Operating Systems Notes

Virtualization, Concurrency, and Persistence [3 Easy Pieces]

Jaskirat Singh Maskeen

Aug-Nov 2025

Abstract

These notes were compiled during the Aug-Nov 2025 offering of the CS330: Operating Systems course, based on *Operating Systems: Three Easy Pieces (OSTEP)*.

The aim is to present Virtualization, Concurrency, and Persistence in a clear, and consise manner, making it suitable for quick revision before exams. Or to be used as a cheatsheet, if the professor allows (Hello Abhishek Sir :D) In case of any additions or errors (typos or formatting), please feel free to reach out to me at `jaskirat.maskeen@iitgn.ac.in`. Also, special thanks to Abhinav Khot, for helping me proofread these notes.

To get the latest version, visit: https://jsmaskeen.github.io/blog/2025/os-notes/. Last Updated: December 9, 2025

I hope this helps future readers ¯_(ツ)_/¯.

Table of Contents

Introduction to Operating Systems

The Von Neumann Model

The processor fetches an instruction from memory, then decodes it, and executes it. This is the basis of the **Von Neumann model**. The hardware is responsible for executing these instructions, managing the flow between fetching, decoding, and execution.

Operating System Basics

The **Operating System (OS)** is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner. The

Virtualization is the process of taking a physical resource and transforming it into a more general, easy-to-use version of itself. The goal is for each program to think it has all the resources required for it exclusively (e.g., its own CPU or memory).

The OS provides **system calls** which are, well-defined interfaces that allow user programs to request services from the operating system. System calls act as the gateway between user programs and the hardware, enabling controlled access to resources such as files, memory, and devices. Because the OS manages and arbitrates access to these resources, it is often referred to as a **Resource Manager**.

CPU Virtualization Concepts

- `spin()`: A function (often used in OSTEP examples) that repeatedly checks the time and returns once it has run for a second.
- **Policy:** When multiple programs are active, the question of "which should run next?" is answered by the **policy** of the OS (e.g., Scheduling).
- **Process Identifier (PID):** A unique identifier assigned to every running process.

Memory Virtualization

Memory is effectively an array of bytes.

- **To read memory:** One must specify an address to access the data stored there.
- **To write (or update) memory:** One must specify the address and the data to be written.

Address Spaces: Each process accesses its own private **virtual address space** (often just called its address space). The OS maps this virtual space onto the physical memory of the machine.

Concurrency

Programs can create threads using `pthread_create()`. This introduces concurrency issues, specifically regarding **atomicity**.

Example: The statement `x++;` is technically three instructions, not one:

1. $$\text{Load } x \rightarrow \text{register/accumulator}$$
2. $$\text{Increment register}$$
3. $$\text{Store register} \rightarrow x$$

- **Atomic:** An operation is atomic if it completes entirely without any possibility of interruption or interference from other operations. Atomicity ensures that the operation appears indivisible, so no other thread or process can observe it in a partially completed state.
- **Individually:** Each of these three hardware instructions is atomic (the hardware executes them one by one).
- **As a group:** The statement `x++;` is **not atomic**. A context switch can occur *between* instructions 1 and 2, or 2 and 3, leading to concurrency bugs.

OS Design Goals

- Provide high performance.
- Minimize overheads. Overheads arise in two forms:
  - **Extra time** (more instructions).
  - **Extra space** (in memory or on disk).
- Provide protection between applications, as well as between the OS and applications.
- Ensure **isolation** of processes.

Limited Direct Execution & System Calls

The key difference between a **system call** and a standard procedure call is that a system call transfers control (jumps) into the OS while simultaneously raising the hardware privilege level.

User Mode vs. Kernel Mode

- **User Mode:** User applications run here. The hardware restricts what applications can do. For example, an application in user mode cannot typically initiate an I/O request to the disk, access physical memory pages directly, or send a packet on the network.
- **Kernel Mode:** When the privilege level is raised, the OS has full access to the hardware (e.g., initiating I/O requests or allocating memory).

The Trap Mechanism

1. When a system call is initiated (usually through a special hardware instruction called a **trap**), the hardware detects this event and transfers control to a pre-specified **trap handler** (that the OS set up previously).
2. Simultaneously, the hardware raises the privilege level to **Kernel Mode**, ensuring the OS has full access to system resources.
3. When the OS is done servicing the request, the hardware helps pass control back to the user via a special **return-from-trap** instruction.
4. This instruction, executed by the hardware, reverts the system to **User Mode** and passes control back to where the application left off.

   Note: It is not necessary to execute the same program after returning from a syscall (the OS may switch to a different process).

Virtualization: The Process

1. The Process Abstraction

A **process** is an abstraction provided by the OS for a running program.

Time Sharing (CPU)

- **The Goal:** The OS must provide the illusion of many CPUs to the user, even though typically only one program can run on one CPU at a time.
- **The Mechanism:** The OS runs one program for a short while, then switches to another in fixed time slices. This technique is called **Time Sharing**.
  - **Consequence:** With more CPU sharing (more processes), the time each individual program takes to complete increases (assuming equal priority and duration).

  Comparison: The counterpart to time sharing is **Space Sharing**. Disk space is naturally space-shared; once a block is assigned to a file, it remains assigned until deleted. CPU is time-shared.

Mechanisms

1. **Context Switch:** The act of stopping one program and running another.

2. **Scheduling Policy:** Algorithms (policies) used by the OS to decide which process to run next (e.g., SJF).

2. Process Memory & Creation

The memory that a process can address is called its **Address Space**. It is considered part of the process itself.

Process State Requirements:

To track a process, the OS (specifically the `proc` struct) requires the Program Counter (PC), the Stack Pointer (top of stack), the Kernel Stack (kstack), and I/O information.

How is a process created?

1. **Load Code:** The OS loads the code and static variables into the address space. Modern OSs perform this **lazily** (loading code only when needed).
2. **Allocate Stack:** The runtime stack is reserved in the address space for local variables and return addresses.
3. **Allocate Heap:** Memory is reserved for the heap (for dynamic memory allocation/deallocation).
4. **I/O Setup:** The OS loads file descriptors (standard input/output/error).
5. **Start:** The OS jumps to the entry point, usually `main()`.

3. Process States

- **New:** The process is being set up by the OS.
- **Running:** The process is currently executing instructions.
- **Ready (Runnable):** The process is ready to run but waiting for the OS to schedule it.
- **Sleeping (Blocked):** The process is waiting for an event, such as I/O completion.
- **Zombie (Terminated):** The process has finished but has not been cleaned up yet.

The "Zombie" State

A process enters the **Zombie** state when it calls `exit()`.

- **Why?** It allows the parent process to examine the return code of the child to see if it executed successfully.
- **Cleanup:**
    1. When a process exits, its resources (memory, FDs) are freed, but the entry (PID, exit status) remains in the process table.
    2. The parent must call `wait()` to collect this status. Once done, the OS removes the zombie entry completely.
    3. If the parent exits *without* waiting, the zombie is reassigned to `init` (PID 1), which automatically waits for and cleans up orphaned zombies.

4. Data Structures

- **Process List:** The OS maintains a list to keep track of all processes.

- **Process Control Block (PCB):** The structure holding process information (like the `proc` struct in xv6).
- **Register Context:** Holds the contents of the process's registers when it is not running.

5. The Process API (System Calls)

`fork()`

Creates a new process by making a copy of the parent.

- **Return Values:**
  - Returns `0` to the **child** process.
  - Returns the `child PID` to the **parent** process.
  - Returns `< 0` if the fork failed.
- **Behavior:** It makes a **deep copy** of the stack, heap, and code. The child gets its own address space.
- **File Descriptors:** Open file descriptors are **shared**.
  - However, if the child closes a file descriptor, the parent's copy remains valid (and vice versa).

`wait()`

Used by the parent to wait for the child to finish.

- **Blocking:** `wait()` blocks the parent until the child calls `exit()`.
- **Non-Blocking:** `waitpid(-1, &status, WNOHANG)` waits for any child but returns immediately (`0`) if no child has exited, rather than blocking.

`exec()`

Given an executable name and arguments, `exec()` loads code and static data from that executable and **overwrites** the current process's code segment.

- The heap and stack are re-initialized.
- **Retains File Descriptors:** The new program inherits the open file descriptors of the calling process.
- **No Return:** A successful `exec()` **never returns** (because the code that called it has been overwritten).

Shell Redirection Example

How does `prompt> wc p3.c > newfile.txt` work? 1. The shell calls `fork()` to create a child. 2. **Before** calling `exec()`, the child closes Standard Output (`STDOUT`) and opens `newfile.txt`. 3. The child calls `exec()`. 4. The program `wc` writes to `STDOUT`, but since that descriptor now points to the file, output goes to `newfile.txt`.

`kill()`

The `kill()` system call is used to send **signals** to a process (e.g., sleep, die). Signals are a method of Inter-Process Communication (IPC).

Limited Direct Execution (Mechanism)

1. The Basic Approach

The strategy is **Limited Direct Execution**.

- **Direct Execution:** Just load the program and run it directly on the CPU.
- **The Problem:** What if the program needs to perform a restricted action (like I/O)? We cannot give an unprotected program full access to hardware resources.

2. Solution: User Mode vs. Kernel Mode

To solve this, the system distinguishes between two execution modes:

1. **User Mode:** Applications run here. They do *not* have full access to hardware resources.
   - *Constraint:* If a process attempts a privileged operation (like I/O) in user mode, the OS will kill the process.
2. **Kernel Mode:** The OS runs here. It has access to the full resources of the machine.

3. System Calls & Traps

To perform a privileged operation, a user program must perform a **System Call** (syscall).

The Mechanism

1. **Trap Instruction:** The program executes a special `trap` instruction.
   - It simultaneously **jumps into the kernel** and **raises the privilege level** to kernel mode.
2. **Hardware Actions:** The processor pushes the **Program Counter (PC)**, **Flags**, and other registers onto the per-process **Kernel Stack**.
3. **Execution:** The OS performs the required privileged operations.
4. **Return-from-Trap:** When finished, the OS calls this special instruction.
   - It pops the values (PC, flags) off the stack.
   - It simultaneously returns to the user program and **reduces the privilege level** back to user mode.

   Note: The OS may not necessarily return to the *calling* process; it might switch to another.

The Trap Table

How does the hardware know where to jump during a trap?

- **Boot Time:** The OS sets up a **Trap Table** and informs the hardware of the locations of these **Trap Handlers**.

4. Regaining Control (The Switch)

If a process is running on the CPU, the OS is *not* running. How does the OS regain control to switch processes?

Approach 1: Cooperative (Wait for System Call)

The OS trusts programs to behave reasonably.

- The process voluntarily gives up the CPU via a `yield` system call.
- Or, the process does something illegal (misbehaves), triggering a trap, which hands control to the OS to terminate it.

Approach 2: Preemptive (Timer Interrupt)

The OS does not trust the process.

- **Timer Device:** Raises an interrupt every few milliseconds.
- **Action:** The current process is halted, and a pre-configured interrupt handler in the OS runs.

5. Context Switching

When switching from Process A to Process B, the OS performs a **Context Switch**.

1. **Save State:** The OS saves a few register values for the currently running process and pushes them to its kernel stack (kstack).
2. **Restore State:** The OS restores the saved registers for the next process.

Distinction: Interrupt vs. Context Switch

There is a subtle difference in *what* gets saved and *who* saves it:

Scenario	Who Saves?	What is Saved?	Where?
Timer Interrupt	**Hardware**	User Registers (Current)	Running Process's **Kernel Stack**
OS Switch Decision	**OS Software**	Kernel Registers (OS Context)	Process A's **PCB / Proc Structure**

- **Interrupt:** The hardware automatically saves user registers to the kernel stack so the OS code can run.
- **Switch:** When the OS decides to switch processes (scheduler), it explicitly saves its own kernel-mode registers into the PCB of Process A, then loads the saved state of Process B.

6. Concurrency Control

Problem: What if an interrupt occurs while the OS is already handling an interrupt?

Solution: The OS often **disables interrupts** during interrupt processing. This ensures that when one interrupt is being handled, no other one will be delivered to the CPU, preventing race conditions. If another interrupt occurs while interrupts are disabled, most hardware will either queue (or latch) the interrupt so it can be delivered once interrupts are re-enabled, or simply remember that an interrupt occurred and deliver it later. The exact behavior depends on the hardware architecture.

CPU Scheduling

1. Scheduling Metrics

To evaluate scheduling policies, we use two primary metrics:

Turnaround Time (Performance)

Defined as the time at which the job completes minus the time at which the job arrived in the system.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Response Time (Fairness)

Defined as the time from when the job arrives in a system to the first time it is scheduled.

$$T_{response} = T_{first\_run} - T_{arrival}$$

2. Non-Preemptive Algorithms

FIFO (First In, First Out) / FCFS

- **Mechanism:** Simple and easy to implement.
- **Preemption:** None.
- **Issue (Convoy Effect):** A number of relatively short potential consumers of a resource get queued behind a heavyweight resource consumer. This hurts average turnaround time significantly.

SJF (Shortest Job First)

- **Mechanism:** Given that all jobs arrive at the same time, the shortest job is run first.
- **Preemption:** None.

Proof of Optimality [Using Exchange Argument (DSA2!!)] (for simultaneous arrival)
Goal: Minimize Average Waiting Time (WT).

$$\text{Average } WT = \frac{1}{n}\sum_{i=1}^{n} WT_i$$

1. Suppose we have n processes with burst times $B_1, B_2, ...B_n$ arriving at $t = 0$.
2. Take any scheduling order that is **not** sorted by burst time.
3. Find two adjacent processes P_i, P_{i+1} where the longer process precedes the shorter one ($B_i > B_{i+1}$).
4. **Swap them.**
    - The waiting time of P_i increases by B_{i+1}.
    - The waiting time of P_{i+1} decreases by B_i.

- Since $B_i > B_{i+1}$, the net waiting time **decreases**.

5. Repeating this swap sorts the processes in ascending order of burst time (SJF), proving it is optimal for minimizing waiting time.

3. Preemptive Algorithms

STCF (Shortest Time to Completion First) / PSJF

- **Also known as:** Preemptive Shortest Job First (PSJF).
- **Mechanism:** Any time a new job enters the system, the scheduler determines which of the remaining jobs (including the new one) has the least time left, and schedules that one.

Round Robin (RR)

- **Mechanism:** Run each job for a fixed time slice (scheduling quantum).
- **Goal:** Optimize **Response Time** (Fairness).

The Time Slice Trade-off:

- **Shorter Slice:** Better response time.
- **Longer Slice:** Better overall system efficiency (amortizes switching costs).
- **The Cost:** Reducing slice time induces the cost of context switching. We need to make the slice long enough to amortize this cost without making the system unresponsive.

Context Switch Overheads: Switching is not free. It involves:

1. Saving register states.
2. Performance penalties from losing the **Cache**, **TLB**, and **Branch Predictors**.

4. Handling I/O

How do schedulers incorporate I/O?

- The scheduler treats each CPU burst as a **separate job**.
- When a process initiates I/O, it blocks, and the scheduler picks another job. When I/O completes, the process becomes ready again with a new (usually short) CPU burst.

Multi-Level Feedback Queue (MLFQ)

1. The Goals

The MLFQ scheduler tries to address two conflicting goals simultaneously:

1. **Optimize Turnaround Time:** This is usually done by running short jobs first (like SJF).
2. **Minimize Response Time:** This is critical for interactive users (like Round Robin).

2. Basic Structure

MLFQ has a fixed number of queues, each with its own priority level.

- A job that is ready to run will always be on a single queue.

- **Dynamic Priority:** Rather than giving a fixed priority, MLFQ varies the priority of a job based on its observed behavior (using the history of the job to predict its future).
  - **Interactive Jobs:** If a job repeatedly gives up the CPU (waiting for I/O), it remains at a high priority.
  - **CPU-bound Jobs:** If a job uses the CPU intensively, it moves down in priority.

3. The Basic Rules (Attempt 1)

Priority Rules

- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A and B run in Round Robin (RR).

Placement Rule

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).

Behavior Rules (Naive Approach)

- **Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (moves down one queue).
  - *Note:* Queues at lower priorities may have longer time slices.
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the same priority level.

Logic: Approximating SJF

MLFQ approximates **Shortest Job First (SJF)** without knowing the length of the job in advance. 1. It assumes a new job might be short, giving it high priority. 2. If it is indeed short, it runs quickly and completes. 3. If it is not short, it slowly moves down the queues, proving itself to be a long-running, batch-like process.

4. Problems with the Basic Approach

1. **Starvation:** If too many interactive jobs enter the system, they will monopolize the top queue. Long-running jobs (at the bottom) will never get CPU time.
2. **Gaming the Scheduler:** A malicious program can "trick" the scheduler.
   - *The Exploit:* The program runs for 99% of the time slice, then voluntarily gives up the CPU (yields) right before the slice ends.
   - *The Result:* Under **Rule 4b**, its priority is never reduced, allowing it to hog the CPU at the highest priority level.

5. Refinements (The Solutions)

Solution to Starvation: Priority Boost

- **Rule 5:** After some time period S, move **all** the jobs in the system to the topmost queue.

- This resets the system and guarantees that low-priority jobs eventually get a turn.
- *Tuning S:* If S is too high, long-running jobs starve. If S is too low, short-running jobs might not get a proper share of the CPU.

Solution to Gaming: Better Accounting

We must account for how much CPU a process uses *in total* at a given priority level, not just in a single burst.

- **Rule 4 [Updated]:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Proportional Share Scheduling

1. The Concept

The idea is **not** to optimize for turnaround or response time directly, but to guarantee that each job obtains a certain percentage of CPU time.

2. Lottery Scheduling

Mechanism:

- Assign **tickets** to processes proportional to the percentage of the resource they should receive.
- At each time tick (scheduling quantum), hold a lottery.
- Whichever program holds the winning ticket is scheduled.

Characteristics:

- **Probabilistic:** It achieves the goal probabilistically, not deterministically.
- **Implementation:** Uses a Random Number Generator (RNG) to hold the lottery (simple to implement).
- **Convergence:** Over a long period, the share of CPU for each program stabilizes to the desired percentage.

Ticket Manipulation Mechanisms

1. **Ticket Currency:** Allows a user with a set of tickets to allocate them among their own processes in any denomination. The OS converts this to the global currency.
   - *Example:* User has 100 global tickets. They run 3 processes and give them 100, 300, and 500 "local" tickets. The OS calculates the global share (e.g., $100/900 \times 100$ global tickets).
2. **Ticket Transfer:** A process can transfer its tickets to another process temporarily.
   - *Use Case:* Client/Server. A client sends a request to a server and "hands over" its tickets so the server runs faster to complete that specific request.
3. **Ticket Inflation:** A process can temporarily increase its own number of tickets to run more.
   - *Requirement:* This only works in a non-competitive scenario where processes trust each other (no one is greedy).

3. Stride Scheduling (Deterministic Lottery)

To avoid the randomness of Lottery scheduling, Stride scheduling is used.

Algorithm:

1. **Calculate Stride:** Divide a very large number (e.g., 10,000) by each process's ticket value.

$$\text{Stride} = \frac{\text{Large Constant}}{\text{Tickets}}$$

2. **Pass Value:** Each process has a `pass` value, initially 0.
3. **Schedule:** Run the process with the **lowest** pass value.
4. **Update:** After running, increment the process's pass value by its stride.

$$\text{pass} = \text{pass} + \text{stride}$$

What if a new process arrives? When a new process enters the system, its stride is calculated based on its ticket value, and its pass value is typically initialized to the current minimum pass value among all processes (not zero). This prevents the need to reset all strides or pass values to zero and ensures fairness. Existing processes keep their current pass values.

Comparison:

- Why prefer Lottery over Stride? Lottery requires **no global state** (like storing pass values for every process). It is easier to add new processes without calculating appropriate pass values.

4. Linux CFS (Completely Fair Scheduler)

CFS implements proportional share principles efficiently.

Basic Mechanism:

- There is no fixed time slice.
- **vruntime (Virtual Runtime):** Each process accumulates `vruntime` as it runs.
- **Scheduling Decision:** CFS always picks the process with the **lowest** `vruntime`.

Determination of Time Slice

CFS uses a target latency window called `sched_latency` (e.g., 48ms) to determine how long processes should run before considering a switch.

1. **Calculation:** CFS divides `sched_latency` by the number of processes (n) to determine the per-process time slice.

$$\text{time\_slice} = \frac{\text{sched\_latency}}{n}$$

   - *Example:* 4 processes, `sched_latency` = 48ms → Time slice = 12ms.
   - *Dynamic Adjustment:* If 2 processes finish, the remaining 2 get $48/2 = 24$ms each.

2. **Minimum Granularity:** If there are too many processes, the calculated time slice becomes too small (causing context switch overhead).
    - CFS enforces `min_granularity` (e.g., 6ms). The time slice will never drop below this value.

Weighting (Nice Values)

Users can assign priority using **nice** values.

- **Range:** -20 (Highest Priority) to +19 (Lowest Priority). Default is 0.
- **Mapping:** Each nice value is mapped to a geometric weight.

Formulas:

The time slice for process k is proportional to its weight:

$$\text{timeslice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \times \text{sched\_latency}$$

The `vruntime` accumulation is scaled inversely by weight (higher weight = slower vruntime growth = runs more often):

$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \times \text{runtime}_i$$

Data Structure

CFS uses a **Red-Black Tree** to store running processes, ordered by `vruntime`, allowing efficient retrieval of the next process to run.

Multiprocessor Scheduling

1. Background: Multiprocessor Hardware

To understand scheduling on multiple CPUs, we must understand the hardware differences compared to a single CPU.

- **Caches:** In a multiprocessor system, each CPU has its own cache.
- **Locality:** Caches rely on two types of locality:
    - **Temporal Locality:** Currently accessed info might be accessed again soon.
    - **Spatial Locality:** The next access might be close in memory to the currently accessed one.

The Cache Coherence Problem

Because each CPU has its own cache, updates to memory on one CPU might not be immediately visible to others, leading to the problem of **Cache Coherence**.

Cache Affinity

When a program runs on a specific CPU, it builds up a significant amount of state in that CPU's caches and TLBs.

- **Benefit:** Next time the process runs, it is favorable to run it on the *same* CPU to leverage this cached state.
- **Cost:** If moved to a new CPU, execution will be slower because the cache is "cold" (data must be reloaded).

2. Approach 1: SQMS (Single Queue Multiprocessor Scheduling)

Mechanism:

- Maintain a single global queue of ready jobs.
- When a CPU is available, it pops the next job from this queue.

Issues:

1. **Synchronization:** We need locks to ensure the global queue is accessed safely, which limits scalability.
2. **Cache Affinity:** Jobs effectively bounce randomly between CPUs, losing the benefits of cache affinity.
    - *Fix:* SQMS implementations often use **affinity mechanisms** to try to keep processes on the CPUs where they started.

3. Approach 2: MQMS (Multi-Queue Multiprocessor Scheduling)

Mechanism:

- Each processor has its own private queue.
- When a job enters the system, it is assigned to a particular queue (using a load balancer) and remains there until completion.
- **Example:** With 2 processors (P_1, P_2) and jobs A, B, C, D:
    - P_1 runs A and B in Round Robin.
    - P_2 runs C and D in Round Robin.

Pros & Cons:

- **Pro:** It is more scalable (less lock contention) and naturally respects cache affinity.
- **Con:** It suffers from **Load Imbalance**.
    - *Scenario:* If A and B finish, P_1 sits idle while P_2 is still busy with C and D.

4. Solving Load Imbalance

To fix the imbalance in MQMS, we use **Work Migration**.

Work Stealing

This is a common technique to implement migration.

- **Mechanism:** A "source" queue (which is empty or low on work) looks at "target" queues (which are full).
- **Action:** The source attempts to "steal" jobs from the target to balance the load.

Address Spaces: The Concept

1. Motivation

As time-sharing systems became more popular, the demand to keep multiple programs in memory simultaneously increased. This introduced critical requirements:

- **Isolation:** We need to isolate programs from each other.
- **Protection:** We must prevent a program from altering the code or data of another program or the operating system kernel itself.

2. The Abstraction

To achieve this, the OS creates an abstraction of physical memory called the **Address Space**.

- **Definition:** It is a contiguous block of memory sized to fit what the running program needs.
- **Contents:** The address space contains all the memory state of the running program: its code, stack, and heap.

3. Memory Layout

The typical layout of a process's address space is arranged as follows:

1. **Code:** The program instructions are located at the bottom of the address space (starting at address `0x0`).
2. **Heap:** Grows positively (upwards) from the code segment.
3. **Stack:** The top of the stack is located at the very end of the address space and grows negatively (downwards).

   Efficiency: The stack and heap grow in opposite directions to allow them to share the free space in the middle efficiently.

4. Goals of Virtual Memory

The virtual memory system should be implemented with three key goals:

1. **Transparency (Invisibility):** The implementation should be invisible to the running program. The program should behave as if it has its own private physical memory.
2. **Efficiency:** The assignment of memory should be efficient to minimize:
   - **Space Wastage:** Reducing fragmentation.
   - **Time Overhead:** Minimizing the time required to look up values.
3. **Protection:** The OS must ensure protection, enabling isolation between processes and protecting the OS itself from errant processes.

Memory API

1. Types of Memory

In C programming, there are two primary types of memory allocation:

Stack Memory

- **Management (in C):** Allocations and deallocations are managed implicitly by the compiler.
- **Terminology:** Often referred to as **automatic memory**.
- **Note:** In other languages (e.g., C++), stack allocation is still managed by the compiler, but heap allocation can also be managed by the compiler or standard library (e.g., using `std::vector`).

Heap Memory

- **Management (in C):** Allocation and deallocation are handled explicitly by the programmer.
- **Note:** In languages like C++, heap allocation can be managed by the compiler or standard library (e.g., containers such as `std::vector` automatically manage heap memory).

2. Allocation: `malloc()`

To allocate memory on the heap, we use `malloc()`.

```
int *x = (int *) malloc(sizeof(int));
```

Memory Layout:

In the line of code above:

- The pointer variable `x` itself is allocated on the **stack**.
- The actual integer memory it points to is allocated on the **heap**.

Details:

- **`sizeof()` Operator:** This is a **compile-time operator**, not a function call. It determines the size of the type at compile time.
- **Casting:** Casting the result of `malloc` (e.g., `(int *)`) does not accomplish anything functional; it is purely semantics (and often unnecessary in modern C).

3. Deallocation: `free()`

To release heap memory, we use `free()`.

- **Syntax:** `free(p)` frees the memory pointed to by `p`.
- **Mechanism:** The memory manager tracks the size of the allocation internally, so it knows exactly how much memory to free.

4. Common Errors

Managing heap memory manually introduces several potential bugs:

1. **Memory Leak:** Occurs when the programmer forgets to free memory.
2. **Dangling Pointer:** This is a pointer that continues to reference a memory location after that memory has been deallocated (freed). Accessing or dereferencing a dangling pointer can result in undefined behavior, including crashes, data corruption, or security vulnerabilities, because the memory may now be used by other parts of the program or the operating system.
3. **Double Free:** Occurs when `free()` is called repeatedly on the same pointer. The result of this operation is **undefined**.

Address Translation & Dynamic Relocation

1. The Basic Mechanism

Hardware-based address translation converts a Virtual Address (VA) into a Physical Address (PA).

Base and Bounds (Dynamic Relocation)

To implement this efficiently, we use a technique called **Dynamic Relocation**, relying on two hardware registers within each CPU (part of the **Memory Management Unit (MMU)**).

1. **Base Register:** Defines the start of the physical memory region.
2. **Bounds (Limit) Register:** Defines the size (or end limit) of the region.

The Translation Formula:

$$PA = VA + \text{base}$$

- **Constraint:** The access is valid only if provided that $VA + \text{base} < \text{bounds}$ (or strictly within the size limit).
- **Note:** The bounds register may hold the physical end address *or* the size of the address space.

2. Hardware Requirements

To support this, the hardware must provide:

1. **Privileged Instructions:** Special instructions to modify the base and bounds registers.
    - These allow the OS to change values when different processes run.
    - **Constraint:** These instructions are privileged and can **only** be run in **Kernel Mode**.
2. **Exception Handling:** The CPU must be able to generate an exception if an address is out of bounds, invoking a specific handler.

3. OS Responsibilities

The OS manages memory using a structure called the **Free List**, which stores which ranges of physical memory are currently free.

Lifecycle Management

1. **Process Creation:** The OS must search the free list to find space to allot the memory range for the new process.
2. **Context Switch:** Because the hardware only has *one* pair of base and bounds registers, the OS must save the current values into the **Process Control Block (PCB)** and restore them when the process runs again.
3. **Process Exit:** The OS must clean/reclaim the memory once the process exits.
4. **Relocation:** The OS can move an address space from one base-bounds range to another while the process is not scheduled.

4. Issues with Base and Bounds

Internal Fragmentation

Since the address space is allocated as a contiguous unit, the space inside the region (specifically between the heap and stack) is often not used. This wasted space inside the allocated block is called **Internal Fragmentation**.

External Fragmentation

As processes are allocated and freed, physical memory can develop "little holes" of free space that are too small to be useful. This is called **External Fragmentation**.

Segmentation

1. The General Concept

Segmentation is essentially **generalized base and bounds**. Instead of one pair of base and bounds registers for the entire process, we have multiple pairs—one for each logical segment of the address space:

- Code
- Stack
- Heap

2. Explicit Approach (Address Translation)

In the explicit approach, the hardware uses specific bits of the Virtual Address (VA) to determine which segment is being accessed.

Example: 14-bit Virtual Address

- **Top 2 bits:** Segment ID.
- **Bottom 12 bits:** Offset.

Translation Logic:

The hardware takes the offset and adds it to (or subtracts it from) the base register of the selected segment to obtain the Physical Address (PA).

Segment Register Table

The hardware maintains a structure defining the bounds and growth direction for each segment.

Segment Bits	Segment	Base	Size (Bound)	Grows Positive?
`00`	Code	32K	2K	1 (Yes)
`01`	Heap	34K	3K	1 (Yes)
`11`	Stack	28K	2K	0 (No)

Note: Accessing an address outside these bounds leads to a **Segmentation Fault**.

3. Issues and Limitations

1. **Wasted Segment Space:** With 2 bits for segments (4 combinations), if we only use 3 segments (Code, Heap, Stack), one segment (e.g., `10`) goes unused.
2. **Limited Size:** The maximum effective segment size is reduced (chopped), as the total address space is divided by 4.

4. Advanced Features

- **Code Sharing:** We can implement code sharing by adding **protection bits** (e.g., Read-Only for code segments).
- **Granularity:**
  - **Coarse-Grained:** Small number of segments (Code, Heap, Stack) as described above.
  - **Fine-Grained:** Large number of smaller segments (used in early OSs). This requires a **Segment Table** stored in memory.

5. Fragmentation

Segmentation introduces specific memory management challenges.

External Fragmentation

Because segments vary in size, physical memory becomes filled with small "holes" of free space between allocated segments.

- **Compaction:** Moving segments to compact memory is possible but **expensive**.
- **Allocation Algorithms:** To mitigate this, OSs use algorithms such as:
  - Best Fit
  - Worst Fit
  - First Fit
  - Buddy Algorithm (used in xv6).

Internal Fragmentation

This occurs within the allocator itself. If an allocator hands out chunks of memory bigger than requested, the unused space inside that chunk is considered **Internal Fragmentation**.

Paging: Introduction

1. The Concept

Paging divides memory into fixed-sized chunks.

- **Virtual Memory:** Divided into **Pages**.
- **Physical Memory:** Divided into **Frames**.

Advantages:

- Most flexible approach.
- **No External Fragmentation** (because all chunks are the same size).

2. Address Translation Mechanism

The Virtual Address (VA) is split into two parts:

1. **VPN:** Virtual Page Number.
2. **Offset:** The offset within the page.

Example Calculation

Assume a system where the first 2 bits are VPN and the last 4 bits are the offset:

- **Page Size:** $2^{\text{offset bits}} = 2^4 = 16$ bytes.
- **Number of Pages:** $2^{\text{VPN bits}} = 2^2 = 4$ pages.
- **Total Address Space:** 4 pages $\times$ 16 bytes/page = 64 bytes.

The Translation Logic

The OS uses a **Page Table** to map `VPN` $\rightarrow$ `PFN` (Physical Frame Number). The Physical Address (PA) is calculated using bitwise operations:

1. **Extract VPN:** Mask and shift the VA.
2. **Lookup PFN:** `PFN = PageTable[VPN]`
3. **Formulate PA:** Shift the PFN back and combine with the offset.

$$PA = (\text{PFN} \ll \text{num\_bits\_offset}) \mid \text{offset}$$

3. The Page Table

The Page Table is a per-process data structure recorded by the OS to store where each virtual page is physically located.

- **Location:** The page table itself lives in **Kernel Space** (Main Memory).
- **Structure:** A simple **Linear Page Table** is an array indexed by the VPN.

Page Table Entry (PTE)

A PTE is usually 4 bytes (32 bits) in size. It contains the PFN and several control bits:

Bit/Flag	Description
Valid Bit	Indicates if the translation is valid. Unused space in the address space is marked invalid.
Present Bit	Indicates if the page is in physical memory (1) or swapped out to disk (0).
Dirty Bit	Indicates if the page has been modified since it was loaded into memory.
Accessed Bit	Used by the OS to track usage (e.g., for circular page replacement algorithms).
Protection Bits	Stores permissions (Read/Write/Execute).

4. Context Switching

With paging, what must be saved/restored during a process context switch?

- A **pointer** to the page table (Page Table Base Register).
- The **size** of the page table.

5. Issues (Cons)

1. **Performance (Too Slow):** Address translation requires an extra memory lookup (to read the page table) before accessing the actual data.
2. **Memory Overhead (Too Much Memory):** Page tables can become very large, consuming significant physical memory.
3. **Internal Fragmentation:** While paging solves external fragmentation, it can suffer from internal fragmentation (wasted space *inside* a page if the process doesn't use the full page size).

Advanced Paging & Demand Paging

1. The Problem with Linear Page Tables

We cannot simply increase the page size to reduce page table size, as this causes **internal fragmentation**. To solve the memory overhead of large linear page tables, we look at alternative structures.

2. Approach 1: Hybrid Approach (Paging + Segmentation)

This approach combines paging with segmentation.

- **Mechanism:** Instead of one large page table, we have three separate page tables: one for **Code**, one for **Heap**, and one for **Stack**.
- **Hardware:** The Base/Bounds register pair for each segment now indicates the **start and end of each page table** (rather than the data segment itself).
- **Benefit:** This reduces unused space (invalid entries) in the page table, as we only allocate page table space for valid segments.

Address Translation:

The Virtual Address (VA) is split as follows:

- **Top 2 bits:** Segment ID (identifies Code, Heap, or Stack).
- **Next 18 bits:** VPN (Virtual Page Number).
- **Last bits:** Offset.

The Issue: While page sizes are fixed, the *Page Tables* themselves can now be of arbitrary sizes. This re-introduces the problem of **External Fragmentation** when allocating memory for the page tables.

3. Approach 2: Multi-Level Page Tables (MLPT)

This approach converts a linear page table into a **tree** structure.

Structure

1. **Page Directory (PD):** The root of the tree. It contains Page Directory Entries (PDE).
2. **Page Directory Entry (PDE):**
   - Contains a **Valid Bit** and a **PFN** (Physical Frame Number).
   - **Logic:** If a PDE is valid, it means at least one page in that specific page table chunk is valid. If the PDE is invalid, the entire corresponding page of the page table is invalid/unallocated.

Benefit: It allocates space in proportion to the addresses actually used, drastically saving memory for sparse address spaces.

Address Translation Example

Assuming a 14-bit VPN:

1. **First 4 bits:** Page Directory Index (to locate the PDE).
2. **Next 4 bits:** Page Table Index (to locate the PTE).
3. **Last 6 bits:** Offset.

   Note: We can go deeper, for example, a three-level page table (PD0 bits $\rightarrow$ PD1 bits $\rightarrow$ PT bits $\rightarrow$ Offset).

4. Demand Paging

To support address spaces larger than physical memory, we use **Demand Paging**.

- **Swap Space:** Disk space used to move pages back and forth between RAM and disk.
- **Requirement:** The OS must remember the disk address of every page.

Mechanisms

- **Page Fault:** The act of accessing a page that is not currently in physical memory.
- **Page-Fault Handler:** The OS code that runs upon a page fault.
- **Present Bit:** A bit in the PTE that indicates if the page is in physical memory (1) or on disk (0).

Operations:

- **Swap In:** Moving data from Disk $\rightarrow$ Memory.
- **Swap Out:** Moving data from Memory $\rightarrow$ Disk.

5. Page Replacement Policy (Swap Daemon)

Most operating systems try to keep a small amount of memory free using a background thread (often called the swap daemon).

Watermarks:

The OS uses a **High Watermark (HW)** and **Low Watermark (LW)** to decide when to evict pages.

1. **Trigger:** When the OS notices that `Available Pages < LW`, the background thread runs.

2. **Action:** The thread evicts pages (frees memory) until `Available Pages > HW`.

Translation Lookaside Buffer (TLB)

1. Introduction

To speed up address translation, we use the **Translation Lookaside Buffer (TLB)**.

- **Location:** It is part of the **Memory Management Unit (MMU)**.
- **Function:** It is a hardware cache that stores **VPN** $\rightarrow$ **PFN** translations.

2. Basic Operation

Upon each memory reference, the hardware first checks the TLB:

1. **TLB Hit:** The TLB holds the PFN for the queried VPN. The translation is immediate.
2. **TLB Miss:** The TLB does *not* hold the PFN for the queried VPN.
    - **Action:** The system must query the page table, update the TLB with the new `(VPN, PFN)` pair, and retry the instruction.

3. Handling TLB Misses

Who handles the miss?

- Once a miss occurs, the hardware raises an **exception**.
- Control is passed to the **Kernel Mode** trap handler.
- **OS Action:** The OS queries the page table, updates the TLB, and executes a **Return-from-Trap**.

Distinction: Return-from-Trap

There is a key difference in where the execution resumes compared to a standard system call:

- **System Call:** Returns to the **next** instruction after the trap calling instruction.
- **TLB Miss:** Returns to the **same** instruction that caused the trap. This allows the hardware to **retry** the instruction (which should now result in a TLB Hit).

4. Performance & Locality

The performance of the TLB is measured by the Hit Rate:

$$\text{TLB Hit Rate} = \frac{\text{Num Hits}}{\text{Num Accesses}} \times 100\%$$

Locality:

- **Spatial Locality:** TLB improves performance because accessing one address often means accessing nearby addresses (which are on the same page).
    - *Page Size Effect:* Larger page sizes result in fewer TLB misses (as one entry covers more memory).

- **Temporal Locality:** If a program is re-run after its first execution, the TLB hit rate can approach **100%** because the translations are already cached.

5. Context Switching

When switching between processes, the TLB entries for the old process are no longer valid.

- **Approach 1 (Flush):** Flush (empty) the TLB on every context switch. This ensures correctness but hurts performance (cold cache).
- **Approach 2 (ASID):** Provide an **Address Space Identifier (ASID)** or PID to the TLB entries. This allows the TLB to differentiate entries between processes without flushing.

6. Replacement Policy

When the TLB is full, we must evict an entry to add a new one. The standard policy mentioned is **LRU** (Least Recently Used).

Concurrency: Introduction & Threads

1. The Thread Abstraction

Each thread acts as an independent agent running within a program. * **Multiple Execution Points:** A multi-threaded program has more than one point of execution, meaning it has multiple Program Counters (PCs) being fetched and executed simultaneously.

Shared vs. Private Data

Threads share the same **Address Space**, meaning they can access the same data. However, there are key differences in storage:

Resource	Scope	Description
Address Space	**Shared**	All threads in the program share the heap and code.
Registers	**Private**	Each thread has its own set of registers for computation. These are virtualized by saving/restoring them during context switches.
Stack	**Private**	Each thread has its own stack (often called **Thread-Local Storage**).

Context Switching

- **Mechanism:** Switching from Thread 1 to Thread 2 requires a context switch.
- **Data Structure:** The OS uses a **Thread Control Block (TCB)** to manage this state, similar to a PCB for processes.

2. The Problem: Race Conditions

Race Condition: A situation where the results of a program depend on the timing execution of the code.

- **Indeterminacy:** Due to race conditions, we might get **indeterministic** output (varying from run to run) rather than the deterministic output usually expected from computer systems.

Critical Sections

A **Critical Section** is a piece of code that accesses a shared variable (or shared resource) and must **not** be concurrently executed by more than one thread.

- **The Danger:** A race condition arises if multiple threads enter the critical section at roughly the same time and attempt to update the shared data structure, leading to surprising or undesirable outcomes.
- **Example:** Code that updates shared structures like a file allocation bitmap or an inode during a `write()` operation acts as a critical section.

3. The Solution: Atomicity & Synchronization

To solve these issues, we rely on **Atomicity** and **Synchronization Primitives**.

Atomicity

- **Definition:** Atomic means "all or nothing".
- **Hardware Guarantee:** Atomic instructions cannot be interrupted. Either they complete entirely, or they do not execute at all.
- **Transactions:** The grouping of many actions into a single atomic action is called a transaction.

Synchronization Primitives

Using a few hardware instructions, we can build a set of synchronization primitives to ensure correctness.

- **Mutual Exclusion:** Threads should use primitives (locks) so that only a **single thread** ever enters a critical section.
- **Condition Variables:** Useful when signaling must take place between threads (e.g., if one thread is waiting for another to do something before it can continue).

4. Thread Usage Patterns

- `pthread_join()`: A function used to wait for a particular thread to complete.
- **Independent Threads:** It is not always necessary to join threads.
  - *Example:* An indefinite web server where the main program passes requests to thread workers and they run independently.

Pthread API

1. Function Pointers in C

To use pthreads, we often need to understand function pointers.

- **Why `void *`?** We use `void *` in signatures because it allows us to pass and return data of **any type**.

Syntax Examples:

- Function taking `int`, returning `void*`: `void * (*start_routine)(int)`.
- Function taking `void*`, returning `int`: `int (*start_routine)(void *)`.

2. Thread Creation

To create a thread, we use `pthread_create`.

```
int pthread_create(
    pthread_t *thread,                  // pointer to thread identifier variable
    const pthread_attr_t *attr,         // specifies thread attributes (can be NULL)
    void *(*start_routine)(void *),     // func pointer: takes void*, returns void*
    void *arg                           // argument passed to start_routine
);
```

- **Return Value:** Returns `0` for success and non-zero for failure.
- **Arguments:** The `arg` passed must match the type expected by the `start_routine`.

3. Thread Completion

`pthread_join()`

Waits for the specified thread to complete execution.

```
int pthread_join(pthread_t thread, void **retval);
```

- **`retval`:** Used to retrieve the return value from the thread's start routine (which returns `void *`).

Equivalence:

Immediately creating a thread and then joining it is equivalent to a standard **function/procedure call**.

`pthread_detach()`

If you call `pthread_detach(thread)`, you explicitly tell the OS that you will **not** join this thread.

- **Effect:** You cannot call `pthread_join` on that thread anymore.
- **Cleanup:** Detached threads clean up their own resources automatically when they exit.

  Warning (Deadlock): Calling `pthread_join` on **yourself** (the current thread) will cause the program to hang forever (deadlock).

4. Code Example

Here is a safe implementation of passing arguments and returning results:

```c
void *compute_sum(void *arg) {
    int n = *(int *)arg;

    // Allocate memory on heap for the result to persist after thread exits
    int *result = malloc(sizeof(int));
    *result = n * (n + 1) / 2;
    return result;
}

int main() {
    pthread_t tid;
    int n = 10;
    void *res;

    pthread_create(&tid, NULL, compute_sum, &n);
    pthread_join(tid, &res);

    printf("Sum = %d\n", *(int *)res);
    free(res); // Important: free the allocated memory
    return 0;
}
```

5. Memory Safety Guidelines

Passing Arguments (Stack vs. Heap)

If you pass the address of a **local variable** (stack memory) to a thread:

- You must ensure the main thread does not **modify** that variable or **go out of scope** (exit) before the new thread finishes using it.

Returning Values

Never return a pointer to a local variable defined on the thread's stack.

- **Reason:** When the thread exits, its stack memory is destroyed/reclaimed.
- **Result:** The parent will receive a pointer to garbage values or unmapped memory, potentially causing a **Segmentation Fault**.

6. Synchronization: Condition Variables

Condition variables are used when one thread is waiting for another to do something before it continues.

Locking Rules

1. **Waiting (`cond_wait`):** Requires the **lock** (mutex) and the **condition variable**.

- *Internally:* `cond_wait` releases the lock (so other threads can run) and puts the calling thread to sleep.
- *Upon Return:* When returning from the wait, the thread automatically **re-acquires** the lock.

2. **Signaling:** We should hold the lock while signaling to prevent race conditions.

Spurious Wakeups

`cond_wait` should always be run inside a **`while` loop**, not an `if` statement.

- **Reason:** To handle **spurious wakeups** (the thread might wake up even if the condition hasn't been met).

Anti-Pattern: Spin Waiting

An alternative to condition variables is spinning:

```
while (init == 0); // In thread A
init = 1;          // In thread B
```

- **Verdict:** This is **not recommended**. It wastes CPU cycles (performs poorly) and is error-prone.

Locks & Synchronization

1. The Lock Abstraction

A lock is simply a variable that exists in one of two states:

1. **Available** (unlocked, free).
2. **Acquired** (locked, held).

Constraint: Exactly one thread can hold a lock at a given time. This property allows locks to provide **Mutual Exclusion** between threads.

Granularity

- **Fine-grained:** Locks protect a small number of instructions (increases concurrency).
- **Coarse-grained:** Locks protect large segments of code.

2. Goals of a Lock

Any lock implementation should satisfy three goals:

1. **Mutual Exclusion:** Ensure only one thread enters the critical section.
2. **Fairness:** When the lock becomes free, threads waiting for it should have a fair chance to acquire it. No thread should starve.
3. **Performance:** Minimize the overhead introduced by using locks (especially regarding how they perform on multiple CPUs).

---

3. Method I: Controlling Interrupts

The earliest strategy (for single-processor systems) was to disable interrupts.

Mechanism:

- `lock()` → Disable interrupts.
- `unlock()` → Enable interrupts.

Analysis:

- **Benefit:** Simplicity.
- **Pitfalls:**
  1. **Trust:** Requires the thread to perform a privileged instruction. A greedy program could acquire the lock and never release it (OS never regains control).
  2. **Multiprocessor Failure:** This does **not** work on multiple processors. Disabling interrupts only affects one CPU; threads on other CPUs can still enter the critical section.
  3. **Inefficiency:** Interrupt masking is slow on modern CPUs.

---

4. Method II: Software-Only (Peterson's Algorithm)

A 2-thread algorithm that works without special hardware instructions.

The Code

```
int flag[2]; // flag[i]=1 means thread i wants the lock
int turn;    // whose turn is it?

void lock() {
    flag[self] = 1;      // I want the lock
    turn = 1 - self;     // I give priority to the other thread

    // Spin wait while:
    // 1. The other thread wants the lock
    // 2. AND it is the other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self));
}

void unlock() {
    flag[self] = 0; // I no longer want the lock
}
```

How it works

1. **Intent:** Thread i sets `flag[i] = 1` to indicate it wants to enter.
2. **Priority:** Thread i sets `turn = 1 - i` to yield priority to the other thread.
3. **Spin:** It waits only if the other thread wants in **AND** it is the other thread's turn.

Note (Refer to Book): For extending this logic to N threads, look up the **Filter Algorithm** and **Bakery Algorithm**.

---

5. Method III: Hardware Primitives (Spin Locks)

Modern systems use atomic hardware instructions to build locks.

A. Test-and-Set (Atomic Exchange)

This instruction updates a value and returns the *old* value atomically.

```
int test_and_set(int *ptr, int new) {
    int old = *ptr;
    *ptr = new;
    return old;
}

void lock(lockt *mutex) {
    // Spin while the old value was 1 (meaning it was already locked)
    while (test_and_set(&mutex->flag, 1) == 1);
}
```

B. Compare-and-Swap (Compare-and-Exchange)

This instruction checks if the current value equals `old`; if so, it updates it to `new`. It is generally more powerful than Test-and-Set.

```
int comp_and_swap(int *ptr, int old, int new) {
    int cur = *ptr;
    if (cur == old) {
        *ptr = new;
    }
    return cur;
}

void lock(lockt *mutex) {
    while (comp_and_swap(&mutex->flag, 0, 1) == 1);
}
```

Note (Refer to Book): Read about **Load-Linked / Store-Conditional** and **Fetch-And-Add** (which provides better fairness).

Analysis of Spin Locks

- **Correctness:** They provide mutual exclusion.
- **Fairness:** They are **not fair**. A thread may spin forever (starvation).
- **Performance:**

- **Single CPU:** Poor performance. If a thread holding the lock is preempted, the scheduler might run $N-1$ other threads that just spin and waste time slices.
- **Multiple CPUs:** Works well. Spinning to wait for a lock held on another processor is often effective (doesn't waste many cycles if the critical section is short).

---

6. Method IV: Sleeping (Queue-Based Locks)

To solve the problem of wasting CPU cycles while spinning, we use queues to put waiting threads to sleep.

Mechanisms

- `yield()`: A thread voluntarily gives up the CPU. Better than spinning, but still inefficient with many threads (context switch overhead) and allows starvation.
- `park()`: Puts the calling thread to sleep.
- `unpark(threadID)`: Wakes a specific thread.

The Solaris-Style Lock (Flag + Guard + Queue)

```
typedef struct __lock_t {
    int flag;      // The actual lock
    int guard;     // Protects the lock structure (spinlock)
    queue_t *q;    // Queue of waiting threads
} lock_t;
```

The Algorithm:

1. **Acquire Guard:** Use `test-and-set` on `guard` to safely manipulate the struct.
2. **Check Lock (`flag`):**
   - If `flag == 0` (free): Set `flag = 1`, set `guard = 0` (release guard), and enter critical section.
   - If `flag == 1` (busy):
     1. Insert `getthread_id()` into queue `q`.
     2. Call `setpark()` (prepare to sleep).
     3. Release `guard = 0`.
     4. `park()` (sleep).

The Handover (Unlock):

1. Acquire `guard`.
2. If the queue is empty, release `flag` (`flag = 0`).
3. If the queue is **not** empty:
   - `unpark()` the front thread.
   - **Note:** Do *not* set `flag = 0`. The waking thread assumes it holds the lock immediately upon returning from `park()`.

4. Release `guard`.

Condition Variables

1. The Concept

A **Condition Variable (CV)** is an explicit queue that threads can put themselves on when some condition is not as desired (waiting on the condition).

- **Goal:** To put a thread to sleep (waiting for another thread to complete/signal) and wake it up when the state changes, avoiding the CPU waste of busy-waiting (spinning).

The API

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

How `wait` works:

1. It assumes the mutex `m` is **locked** when called.
2. It internally **releases** the lock and puts the calling thread to sleep.
3. When the thread wakes up (after a signal), it **re-acquires** the lock before returning to the caller.

2. Usage Guidelines

State Variables

A CV is stateless; it must be used in conjunction with a **state variable** (e.g., an integer `done` or a buffer counter) to track the actual condition.

Locking Rule

It is recommended to **hold the lock** while signaling.

- While there are edge cases where it is okay not to, holding the lock prevents race conditions where a thread checks the state and tries to sleep just as another thread signals.

Mesa Semantics (The `while` Loop Rule)

When using condition variables, **always use `while` loops**, not `if` statements.

```
while (done == 0)
    pthread_cond_wait(&c, &m);
```

Reasons:

1. **Signaling is a hint:** The signal wakes the thread, but there is no guarantee that the state is still desired when the thread actually runs.
2. **Spurious Wakeups:** Threads might wake up without a signal due to OS implementation details.

   Note (Refer to Book): Look up **Hoare Semantics** vs. **Mesa Semantics** to understand the theoretical differences in how locks are transferred upon signaling.

3. The Producer-Consumer Problem (Bounded Buffer)

Setup:

- **Producers:** Place items in a buffer. Cannot produce if the buffer is full.
- **Consumers:** Take items out of the buffer. Cannot consume if the buffer is empty.
- **Real-world Example:** Unix pipes use a bounded buffer.

The Single CV Problem

Using a single condition variable is problematic with multiple producers/consumers.

- *Scenario:* A consumer (`C1`) sleeps. A producer (`P`) fills a slot and signals, but accidentally wakes another consumer (`C2`) instead of `C1`. `C2` sees the buffer is empty (because `P` filled it but maybe `C1` ate it? Or `C2` runs and finds nothing).
- *Result:* Everyone might end up sleeping.

The Solution: Two CVs

Use two separate condition variables:

1. `empty`: To signal/wait that the buffer is empty (used by producers).
2. `full`: To signal/wait that the buffer is full (used by consumers).

4. Covering Conditions (Broadcast)

Sometimes, we do not know *which* thread to wake up.

Example: Memory Allocation (`malloc` & `free`)

- Threads `A` (needs 100 bytes) and `B` (needs 50 bytes) are waiting.
- Thread `C` calls `free(60)`.
- If `C` signals and wakes `A`: `A` checks, sees $60 < 100$, and goes back to sleep. `B` remains asleep despite 60 bytes being enough for it.

Solution: Use **Broadcast** (`pthread_cond_broadcast`).

- This wakes **all** sleeping threads. They all check their conditions; the one that can proceed does, and the others go back to sleep.
- *Trade-off:* Performance cost (thundering herd problem).

5. Barriers

A barrier ensures that N threads reach a common point of execution before any of them proceed.

Implementation

```
typedef struct {
    int num;      // Counter for threads arrived
    int N;        // Total threads required
    mutex m;
    cond c;
} barrier_t;
```

```c
void barrier_check(barrier_t *b) {
    lock(&b->m);
    b->num++;

    if (b->num < b->N) {
        // Not everyone is here yet, so wait
        cond_wait(&b->c, &b->m); // Spurious wakeups handled by re-check logic usually
    } else {
        // Everyone arrived! Reset and wake everyone
        b->num = 0;
        broadcast(&b->c);
    }
    unlock(&b->m);
}
```

Note (Refer to Book): Generalized **Semaphores** can also be used as a lock or a condition variable. Check the book for the specific implementation details.

Semaphores

1. Definition and Initialization

A **Semaphore** is an object with an integer value that can be manipulated using two specific routines: `sem_wait()` and `sem_post()`.

Initialization:

We must initialize the semaphore with a positive integer value before using it.

```c
#include <semaphore.h>
sem_t s;

// args: pointer to sem, pshared (0=threads, 1=processes), initial value
sem_init(&s, 0, 1);
```

- **Initial Value:** Here, initialized to 1.
- **Sharing:** The second argument `0` indicates that the semaphore is shared among threads in the current process. To synchronize across different **processes**, change this value to 1.

2. The Operations

These operations must be performed **atomically**.

`sem_wait()` (semdown / P)

```c
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one;
    wait if value of semaphore s is negative;
}
```

`sem_post()` (semup / V)

```
int sem_post(sem_t *s) {
    increment the value of semaphore s by one;
    if there are one or more threads waiting, wake one;
}
```

> **Note:** Internally, if the value of the semaphore is negative, the absolute value typically represents the number of waiting threads. However, the user cannot see this internal value directly.

3. Usage Patterns

A. Binary Semaphores (Locks)

A semaphore initialized with the value **1** acts as a lock (Mutex).

- `sem_wait()` ≈ `lock()`
- `sem_post()` ≈ `unlock()`

B. Ordering (Condition Variables)

Semaphores can be used to order events (e.g., making a parent wait for a child).

- **Initial Value:** Must be **0**.

```
sem_t s;

void child() {
    // Do work
    sem_post(&s); // Signal parent
}

void parent() {
    sem_init(&s, 0, 0); // Init to 0
    printf("waiting for child");
    child_thread_create();
    sem_wait(&s); // Wait for child
}
```

Why this works (Two Paths):

1. **Parent runs first:** Calls `sem_wait`, decrements to -1, and goes to sleep. Child runs later, calls `sem_post`, increments to 0, and wakes parent.
2. **Child runs first:** Calls `sem_post`, increments to 1. Parent runs later, calls `sem_wait`, decrements to 0, and proceeds immediately without waiting.

4. The Producer-Consumer Problem

(Also known as the Bounded Buffer Problem).

Setup:

- `empty`: Semaphore initialized to `buffer_size` (tracks empty slots).

- `full`: Semaphore initialized to `0` (tracks filled slots).

The Role of Mutex

In a multi-producer/multi-consumer scenario, semaphores alone are insufficient because race conditions can occur when inserting/removing from the buffer index. We need a **Mutex** to protect the buffer modification.

The Deadlock Issue

If we implement the lock naively, we get a deadlock:

```
// BAD CODE (Deadlock Risk)
lock(&mutex);       // Acquire Lock
sem_wait(&empty);   // Wait for empty slot
    put(i);
sem_post(&full);
unlock(&mutex);
```

Scenario: If the buffer is full, the producer acquires the lock and *then* waits on `empty`. The consumer cannot run (to make space) because the producer holds the lock. **Deadlock.**

The Solution: Scope of Lock

Move the `sem_wait` **outside** the mutex lock.

1. Wait for a free slot (`sem_wait`).
2. **Then** acquire the lock to manipulate the buffer.
3. Release the lock.
4. Signal (`sem_post`).

5. Reader-Writer Locks

Useful for concurrent lists where we want many concurrent reads but exclusive writes.

The Logic:

1. **First Reader:** Acquires the write lock (blocks any writers).
2. **Middle Readers:** Just read (since the write lock is already held by the group).
3. **Last Reader:** Releases the write lock (allows writers to proceed).

   Note (Refer to Book): See **Figure 31.9** for the implementation details. Note that this approach can lead to writer starvation.

6. The Dining Philosophers Problem

Setup: 5 philosophers, 5 forks. A philosopher needs **two** forks (left and right) to eat.

Naive Solution:

```
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}
```

Issue: Deadlock. If all 5 philosophers grab their left fork simultaneously, they all wait forever for their right fork.

Solution (Breaking the Cycle): Change the order for the last philosopher (Philosopher 4).

- Philosophers 0-3: Grab **Left**, then **Right**.
- Philosopher 4: Grab **Right**, then **Left**.

Note (Refer to Book): Also look up the **Cigarette Smoker's Problem** and the **Sleeping Barber Problem**.

7. Implementation: Zemaphores

Building a Semaphore using Locks and Condition Variables.

```
typedef struct {
    int val;
    lock_t lck;
    cond_t cond;
} sem_t;

void sem_init(sem_t *s, int v) {
    s->val = v;
    lock_init(&s->lck);
    cond_init(&s->cond);
}

void sem_post(sem_t *s) {
    lock(&s->lck);
    s->val++;
    cond_signal(&s->cond); // Wake up a sleeper
    unlock(&s->lck);
}

void sem_wait(sem_t *s) {
    lock(&s->lck);
    while (s->val <= 0) {
        cond_wait(&s->cond, &s->lck); // Sleep if value is <= 0
    }
    s->val--;
    unlock(&s->lck);
}
```

Concurrency Bugs

1. Non-Deadlock Bugs

Most concurrency bugs are not deadlocks. They typically fall into two categories:

A. Atomicity-Violation Bugs

- **Definition:** Code that is intended to be atomic but is not enforced as atomic during execution.
- **The Fix:** Use **Locks** around the shared variable accesses to enforce atomicity.

B. Order-Violation Bugs

- **Definition:** One thread assumes that a variable created or initialized in another thread is already present/active.
  - *Formal:* Thread A expects to run *before* Thread B, but this order is not enforced.
- **The Fix:** Use **Condition Variables** (or semaphores) to force the correct execution order.

2. Deadlocks

Reasons for Deadlock:

1. **Circular Dependencies:** A cycle in the resource allocation graph.
2. **Encapsulation:** With extensive encapsulation (e.g., in Java libraries), we may not know which locks are held by a function (like `A.add(B)`). If another thread calls `A.add(C)` in a different order, a deadlock may occur.

The Four Necessary Conditions

For a deadlock to occur, **all four** of these conditions must hold:

1. **Mutual Exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
2. **Hold and Wait:** Threads hold resources allocated to them (e.g., Lock L1) while waiting for additional resources (e.g., Lock L2).
3. **No Preemption:** Resources (locks) cannot be forcibly removed from threads that are holding them.
4. **Circular Wait:** There exists a circular chain of threads such that each thread holds a resource that is being requested by the next thread in the chain.

Takeaway: If any *one* of these four conditions is not met, a deadlock cannot occur.

3. Deadlock Prevention

We can prevent deadlocks by writing code that breaks one of the four conditions.

Strategy 1: Break "Circular Wait"

- **Solution:** Provide a **Total Ordering** on lock acquisition.
- **Mechanism:** If the system has locks L1 and L2, strictly enforce that L1 must always be acquired before L2. This prevents cyclic waits.

Strategy 2: Break "Hold and Wait"

- **Solution:** Acquire **all** locks at once, atomically.

- **Mechanism:** Use a global "prevention" lock (meta-lock) to ensure atomicity when grabbing resources.

```
lock(prevention); // Global lock
lock(L1);
lock(L2);
...
unlock(prevention);
```

- **Pros:** Guarantees no context switch occurs while acquiring the batch of locks.

Strategy 3: Break "No Preemption"

- **Solution:** Use `pthread_mutex_trylock()` instead of blocking locks.
- **Mechanism:** If a thread holds L1 and tries to get L2 but fails, it should **release** L1 and try again later.

```
top:
lock(L1);
if (trylock(L2) == -1) {
    unlock(L1); // Preempt yourself
    goto top;
}
```

New Problem: Livelock

- In the scenario above, two threads might repeatedly lock L1, fail to get L2, unlock L1, and repeat this cycle indefinitely. No progress is made (similar to thrashing in paging).
- **Fix for Livelock:** Add a **random delay** before retrying.

Strategy 4: Break "Mutual Exclusion"

- **Solution:** Avoid the need for locks entirely by using **Wait-Free Data Structures**.
- **Mechanism:** Use powerful atomic hardware instructions like **Compare-And-Swap (CAS)**.

Compare-And-Swap Logic:

```
int CompareAndSwap(int *addr, int expected, int new) {
    if (*addr == expected) {
        *addr = new;
        return 1; // Success
    }
    return 0; // Failure
}
```

Example: Lock-Free Atomic Increment

```
void incrementbyx(int *addr, int x) {
    do {
        int current = *addr; // Snapshot the value
    } while (CompareAndSwap(addr, current, current + x) == 0);
}
```

- **Why the loop?** If `*addr` changes between the snapshot and the CAS, the CAS fails (returns 0). The loop forces a retry with the new value.
- **Why not `CAS(addr, *addr, *addr + x)`?** This is flawed because the arguments are evaluated before the function call. By the time CAS runs, `*addr` might have changed, but the function would be using the stale value passed in the arguments.

Example: Lock-Free List Insert

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head; // Point to current head
    } while (CompareAndSwap(&head, n->next, n) == 0); // Try to swap head to n
}
```

> **Note (Refer to Book/Slides):** Review the specific algorithms and graphs for deadlock detection in the slides.

I/O Devices

1. System Architecture

A program without any input produces the same result every time (determinism). To interact with the world, we need Input/Output.

Bus Hierarchy

The speed of a bus is generally **inversely proportional** to its length/size and **directly proportional** to its cost.

- **Memory Bus:** The fastest bus, connecting the CPU to Main Memory.
- **General I/O Bus:** Connects high-speed I/O devices (e.g., PCI = Peripheral Component Interconnect).
- **Peripheral I/O Bus:** The slowest bus, meant to support a large number of devices (e.g., SATA, SCSI, USB).

2. Canonical Device Structure

A device generally consists of two parts:

1. **Hardware Interface:** The interface presented to the rest of the system, typically consisting of three registers:
    - **Status:** Shows the current state of the device.
    - **Command:** Tells the device to perform specific tasks.
    - **Data:** Used to pass data to/from the device.
2. **Internal Structure:** Implementation-specific internals. Complex devices may have their own microcontroller and **Firmware** (software written within a hardware device).

3. Protocols: CPU-Device Communication

Approach 1: Polling (Programmed I/O - PIO)

The protocol follows these steps:

1. **Wait (Poll):** The OS reads the status register repeatedly until the device is not busy.
2. **Write Data:** The OS writes data to the data register. (If the main CPU moves the data, it is called **Programmed I/O**).
3. **Write Command:** The OS writes to the command register. This implicitly signals the device to execute.
4. **Wait (Poll):** The OS polls the device again until it receives a success/failure code.

Drawback: Polling is inefficient. It wastes CPU cycles that could be used for other processes.

Approach 2: Interrupts

To avoid wasting cycles:

1. The OS issues the request, puts the calling process to **sleep**, and switches to another task.
2. When the device finishes, it raises a hardware **Interrupt Request (IRQ)**.
3. The CPU pauses execution, saves the state (e.g., PC), and jumps to a pre-determined **ISR (Interrupt Service Routine)** or interrupt handler.
4. The ISR executes and wakes up the sleeping process.

- **Benefit:** Allows the overlap of computation and I/O.

Issues with Interrupts:

- **Performance:** If a device is extremely fast, the overhead of context switching for interrupts slows down the system. In these cases, polling is better.
- **Livelock:** If too many interrupts occur (e.g., a flood of network packets), the OS might spend 100% of its time processing interrupts and never make progress on the actual tasks.

Optimizations:

- **Hybrid Approach:** Poll for a short while; if not done, switch to interrupts.
- **Coalescing:** The device waits for multiple requests to complete (or a timer to expire) before raising a single interrupt. This improves efficiency but increases **latency**.

4. Data Movement: DMA

With Programmed I/O (PIO), the CPU spends too much time copying data word-by-word from memory to the device.

Solution: Direct Memory Access (DMA)

DMA is a specific device within the system that handles data transfers.

1. **Setup:** The OS tells the DMA engine where the data is (source), how much to copy, and where to send it (destination).
2. **Execution:** The OS continues with other work while the DMA engine copies the data directly from memory to the device.
3. **Completion:** When done, the DMA raises an interrupt.

5. Addressing Devices

How does the OS communicate with specific device registers?

Method 1: Explicit I/O Instructions

- The OS uses special hardware instructions to send data to specific device ports.
- **Protection:** These are **privileged** instructions.
  - Example: Linux uses protection rings (Ring 0 for Kernel/Privileged, Ring 3 for User). If a Ring 3 program tries to execute I/O instructions, the CPU raises a fault, allowing the OS to kill the process.

Method 2: Memory Mapped I/O (MMIO)

- The hardware makes device registers available as if they were memory locations.
- To access a register, the OS simply issues a standard **Load** (read) or **Store** (write) to that address.
- **Benefit:** No new instructions are needed in the Instruction Set Architecture (ISA).

6. The Software Stack: Device Drivers

To support the varying interfaces of diverse devices, we use **Device Drivers**.

- **Definition:** Code that abstracts the specific details of a device, allowing the OS to interact with it via a standard interface.
- **Prevalence:** About 70% of the Linux kernel code consists of device drivers.

The Abstraction Flow:

1. File System issues a generic request (e.g., Block Read).
2. **Generic Block Layer** routes the request.
3. **Device Driver** handles the specific communication with the device hardware.

Downside: Rich device functionalities can go underutilized. For example, a "generic error" might be returned to the application because the generic block layer doesn't understand the specialized error code returned by the device.

Hard Disk Drives (HDD)

1. The Interface

From the perspective of the OS, the drive presents a simple interface:

- **Sectors:** The drive consists of sectors, typically **512 bytes** each.
- **Address Space:** Sectors are numbered from 0 to $n - 1$. This is the address space of the drive.
- **Performance:**
  - Accessing blocks in a **contiguous** chunk is faster than random access.
  - Accessing blocks near each other is faster than accessing blocks far apart.

Atomicity & Torn Writes

- **The Guarantee:** A 512-byte write is atomic (it either completes or doesn't happen).
- **The Risk:** Multi-sector writes are possible, but if a power loss occurs during a larger write (e.g., 4KB), it results in a **Torn Write** (an incomplete write where only some sectors were updated).

2. Disk Geometry

1. **Platter:** A circular hard surface with a magnetic coating to store data. Each platter has 2 sides (surfaces).
2. **Spindle:** Connected to a motor that spins the platters at a constant **RPM** (Rotations Per Minute).
3. **Tracks:** Data is encoded in concentric circles called tracks.
4. **Disk Head & Arm:** There is one disk head per surface to read/write. All heads are attached to a single **disk arm** which moves across the surface to position the head over the desired track.
5. **Operation:**
    - **Read:** Sense a magnetic pattern.
    - **Write:** Induce a change in the magnetic pattern.

Advanced Features

- **Track Skew:** Blocks on subsequent tracks are offset (skewed). This ensures that when the head switches tracks (seek), the next logical block hasn't rotated past the head during the repositioning time.
    - **Multi-Zoned Disk Drives:** Outer tracks are physically longer than inner tracks. To maximize capacity, outer tracks have **more sectors** per track than inner tracks (organized into zones).
- **Cache (Track Buffer):** Small memory (8-16 MB) on the drive. It holds data for write buffering or read-ahead (caching contiguous sectors during a read).

3. I/O Performance: The Math

The total time for an I/O request (T_{IO}) consists of three components:

$$T_{IO} = T_{seek} + T_{rotation} + T_{transfer}$$

1. Rotational Delay

The time waiting for the desired sector to rotate under the disk head.

- **Max:** R (one full rotation).
- **Average:** $R/2$.

2. Seek Time

The time to move the disk arm to the correct track. It involves four phases:

1. **Acceleration:** Arm starts moving.
2. **Coasting:** Arm moves at full speed.
3. **Deceleration:** Arm slows down.
4. **Settling:** Arm positions itself precisely on the track (significant time goes here).

   Rule of Thumb: Average seek time is roughly **1/3** of full seek time (end-to-end).

3. Transfer Time

The time to actually read/write the data.

I/O Rate (R_{IO})

To compare disks or workloads:

$$R_{IO} = \frac{Size_{transfer}}{T_{IO}}$$

- **Sequential vs. Random:** Disks give much higher R_{IO} for sequential reads because there is only one seek and one rotational delay at the start, amortized over many sectors.

4. Write Acknowledgement Policies

When does the disk tell the OS the write is done?

1. **Write Back:** Acknowledge when data is put in the **disk cache** (memory).
   - *Pros:* Makes disk appear faster.
   - *Cons:* Dangerous; data can be lost on power failure before hitting the platter.
2. **Write Through:** Acknowledge when data is actually written to the **disk surface**.
   - *Pros:* Safe.
   - *Cons:* Slower.

5. Disk Scheduling

Given a set of I/O requests, the scheduler decides the order of execution.

Basic Algorithms

1. **SSTF (Shortest Seek Time First):**
   - Pick the request on the track closest to the current head position.
   - *Issue:* The OS doesn't perfectly know the disk geometry (tracks/cylinders).
2. **NBF (Nearest Block First):**
   - Schedule based on the nearest logical block address.
   - *Issue:* **Starvation**. A stream of requests for nearby blocks can prevent far-away blocks from ever being serviced.

SCAN Algorithms (The Elevator)

1. **SCAN:** The arm sweeps across the disk (e.g., inner to outer), servicing requests in order.
2. **FSCAN (Freeze SCAN):** To avoid starvation of far-away requests, it freezes the queue during a sweep. New requests arriving during the sweep are placed in a separate queue for the *next* sweep.
3. **C-SCAN (Circular SCAN):** Sweeps in only one direction (e.g., outer to inner), then resets to the start without servicing. This provides more uniform wait times.

   Note: SCAN and SSTF are not "purely" optimal because they ignore rotational delay.

Optimal: SPTF (Shortest Positioning Time First)

Also called **SPTF** or **SATF** (Shortest Access Time First).

- **Logic:** It considers both **Seek Time** and **Rotational Delay**.
- *Example:* If Seek Time > Rotation Time, it picks the item on the closer track. If Rotation > Seek, it might pick a slightly farther track if the sector is rotationally aligned better.
- **Implementation:** Usually performed inside the **disk drive controller** (since the drive knows the exact geometry and head position).

I/O Merging

The scheduler often merges multiple consecutive block requests into a single, larger request before reordering. This reduces overhead.

RAID (Redundant Array of Inexpensive Disks)

1. Introduction & Hardware

RAID uses multiple disks to work together to provide better performance, capacity, and reliability. * **Hardware RAID:** A specialized controller with its own processor and RAM manages the disks. It presents the array to the OS as a single logical disk. * **Benefits:**

```
* **Performance:** Achieved via parallel I/O.
* **Capacity:** Aggregates multiple disks.
* **Reliability:** Redundancy improves data safety (at the cost of net capacity).
```

Fault Model:

- We assume a **Fail-Stop** model. A disk is either working or permanently failed. The RAID controller can easily detect a failure.

2. Evaluation Metrics

We evaluate RAID levels based on:

1. **Capacity:** How much useful space is available?
2. **Reliability:** How many disk failures can it withstand?
3. **Performance:** Throughput and Latency.

Workload Definitions:

- **Sequential:** Accessing large contiguous ranges of data. Transfer Rate = S MB/s.
- **Random:** Small accesses at random locations. Transfer Rate = R MB/s.
- *Note:* $S \gg R$ (Sequential is much faster than Random).

---

3. RAID Level 0: Striping

RAID 0 is not technically a "Redundant" array as it offers no data protection. It splits data blocks across all disks.

Structure:

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11

- **Stripe:** Blocks in the same row form a stripe (e.g., 0, 1, 2, 3).
- **Chunking:** Instead of striping every single block, we can stripe **chunks** of blocks (e.g., blocks 0,1 on Disk 0; 2,3 on Disk 1).
  - *Small Chunk:* High parallelism (files spread across many disks), but positioning time increases (determined by the slowest disk).
  - *Big Chunk:* Reduces positioning time but reduces intra-file parallelism.

Formulas (Standard Striping):

- Disk = Logical Block%Num_Disks
- Offset = Logical Block/Num_Disks

Analysis (N disks, B size per disk):

- **Capacity:** $N \times B$.
- **Failures Allowed:** 0.
- **Latency:** Read/Write = T (single disk time).
- **Throughput:**
  - Sequential: $N \times S$ (Full parallelism).
  - Random: $N \times R$.

---

4. RAID Level 1: Mirroring

For each logical block, RAID keeps 2 copies (on separate disks).

Structure (RAID 10 - Stripe then Mirror):

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5

- **RAID 10:** Mirrors pairs (0-1, 2-3) and stripes data across them.
- **RAID 01:** Stripes data first, then mirrors the huge stripes (less reliable than 10).

Analysis:

- **Capacity:** $(N \times B)/2$.
- **Failures Allowed:** 1 disk guaranteed; up to $N/2$ if lucky (matching pairs don't fail).
- **Latency:**
  - Read: T
  - Write: Slightly $> T$ (must update both copies; wait for slower of the two).
- **Throughput:**
  - Sequential Read/Write: $(N \times S)/2$.
  - Random Write: $(N \times R)/2$.
  - Random Read: $N \times R$ (Can read distinct blocks from the primary and the mirror in parallel).

Crash Consistency

Problem: If power fails after writing to Disk 0 but before Disk 1, the mirror is inconsistent.

Solution: The controller uses a **Write-Ahead Log (WAL)** in non-volatile RAM to record intentions before writing. On reboot, it replays the log to fix inconsistencies.

---

5. RAID Level 4: Parity-Based

Uses a dedicated parity disk to achieve redundancy without the 50% capacity cost of mirroring.

Structure (5 Disks):

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4 (Parity)
0	1	2	3	**P0**
4	5	6	7	**P1**

Parity Logic:

- $P = \text{XOR}(\text{Block}_0, \text{Block}_1, \text{Block}_2, \text{Block}_3)$.
- **Invariant:** The number of 1s in any stripe (including parity) is even.
- **Recovery:** Lost Block = XOR(Remaining Blocks + Parity).

Analysis:

- **Capacity:** $(N-1) \times B$.
- **Failures Allowed:** 1.
- **Throughput:**
  - Sequential Read: $(N-1) \times S$.
  - Sequential Write: $(N-1) \times S$ (Efficient: known as a **Full Stripe Write**).
  - Random Read: $(N-1) \times R$.
  - Random Write: **Poor**. Limited to $R/2$.

The Small Write Problem:

To update a single block randomly, we must update the parity.

- **Subtractive Parity Formula:** $P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$.
- **Cost:** 2 Reads (C_{old}, P_{old}) + 2 Writes $(C_{new}, P_{new}) = 4$ operations.
- **Bottleneck:** The parity disk is accessed for *every* write. It serializes all writes, halving the throughput.

---

6. RAID Level 5: Rotating Parity

Solves the "Small Write Problem" (parity disk bottleneck) by rotating the parity block across all disks.

Structure:

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	**P0**
5	6	7	**P1**	4
10	11	**P2**	8	9
15	**P3**	12	13	14
P4	16	17	18	19

Analysis:

- **Capacity:** $(N-1) \times B$.
- **Failures Allowed:** 1.
- **Throughput:**
  - Sequential Read/Write: $(N-1) \times S$.
  - Random Read: $N \times R$ (All disks utilized).
  - Random Write: $\frac{N \times R}{4}$.
    - Why divide by 4? Each logical write still generates 4 physical I/Os (Read Data, Read Parity, Write Data, Write Parity). However, since parity is distributed, these ops are spread across all disks rather than hitting one bottleneck.

7. Summary

- **RAID 0:** Best for performance, zero reliability.
- **RAID 1:** Best for random I/O performance and reliability, but expensive (50% capacity loss).
- **RAID 5:** Best balance of capacity and reliability. Good for sequential I/O and random reads.

File System Implementation (VSFS)

1. Overall Organization (VSFS Layout)

We introduce a simplified implementation known as **vsfs** (Very Simple File System), which mimics a typical UNIX file system.

The disk partition is divided into blocks (commonly **4 KB**) addressed from 0 to $N-1$. The layout consists of:

1. **Data Region:** The majority of the disk is reserved for user data.
2. **Inode Table:** A reserved region to hold the array of on-disk inodes (metadata).
3. **Bitmaps:** Used to track free/allocated space.
    - **Data Bitmap:** Tracks free data blocks.
    - **Inode Bitmap:** Tracks free inodes.
4. **Superblock:** Located at the very beginning (Block 0).
    - Contains specific file system parameters: total number of inodes/data blocks, start location of the inode table, and a **magic number** to identify the file system type.
    - The OS reads this first when mounting the file system.

2. The Inode (Metadata)

Inode (Index Node) is the structure that holds the metadata for a given file.

- **Content:** File size, permissions, owner, access/modify times, and the location of data blocks.
- **Identification:** Each inode is referred to by a unique **i-number** (low-level name).
- **System Call:** `stat` or `fstat` retrieves this info from the filesystem.

Calculating Inode Location

Given an i-number, the OS can calculate the exact physical address. Since disks are sector-addressable (usually 512 bytes), not byte-addressable, the formula is:

$$\text{Sector} = \frac{(\text{i-number} \times \text{sizeof(inode)}) + \text{inodeStartAddr}}{\text{sectorSize}}$$

3. Indexing Strategies

How does the inode point to the data?

A. Multi-Level Index (Pointer Based)

Used to support both small and large files efficiently.

- **Direct Pointers:** The inode contains a fixed number of direct pointers (e.g., 12) that point directly to user data blocks.
- **Indirect Pointer:** Points to a block that contains *more* pointers to data blocks.
- **Double/Triple Indirect Pointers:** Points to a block of indirect pointers (imbalanced tree structure).

Capacity Calculation:

Assuming 4KB blocks and 4-byte pointers (1024 pointers per block):

$$\text{Max Size} = (12 + 1024 + 1024^2) \times 4\text{KB} \approx 4\text{GB}$$

B. Extents (Extent Based)

- **Definition:** An extent is a disk pointer plus a **length** (in blocks).
- **Pros:** More compact; requires less metadata than pointers.
- **Cons:** Less flexible; requires finding contiguous free space on disk. If the disk is fragmented, a file may require multiple extents.

4. Directory Organization

Directories are treated as a special type of file.

- **Content:** A list of `(entry name, inode number)` pairs.
- **Special Entries:**
  - `.` (dot): Current directory.
  - `..` (dot-dot): Parent directory.
- **Storage:** The directory has an inode (type marked as "directory"), and its data blocks contain the list of entries.
- **Deletion:** When a file is deleted, it may leave a gap. Directories track **Record Length** vs. **String Length** to manage this empty space or reuse it for new entries.

5. Free Space Management

The file system must track free inodes and data blocks.

- **Bitmaps:** vsfs uses one bitmap for inodes and one for data. A bit is `0` if free, `1` if used.
- **Pre-allocation Policy:** To improve performance (and allow extents/contiguous reads), the OS often looks for a sequence of free blocks (e.g., 8 blocks) rather than just one when writing a file.

6. Access Paths (Reading and Writing)

Understanding the flow of I/O operations is critical (The Crux).

A. Reading a File

Task: `open("/foo/bar", O_RDONLY)` and read it.

1. **Traverse Path:** The FS reads the root inode (usually i-number 2).
2. **Lookup:** It reads root data to find `foo`, gets `foo`'s inode, reads `foo`'s data to find `bar`, then gets `bar`'s inode.
3. **Read:** Access the data blocks pointed to by `bar`'s inode.
4. **Update:** Update the **last accessed time** in the inode.

   Note: Reading does *not* access allocation structures (bitmaps).

B. Writing to a File

Task: Create `/foo/bar` and write data. Writing is much more expensive than reading. Each write logically generates roughly **5 I/Os**:

1. **Read** Data Bitmap (find free block).
2. **Write** Data Bitmap (mark allocated).
3. **Read** Inode (to update pointers).
4. **Write** Inode (commit new pointers).
5. **Write** Data Block (actual data).

   Note (Refer to Book/Slides): *Plus additional I/O for directory updates during file creation!*

7. Caching and Buffering

To mitigate the high cost of I/O, the OS uses system memory (DRAM).

- **Read Caching:**
  - Modern systems use a **Unified Page Cache** (integrating virtual memory pages and file system pages).
  - Subsequent reads of the same file (or directory lookups) often hit the cache, requiring no disk I/O.
- **Write Buffering:**
  - Writes are delayed in memory (5–30 seconds).
  - **Benefits:**
    1. **Batching:** Update a bitmap once for multiple writes.
    2. **Scheduling:** Reorder I/Os for disk efficiency.
    3. **Avoidance:** If a file is created and immediately deleted, the write never hits the disk.
  - **Durability Risk:** If the system crashes before the write propagates, data is lost. Applications needing guarantees use `fsync()`.