Deadlocks: A **deadlock** is a situation in which a group of processes are **permanently blocked**, each waiting for a resource held by another process in the group, so **none** of them can proceed. **Example Scenario:** Process A holds Resource X and is waiting for Resource Y. Process B holds Resource Y and is waiting for Resource X. Both wait indefinitely, so the system is deadlocked.

A deadlock can occur if and only if **all** of the following conditions hold simultaneously: **Mutual Exclusion,** At least one resource is non-shareable (only one process can use it at a time). **Hold and Wait,** A process is holding at least one resource and is waiting to acquire additional resources held by other processes. **No Preemption,** Resources cannot be forcibly taken away from a process; they must be released voluntarily once the process is done. **Circular Wait,** There is a circular chain of processes where each process holds at least one resource that the next process in the chain requests.

Resource-Allocation Graph (RAG) process is a dot, resource is box with dot inside, process can request instance of resource with arrow from dot to box, process is holding an instance of resource if arrow from box to dot

Detecting Deadlock via RAG: Single Instance per Resource Type If the graph has a **cycle**, then a deadlock exists. **Multiple Instances per Resource Type:** If the graph has a cycle, it is a **necessary** but **not sufficient** condition for deadlock. You'd need further checks to confirm if the cycle indicates a true deadlock or if there are still free instances that might resolve the cycle.

Deadlock Prevention: Idea: If you **prevent** at least one of the four necessary conditions from ever occurring, you avoid deadlock. **Mutual Exclusion:** Make resources sharable whenever possible. Not feasible for all resources (e.g., printers, certain hardware devices) that inherently require mutual exclusion. **Hold and Wait:** Strategy: Force processes to request all required resources at once, before execution. If a process needs another resource later, it must release everything it holds, then re-request everything. Downside: Low resource utilization and longer wait times if a process holds many resources for a long time. Starvation is possible. **No: Preemption:** Strategy: If a process is waiting for a resource that is held by another process, preempt the resource from the holding process (if possible). Typically applies to CPU or memory, but not always feasible for all resource types (e.g., we cannot forcibly take away an in-progress print job from a printer in the middle of printing). **Circular Wait:** Strategy: Impose a total ordering on resource types Require that processes request resources in ascending order of enumeration. This effectively breaks the circular chain, because it's impossible for a process holding RjR_jRj to request RiR_iRi if i>ji > ji>j. Trade-Off: Each prevention method can reduce concurrency or increase overhead. In practice, systems might not fully prevent deadlocks but rather use avoidance or detection.

Deadlock Avoidance: Goal: Avoid reaching an unsafe state that might lead to deadlock. The system needs prior knowledge of how many resources each process could need. **Safe State vs. Unsafe State:** Safe State**:** There is at least one sequence of all processes in which each process can obtain its maximum resource requirements, run to completion, and release resources for the next process in sequence. A system in a safe state cannot be deadlocked by definition. **Unsafe State**: The system cannot guarantee that every process can finish.
This doesn't mean it's necessarily in deadlock right now, but it's at risk.

MMU: The MMU is a hardware device that performs the run-time mapping of logical (virtual) addresses to physical addresses. When a process issues a memory request at a **logical address**, the MMU translates that into the correct **physical address** in RAM. **Basic Mechanism: Logical Address** (or Virtual Address): Generated by the CPU when a process accesses memory (load/store instructions). **Physical Address**: The actual address in physical RAM. **The MMU often uses relocation registers** (e.g., a **base register**) to translate addresses: Physical Address = Logical Address + Base Register Alternatively, for more advanced schemes (like paging), the MMU consults page tables. **Benefits: Protection**: Ensures processes cannot access memory outside their own range **Relocation**: Processes can be moved around in physical memory without changing the code. **Sharing**: Can share memory segments (e.g., shared libraries) among processes safely.

Dynamic Storage Allocation: First-Fit: Allocate the first hole that is big enough for the process. Search from the beginning of the free list until we find a suitable spot. Pros: Generally fast (search stops on the first adequate hole). Cons: Can leave many small, unusable fragments near the front. **Best-Fit:** Allocate the smallest hole that is big enough. Must search the entire free list to find the best (smallest) hole. Pros: Leaves larger holes unbroken, which might be more useful later. Cons: More searching overhead; also can create many small leftover holes. **Worst-Fit:** Allocate the largest hole. The idea is to leave medium-size holes unbroken and instead break off from the largest hole. Pros: In theory, might avoid creating lots of tiny fragments. Cons: Tends to create smaller holes that might still be too small to be used. **Which Algorithm Is Better?** Empirical studies suggest that First-Fit and Best-Fit are usually better than Worst-Fit in terms of overall memory utilization. First-Fit is often faster in practice because it doesn't require scanning the entire list (it stops as soon as it finds a fit). Best-Fit might yield slightly less fragmentation in some scenarios, but the difference is often small compared to the extra overhead of searching.

External Fragmentation: Occurs when total free memory space is enough to satisfy a request, but it is not contiguous. **Cause**: Processes are loaded and removed from memory in a dynamic manner, creating "holes" between allocated blocks. **Solutions**: Compaction: Shuffle memory contents to coalesce free holes into one contiguous block. This can be expensive and sometimes impossible if addresses are fixed. Use a technique like Paging or Segmentation to avoid external fragmentation. **Internal Fragmentation:** Definition: Occurs when allocated memory may have unused space inside an allocated block. **Cause**: The allocated chunk is bigger than needed, leaving leftover space inside that chunk. **Common with**: Paging, because the smallest unit of allocation is a page, leading to wasted space if a process does not use the entire page. **Trade-Off**: **Contiguous allocation** suffers from **external fragmentation**. **Paged allocation** solves external fragmentation but introduces **internal fragmentation**.

Address Translation: Page Number (p): Logical Address / Page size, **Page Offset:** Logical address mod Page size, **1Kb = 1024 bytes, Physical Address** = **(**Frame Number from Page Table) * (Page Size) + Offset, **Bits to identify a page:** = (# pages) = (2 ^ (#bits)), **Page Size:** = (# bytes = 2 ^ (#bits)), **Bits to identify a frame:** = (# frames) = (2 ^ (#bits)),

(ALL LOG 2^n) can solve above ^^^^

Paging: Paging is a memory-management scheme that eliminates the need for contiguous allocation of physical memory, thereby solving external fragmentation but causing some internal fragmentation. It reduces memory usage as you only need to allocate what is in use and simplifies searching a large table **Basic Concepts:** Divide physical memory into fixed-size blocks called frames (e.g., each frame is 4 KB). Divide logical memory (the process's address space) - p ages. When a process is loaded, its pages are mapped to available frames. A page table keeps track of which frame each page occupies.**Advantages of Paging:** No External Fragmentation: Any free frame can be used for any page. Efficient Memory Use: Processes can be allocated non-contiguous frames, making it easy to fill in "holes." **Disadvantages:** Internal Fragmentation: On average, half a page per process is wasted. Overhead: Need a page table lookup on every memory access. mitigated by a Translation Lookaside Buffer (TLB),

Page-Fault Handling Sequence: Process Accesses a Page marked invalid. **Hardware Trap** to the OS (page fault). **OS Checks**: If the memory reference is invalid the OS terminates the process (segmentation fault). Else, the OS locates the needed page on disk. **OS Finds a Free Frame**: If no free frame is available, it invokes the page replacement algorithm. **OS Issues I/O** to read the needed page into the chosen frame. **OS Updates** the page table to mark the page as valid and record which frame it now occupies. **Restart the Instruction** that caused the page fault (the CPU re-executes the faulting instruction, this time succeeding). **Performance Considerations** Each page fault incurs a significant overhead because of disk I/O (milliseconds vs. nanoseconds for memory).

Page Table Structures: Single-Level Page Table: One table that directly maps page numbers to frame numbers. **Multi-Level / Hierarchical**: Breaks the page table into smaller pieces to handle large address spaces more efficiently. **Inverted Page Table**: One table entry per physical frame, rather than per virtual page, to reduce memory overhead.

Paging Process: MMU -> **Dynamic Allocation**: **(First-Fit** is typically fast and reasonably efficient. **Best-Fit** might minimize wasted space in some cases, but can be slower. **Worst-Fit** is rarely the best in practice.) -> **Fragmentation**: (**External/Internal) -> Paging**

Demand Paging only loading the pages that are needed into memory aves memory and speeds up process start.

Copy on Write (COW): is an optimization used when a process forks. Instead of immediately copying all pages from parent to child, the OS marks the pages as read-only and shared between parent and child. If either process writes to a shared page, a page fault occurs, and then the OS actually copies the page to give each process its own version. **Benefit**: Many pages never get written (e.g., code pages), so they remain shared, saving memory and reducing overhead.

fork() vs. vfork(): fork(): Creates a new child process; parent and child have **separate** memory spaces. Traditionally copies the entire address space, but **with COW** it defers copying until a write occurs. Child typically calls exec() soon after or continues in the same code with some modifications. **vfork()**: A special, more efficient version of fork(). The child runs **in the parent's address space** until it calls exec() or exits. The parent is **suspended** until the child either calls exec() or exits, because they share the same memory. **Risky** if the child modifies variables because changes will be seen by the parent. Used primarily for immediate exec() calls to avoid the overhead of copying page tables.

Page Replacement Algorithms: Goal: Minimize the number of **page faults**.

First-In-First-Out (FIFO): Algorithm: The OS maintains a queue of pages in memory; the **oldest** page is evicted first. **Implementation**: Keep track of the order in which pages were loaded. **Drawback**: **Belady's Anomaly** – More frames can sometimes lead to **more** page faults with FIFO. **Counting Page Faults**: Initialize an empty queue of length = # of frames. For each page reference in the string: If the page is not in the queue (page fault), load it. If the queue is full, remove the oldest page. Increase the page-fault count.

Least Recently Used (LRU): Algorithm: Evict the page that was used **least recently** in the past. **Rationale**: Pages that haven't been used for a while are less likely to be used soon, approximating the "optimal" approach. **Implementation**: Can keep a time-stamp of the last use for each page, or maintain a stack structure. True LRU can be expensive to implement, so many systems use approximations (like reference bits, second-chance, clock algorithm, etc.). **Performance**: Generally good, widely used in real systems (or approximations thereof).

Optimal (MIN): Algorithm: Evict the page that will **not be used for the longest period of time** in the future. **Ideal** approach that yields the **minimum possible** page faults. **Not implementable** in practice (requires knowing the future). **Used** as a theoretical baseline to compare other algorithms.

File: is a logical storage unit that abstracts the physical storage details. Represents a named collection of related information recorded on secondary storage (disk, SSD, etc.). **Attributes: Name:** Human-readable identifier, e.g., `report.docx`. **Identifier (File ID / inode number):** Unique tag (non-human-readable) identifying the file within the file system. **Type:** May indicate whether it's a text file, binary, executable, etc. (some OSes rely on extensions, others store type in metadata). **Location:** Pointer to where the file is located on disk (e.g., block/sector address). **Size:** Current file size (in bytes) and possibly the maximum size if pre-allocated. **Protection (Permissions):** Who can read/write/execute the file (owner, group, others, etc.). **Timestamps:** Creation time, last modified time, last access time. **User Identification:** Owner ID, group ID, etc., for security/audit.

File Operations: Create: Allocate space in the file system, create a file descriptor/FCB/inode, and add an entry to the directory structure. **Open:** Locate the file in the directory, load file metadata (FCB) into a system-wide open file table. A per-process open file table entry is also created pointing to the system-wide entry. **Read:** Read data from file at the current file pointer position, update the pointer accordingly. **Write:** Write data at the current file pointer, update the pointer. If the file expands, more disk blocks may be allocated. **Seek/Reposition:** Move the file pointer to a given location (e.g., `lseek` in UNIX). **Close:** Remove the entry from the per-process open file table. If no more processes have it open, the system-wide entry can be removed. **Delete:** Deallocate all file blocks, remove directory entry.

System-Wide Open File Table: Maintains an entry for each currently open file (by any process). Stores file control block info (location, size, permissions). **Per-Process Open File Table:** Each process has a list of files it has opened. Each entry points to the corresponding system-wide open file entry. Contains the current file pointer, mode (read/write), etc. **Why Two Levels? System-Wide**: Avoid duplicating metadata for each process. **Per-Process**: Tracks individual processes' positions in the file, access modes, etc.

Internal File Structure: Logical View: The file is just a sequence of bytes, lines, or records. **Physical View**: The file is stored in blocks on disk (e.g., 512-byte sectors).

Packing: Files are often **packed** into fixed-size blocks (e.g., 4 KB), leading to **internal fragmentation** if a file's last block is partially filled. Some systems use **extents** or variable-sized blocks to reduce internal fragmentation.

Sequential Access: Most common access pattern. Operations: read next: read the next block/record, advance the file pointer. write next: append at the end, advance the pointer. Advantage: Simple. Good for reading/writing from start to finish. Disadvantage: Not efficient if you need random access to arbitrary locations.

Direct (Random) Access: Allows reading/writing any block of the file in no particular order. File can be viewed as a series of blocks/records indexed by a position. The user can directly jump to block `i` (e.g., `seek(i)`). Advantages: Essential for databases or large files where random access is needed. Disadvantages: Implementation more complex, especially if file blocks are not contiguous.

Directory Structures: The **directory** is a **special file** that maintains entries for all files in a partition or volume. It organizes and provides info about the files. **Single-Level Directory:** All files in the same directory. **Pros**: Simple. **Cons**: Risk of name conflicts (must have unique names), not scalable for many users. **Two-Level Directory:** A separate directory for each user. **Pros**: Avoids name conflicts across users, more structured. **Cons**: Still not very flexible for complex organization.

Tree-Structured Directories: Hierarchical approach (like UNIX, Windows). Each user or system directory can have subdirectories. **Pros**: Natural, flexible. **Cons**: Must manage pathnames (absolute or relative).

Acyclic-Graph & General Graph Directories: Allow **shared subdirectories** or files. **Acyclic-Graph**: No cycles, so no infinite loops in directory references. **General Graph**: Cycles can exist, more complex to handle (e.g., need to prevent infinite loops when traversing).

Boot Control Block: (sometimes called a **boot block** or **partition boot record**) is a small region on a disk partition that: Identifies if the partition contains a file system that is bootable (i.e., an operating system loader). Contains bootstrap code to load the operating system, if applicable. **Location**: Typically the **very first block** of a volume (partition). **If empty**: The volume is not meant to be bootable (or uses a different partition for booting).

Contiguous Allocation Idea: Each file occupies a set of **contiguous** disk blocks. **Metadata**: The directory entry only needs: **Start block** (where the file begins) **Length** (number of blocks the file occupies) **Advantages Fast sequential and direct access**: Because all file blocks are next to each other, reading the file sequentially is very efficient. Direct access is straightforward (just `start_block + offset`). **Simplicity**: Minimal metadata needed (start + length). **Disadvantages:** External Fragmentation, Difficulty in file growth: If a file grows beyond its initial allocation, the OS may need to relocate it entirely or find a new contiguous region.

Linked Allocation: Concept: Each file is a linked list of disk blocks; each block has a pointer to the next block. **Pros**: No external fragmentation, easy to grow. **Cons**: Poor random access performance, overhead for pointers, reliability concerns if a pointer is lost.

Indexed Allocation: Idea: Each file has its own **index block** (or multiple), which is an array of **block pointers**. **Metadata**: The directory entry (or FCB) points to the file's index block. The index block itself lists all the disk blocks used by that file. **Advantages Supports direct access** well: Access block `i` of the file by reading the `i`th pointer in the index block to find the physical location. No external fragmentation. **Disadvantages:** Overhead of the index block: Must keep at least one entire block for the index, even if the file is small. Potential for large index blocks if the file is very big.

Bit Vector (Bitmap): The disk is divided into blocks. We keep a **bit vector** where **each bit** represents the status of a disk block: **1** = block allocated (in use), **0** = block free **Finding the First Free Block**: Scan the bit vector from the start until you find a `0`. The index of that bit is the first free block number. **Pros**: Simple to understand and implement. Fast to find a free block if the OS uses hardware instructions or partial scanning. **Cons**: If the disk is very large, the bit vector can be sizable, though still usually manageable in memory or at least cached in part. **Alternatives Free Space Management: Grouping/Counting**: Store addresses of free blocks in clusters. **Space Maps**: Advanced structures used in some file systems like ZFS.

Spooling (Simultaneous Peripheral Operations On-Line) is a technique where output is written to a disk file rather than sent directly to a device (e.g., a printer). A spooler (background process) eventually sends this disk file to the device.

Why Use Spooling for Printers? Single Access Device: Most printers can only handle one job at a time. **Decoupling**: The printing device is much slower than the CPU. Instead of making a process wait, the process quickly writes its output to a spool file, then continues. **Scheduling**: The OS can manage the queue of print jobs more efficiently

Disk Structure: A **magnetic disk** typically consists of: One or more **platters**, each with two surfaces for storing data. **Tracks**: concentric circles on a platter surface. **Sectors**: subdivisions of each track. A **disk head** that moves radially across the tracks. All tracks with the same radius on different platters form a **cylinder**.

Disk Mechanism: Seek Time: The time for the read/write head to move to the correct track (dominant factor in I/O). **Rotational Latency**: The time for the desired sector to rotate under the read/write head. **Transfer Time**: The time to actually read or write the data after the head is in position. **Key Goal** of disk scheduling is to minimize the total **seek time** by intelligently ordering the requests.

FCFS (First-Come, First-Served): Algorithm: Service the requests in the **order** they arrive (FIFO queue). **Pros**: Simple to implement and fair (no request is starved). **Cons**: May result in large total head movement if requests are in random order (no optimization of seek time).

SSTF (Shortest Seek Time First): Algorithm: Always service the **closest** request (in terms of cylinder distance) from the current head position next. **Pros**: Minimizes the immediate seek time, often better average performance than FCFS. **Cons**: Can lead to **starvation** for requests that are far from the current head position if many new closer requests keep arriving.

SCAN (Elevator Algorithm): Algorithm: The disk arm moves in **one direction** (say, from outer to inner cylinders), servicing all requests on the way. When it reaches the last cylinder in that direction (or the last request), it **reverses** direction and services requests in the opposite direction. **Pros**: Generally better performance than SSTF in heavy loads, because it systematically sweeps across the disk. Avoids some starvation by ensuring that every cylinder is visited regularly.

C-SCAN (Circular SCAN): Algorithm: Moves the head in **one direction** (e.g., from the lowest cylinder to the highest). Upon reaching the last cylinder, it **immediately returns** to the start cylinder **without servicing any requests on the return trip**. Then it repeats. **Pros**: Provides a more **uniform wait time** because each cylinder gets serviced in a predictable cycle. Less variation in response times than SCAN, because it doesn't handle requests on the "reverse" path.

Context Switch: Definition: The act of the OS switching the CPU from one process/thread to another. **Purpose**: Enables multitasking by allowing multiple processes or threads to share CPU(s). **Advantages**: Improves CPU utilization (while one process waits for I/O, another can run). Allows concurrency and parallelism (especially with multiple cores). **Disadvantages**: **Overhead**: Saving/loading registers, updating the PCB (Process Control Block) or TCB (Thread Control Block). Frequent context switches can degrade performance.

Process vs. Thread Context Switch: Switching between **threads** in the same process is usually cheaper than switching between **processes** (since threads share the same address space).

Critical Section: A section of code where a process (or thread) **accesses shared data** (e.g., global variables, shared memory, files). **Purpose**: Only **one** thread/process should execute in its critical section at a time to prevent **race conditions**. **Mutual Exclusion**: Ensure that if one thread is in the critical section, no others can enter. **Progress**: If no thread is in the critical section, any thread that wants to enter should be able to do so eventually. **Bounded Waiting**: A limit on how many times other threads can enter their critical sections before a waiting thread gets a turn. Common synchronization tools: **Mutexes** (locks), **Semaphores, Monitors** (in high-level languages)

Bankers Algo: Need = Max – Allocation. New Available = Old Available + Allocation[T#]. If any of these are need>available then unsafe if not must ensure that all tests pass to be safe.

Let Z= starting file address (first physical block).: Contiguous Allocation: Steps: Divide the logical address by 512. Let X=⌊Logical Address/512⌋, Y = {Logical Address} mod 512. Physical Block Number = Z+X Displacement within the block = Y **Linked Allocation:** Steps: Divide the logical address by 511. Let X=⌊Logical Address/511⌋, Y = {Logical Address} mod 511. Traverse the linked list for X+1X + 1X+1 blocks. Displacement within the block = Y +1 **Indexed Allocation:** Steps: Divide the logical address by 512. Let X=⌊Logical Address/512⌋, Y = {Logical Address} mod 512. Physical Block Number = X Displacement within the block = Y. Will always be 1 read for the index block + 1 read for the desired data block; if the index block isn't already in memory so X+1

Optimal replacement below VVVV

Ref	Frames (after)	Page Fault?	Evicted?	Explanation (short)
7	[7, –, –]	Yes (1)	–	Fill empty frame.
2	[7, 2, –]	Yes (2)	–	Fill empty frame.
3	[7, 2, 3]	Yes (3)	–	Fill last empty frame.
1	[1, 2, 3]	Yes (4)	7	All frames full → check future use: 7 is used at ref #10, 2 at #4, 3 at #6 → 7 is farthest.
2	[1, 2, 3]	No	–	2 is in frames.

Ref	Frames (after)	Page Fault?	Evicted?	Explanation (short)
7	[7, –, –]	Yes (1)	–	Empty frame used.
2	[7, 2, –]	Yes (2)	–	Another empty frame used.
3	[7, 2, 3]	Yes (3)	–	Last empty frame used.
1	[1, 2, 3]	Yes (4)	7	Evict LRU (7 was used earliest).
2	[1, 2, 3]	No	–	2 is in frames; update recency.
5	[1, 2, 5]	Yes (5)	3	Evict LRU (3).

LRU ^^^^

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Method	1	2	3	4	5	6	7	8	9	10	11	12	13	Total
FCFS	2150	2069	1212	2296	2900	544	1618	356	1523	4965	3681			13011
SSTF	2150	2069	2296	2800	3681	4965	1618	1523	1212	544	356			7586
SCAN	2150	2296	2800	3681	4965	4999	2069	1618	1523	1212	544	356		7492
C-SCAN	2150	2296	2800	3681	4965	4999	0	356	544	1212	1523	1618	2069	9917
LOOK	2150	2296	2800	3681	4965	2069	1618	1523	1212	544	356			7424
C-LOOK	2150	2296	2800	3681	4965	356	544	1212	1523	1618	2069			9137

EAT = (1 − p) × 200 + p × 8ms
= (1 − p) × 200 + p × 8, 000, 000
= 200 + p × 7, 999, 800

1. If one access out of 1,000 causes a page fault, then

p = 0.001
EAT = 8, 199.8ns = 8.2μs

This is a slowdown by a factor of $\frac{8.2\mu s}{200ns} = 40$!!

FCFS is the queue order others show manipulation ^^^^

EAT = (1 − p) × memory access
+p × page fault time