# Computer Organization and Software Systems

## Session 13

Prof. R Sarma
WILP.BITS.Pilani

**BITS** Pilani
Pilani Campus

# Last Session

| Contact Hour | List of Topic Title | Text/Ref Book/external resource |
|---|---|---|
| 23-24 | • Dead Lock (Deadlock Prevention and Deadlock Avoidance) | T2 |

# Today's Session

innovate achieve lead

| Contact Hour | List of Topic Title | Text/Ref Book/external resource |
|---|---|---|
| 25-26 | • Dead Lock (Deadlock Detection)<br>• Memory Management | T2 |

# Example of Banker's Algorithm

5 processes $P_0$ through $P_4$;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances).

Snapshot at time $T_0$:

| | Allocation | Max | Available | Need |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | |
| $P_1$ | 2 0 0 | 3 2 2 | | |
| $P_2$ | 3 0 2 | 9 0 2 | | |
| $P_3$ | 2 1 1 | 2 2 2 | | |
| $P_4$ | 0 0 2 | 4 3 3 | | |

# Example of Banker's Algorithm

5 processes $P_0$ through $P_4$;

3 resource types:

A (10 instances), B (5instances), and C (7 instances).

Snapshot at time $T_0$:

| | Allocation | Max | Available | Need |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | 7 4 3 |
| $P_1$ | 2 0 0 | 3 2 2 | | 1 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 | | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | | 4 3 1 |

# Example (Cont.)

The content of the matrix *Need* is defined to be *Max* – *Allocation*.

| | *Allocation* | *Max* | *Available* | *Need* |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | 7 4 3 |
| $P_1$ | 2 0 0 | 3 2 2 | | 1 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 | | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | | 4 3 1 |

The system is in a safe state since the sequence $< P_1, P_3, P_4, P_0, P_2>$ satisfies safety criteria.

Work = 332

Finish = { F F F F F }

P1: Finish [1] = T;
Work = 5 3 2

P3: Finish [3] = T;
Work = 7 4 3

P4: Finish [4] = T;
Work = 7 4 5

P0: Finish [0] = T;
Work = 7 5 5

P2: Finish [2] = T;
Work = 10 5 7

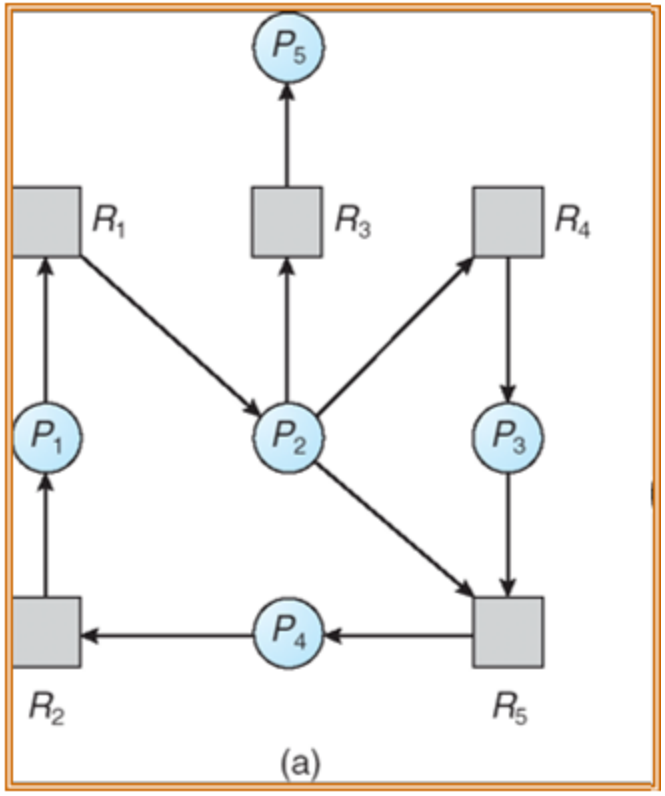
# Deadlock Detection

- Allow system to enter deadlock state
  - Detection algorithm
  - Recovery scheme

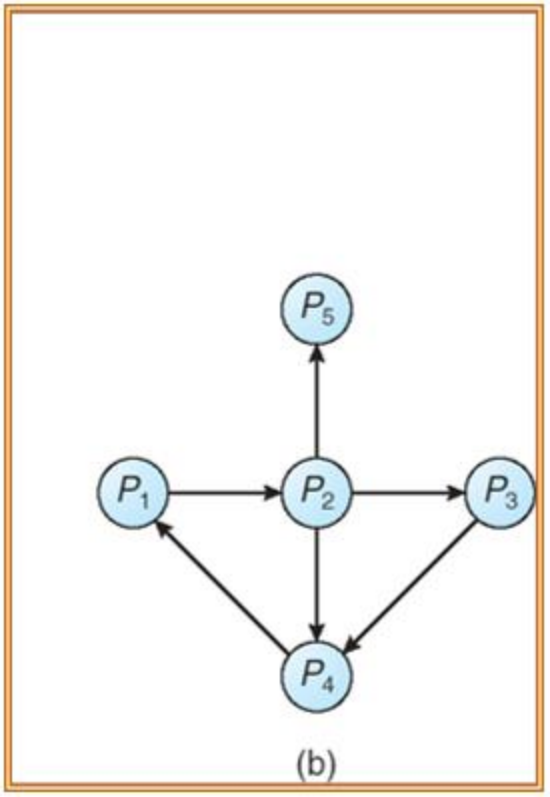
# Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$ to release the resource
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph.

# Resource-Allocation Graph and Wait-for Graph




Resource-Allocation Graph




Corresponding wait-for graph

# Several Instances of a Resource Type

***Available***: A vector of length *m* indicates the number of available resources of each type.

***Allocation***: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

***Request***: An *n* x *m* matrix indicates the current request of each process. If *Request* [*i*,*j*] = *k*, then process $P_i$ is requesting *k* more instances of resource type $R_j$.

# Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4.
3. *Work* = *Work* + $Allocation_i$
   *Finish*[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then $P_i$ is deadlocked.

# Example of Detection Algorithm

Five processes $P_0$ through $P_4$; three resource types
*A* (7 instances), *B* (2 instances), and *C* (6 instances).

Snapshot at time $T_0$:

| | *Allocation* | *Request* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 0 0 0 | 0 0 0 |
| $P_1$ | 2 0 0 | 2 0 2 | |
| $P_2$ | 3 0 3 | 0 0 0 | |
| $P_3$ | 2 1 1 | 1 0 0 | |
| $P_4$ | 0 0 2 | 0 0 2 | |

Sequence $\langle P_0, P_2, P_3, P_4, P_1 \rangle$ will result in *Finish*[*i*] = true for all *i*.

# Example (Cont.)

$P_2$ requests an additional instance of type *C*.

| | Request A B C |
|---|---|
| $P_0$ | 0 0 0 |
| $P_1$ | 2 0 1 |
| $P_2$ | 0 0 1 |
| $P_3$ | 1 0 0 |
| $P_4$ | 0 0 2 |

State of system?

- Can reclaim resources held by process $P_0$, but insufficient resources to fulfill other processes; requests.
- Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$, and $P_4$.

# Recovery from Deadlock: Process Termination

1) Abort all deadlocked processes.

2) Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?
  - Priority of the process, how long the process has computed and resources a process needs.
- Cost of process abortion
  - Priority of the process,
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated?
  - Is process interactive or batch?

innovate achieve lead

BITS Pilani

Pilani Campus

# Memory Management

# Memory Management Requirements



- Five requirements
  - Relocation
  - Protection
  - Sharing
  - Logical organization
  - Physical organization

# Memory Management Requirement 1: Relocation



- Program must be brought into memory and placed within a process for it to be executed
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- The programmer does not know where the program will be placed in memory when it is executed,
  - it may be swapped to disk and return to main memory at a different location (relocated)
- Memory references must be translated to the actual physical memory address
- **Address binding** : Mapping from one address space to another

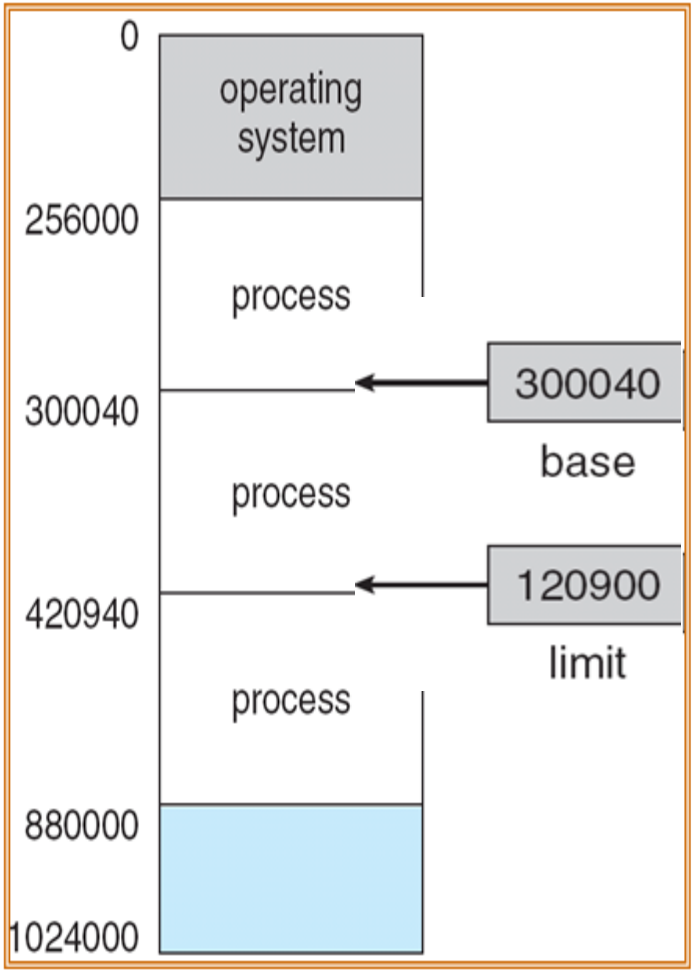
# Memory Management Requirement 1: Relocation

- 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**: Compiler must generate *relocatable* code if memory location is not known at compile time.
    - Final address binding will be done at load time
    - If starting address changes then reload the program
  - **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*).

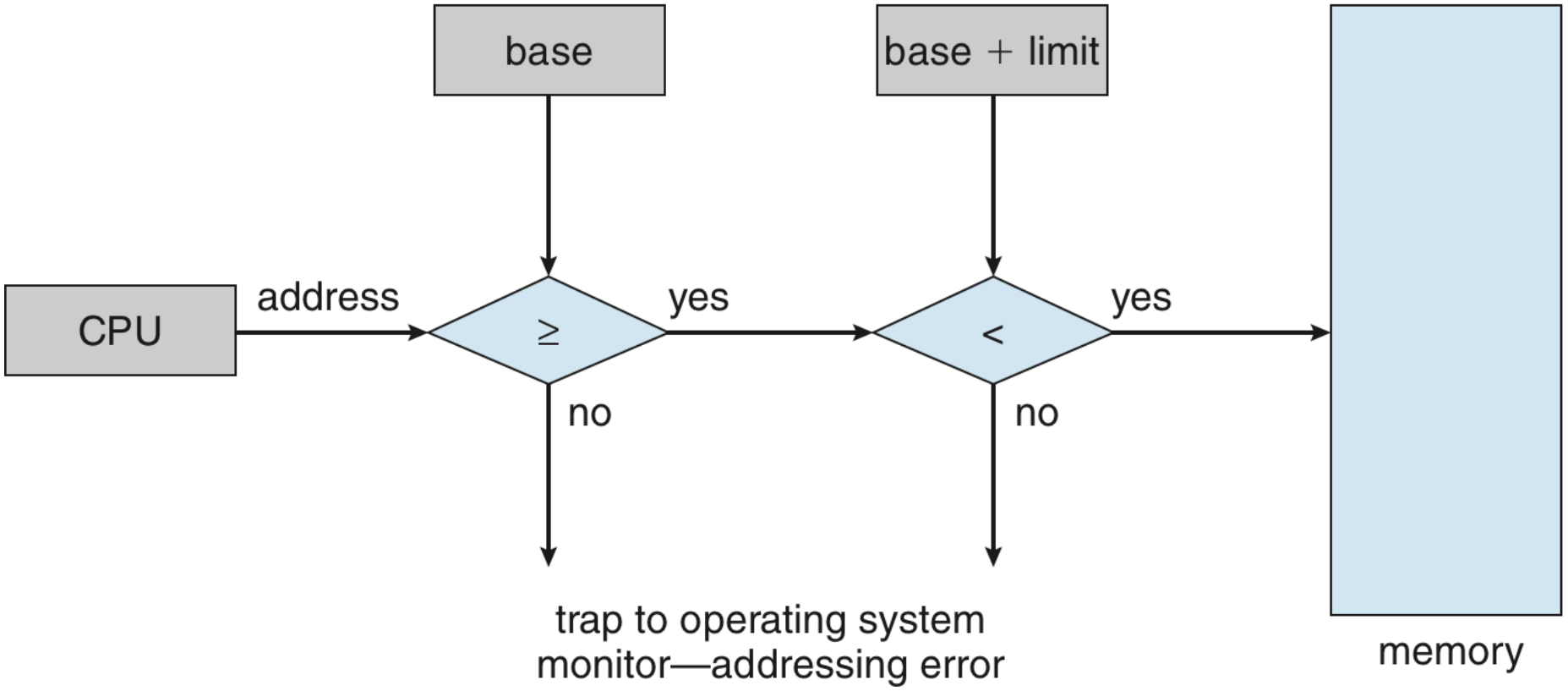
# Memory Management Requirement 2: Protection



- Each process has a separate memory space
  - security ?
- provide security using 2 registers
  - base register
  - limit register
- The base register holds the smallest legal physical memory address; the limit register specifies the size of the range.
- Need a hardware for address comparison



# Hardware Support for Relocation and Limit Registers



# Points to be noted



- Illegal access results in trap to OS ➔ fatal error
- Special privileged instructions are used to load base and limit registers
- The operating system has unrestricted access to both operating system and users' memory.

# Memory Management Requirement 3: 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 Requirement4: Logical Organization



- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
  - Different degrees of protection given to modules (read-only, execute-only)
  - Share modules among processes
- Logical Organization
- Segmentation helps here

# Memory Management Requirement5: Physical Organization



- Two Level organization : Main memory and Secondary Memory
  - Main memory : Faster access at relatively high cost, volatile, smaller capacity
  - Secondary memory: Slower, cheaper, nonvolatile, large capacity
- Cannot leave the programmer with the responsibility to manage memory
- Memory Management Unit

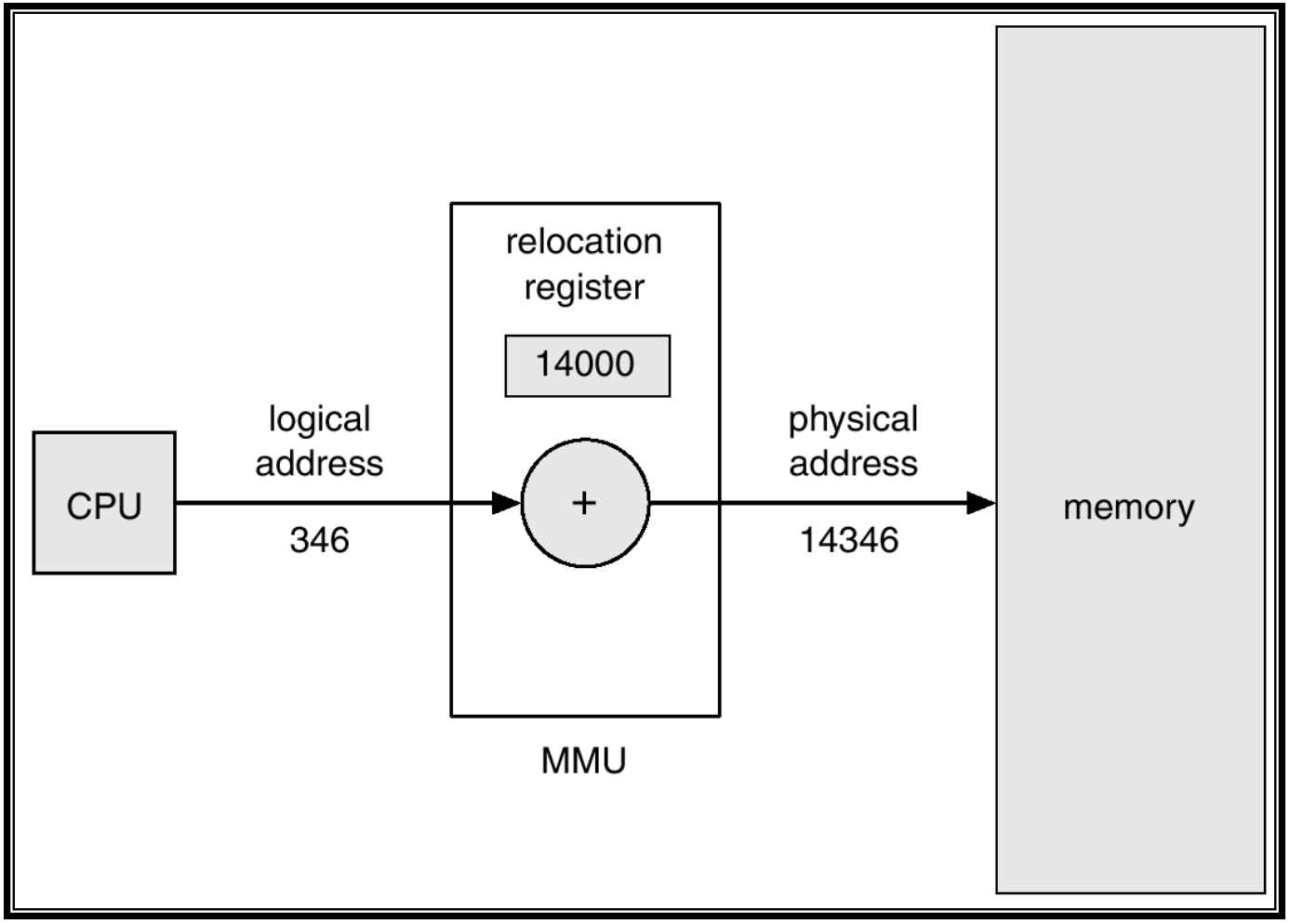
# Logical vs. Physical Address Space

- *Logical address* – generated by the CPU; also referred to as *virtual address*.
- *Physical address* – MAR register- address seen by the memory unit.

# Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
- Two different types of addresses:
  - Logical: range 0 to max.
  - Physical: range R+0 to R+max; where R is a relocation register value.
- Note: The user generates only logical addresses and thinks that the process runs in locations 0 to max.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

# Dynamic relocation using a relocation register



# Memory Management Techniques



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

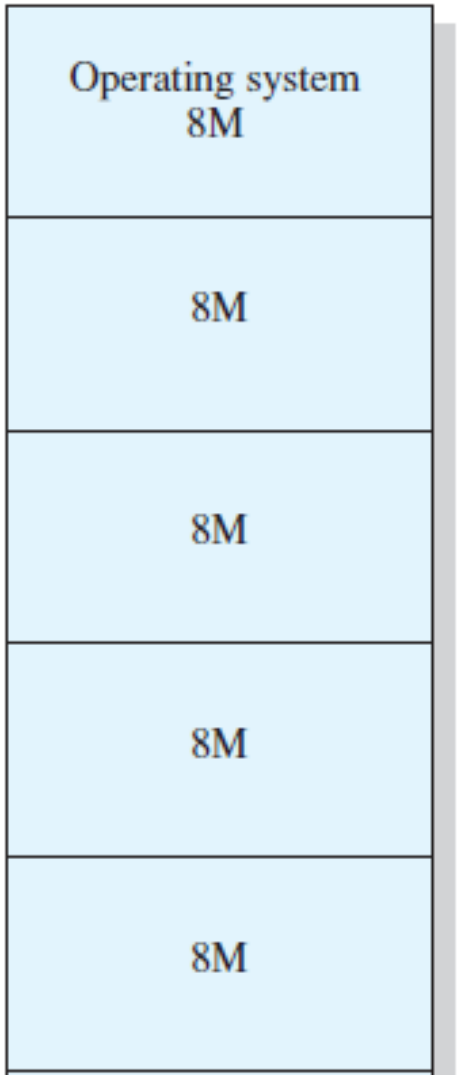
# Fixed partition



- 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 smaller size.
- Simple to implement
- The degree of multiprogramming is bound by the number of partitions
- Little operating system overhead
- EX: IBM OS/360 ➔ MFT : Multiprogramming with Fixed Number of Tasks

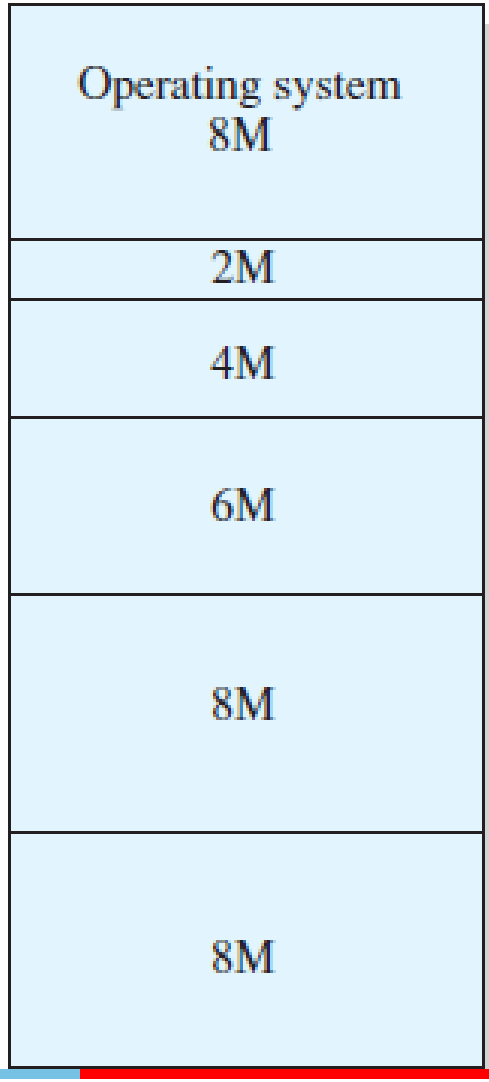
# Fixed partition (Contd…)

- Two difficulties:
  - A program may be too big to fit into a partition :
    - Use overlays
  - Inefficient main memory utilization ➔ Internal Fragmentation
- which process to swap out is based on scheduling

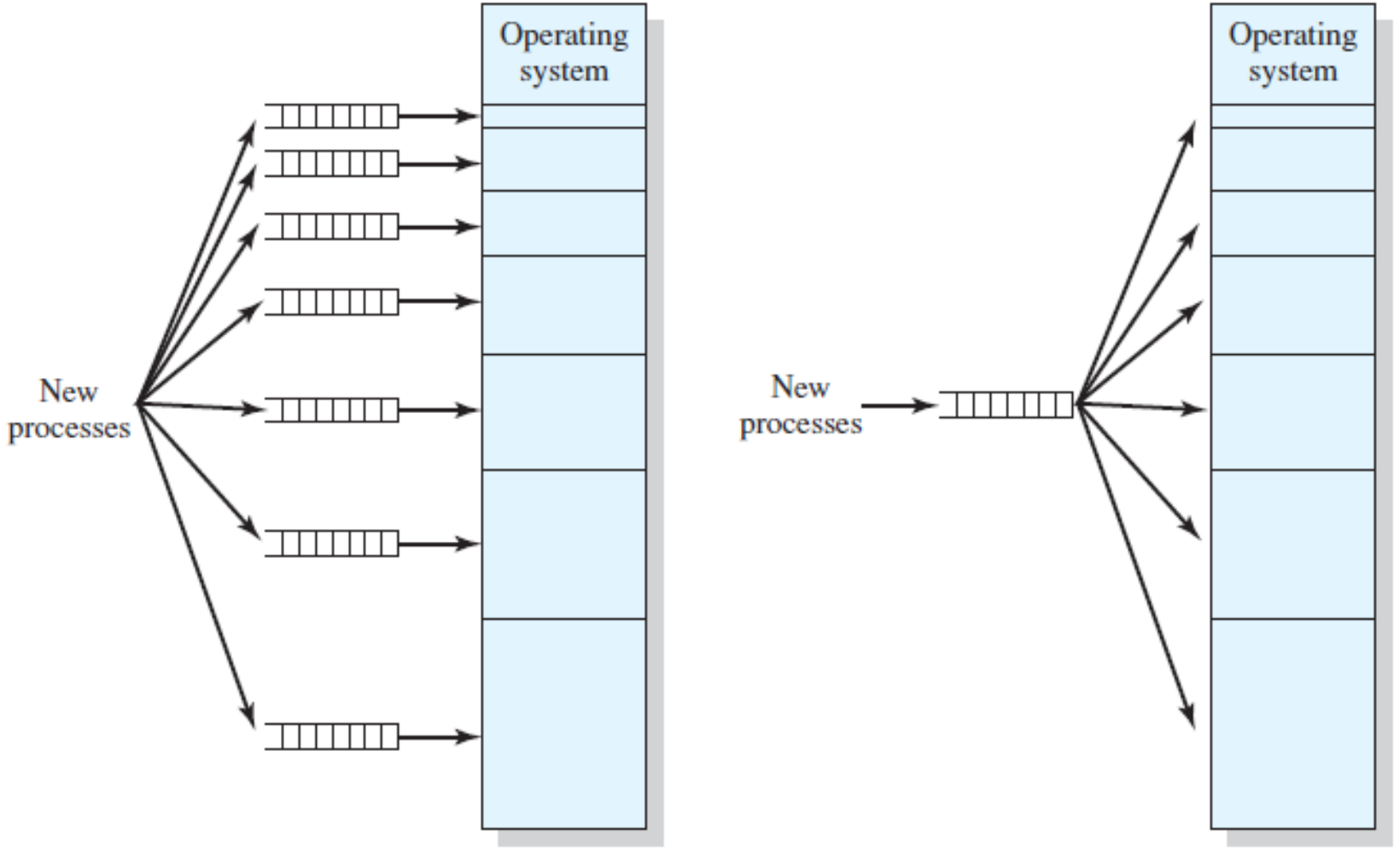


# Fixed partition (Contd…)

- Unequal size partitions
  - provides a degree of flexibility
- Two ways to assign processes to partitions
  - One process queue per partition
  - Single queue



# Contd...




(a) One process queue per partition

(b) Single queue

# Fixed partition (Contd…)

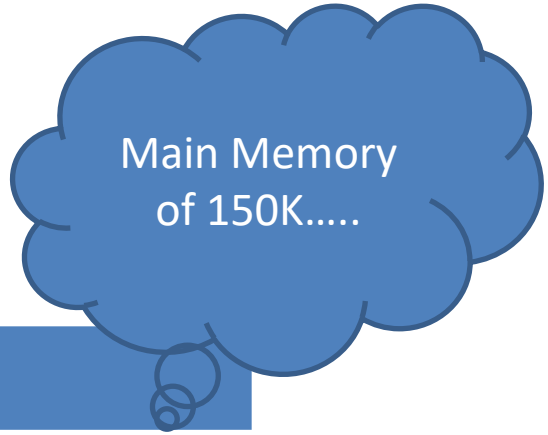


- Unequal size partitions (Contd…)
  - One process queue per partition
    - Lesser internal fragmentation
    - Not optimum solution
  - Single queue
    - Optimum
- Disadvantages of fixed partition:
  - The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.
  - it is an inefficient technique: small jobs will not utilize partition space efficiently

# Example : Assembler



- Two passes : Pass 1 and Pass 2
  - Pass 1: Constructs Symbol Table
  - Pass 2: Generates Machine code




| | Size |
|---|---|
| Pass 1 | 70K |
| Pass 2 | 80K |
| Symbol Table | 25K |
| Common Routines | 25K |
| Total | 200K |

# Overlays

- If a process is larger than the amount of memory, a technique called overlays can be used.
- Overlays is to keep in memory only those instructions and data that are needed at any given time.
- When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.
- Overlays are implemented by user, no special support needed from operating system, programming design of overlay structure is complex.

# Example : Assembler

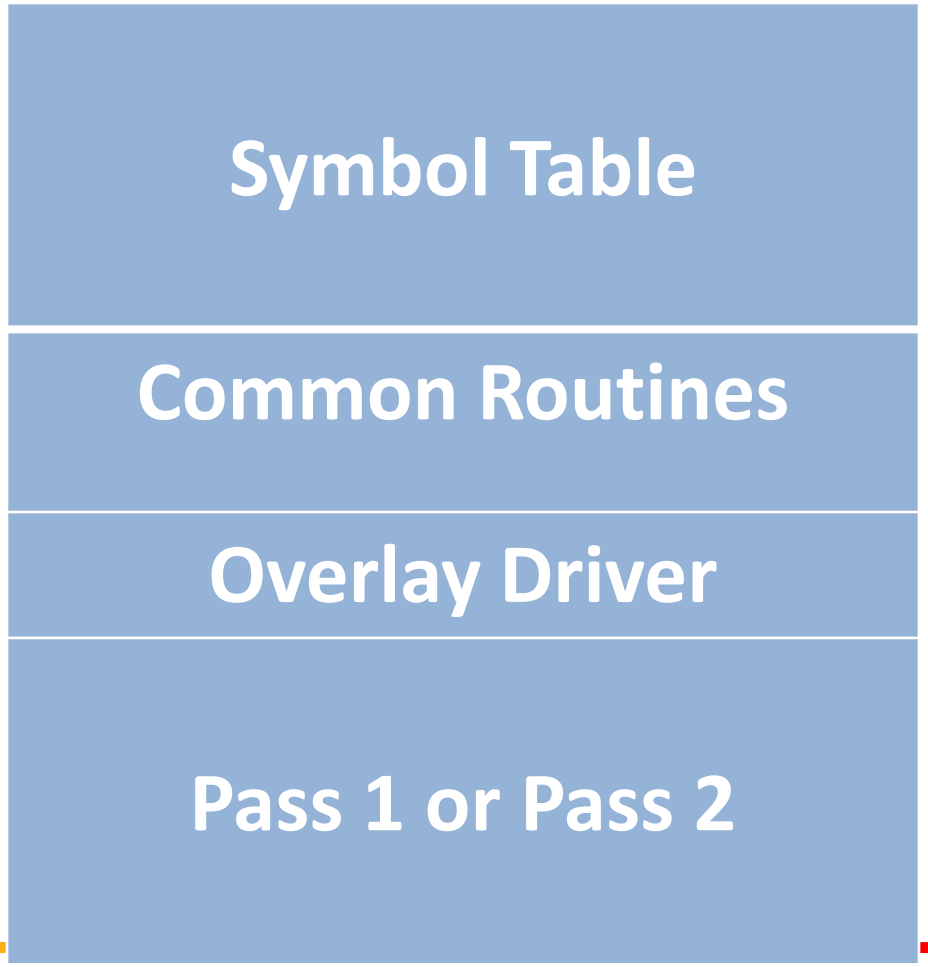


- Overlay A: symbol table, common routines, and Pass1.
- Overlay B: symbol table, common routines, and Pass2.
- Add overlay driver 10k

| | Size |
|---|---|
| Pass 1 | 70K |
| Pass 2 | 80K |
| Symbol Table | 25K |
| Common Routines | 25K |
| Total | 200K |

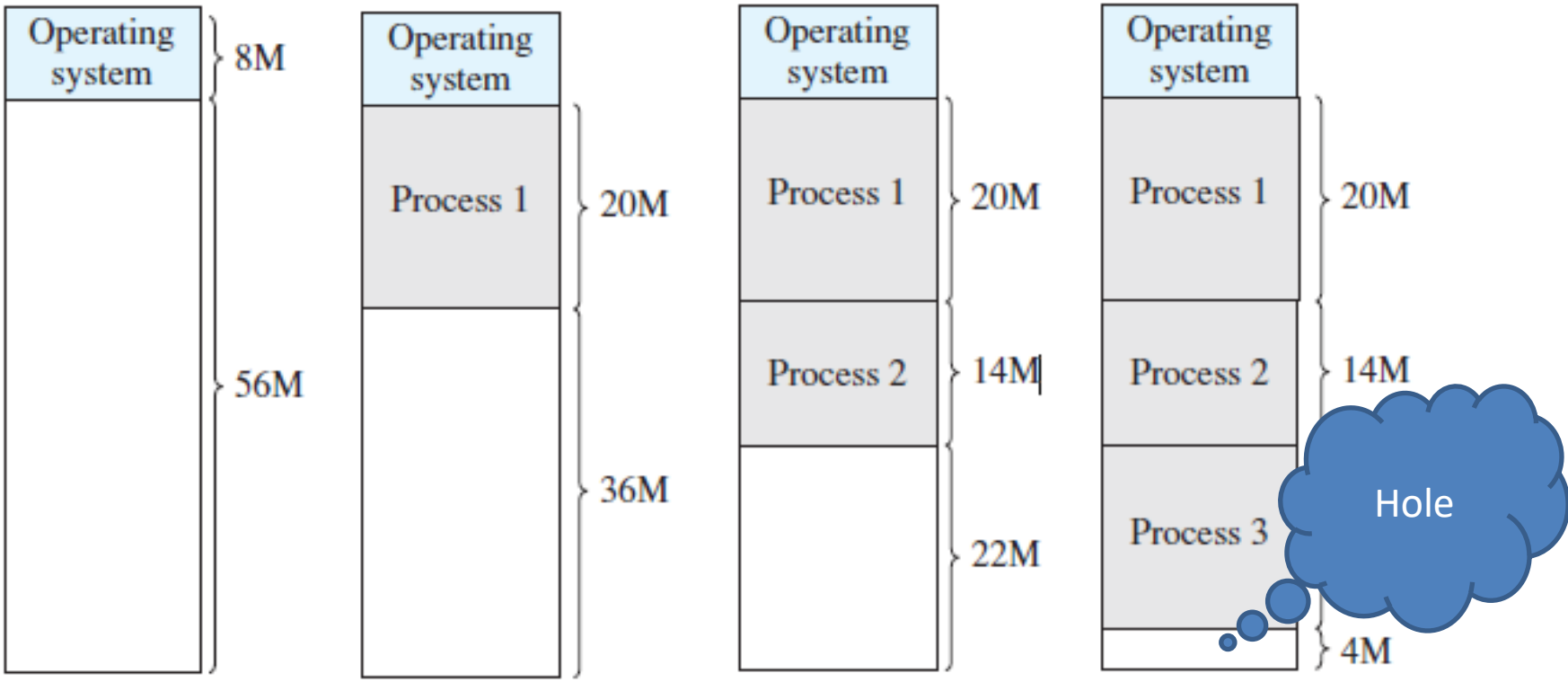
# Contd…

Overlay A requires 130K and Overlay B requires 140K

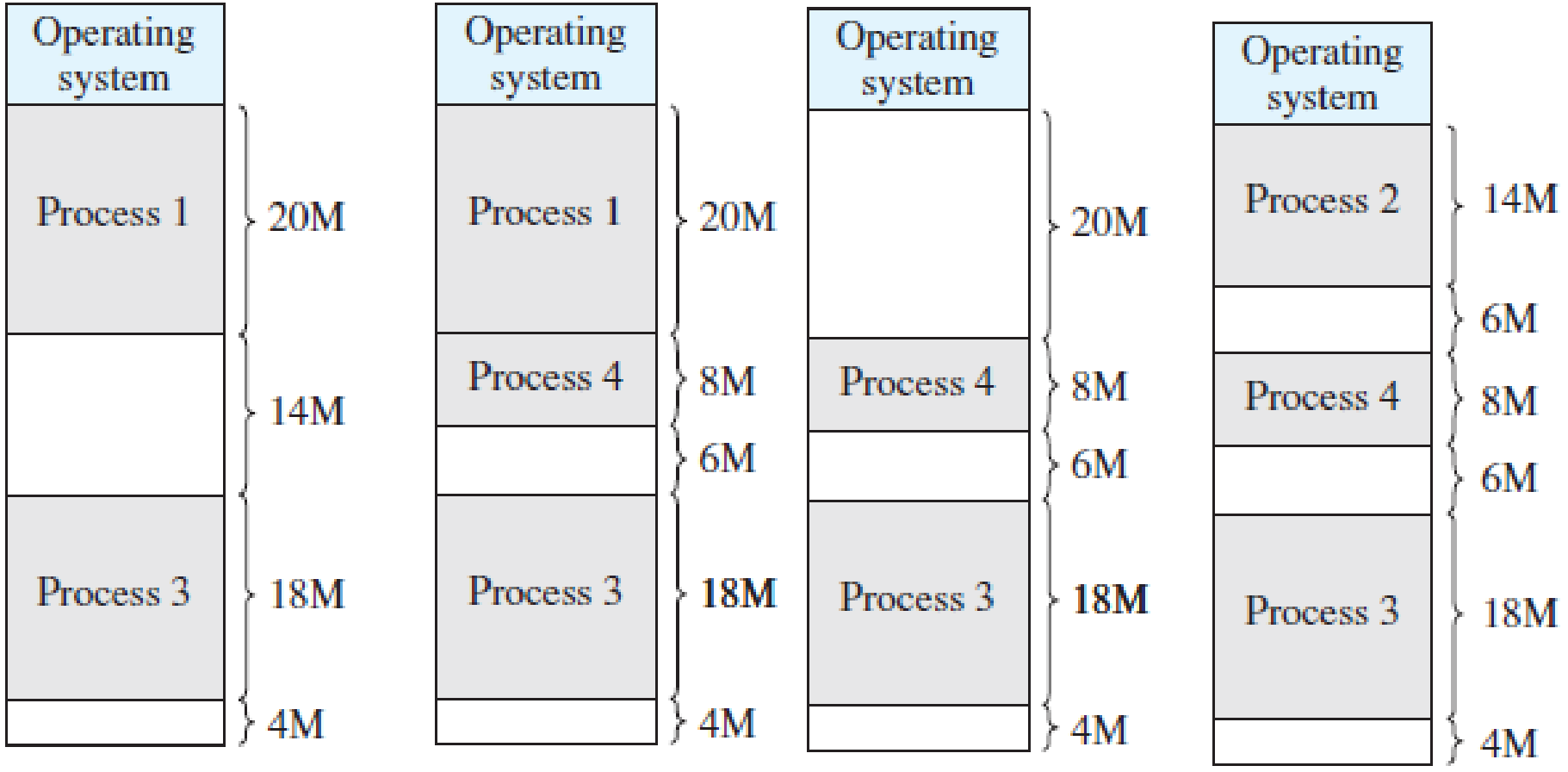


# Dynamic Partitioning or Variable partition allocation

- The partitions are of variable length and number.
- Exact memory allocation

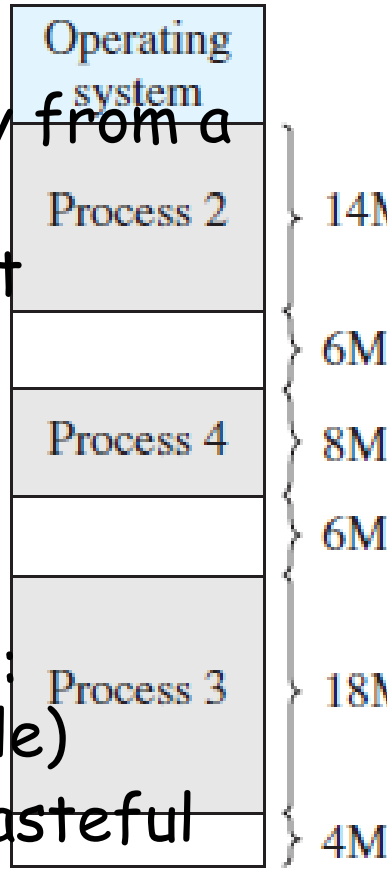


# Contd...


Operating system
Process 1
20M
14M
Process 3
18M
4M
Operating system
Process 1
20M
Process 4
8M
6M
Process 3
18M
4M
Operating system
20M
Process 4
8M
6M
Process 3
18M
4M
Operating system
Process 2
14M
6M
Process 4
8M
6M
Process 3
18M
4M

# Cont.

- *Hole* – block of available memory;
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Holes of various size are scattered throughout memory.
- External Fragmentation
- Compaction: technique to overcome external fragmentation
- Operating system maintains information about: a) allocated partitions b) free partitions (hole)
- Compaction is a time consuming process and wasteful of processor time



# Contd…

- When no available block of memory (hole) is large enough to hold process, the O.S. waits until a large block is available.
- In general, there is at any time a set of holes, of various sizes, scattered throughout memory.
- When a process arrives and needs memory, search for a hole that is large enough
- If the hole is too large, it is split into two:
  - One part is allocated to the arriving process.
  - The other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, merge these adjacent holes to form one larger hole.

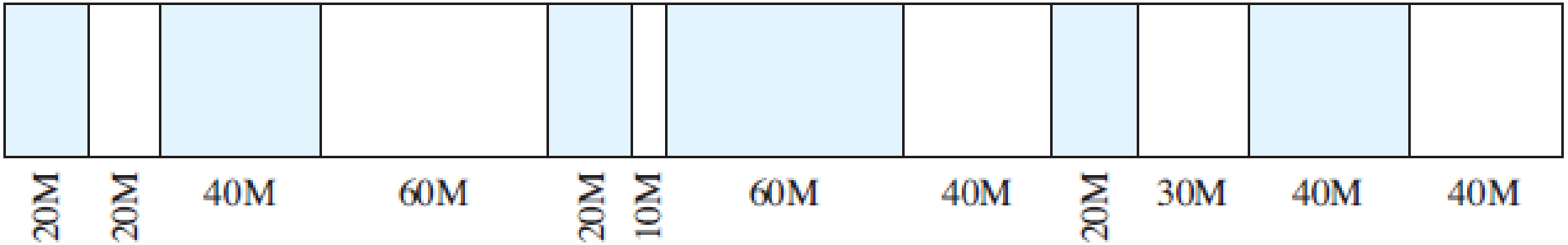
# Dynamic Storage-Allocation



- Four schemes:
  - **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.
  - **Next-fit**: begins to scan memory from the location of the last placement, and chooses the next available block that is large enough.
  - **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
- Which one is better?

# Problem

A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:

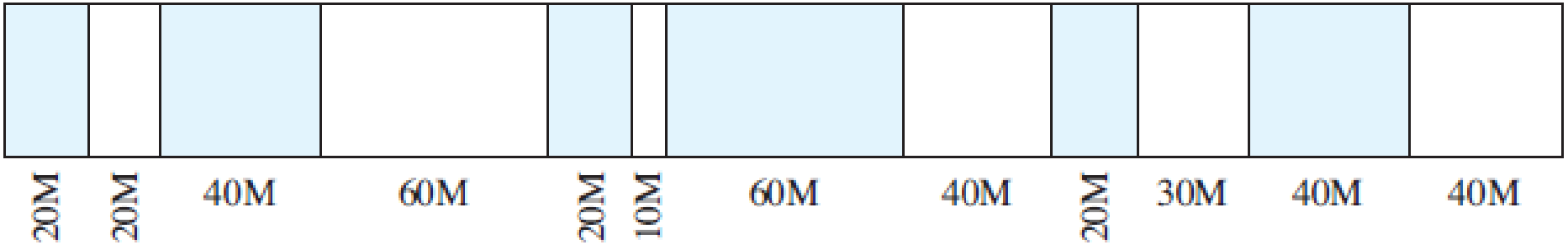



The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the following placement algorithms:

a. First-fit
b. Best-fit
c. Next-fit. Assume the most recently added block is at the beginning of memory.
d. Worst-fit

# Problem

A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:

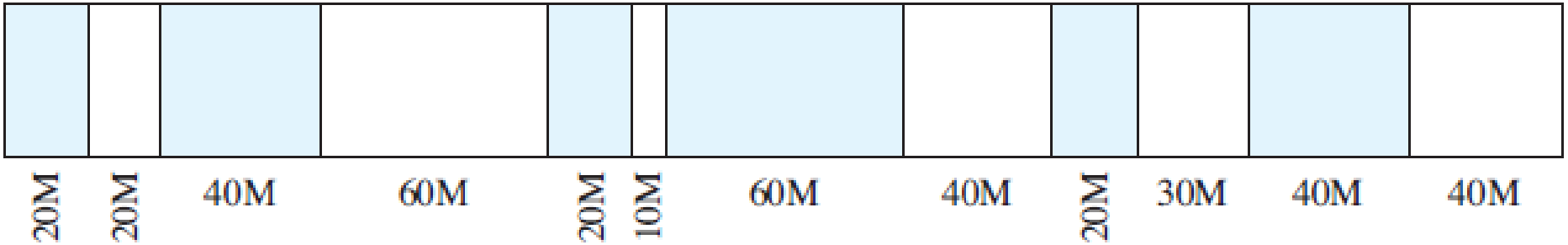



The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the following placement algorithms:

a. First-fit
b. Best-fit
c. Next-fit. Assume the most recently added block is at the beginning of memory.
d. Worst-fit

# Problem

A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:

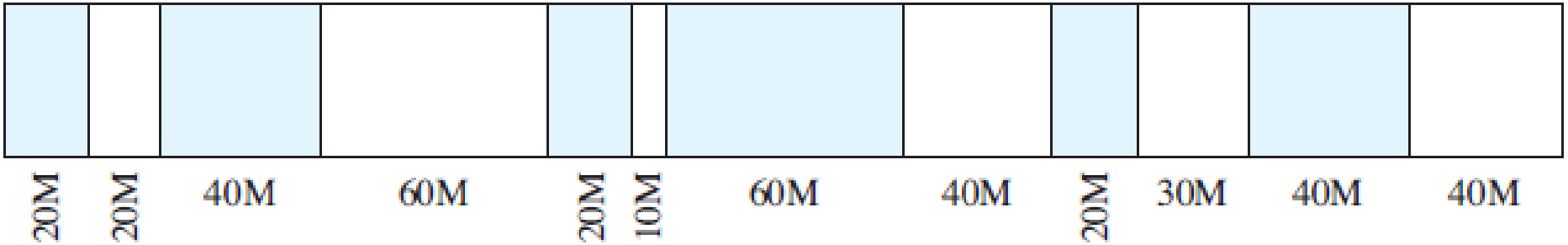



The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the following placement algorithms:

a. First-fit
b. Best-fit
c. Next-fit. Assume the most recently added block is at the beginning of memory.
d. Worst-fit

# Problem

A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:




The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the following placement algorithms:

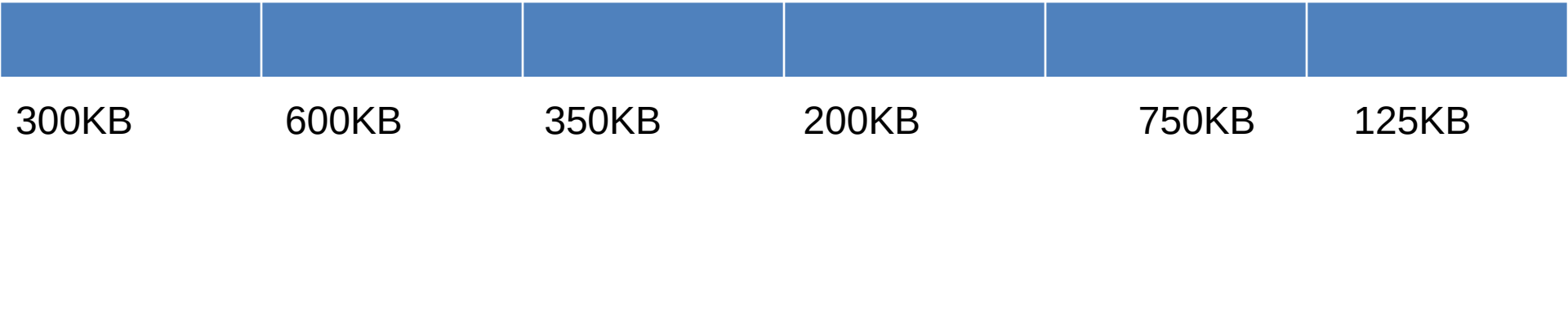
a. First-fit
b. Best-fit
c. Next-fit. Assume the most recently added block is at the beginning of memory.
d. Worst-fit

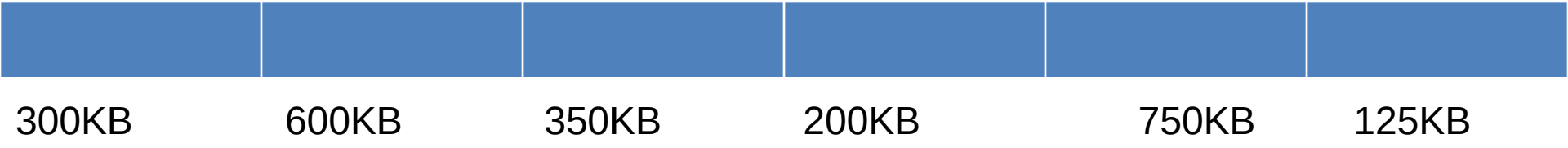
# Problem

Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?


# Problem (Contd…)

Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?



# Fragmentation

- **fragmentation** is a phenomenon in which storage space is used inefficiently, reducing capacity and often performance.
  - **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.
- 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.