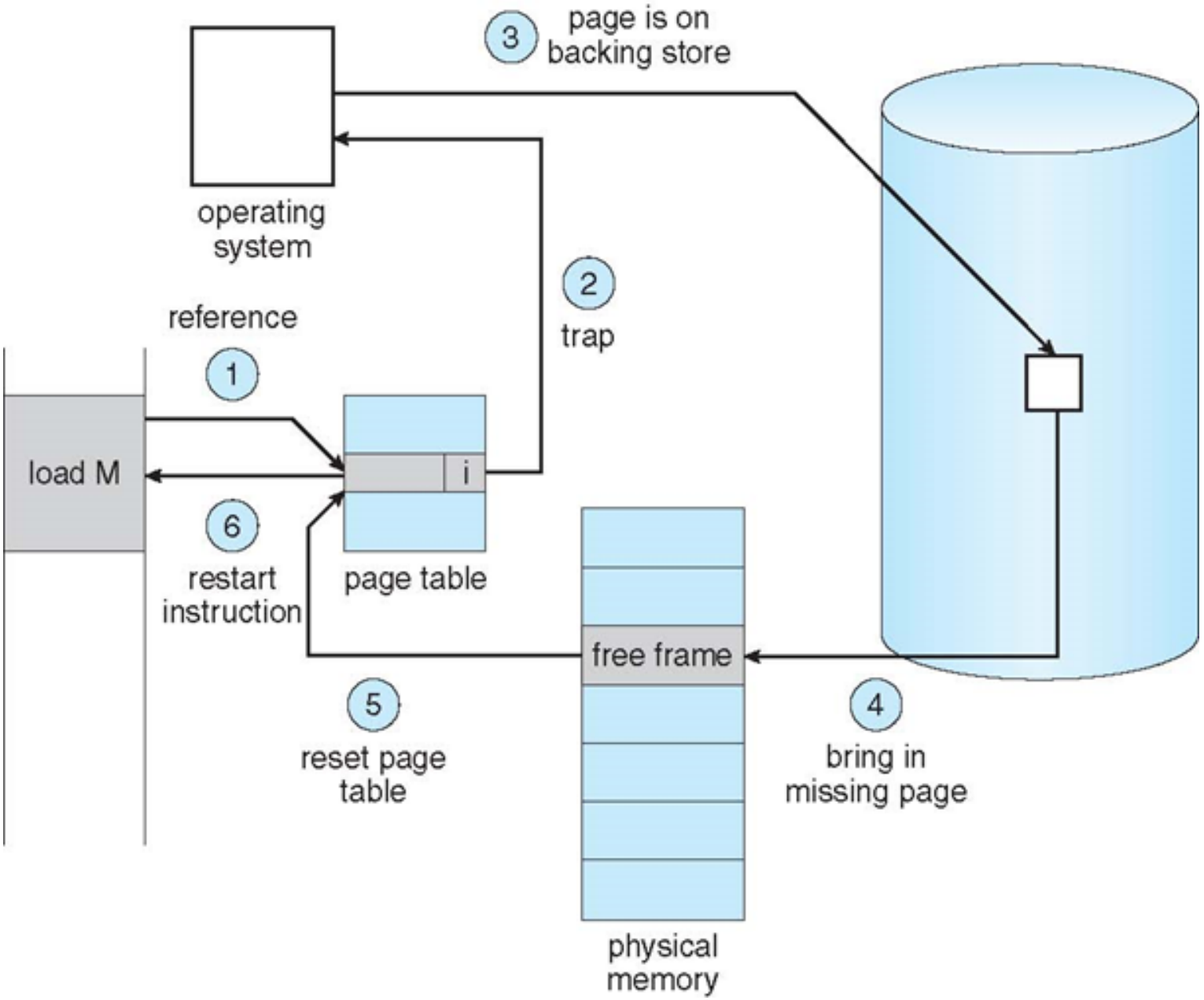
Page Fault

If there is a reference to a page, first reference to that page will trap to operating system:

page fault

1. Operating system looks at another table to decide:
   - Invalid reference $\rightarrow$ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory set validation bit=**v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$ every reference is a fault
- Effective Access Time (EAT) EAT = (1 - p) $\times$ memory access + p (page fault overhead + swap page out + swap page in)

Demand Paging Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800

If one access out of 1,000 causes a page fault, then

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

- 220 > 200 + 7,999,800 x p
  20 > 7,999,800 x p
- p < .0000025
- < one page fault in every 400,000 memory accesses

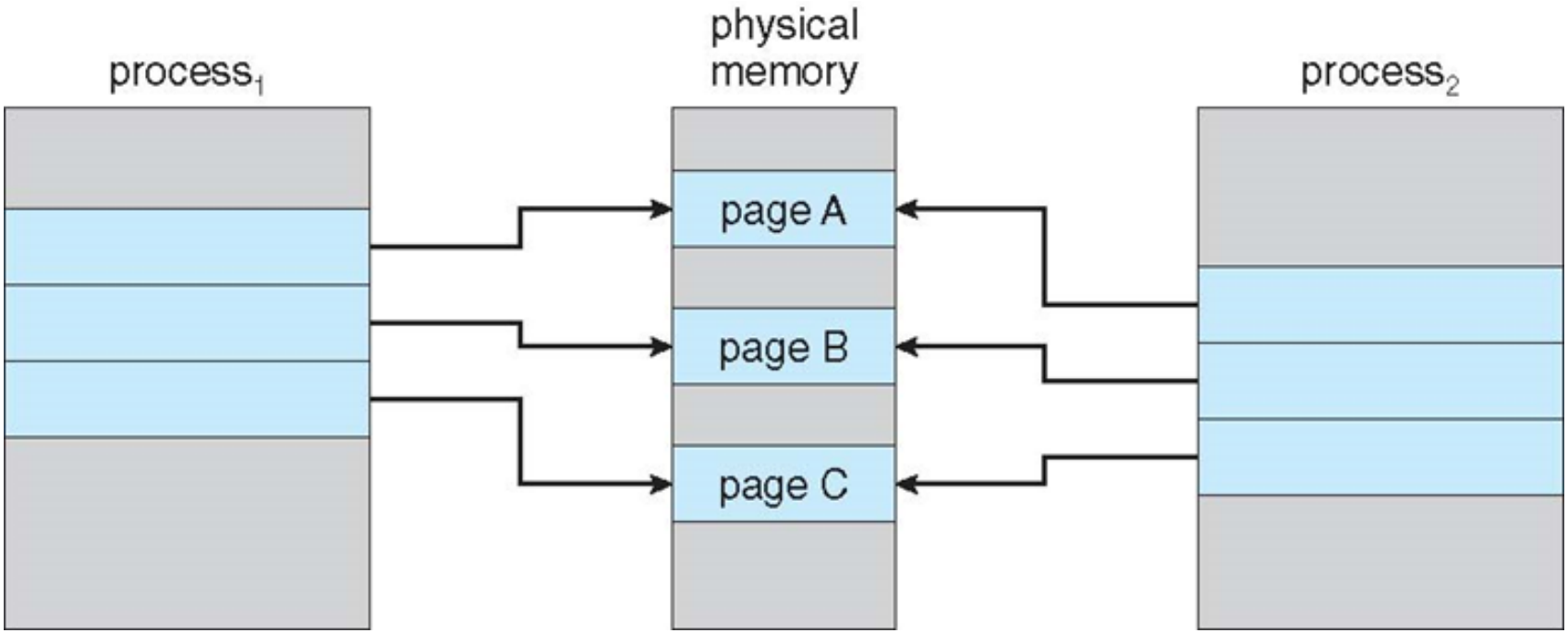
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

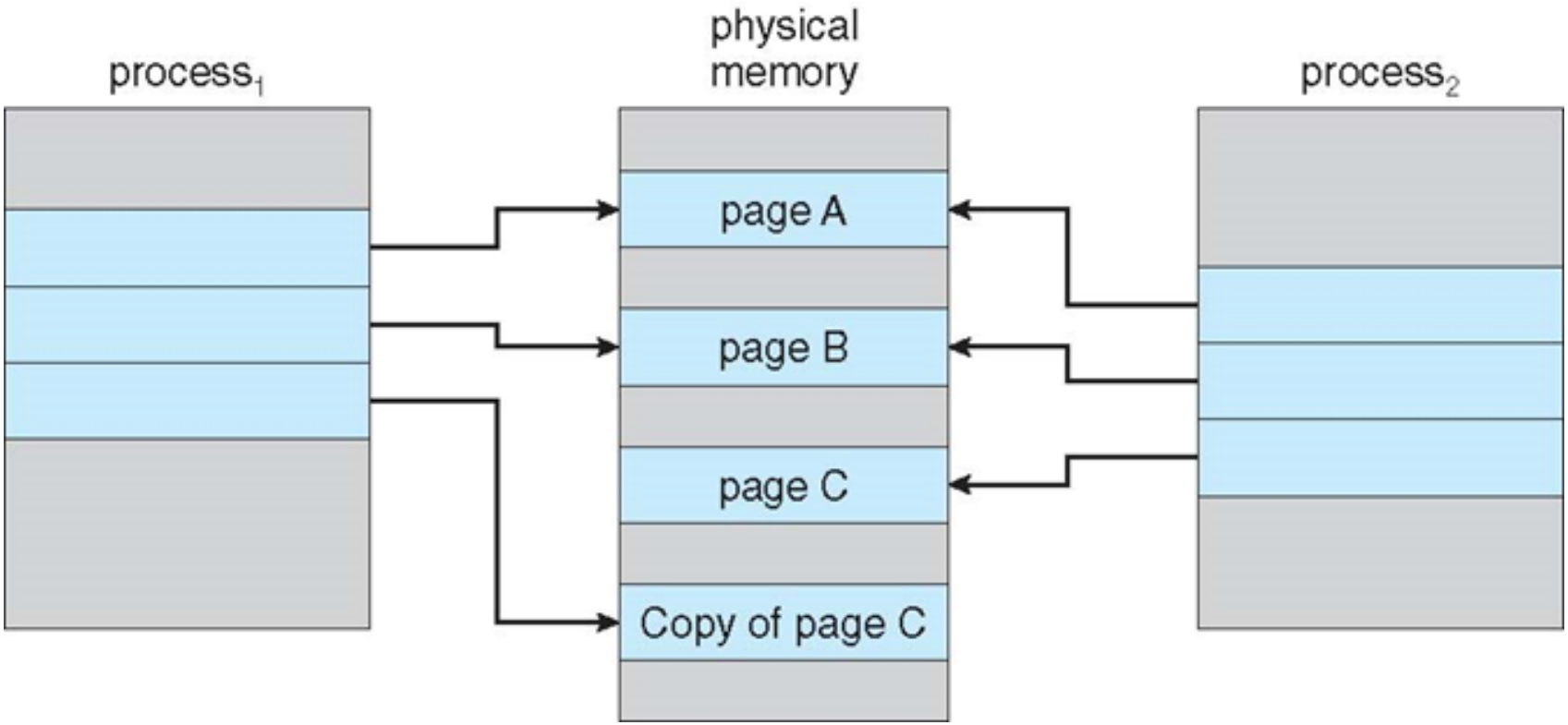
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
  - Don't want to have to free a frame as well as other processing on page fault

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



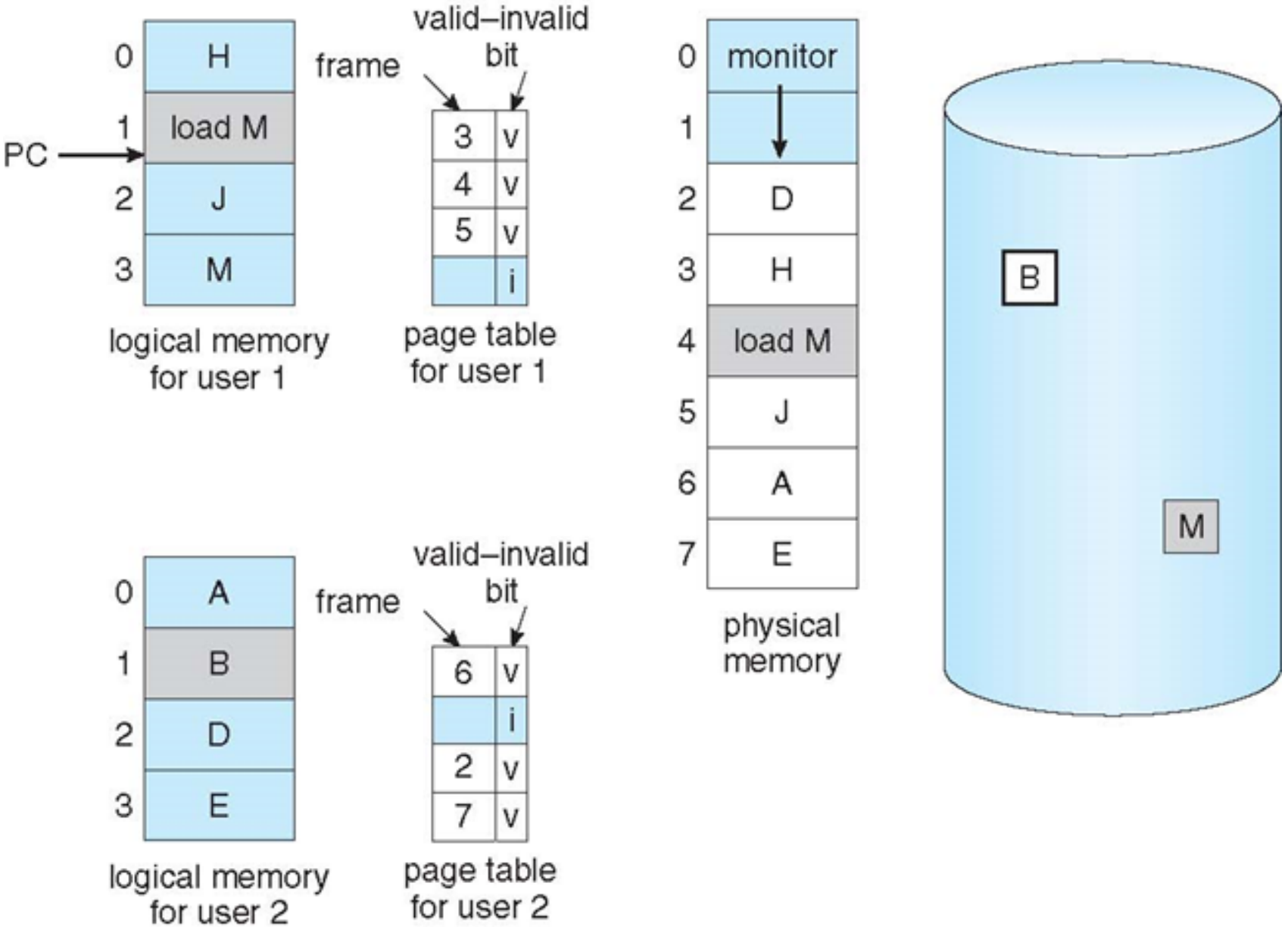
What Happens if There is no Free Frame ?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each ?
- Page replacement - find some page in memory, but not really in use, page it out
  - Algorithm - terminate ? swap out? replace the page ?
  - Performance - want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over–allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty)** bit to reduce overhead of page transfers only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory large virtual memory can be provided on a smaller physical memory

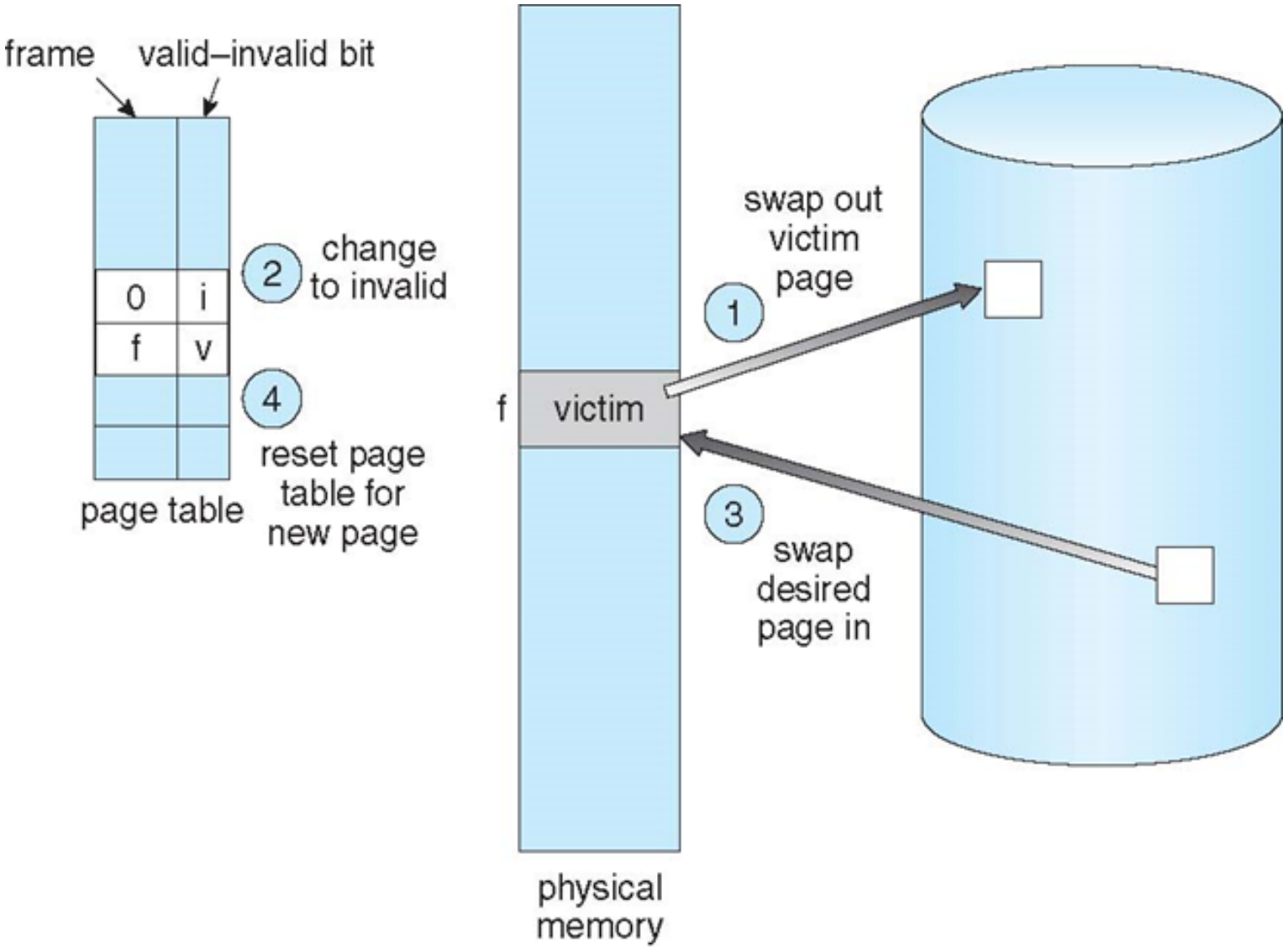
Need For Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Page Replacement



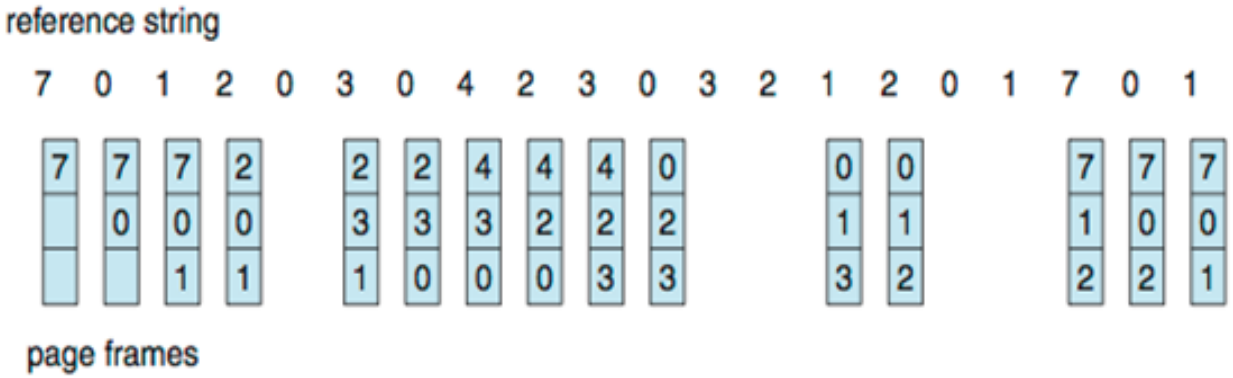
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- Page-replacement algorithm
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the reference string of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

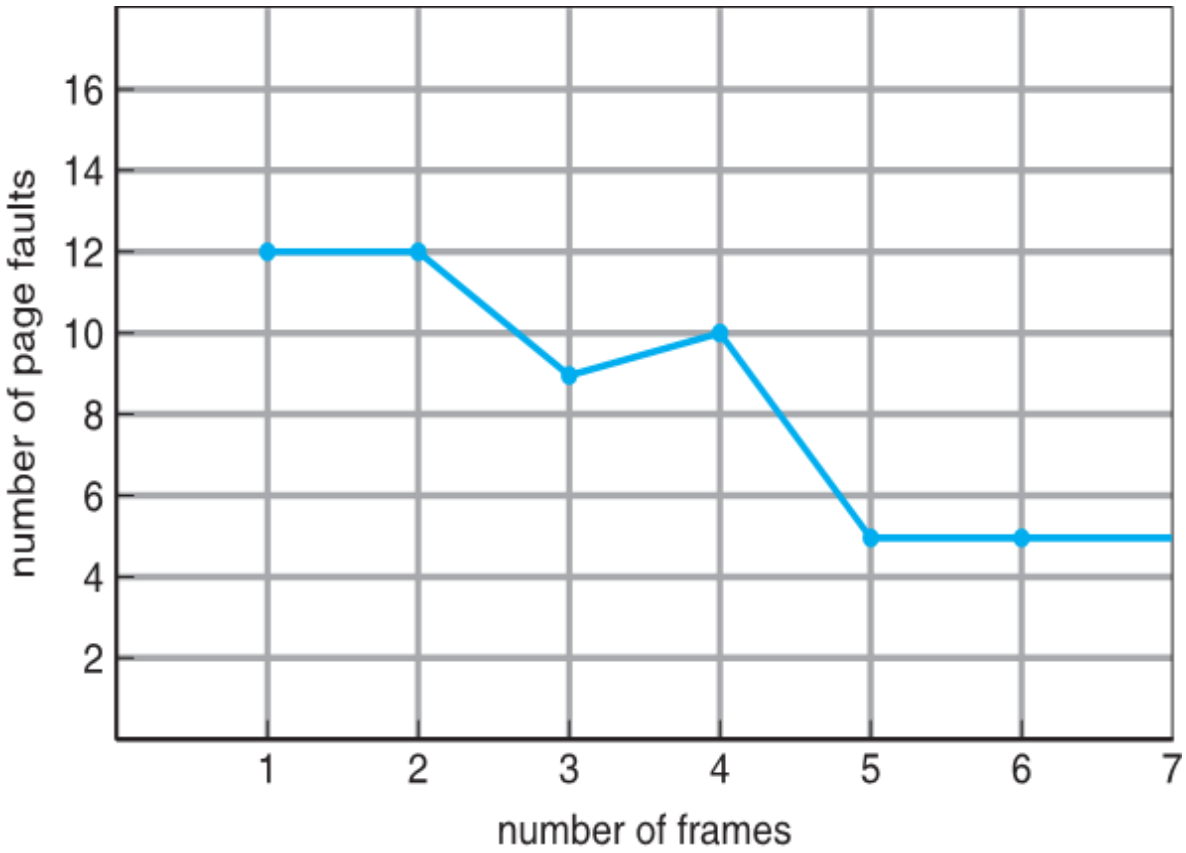
First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



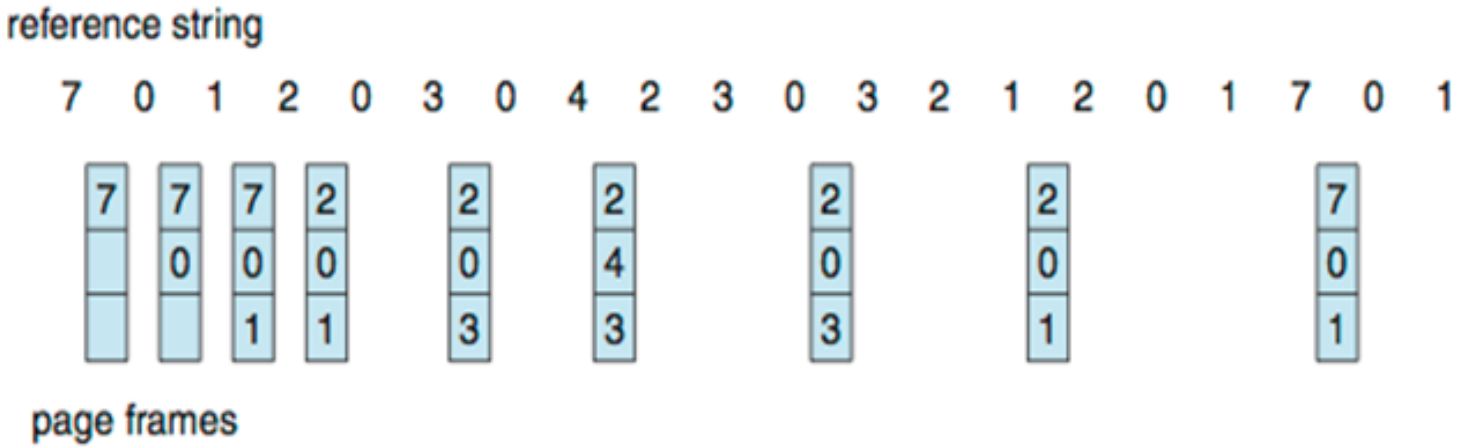
  15 page faults
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
- Adding more frames can cause more page faults ! (Belady's Anomaly)
- How to track ages of pages? (Just use a FIFO queue)

FIFO Illustrating Belady's Anomaly



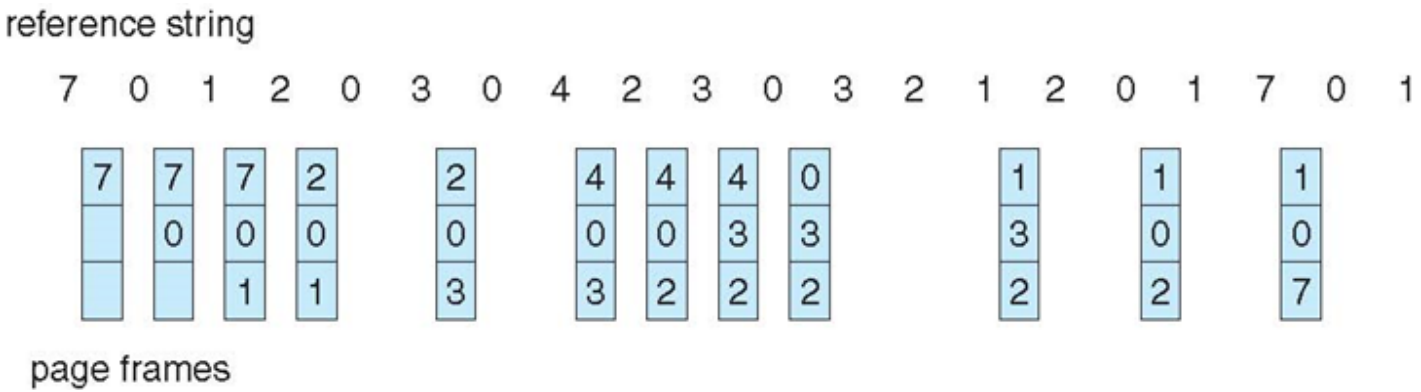
Optimal Algorithm

- Replace page that will not be used for longest period of time
- Replace page that has not been used in the most amount of time in future
- This algorithm replaces the page that will not be referred by the CPU in future for the longest time.
- It is practically impossible to implement this algorithm.
- This is because the pages that will not be used in future for the longest time can not be predicted. However, it is the best known algorithm and gives the least number of page faults.



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Associate time of last use with each page

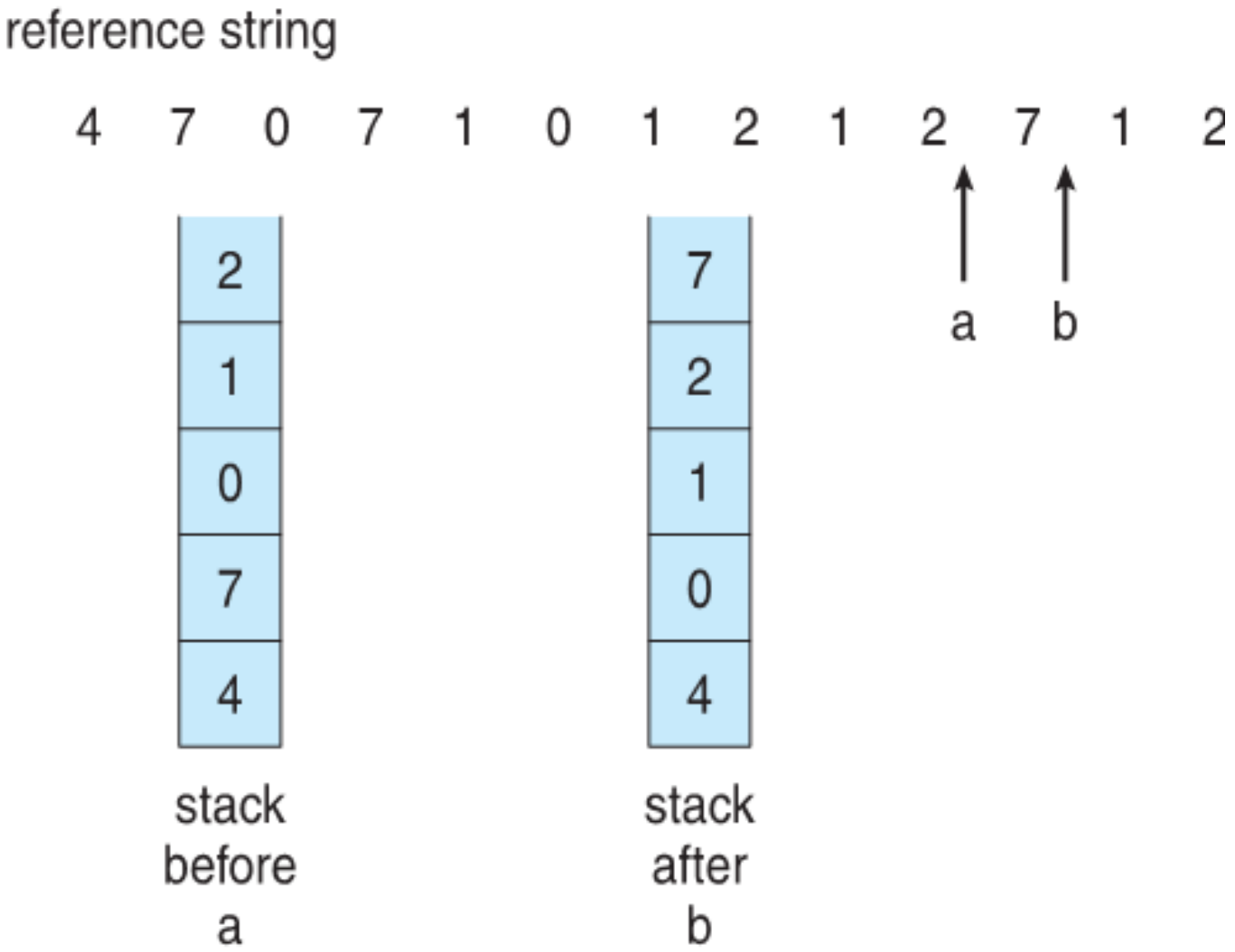


- 12 faults better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced: 1. move it to the top 2. requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that dont have Beladys Anomaly

Use Of A Stack to Record Most Recent Page Referenc



Beladys Anomaly

Beladys Anomaly is the phenomenon of increasing the number of page faults on increasing the number of frames in main memory.

Following page replacement algorithms suffer from Beladys Anomaly-

- FIFO Page Replacement Algorithm
- Random Page Replacement Algorithm
- Second Chance Algorithm

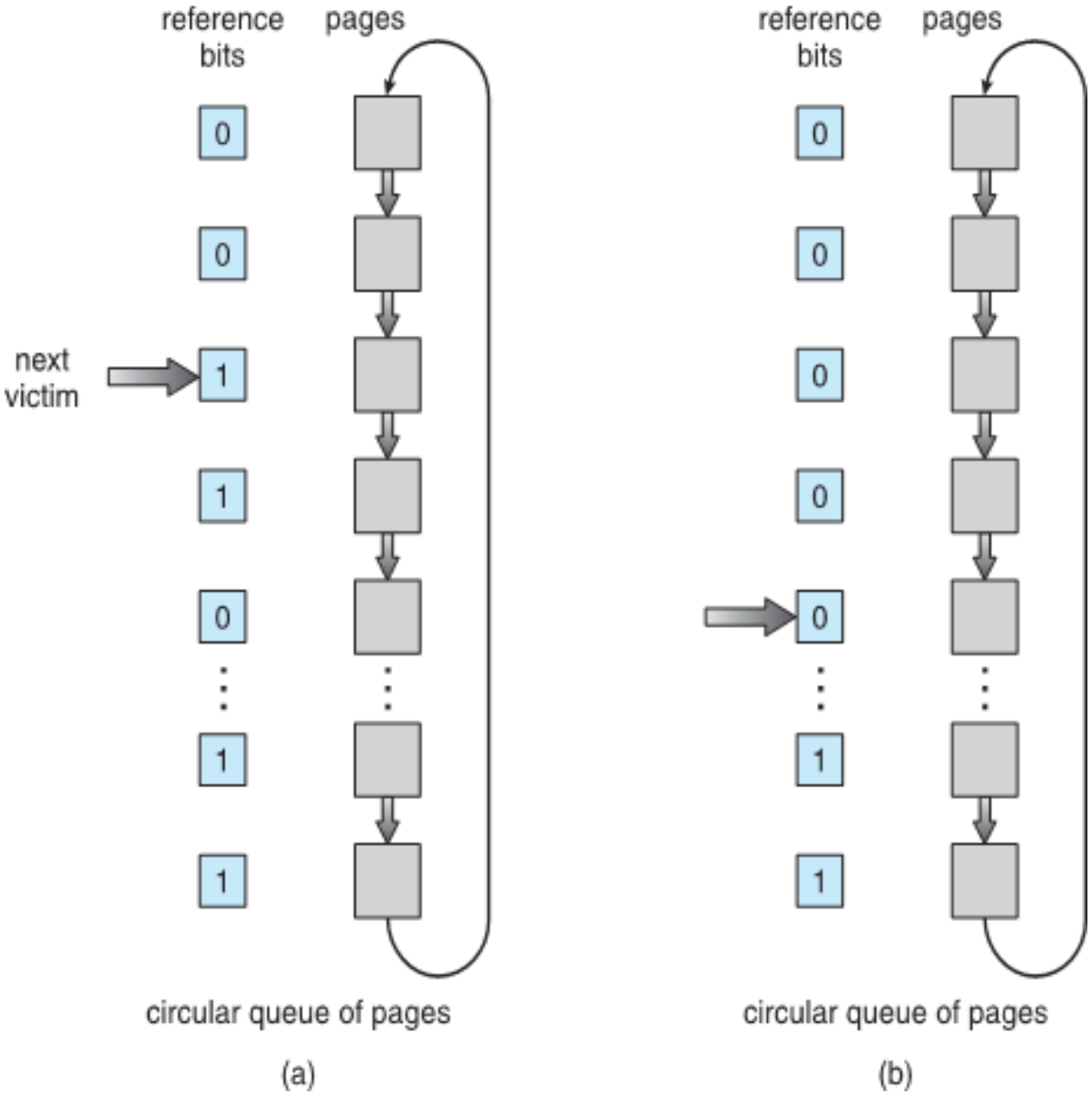
Reason Behind Beladys Anomaly

- An algorithm suffers from Beladys Anomaly if and only if it does not follow stack property.
- Algorithms that follow stack property are called as stack based algorithms.
- Stack based algorithms do not suffer from Beladys Anomaly.
- This is because these algorithms assign priority to a page for replacement that is independent of the number of frames in

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
  - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock replacement** If page to be replaced has
  - Reference bit = 0 $\rightarrow$ replace it
  - reference bit = 1 then:
  - set reference bit 0, leave page in memory
  - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
  1. (0, 0) neither recently used not modified best page to replace
  2. (0, 1) not recently used but modified not quite as good, must write out before replacement
  3. (1, 0) recently used but clean probably will be used again soon
  4. (1, 1) recently used and modified probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
- Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - No common
- **Lease Frequently Used (LFU) Algorithm:** replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

Allocation of Frames

- Each process needs minimum number of frames
- Example: IBM 370 - 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle from
  - 2 pages to handle to
- **Maximum** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

- Keep some as free frame buffer pool

Proportional allocation – Allocate according to the size of process

- Dynamic as degree of multiprogramming, process sizes change
  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

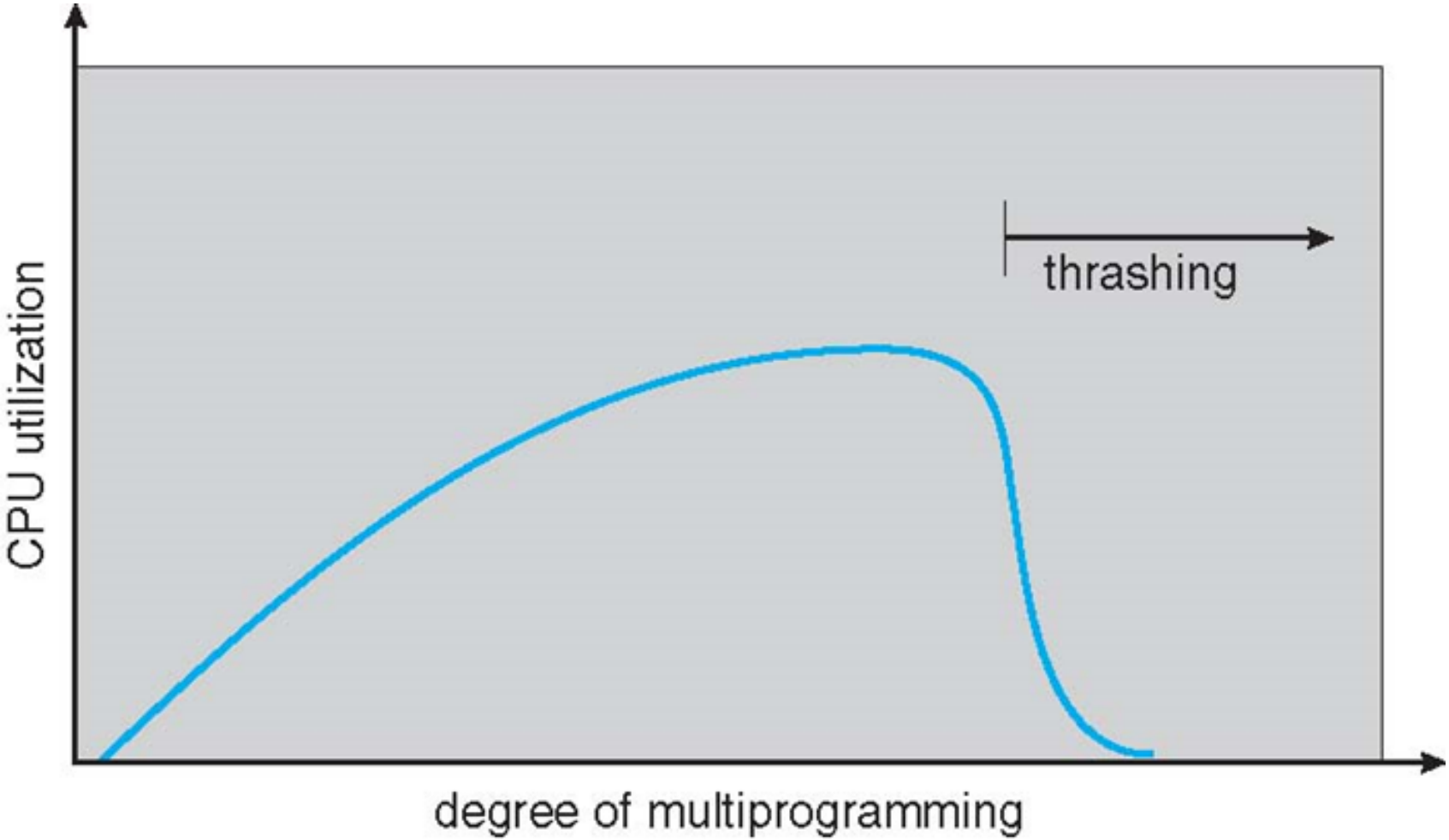
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- Thrashing $\approxeq$ a process is busy swapping pages in and out

Thrashing

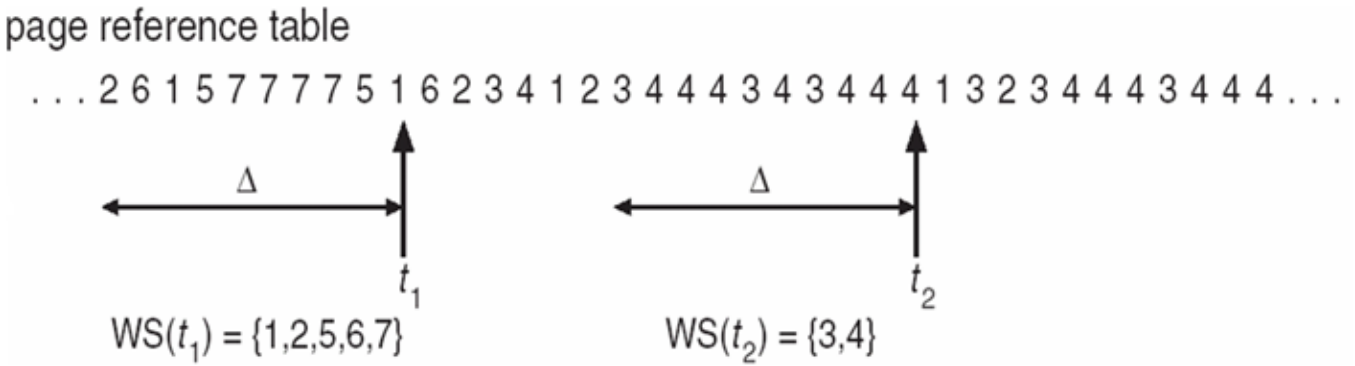


Demand Paging and Thrashing

- Why does demand paging works ?
  (**Locality Model**)
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur ?
  Σ size of locality $>$ total memory size
  - Limit effects by using local or priority page replacement

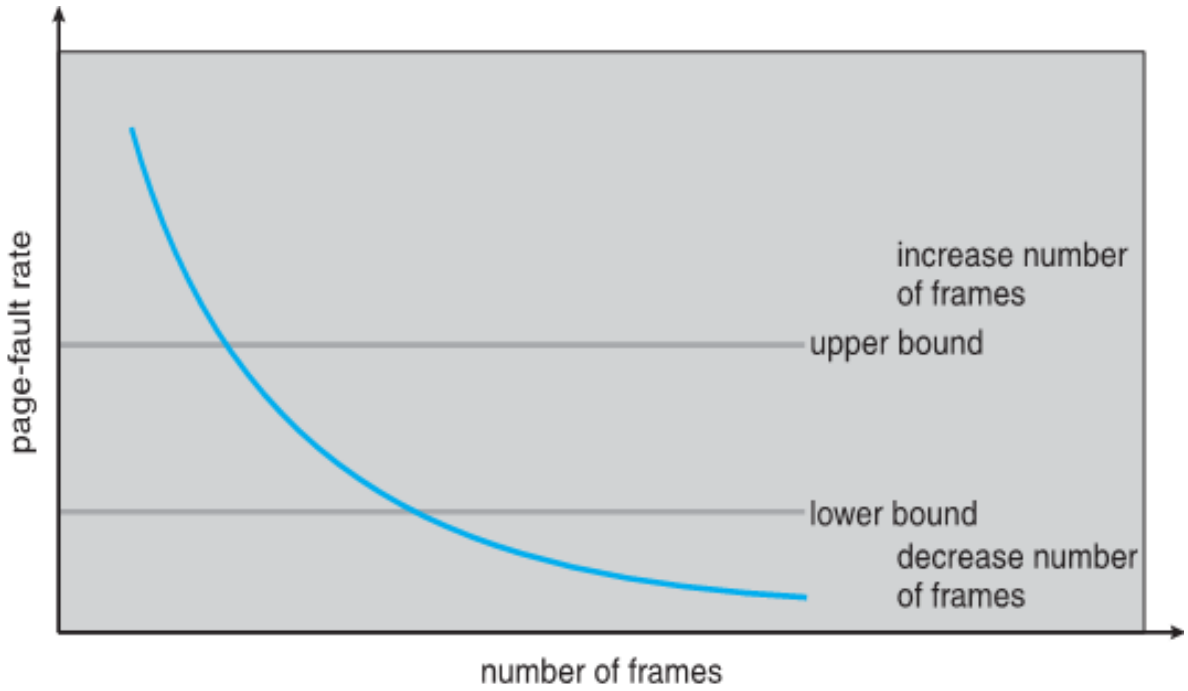
Working-Set Model

- $\Delta \approxeq$ working set window $\approxeq$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \approxeq$ total demand frames (Approximation of locality)
- if $D > m \rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes



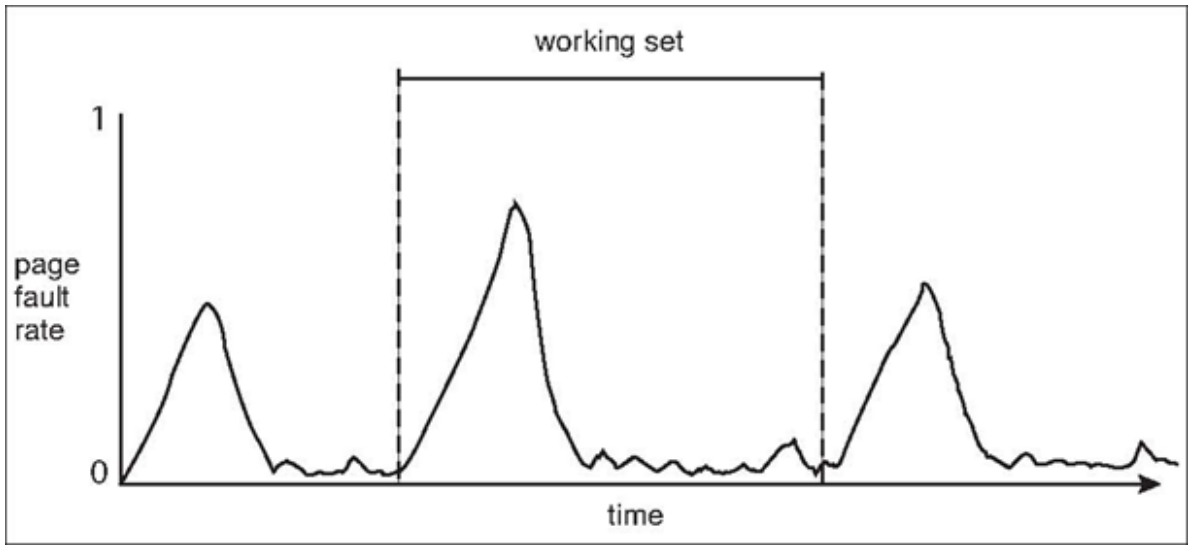
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency (PFF)** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Thank You!!!