Introduction to Operating Systems

Core Concepts

- **Operating System (OS)**: Software that acts as an intermediary, coordinating application execution, providing common services, and managing hardware resources.
- An OS is a reactive, "event-driven" system, unlike typical programs that run start-to-finish. It waits for events and responds to them.
- **Kernel**: The central part of the OS that manages all tasks and hardware. A **bare metal OS** is a non-virtualized OS.
- **Events**: Can be **asynchronous** (from external devices like keyboards, networks) or **synchronous** (from program execution like syscalls, page faults).
- **Core OS Services**:
  - **Isolation**: Prevents processes from interfering with each other, often via multiplexing.
  - **Abstraction**: Hides hardware complexity and provides high-level, portable interfaces.
  - **Resource Sharing**: Manages hardware like the CPU and memory among multiple applications.
- **Privilege Levels (CPL)**: The CPU defines **Kernel Mode** ($CPL = 0$) for privileged instructions and **User Mode** ($CPL = 3$) for applications.
- **System Call (Syscall)**: The gateway for user-mode apps to request kernel services. It raises the privilege level to CPL=0 and transfers control to the kernel. On error, they often set the global `errno` variable.
- **POSIX**: A set of standard OS interfaces based on the Unix operating system.

Virtualization

- **Virtualization**: An abstraction that presents each process with virtual versions of resources (CPU, memory). This provides convenience, portability, and protection, but adds overhead.
- **Multiplexing**: Creating the illusion that each process has its own dedicated resources when they are actually shared. The OS decides how to divide them.
- **Hypervisor**: The underlying OS in a virtualized environment that allows running multiple virtual servers on a single physical server.

Basic Hardware

- **Processor/Core**: Executes machine instructions. Multiple cores run in parallel. (Typical access time: ~3ns).
- **Memory (DRAM)**: Fast, volatile storage accessible to all cores. (Typical access time: ~100ns).
- **Disk (Storage)**: Slow, non-volatile (durable) storage. Information must be moved to memory for the CPU to process it. (Typical access time: ~1ms).

Processes & Threads

Processes

- **Process**: An instance of a program in execution.
- **Multiprogramming**: Running multiple processes concurrently on a single core by multiplexing the CPU.
- **Multiprocessing**: Using more than one core to run a single process.
- **Address Space**:
  - **Code (Text)**: Program instructions.
  - **Static Data**: Global/static variables.
  - **Heap**: For dynamic memory allocation (`malloc`).
  - **Stack**: For function calls and local variables.
- **PCB (Process Control Block)**: OS data structure (`task_struct`) to manage a process, pointed to by the `current` pointer. The process list is a cyclic doubly-linked list.
- **PCB Fields**: Contains PID, TGID, state, registers, memory info (`mm_struct`), family pointers (`parent`, `children`), exit code, etc.
- **Process States**:
  - **Running**: Executing on the CPU.
  - **Ready**: Runnable, but waiting for the CPU.
  - **Waiting (Sleeping)**: Waiting for an event (e.g., I/O). States can be `TASK_INTERRUPTIBLE` (can be woken by signals) or `TASK_UNINTERRUPTIBLE` (cannot, e.g., during a critical I/O operation).
  - **Stopped**: Execution suspended (e.g., by a debugger via Ctrl+Z).
  - **Zombie**: Terminated, but its PCB still exists for the parent to collect the exit status via `wait()`.
- **Process Family**: **Idle Process** (PID 0) is the first process, which creates the **init process** (PID 1). **Real Parent** is the original creator. **Parent** is the process currently tracing it. **Orphans** are adopted by the **init** process.
- **Syscalls**: `fork()` (create child), `exec()` (replace program image), `wait()`/`waitpid()` (wait for child), `exit()` (terminate), `kill()` (send signal), `getpid()` (get Thread Group ID), `getppid()` (get parent PID).
- `fork()` returns 0 to the child and the child's PID to the parent.

Threads

- **Threads**: Units of execution within a process. In Linux, they are implemented as processes that share resources.
- **Multithreading**: A process with multiple threads of execution.
  - **Shared**: Memory space (heap, data, code), file descriptors (FDT), working directory, signal handlers.
  - **Unique**: Stack, registers (incl. PC, SP), Thread ID (TID).
- **Thread Group**: A set of threads sharing a process ID (PID), which is the Thread Group ID (TGID). `getpid()` returns TGID, `gettid()` returns the unique TID.
- An `exit()` call by any thread terminates the entire thread group.
- **Pthreads**: POSIX API for threads. `pthread_create()`, `pthread_join()`, `pthread_exit()`.
- **User Level Threads (ULTs)**: Managed by a user-level library, invisible to the kernel. Faster context switching but no true parallelism on multi-core systems.
- **Kernel Threads**: Processes created by the OS to perform kernel tasks. They do not switch the memory space on context switch.

Signals

- **Signal**: An asynchronous notification sent to a process about an event. Handled in user mode when returning from the kernel.
- A signal can wake a waiting process (e.g., from a blocking syscall).
- **Handling**: A process can define a custom handler (`signal()`, `sigaction()`), ignore the signal (`SIG_IGN`), or use the default action (`SIG_DFL`, usually "die" or "ignore").
- **Masking**: A process can block signals using `sigprocmask()` to prevent race conditions. Blocked signals become "pending".
- **Unblockable Signals**: `SIGKILL` (terminate) and `SIGSTOP` (suspend) cannot be caught, blocked, or ignored. `SIGCONT` (resume) is also special.
- **Common Signals**: `SIGSEGV` (segfault), `SIGILL` (illegal instruction), `SIGCHLD` (child status changed), `SIGINT` (Ctrl+C), **Signal 0** (null signal to check if a process exists).

IPC & System Calls

- **clone()**: Linux syscall to create processes/threads. Arguments control resource sharing:
  - `CLONE_VM`: Share memory space.
  - `CLONE_FILES`: Share file descriptor table.
  - `CLONE_SIGHAND`: Share signal handlers.
  - `CLONE_THREAD`: Place in same thread group (implies parent is caller's parent).
- **Copy-on-Write (COW)**: A `fork()` optimization. The parent's memory pages are not physically copied for the child. A page is only duplicated when the parent or child writes to it.
- **Pipe**: Unidirectional kernel memory buffer (`int fd[2]`) for communication between related processes. Read blocks if the pipe is empty; write blocks if it is full.
- **FIFO (Named Pipe)**: A special file in the filesystem (`mkfifo()`) for IPC between unrelated processes. A process opening a FIFO for read-only will block until another opens for write, and vice-versa.

Boot & Modules

- **Booting Sequence**:
  1. **BIOS**: Initializes hardware, finds a bootable device, loads the MBR.
  2. **MBR** (Master Boot Record, 512 bytes): Loads the boot loader.
  3. **Boot Loader (GRUB)**: Loads the kernel (`bzImage`) and **initramfs** (initial RAM filesystem with essential drivers).
  4. **Kernel**: Uses initramfs, mounts the real root filesystem, and runs `/sbin/init` (PID 1).
- **Kernel Modules**: Dynamically loadable code (e.g., drivers) that can be added at runtime without rebooting. Handled by `init_module()` and `cleanup_module()`.
- **Device Files**: Special files in `/dev` representing hardware. Identified by a **major** number (driver) and a **minor** number (device instance).
  - **Character**: Stream-based (e.g., `/dev/tty`).
  - **Block**: Block-based, addressable (e.g., `/dev/sda`).
  - **Pseudo**: Virtual, e.g., `/dev/null`, `/dev/zero`.
- `file_operations` struct: Kernel struct with function pointers (`open`, `read`, etc.) that a driver implements to define its behavior. Registered with `register_chrdev()`.

CPU Scheduling

Concepts

- **Time Sharing**: The OS virtualizes the CPU by dividing its time among multiple processes, creating an illusion of parallelism.
- **Process Types**: **I/O-bound** tasks are latency-sensitive (e.g., interactive shells). **CPU-bound** tasks need raw computation speed.
- **Preemption**: Forcibly stopping a running process to run another. This is managed using a timer interrupt and a time slice called a **Quantum**.
- **Scheduling Metrics**:
  - Wait Time: $T_{wait} = T_{start} - T_{submit}$.
  - Response Time: $T_{resp} = T_{end} - T_{submit}$.
  - Slowdown: $1 + \frac{T_{wait}}{T_{run}}$.

Batch (Non-Preemptive) Schedulers

- **FCFS (First-Come, First-Served)**: Simple and fair, but can suffer from the "convoy effect" where short jobs get stuck behind long ones.
- **SJF (Shortest Job First)**: Optimal for average wait time but requires knowing runtimes in advance and can starve long jobs.
- **EASY**: An FCFS-based scheduler with **backfilling**, which allows short, later-arriving jobs to run early if they won't delay the job at the head of the queue.

Preemptive Schedulers

- **Round Robin (RR)**: Each process gets one quantum in a circular queue. Good for interactivity. With a very large quantum, it behaves like FCFS.
- **SRTF (Shortest Remaining Time First)**: Preemptive version of SJF. Optimal for average response time but still risks starvation.
- **Selfish RR**: A two-level queue system. New processes wait in a FIFO queue while "old" processes are scheduled with RR. Promotes fairness using an "aging" mechanism to eventually move new jobs to the old queue.
- **Gang Scheduling**: An RR variant for parallel jobs, where all threads of a job are scheduled to run concurrently in the same time slot.

Linux Schedulers

- **Linux $\leq$ 2.4 ($O(n)$ scheduler)**:
  - **Policies**: `SCHED_RR`, `SCHED_FIFO` (real-time), `SCHED_OTHER` (default).
  - **Epoch**: A period where every runnable task gets a chance to run. At the end of an epoch, priorities are recalculated for *all* tasks.
  - **Priority**: A combination of fixed **static priority** (user-space `nice` value from -20 to +19) and a **dynamic priority**, or `counter`, which is the remaining time slice.

  - The counter is recalculated for a task at a new epoch: $C_{new} = \lfloor \frac{C_{old}}{2} \rfloor + P_{static}$. This formula favors I/O-bound tasks, as they often don't use their full quantum, so their old counter is not zero when recalculated.
  - **Core Function**: `schedule()` iterates all runnable tasks to find the one with the best `goodness()` value (based on counter and other factors).
- **CFS (Completely Fair Scheduler,** $O(\log N)$**)**:
  - **Goal**: Give each task a fair share of the processor. It aims to keep each task's virtual runtime (`vruntime`) equal.
  - **vruntime**: An accounting variable. The task with the minimum `vruntime` is chosen to run next.
  - Vruntime advances more slowly for higher-priority (higher-weighted) tasks: $\Delta vruntime = \Delta T_{actual} \times \frac{W_{ideal}}{W_{task}}$.
  - **Data Structure**: Uses a **red-black tree** to store runnable tasks, ordered by `vruntime`. This makes picking the next task to run very efficient.

Context Switch

- **Context**: The complete state of a process (CPU registers, Program Counter, stack pointer, memory state).
- **Overhead**: **Direct** (time to save/load CPU state) and **Indirect** (performance loss from cache pollution, TLB flushes, etc.).
- **Types**: **Forced** (involuntary, due to an interrupt) vs. **Initiated** (voluntary, due to a syscall like `wait()`).
- **Mechanism**: The CPU state is saved to the task's kernel stack (in the `thread_struct` field of the PCB). A per-core hardware struct, the **TSS (Task State Segment)**, is updated to point to the new task's kernel stack.
- **Flow**: `schedule()` → `context_switch()` → `_switch_to_asm()` (saves/restores general-purpose registers) → `_switch_to()` (updates kernel stack pointer in TSS).

Synchronization

Concepts

- **Race Condition**: Program outcome depends unpredictably on the timing of concurrent executions.
- **Critical Section**: A piece of code that accesses shared data and must not be concurrently executed by more than one thread.
- **Atomicity**: An operation that executes as a single, indivisible, all-or-nothing unit. On multi-core CPUs, this requires hardware support (e.g., `lock` prefix on x86 instructions).
- **Memory Consistency**: Defines the order in which memory operations from one CPU core are visible to others. Enforced with a **memory fence** (or barrier).
- **Amdahl's Law**: Max speedup from parallelization, where s is the serial fraction and n is processors.

$$Speedup = \frac{1}{s + \frac{1-s}{n}} \leq \frac{1}{s}$$

Mechanisms

- **Lock**: A primitive with `acquire()` and `release()` operations to enforce mutual exclusion.
- **Spinlock**: A lock that causes a thread to "busy-wait" (spin in a tight loop) until the lock is free. Implemented with atomic instructions like `test_and_set` or `compare-and-swap`. The `volatile` keyword is used to prevent compiler reordering of accesses to the lock variable.
- **Mutex**: A lock that may cause the thread to sleep (**block**) if the lock is unavailable. Use a spinlock if the expected wait time is less than a context switch; otherwise, use a mutex.
- **Semaphore**: An integer counter with a waiting queue. `wait()` (P) decrements (and potentially blocks if 0); `signal()` (V) increments (and potentially wakes a waiting thread). Can be binary (like a mutex) or counting.
- **Condition Variable**: Used with a mutex to allow threads to wait for a specific condition to become true (`cond_wait`) without busy-waiting. Another thread uses `cond_signal` to notify waiters that the condition may now be true.

Deadlocks

- **Deadlock**: A set of processes are blocked, each waiting for a resource held by another process in the set.
- **Livelock**: Processes continuously change state in response to each other but make no forward progress.
- **Starvation**: A process is perpetually denied necessary resources to proceed.
- **Four Necessary Conditions for Deadlock**:
  1. **Mutual Exclusion**: At least one resource must be non-sharable.
  2. **Hold and Wait**: A process holds at least one resource and is waiting for another.
  3. **No Preemption**: A resource cannot be forcibly taken from a process.
  4. **Circular Wait**: A cycle of processes exists, e.g., P_0 waits for P_1's resource, P_1 waits for P_2's, and P_2 waits for P_0's.
- **Handling Methods**:
  - **Prevention**: Violate one of the 4 conditions. A common method is to enforce a total ordering on lock acquisition to prevent circular wait.
  - **Avoidance**: Use algorithms like the **Banker's Algorithm** to ensure the system never enters an unsafe state by analyzing resource requests.
  - **Detection & Recovery**: Allow deadlocks to occur, detect them (e.g., find a cycle in the resource allocation graph), and recover (e.g., kill a process).

Virtual Memory

Concepts

- **Virtual Memory (VM)**: An abstraction that provides each process with a private, large, contiguous address space, isolating it from other processes and from the physical memory layout.
- **Page** & **Frame**: VM is divided into fixed-size **pages** (e.g., 4KB). Physical memory is divided into same-sized **frames**.

- **Address Translation**: The **MMU** (Memory Management Unit) is hardware that translates Virtual Addresses (VAs) to Physical Addresses (PAs) using page tables.
- **Page Table**: A per-process data structure that maps virtual pages to physical frames.
- **PTE (Page Table Entry)**: An entry in the page table. Contains the physical frame number and control bits: **Present** (is page in RAM?), **Dirty** (has page been written to?), **Accessed**, Read/**Write**, **User**/Supervisor.
- **Page Fault**: A hardware trap to the OS, triggered when a program accesses a page that is not mapped in physical memory (e.g., 'Present' bit is 0).
  - **Major Fault**: Page must be fetched from disk.
  - **Minor Fault**: Page is in memory, but the mapping is missing (e.g., first access, Copy-on-Write).
- **TLB (Translation Lookaside Buffer)**: A fast hardware cache on the CPU for recent VA → PA translations to accelerate address translation. A TLB miss requires a hardware page table walk.

Paging & Swapping

- **On-Demand Paging**: Pages are loaded from disk into memory only when a page fault occurs for them.
- **Swap Area**: A dedicated space on the disk used to store pages that are temporarily evicted from physical memory.
- **Paging out**: Copying a page from DRAM to the disk (to the swap area for anonymous pages, or its file for named pages). Paging out a named page is only needed if its dirty bit is set.
- **kswapd**: A kernel thread that runs when free memory is low to reclaim page frames by paging them out.
- **Thrashing**: A state where the system spends excessive time paging data between memory and disk, making little progress on actual computation, because there is not enough physical memory to hold the working sets of active processes.

Page Replacement Algorithms

- **Goal**: When memory is full, choose a victim page to evict.
- **Belady's Optimal**: Replace the page that will be used furthest in the future (unrealizable, used for benchmarking).
- **LRU (Least Recently Used)**: Replace the page that has not been used for the longest time. Hard to implement perfectly.
- **Clock Algorithm**: An efficient approximation of LRU using a circular list and a single "referenced" bit per page frame.
- **Linux PFRA**: Uses two lists (**active**, **inactive**) and promotes/demotes pages between them to approximate LRU.

Linux Implementation

- **Multi-Level Page Tables**: x86-64 uses a 4-level hierarchy (PML4, PDPT, PDT, PT) to avoid needing large, contiguous page tables. The **CR3** register points to the top-level table (PML4).
- **mm_struct**: The kernel's primary memory descriptor for a process address space. Contains pointers to page tables (`pgd`), a list of memory regions (`mmap`), and usage counters (`mm_users`, `mm_count`).
- **vm_area_struct (VMA)**: Represents a contiguous memory region (e.g., stack, heap, code, mapped file). Has permissions like `VM_READ`, `VM_WRITE`, `VM_EXEC`, and flags like `VM_SHARED`.
- **mmap()**: Syscall to create a new VMA, mapping files into the address space (file-backed) or allocating anonymous memory.
- **Copy-on-Write (COW)**: On `fork()`, the parent's writable, private pages are shared with the child and marked as read-only in both processes' page tables. A page fault on a write triggers the kernel to make a private, writable copy for the writing process.

Storage & Filesystems

Storage Devices

- **HDD (Hard Disk Drive)**: Mechanical storage. Access time = seek time + rotational latency. Performance is best for sequential access.
- **SSD (Solid State Drive)**: Flash memory-based. Much lower latency for random access than HDDs. Cannot overwrite data in place; must erase larger blocks.
- **DMA (Direct Memory Access)**: A hardware mechanism allowing devices (like disks) to transfer data directly to/from main memory without involving the CPU, which is notified by an interrupt upon completion.

RAID

- **RAID 0 (Striping)**: Data is striped across disks. Provides the best performance but no redundancy. One disk failure leads to total data loss.
- **RAID 1 (Mirroring)**: Data is duplicated across disks. Provides full redundancy but has a 50% capacity cost.
- **RAID 4 (Striping with Parity Disk)**: Stripes data and stores parity information on a single dedicated disk. The parity disk becomes a write bottleneck.
- **RAID 5 (Striping with Distributed Parity)**: Stripes data and distributes parity blocks across all disks, avoiding the RAID 4 bottleneck. Protects against a single disk failure.
- **RAID 6 (Striping with Dual Parity)**: Distributes two independent parity blocks. Protects against two concurrent disk failures.

Filesystem Concepts

- **File**: A logical, non-volatile unit of information with a name, content, and metadata.
- **inode (index node)**: A data structure containing file metadata: permissions, size, owner, timestamps, and pointers to the file's data blocks. The filename is not in the inode.
- **dirent (directory entry)**: An entry in a directory file that maps a human-readable filename to an inode number.
- **Links**:
  - **Hard Link**: A second directory entry pointing to the same inode. Must be on the same filesystem. The file is deleted only when its link count (in the inode) drops to zero.

  - **Symbolic (Soft) Link**: A special file that stores a path string to another file. Can cross filesystems and can be "dangling" if the target is removed.
- **File Descriptor (FD)**: A small, per-process integer handle for an open file. It is an index into the per-process **File Descriptor Table (FDT)**.
- Each FDT entry points to a system-wide **File Object** (representing an open file instance), which in turn points to the file's **inode**.

Filesystem Implementation

- **Layout**: A filesystem on disk typically contains a **superblock** (global FS metadata), **bitmaps** (to track free inodes and data blocks), an **inode table**, and **data blocks**.
- **Allocation Methods**:
  - **Multi-Level Index**: An inode contains direct pointers to data blocks, plus single, double, and triple indirect pointers to handle very large files.
  - **Extents**: A contiguous run of data blocks, stored as (start block, length) pairs. Reduces fragmentation and metadata overhead compared to block pointers. Used in ext4.
  - **FAT (File Allocation Table)**: A simple FS that uses a table in memory to form linked lists of data blocks for each file. No inodes; metadata is in the directory entry.
- **Journaling**: Writing metadata changes to a transaction log (**journal**) before applying them to the main filesystem. This ensures filesystem consistency after a crash. Used in ext3/ext4.
- **Path Resolution**: Translating a path like '/home/user/file.txt' to an inode involves reading each directory in the path to find the next component's inode number. The user needs execute (search) permission on all directories in the path.

Networking

- **Protocol Stack**: Layered model for network communication. e.g., L5-Application (HTTP), L4-Transport (TCP/UDP), L3-Network (IP), L2-Data Link (Ethernet).
- **TCP (Transmission Control Protocol)**: Reliable, connection-oriented, stream-based protocol. Uses a 3-way handshake to establish connections, sequence numbers and ACKs for reliability, and manages flow control and congestion control.
- **UDP (User Datagram Protocol)**: Unreliable, connectionless, datagram-based. Low overhead, but packets can be lost, reordered, or duplicated.
- **IP Address**: A logical address for a host, composed of a **network part** and a **host part**, as defined by the subnet mask.
- **Socket**: A communication endpoint represented by a file descriptor. A TCP connection is uniquely identified by a 5-tuple: (protocol, source IP, source port, destination IP, destination port).
- **Client-Server Flow (TCP)**: Server: `socket` → `bind` → `listen` → `accept`. Client: `socket` → `connect`.
- **Ethernet (L2)**: The dominant LAN technology. Uses **MAC addresses** (48-bit unique hardware addresses) for local delivery.
- **Key Protocols**:
  - **ARP** (Address Resolution Protocol): Translates an IP address to a MAC address on a local network.
  - **DHCP** (Dynamic Host Config. Protocol): Allows a host to automatically obtain an IP address and other network settings.
  - **NAT** (Network Address Translation): Allows multiple devices on a private network to share a single public IP address.