There are **4 main methods** to deal with deadlock in an Operating System:

---

1. Deadlock Prevention

Here, the system is designed in such a way that **deadlock can never happen**.
We do this by making sure at least one of the 4 deadlock conditions (mutual exclusion, hold and wait, no preemption, circular wait) never occurs.
👉 Example: For circular wait, give resources a fixed order, so no cycle can form.

---

2. Deadlock Avoidance

In this method, the system **carefully checks resource requests before granting them**.
It uses algorithms (like the **Banker's Algorithm**) to decide whether giving resources to a process will keep the system in a safe state or not.
If giving resources may cause deadlock, the request is not allowed.
👉 Example: Like a banker only gives loans if he is sure he can still satisfy all customers later.

---

3. Deadlock Detection and Recovery

Here, the system **does not prevent deadlock**. Instead, it allows deadlock to occur but has mechanisms to **detect it** and then **recover**.

- **Detection:** The OS keeps checking for circular waits using algorithms (like Wait-for Graph).
- **Recovery:** Once detected, the OS may stop (kill) one or more processes, or take back resources to break the deadlock.
  👉 Example: If two processes are stuck waiting forever, the system kills one so the other can continue.

---

4. Deadlock Ignorance (Ostrich Approach)

Sometimes the system just **ignores deadlock**. It assumes that deadlock is rare and not worth the overhead of prevention/avoidance.
This is used in many general-purpose systems like Windows, Linux.
👉 Example: If deadlock happens, the user may just restart the system.

✅ **Summary:**

- **Prevention** → Stop deadlock from happening.
- **Avoidance** → Check before allocating resources.
- **Detection & Recovery** → Let it happen, then fix it.
- **Ignorance** → Do nothing and assume it won't happen often.

4 ways of Deadlock Prevention:

---

1. Prevent Mutual Exclusion
Deadlock happens when a resource can be used by only one process at a time. To prevent this, we can make some resources *sharable*. For example, many processes can read the same file at the same time. If resources don't have to be locked by only one process, deadlock won't happen. However, some resources like printers cannot be shared, so this method cannot always work.

---

2. Prevent Hold and Wait
Deadlock also happens when a process is holding one resource and waiting for another. To stop this, we can make a rule that a process must ask for *all the resources it will need at once* before starting. Another way is that if a process wants more resources, it must release the ones it already has first. This prevents "waiting while holding," but the problem is resources may stay unused for a long time.

---

3. Prevent No Preemption
Deadlock becomes possible when a resource cannot be taken back from a process. To prevent this, we can allow preemption, meaning if a process is waiting for a resource that is busy, it must release all the resources it already has. Later, it can try again to get everything it needs. This way, no process blocks others forever, because resources can be given back.

---

4. Prevent Circular Wait
Deadlock often forms a circle, like A waits for B, B waits for C, and C waits for A. To prevent this, we can give every resource a number and tell processes they must ask for resources in *increasing order of numbers*. For example, if a printer is number 1 and a scanner is number

2, a process must ask for the printer before asking for the scanner. This breaks the circle, so deadlock cannot occur.

🔹 What is Deadlock Avoidance?

Deadlock avoidance means the system **carefully checks every resource request** before giving it to a process.
👉 The idea is: *"Don't enter a situation that may lead to deadlock."*
So, resources are only allocated if the system can still remain in a **safe state**.

🔹 Safe State

- A **safe state** is a condition where the system can give resources to all processes one by one (in some order) without causing deadlock.
- If the system cannot find such an order, it is an **unsafe state**, which may lead to deadlock.

Example:
Imagine there are 10 chocolates 🍫. Three kids ask for chocolates, but if you give too many to one kid, the others may get stuck waiting. So, you give them only in such a way that later you can still satisfy everyone → that's a safe state.

🔹 Technique Used → Banker's Algorithm

The most common method for deadlock avoidance is the **Banker's Algorithm** (named because it works like a banker giving loans).

How it works:

1. Each process must tell in advance the **maximum resources** it may need.
2. Whenever a process requests resources, the OS checks:
    - If enough resources are available now.
    - If granting the request keeps the system in a **safe state**.

3. If yes → the request is granted.
   If no → the process must wait.

Main memory

Main memory is the **primary storage** of a computer, also called **RAM (Random Access Memory)**. It is the place where data, instructions, and programs are stored **temporarily** while the CPU is working on them.

Here's a simple breakdown:

- **Fast but temporary**: Main memory is much faster than hard disk or secondary storage, but all data is lost when the computer is turned off.
- **Direct access**: The CPU can directly read from and write to main memory, which makes processing very quick.
- **Stores running programs**: When you open a program (like MS Word or a browser), it is loaded from the hard drive into main memory so the CPU can execute it.
- **Volatile**: Since it loses data after shutdown, important files must be saved in permanent storage (hard disk, SSD).

Memory Management

Memory Management is a part of the **Operating System (OS)** that handles how the computer's memory is used. Since the CPU needs memory to run programs, the OS makes sure memory is given to each program properly and taken back when not needed.

Here's a very easy explanation of its main tasks:

1. **Allocation**: The OS gives memory to programs when they need it. For example, if you open MS Word, the OS gives it some space in memory.
2. **Deallocation**: When you close the program, the OS frees that memory so other programs can use it.
3. **Protection**: The OS ensures that one program cannot use the memory of another program without permission (to avoid errors or crashes).
4. **Efficient use**: The OS tries to use memory smartly so that no space is wasted and programs run faster.

🔹 Why Memory Management is Required?

1. **Efficient Use of Memory**
    - Memory is limited, but many programs run at the same time.
    - Memory management ensures no part of memory is wasted and all programs get enough space to run.
2. **Process Isolation and Protection**
    - Each program should use only *its own memory*.
    - Memory management protects one program from accidentally (or intentionally) using another program's memory.
3. **Multiprogramming**
    - To run multiple programs together, the OS must divide memory fairly among them.
    - Example: You can play music while browsing the internet because memory is managed properly.
4. **Relocation and Sharing**
    - Programs may move in memory during execution.
    - Some parts of memory (like libraries or shared files) can be used by many programs together.
    - Memory management handles this sharing safely.
5. **Avoid Fragmentation**
    - Memory can get broken into small pieces (fragments) when programs are loaded and removed.
    - Memory management tries to reduce this problem, so large programs can still fit.
6. **Support for Virtual Memory**
    - Sometimes programs need more memory than physically available.

    - Memory management uses **virtual memory** (part of hard disk used like RAM) to handle big programs.

1️⃣ What is Swapping?

Swapping is a memory management technique where a process is **temporarily moved from main memory (RAM) to secondary storage (like a hard disk) and brought back later**.

- This is done **when RAM is full** and there's not enough space to load a new process.
- The part moved to disk is called a **"swap space"** or **"swap file"**.

2️⃣ How Swapping Works

1. A process **resides in RAM** and executes.
2. If **another process needs memory** but RAM is full:
    - The OS selects a process in RAM to **move to disk** (swap out).
    - The new process is **loaded into RAM**.
3. Later, when the swapped-out process needs to execute again:
    - It is **brought back into RAM** (swap in).

3️⃣ Key Points

- **Purpose**: Free up RAM so more processes can run.
- **Pros**:
    - Increases the number of processes that can execute.
    - Helps in multitasking.
- **Cons**:
    - Slower than RAM because disk access is much slower.

  - Excessive swapping can lead to **“thrashing”**, where the CPU spends more time swapping than executing processes.

4️⃣ Example (Simple)

Suppose RAM can hold **3 processes**: P1, P2, P3.

- A new process P4 arrives. RAM is full → OS swaps P1 to disk.
- P4 is loaded into RAM.
- Later, P1 is needed → OS swaps P4 or another process back to disk and brings P1 back to RAM.

1️⃣ Contiguous Memory Allocation

In contiguous memory allocation, each process is stored in **one single continuous block of memory**. This means the process must fit into one straight portion of RAM.

There are two types of this:

- **Fixed-Size Partitioning:** Here, memory is divided into fixed blocks. Each process goes into one block, even if it does not use the whole space. This is very simple but wastes memory if the process is smaller than the block (this waste is called *internal fragmentation*). Also, if a process is bigger than the block, it cannot be loaded. For example, if memory is 100 MB divided into five partitions of 20 MB each, a 12 MB process will waste 8 MB, and a 25 MB process cannot fit at all.

- **Variable-Size Partitioning:** Here, memory is divided according to the size of the process. Each process gets exactly the space it needs, so no memory is wasted inside partitions. But this can create small empty spaces between processes, known as *external fragmentation*. Also, it needs extra work to find a proper space for a new process. For example, in 100 MB memory, if a 12 MB process and a 25 MB process are stored, the remaining 63 MB may be split further for other processes.

2️⃣ Non-Contiguous (Uncontiguous) Memory Allocation

In non-contiguous allocation, a process does **not need to be in one continuous block**. Instead, its memory can be broken into smaller parts and placed in different areas of RAM. The operating system keeps track of these pieces using tables or linked lists. This method avoids external fragmentation and can store large processes even when big continuous memory is not available. However, it is more complex to manage and can be slightly slower because the process is spread out in different places. For example, if a process needs 40 MB, the OS may give it 10 MB from one place, 15 MB from another, and 15 MB from somewhere else.

1️⃣ What is Fragmentation?

Fragmentation means **wastage of memory** that happens when free space is not used properly. Even though memory is available, it may not be usable for new processes because of the way it is divided. This creates a problem in allocating memory to new processes. There are two types of fragmentation: internal and external.

2️⃣ Internal Fragmentation

Internal fragmentation happens in **fixed-size partitioning**. In this case, memory is divided into fixed blocks, and each process must fit inside one block. If a process is smaller than the block, the extra space inside the block remains unused and is wasted. For example, if a memory block is 20 MB and a process is only 12 MB, then 8 MB inside that block is wasted. This type of wastage is called internal fragmentation because the wasted space is **inside** the allocated block.

3️⃣ External Fragmentation

External fragmentation happens in **variable-size partitioning**. Here, memory is divided according to the process size, but over time, the free memory becomes scattered in small pieces across RAM. Even if the total free memory is enough, it may not be possible to allocate it to a new process if it requires one large continuous block. For example, if free memory is available as 10 MB, 5 MB, and 15 MB, and a process needs 20 MB, it cannot be allocated, even though total free memory is 30 MB. This is called external fragmentation because the wasted space is **outside** the allocated blocks, scattered between them.

1️⃣ First Fit

In **First Fit allocation**, the operating system scans memory **from the beginning** and gives the process the **first block** that is large enough to fit.
Steps:

1. Start checking memory blocks from the beginning.
2. Find the first block that is equal to or larger than the process size.
3. Allocate the process to that block.

This method is **fast and simple**, but it can create external fragmentation near the beginning of memory because small unusable gaps may be left.
Example: Suppose memory blocks are 10 MB, 20 MB, 15 MB, and 30 MB. If a process needs 12 MB, the first block (10 MB) is too small. The second block (20 MB) is big enough, so the process is placed there.

2️⃣ Best Fit

In **Best Fit allocation**, the operating system looks at **all the free blocks** and chooses the **smallest block** that is big enough for the process.
Steps:

1. Check all available memory blocks.
2. Collect the blocks that are equal to or larger than the process size.
3. From these, choose the smallest one and allocate the process.

This method **reduces wasted space inside the block**, but it is **slower** because it checks all blocks. Also, it may leave many very small gaps that cannot be used later (external fragmentation).
Example: If memory blocks are 10 MB, 20 MB, 15 MB, and 30 MB, and a process needs 12 MB, suitable blocks are 20 MB and 15 MB. The smallest suitable one is 15 MB, so the process is placed there.

3️⃣ Worst Fit

In **Worst Fit allocation**, the operating system chooses the **largest available block** for the process.
Steps:

1. Look at all free memory blocks.

2. Find the largest block among them.

3. Allocate the process to that block.

This method leaves **larger blocks free for future bigger processes**, but it can also waste a lot of memory space unnecessarily.
Example: If memory blocks are 10 MB, 20 MB, 15 MB, and 30 MB, and a process needs 12 MB, the largest block is 30 MB, so the process is placed there.

📘 Paging (Non-Contiguous Memory Allocation)

🔹 What is Paging?

Paging is a **memory management technique** used in operating systems. Its main purpose is to load a process from secondary storage (hard disk) into main memory (RAM) in the form of **pages**.

- A **process is divided into small equal parts called *pages***.

- The **main memory is divided into small equal parts called *frames***.

- **One page fits into one frame**.

- Pages of a process can be stored in different places in memory (not necessarily together), so paging supports **non-contiguous allocation**.

- Pages are brought into memory only when needed; otherwise, they stay in secondary storage.

✅ Important rule: **Page size = Frame size** (they must be equal).

🔹 Page Table

The **Page Table** is a special data structure used by the operating system to keep track of where each page of a process is stored in memory.

- The CPU generates a **logical address** (also called virtual address).
- This logical address is divided into two parts:
    1. **Page number** → used as an index in the page table.
    2. **Offset** → the exact location inside the page.
- The page table maps the **page number** to the **frame number** where the page is actually stored in RAM.
- The final **physical address** is made by combining the **frame number** with the **offset**.

So, the page table works like a dictionary that translates **logical address** → **physical address**.

🔹 Logical Address vs Physical Address

- **Logical Address (generated by CPU):** Made of *page number* + *offset*.
- **Physical Address (used in RAM):** Made of *frame number* + *offset*.
- The offset remains the same in both logical and physical addresses. Only the page number changes to a frame number using the page table.

👉 This process is called **address translation**.

🔹 Page Replacement

When memory is full and a new page needs to be loaded, the operating system must decide **which existing page to remove**. This is done using **Page Replacement Algorithms**:

1. **FIFO (First In, First Out):**
   - Remove the page that entered memory first.
   - Works like a queue (oldest page goes out).
2. **LRU (Least Recently Used):**
   - Remove the page that has not been used for the longest time.
   - Based on past usage.
3. **Optimal (OPT):**
   - Replace the page that will **not be used for the longest time in the future**.
   - This is the best method but difficult to implement in real life, because it requires knowing future use.
4. **MRU (Most Recently Used):**
   - Remove the page that was used **most recently**.
   - Used in some special cases.

Aspect	Logical Address	Physical Address
Definition	Address generated by the **CPU** during program execution.	Actual address in **main memory (RAM)** where data/instruction is stored.

Who generates it?	Generated by the **CPU**.	Determined by the **Memory Management Unit (MMU)** after translation.
Seen by	Visible to the **user/programmer**.	Not visible to the user, only to the **system hardware**.
Range	Depends on the CPU architecture (e.g., 16-bit, 32-bit, 64-bit).	Depends on the size of **physical memory (RAM)**.
Example in paging	Given as **page number + offset**.	Translated into **frame number + offset**.
Mapping	Logical → mapped to Physical using **page table / segment table**.	Directly points to an actual memory cell in RAM.
Other name	Also called **Virtual Address**.	Also called **Real Address**.

Ans of assignment

a) A process of size 212 KB needs to be allocated in memory. Free partitions are of sizes: 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB.

- Show allocations using First Fit, Best Fit, and Worst Fit.

- Which allocation leaves the least external fragmentation?

b) Explain the difference between internal and external fragmentation

Given

- Process size = **212 KB**

- Free partitions: **100 KB, 500 KB, 200 KB, 300 KB, 600 KB**

---

a) Allocation by different strategies

1. First Fit

Scan from the beginning and place the process in the **first** partition big enough.

- 100 KB — too small
- 500 KB — fits → allocate 212 KB here
    - leftover in that partition = 500 − 212 = **288 KB**

Resulting free partitions after allocation:
`100 KB, 288 KB, 200 KB, 300 KB, 600 KB`

---

2. Best Fit

Choose the **smallest** partition that is still ≥ the process size.

- Candidates ≥ 212 KB: 500, 300, 600 (200 is too small)
- Smallest of these = **300 KB** → allocate 212 KB here
    - leftover = 300 − 212 = **88 KB**

Resulting free partitions:
`100 KB, 500 KB, 200 KB, 88 KB, 600 KB`

---

3. Worst Fit

Choose the **largest** partition available.

- Largest partition = **600 KB** → allocate 212 KB here
    - leftover = 600 − 212 = **388 KB**

Resulting free partitions:
`100 KB, 500 KB, 200 KB, 300 KB, 388 KB`

Which allocation leaves the least external fragmentation?

Short answer: **Worst Fit** tends to leave the least *external fragmentation* in this scenario.

Why (simple explanation):

- **External fragmentation** = many small free holes that cannot fit incoming processes even though the total free memory is enough.
- **Best Fit** produced a very small leftover **88 KB** — that small hole is likely useless for future processes of similar size, so it *increases* external fragmentation.
- **Worst Fit** left a large leftover **388 KB**, which is still usable for many future requests. Large holes are *less* fragmenting because they can accommodate more processes.
- **First Fit** is in between: it left **288 KB**, usable but not as large as Worst Fit's leftover.

So, Best Fit minimizes the wasted space inside the chosen partition (small leftover), but that often *creates many small unusable holes* overall → **more external fragmentation**. Worst Fit purposely leaves big holes → **less external fragmentation**.

Aspect	Internal Fragmentation	External Fragmentation
Where it happens	Inside an allocated partition/block	Between allocated partitions (in free memory)
Cause	Allocated block is bigger than process size → unused space **inside** block	Free memory is divided into many small scattered holes

Example	Process needs 212 KB, but allocated 300 KB → extra 88 KB wasted inside	Total free memory = 400 KB, but scattered as 100 KB + 150 KB + 150 KB → no single block large enough for 212 KB
Type of waste	Wasted space is **inside allocated partitions**	Wasted space is **outside allocated partitions** (in gaps)
Contiguity required?	Yes, but problem is due to block size being larger than need	Yes, because process needs one large continuous block, but memory is split
Solution	Reduce block size, paging, segmentation	Compaction, paging, segmentation
Effect	Space inside partitions is not used	Large memory request may fail even if total free space is enough

3 a) Explain **paging** in memory management. What problem does it solve compared to continuous allocation?

b) A system uses paging with:

- Logical Address = 12 bits
- Page Size = 256 bytes

(i) Find the **number of pages** in the logical address space.
(ii) If the physical memory is 4 KB, how many frames are available?

a) What is paging and what problem does it solve?

Paging is a memory-management technique that divides both the **logical (process)** address space and **physical** memory into equal fixed-size blocks:

- Logical blocks = **pages** (size = page size).

- Physical blocks = **frames** (same size as a page).

The OS keeps a **page table** for each process that maps each page number → frame number.
A logical address is split into **(page number, offset)**. The page number finds the frame from the page table; the offset is added to the frame base to get the physical address.

What problem does paging solve (vs contiguous allocation)?

- **No need for a single contiguous region** for a process. A process's pages can be scattered across physical memory frames.
- This **eliminates external fragmentation** (no many small unusable holes spread across memory).
- It makes memory allocation flexible and easier (can load pages wherever there is a free frame).
- **Caveat:** paging can cause **internal fragmentation** inside the last page of a process (because process size may not be an exact multiple of page size).

---

b) Given data and calculations

Given:

- Logical address length = **12 bits**.
- Page size = **256 bytes**.
- Physical memory = **4 KB** = 4096 bytes.

Key idea (binary/powers of two):
Page size 256 bytes = 282^828 bytes ⇒ **offset needs 8 bits**.
Logical address is 12 bits total ⇒ remaining bits for page number = 12−8=412 - 8 = 412−8=4 bits.

(i) Number of pages in logical address space

Number of pages = 2(page-number bits)=24=162^{\text{(page-number bits)}} = 2^4 = \mathbf{16}2(page-number bits)=24=16 pages.

(ii) Number of frames in physical memory

Frame size = page size = 256 bytes.
Number of frames = physical memory ÷ frame size = 4096÷256=164096 \div 256 = 164096÷256=16 frames.
So there are **16 frames**.