UNIT-I

Operating system:

An operating system acts as an intermediary between the user and the computer hardware. The primary purpose of an operating system is to provide an environment in which a user can execute programs in a convenient, efficient, and safe manner.

The operating system is a software layer that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent programs from interfering with one another or with the system's normal functioning.

Internally, operating systems may differ widely in structure because they can be designed using many different approaches. Designing a new operating system is a major and complex task, and therefore, the goals of the system must be clearly defined before development begins.

Since an operating system is large and complex, it needs to be built piece by piece. Each piece or component should be a well-defined part of the system with clear inputs, outputs, and specific functions. This modular approach ensures easier development, testing, and maintenance of the operating system.

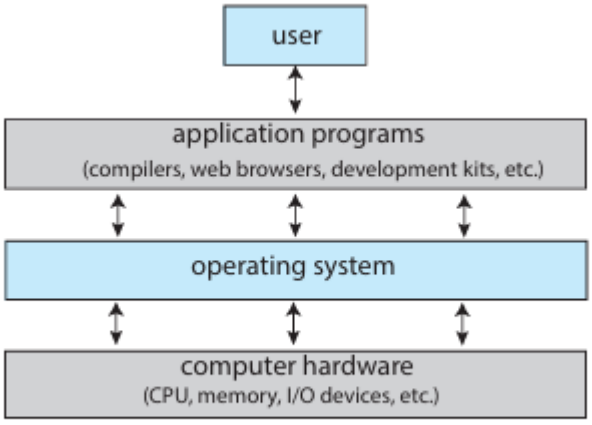


Figure 1.1 Abstract view of the components of a computer system.

Functions of Operating System:

Process Management

- Creates, schedules, and terminates processes
- Manages CPU allocation
- Handles deadlocks and process communication

Memory Management

- Allocates and deallocates memory
- Manages RAM efficiently
- Supports paging, segmentation, and virtual memory

File Management

- Creates, deletes, reads, and writes files
- Manages directories and file permissions
- Controls file allocation and protection

I/O Device Management

- Controls I/O devices using drivers
- Handles interrupts, buffering, and caching
- Coordinates communication between hardware and software

Secondary Storage Management

- Manages hard disks and SSDs
- Performs disk scheduling (FCFS, SSTF, SCAN, etc.)
- Manages free space and file placement

Security and Protection

- Prevents unauthorized access
- Provides authentication and access control
- Protects memory, files, and system resources

User Interface

- Provides CLI (command line interface) and GUI (graphical interface)
- Helps users interact with the system easily

Error Detection and Handling

- Detects hardware and software errors
- Takes corrective actions and provides warnings

OS structures:

1. Monolithic Structure of Operating System :

The monolithic structure is the earliest and most traditional way of designing an operating system. In this structure, the entire OS is built as one single, large program that runs in kernel mode.

Characteristics

- All OS components such as:
    - Process Management
    - Memory Management
    - File System
    - Device Drivers
    - I/O Management
    - System Call Interface
      — are tightly integrated into one kernel.
- All these services can directly communicate with each other without restrictions.
- System calls invoke kernel functions directly.

How it works

- When a user program makes a system call (e.g., read, write), control switches from user mode to kernel mode.
- Kernel executes the requested service and returns control back.

Advantages

1. High Performance:
   No overhead of message passing or modular communication.
2. Fast System Calls:
   Functions are directly available inside the kernel.
3. Efficient CPU and memory utilization:
   Less switching between services.

Disadvantages

1. Poor Maintainability:
   As everything is combined, modifying one part may affect the entire OS.
2. Less Security:
   All services run in kernel mode, so any bug or malicious driver can crash the entire system.

Examples

- MS-DOS
- Early UNIX

- Linux (partially monolithic but modular)

2. Layered Structure of Operating System

The layered structure divides the operating system into different layers. Each layer is built on top of the lower layer and provides services to the layer above it.

Concept

- The lowest layer interacts with hardware.
- The highest layer interacts with the user (UI).
- Intermediate layers perform specific tasks like memory, I/O, and device management.

Typical Layer Model

1. Layer 0 – Hardware
2. Layer 1 – CPU Scheduling & Process Management
3. Layer 2 – Memory Management
4. Layer 3 – I/O Management
5. Layer 4 – File System
6. Layer 5 – User Interface & Applications

Characteristics

- Communication is restricted
- Modular design allows systematic implementation.

Advantages

1. Modularity:
   Each layer is independent, making the OS well-organized.
2. Easy Debugging:
   If an error occurs, it can be traced to a specific layer.
3. Easy to Update or Modify:
   Changing one layer does not affect others.
4. Improved Security and Simplicity:
   Layers provide protection boundaries.

Disadvantages

1. Performance Overhead:
   Requests must pass through multiple layers → slower response.
2. Complex to Design Layer Boundaries:
   Hard to decide what each layer should contain.
3. Rigid Structure:
   Lower layers cannot use services of upper layers.

Examples

- Early UNIX versions

3. Microkernel Structure of Operating System

A microkernel is a minimal OS kernel that contains only the most essential functionalities. Other services run in user space rather than kernel space.

Microkernel Contains Only:

- Low-level Memory Management
- CPU Scheduling
- Inter-process Communication (IPC)
- Basic Device Control

Other Services (in User Space):

- File Systems
- Device Drivers
- Networking Services
- System Libraries
- GUI components

These services communicate with the microkernel using message passing.

Characteristics

- Very small and secure kernel.
- Each OS component behaves like a separate server.
- Crashes in user-level servers do not crash the entire OS.

Advantages

1. High Security:
   Most services run in user mode → less chance of kernel failure.
2. Better Stability and Reliability:
   If a driver crashes, the OS continues running.
3. Modularity:
   Services can be added, removed, or updated easily.
4. Portability:
   Microkernel can be easily adapted to different hardware.

Disadvantages

1. Performance Overhead:
   Heavy message-passing between kernel and user-level services.
2. Slow Service Execution:
   More context switches → slower performance.
3. Complex Architecture:
   Designing IPC efficiently is difficult.

Examples

- MINIX
- QNX

Operating-System Design Issues:

Designing an operating system is a complex task with many challenges. The OS must satisfy both **user goals** and **system (developer) goals**, while working correctly on the chosen hardware and system type. The major design issues are:

1. Defining Goals and Requirements

- First and most important step in OS design.
- Goals depend on type of system: desktop, mobile, real-time, distributed.
- Requirements are divided into two types:
  • **User goals** – easy to use, reliable, safe, fast
  • **System goals** – easy to design, maintain, flexible, efficient
- These goals are often **vague** and hard to convert into exact rules.

2. Hardware Dependency

- OS design depends heavily on hardware architecture:
  CPU, memory, storage, I/O devices.
- The OS must manage each of these efficiently.
- Supports hardware mechanisms like interrupts, timers, modes (kernel/user).

3. Resource Allocation

- OS must decide **how to distribute CPU, memory, and I/O** among processes.
- Policies affect performance and efficiency.
- Includes scheduling, deadlock handling, memory allocation, and storage management.

4. Structure and Modularity

- Because OS is large and complex, it must be built **piece by piece**.
- Each module requires clear **inputs, outputs, and functions**.
- Design choices include:
  - Monolithic kernel
  - Layered design
  - Microkernel
  - Modular kernel
  - Hybrid kernel

5. Efficiency vs. Functionality

- Designers must balance:
  - **Speed and performance**
  - **Security**
  - **Reliability**
  - **Convenience**
- Adding more features may reduce speed; optimizing for speed may reduce usability.

6. Security and Protection

- Prevent unauthorized access to files, memory, and system resources.
- OS must support authentication, access control, isolation of processes.

- Critical in cloud, mobile, and distributed environments.

7. Reliability and Fault Tolerance

- OS must detect errors and handle failures safely.
- Must ensure stable operation even when part of system fails.
- Important for **real-time and mission-critical systems**.

8. Flexibility and Portability

- OS should support changing hardware and new devices.
- Should be portable across different architectures.
- Example: Linux runs on mobiles, servers, supercomputers.

9. Scalability

- OS must work efficiently as number of users/processes increases.
- Important for servers, cloud systems, enterprise systems.

Types of Operating Systems:

1. Batch Operating System

- Jobs are grouped and executed in batches.
- No user interaction during execution.
- Used for long, repetitive tasks.
  Example: Payroll systems.

2. Time-Sharing Operating System

- CPU time is shared among multiple users.
- Each user gets a short time slice.
- Gives fast response time.
  Example: UNIX.

3. Distributed Operating System

- Multiple computers work together as a single system.
- Resources are shared through network.
- High reliability and fault-tolerance.
  Example: Amoeba, LOCUS.

4. Real-Time Operating System (RTOS)

- Gives fast and guaranteed response.
- Used where timing is critical.

Types:

1. **Hard RTOS** – Strict deadlines.
2. **Soft RTOS** – Deadlines flexible.

Examples: QNX, VxWorks.

5. Mobile Operating System

- Designed specially for smartphones and tablets.
- Touch support, battery optimization, app environment.
  Examples: Android, iOS, Windows Mobile.

6. Network Operating System (NOS)

- Manages networking functions: file sharing, printer sharing, communication.
- Manages connected computers in LAN/WAN.
  Examples: Linux Server OS, Novell NetWare, Windows Server.

7. Multi-Programming / Multi-Tasking OS

- Allows multiple programs to run simultaneously.
- Improves CPU utilization.
  Examples: Modern Windows, Linux.

8. Multi-Processing OS

- Supports multiple CPUs.
- Work distributed between processors.
  Example: Symmetric Multiprocessing (SMP Linux).

9. Database Operating System (DBOS)

A Database Operating System is an operating system specially designed to support database management, transaction processing, and concurrency control.

Process Scheduling in Operating Systems:

Process Scheduling is a fundamental function of an Operating System (OS) responsible for **deciding which process runs on the CPU at a given time**. Since CPU time is limited and

many processes compete for it, the OS uses scheduling techniques to ensure efficient and fair CPU utilization.

Why Process Scheduling?

Process scheduling is needed to:

- Maximize **CPU utilization**
- Increase **throughput** (number of processes completed)
- Reduce **waiting time** of processes
- Minimize **response time** for interactive systems
- Ensure **fairness** among processes

Types of Scheduling Queues

1. Job Queue

Stores all processes in the system.

2. Ready Queue

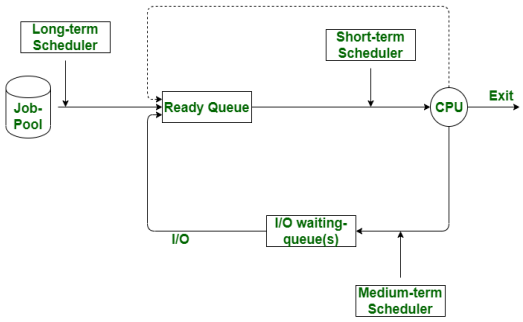
Processes that are loaded into RAM and waiting for CPU.

3. Device Queue

Processes waiting for I/O operations.

Types of Schedulers

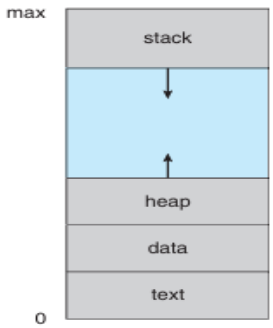
1. **Long-Term Scheduler (Job Scheduler)**
    - Decides **which jobs enter the ready queue**
    - Controls the degree of multiprogramming
    - Runs **less frequently**
2. **Short-Term Scheduler (CPU Scheduler)**
    - Selects **which ready process executes next**
    - Runs **very frequently** (milliseconds)
3. **Medium-Term Scheduler (Swapper)**
    - Performs **swapping**: removes processes from memory and brings them back
    - Helps in managing memory effectively



Process Concepts:

A **process** is a *program in execution*. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently. It is more than just code—once the program starts running, it becomes a dynamic entity with several components. Each process is represented in the operating system by a **process control block (PCB)**—also called a task control block. It contains many pieces of information associated with a specific process.



ure 3.1 Layout of a process in memory.

✔ A process contains:

1. **Program Code (Text Section)**
2. **Program Counter (PC)** – indicates the next instruction
3. **CPU Registers** – stores temporary values
4. **Stack** – function calls, local variables
5. **Data Section** – global variables
6. **Heap** – dynamically allocated memory
7. **Process Control Block (PCB)** – stores process information (state, PID, registers, etc.)

Process States:

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states**.**

Process States in Operating System (Clean Theory)

1. New

The process is being created. All required resources are allocated, and the OS adds the process to the process table.

2. Running

The process is currently being executed by the CPU. Instructions are fetched and executed in this state.

3. Waiting (Blocked)

The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

4. Ready

The process is loaded in main memory and ready to run, but it is waiting for the CPU to be assigned. Multiple processes may remain in the ready queue.

5. Terminated (Exit)

The process has finished execution. The OS deallocates resources and removes the process from the process table.

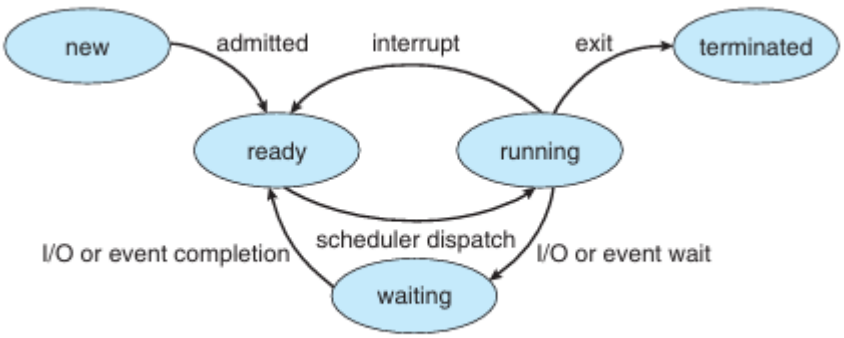


Figure 3.2 Diagram of process state.

Operation on processes:

Process Creation:

- A new process is created by a parent process using system calls like fork().
- The OS allocates memory, creates a PCB, and initializes resources.
- The process is placed in the ready queue.

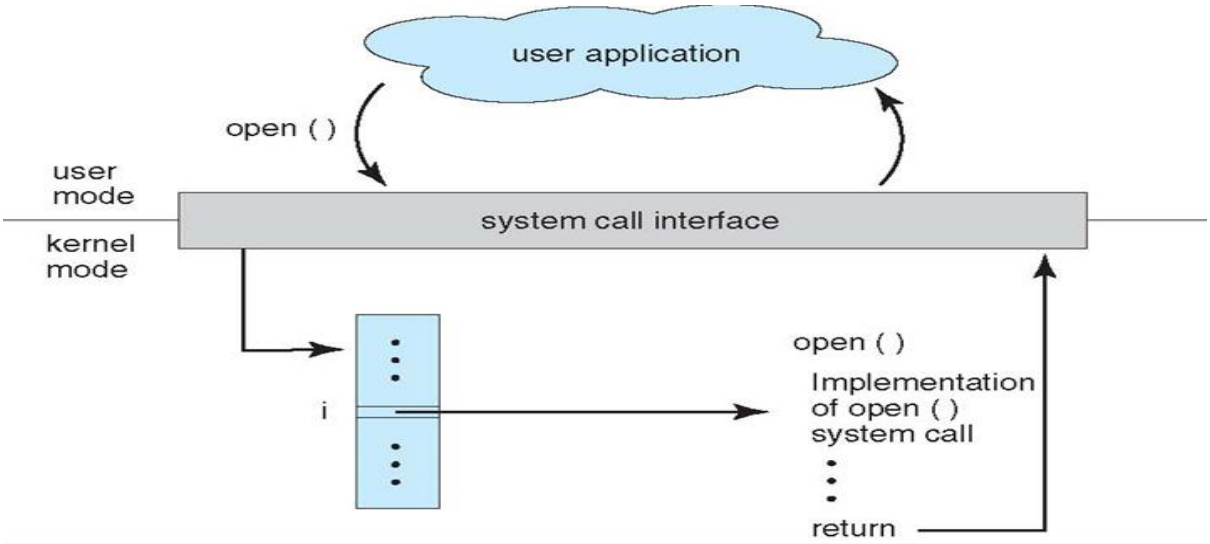
- Execution starts when the CPU is assigned.

Process Termination:

- A process terminates when it finishes execution or encounters an error.
- The OS releases all allocated resources and removes the PCB.
- The parent is informed using wait().
- The process enters the terminated (exit) state

System calls:

User programs cannot directly access hardware or critical OS resources because it would make the system unstable and insecure. To maintain safety, the operating system provides system calls — controlled interfaces that allow user programs to request services from the kernel. A system call is typically initiated by an **application program**, which uses a library function to invoke the system call. The library function translates the application request into a system call and transfers control to the OS kernel. The OS kernel then performs the requested service on behalf of the application and returns the results to the application program.

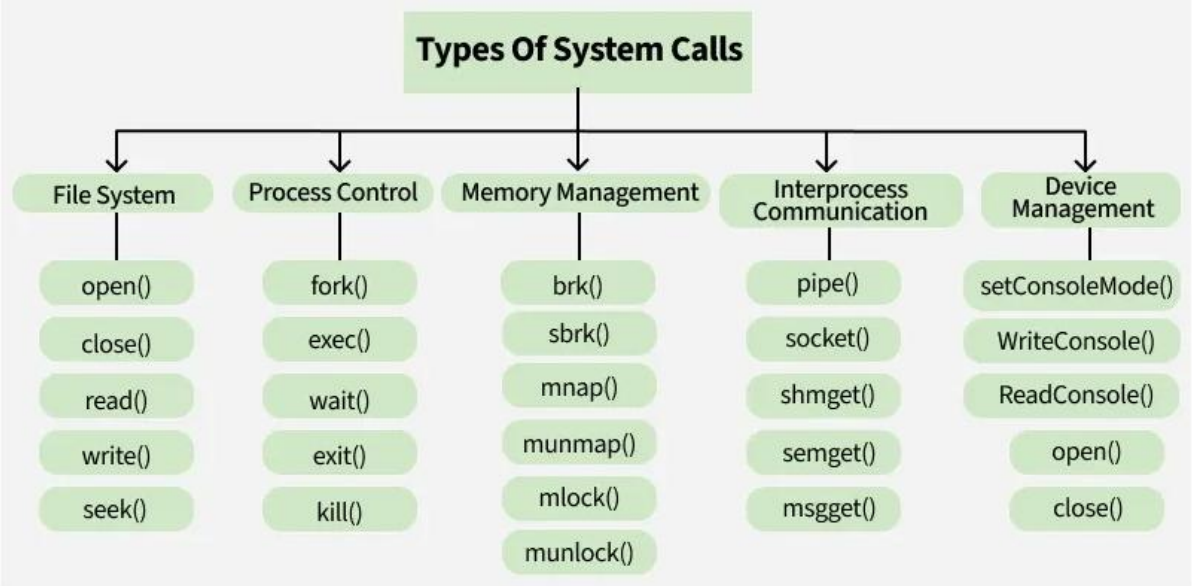


How the system call will be performed

Types of system calls:

1. **Process management:** Creating, deleting, and managing processes.
2. **File management:** Creating, opening, reading, writing, and deleting files.
3. **Memory management:** Allocating and freeing memory.
4. **Device management:** Managing input/output devices such as printers, scanners, and network adapters.

5. **Network management:** Configuring network settings and managing network connections.
6. **Inter-process communication:** Allowing processes to communicate with each other and synchronize their activities



Interprocess Communication (IPC):

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it does not share data with any other processes executing in the system. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Reasons for Process Cooperation

• Information Sharing

Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.

• Computation Speedup

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such speedup is possible only if the computer has multiple processing cores.

• Modularity

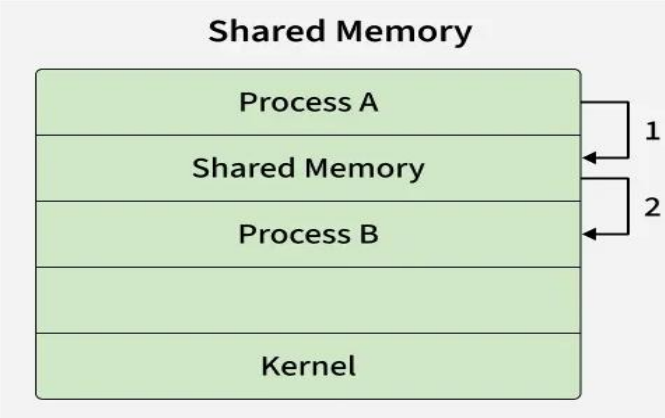
We may want to construct the system in a modular fashion, dividing system functions into separate processes or threads.

Cooperating processes require an **interprocess communication (IPC)** mechanism that allows them to exchange data — that is, to **send and receive information**. There are **two fundamental models** of interprocess communication:

1.Shared Memory:

Communication between processes using shared memory requires processes to share some variable and it completely depends on how the programmer will implement it. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

- In shared memory model, a common memory space is allocated by the kernel.
- Process A writes data into the shared memory region (Step 1).
- Process B can then directly read this data from the same shared memory region (Step 2).
- Since both processes access the same memory segment, this method is fast but requires synchronization mechanisms (like semaphores) to avoid conflicts when multiple processes read/write simultaneously.

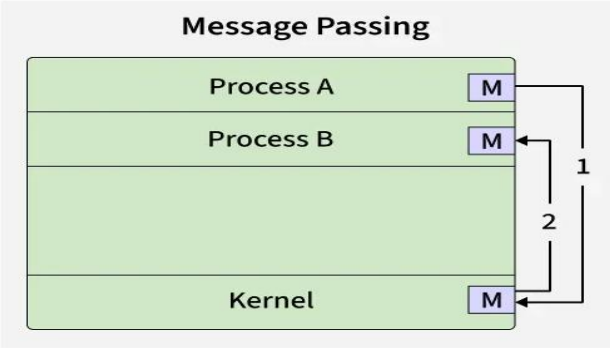


2. Message Passing:

Message Passing is a method where processes communicate by sending and receiving messages to exchange data.

- In this method, one process sends a message and the other process receives it, allowing them to share information.
- Message Passing can be achieved through different methods like **Sockets**, **Message Queues or Pipes.**

- In the above message passing model, processes exchange information by sending and receiving messages through the kernel.
- Process A sends a message to the kernel (Step 1).
- The kernel then delivers the message to Process B (Step 2).
- Here, processes do not share memory directly. Instead, communication happens via system calls (send(), recv(), or similar).
- This method is simpler and safer than shared memory because there's no risk of overwriting shared data, but it incurs more overhead due to kernel involvement.



Message Passing

Pipes

A **Pipe** is a communication mechanism used in operating systems that enables data transfer between **two related processes**, usually a parent and child.
It behaves like a **unidirectional data channel** where the output of one process becomes the input of another.

Features of Pipes

1. **Unidirectional** → data flows from **write-end** to **read-end**.
2. Used only between **related processes** created using fork().
3. Works like a temporary buffer in kernel memory.
4. Implemented using the **pipe() system call** in UNIX/Linux.
5. Pipe exists only until both processes close it or terminate.

How Pipes Work

- When a pipe is created, the OS provides **two file descriptors**:
    - fd[0] → Read end
    - fd[1] → Write end

- One process writes data into fd[1].
- The second process reads data from fd[0].
- No direct access is allowed; only sequential read/write.

Named Pipes (FIFOs)

A **Named Pipe**, also known as a **FIFO (First-In-First-Out)**, is similar to a pipe but has a **name stored in the file system**, which allows **unrelated processes** to communicate.

Features of Named Pipes

1. Can be used by **any processes**, even if not related.
2. Provides **bidirectional** or **unidirectional** communication.
3. Created using:

   mkfifo filename
4. Operates on FIFO principle → data read in the same order written.

Difference Between Pipes and FIFOs

Feature	**Pipe**	**Named Pipe (FIFO)**
Relation	Only related processes	Any processes
Storage	Temporary	Permanent (on file system)
Direction	Unidirectional	Uni/Bidirectional
Creation	pipe() call	mkfifo or mknod

Advantages:

Pipes

- Simple to use
- Fast communication
- No file system overhead

FIFOs

- Communication between unrelated programs
- Useful for client–server applications

- Persistent communication channel

Disadvantages

Pipes

- Only for related processes
- Unidirectional
- Not persistent

FIFOs

- Slower than pipes
- Must be explicitly created
- Less secure if permissions are not set

Process Synchronization:

Process Synchronization is a mechanism in operating systems used to manage the execution of multiple processes that access shared resources. Its main purpose is to ensure data consistency, prevent race conditions and avoid deadlocks in a multi-process environment.

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

Process Synchronization is the coordination of multiple cooperating processes in a system to ensure controlled access to shared resources, thereby preventing race conditions and other synchronization problems.

Improper Synchronization in Inter Process Communication Environment leads to following problems:

Inconsistency: When two or more processes access shared data at the same time without proper synchronization. This can lead to conflicting changes, where one process's update is overwritten by another, causing the data to become unreliable and incorrect.

Loss of Data: Loss of data occurs when multiple processes try to write or modify the same shared resource without coordination. If one process overwrites the data before another process finishes, important information can be lost, leading to incomplete or corrupted data.

Deadlock: Lack of proper Synchronization leads to Deadlock which means that two or more processes get stuck, each waiting for the other to release a resource. Because none of the processes can continue, the system becomes unresponsive and none of the processes can complete their tasks.

Role of Synchronization in IPC:

Preventing Race Conditions: Ensures processes don't access shared data at the same time, avoiding inconsistent results.

Mutual Exclusion: Allows only one process in the critical section at a time.

Process Coordination: Lets processes wait for specific conditions (e.g., producer-consumer).

Deadlock Prevention: Avoids circular waits and indefinite blocking by using proper resource handling.

Safe Communication: Ensures data/messages between processes are sent, received and processed in order.

Fairness: Prevents starvation by giving all processes fair access to resources.

The Critical-Section Problem :

Consider a system consisting of **n processes** $\{P_0, P_1, \ldots, P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be **accessing and updating data** that is **shared** with at least one other process.

The important feature of the system is that, **when one process is executing in its critical section, no other process is allowed to execute in its critical section**. That is, **no two processes** are executing in their critical sections at the same time.

The **critical-section problem** is to design a **protocol** that the processes can use to **synchronize their activity** so as to **cooperatively share data**.

Each process must **request permission** to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

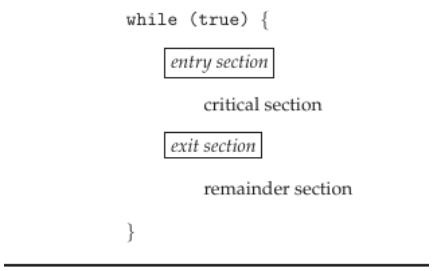


Figure 6.1 General structure of a typical process.

A solution to the critical-section problem must satisfy the following **three requirements**:

1. Mutual Exclusion

If process Pi is executing in its critical section, then **no other processes** can be executing in their critical sections.

2. Progress

If **no process** is executing in its critical section and **some processes wish to enter** their critical sections, then **only those processes** that are not executing in their remainder sections can participate in deciding **which will enter its critical section next**, and this selection **cannot be postponed indefinitely**.

3. Bounded Waiting

There exists a **bound**, or limit, on the number of times that other processes are allowed to enter their critical sections **after a process has made a request** to enter its critical section and **before that request is granted**.

Race Conditions :

A **race condition** occurs when **multiple processes or threads** access and manipulate **shared data concurrently**, and the **final result depends on the particular order** in which the operations are executed. Because the execution order is not predictable in a concurrent system, **uncontrolled access** to shared data may lead to **inconsistent or incorrect** data values.

In a system, several processes may be running at once, and many of them may access **shared variables**, **shared memory**, files, buffers, or data structures maintained by the operating system. If such access is **not controlled**, the actions of one process may **interfere** with the actions of another.

A race condition thus exists when the correctness of the computation **depends on the sequence or timing** of concurrent execution. The operating system and the user must take steps to ensure that **shared data are accessed in a controlled manner**, typically through **synchronization mechanisms**, so that these timing-based errors do not occur.

Race conditions are a major concern in **multi-threaded**, **multi-core**, and **multi-process** systems, where the overlap of operations may cause **corrupted values**, **data inconsistency**, or **unpredictable behaviour** if proper synchronization is not used.

Here is a **race condition example** written in **textbook-style (same words style)** — clear and suitable for notes.

Example of a Race Condition:

Consider a shared integer variable:

x = 10

Two processes, **P1** and **P2**, both wish to update this variable.

Process P1

x = x + 1

Process P2

x = x * 2

If these processes execute **concurrently** without synchronization, the **final value of x depends on the order** in which their individual instructions are carried out.

Possible Execution Order 1

P1 executes first, then P2:

1. P1 reads x = 10
2. P1 computes x = 10 + 1 = 11
3. P1 writes x = 11
4. P2 reads x = 11
5. P2 computes x = 11 * 2 = 22
6. P2 writes x = 22

Final value of x = 22

Possible Execution Order 2

P2 executes first, then P1:

1. P2 reads x = 10
2. P2 computes x = 10 * 2 = 20
3. P2 writes x = 20
4. P1 reads x = 20
5. P1 computes x = 20 + 1 = 21
6. P1 writes x = 21

Final value of x = 21

Conclusion: Because P1 and P2 execute their operations at the same time, the order in which they read and update the shared variable x is not fixed. Different orders of execution produce

different final results (22 in one case and 21 in another).This means the final value of x is not predictable, as it depends entirely on the timing and the interleaving of the two processes' instructions. This unpredictable behaviour clearly shows a race condition, because the correctness of the result depends on which process reaches the shared data first.

Threats:

In the context of process synchronization, a threat refers to anything that can harm the correctness of shared data when multiple processes access it concurrently.

Main Threats to Shared Data

Race Conditions

When two or more processes access and modify shared data at the same time, the final result becomes unpredictable.

Data Inconsistency

Incorrect values arise when processes interleave their operations improperly.

Deadlock

Processes wait forever for each other, causing the system to halt.

Starvation

A process never gets a chance to enter the critical section.

These threats make synchronization mechanisms essential.