System Execution Architectures

- **Multiprogramming:** Organizes jobs so the CPU always has a process to execute, keeping a subset of all jobs in main memory. This is heavily managed by the **long-term scheduler**, which decides which jobs are admitted into the ready queue.
- **Multitasking (or Timesharing):** A logical extension of multiprogramming. The CPU switches between jobs so rapidly (in milliseconds) that users can interact with each program concurrently, creating an interactive environment. This rapid switching is controlled by the **short-term scheduler**.
- **Multiprocessing:** While multitasking fakes simultaneous execution on one CPU, multiprocessing uses multiple physical CPUs sharing the same memory and bus to physically execute tasks at the same time, increasing throughput.
- **Multithreading:** Normally, a process has a single "program counter," meaning it executes its code sequentially, one instruction at a time. Multithreading allows a single process to have multiple "threads of control," where each thread gets its own program counter. This allows the process to execute different parts of its code at the exact same time. To manage this concurrency, the operating system must store the individual details and program counters for each thread inside the Process Control Block (PCB).

Process Management & Memory

- **Context Switching and overhead:** When the CPU needs to switch from one process to another. The **Kernel** stops the current process, saves its exact state into its **Process Control Block (PCB)**, and loads the saved PCB of the incoming process. This process is often triggered by an interrupt and executed by the **dispatcher**. When the CPU switches to another process, the operating system must save the state of the old process into its PCB, and then load the saved state from the new process's PCB. The time it takes to do this is considered **"overhead"**, meaning the system does no useful work while the switch is happening.

The short-term scheduler (or CPU scheduler) is the decision-maker that selects which process in the ready queue should be executed next. The dispatcher is the mechanism that actually enacts this decision, moving the selected process from the "ready" state into the "running" state.

- **Swapping:** Moving an entire process out of main memory and onto a disk, then bringing it back later to resume. Swapping is necessary when all active processes do not fit into the computer's main memory. To free up space, the **medium-term scheduler** removes a partially executed process from memory, swaps it out to a disk, and brings it back later to continue execution.
- **Virtual Memory:** A technique that allows a process to execute even if it is not completely loaded into physical memory. It abstracts physical memory to give each program the illusion of having its own large, private memory space. It achieves this by bringing data into RAM only when it is actually needed.

Process Hierarchies

- **Zombies vs. Orphans:** When a process finishes, it waits for its parent to acknowledge its termination (using a `wait()` system call). If a process has terminated but the parent hasn't acknowledged it yet, it sits in the system as a **zombie**. If the parent terminates entirely without ever calling `wait()`, the left-behind child becomes an **orphan**.
- **Starvation** occurs when a process is continuously denied the resources it needs to execute. For example, in a scenario where multiple processes want to read or write to a shared file, a writer process might experience starvation if a continuous stream of new reader processes keeps getting priority, meaning the writer never gets its turn. (Readers-Writers problem in note 4 causes starvation)

Communication

- **Interprocess Communication (IPC):** The mechanism that allows cooperating processes to share data. The two main models are **shared memory** (processes read/write to the same memory space) and **message passing** (the OS moves packets of data between processes).
- **Direct vs. Indirect Communication:** In message passing, **direct communication** means processes must explicitly name each other to link up (e.g., `send(P, message)`). **Indirect communication** means processes send and receive messages through shared mailboxes or ports instead of addressing each other directly.

Interacting with the System

- **System Call:** The way a program requests a service from the OS. It triggers a software interrupt (a trap) that shifts the system from User Mode to Kernel Mode. The **Interrupt Service Routine (ISR)** then determines what action to take, executes the request in the Kernel, and returns control to the user.
- **API vs. CLI:** An **API (Application Program Interface)** is for *programs*—it is a library of functions that hides the complex details of system calls from developers. A **CLI (Command Line Interface)** is for *humans*—it allows users to type commands directly to the OS.

To trigger a system call, applications generally use an Application Program Interface (API). When a program calls an API function, the system's run-time support library takes over. It looks up the requested system call in an indexed table, triggers an interrupt to shift into Kernel Mode, and invokes the intended system call inside the kernel. CLIs do not send system calls directly; instead, the CLI executes system programs, and those system programs use APIs to make the necessary system calls.

Here is the breakdown of the differences between shells, the CLI, and batch processing based on your materials:

- **CLI (Command Line Interface):** This is the general interface style that allows you to interact with the operating system by typing commands instead of using a mouse. It acts as a command interpreter for direct command entry.
- **Shells:** These are the specific system programs that implement the CLI. For example, the **Bourne Shell** is a specific "flavor" of command interpreter that fetches your command and executes it. Essentially, the CLI is the *concept*, and the shell is the *program* you actually use.
- **Batch Processing:** Unlike the interactive CLI, batch processing involves executing a series of programs or commands automatically **without human interaction**.

In short: the **CLI** is the style of interface, the **shell** is the software that runs it, and **batch** is running a sequence of tasks automatically.

The CLI is the general interface style that allows you to enter commands directly. A "shell" is the specific system program that implements the CLI, acting as the command interpreter that fetches the command you typed and executes it (like a compiler).

Large-Scale Computing

- **Computing Environments** This is the broad umbrella for how systems are structured for user interaction and resource sharing.
    - **Traditional:** Evolving from standalone PCs to networked systems in office environments, often using terminals attached to mainframes providing batch and timesharing services.
    - **Client-Server:** A model where smart PCs (clients) request services (like database or file storage) from central servers.
    - **Peer-to-Peer (P2P):** A distributed model where all nodes are equal, acting as both clients and servers without a central hub.
- **Distributed Environments** These are a collection of separate, physically independent systems linked by a network (LAN, WAN, or MAN).
    - A Network Operating System allows these independent computers to exchange messages, creating the illusion for the user that they are interacting with a single unified system.
    - In these environments, the computers do not have to be doing the exact same thing; they often use models like Client-Server (where many clients request services from a central server) or Peer-to-Peer (where all nodes act as both clients and servers).
- **Clustered Systems** A clustered system is a much more specialized and highly coordinated setup.
    - Instead of just communicating over a standard network, these clustered nodes usually share a central storage space via a Storage-Area Network (SAN). The slides make this comparison because both setups combine multiple units to act as a single, powerful entity, just at different scales.
    - It consists of loosely or tightly connected computers that are specifically configured by software to perform the *exact same task*.

- In a **multiprocessor system**, you have multiple *processors (CPUs)* working together inside a single physical computer, sharing the same main memory and hardware bus.
- A **clustered system** scales this concept up. Instead of multiple CPUs inside one box, you have multiple *complete computers* (systems) working together over a network, usually sharing storage via a Storage-Area Network (SAN).

Basically, a multiprocessor system combines multiple CPUs to act as one computer, while a clustered system combines multiple computers to act as one giant system.

The primary goal of a cluster is reliability and power:

- **High-Availability:** They are designed to survive hardware failures. For example, in *asymmetric clustering*, one machine actively works while another sits in "hot-standby" mode, waiting to instantly take over if the first one fails. In *symmetric clustering*, multiple nodes run applications and monitor each other.
- **High-Performance Computing (HPC):** Clusters can also be used to run heavily parallelized applications, combining the power of the computers to solve complex problems faster.

Special-purpose systems: Is the broad category for computers designed to perform a very specific, limited job rather than general computing. Both embedded systems and handheld systems fall under this umbrella.

- **Embedded Systems:** These are the most common type of special-purpose system. They are usually built directly into a larger piece of equipment, like an automobile, and often have little to no user interface. They typically run real-time operating systems to handle specific hardware tasks seamlessly in the background.
- **Handheld Systems:** These are portable special-purpose devices like smartphones and PDAs. Because they are small and battery-operated, they are considered "resource poor". They have limited CPU, memory, and input/output capabilities, so their operating systems are stripped down and highly optimized for battery life and usability,,.

In short, **all embedded and handheld devices are special-purpose systems**, but embedded systems run machines quietly in the background, while handhelds are battery-optimized personal devices.

The **Dispatcher** and the **Short-Term Scheduler** cooperating as a team, and where they sit in Process Management/Scheduling:

- The **Short-Term Scheduler** is simply the decision-maker; it looks at the ready queue and picks which process gets to go next.
- The **Dispatcher** is the action-taker that actually gives physical control of the CPU to the process. It is the component specifically responsible for "switching context" and jumping to the correct location to restart the program.

1. Does the PCB work similarly to the dispatcher? No, they do very different things.

- The **PCB (Process Control Block)** is just a passive storage box (a data structure) that holds a process's saved information, like its current state and instruction pointer.
- The **Dispatcher** is the active worker that reads from and writes to the PCB to actually swap the programs in and out of the CPU.

2. Are there other components they cooperate with? You already know the main pieces! The only other major components they cooperate with are **Queues**.

- When programs are waiting to use the CPU, they sit in a **Ready Queue**.
- If they are waiting for a file or user input, they sit in a **Device Queue** (or I/O Queue).
- The Schedulers simply look at these queues to decide who goes next, and the Dispatcher physically takes them from the queue and gives them the CPU.

Context Switch VS Swapping:

- **Context Switching:** This happens entirely within the CPU and main memory. It is simply the **Dispatcher** pausing one process and loading another by updating their PCBs.
- **Swapping:** Taking a process completely out of the main memory queue and moving it to stable secondary storage (like a hard drive) to free up space is called swapping.
- **The Scheduler Responsible:** The **Medium-Term Scheduler** is the specific component responsible for swapping processes in and out of memory. While the **Dispatcher** is responsible for context switching.

Checkpoints VS. Crash Dump:

- **Crash dump** is a file generated specifically when an **operating system fails**; it captures the state of the **kernel memory** at the exact time of the failure to help with debugging.
- **Checkpoints** are a planned mechanism used to **shorten recovery time** by periodically saving data. During a checkpoint, the system moves log records and modified data from volatile storage to stable storage, ensuring that only transactions occurring since the most recent checkpoint need to be processed during a recovery.

Essentially, a crash dump is a "snapshot" of a disaster for later analysis, while a checkpoint is a "save point" to prevent long delays after a system restart.