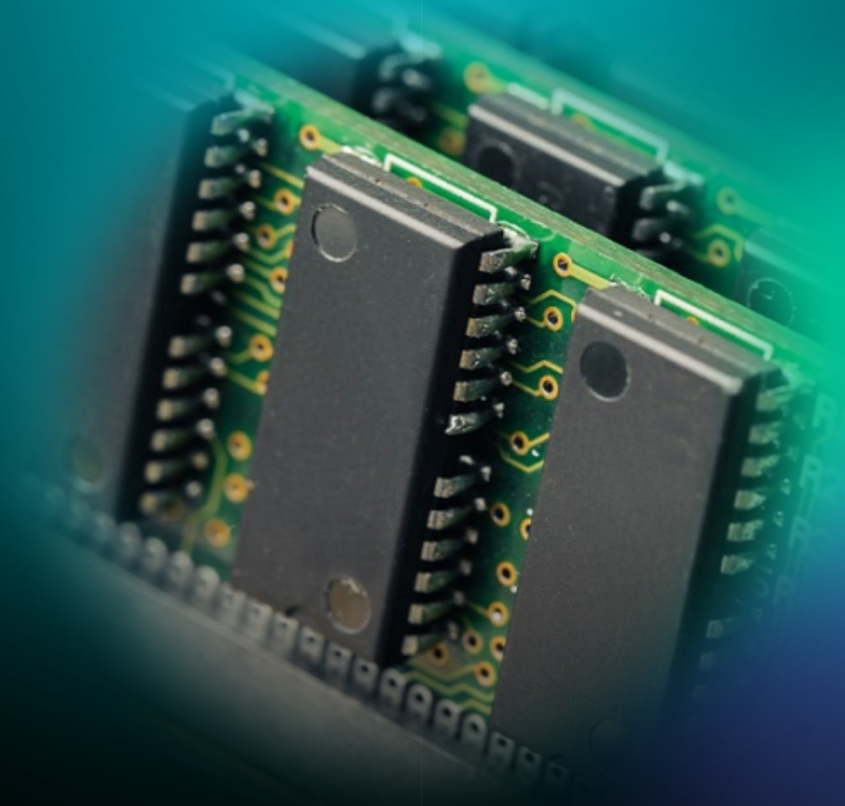
BSCS - 502

CONCEPTS OF OPERATING SYSTEM

PRESENTATION

BY: GROUP 1



CHAPTER # 07

MEMORY MANAGEMENT

CONTENTS

Computer Memory?

Computer's memory refer to as **storage areas** where information such as data and programs are stored for immediate use in computer.

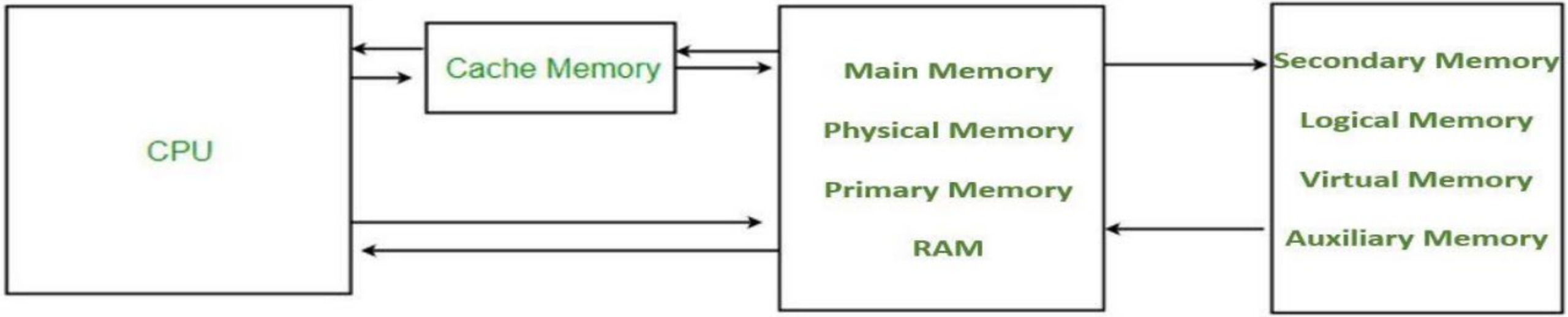
PRIMARY MEMORY: also known as main memory, used to store data and programs or instructions during computer operations.
Example: ROM, RAM etc.

SECONDARY MEMORY: used to store data and programs that can be accessed or retrieved even after the computer is turned off.
Example: Hard Disk, CDs, DVDs etc.

1. MEMORY

1.1 What is Memory ?

CACHE MEMORY: is a type of high speed semi-conductor memory that can help run the CPU faster.



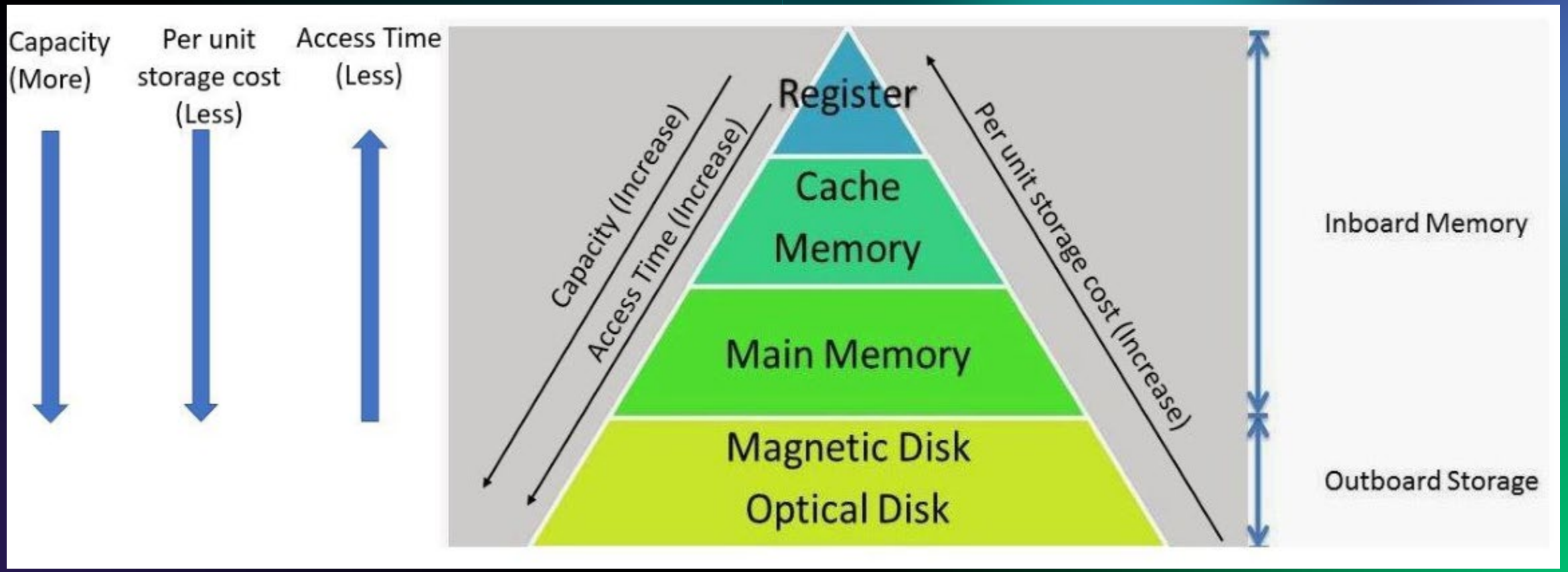
ANALOGY:

1.2 Memory Hierarchy

What we need from a memory?

1. Large capacity
2. Less per unit cost
3. Less access time (fast access)

ANALOGY:



Memory Mangement in Multi-Programmed Systems

➢ What is Memory Management?

In **single-program systems**, memory is split into two parts:

- One for the **Operating System** (OS)
- One for the **Program currently running**

➢ Multiprogramming Systems

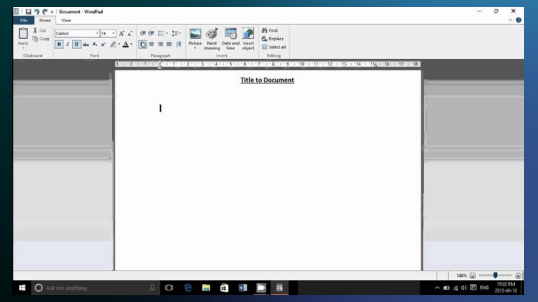
When multiple programs are running, the **user part** of the memory must be **divided** further to allow several programs (or processes) to run at the same time.

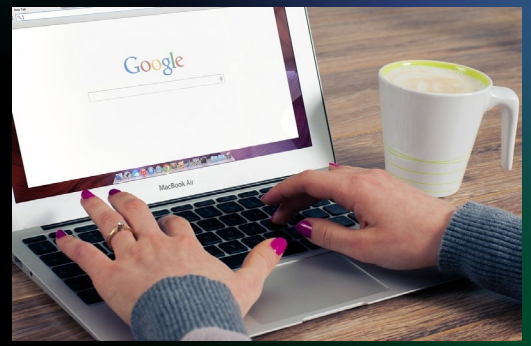
➢ Why is this Important?

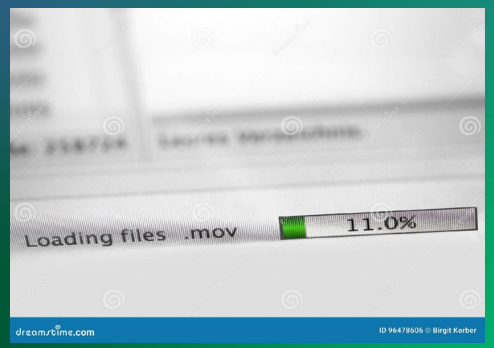
- If there are **few programs in memory**, the system can become **slow**, as programs often wait for I/O operations (e.g., reading a file or printing).
- The **OS allocates memory dynamically** to make sure enough programs are ready to run, so the CPU doesn't sit idle.

Example:

Imagine 4 programs running on your computer:
•**Program 1**: Writing a document
•**Program 2**: Playing music
•**Program 3**: Browsing the web
•**Program 4**: Waiting for a file to load
If memory isn't managed well, some programs may be delayed, and the system could become **slow**.







Relocation in Memory Management

1. What is Relocation?

- In a **multiprogramming system**, many processes share **main memory**.
- Programs can be **moved** (swapped) in and out of memory to optimize CPU use.

2. Why Relocation is Needed?

- Programs **don't know** in advance where they will be placed in memory.
- When swapped out and back in, a program **may not return** to the same location.

3. Challenges with Relocation:

- The **OS** needs to **track memory addresses** for processes (program instructions and data).
- The processor must **translate memory references** to actual physical addresses, based on the program's **current memory location**.

Memory Protection in Multiprogramming Systems

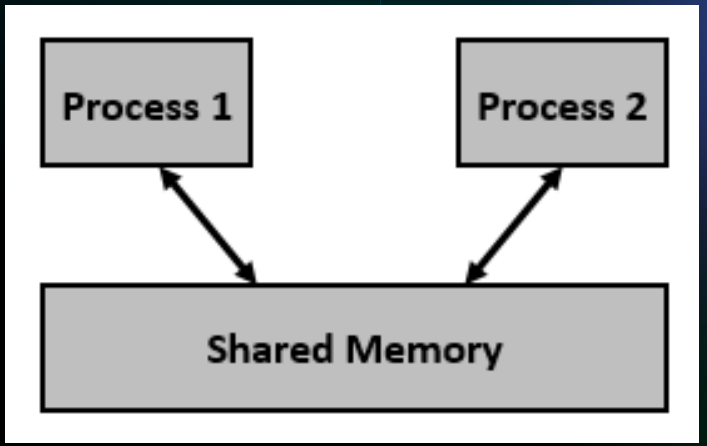
1. **What is Memory Protection?**
   - Each process must be **protected** from interference by other processes.
   - Other programs **cannot read** or **write** to another process's memory **without permission**.
2. **Challenges with Protection:**
   - **Relocation** makes it harder to protect memory, as programs move in memory and absolute addresses can't be checked in advance.
   - Many programs calculate memory addresses **dynamically** at runtime.
3. **How Protection Works:**
   - Every **memory reference** made by a process must be **checked** at runtime to ensure it accesses only its own allocated memory.
   - The **processor hardware** handles this check (not the OS), as it can **abort** unauthorized memory access when it happens.
4. **Examples of Protection:**
   - One process **cannot access** another process's memory or instructions.
   - **OS memory** is also protected from user processes.

Sharing

- Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory.

For example

- If a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its own separate copy.



Logical Organization

Linear Memory Organization:

- Main memory in a computer system is typically organized as a one-dimensional ,address space, consisting of a sequence of bytes or words.

Modular Program Organization:

- Programs are often constructed in modules, where different parts of the program serve specific functions (e.g., code, data).
- Some modules are unmodifiable (such as read-only or execute-only code), while others contain modifiable data that can change during program execution.

OS and Hardware Management:

- The operating system (OS) and hardware must manage memory at the module level, ensuring that different modules (code, read-only data, and writable data) are treated according to their usage. This modular approach allows for better security
- (read/write protection) and efficient memory usage in complex programs.

Advantages

1.Independent Modules and Runtime Resolution

What are Modules?

- Modules are independent parts of a program that can be written, compiled, and tested separately. Each module typically contains a specific functionality, such as handling input/output, performing calculations, or managing a database.

Why Compile Modules Independently?

- Modules can be compiled on their own without needing other parts of the program to be completed. This allows different developers to work on different modules simultaneously. At runtime, the system resolves any references between modules, meaning the modules are "connected" when the program is running.

Example:

- Let's say you are building a software system with two modules:
- Module A (Input Module): Handles user input.
- Module B (Processing Module): Processes data entered by the user.

Step 1:

- **Independent Development**

  Module A can be written to collect input from the user (e.g., name and age).
  Module B can be written separately to calculate or process the data (e.g., verify if the age is above 18).

Step 2:

- **Compilation**

  Both Module A and Module B are compiled independently. At this point, Module A doesn't need to know how Module B is implemented or where it is located in memory.

Step 3:

- **Runtime Resolution**

  At runtime, when the user provides input (Module A), the system will link the output from Module A to Module B, which processes the data. The system automatically resolves references like passing data between the modules, even though they were compiled independently.

2. Different Degrees of Protection for Modules

What is Module Protection?

In a system, protection can be applied to each module to control how other parts of the program interact with it. Common protections include.

- Read-Only: The module's data can only be read, but not changed.
- Execute-Only: The module's functions can be executed, but its internal data or code cannot be read or modified.

Why is Protection Important?

- This ensures data security and stability by preventing unauthorized or unintended access to sensitive parts of the program. Different modules can have different protection levels based on their role in the program.

Example:

- Imagine a system with two modules:
- Module A (Configuration Module): Stores critical configuration settings.
- Module B (Processing Module): Performs operations based on the configuration.

Step 1:

- Protection SetupModule A is set to Read-Only:
  Other modules can read the configuration settings but cannot modify them.This prevents accidental or malicious changes to crucial settings.
- Module B is set to Execute-Only:
  Other modules can execute functions in Module B (like a calculation function), but cannot access or change its internal code or data.

Step 2:

- Runtime Interaction
  Module B reads configuration data from Module A but is not allowed to alter it.
  Another module can call a function in Module B to perform operations, but cannot view or modify Module B's internal logic or data.

3. Module Sharing Among Processes

What is Module Sharing?

- Module sharing allows different processes (programs) to use the same module at the same time.
- This means multiple processes can access and use a module without needing separate copies of it, saving memory and resources.

Why Share Modules?

- Sharing modules at the module level aligns with how users think about problems.
- Instead of duplicating functionality in each process, a single module can be shared across multiple processes, simplifying design and saving resources.

Example:

- Imagine a system where two processes need to perform logging:
- Module A (Logging Module): Handles writing log messages to a file.

Step 1:

- Independent Processes

  Process 1 (User Input Program): Reads user input and needs to log each action.Process 2 (Data Processing Program): Processes data and logs events during processing.

Step 2:

- Sharing the Logging Module

  Both processes can share Module A (Logging Module) instead of each having its own logging system. When either process logs a message, it uses the same module to write the message to a log file.

Physical Organization

- Computer memory is organized into at least two levels, referred to as main memory and secondary memory. Main memory provides fast access at relatively high cost. In addition, main memory is volatile; that is, it does not provide permanent storage. Secondary memory is slower and cheaper than main memory and is usually not volatile. Thus secondary memory of large capacity can be provided for long-term storage of programs and data, while a smaller main memory holds programs and data currently in use.

  - In this two-level scheme, the organization of the flow of information between main and secondary memory is a major system concern. The responsibility for this flow could be assigned to the individual programmer, but this is impractical and undesirable for two reasons:

Reasons:

1. The main memory available for a program plus its data may be insufficient. In that case, the programmer must engage in a practice known as overlaying, in which the program and data are organized in such a way that various modules can be assigned the same region of memory, with a main program responsible for switching the modules in and out as needed. Even with the aid of compiler tools, overlay programming wastes programmer time.

2. In a multiprogramming environment, the programmer does not know at the time of coding how much space will be available or where that space will be.

Fixed Partitioning in Memory Management

KEY CONCEPTS AND CHALLENGES

BASED ON WILLIAM STALLINGS' OPERATING SYSTEMS, 7TH EDITION

CONTENT

- MEMORY PARTITIONING
- Fixed Partitioning
- Equal-Size Partitions
- Unequal-Size Partitions
- Placement Algorithms for Fixed Partitioning
- Scheduling Approaches
  - Multiple Queues
  - Single Queue
- Limitations
- Conclusion

MEMORY PARTITIONING

- Memory partitioning is a technique used in operating systems to divide the main memory (RAM) into sections or partitions for the allocation of processes. It ensures that multiple programs can run concurrently by assigning each process to a specific partition of memory. There are two main types:

1. **Fixed Partitioning**: Memory is divided into fixed-size regions. Each partition can hold one process, but this method can lead to **internal fragmentation**, where memory is wasted because the partition size is larger than the process.
2. **Dynamic Partitioning**: Partitions are created dynamically based on the size of the processes. This method reduces internal fragmentation but can result in **external fragmentation**, where small gaps of unused memory appear between allocated spaces.

Fixed Partitioning

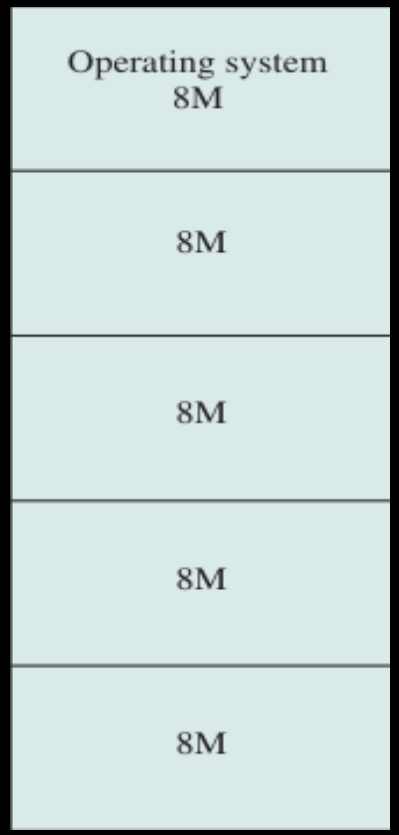
- Memory management involves allocating memory space to processes.
- Fixed partitioning is a simple scheme where memory is divided into fixed-size regions (partitions).
- The operating system (OS) occupies a fixed portion of main memory, and the rest is available for multiple processes.
- Each partition can hold one process, and swapping between partitions occurs when necessary.

Fixed Partitioning

- Description:
  - 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.
- Strengths:
  - Simple to implement; little operating system overhead.
- Weaknesses:
  - Inefficient use of memory due to internal fragmentation
  - maximum number of active processes is fixed.

Equal-Size Partitions

- In this method, memory is divided into equal-sized partitions.
- A process can be loaded into any available partition as long as its size fits within the partition.
- **Example**: If all partitions are 8 MB, any process smaller than or equal to 8 MB can be loaded.
- **Key Point**: If all partitions are full, the OS may swap a process out of one of the partitions to load a new process.



Equal-Size Partitions Issues

- **Issues**:
- A program may be too large to fit into a partition, requiring the use of overlays.
- Internal fragmentation occurs, where small programs waste memory because they occupy entire partitions.

Challenges with Equal-Size Partitions

- **Internal Fragmentation**: Any program, regardless of its size, occupies the full partition it is loaded into.
- **Overlay Requirement**: If a program is too large to fit, the programmer must split it into smaller parts (overlays), with only some parts loaded at a time.
- **Inefficiency**: Memory utilization is extremely inefficient, especially when small programs occupy large partitions.
- **Example**: A 2 MB program may occupy an 8 MB partition, leaving 6 MB of unused memory (internal fragmentation).

Internal Fragmentation

- **Definition**: Internal fragmentation occurs when allocated memory blocks are larger than the required size of a process, leading to unused space within the allocated partition.
- **Cause**: This typically happens in **fixed partitioning** schemes where partitions are predefined and processes may not fully utilize the allocated space.
- **Example**: A 2 MB process placed in an 8 MB partition leaves 6 MB of memory wasted.

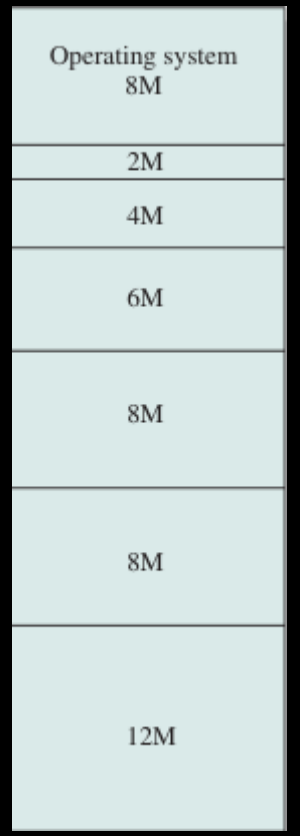
Internal Fragmentation

- **Impact:**
- Wasted memory space reduces **overall memory utilization**.
- Leads to inefficient memory management, especially with small processes occupying large partitions.
- **Solutions:**
- **Dynamic partitioning** reduces internal fragmentation by allocating memory based on process size.
- Alternatively, smaller partitions can minimize waste, but this can increase **external fragmentation**.

Unequal-Size Partitions

- Unequal-size partitions are designed to reduce the inefficiency of equal-size partitions.
- **Key Benefit**: Larger programs can be accommodated without overlays, and smaller programs occupy appropriately sized partitions, reducing internal fragmentation.
- **Example**: Memory may have partitions of 16 MB, 8 MB, and smaller sizes, fitting various program sizes more efficiently.
- **Challenges**:

This method improves memory efficiency but doesn't eliminate all fragmentation.

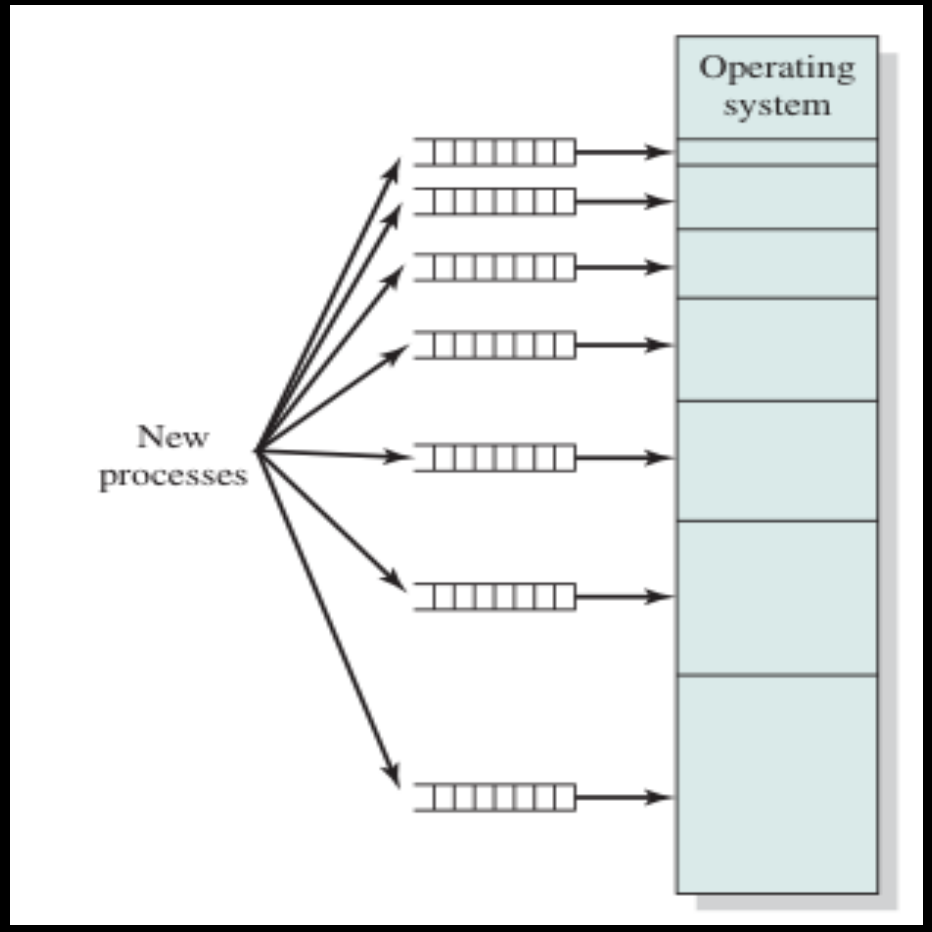


Placement Algorithms for Fixed Partitioning

- In equal-size partitions, placement is simple; any available partition can be used.
- In unequal-size partitions, processes should ideally be placed in the smallest available partition that fits them.

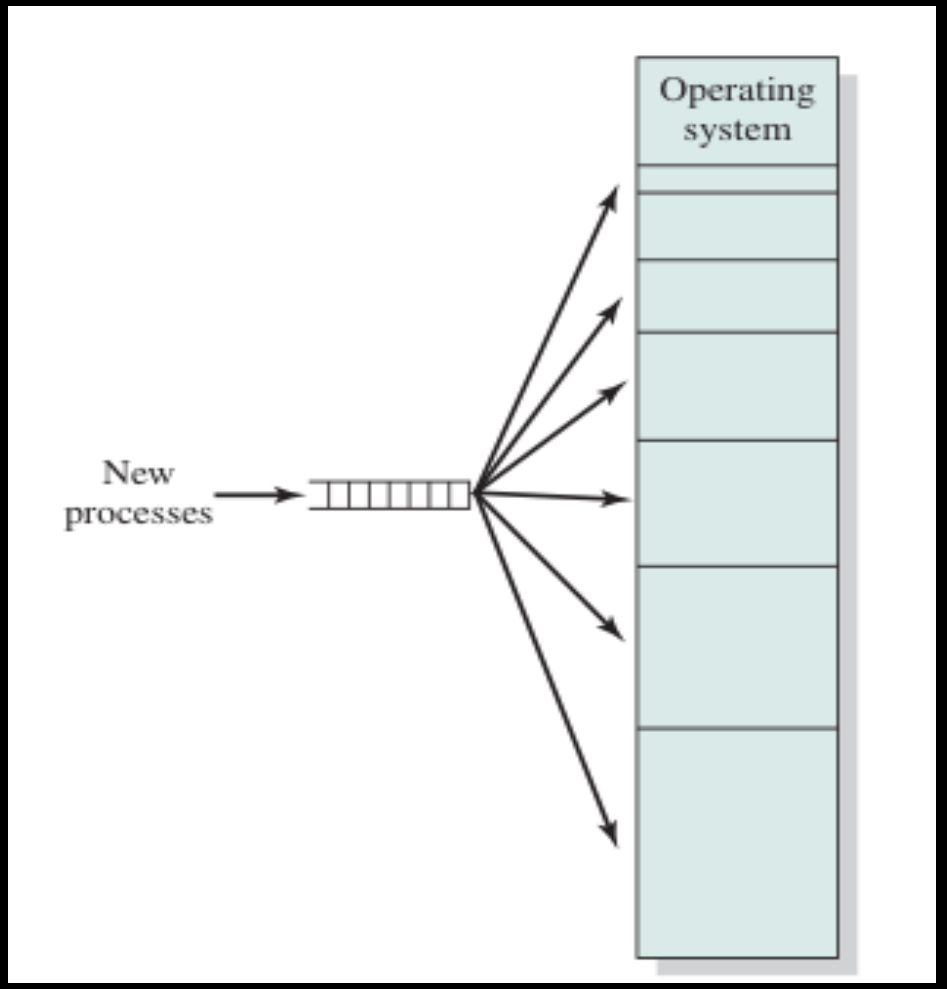
Scheduling Approaches:

- **Multiple Queues**: Separate queues for each partition size, where processes are swapped in and out based on the specific partition they are assigned to.



Scheduling Approaches:

- **Single Queue**: A single queue for all processes, where the smallest available partition is selected for a process. This minimizes wasted memory across the entire system.



Limitations

- **Limitations**:
  - The number of partitions is fixed, limiting the number of active processes in the system.
  - Small programs do not fully utilize partition space, leading to wasted memory.
  - Partition sizes are predefined at system generation time, making it hard to adjust to changing workloads.
- **Obsolescence**: Fixed partitioning is largely obsolete today due to its inefficiency.
- **Example**: One of the few systems that used fixed partitioning was IBM's OS/MFT (Multiprogramming with a Fixed Number of Tasks).

Conclusion:

- Fixed partitioning is simple and requires minimal overhead, but its inefficiencies led to the development of more dynamic memory management techniques.

DYNAMIC PARTITIONING

Dynamic Partitioning

- To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. Again, this approach has been supplanted by more sophisticated memory management techniques. An important operating system that used this technique was IBM's mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks).

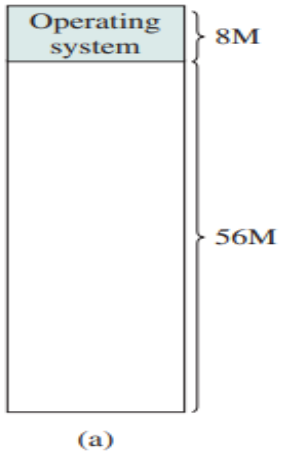
- With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

An example, using 64 Mbytes of main memory

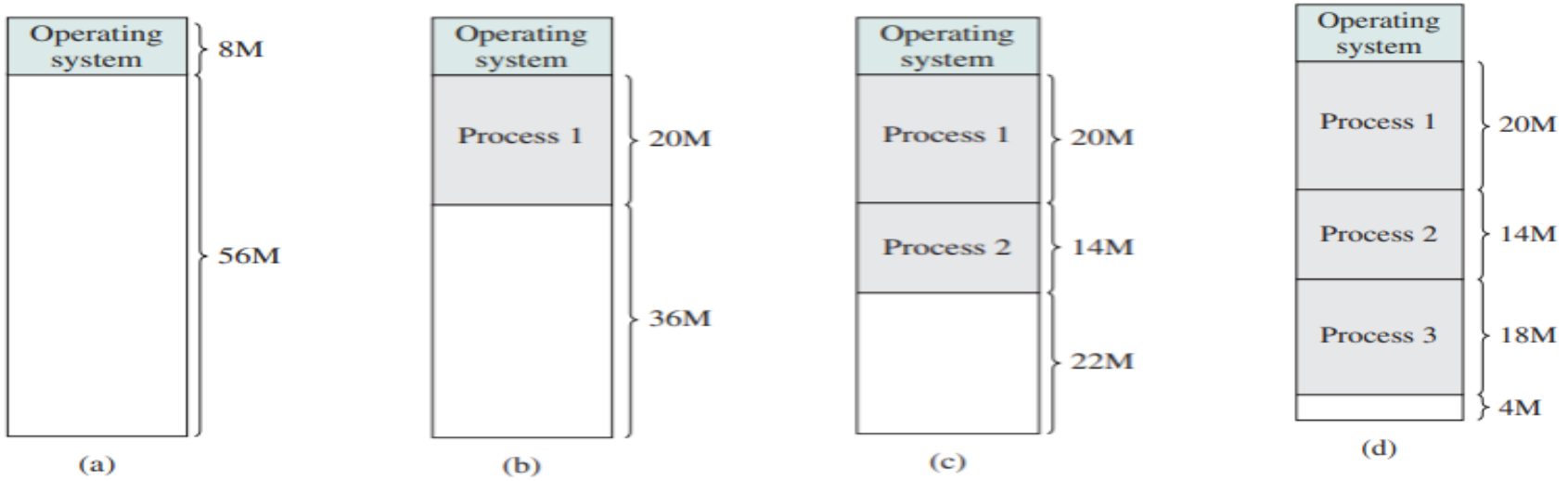
- Let suppose we have 4 processes.

- Process 1 (20 M)
- Process 2 (14 M)
- Process 3 (18 M)
- Process 4 (8 M)

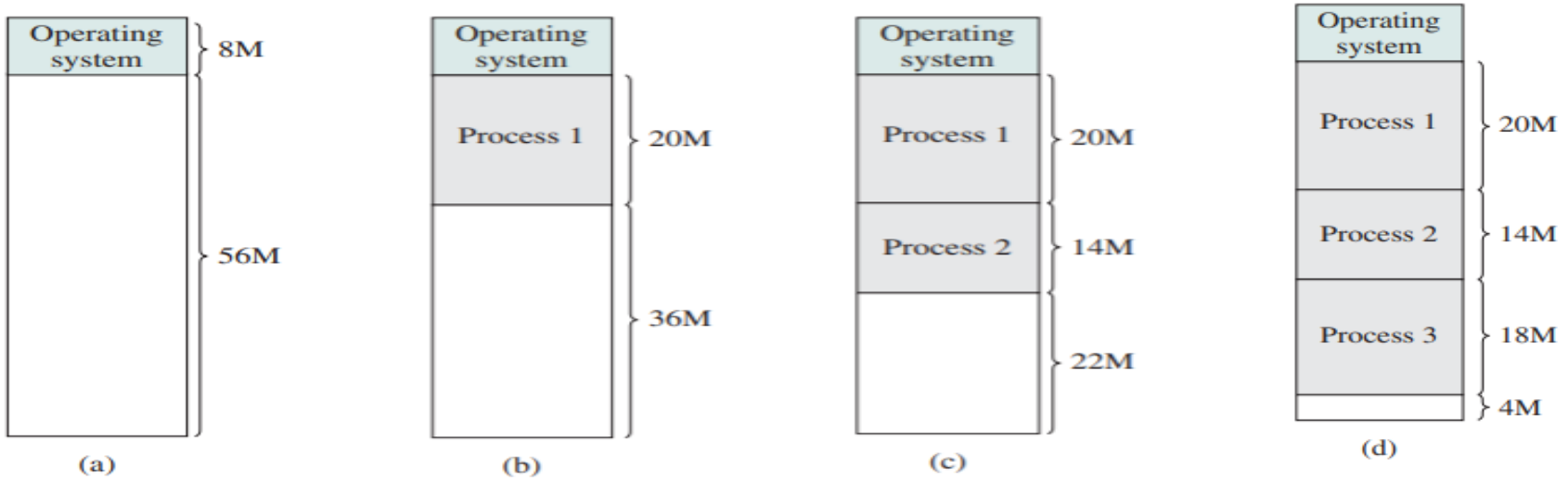
Initially, main memory is empty, except for the OS (a).



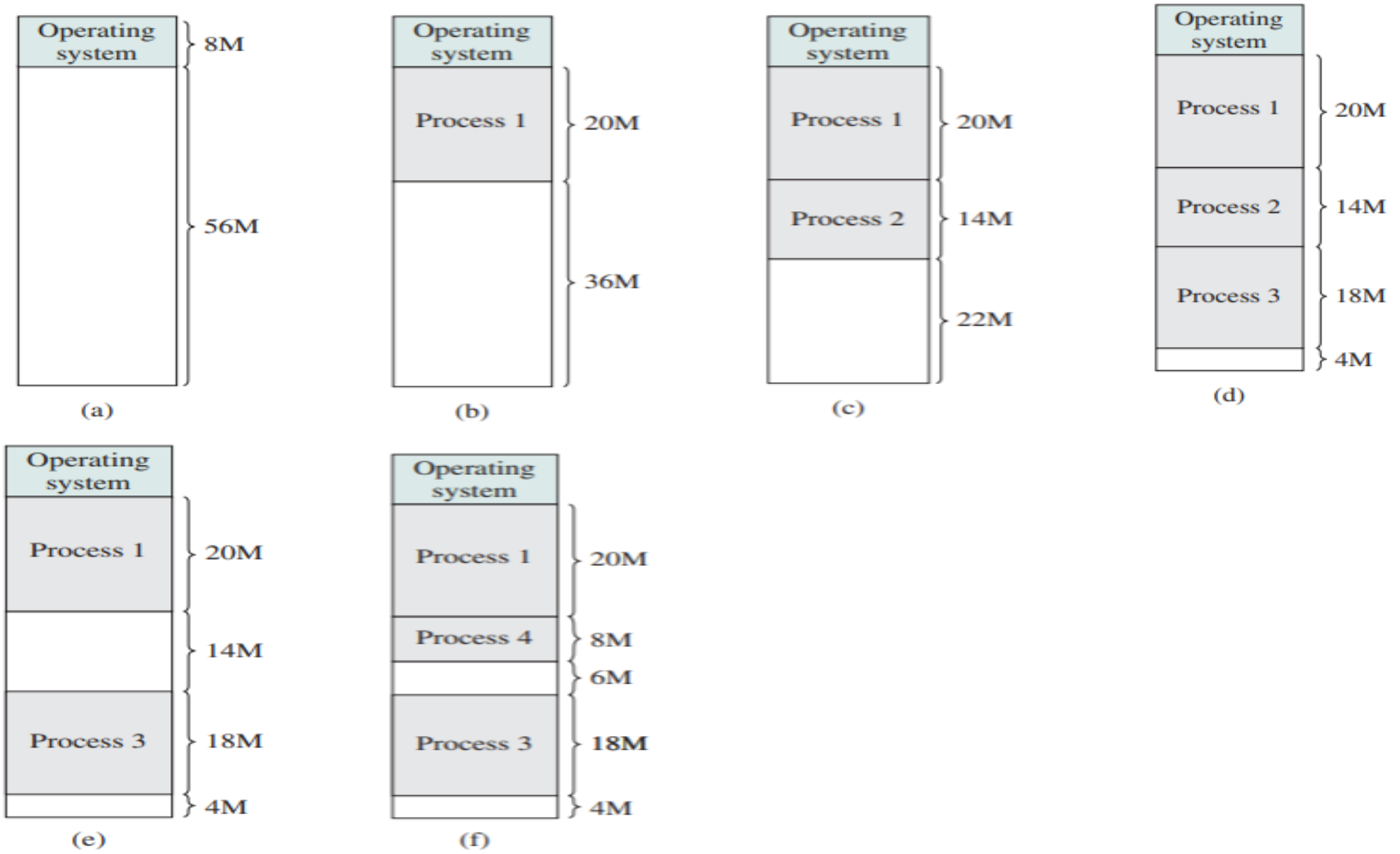
The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process (b, c, d).



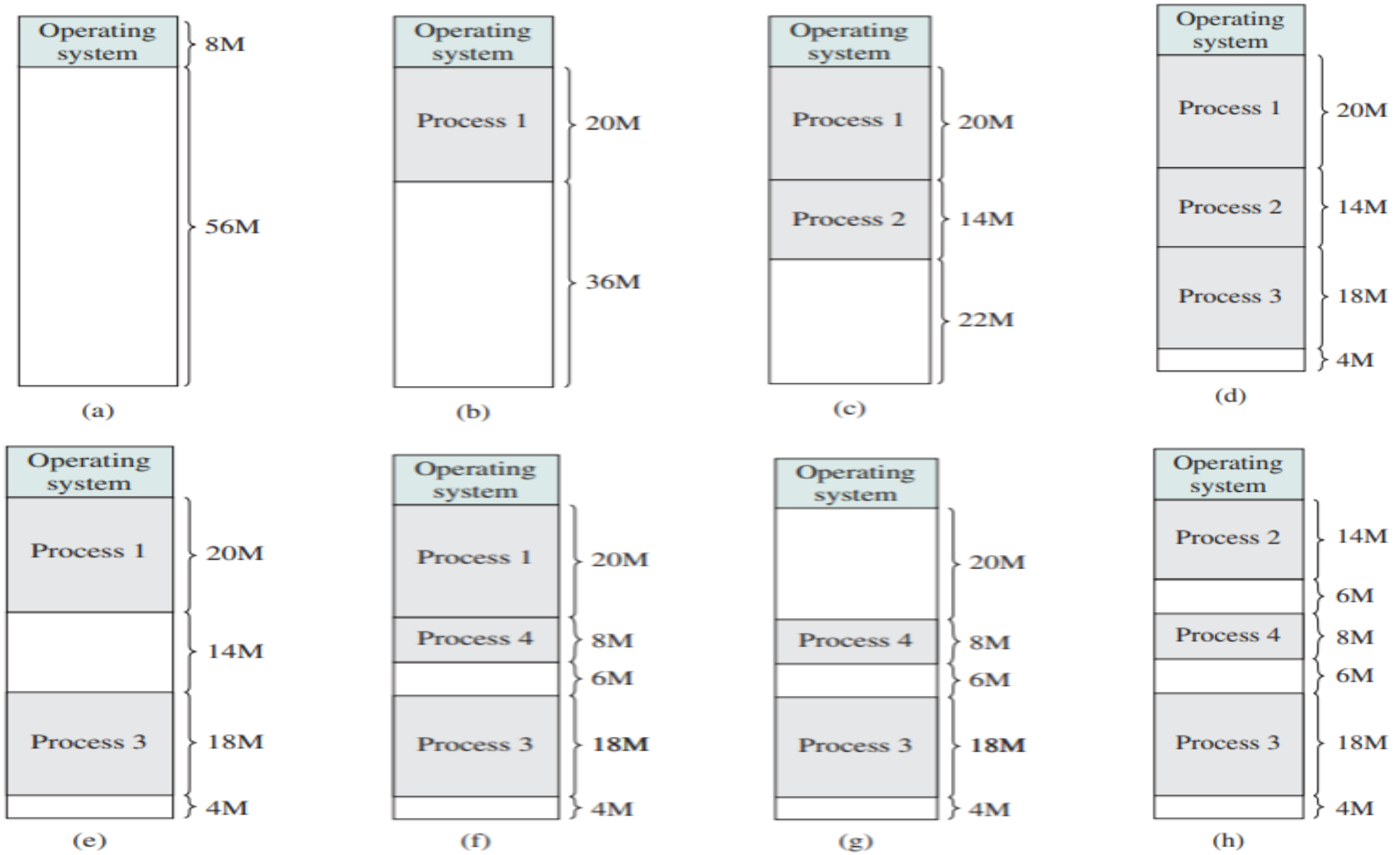
This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready.



The operating system swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created.



Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-Suspend state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out (g) and swaps process 2 back in (h).

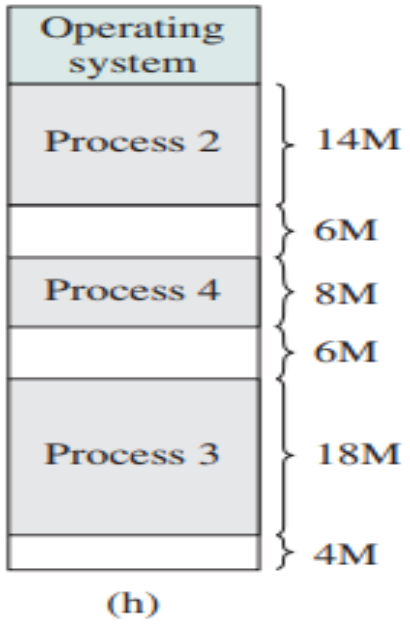


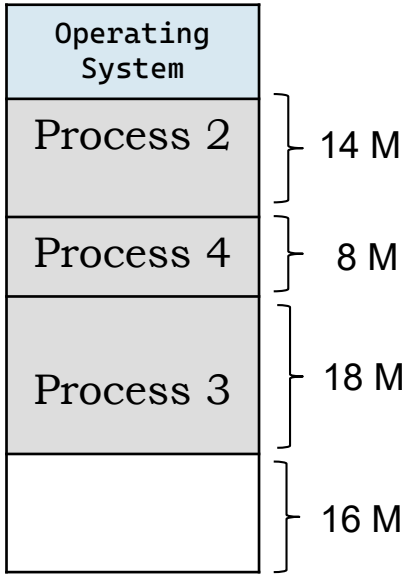
As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as external fragmentation , indicating that the memory that is external to all partitions becomes increasingly fragmented. This is in contrast to internal fragmentation, referred to earlier.

Compaction

- One technique for overcoming external fragmentation is compaction : From time to time, the OS shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure 7.4h , compaction will result in a block of free memory of length 16M. This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time.

Applying Compaction



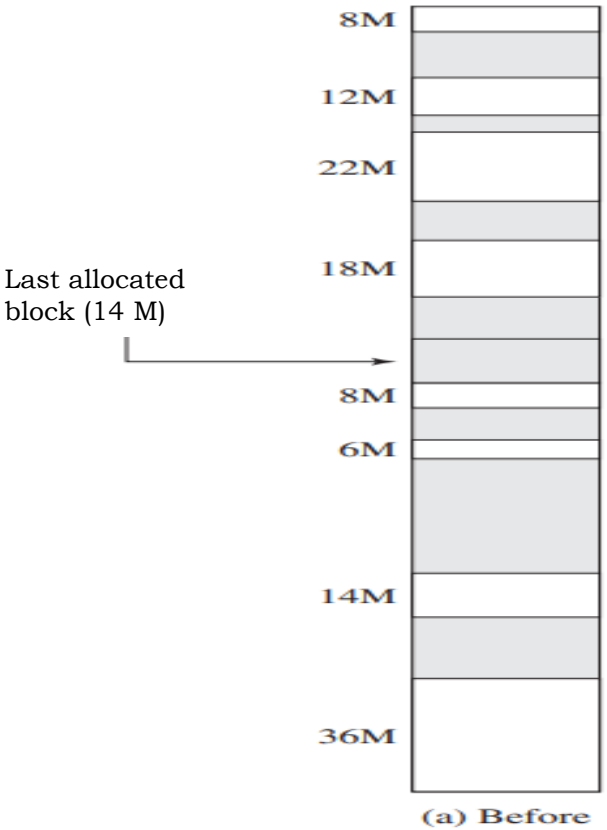


PLACEMENT ALGORITHM

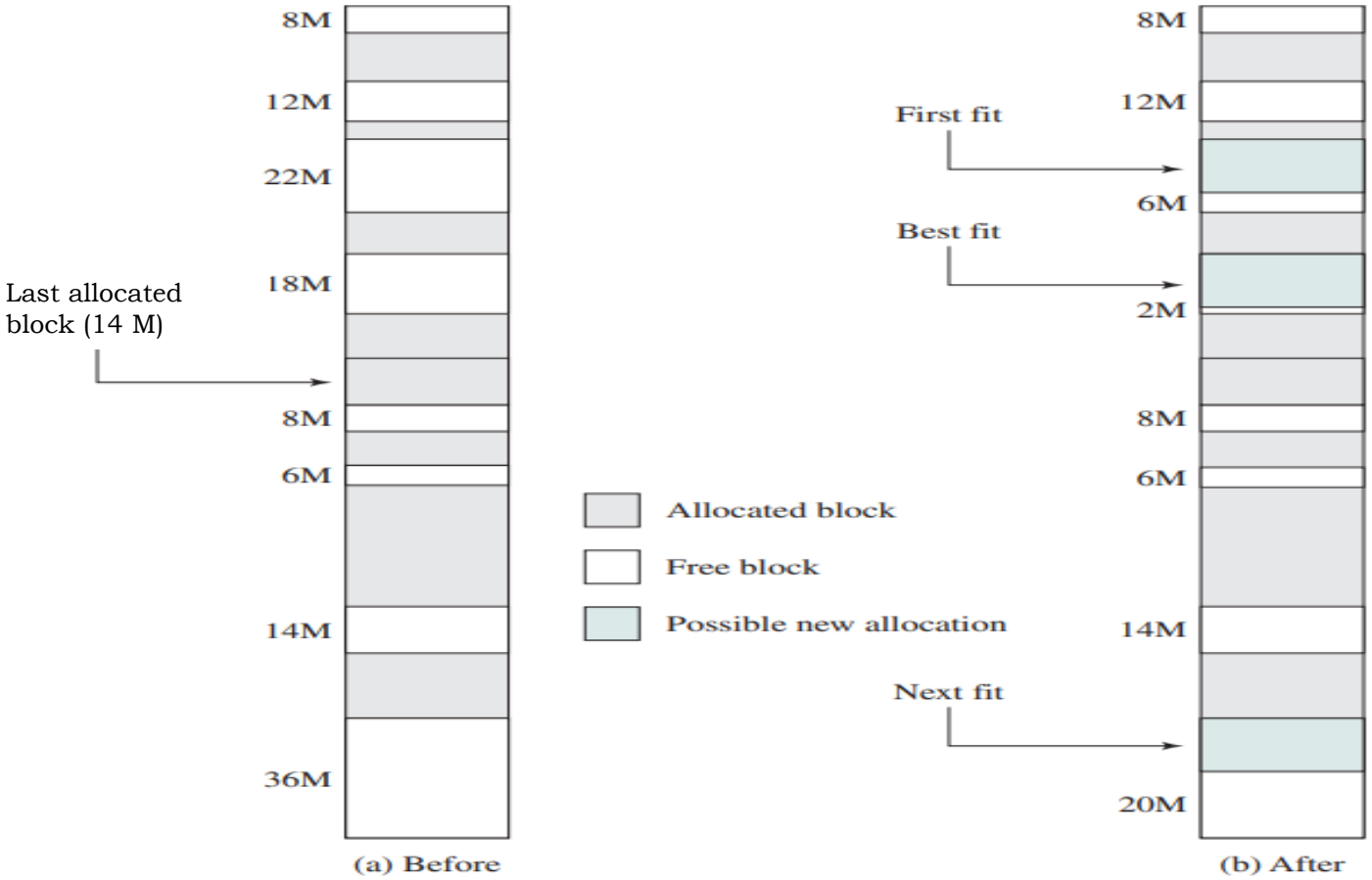
Three placement algorithms that might be considered are best-fit, first-fit, and next-fit.

- **Best-fit** chooses the block that is closest in size to the request.
- **First-fit** begins to scan memory from the beginning and chooses the first available block that is large enough.
- **Next-fit** begins to scan memory from the location of the last placement, and chooses the next available block that is large enough.

Example (a) memory configuration after a number of placement and swapping-out operations. The last block that was used was a 22-Mbyte block from which a 14-Mbyte partition was created.



(b) shows the difference between the best-, first-, and next-fit placement algorithms in satisfying a 16-Mbyte allocation request. Best-fit will search the entire list of available blocks and make use of the 18-Mbyte block, leaving a 2-Mbyte fragment. First-fit results in a 6-Mbyte fragment, and next-fit results in a 20-Mbyte fragment.



Which of these approaches is best will depend on the exact sequence of process swappings that occurs and the size of those processes

- **The first-fit algorithm** is not only the simplest but usually the best and fastest as well.
- **The next-fit algorithm** tends to produce slightly worse results than the first-fit. The next-fit algorithm will more frequently lead to an allocation from a free block at the end of memory. The result is that the largest block of free memory, which usually appears at the end of the memory space, is quickly broken up into small fragments. Thus, compaction may be required more frequently with next-fit.
- **The best-fit algorithm**, despite its name, is usually the worst performer. Because this algorithm looks for the smallest block that will satisfy the requirement, it guarantees that the fragment left behind is as small as possible. Although each memory request always wastes the smallest amount of memory, the result is that main memory is quickly littered by blocks too small to satisfy memory allocation requests. Thus, memory compaction must be done more frequently than with the other algorithms

REPLACEMENT ALGORITHM

- In a multiprogramming system using dynamic partitioning, there will come a time when all of the processes in main memory are in a blocked state and there is insufficient memory, even after compaction, for an additional process. To avoid wasting processor time waiting for an active process to become unblocked, the OS will swap one of the processes out of main memory to make room for a new process or for a process in a Ready-Suspend state. Therefore, the operating system must choose which process to replace.

The End