Operating System (24CSH-242)

Compiled by : Subhayu

Unit-2

Chapter 4 : Deadlocks - Deadlock characterization and conditions for deadlock,deadlock prevention, Deadlock avoidance, safe state, resource allocation graph algorithm, Banker's algorithms-Safety algorithm, Dead lock detection, Recovery from dead lock.

Ch 5 : Memory Management - Address binding, logical versus physical address space, dynamic loading, Swapping, contiguous memory allocation, Fragmentation, Segmentation and Paging.

Ch 6 : Virtual Memory - Virtual Memory Concept, Demand Paging, Page Replacement Algorithms (FIFO, LRU, Optimal, Other Strategies), Thrashing, Cache memory organization , Locality of reference.

Chapter 4: Deadlocks

4.1 Understanding Deadlock

Imagine four friends playing a board game. Each needs two dice to play, but there are only four dice on the table. If each friend picks up one die and waits for a second die (which others are holding), no one can complete their turn. This is exactly what happens in a computer when processes compete for limited resources.

Key Definition

A **Deadlock** occurs when a set of processes are *permanently blocked* because each process is holding a resource and waiting for another resource that is held by some other process.

Real-life analogy:

- **Traffic Intersection:** Cars are stopped at a four-way crossing. Each waits for the other to move first. Nobody moves.
- **Dining Philosophers Problem:** Philosophers need two chopsticks to eat. If each grabs only one and waits, dinner never starts.

4.2 Deadlock Characterization

A deadlock can only occur if the following four conditions hold **simultaneously**. These were first described by Coffman in 1971.

1. **Mutual Exclusion:** Resources cannot be shared; only one process can use a resource at a time. *Example:* A printer cannot print two different documents at the exact same time.
2. **Hold and Wait:** A process holding at least one resource is waiting to acquire additional resources that are currently held by other processes. *Example:* Process A holds a scanner and waits for a printer while Process B holds the printer and waits for the scanner.
3. **No Preemption:** Resources cannot be forcibly taken away from a process; they must be released voluntarily. *Example:* You can't snatch a pen from someone while they are writing.

4. **Circular Wait:** A closed chain of processes exists, where each process waits for a resource held by the next process in the chain. *Example:* A → B → C → A waiting loop.

Remember

All four of these conditions must occur together for a deadlock to happen. Breaking even **one** condition prevents deadlock.

4.3 Deadlock Prevention

Deadlock prevention is like putting traffic rules in place *before* the jam happens.

- **Mutual Exclusion:** Make resources shareable if possible (e.g., read-only files).
- **Hold and Wait:** Force processes to request all resources at once.
- **No Preemption:** Allow resources to be taken away if needed (e.g., save work, preempt CPU).
- **Circular Wait:** Impose a strict ordering of resource acquisition.

Analogy

Think of a restaurant. If customers order all dishes at once (breaking Hold and Wait) or the waiter can take away a plate temporarily (breaking No Preemption), you can avoid a dining deadlock!

4.4 Deadlock Avoidance

Sometimes we cannot prevent deadlock entirely, but we can **avoid** it by careful resource allocation.

- The system examines each request and decides whether granting it keeps the system in a "safe" state.
- If granting a request leads to a potential deadlock, it is delayed.

Safe State

A system is in a **safe state** if it can allocate resources to every process in some order and still avoid deadlock.

Banker's Algorithm (Simplified View): Like a banker lending money, the OS checks if it can still fulfill everyone's maximum future needs before granting a loan (resource). If not, the request is postponed.
Suppose we have 3 processes {P1, P2, P3} and 10 resource units.

- Current allocation: P1(3), P2(2), P3(2) → Total used = 7
- Available = 10 – 7 = 3
- Max future demand: P1(4), P2(3), P3(5)

If we allocate 2 more units to P3, available = 1. Can we still satisfy everyone's maximum needs eventually? If yes, we remain in a safe state.

4.5 Key Takeaways

- Deadlock is like a "computer traffic jam."
- Four Coffman conditions must all hold for deadlock to occur.
- Prevention breaks one or more conditions.
- Avoidance (like Banker's algorithm) ensures the system stays in a safe state.

Pro Tip

In exams and interviews, always start by defining the four Coffman conditions first. Then explain prevention and avoidance strategies with examples.

4.6 Resource Allocation Graph (RAG) Algorithm

The **Resource Allocation Graph** is a graphical representation of the state of system resources and processes.

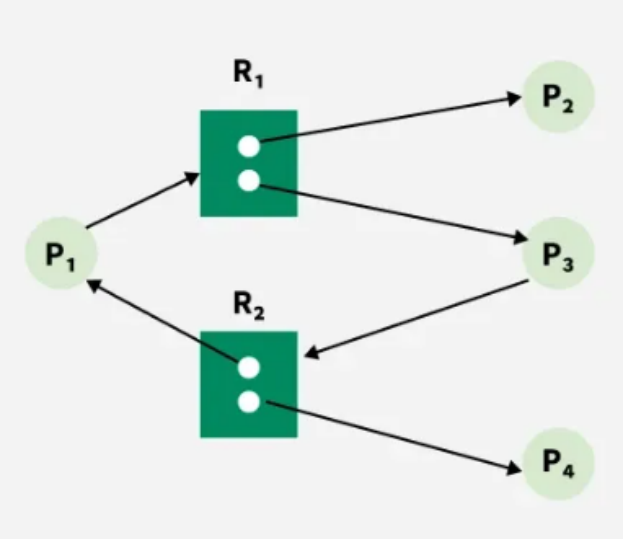
- **Vertices:**
  - Processes (P): Represented by circles (◯)

  - Resources (R): Represented by squares (□)

- **Edges:**
  - **Request Edge** ($P \rightarrow R$): Process P requests resource R.
  - **Assignment Edge** ($R \rightarrow P$): Resource R is allocated to process P.

Deadlock Detection Rule:

If the RAG contains a cycle,
→ If each resource type has only one instance, a cycle implies a **deadlock**.
→ If there are multiple instances, a cycle **may** indicate a deadlock.



Real-life Analogy: Think of a group of friends exchanging books. If Alice waits for Bob's book, Bob waits for Charlie's book, and Charlie waits for Alice's book, this forms a **cycle**—just like a deadlock!

4.7 Banker's Algorithm

This algorithm is used to **avoid deadlocks** by checking whether a system remains in a safe state before granting a resource request.

Key Idea:

Before granting a request, the OS behaves like a cautious banker who only gives loans if it is sure that all borrowers (processes) can finish eventually.
Inputs Required:

- **Available**: Vector of available resources.
- **Max**: Maximum demand of each process.
- **Allocation**: Currently allocated resources.
- **Need**: Calculated as Max − Allocation.

Safety Algorithm Steps:

1. Initialize **Work** = Available and **Finish[i]** = false for all processes.
2. Find a process P_i such that:
   - **Finish[i] = false**, and
   - **Need[i]** $\leq$ **Work**.
3. If such a process is found:
   - Work = Work + Allocation[i]
   - Finish[i] = true
   - Repeat step 2.
4. If all processes can finish (all Finish[i] = true), the system is in a **safe state**.

Example: Suppose we have 3 resource types (A, B, C). Available = (3, 3, 2). The algorithm checks if there exists a sequence of processes such that each can complete with the available resources. If yes, the system is SAFE.

Analogy: Think of a cautious banker who ensures that even after giving you a loan, there's always enough money left in the bank so that every customer can eventually be paid.

4.8 Deadlock Detection

If the system does **not** use Banker's Algorithm and does not attempt prevention, then the OS must periodically **check for deadlocks**.
Detection Algorithm:

1. Similar to the Safety Algorithm but uses the **current allocation** and **requests**.
2. Attempts to find a sequence of processes that can finish.
3. If some processes remain unfinished, a **deadlock exists**.

Note: Detection is like running a security check **after** the problem has occurred.

4.9 Recovery from Deadlock

Once a deadlock is detected, the system must recover. There are two main strategies:

1. **Process Termination**
   - Abort all deadlocked processes, or
   - Abort one process at a time until the cycle is broken.
2. **Resource Preemption**
   - Temporarily take a resource away from a process.
   - Roll back the process to a safe state and restart it later.

Real-life Example: Imagine a traffic jam in a narrow lane where cars are stuck nose-to-nose. Recovery could mean **backing up** some cars (preemption) or **removing** certain cars (termination).

Summary Table

Method	Goal	Key Idea
Resource Allocation Graph	Detect potential deadlocks	Look for **cycles** in the graph
Banker's Algorithm	Avoid deadlock	Grant resources only if **safe**
Detection Algorithm	Detect actual deadlock	Periodically check system state
Recovery	Resolve deadlock	**Terminate** processes or **preempt** resources

Key Takeaway for Students: Deadlock handling is like managing a busy traffic intersection.

- **Prevention/Avoidance** = Installing traffic signals to prevent jams.
- **Detection/Recovery** = Sending police officers to clear a jam after it occurs.

5. Memory Management

We will learn four fundamental concepts:

1. Address Binding
2. Logical vs Physical Address Space
3. Dynamic Loading
4. Swapping

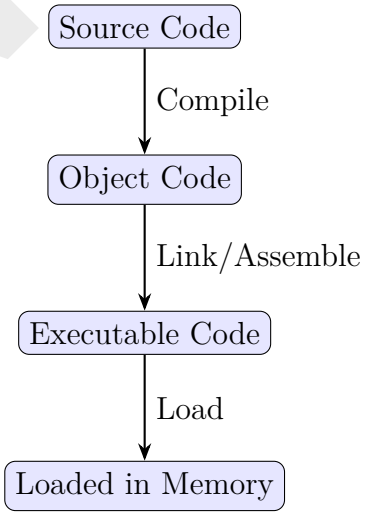
5.1 Address Binding

Address binding is the process of mapping a program's **logical addresses** (used by the CPU) to **physical addresses** (actual RAM locations).

Analogy: Booking a train seat. You can fix the seat while buying the ticket (compile time), at the station (load time), or even change during travel (execution time).

Types of Binding:

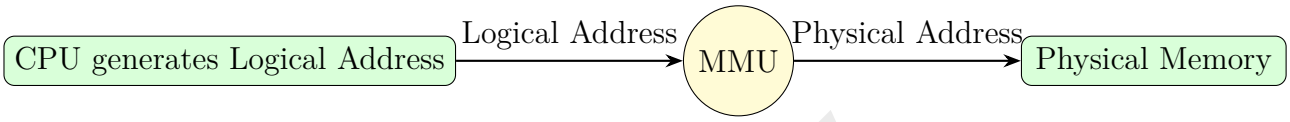
- **Compile-time:** Address decided during compilation.
- **Load-time:** Final address fixed when the program is loaded.
- **Execution-time:** Address resolved dynamically using the *Memory Management Unit (MMU)*.



Binding can occur at any of these stages.

5.2 Logical vs Physical Address Space

- **Logical Address:** Generated by CPU during program execution (also called *virtual address*).
- **Physical Address:** Actual location in RAM.



Example: A student writes a letter using a friend's name (*logical*). The postman needs the house number (*physical*) to deliver it.

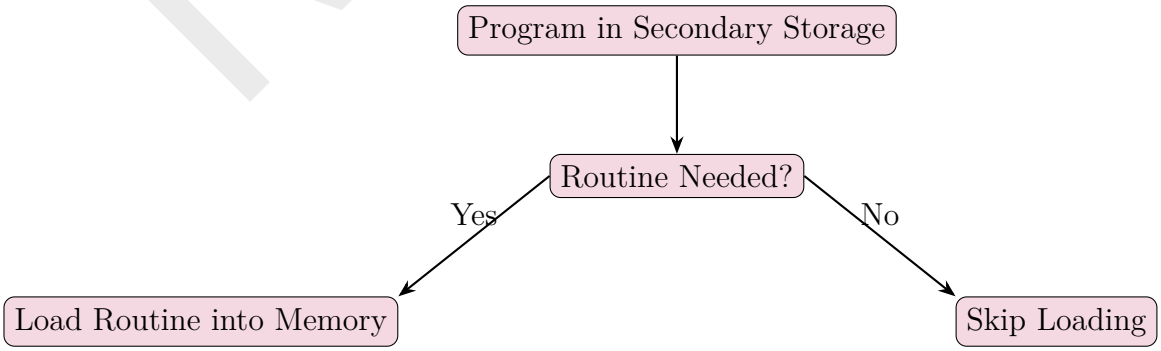
5.3 Dynamic Loading

Only the required parts of a program are loaded into memory when needed.

Real-life analogy: A restaurant prepares a dish *only after it is ordered*, saving kitchen space.

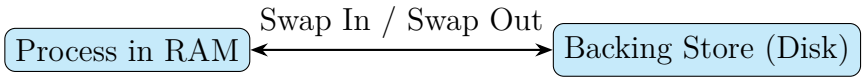
Benefits:

- Reduces memory usage.
- Speeds up program start-up.



5.4 Swapping

Swapping is the process of temporarily moving a process out of main memory to a backing store (disk) and later bringing it back for execution.



Analogy: Imagine a busy parking lot. To accommodate a VIP car, one car is temporarily moved to a side street and brought back later.

Frequent swapping can lead to **thrashing**—the system spends more time swapping than executing.

Summary Table

Concept	Definition	Key Analogy
Address Binding	Mapping logical to physical addresses	Train seat booking at different stages
Logical Address	Address generated by CPU	Friend's name in a letter
Physical Address	Actual RAM location	Friend's home address
Dynamic Loading	Load modules only when required	Restaurant cooks on order
Swapping	Moving processes between RAM and disk	Moving cars in/out of a parking lot

Takeaway: Efficient memory management ensures smooth multitasking, faster execution, and optimum use of limited RAM.

5.5 Contiguous Memory Allocation

Contiguous memory allocation is the **simplest** memory management technique. Here, each process is allocated a **single contiguous block of memory** in the main memory.

- The operating system and user processes are kept in **separate partitions**.

- Each process must occupy a single continuous segment of physical memory.

Real-life Analogy: Think of a bookshelf where each book (process) must be placed in a single uninterrupted section of the shelf. If there is not enough continuous space, the book cannot be placed—even if the total free space across the shelf is enough.

Types of Contiguous Allocation

1. **Single Partition Allocation:** Entire memory is divided into two parts—one for the OS and the other for user processes.
2. **Multiple Partition Allocation:** Memory is divided into several partitions of fixed or variable sizes.

Advantages:

- Simple to implement.
- Low overhead for address translation.

Disadvantages:

- Leads to **fragmentation** (explained next).
- Requires knowing the process size beforehand.

5.6 Fragmentation

Fragmentation occurs when memory is wasted because of how processes are allocated.

Two Types of Fragmentation

1. **External Fragmentation:** Free memory exists but is not contiguous, so a process cannot be allocated even though total free memory is sufficient. *Example:* Imagine a parking lot with enough total free spots, but only as scattered single spots. A large bus cannot park.
2. **Internal Fragmentation:** Occurs when a process is allocated a fixed-size block larger than needed. The unused space inside the block is wasted. *Example:* Booking a hotel room bigger than required; you pay for unused space.

Solutions:

- Compaction (to reduce external fragmentation): Shift processes to combine free memory into one large block.
- Dynamic allocation strategies (best-fit, worst-fit, first-fit) to reduce internal fragmentation.

5.7 Segmentation

Segmentation divides a process into logical segments such as:

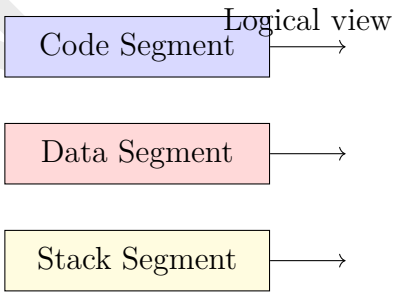
- Code
- Data
- Stack

Each segment has a **name** and a **length**, and can grow or shrink independently.

Key Idea

Segments are based on **logical divisions** of a program rather than fixed-size blocks.

Analogy: Think of a multi-chapter book where each chapter can be stored in different drawers. You can rearrange chapters without worrying about their physical locations.



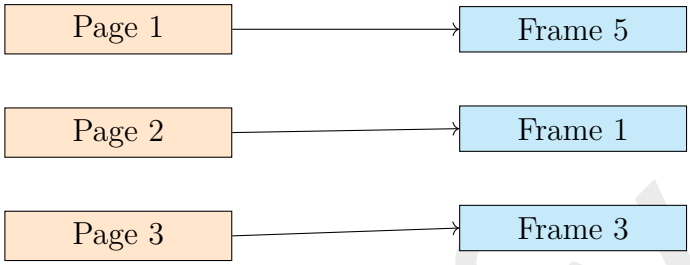
Advantages:

- Easy to grow/shrink individual segments.
- Reflects the programmer's logical structure.

Disadvantage: External fragmentation may still occur because segments can have varying sizes.

5.8 Paging

Paging eliminates external fragmentation by dividing physical memory into fixed-size blocks called **frames**, and dividing processes into blocks of the same size called **pages**.



Key Features:

- Pages and frames are of equal size (commonly 4 KB).
- A **page table** maps logical page numbers to physical frame numbers.

Analogy: Imagine a library storing a large book. Instead of keeping all chapters together, it splits them into equal-sized folders (pages) which can be placed anywhere on shelves (frames). A catalog (page table) keeps track of which folder is on which shelf.

Advantages of Paging

- No external fragmentation.
- Easy to allocate memory.

Disadvantages:

- Internal fragmentation may occur if a process does not completely fill its last page.
- Requires additional memory and time for page table lookups (mitigated using TLBs).

Summary Table

Method	Division Basis	Fragmentation Type
Contiguous Allocation	Continuous block	Both internal & external
Segmentation	Logical units (variable size)	External
Paging	Fixed-size blocks	Internal only

Takeaway

The evolution from contiguous allocation to segmentation and paging shows how OS designers trade simplicity for flexibility and efficiency. Modern systems often use a **combination of paging and segmentation** to get the best of both worlds.

Chapter 6 : Virtual Memory

Up to now, we have explored how operating systems manage memory using techniques like paging, segmentation, and contiguous allocation. But what happens when a process needs **more memory than is physically available**? The answer lies in one of the most revolutionary concepts in operating systems: **Virtual Memory.**

6.1 Virtual Memory Concept

Virtual memory is a memory management technique where the operating system gives the **illusion of a very large main memory** to programs, even if the physical RAM is smaller.

Key Idea

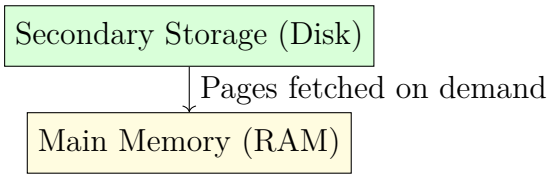
Virtual memory allows a process to execute even when it is not completely in physical memory. It uses a combination of **RAM** and **secondary storage (disk)** to create a large logical address space.

Real-life Analogy: Think of a small study table (RAM) with limited space for open books. Your bookshelf (hard disk) stores all your books. When you need a new book, you can temporarily swap out an unused book from the table to the shelf and bring in the required one. The table appears to hold all books over time, even though only a few fit at once.
Advantages:

- Programs can be larger than physical memory.
- Increases CPU utilization by allowing more processes to run concurrently.
- Provides isolation and protection between processes.

6.2 Demand Paging

Virtual memory is commonly implemented through **demand paging**. Instead of loading the entire process into RAM at once, the OS loads only the **required pages** when they are needed.



If a process accesses a page that is not in memory, a **page fault** occurs:

1. The CPU traps to the OS.
2. The OS locates the required page on disk.
3. The page is brought into a free frame in memory.

Page Fault

A page fault occurs when the referenced page is not in physical memory. Too many page faults lead to **thrashing**, where the CPU spends more time swapping pages than executing instructions.

6.3 Page Replacement Algorithms

When memory is full and a new page is needed, the OS must choose a page to remove. This decision is made using a **page replacement algorithm**.

Below are the most common algorithms with explanations, examples, and diagrams.

1. FIFO (First-In-First-Out) Algorithm

The page that has been in memory the longest is replaced first.

- Simple and easy to implement using a queue.
- However, it can suffer from **Belady's Anomaly**, where increasing the number of frames increases page faults.

Example: Reference string: 7, 0, 1, 2, 0, 3, 0, 4 Frames = 3

Step	Memory State	Page Fault?
7	7	Fault
0	7 0	Fault
1	7 0 1	Fault
2	2 0 1	Fault
0	2 0 1	No
3	3 0 1	Fault
0	3 0 1	No
4	4 0 1	Fault

Total Page Faults = **6**

2. LRU (Least Recently Used) Algorithm

Replaces the page that has not been used for the longest time.

- Better performance than FIFO in most cases.
- Requires keeping track of page usage history.

Example (Same reference string):

Step	Memory State	Page Fault?
7	7	Fault
0	7 0	Fault
1	7 0 1	Fault
2	2 0 1	Fault
0	2 0 1	No
3	3 0 2	Fault
0	0 3 2	Fault
4	4 0 3	Fault

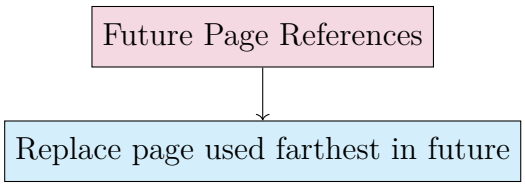
Total Page Faults = **7**

Practical Tip

LRU is close to the **optimal algorithm** in performance, but tracking usage can add overhead.

3. Optimal Page Replacement

Replaces the page that will **not be used for the longest time in the future**. It gives the best performance but is impractical because future references are unknown.



Example: Same reference string with 3 frames gives only **5 page faults**.

4. Other Strategies

- **Clock Algorithm (Second Chance):** Pages are arranged in a circular list with a reference bit. Pages with reference bit 0 are replaced first.
- **Least Frequently Used (LFU):** Page with the fewest references is replaced.

Key Observations

1. Optimal gives the fewest page faults but is theoretical.
2. LRU approximates optimal performance well.
3. FIFO is simple but can perform poorly due to Belady's anomaly.

Numerical Example: Comparing Algorithms

Consider the reference string: 1, 3, 0, 3, 5, 6, 3 Frames = 3

Algorithm	Total Page Faults
FIFO	6
LRU	5
Optimal	4

Summary and Takeaway

Virtual memory is the backbone of modern operating systems. It allows programs to run as if they have unlimited memory, while page replacement algorithms ensure efficient use of the limited physical memory.

Final Thoughts

Without virtual memory, we would need massive physical RAM to run today's applications. Thanks to this concept, even a laptop with 8 GB RAM can run programs requiring tens of GBs by cleverly swapping pages between RAM and disk.

6.4 Thrashing

Thrashing is a situation where the operating system spends most of its time **swapping pages in and out of memory** rather than executing actual processes.

Definition

Thrashing occurs when the page fault rate becomes so high that the CPU is busy servicing page faults instead of running user instructions.

Real-life Analogy: Imagine you are studying on a small desk. Every time you need a different book, you remove the current ones and fetch new ones from the shelf. If you are constantly swapping books and hardly reading, that's **thrashing**.

Causes of Thrashing:

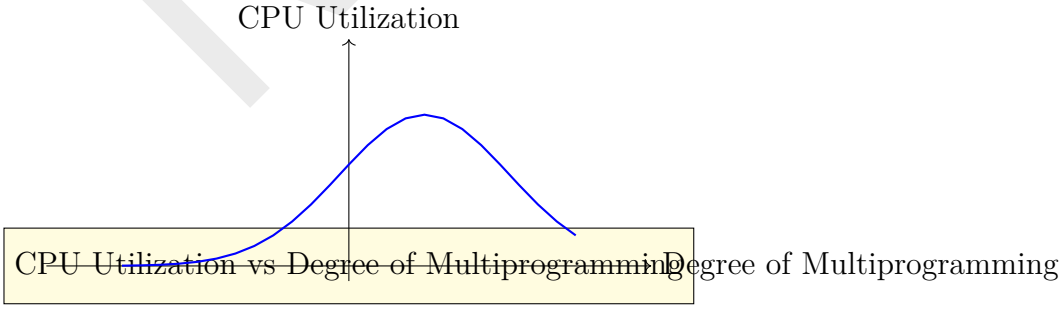
- Too many processes competing for limited physical memory.
- Each process does not have enough frames to hold its working set.
- Excessive multiprogramming beyond the system's capacity.

Detection:

- Monitor the **CPU utilization** and **page-fault rate**.
- If CPU utilization is low but page faults are high, thrashing is likely occurring.

Solution:

- Use a **working set model** to provide each process with enough frames.
- Reduce the degree of multiprogramming.



Too high multiprogramming leads to thrashing (drop in CPU utilization).

6.5 Cache Memory Organization

Cache memory is a **small, high-speed memory** located between the CPU and the main memory. Its goal is to **bridge the speed gap** between the fast CPU and slower main memory.

Key Idea

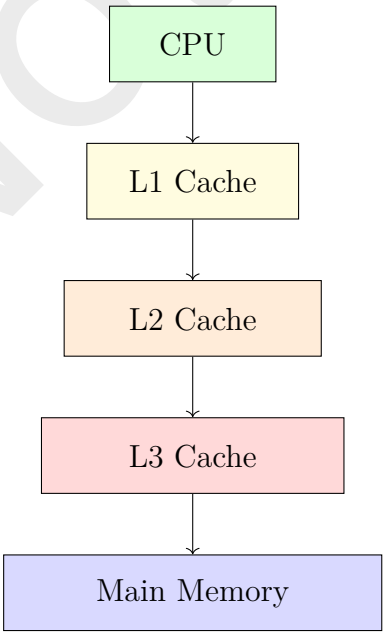
Cache stores frequently accessed data and instructions so that the CPU can retrieve them faster than from main memory.

Real-life Analogy: Think of cache as a **desk drawer** where you keep your most-used stationery like pens and notes, while main memory is like a big cupboard across the room.

6.6 Cache Levels

Modern computers use multiple levels of cache:

- **L1 Cache:** Smallest and fastest, located inside the CPU.
- **L2 Cache:** Larger but slightly slower, often on the CPU chip.
- **L3 Cache:** Shared among cores, larger and slower than L2.



6.7 Cache Mapping Techniques

To decide where to place data in the cache, three mapping methods are used:

1. **Direct Mapping:** Each block of main memory maps to exactly one cache line. Simple but may cause frequent replacements.
2. **Associative Mapping:** A block can be placed in any cache line. Flexible but requires searching the entire cache.
3. **Set-Associative Mapping:** A hybrid of direct and associative mapping. Cache is divided into sets, and each block maps to a set.

Performance Metric

$$\text{Cache Hit Ratio} = \frac{\text{Number of Cache Hits}}{\text{Total Memory Accesses}}$$

Higher hit ratio means faster program execution.

6.8 Locality of Reference

Locality of Reference refers to the tendency of a program to access a relatively small portion of its address space repeatedly over a short period of time.

Types of Locality:

- **Temporal Locality:** Recently accessed items are likely to be accessed again soon. *Example:* Reusing a variable in a loop.
- **Spatial Locality:** Items near a recently accessed location are likely to be accessed soon. *Example:* Accessing consecutive elements of an array.

Why Locality Matters

Cache memory relies heavily on locality of reference. Programs with strong locality achieve higher cache hit ratios, resulting in faster execution.

Example Calculation:

Suppose a program makes 1000 memory accesses. 800 are cache hits and 200 are cache misses. If cache access time is 10 ns and main memory access time is 100 ns:

$$\text{Average Access Time} = (0.8 \times 10) + (0.2 \times 100) = 8 + 20 = \boxed{28\,ns}$$

If locality improves and hit ratio rises to 0.9:

$$\text{Average Access Time} = (0.9 \times 10) + (0.1 \times 100) = 9 + 10 = \boxed{19\,ns}$$

Better locality directly reduces access time!

Summary

Key Takeaways

- **Thrashing** occurs when excessive page faults overwhelm the CPU.
- **Cache memory** acts as a high-speed buffer between CPU and main memory.
- **Locality of reference** is the principle that makes cache effective.

Understanding these concepts allows engineers to design faster, more efficient memory systems and to avoid performance disasters in real-world applications.

Unit-2 : Deadlocks & Memory Management – Solved Numericals

The following numericals cover all major concepts of Deadlocks, Memory Management and Virtual Memory.

Deadlocks

1. **Deadlock Characterization**
   Question: A system has 3 processes P_1, P_2, P_3 and 3 resource types A, B, C.
   Total instances: $A = 6, B = 3, C = 4$.
   Allocated resources:

	A	B	C
P_1	2	1	1
P_2	2	1	0
P_3	1	0	1

   Available resources: $A = 1, B = 1, C = 2$.
   Check whether the system is in a safe state using the **Banker's Safety Algorithm**.

   Solution:

   - Let maximum needs be:

	A	B	C
P_1	3	2	2
P_2	2	2	2
P_3	3	1	1

   - Need matrix = Max – Allocation:

	A	B	C
P_1	1	1	1
P_2	0	1	2
P_3	2	1	0

- Work = Available = (1,1,2). Finish = (F,F,F).
- Check P2: Need (0,1,2) $\leq$ Work (1,1,2) $\Rightarrow$ Execute P2.
  Work = Work + Allocation(P2) = (1+2, 1+1, 2+0) = (3,2,2).
  Finish(P2) = T.
- Check P1: Need (1,1,1) $\leq$ Work (3,2,2) $\Rightarrow$ Execute P1.
  Work = (3+2, 2+1, 2+1) = (5,3,3). Finish(P1) = T.
- Check P3: Need (2,1,0) $\leq$ Work (5,3,3) $\Rightarrow$ Execute P3.
  Work = (5+1, 3+0, 3+1) = (6,3,4). Finish(P3) = T.

Since all processes can finish, **the system is in a safe state**. Safe sequence: $\boxed{\text{P2} \rightarrow \text{P1} \rightarrow \text{P3}}$.

2. **Resource Allocation Graph**
   Question: Consider a system with 4 processes $\{P_1, P_2, P_3, P_4\}$ and 3 resources $\{R_1, R_2, R_3\}$.
   Edges are: $P_1 \rightarrow R_1$, $R_1 \rightarrow P_2$, $P_2 \rightarrow R_2$, $R_2 \rightarrow P_3$, $P_3 \rightarrow R_3$, $R_3 \rightarrow P_1$, $P_4 \rightarrow R_2$.
   Is there a deadlock?

   Solution:
   The cycle $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_1$ exists.
   Since each resource has only one instance, the cycle implies a **deadlock** among $\{P_1, P_2, P_3\}$.

3. **Banker's Algorithm – Need Calculation**
   Question:
   Max =
   $$\begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix}$$

Alloc =

$$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

Compute the **Need** matrix.

Solution:
Need = Max – Alloc:

$$\begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

4. **Deadlock Prevention**
   Question: A system allows at most 4 processes to compete for 3 identical printers.
   If each process may request 2 printers, prove that deadlock **cannot occur** if we apply the condition *"at least one resource less than total"*.

   Solution:
   Total printers $R = 3$, each needs at most 2.
   Using $R \geq m \times (n-1) + 1$ for avoidance, with $m = 2$, $n = 4$:
   Required $R \geq 2(4-1) + 1 = 7$. Not satisfied.
   Hence we limit **at most 1 printer per request**.
   By holding only 1, circular wait is prevented $\Rightarrow$ **Deadlock prevented**.

5. **Deadlock Detection**
   Question: A system has Allocation = [2,1,1], Request = [1,1,0], Available = [1,0,1].
   Should the OS initiate recovery?

   Solution:
   Check if Request $\leq$ Available.
   $[1, 1, 0] \leq [1, 0, 1]$? No (second resource).
   Process cannot proceed $\rightarrow$ **Deadlock detected**.
   OS must recover (preempt or terminate a process).

Memory Management

6. **Address Binding**
Question: A variable has a logical address of 0x1A3F. The base register contains 0x4000.
Find the physical address.

Solution:
Physical Address = Base + Logical
= 0x4000 + 0x1A3F = $\boxed{\text{0x5A3F}}$.

7. **Swapping Time**
Question: Assume transfer rate of disk is 40 MB/s.
To swap out a 20 MB process and swap in a 30 MB process, how long will it take?

Solution:
Total data moved = 20 + 30 = 50 MB.
Time = 50 / 40 = $\boxed{\text{1.25 seconds}}$.

8. **Fragmentation Calculation**
Question: A memory has 1000 KB free space divided into holes of 200 KB, 300 KB, and 500 KB.
Processes of size 180 KB, 290 KB, and 490 KB arrive.
Find internal fragmentation if first-fit is used.

Solution:

- 180 KB in 200 KB hole ⇒ internal = 20 KB
- 290 KB in 300 KB hole ⇒ internal = 10 KB
- 490 KB in 500 KB hole ⇒ internal = 10 KB

Total internal fragmentation = $\boxed{\text{40 KB}}$.

9. **Paging Calculation**
Question: Logical address space = 32 bits, page size = 4 KB.
Find number of bits for page number and offset.

Solution:
Page size = 4 KB = 2^{12} bytes $\Rightarrow$ Offset = 12 bits.
Remaining = 32 - 12 = $\boxed{20 \text{ bits}}$ for page number.

10. **Segmentation**
Question: A segment table contains:

Segment	*Base*	*Limit*
0	219	600
1	2300	14
2	90	100
3	1327	580

Find physical address for logical address (2,50).

Solution:
Check limit: 50 ¡ 100 $\Rightarrow$ valid.
Physical = Base + d = 90 + 50 = $\boxed{140}$.

11. **Dynamic Loading Advantage**
Question: A program needs 500 KB code but only 300 KB is frequently used.
If dynamic loading loads only frequently used portion initially, how much memory is saved?

Solution:
Saved = 500 – 300 = $\boxed{200 \text{ KB}}$ initially.

Virtual Memory

12. **Demand Paging Page Faults**
Question: Reference string = 1,2,3,4,1,2,5,1,2,3,4,5 with 3 frames using FIFO.

Compute the number of page faults.

Solution:
Page faults sequence = [1*,2*,3*,4*,1,2,5*,1,2,3*,4*,5*]
Total = $\boxed{\text{9 page faults}}$.

13. **LRU Calculation**
Question: Using the same reference string with 3 frames and LRU, find page faults.

Solution:
Apply LRU replacement ⇒ Faults = $\boxed{8}$.

14. **Optimal Algorithm**
Solution:
Optimal replacement ⇒ Faults = $\boxed{7}$ (least of all).

15. **Effective Access Time**
Question: Memory access time = 100 ns, Page fault service time = 8 ms, Page fault rate = 0.001.
Compute effective access time (EAT).

Solution:
EAT = (1 - p) × ma + p × page_fault_time
= 0.999 × 100 ns + 0.001 × 8 ms
= $\boxed{8.1\ \mu s}$ (dominated by faults).

16. **Thrashing Detection**
Question: CPU utilization drops from 80% to 20% when degree of multiprogramming is increased.
What is happening?

Solution:
This indicates **Thrashing** as processes spend most time paging.

17. **Cache Hit Ratio**
Question: Cache access = 20 ns, Main memory access = 100 ns, Hit ratio = 0.9.
Find effective access time.

Solution:
EAT = (Hit ratio × Cache) + (Miss ratio × (Cache + Memory))
= 0.9 × 20 + 0.1 × (20 + 100) = 18 + 12 = $\boxed{30 \text{ ns}}$.

18. **Locality of Reference**
Question: Explain why programs with loops exhibit high temporal locality.

Solution:
Loop instructions are fetched repeatedly in a short time ⇒ recent references are reused ⇒ $\boxed{\text{High temporal locality}}$.

19. **Working Set**
Question: A process references 100 pages repeatedly. If working set window W=20,
how many frames are needed to avoid thrashing?

Solution:
Frames ≥ Working set size = $\boxed{20 \text{ frames}}$.

20. **Translation Lookaside Buffer (TLB)**
Question: TLB lookup time = 10 ns, Memory access = 100 ns, Hit ratio = 80%.
Compute EAT.

Solution:
EAT = (0.8 × (10 + 100)) + (0.2 × (10 + 100 + 100))
= 0.8 × 110 + 0.2 × 210 = 88 + 42 = $\boxed{130 \text{ ns}}$.

MST 2 - Practice Sheet

General Instructions: Attempt all questions
Total Time: 1 hour
Total Marks: 20

Q.No	Statement
	Section A
1	Explain the concept of deadlock in operating systems. Describe the four necessary conditions for a deadlock to occur.
2	Discuss the Banker's Algorithm for deadlock avoidance. Explain how the safety algorithm works in determining whether a system is in a safe state.
3	Describe the concept of memory fragmentation. Explain the differences between internal and external fragmentation.
4	Explain the concept of virtual memory. Describe the process of demand paging and its advantages in modern operating systems.
5	What is the difference between contiguous memory allocation and paging? Discuss the advantages and disadvantages of paging.
	Section B
6	Given the following system details, apply the Banker's Algorithm to determine if the system is in a safe state. Available resources: (3, 2, 2) Processes: P1, P2, P3 Maximum resources required by P1: (7, 5, 3) Maximum resources required by P2: (3, 2, 2) Maximum resources required by P3: (9, 0, 2) Allocated resources for P1: (2, 1, 1) Allocated resources for P2: (2, 1, 1) Allocated resources for P3: (3, 2, 2)
7	A system has 4 frames of physical memory. The page reference string is: 2, 3, 2, 1, 3, 4, 1, 3, 2. Using the FIFO page replacement algorithm, calculate the number of page faults.

Answers

Section A

1. **Deadlock and Conditions:**
   Deadlock occurs when a set of processes are in a state where they are unable to proceed due to each waiting for a resource held by another. The four necessary conditions for deadlock are:
   1. **Mutual Exclusion:** At least one resource is held in a non-shareable mode.
   2. **Hold and Wait:** A process holding at least one resource is waiting for additional resources held by other processes.
   3. **No Preemption:** Resources cannot be preemptively taken from processes.
   4. **Circular Wait:** A set of processes exists such that each process is waiting for a resource held by the next process in the set.

2. **Banker's Algorithm for Deadlock Avoidance:**
   The Banker's Algorithm is used to avoid deadlocks by ensuring that resource allocation always keeps the system in a safe state. The safety algorithm checks if there exists a sequence of process execution that can complete without causing a deadlock. Steps include:
   1. Check if the process's maximum request can be satisfied with available resources.
   2. If yes, assume the process completes and releases its resources.
   3. Repeat until all processes can complete, or there's no sequence left (unsafe state).

3. **Memory Fragmentation:**
   Memory fragmentation occurs when free memory is split into small blocks, making it difficult to allocate large contiguous chunks of memory. There are two types:
   - **Internal Fragmentation:** Occurs when allocated memory may be larger than requested memory.
   - **External Fragmentation:** Occurs when free memory is scattered in small blocks, preventing large contiguous blocks from being allocated.

4. **Virtual Memory and Demand Paging:**
   Virtual memory allows programs to use more memory than physically available by swapping pages between disk and RAM. Demand paging only loads pages into memory when they are needed, improving system efficiency. Advantages include:

1. Efficient memory use.
2. Allows programs to run larger than physical memory.
3. Isolation between processes, preventing one from corrupting another.

5. **Contiguous Memory Allocation vs Paging:**
- **Contiguous Memory Allocation:** Memory is allocated in one continuous block to each process.
- Advantages: Simple and fast access. - Disadvantages: External fragmentation.

- **Paging:** Memory is divided into fixed-size pages and processes are allocated non-contiguous pages.
- Advantages: Eliminates external fragmentation. - Disadvantages: Overhead for managing page tables.

Section B

6. **Banker's Algorithm Safety Check:**
Given:
Available Resources: (3, 2, 2)
Maximum Resource Needs:
- P1: (7, 5, 3)
- P2: (3, 2, 2)
- P3: (9, 0, 2)
Allocated Resources:
- P1: (2, 1, 1)
- P2: (2, 1, 1)
- P3: (3, 2, 2)

1. Calculate the remaining needs for each process:
- P1: (7-2, 5-1, 3-1) = (5, 4, 2)
- P2: (3-2, 2-1, 2-1) = (1, 1, 1)
- P3: (9-3, 0-2, 2-2) = (6, -2, 0)
(P3 has negative resources, so it cannot proceed.)

2. Check if processes can be completed. P1 and P2 can still proceed as their requirements can be satisfied with available resources. P3 cannot proceed. Hence, the system is in a safe state.

7. **Page Faults using FIFO Algorithm:**
Page reference string: 2, 3, 2, 1, 3, 4, 1, 3, 2.
With 4 frames, the FIFO algorithm works as follows:
- Initially, the frames are empty.

- 2 is added → (2)
- 3 is added → (2, 3)
- 2 is already in the frame → (2, 3)
- 1 is added → (2, 3, 1)
- 3 is already in the frame → (2, 3, 1)
- 4 is added → (2, 3, 1, 4)
- 1 is already in the frame → (2, 3, 1, 4)
- 3 is already in the frame → (2, 3, 1, 4)
- 2 is already in the frame → (2, 3, 1, 4)

Total page faults = 6 (when 2, 3, 1, and 4 are initially loaded).