

Main Memory Management

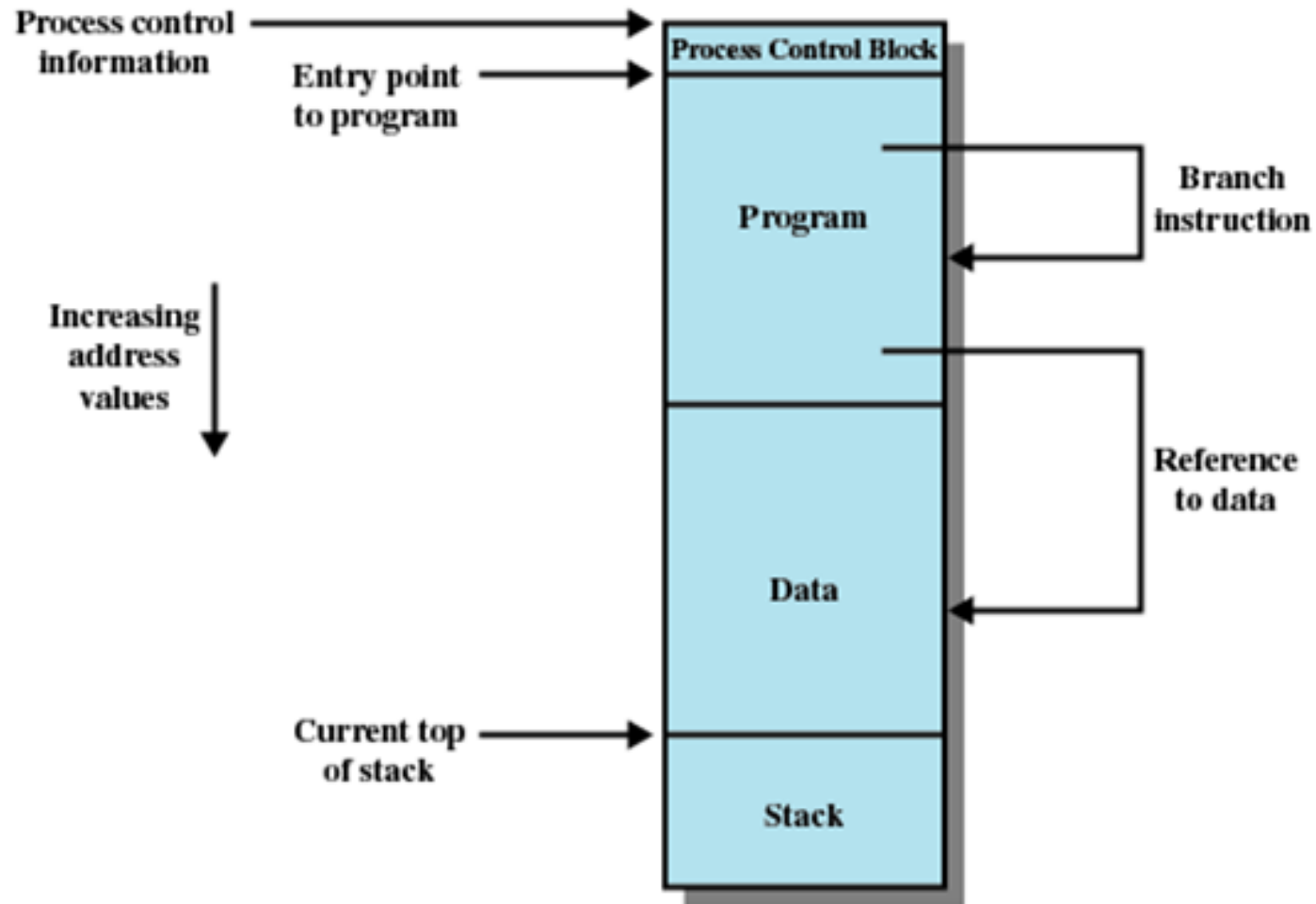
Reference – [Stallings] [Galvin]

Memory Management

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time
- Features
 - Relocation
 - Protection
 - Sharing
 - Logical Organization
 - Physical Organization

Memory Management Requirements

- Relocation
 - Programmer does not know where the program will be placed in memory when it is executed
 - While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
 - Memory references must be translated in the code to actual physical memory address



Addressing Requirements for a Process

Memory Management Requirements

- Protection
 - Processes should not be able to reference memory locations in another process without permission
 - Impossible to check absolute addresses at compile time
 - Must be checked at run time
 - Memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
 - Operating system cannot anticipate all of the memory references a program will make

Memory Management Requirements

- Sharing
 - Allow several processes to access the same portion of memory
 - Better to allow each process access to the same copy of the program rather than have their own separate copy

Memory Management Requirements

- Logical Organization
 - Programs are written in modules
 - Modules can be written and compiled independently
 - Different degrees of protection given to modules (read-only, execute-only)
 - Share modules among processes

Memory Management Requirements

- Physical Organization
 - Two levels of memory (Main and Secondary)
 - Memory available for a program plus its data may be insufficient.
 - Overlaying (Programmer dependent) allows various modules to be assigned the same region of memory
 - Programmer does not know how much space will be available

Fixed Partitioning

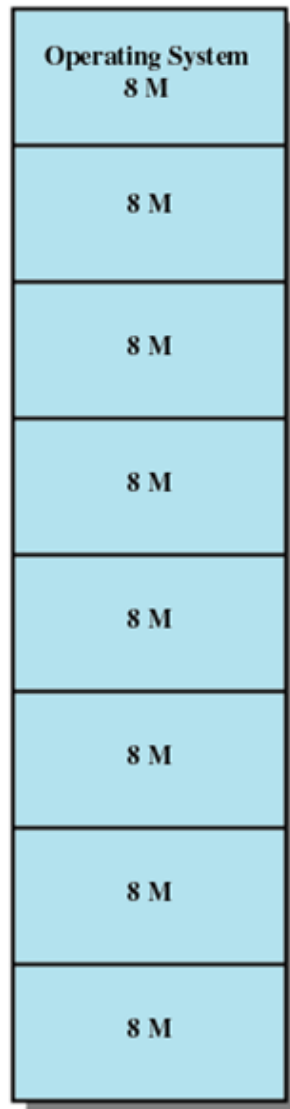
- Equal-size partitions
- Any process whose size is less than or equal to the partition size can be loaded into an available partition
 - If all partitions are full, the operating system can swap a process out of a partition

Difficulties

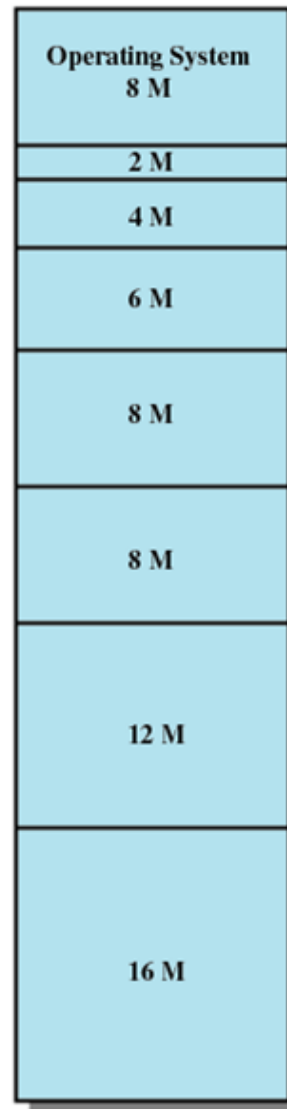
- Bigger processes - Overlaying
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation
- Fixed number of processes can be accommodated

Placement Algorithm with Partitions

- Equal-size partitions
 - Because all partitions are of equal size, it does not matter which partition is used
- Unequal-size partitions
 - Can assign each process to the smallest partition within which it will fit
 - Queue for each partition
 - Processes are assigned in such a way as to minimize wasted memory within a partition

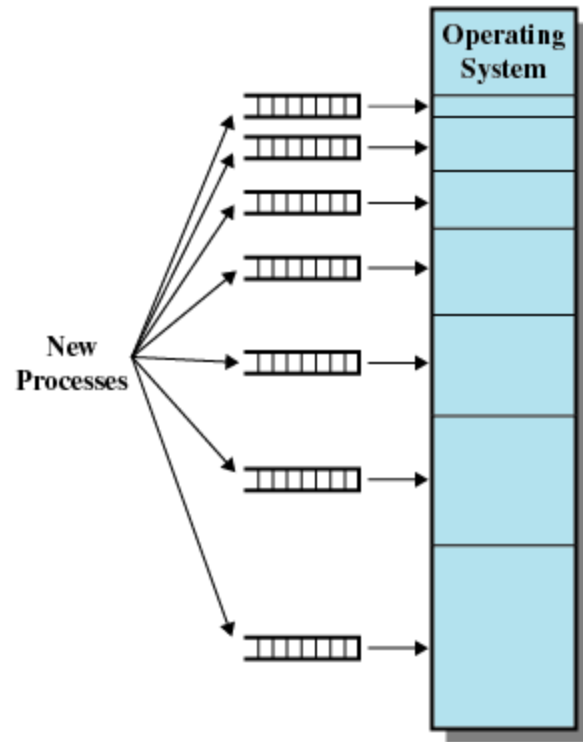


(a) Equal-size partitions

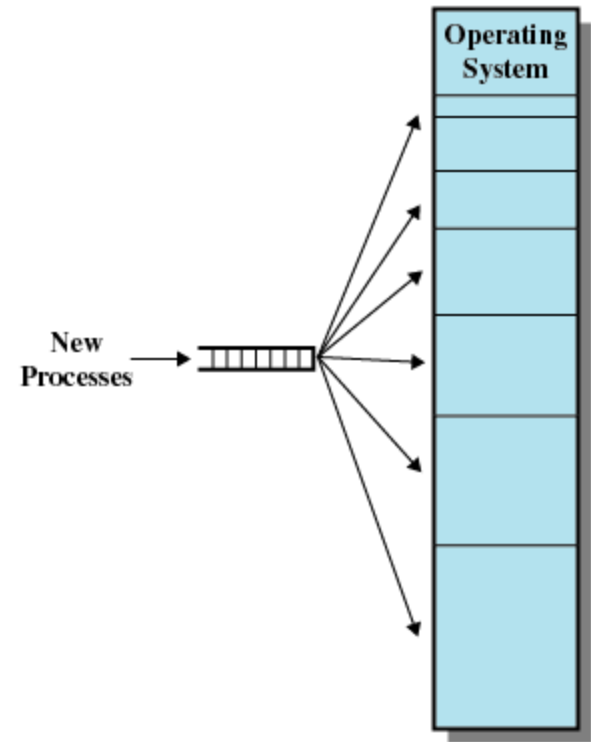


(b) Unequal-size partitions

Example of Fixed Partitioning of a 64-Mbyte Memory



(a) One process queue per partition

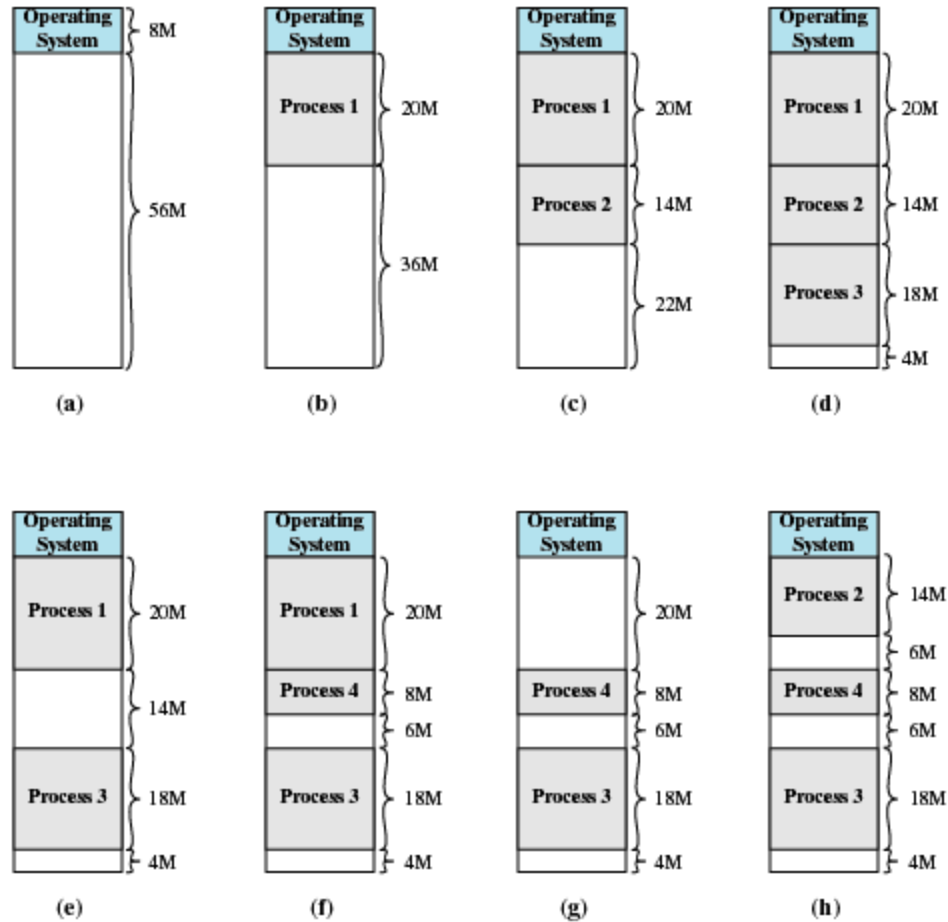


(b) Single queue

Memory Assignment for Fixed Partitioning

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required (IBM Mainframes)
- Eventually get holes in the memory. This is called *external fragmentation*
- Must use *compaction* to shift processes so they are contiguous and all free memory is in one block



The Effect of Dynamic Partitioning

Dynamic Partitioning Placement Algorithm

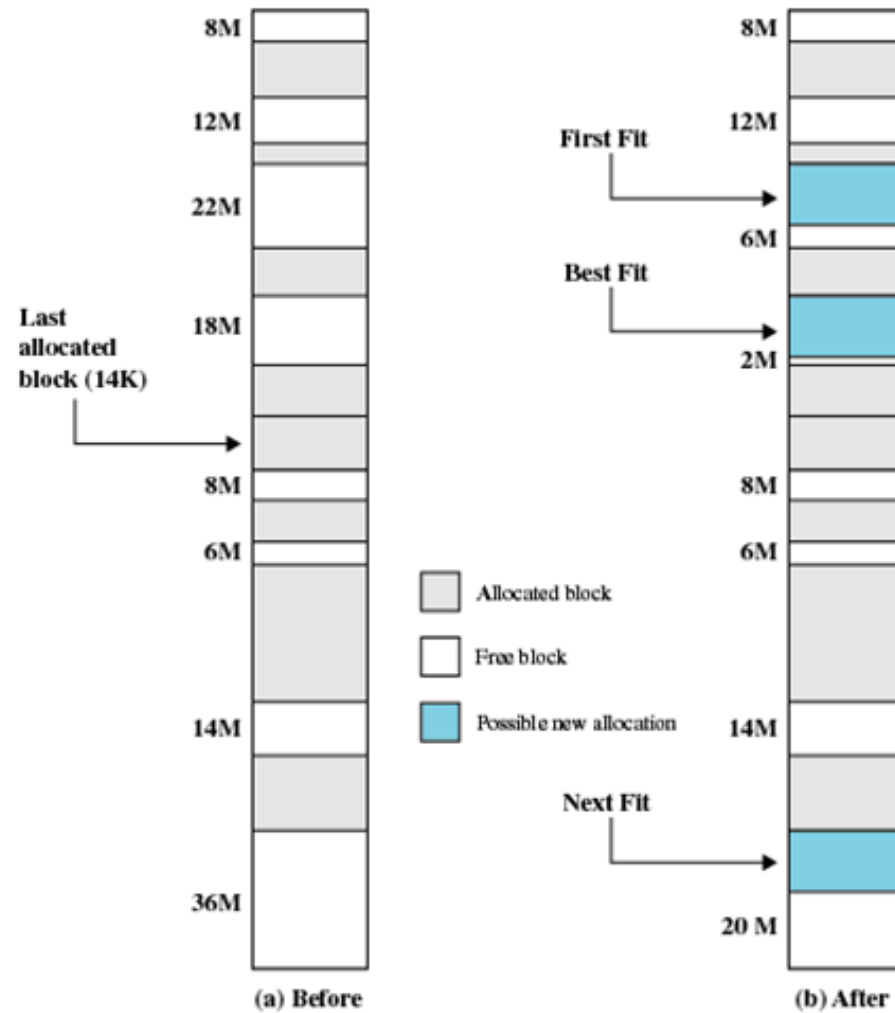
- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
 - Chooses the block that is closest in size to the request
 - Worst performer overall
 - Since smallest block is found for process, the smallest amount of fragmentation is left
 - Memory compaction must be done more often

Dynamic Partitioning Placement Algorithm

- First-fit algorithm
 - Scans memory from the beginning and chooses the first available block that is large enough
 - Fastest
 - May have many process loaded in the front end of memory that must be searched over when trying to find a free block

Dynamic Partitioning Placement Algorithm

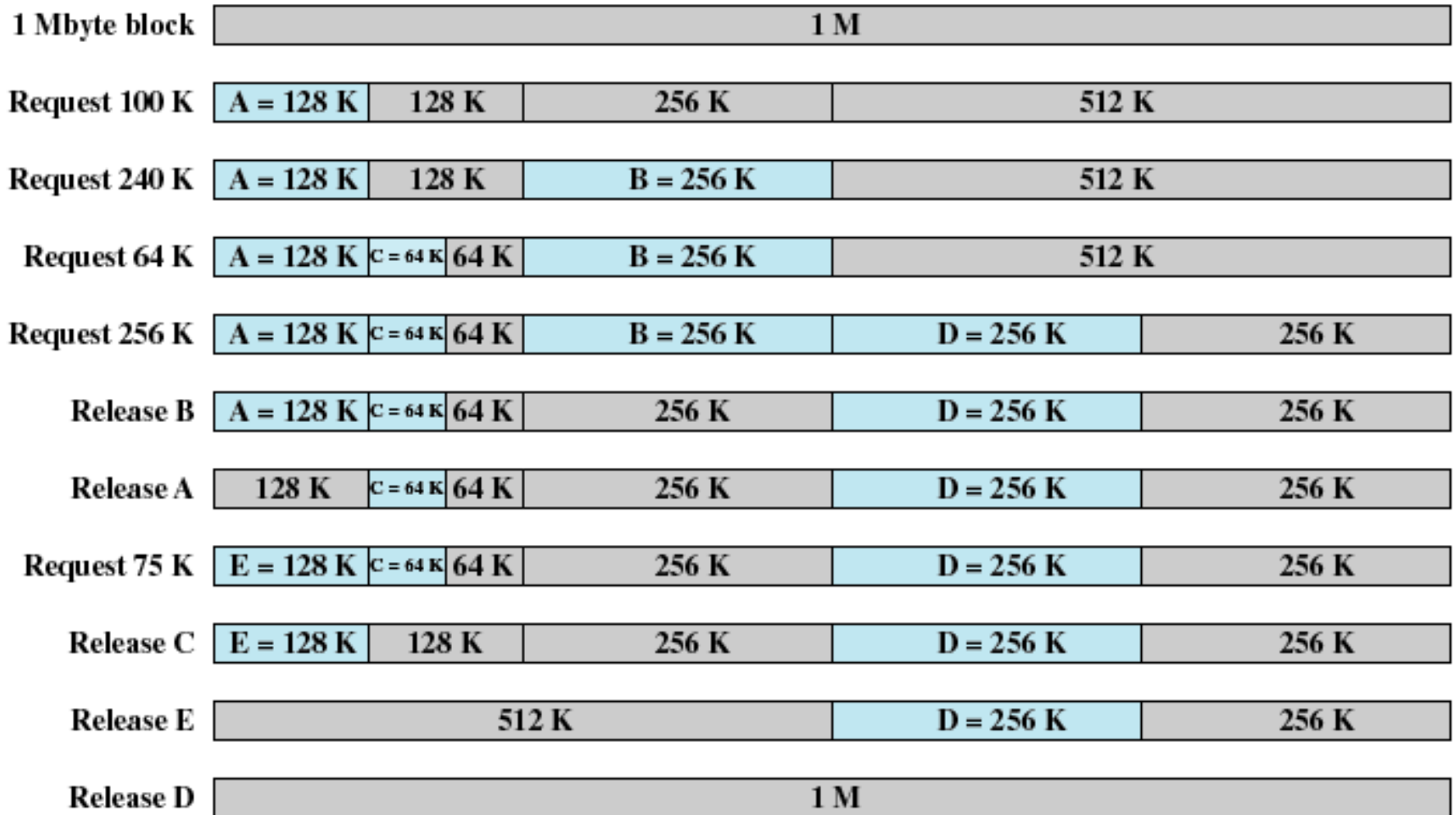
- Next-fit
 - Scans memory from the location of the last placement
 - More often allocate a block of memory at the end of memory where the largest block is found
 - The largest block of memory is broken up into smaller blocks
 - Compaction is required to obtain a large block at the end of memory



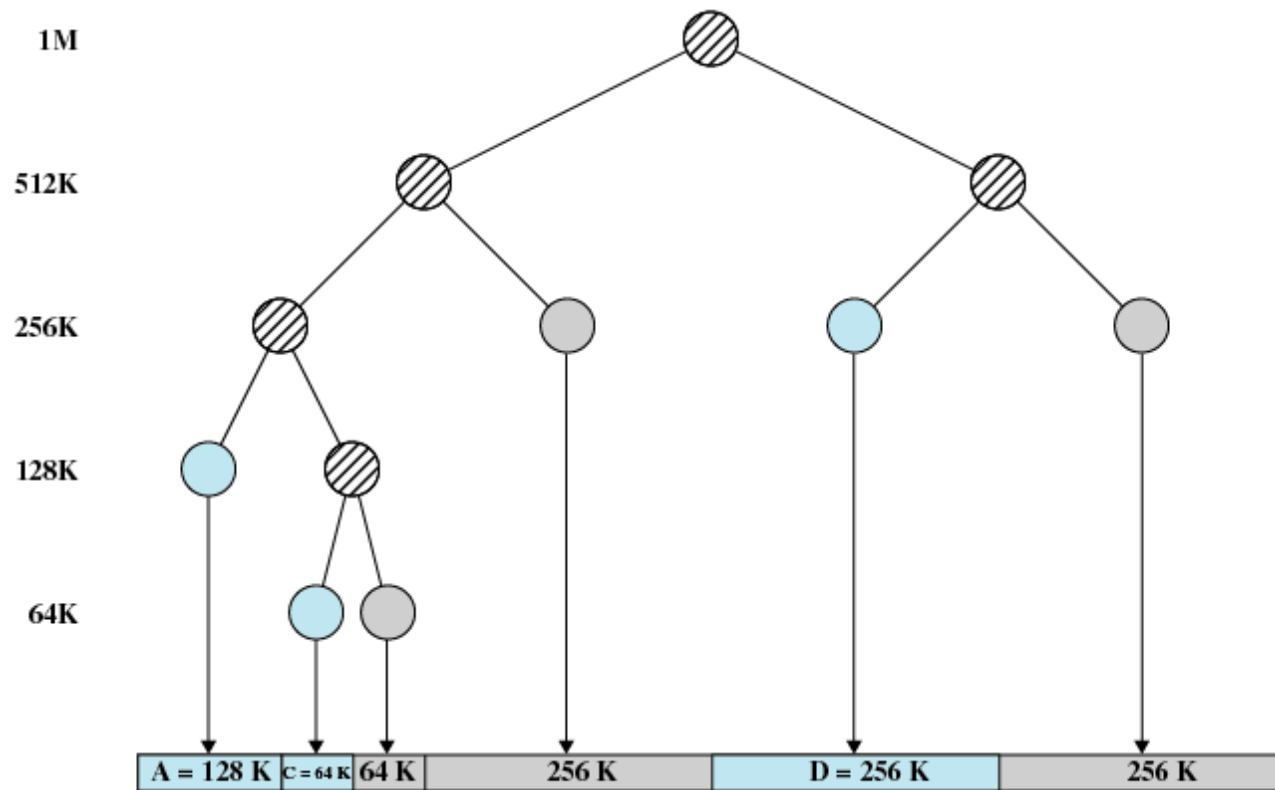
Example Memory Configuration Before and After Allocation of 16 Mbyte Block

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s \leq 2^U$, entire block is allocated
 - Otherwise block is split into two equal buddies
 - Process continues until smallest block greater than or equal to s is generated
- Used for Parallel Systems



Example of Buddy System

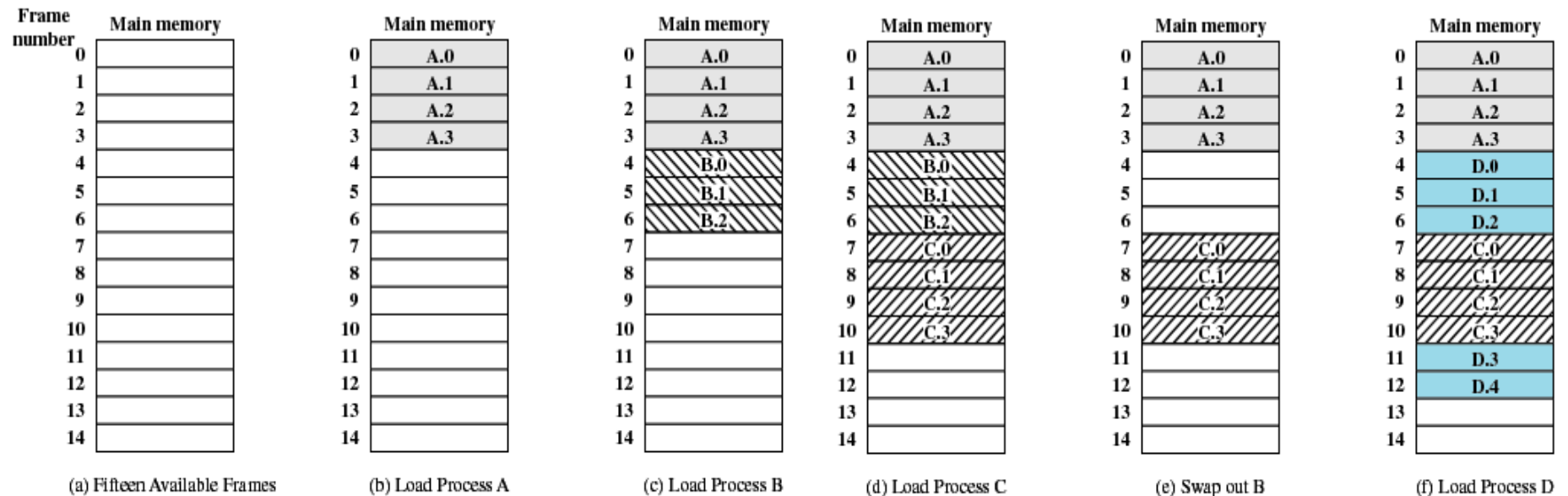


Tree Representation of Buddy System

Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called pages and chunks of memory are called frames
- Operating system maintains a page table for each process
 - Contains the frame location for each page in the process
 - Memory address consist of a page number and offset within the page

Assignment of Process Pages to Free Frames



Assignment of Process Pages to Free Frames

Page Tables for Example

0	0
1	1
2	2
3	3

Process A
page table

0	N
1	N
2	N

Process B
page table

0	7
1	8
2	9
3	10

Process C
page table

0	4
1	5
2	6
3	11
4	12

Process D
page table

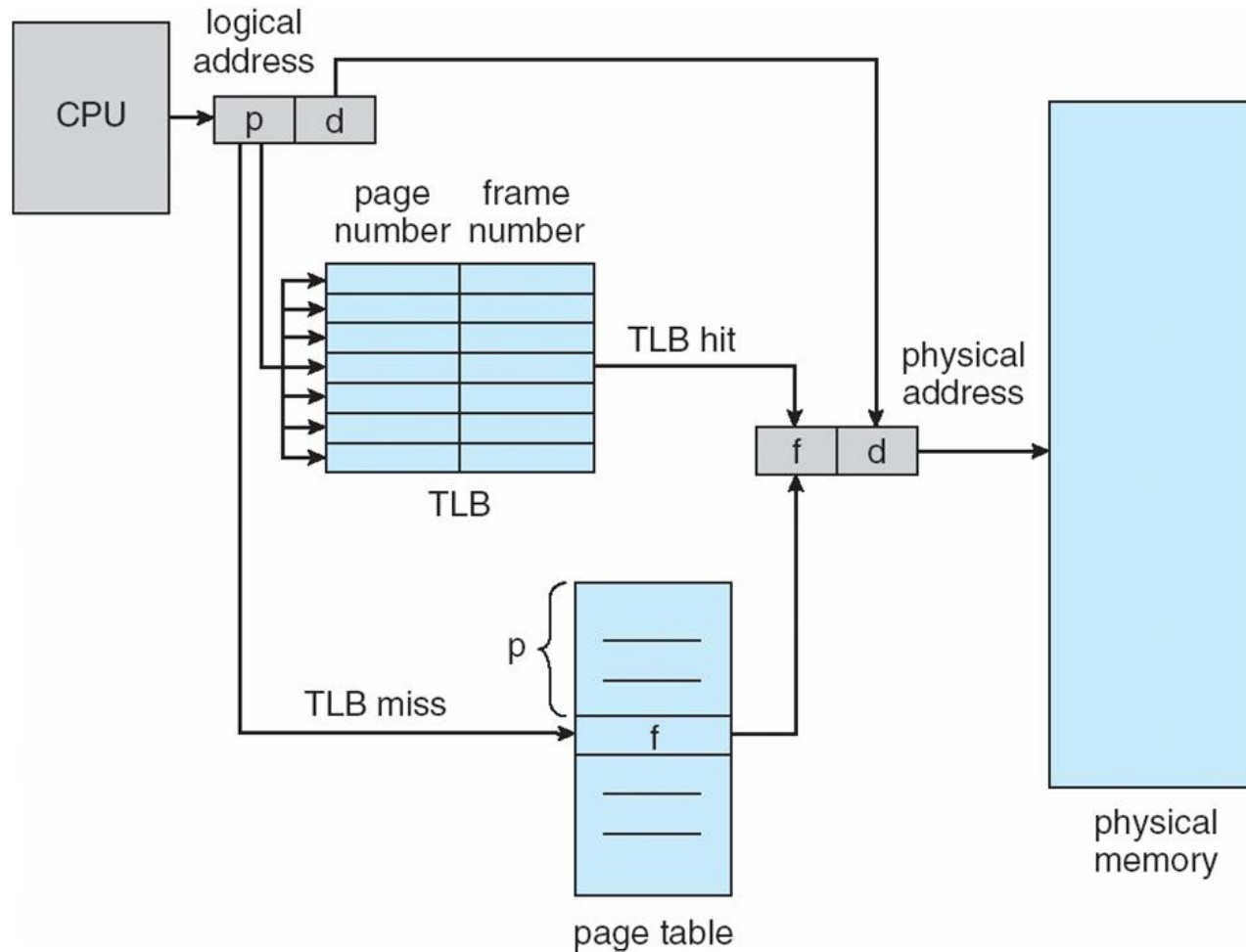
13
14

Free frame
list

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 (PRLR)** 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)**
- TLB keeps some of the recently used physical address mappings.
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
- In case of TLB miss a *Page Walk* needs to be performed.

Paging Hardware With TLB

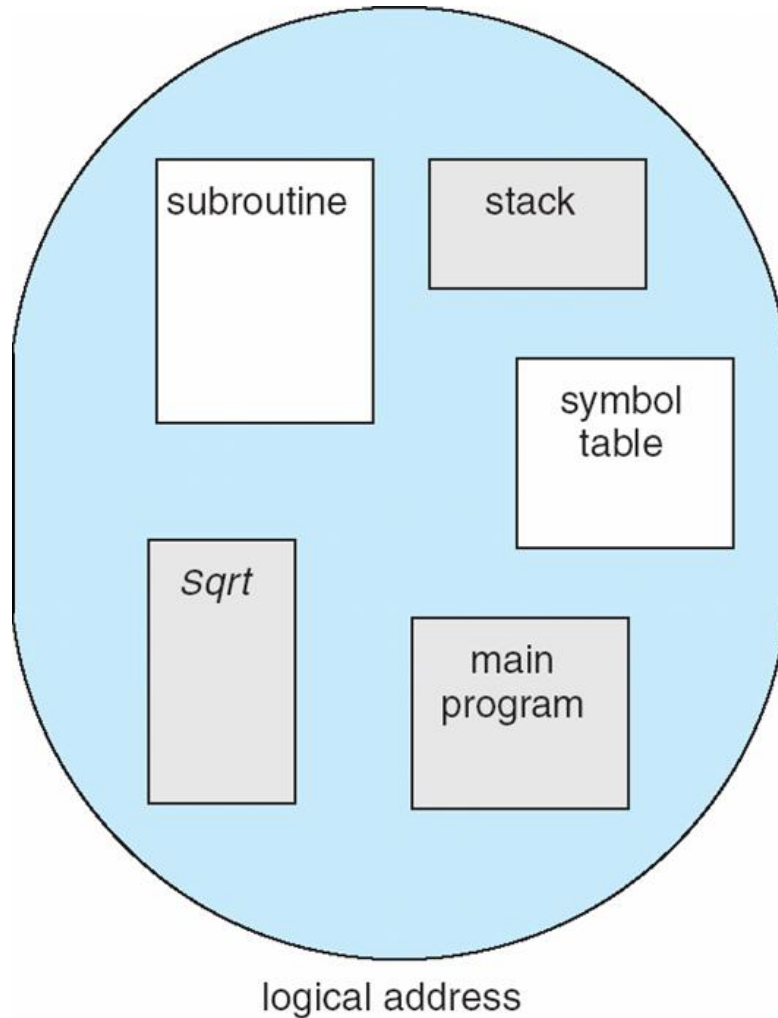


Effective Access Time

Time required

- if page is in TLB
 - $HIT(T) = TLB(T) + MEM(T)$
- if page is not available in TLB
 - $MISS(T) = TLB(T) + PAGE(T) + MEM(T)$
- $EAT = h * HIT(T) + m * MISS(T)$
- h = hit ratio
- m = miss ratio or 1-hit ratio

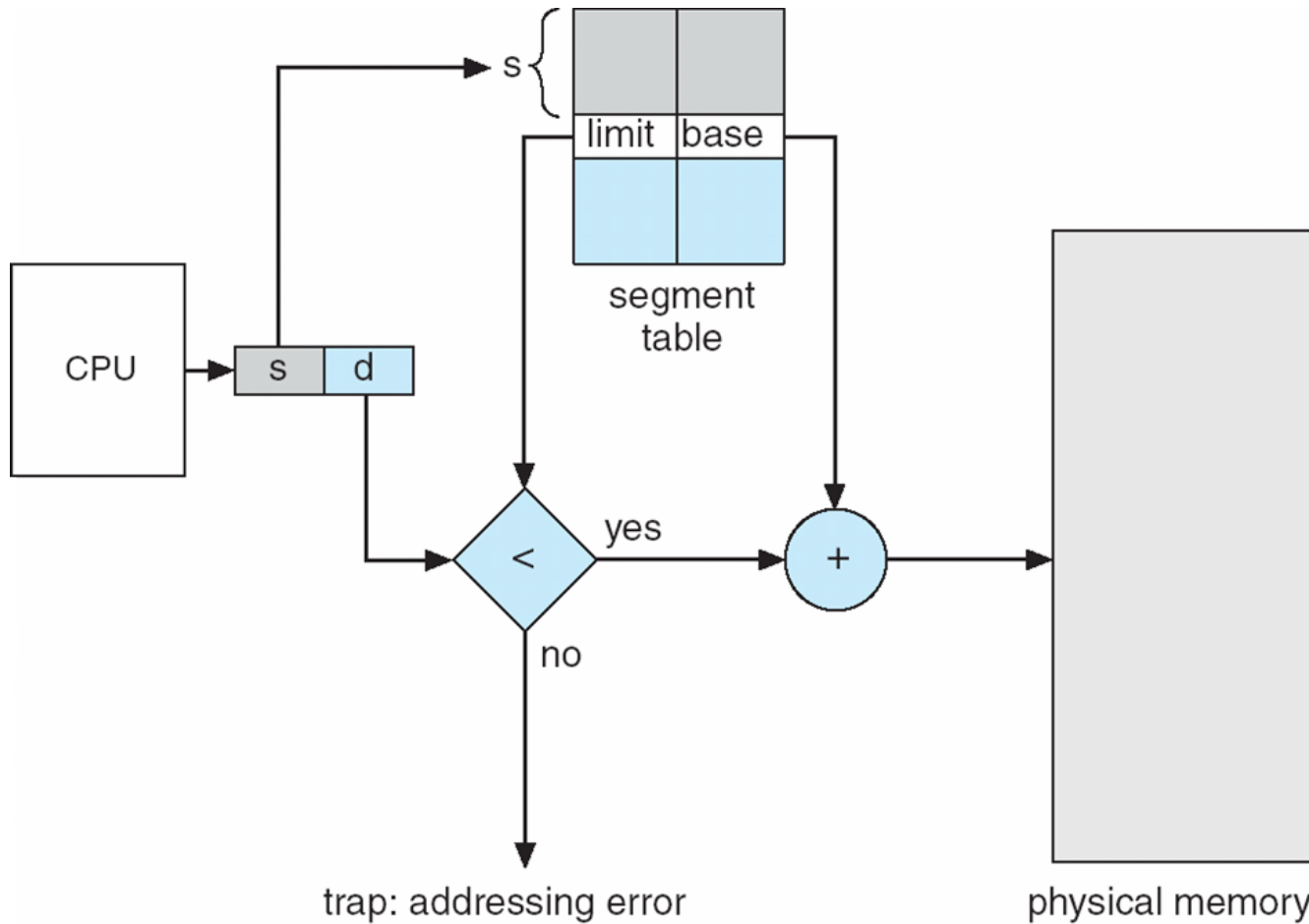
User's View of a Program



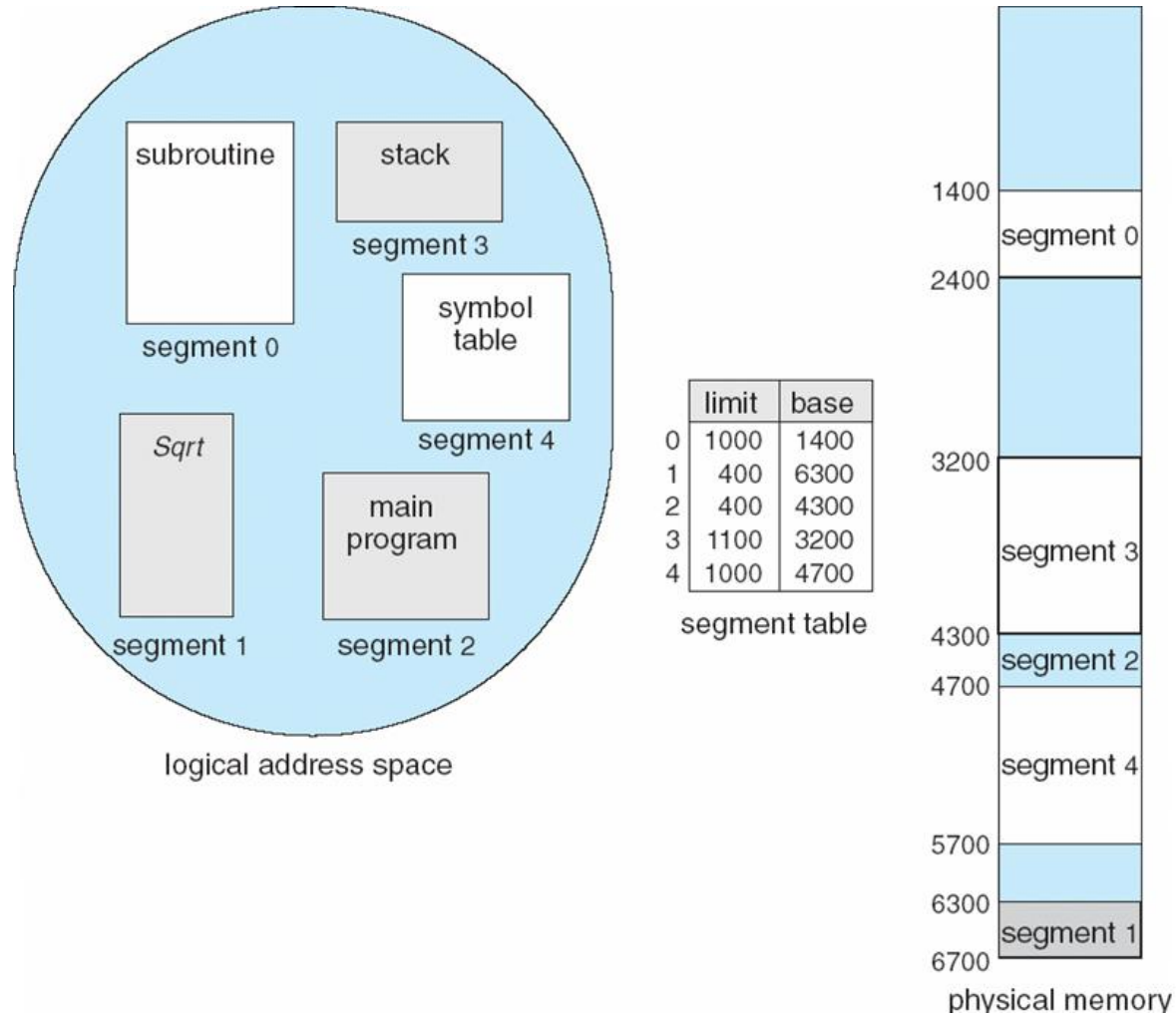
Segmentation Architecture

- Logical address consists of a two tuple:
 $\langle \text{segment-number}, \text{offset} \rangle$
- **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 table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program; segment number s is legal if $s < \text{STLR}$

Segmentation Hardware



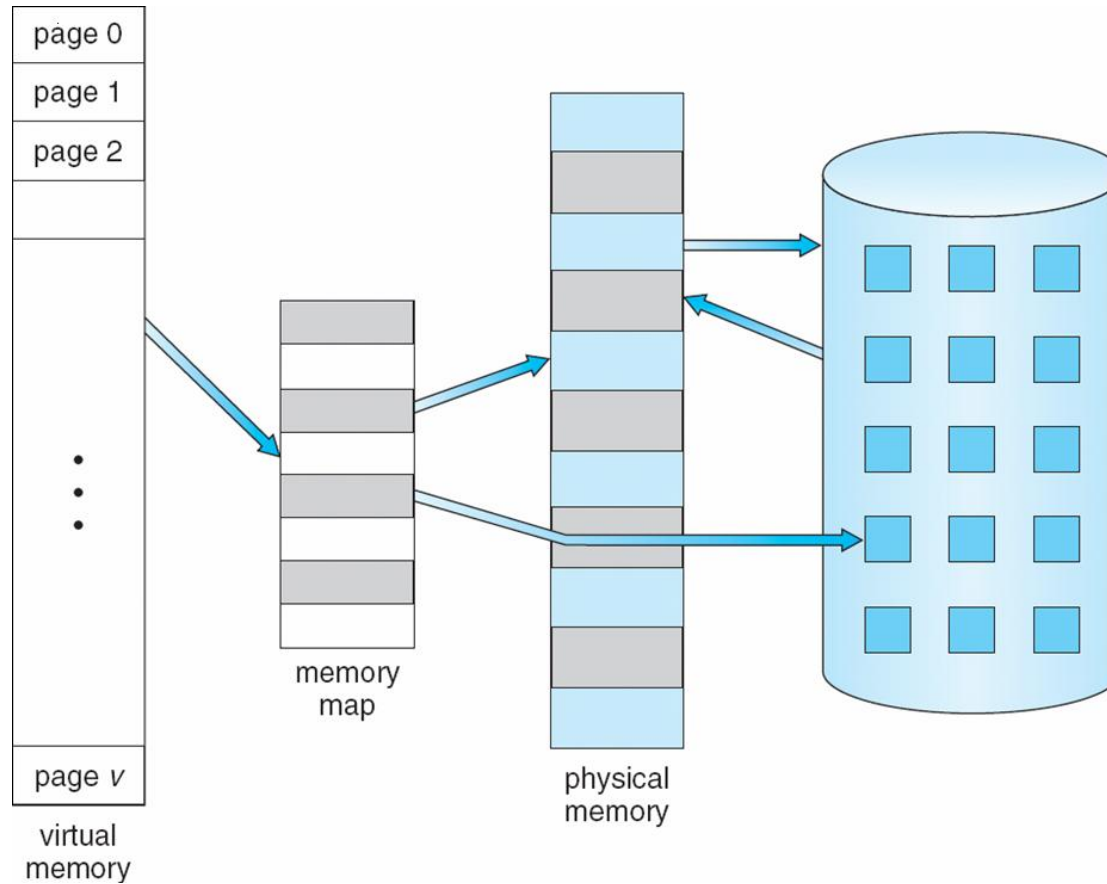
Example of Segmentation



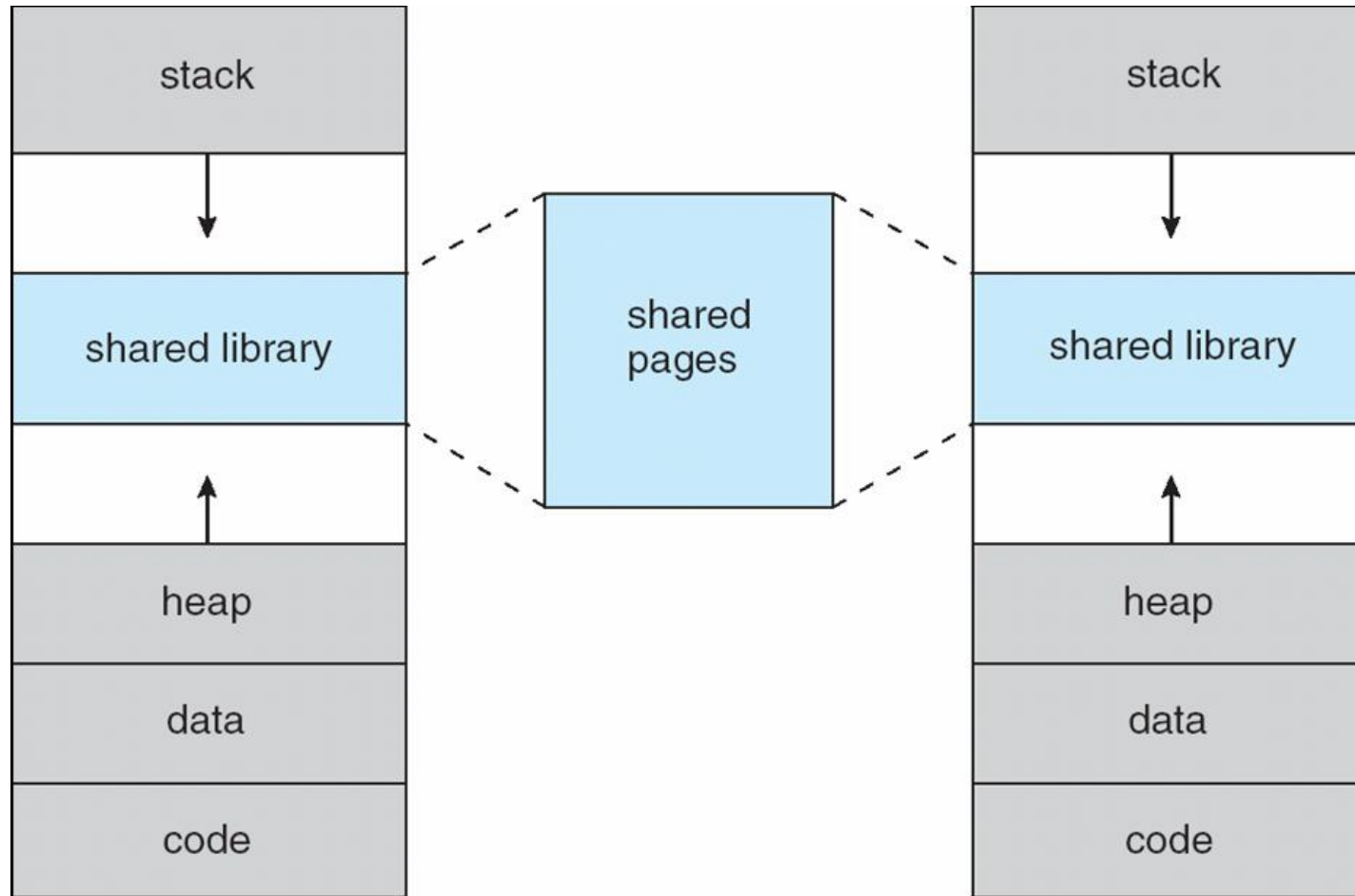
Background

- **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
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



Shared Library Using Virtual Memory



Demand Paging

- Bring a page into memory only when it is needed
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**

Valid-Invalid Bit

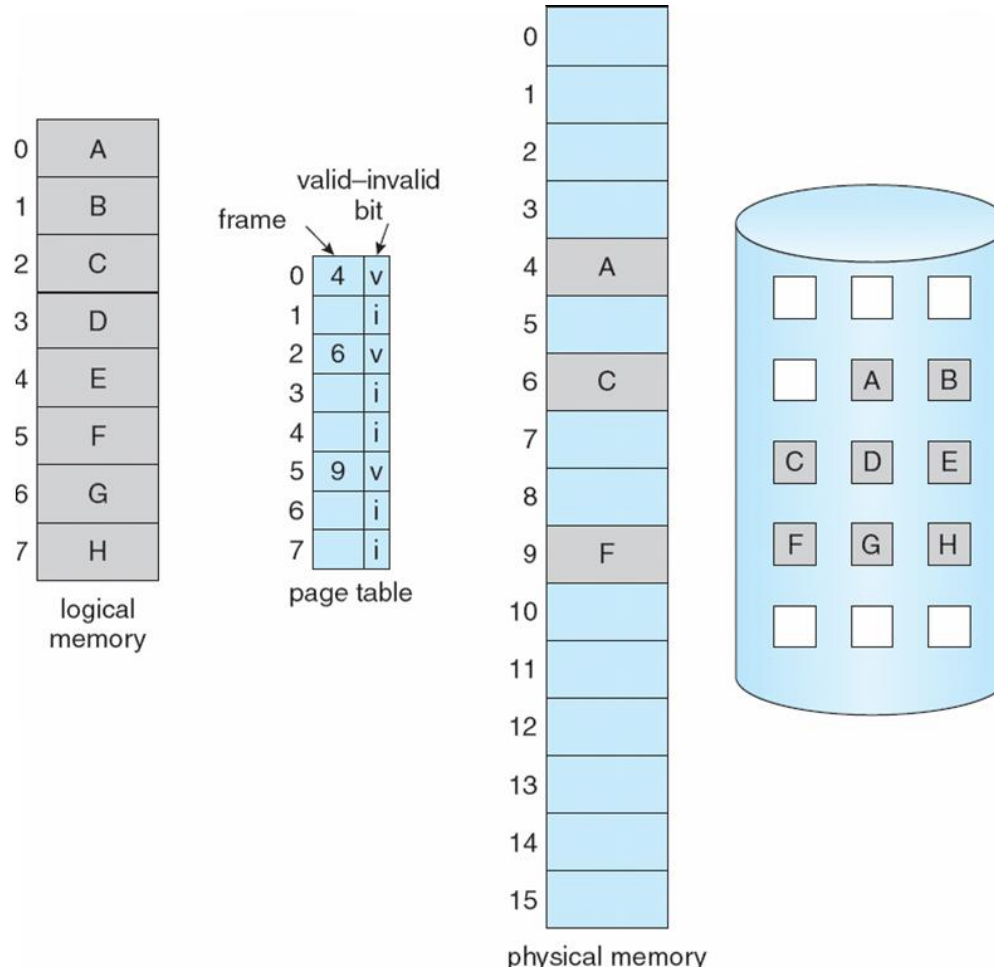
- With each page table entry a valid–invalid bit is associated (**v** \Rightarrow in-memory, **i** \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is **I** \Rightarrow page fault

Page Table When Some Pages Are Not in Main Memory

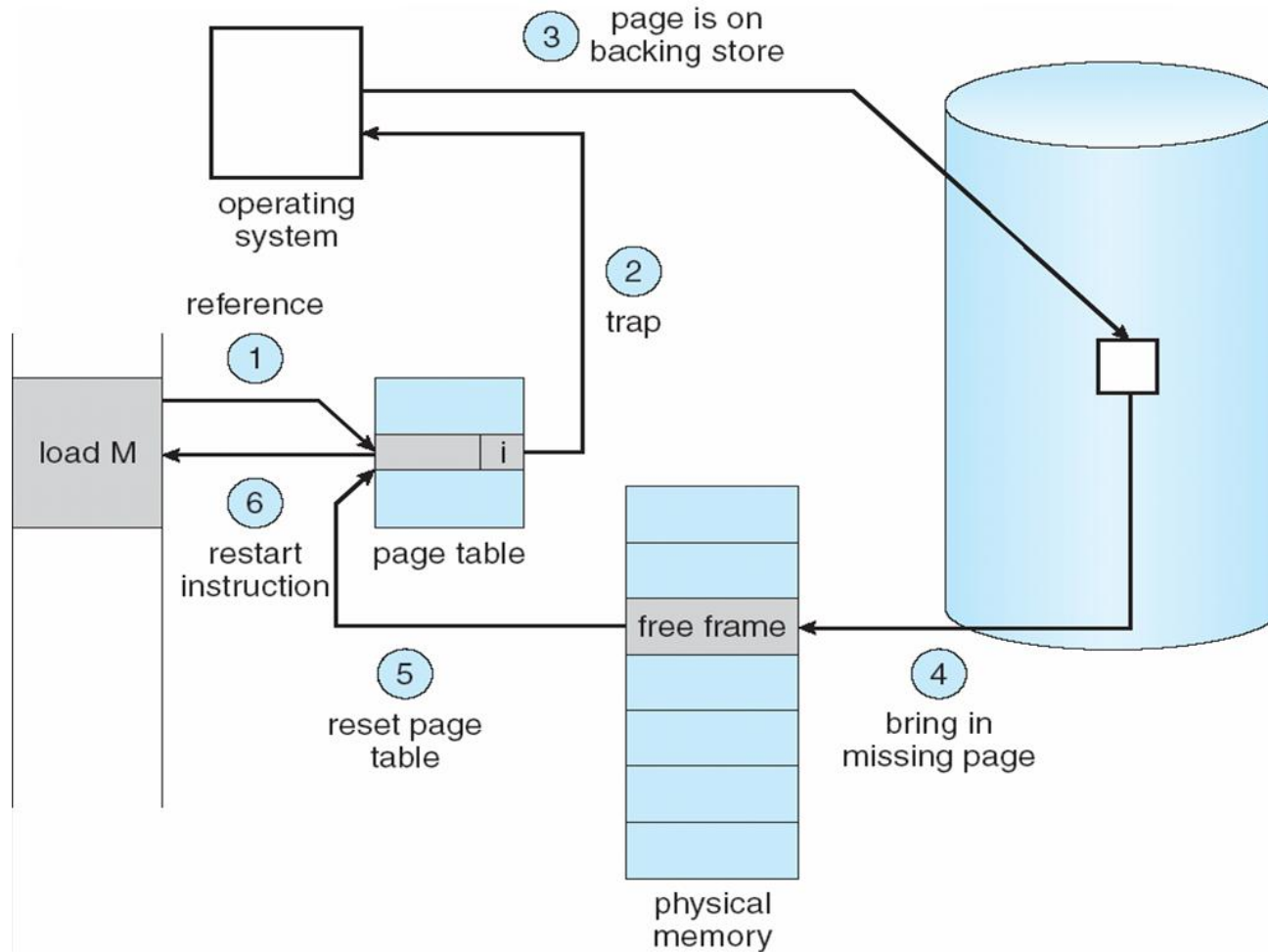


Page Fault

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. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead} \\ &) \end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then
 $\text{EAT} = 8.2 \text{ microseconds.}$
 This is a slowdown by a factor of 40!!

Process Creation

- Virtual memory allows other benefits during process creation:
 - Copy-on-Write

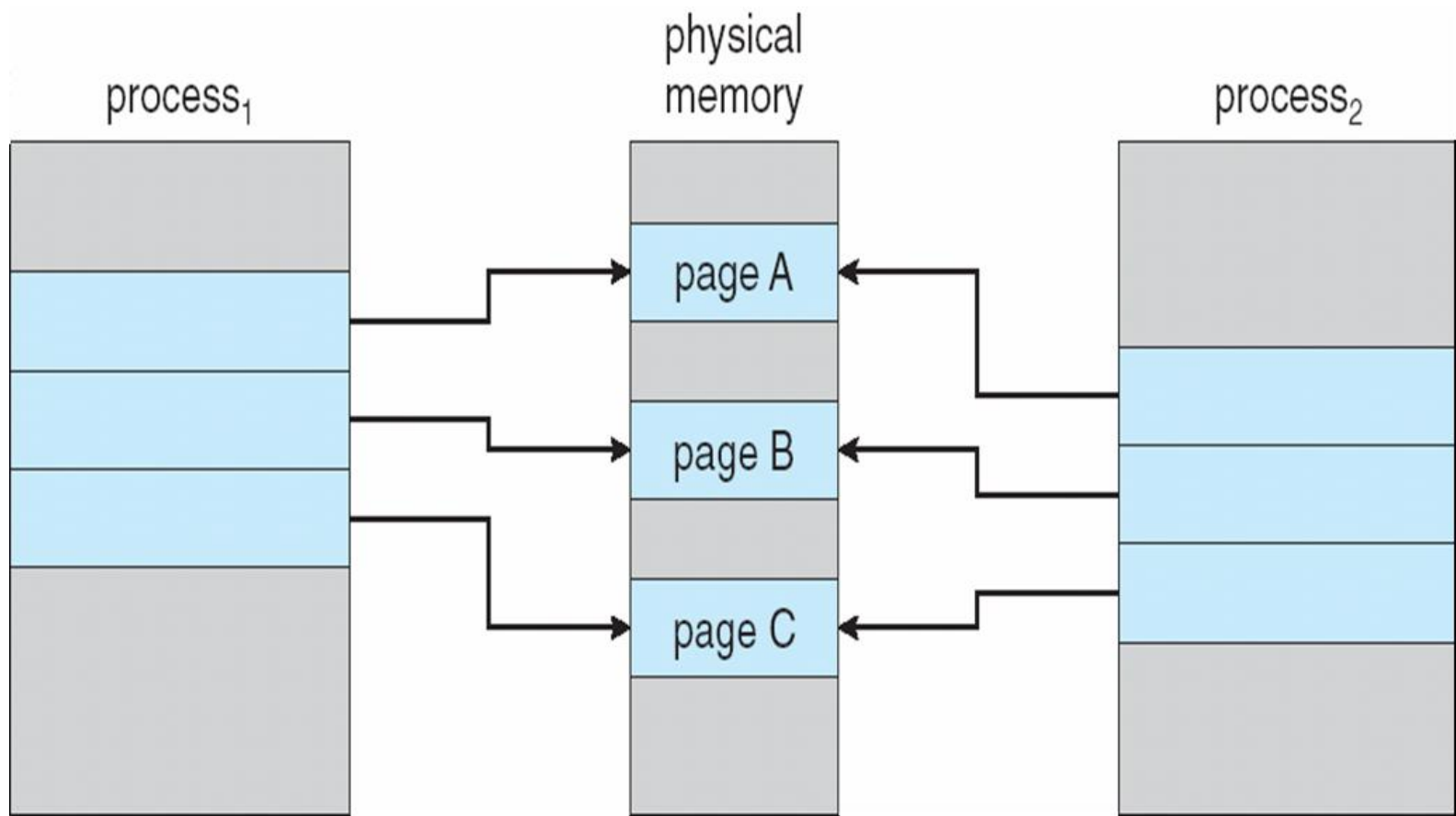
Copy-on-Write

- Copy-on-Write allows both parent and child processes to initially *share* the same pages in memory

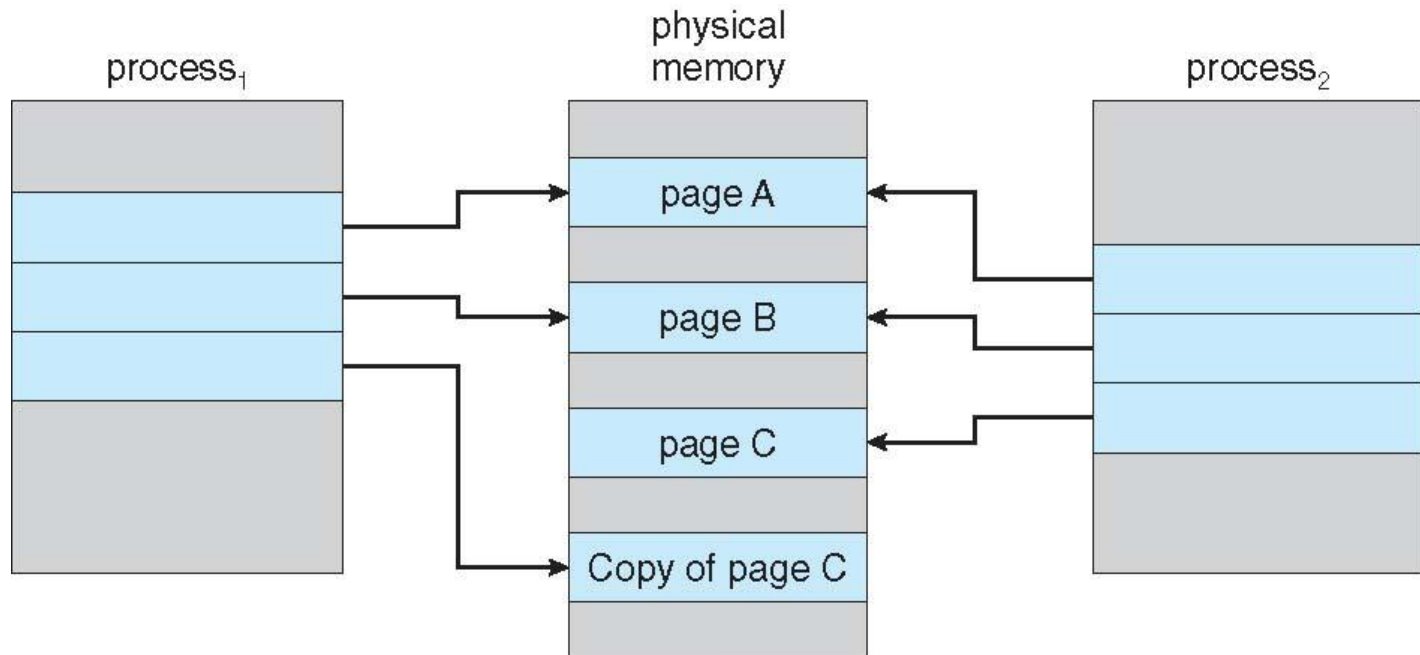
If either process modifies a shared page, only then is the page copied

- It allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C

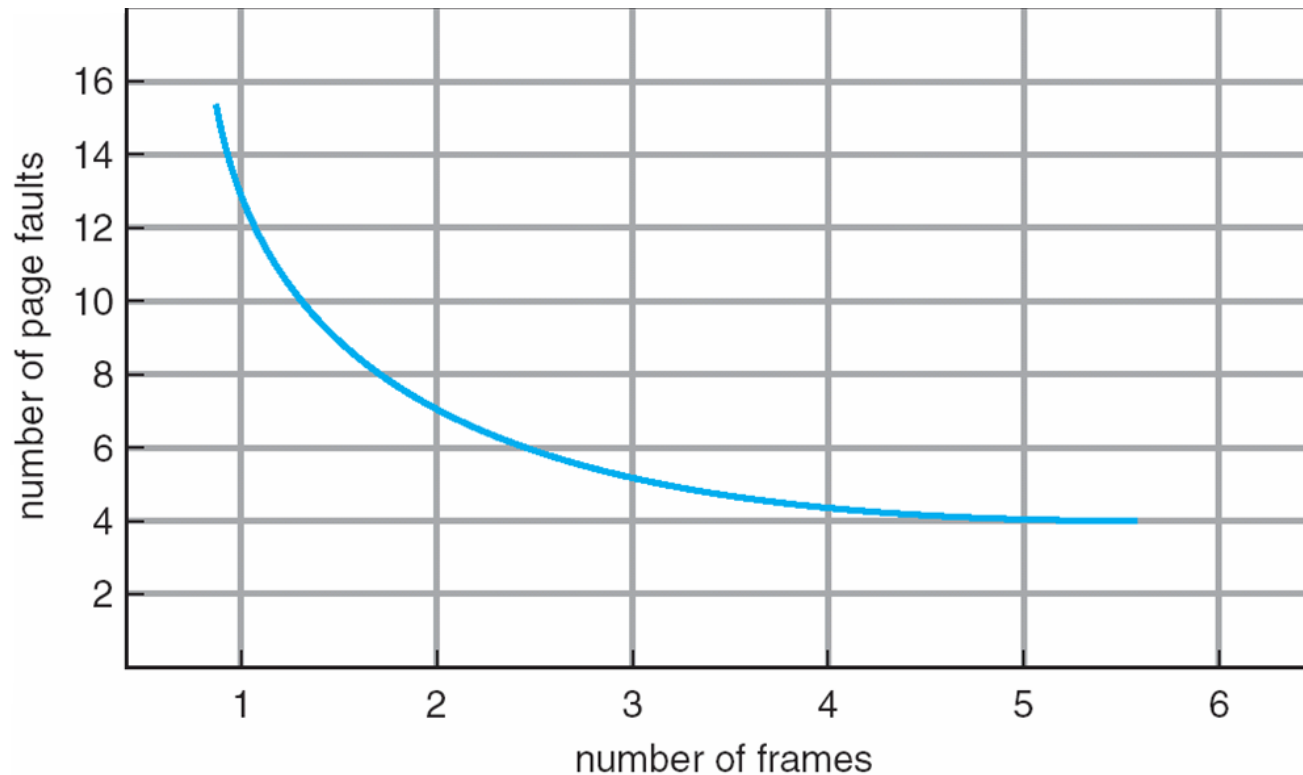


Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Belady's Anomaly

FIFO Page Replacement

reference string

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

7	7	7	2																
	0	0	0																
		1	1																

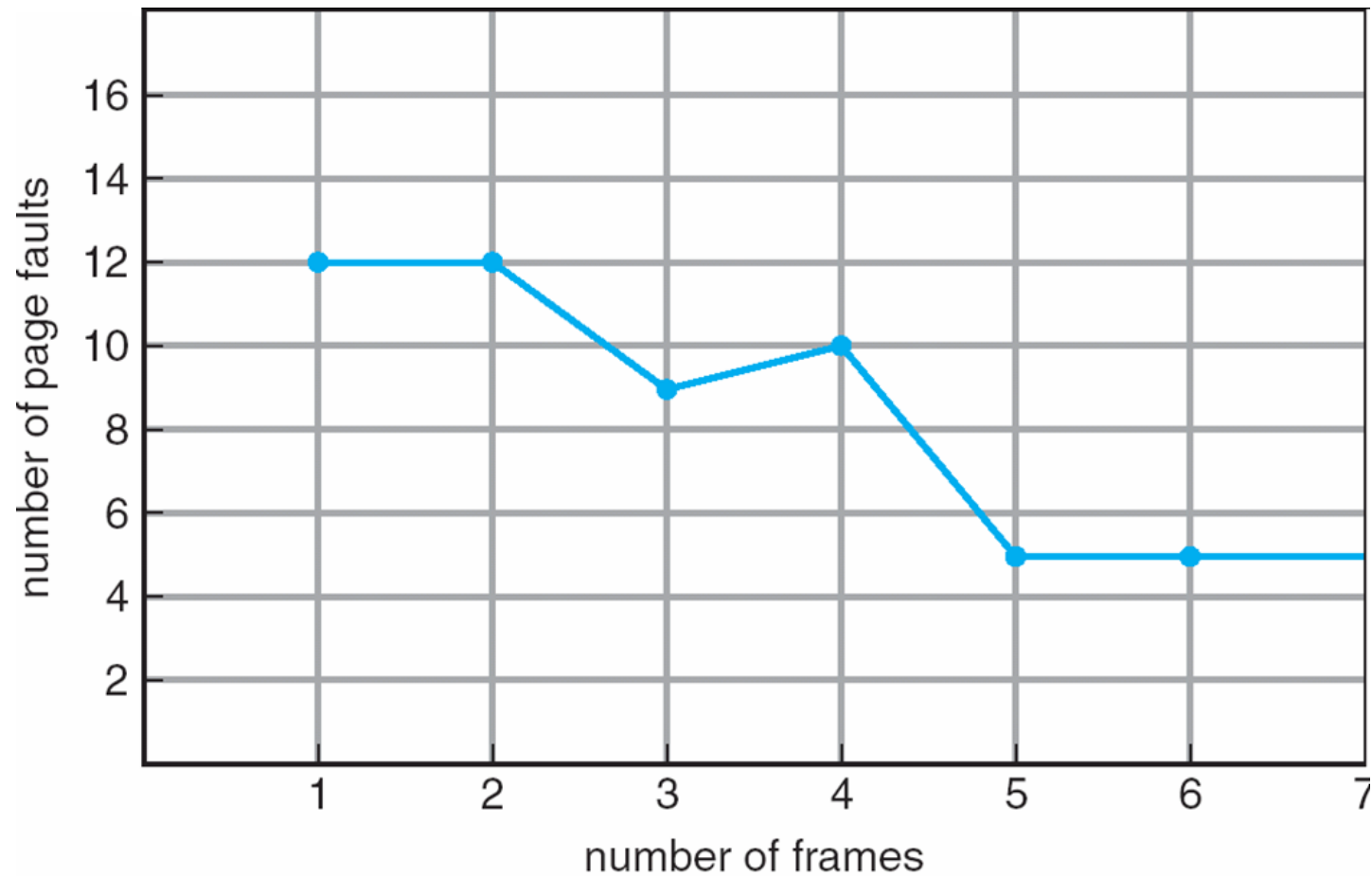
2	2	4	4	4	0														
3	3	3	2	2	2														
1	0	0	0	3	3														

0	0																		
1	1																		
3	2																		

7	7	7																	
1	0	0																	
2	2	1																	

page frames

FIFO Illustrating Belady's Anomaly



Optimal 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

1
2
3
4

6 page faults

- Future knowledge
- Used to measure how well your algorithm performs

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	5
2	2	2	2	2
3	5	5	4	4
4	4	3	3	3

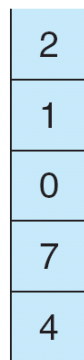
- Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to determine which are to change

LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
 - Page referenced:
 - move it to the top
 - requires 6 pointers to be changed
 - No search for replacement

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



LRU Approximation Algorithms

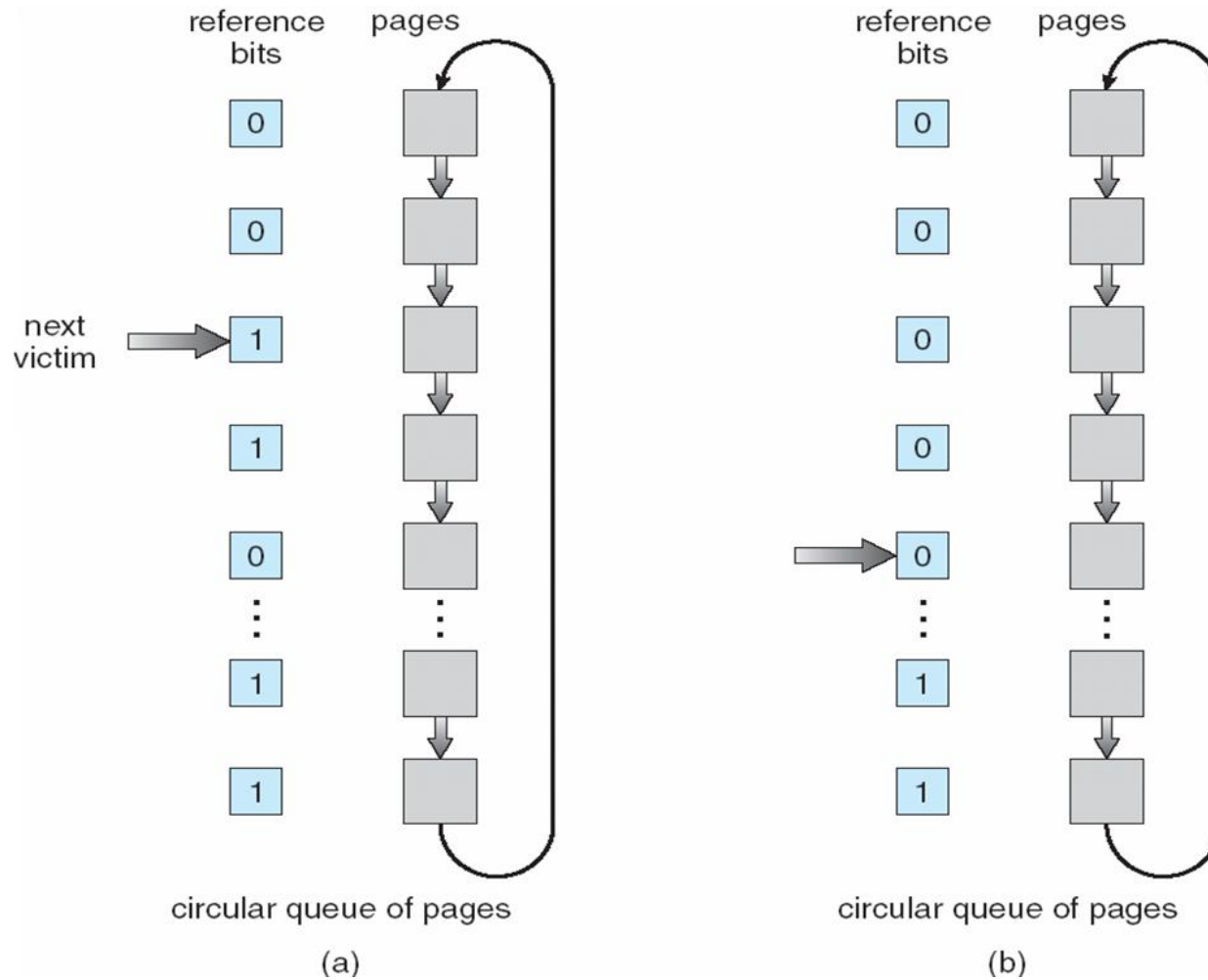
■ Reference bit

- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace the one which is 0 (if one exists)
 - We do not know the order, however

■ Second chance

- Need reference bit
- Clock replacement
- If page to be replaced (in clock order) has reference bit = 1 then:
 - set reference bit 0
 - leave page in memory
 - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm:** replaces page with smallest count
- **MFU Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs *minimum* number of pages
- 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*
- Two major allocation schemes
 - fixed allocation
 - priority allocation

Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

s_i = size of process p_i

$$S = \sum s_i$$

m = total number of frames

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

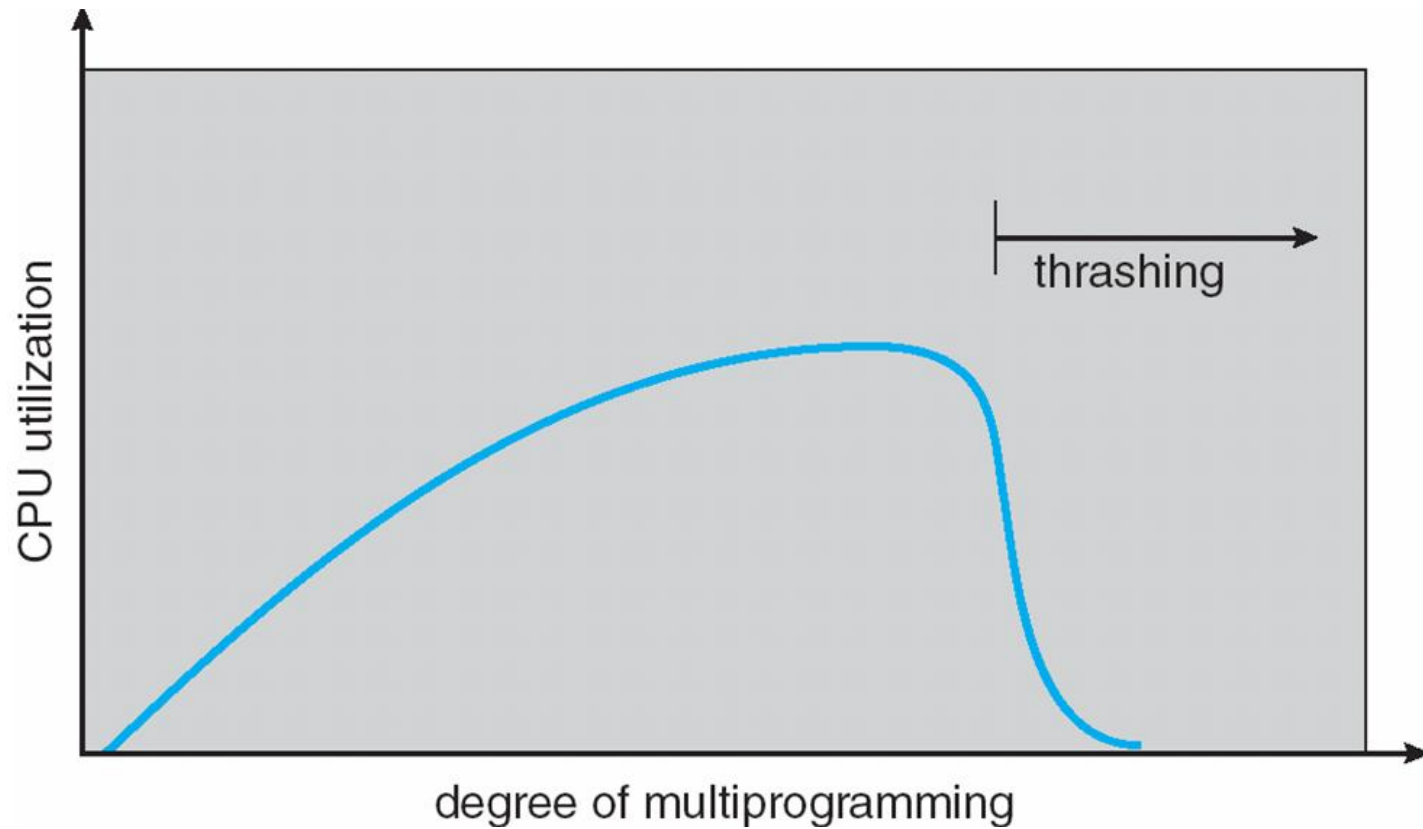
Priority Allocation

- 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
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing



Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - operating system thinks that it needs to increase the degree of multiprogramming
 - another process added to the system
- **Thrashing** \equiv a process is busy swapping pages in and out

Demand Paging and Thrashing

- Why does demand paging work?
Locality model
 - Process migrates from one locality to another
 - Localities may overlap
- Why does thrashing occur?
 Σ size of locality > total memory size

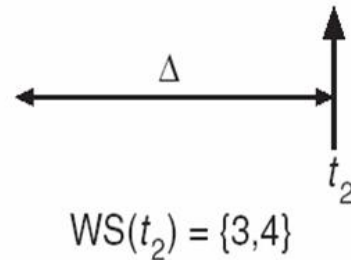
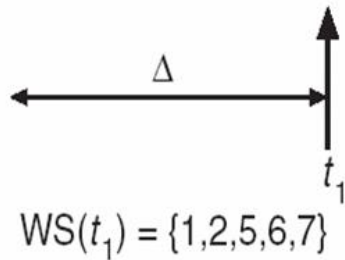
Working-Set Model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instruction
- 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
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes

Working-set model

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



Thanks