Chapter 1(Intorduction):

Questions:

What is an Operating System?

An **Operating System (OS)** is **system software** that manages computer hardware and software resources while providing services for application programs. It acts as an interface between the **user** and the **hardware**, enabling efficient execution of programs.

Process Management – Handles process creation, execution, and termination.
Memory Management – Allocates and deallocates memory to processes.
File System Management – Organizes, stores, and retrieves data in files.
Device Management – Controls hardware devices like keyboards, printers, and storage.
Security & Access Control – Protects data and system resources from unauthorized access.
User Interface (UI) – Provides command-line (CLI) or graphical (GUI) interfaces for user interaction.

What is the difference between an Interrupt and a System Call?

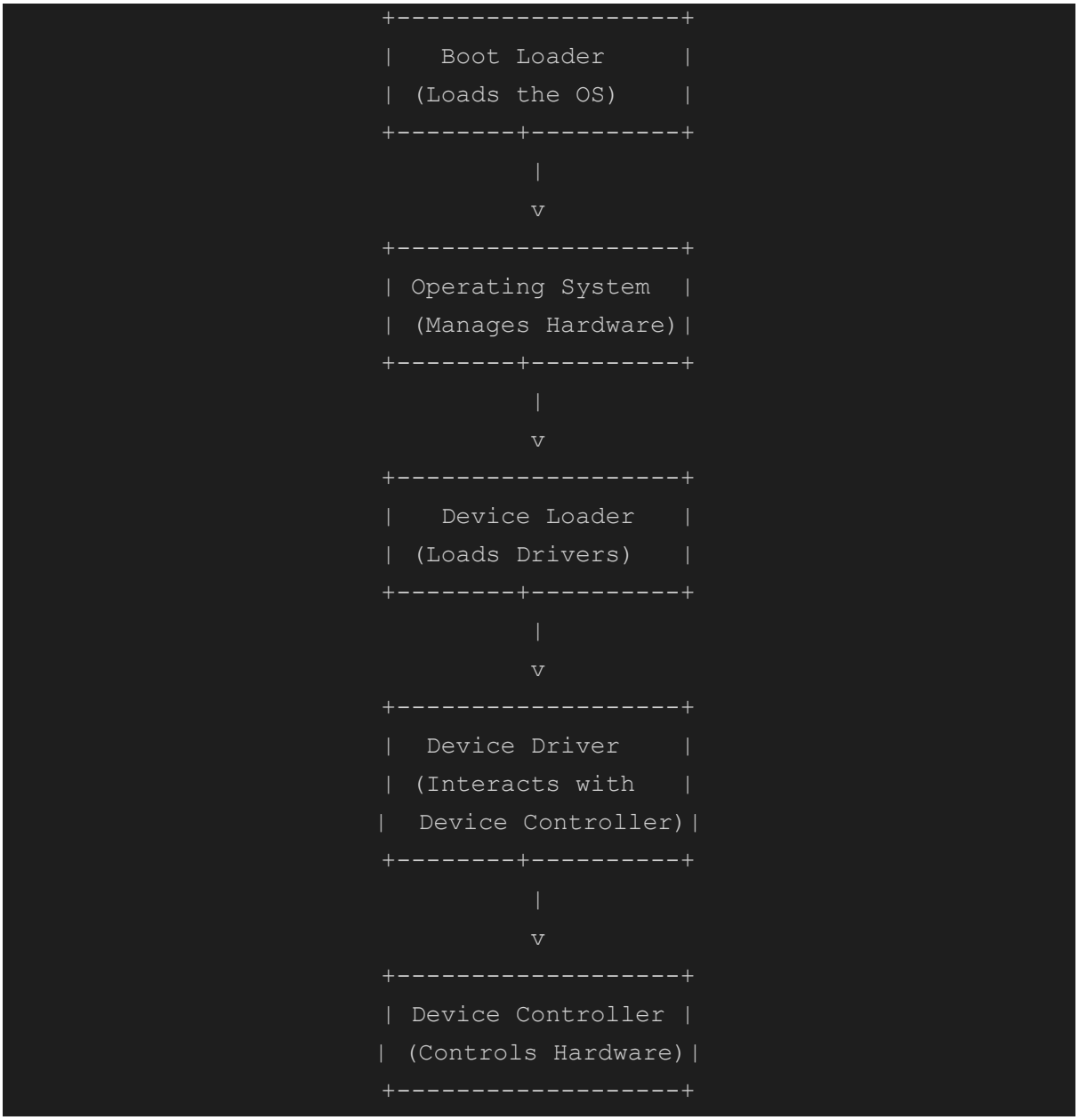
Feature	Interrupt	System Call
Definition	An event that signals the CPU to stop its current execution and handle a high-priority task.	A request made by a program to the OS to perform a privileged operation.
Trigger	Can be triggered by hardware (external) or software (internal).	Explicitly invoked by a user program using an API.
Source	Hardware devices (e.g., keyboard, timer, disk I/O) or software-generated exceptions (e.g., divide by zero).	System API functions (e.g., read(), write(), fork()).
Purpose	Handles urgent events like I/O completion, errors, or timeouts.	Requests OS services such as file operations, memory allocation, or process creation.
Execution	An interrupt handler (part of OS) executes when an interrupt occurs.	The OS executes the requested system call.
Mode Change	Can switch from user mode to kernel mode (hardware interrupts).	Always switches from user mode to kernel mode.
Examples	- Hardware Interrupts: Pressing a key, mouse movement, disk I/O completion. - Software Interrupts: Division by zero, invalid memory access.	- open(), read(), write(), fork(), exec() in Linux.

Key points:

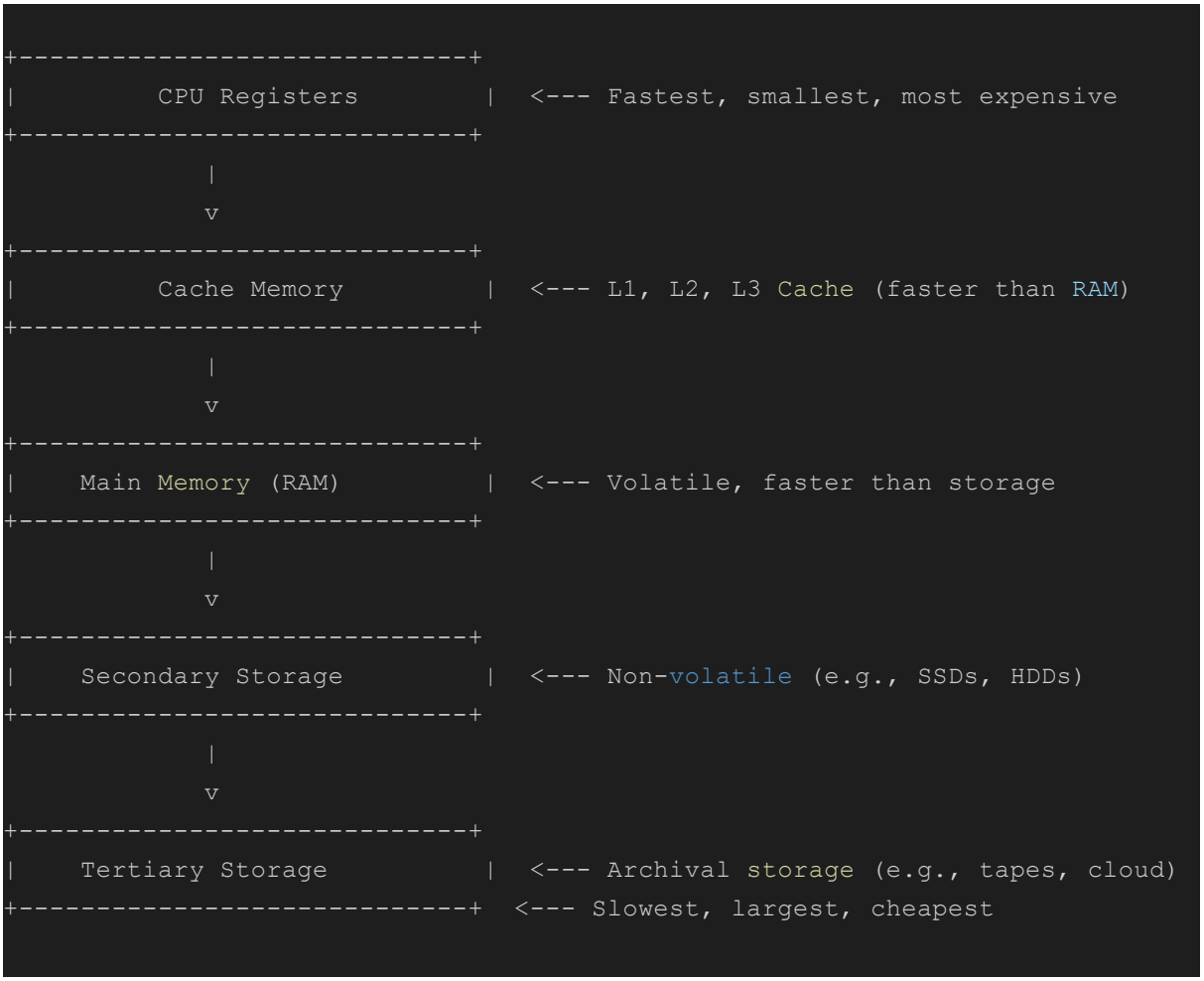
- Interrupts are event-driven, while system calls are request-driven.
- Interrupts can be hardware- or software-triggered, while system calls are always software-initiated.
- System calls invoke OS services, whereas interrupts notify the CPU of urgent events.

Explain what is Device Loader, Device Driver, and Device Controler.

Flow of control.



What is the hierarchy of memory devices?



Comparison Table

Memory Level	Speed	Size	Cost	Volatility	Location	Purpose
CPU Registers	Fastest	Smallest (KB)	Most Expensive	Volatile	Inside the CPU	Stores data actively used by the CPU.
Cache Memory	Very Fast	Small (MB)	Expensive	Volatile	On-chip (L1, L2, L3)	Stores frequently accessed data.
Main Memory (RAM)	Fast	Moderate (GB)	Moderate	Volatile	Connected to CPU via bus	Stores data and instructions in use.
Secondary Storage	Slow	Large (TB)	Cheap	Non-Volatile	External (SSD, HDD)	Stores data persistently for the long term.
Tertiary Storage	Slowest	Largest (PB)	Cheapest	Non-Volatile	External (Tapes, Cloud)	Used for backup and archival storage.

What is the difference between a process running concurrently and parallelly?

Explain the difference between Multitasking and Multiprogramming.

Feature	Multitasking	Multiprogramming
Definition	Executing multiple tasks or processes simultaneously by rapidly switching between them.	Running multiple programs concurrently by keeping them in memory and switching between them.
Focus	Maximizing CPU utilization and improving user responsiveness.	Maximizing CPU utilization by keeping it busy at all times.
User Interaction	Designed for interactive systems where users expect immediate responses.	Designed for batch processing systems where user interaction is minimal.
Time Sharing	Uses time-sharing to allocate CPU time slices to each task.	Uses context switching to switch between programs when one is waiting for I/O.
Goal	To provide a smooth and responsive experience for users running multiple applications.	To maximize CPU usage and throughput by overlapping I/O and CPU operations.
Example	A user running a web browser, music player, and word processor at the same time.	A system running multiple batch jobs (e.g., payroll processing, report generation).
Resource Management	Manages resources for multiple tasks simultaneously to ensure fair allocation.	Manages resources for multiple programs concurrently to keep the CPU busy.
Context Switching	Frequent context switching between tasks to give the illusion of simultaneous execution.	Context switching occurs primarily when a program is waiting for I/O.
User Experience	Provides a responsive and interactive experience for the user.	Focuses on efficiency rather than user interaction.
Common Use Case	Modern operating systems (e.g., Windows, macOS, Linux) for desktops, laptops, and mobile devices.	Older batch processing systems and mainframes.

Key Differences:

- **Multitasking** is about running multiple tasks **simultaneously** (or appearing to do so) to improve user experience and responsiveness. It is commonly used in modern interactive systems.
- **Multiprogramming** is about running multiple programs **concurrently** to maximize CPU utilization and throughput. It was more common in older batch processing systems.

Summary:

- **Multitasking** is user-centric, focusing on responsiveness and smooth performance for interactive applications.
- **Multiprogramming** is system-centric, focusing on efficiency and keeping the CPU busy by overlapping I/O and CPU operations.

What is the difference between Policies and Mechanisms?

What is the difference between Kernle Mode and User Mode and how the switching of modes is done?

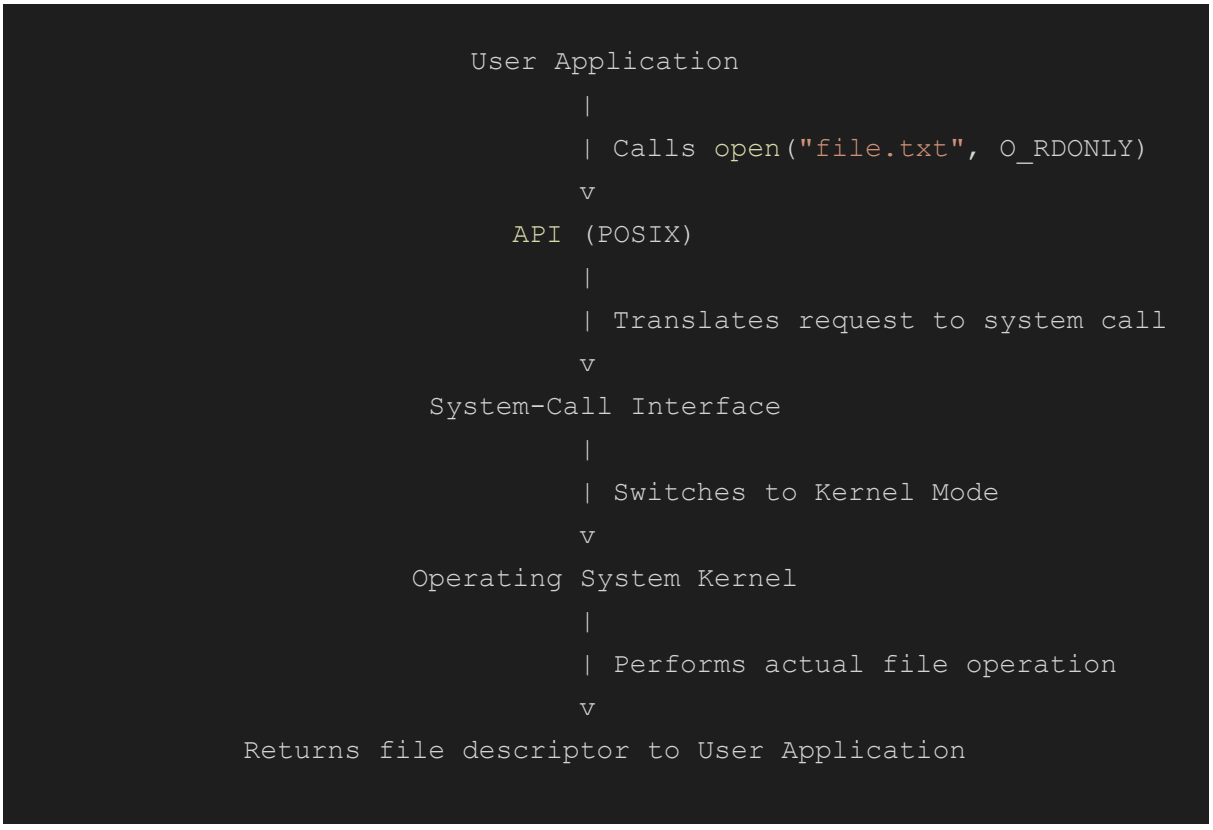
Chapter 2 (Operating system structure)

API (2.3.2)

Feature	API	System Call
Definition	A set of functions for software interaction	A direct request to the OS kernel
Level	High-level	Low-level
Performance	Faster (abstracts complexity)	Slower (direct kernel access)
Example	printf(), fopen()	write(), open()

The main idea is that a user application does not need to know the internal details of system call execution—only how to use the **API** that interacts with the **operating system**.

The **API hides system call complexity** from the programmer.
The **system-call interface** acts as a bridge between user programs and the OS.
The **runtime environment (RTE)** helps manage system call execution.
Example: The `open()` function in C ultimately invokes a system call in the OS kernel.



Parameter Passing in System Call

Method	How it Works	Advantages	Disadvantages
Registers	Store parameters directly in CPU registers	🔹 Very fast	❌ Limited by the number of registers
Memory Block (Table)	Store parameters in a block and pass its address in a register	🔹 Supports many parameters	❌ Slightly slower (requires memory access)
Stack	Push parameters onto the stack before making the system call	🔹 No limit on number of parameters	❌ Slowest due to memory access

Types of System Calls (2.3.3)

Category	Example System Calls	Purpose
Process Control	fork(), exec(), exit(), wait()	Create, execute, and terminate processes
File Management	open(), read(), write(), close()	File operations (read, write, delete)
Device Management	ioctl(), read(), write()	Interact with hardware devices
Information Maintenance	getpid(), getuid(), sysinfo()	Get system and process info
Communication	pipe(), msgget(), socket()	Interprocess and network communication
Protection	chmod(), chown(), umask()	Set file permissions and security

Chapter 3 (Processes)

- ☐ Explain Context Switching.
- ☐ Explain the state cycle of a process (process life cycle).
- ☐ Explain Cascading termination, Orphan process, and Zombie process.
- ☐ Compare the advantages and disadvantages of two methods of IPC, Message Sharing and Shared Memory.

Definitions:

- **Process**: is a program in execution
-
- **Process Control Block**: is a data structure used by the operating system to store information about a process. It acts as a record of the process's attributes and is essential for process management. Contents of a typical PCB:
    - Process ID (PID)
    - Process State
    - Program Counter
    - CPU Registers
    - Memory Management Information
    - Scheduling Information
    - I/O Status Information
    - Accounting Information
    -
- **Thread:** is the smallest unit of execution within a process. It represents a sequence of instructions that the CPU can execute independently.

Resource allocation to a process (3.1.1)

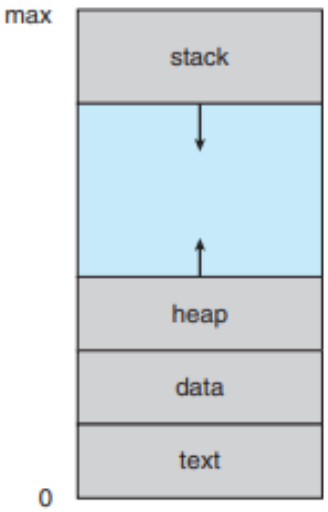


Figure 3.1 Layout of a process in memory.

Process Life Cycle (3.1.2)

A **process state cycle** represents the different stages a process goes through from creation to termination. The **five main process states** are:

1. **New** – The process is created but not yet ready for execution.
2. **Ready** – The process is waiting for CPU allocation.
3. **Running** – The process is being executed by the CPU.
4. **Waiting (Blocked)** – The process is waiting for an event (e.g., I/O completion).
5. **Terminated** – The process has finished execution.

The following describes the transitions between these states:

1. **New → Ready**:
   - A process is created and placed in the ready queue.
2. **Ready → Running**:
   - The CPU scheduler selects a process from the ready queue and assigns it to the CPU.
3. **Running → Waiting**:
   - If the process requests I/O or waits for an event, it moves to the waiting state.
4. **Running → Ready**:
   - If the process is interrupted (e.g., time quantum expires in a time-sharing system), it moves back to the ready state.
5. **Waiting → Ready**:
   - Once the process's I/O operation is completed, it moves back to the ready queue.
6. **Running → Terminated**:
   - When the process completes execution or is forcefully stopped, it enters the terminated state.

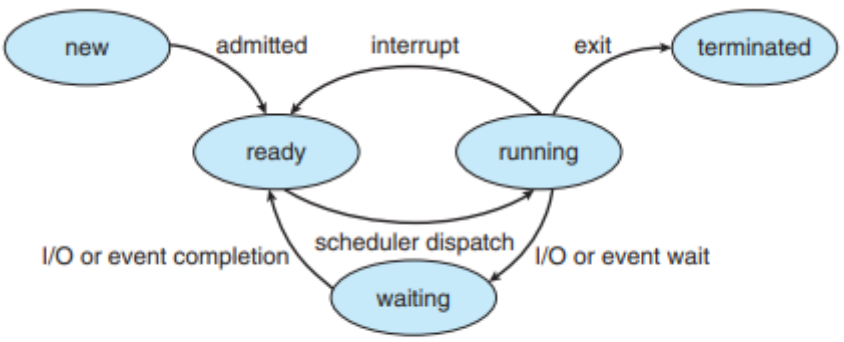
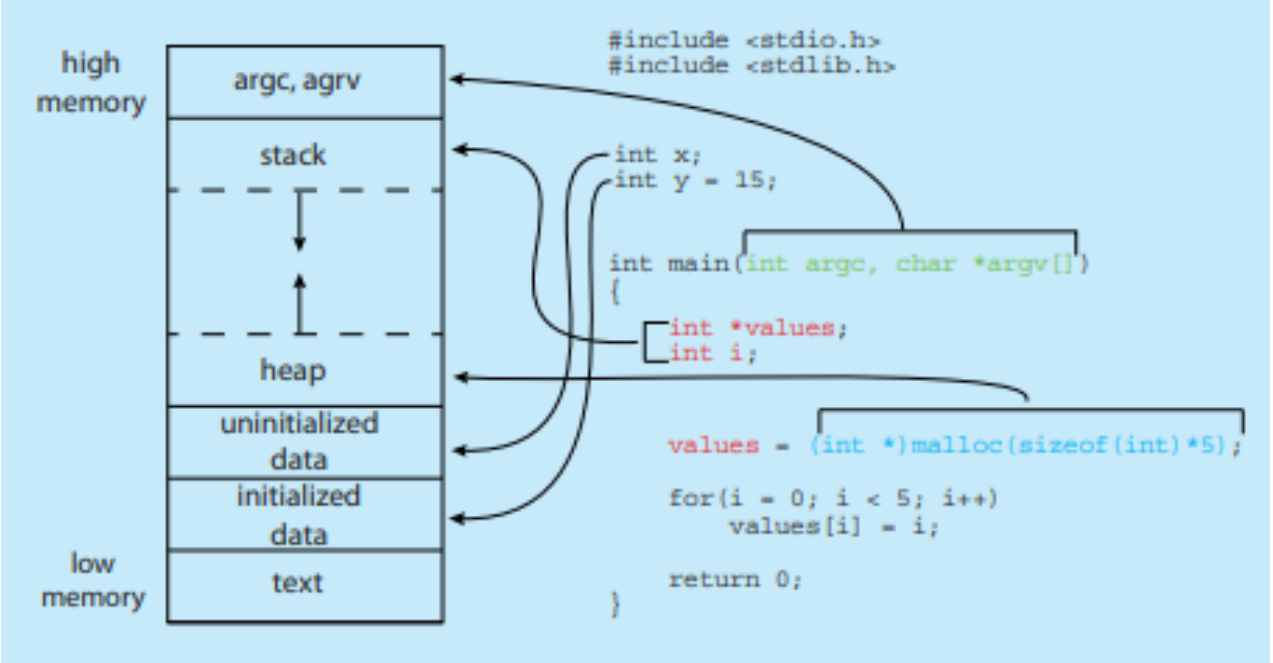


Figure 3.2 Diagram of process state.

Memory layout of a program in C language



Process Control Block (3.1.3)

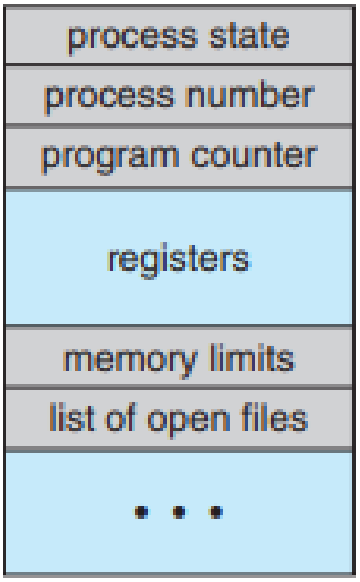


Figure 3.3 Process control block (PCB).

Scheduling queues (3.2.1)

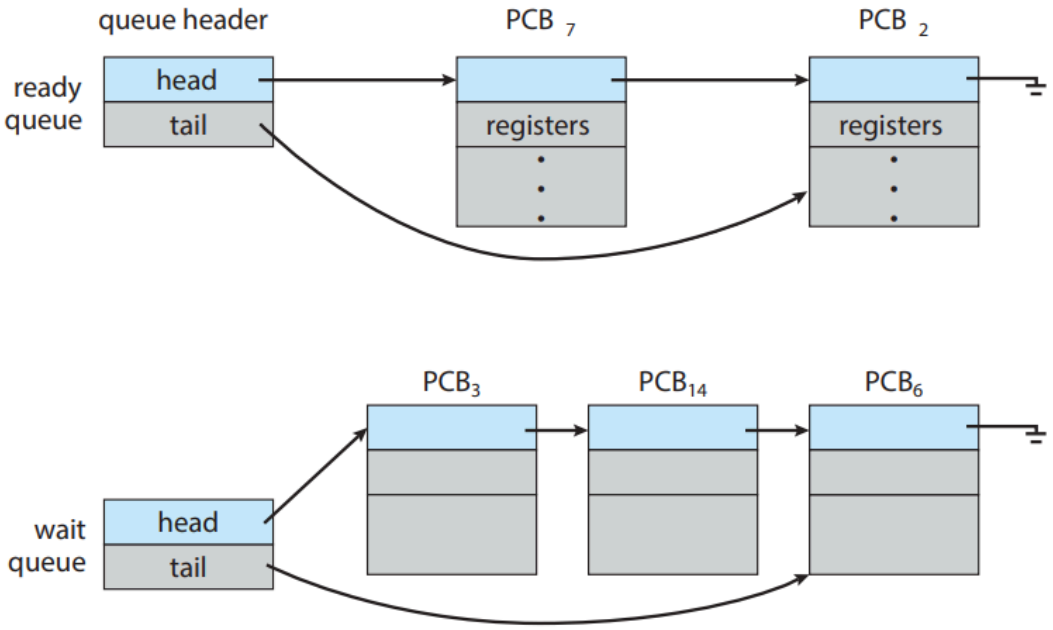


Figure 3.4 The ready queue and wait queues.

Queuing diagram (3.2.1)

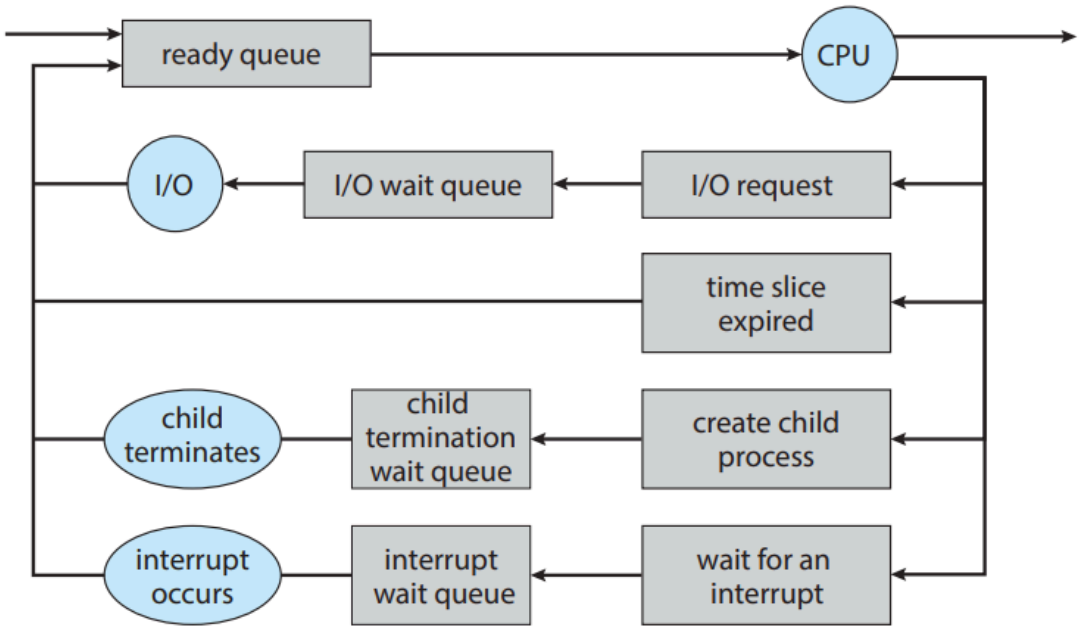


Figure 3.5 Queueing-diagram representation of process scheduling.

Types of Process Termination (3.2.2)

1. **Zombie Process:** is a process that has completed execution but still has an entry in the process table because its parent process has not yet read its exit status. It is also known as a defunct process.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>   // For fork(), sleep()
#include <sys/types.h> // For pid_t
#include <sys/wait.h>  // For wait()

int main() {
   pid_t pid = fork(); // Create a child process

   if (pid > 0) {
       // Parent process
       printf("Parent process (PID: %d) created child process (PID: %d).\n", getpid(), pid);
       sleep(5); // Simulate delay before calling wait()
       printf("Parent calling wait() to clean up zombie process.\n");
       wait(NULL); // Parent collects child's exit status, preventing a zombie process
   }
   else if (pid == 0) {
       // Child process
       printf("Child process (PID: %d) executing.\n", getpid());
       sleep(2); // Simulate execution time
       printf("Child process exiting.\n");
       exit(0); // Child process terminates, becoming a zombie until parent calls wait()
   }
   else {
       // Fork failed
       perror("Fork failed");
       return 1;
   }

   printf("Parent process exiting normally.\n");
   return 0;
}
```

2. **Orphan Process:** is a child process whose parent process has terminated before the child itself has finished execution. When a parent process terminates, its orphaned child processes are adopted by the `init` process (PID 1) in Linux/Unix systems, which continues to manage them.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
   pid_t pid = fork(); // Create a child process

   if (pid > 0) {
       // Parent process
       printf("Parent process (PID: %d) is terminating.\n", getpid());
       exit(0); // Parent exits, making the child an orphan
   }
   else if (pid == 0) {
       // Child process
       sleep(5); // Simulate some work
       printf("Orphan child process (PID: %d), adopted by init (PPID: %d).\n", getpid(), getppid());
   }
   else {
       // Fork failed
       perror("Fork failed");
       return 1;
   }

   return 0;
}
```

3. **Cascading Termination:** is a process termination mechanism in which the termination of a parent process automatically leads to the termination of all its child processes. This ensures that no orphan processes are left running unintentionally.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int main() {
   pid_t pid = fork(); // Create a child process

   if (pid > 0) {
       // Parent process
       printf("Parent process (PID: %d) created child process (PID: %d).\n", getpid(), pid);
       sleep(3); // Simulate work
       printf("Parent process is terminating.\n");
       exit(0); // Parent terminates
   }
   else if (pid == 0) {
       // Child process
       printf("Child process (PID: %d) is running.\n", getpid());
       sleep(10); // Child tries to keep running
       printf("Child process still running (PID: %d, PPID: %d).\n", getpid(), getppid());
   }
   else {
       // Fork failed
       perror("Fork failed");
       return 1;
   }

   return 0;
}
```

Context Switching (3.2.3)

Context switching is the process of saving the state of a currently running process or thread and restoring the state of another process or thread so that execution can resume. It allows the operating system to multitask by switching between multiple processes or threads efficiently.

Steps in Context Switching

- Saving current process state
- Load or restore another process state
- Resume execution

Types of Context Switching

1. Process context switch
2. Thread context switch

Implementation: Multitasking and Multiprogramming

Disadvantage: Overhead – Context switching takes time and consumes CPU cycles without doing useful work.

Example:
If you are running a video player and a web browser simultaneously, the OS will perform context switching to allocate CPU time to both applications, making it appear as if they are running simultaneously.

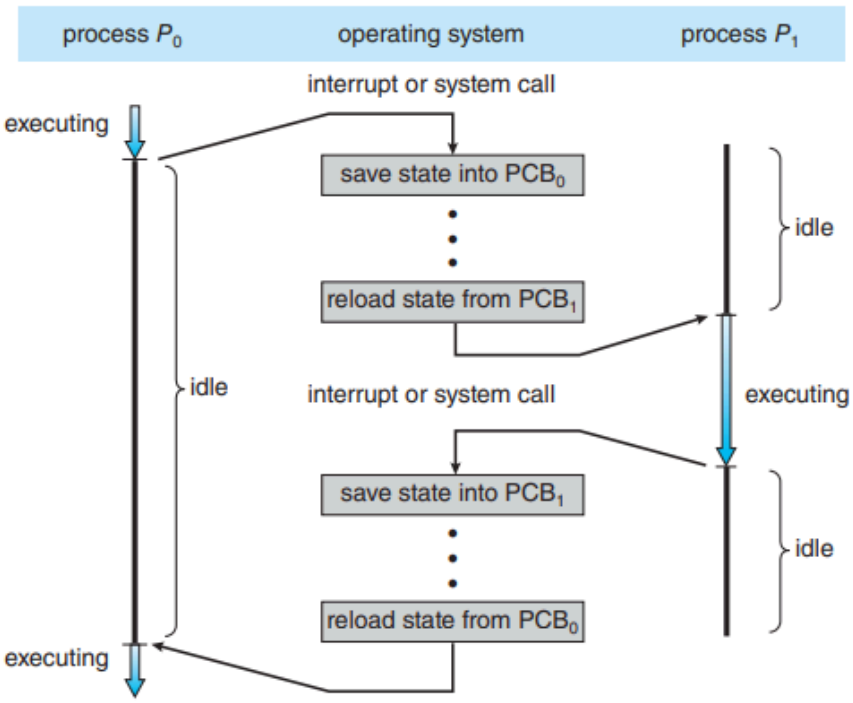


Figure 3.6 Diagram showing context switch from process to process.

Interprocess Communication (3.4)

Feature	Message Passing	Shared Memory
Definition	Processes exchange data by sending and receiving messages via the OS kernel.	Multiple processes share a common memory region for direct data exchange.
Speed	Slower (due to kernel involvement and system calls).	Faster (direct access to memory, minimal OS intervention).
Synchronization	Built-in (synchronization is implicit via messaging).	Requires additional synchronization (e.g., semaphores, mutexes).
Complexity	Simpler to implement, as the OS handles communication.	More complex, as processes must manage synchronization and consistency.
Overhead	Higher (involves system calls, context switching, and copying data).	Lower (direct memory access, no kernel overhead for data transfer).
Data Consistency	Ensured by OS mechanisms.	Must be manually managed by processes.
Security	More secure (OS enforces access control).	Less secure (all processes with access can modify data).
Scalability	Better suited for distributed systems (e.g., across networked computers).	Best for local processes on the same system.
Use Cases	Ideal for networked IPC, small data exchange (e.g., sockets, pipes, message queues).	Ideal for high-speed IPC, large data sharing (e.g., databases, shared buffers).

	Shared Memory	Message Passing
Advantages	✔ Faster (direct memory access) ✔ Low overhead (no system calls needed) ✔ Efficient for large data transfer	✔ Simpler to implement (OS manages communication) ✔ More secure (access control by OS) ✔ Works well for distributed systems
Disadvantages	❌ Requires synchronization (e.g., semaphores) ❌ Less secure (all processes can access shared memory) ❌ Complex to manage consistency	❌ Slower (due to kernel involvement) ❌ Higher overhead (data copying and context switching) ❌ Less efficient for large data transfers

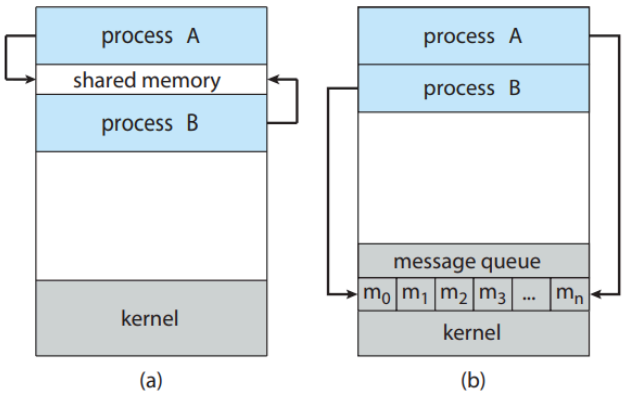


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.