OPERATING SYSTEMS

UNIT-II

PROCESS

A process is a program that is currently running. For example, when we write a program in C or C++ and compile it, the compiler creates a binary file. Both the original code and the binary file are just programs. But when we run the binary file, it becomes a process.

A process is "active" because it is running, while a program is "passive" because it just exists as a file. One program can create multiple processes if it is run multiple times. For example, if we open a .exe or binary file several times, each instance creates a new process.

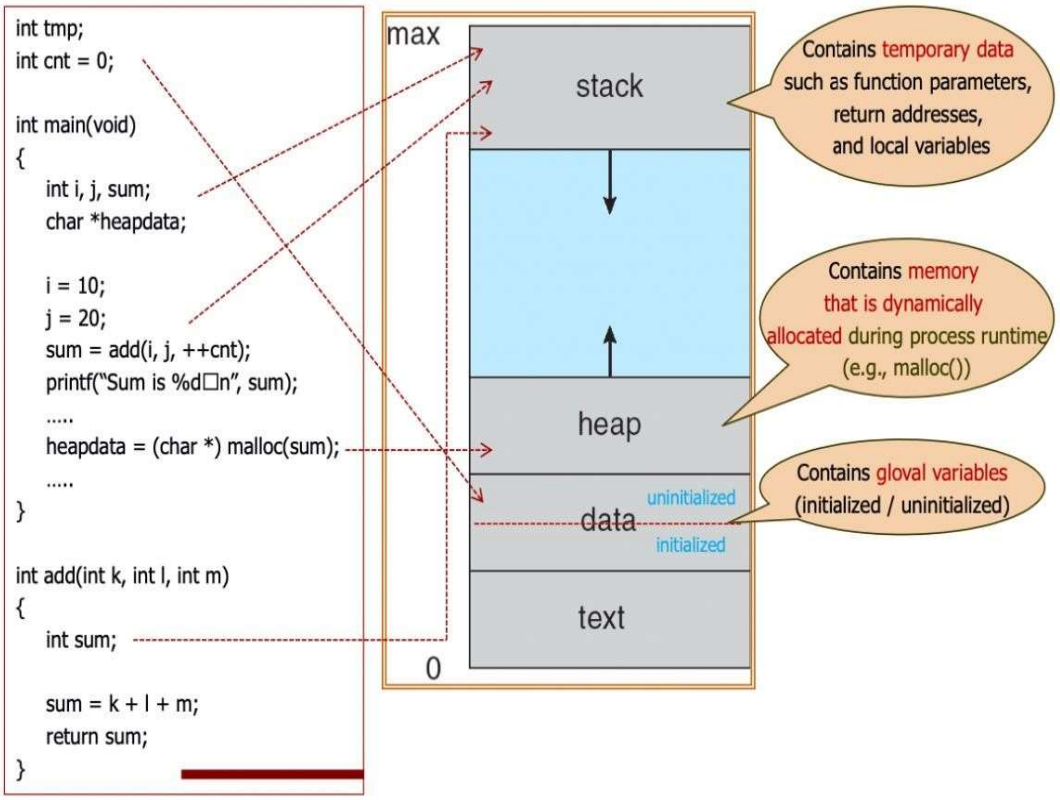
PROCESS VS PROGRAM

- A process runs instructions in machine code, while a program contains instructions written in a programming language.
- A process is dynamic (changes as it runs), while a program is static (remains the same as a file).
- A process stays in the main memory (RAM) while running, but a program is stored in secondary memory (like a hard drive).
- A process exists only while it is running, but a program stays in memory permanently unless deleted.
- A process is active (currently executing), while a program is passive (just stored as a file).

Process memory is divided into four parts to work efficiently:

- **Text Section:** Contains the compiled program code, which is loaded from storage when the program starts.
- **Data Section:** Stores global and static variables that are allocated and initialized before the main function runs.
- **Heap:** Used for dynamic memory allocation, managed using functions like new, delete, malloc, and free.
- **Stack:** Holds local variables. Space is allocated on the stack when a local variable is declared.

OPERATING SYSTEMS

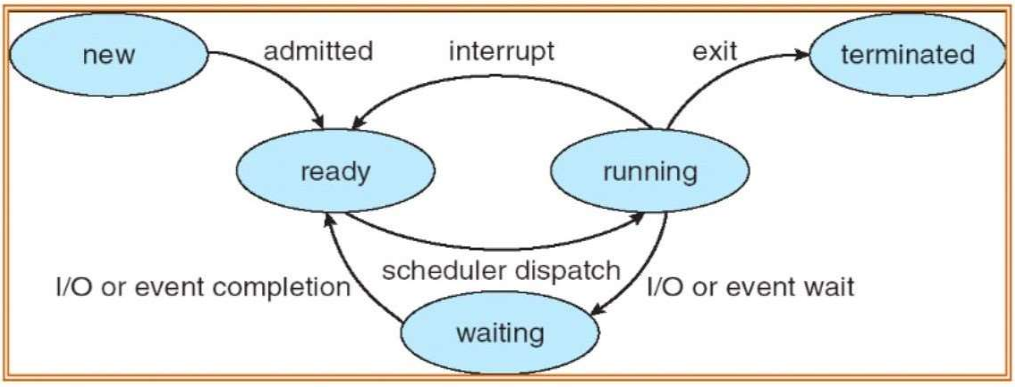


PROCESS STATES

Process states in an operating system help manage resources efficiently by tracking each process's current condition. The main process states are:

- **New State:** When a process is first created or initialized, it is in the new state. It is being prepared to move to the ready state.
- **Ready State:** The process is ready to run but is waiting for the CPU to be assigned. It stays in this state until the CPU becomes available.
- **Running State:** When the CPU is assigned to a process, it enters the running state, where it executes instructions and uses system resources. Only one process can be running at a time, and the operating system decides which process runs next.
- **Waiting/Blocked State:** If a process is waiting for something, like user input or data from a disk, it enters the blocked state. It remains here until the required event happens.
- **Terminated State:** When a process finishes or is stopped by the operating system, it moves to the terminated state. At this point, it no longer uses system resources, and its memory is freed.

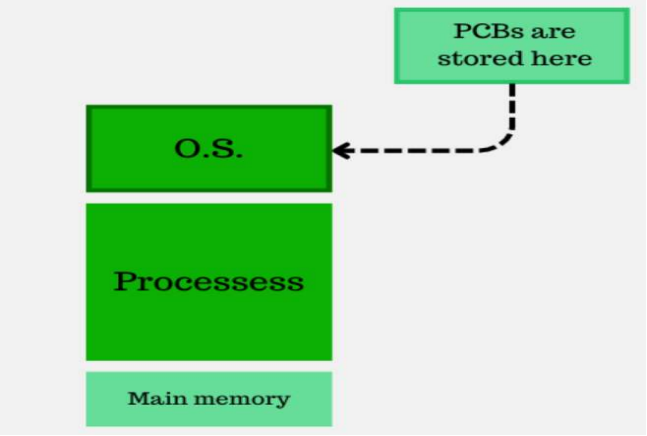
Now let us see the state diagram of these process states −



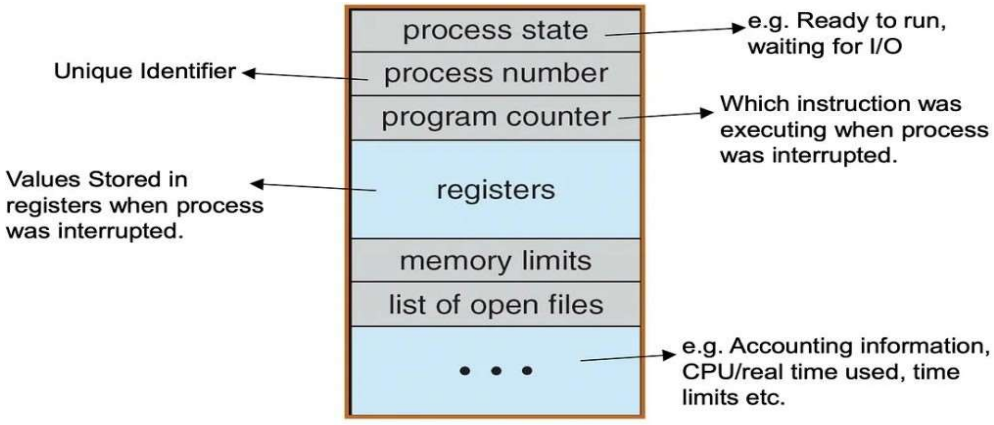
- When a new process is created, it enters the **ready state**.
- If no other process is running, the scheduler dispatcher moves it to the **running state**.
- If a higher-priority process becomes ready, the current process is moved to the **waiting state**.
- Once an I/O operation or event is completed, the process returns to the **ready state** based on an interrupt signal.
- When the process finishes execution, it moves to the **terminated state**, marking its completion.

PROCESS CONTROL BLOCK

The Process Control Block (PCB) is a data structure that stores information about a process. It is also called a task control block or an entry in the process table. Each time a new process is created, a unique PCB is generated, and it is removed when the process finishes. The PCB helps the operating system manage multiple tasks efficiently.



PCB Contains the following attributes



- **Process State:** A process goes through different phases during its execution. Its current phase is called the process state.
- **Program Counter:** This holds the address of the next instruction to be executed. It starts with the address of the first instruction and updates automatically after each step until the program completes.
- **Registers:** These vary based on the computer architecture and include accumulators, index registers, stack pointers, and general-purpose registers. When an interrupt occurs, the system saves these registers, along with the program counter, to ensure the process resumes correctly.
- **List of Open Files:** Stores details of files used by the process. This helps the operating system close all open files when the process terminates.

CONTEXT SWITCHING

Context switching is the process of switching the CPU from one task to another. During this switch, the operating system saves the state of the current process so it can resume later and loads the state of the next process. This allows a single CPU to manage multiple processes efficiently without needing extra processors.

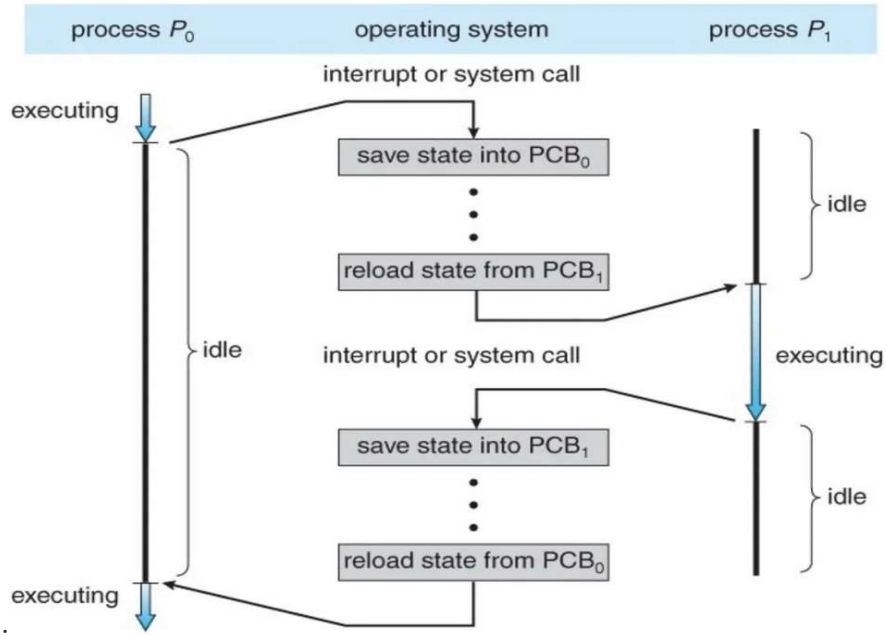
Why Context Switching is Needed:

Priority-Based Switching: If a higher-priority process enters the ready queue, the currently running process is paused so the high-priority task can execute first.

I/O Handling: If a process needs I/O resources, it is switched out so another process can use the CPU. Once the I/O operation is complete, the original process moves back to the ready state and resumes execution.

Handling Interrupts: If an interrupt occurs during execution, the system saves the process state using context switching. Once the interrupt is handled, the process moves from the waiting state back to the ready state and resumes execution from where it left off.

OPERATING SYSTEMS



Steps in Context Switching

1. The CPU executes **Process 0**.
2. A **triggering event** occurs, such as an interrupt or system call.
3. The system **pauses Process 0** and saves its state in **PCB 0** (Process Control Block).
4. The system selects **Process 1** from the queue and loads its state from **PCB 1**.
5. The CPU executes **Process 1**, resuming from where it left off (if it had been running before).
6. When another **triggering event** occurs, the system **pauses Process 1** and saves its state in **PCB 1**.
7. The system **reloads Process 0**, allowing it to continue execution. **Process 1 remains idle** until it is scheduled again.

Advantages of Context Switching

- Enables **multitasking**, creating an illusion that multiple processes run simultaneously.
- Since context switching happens **very quickly**, users feel that multiple tasks are executing at the same time.

Disadvantages of Context Switching

- Takes **extra time** to save one process's state and load another, known as **context switching time**.
- During switching, the CPU does **no useful work**, making it an **overhead** from the user's perspective.

OPERATING SYSTEMS

PROCESS OPERATIONS IN AN OPERATING SYSTEM

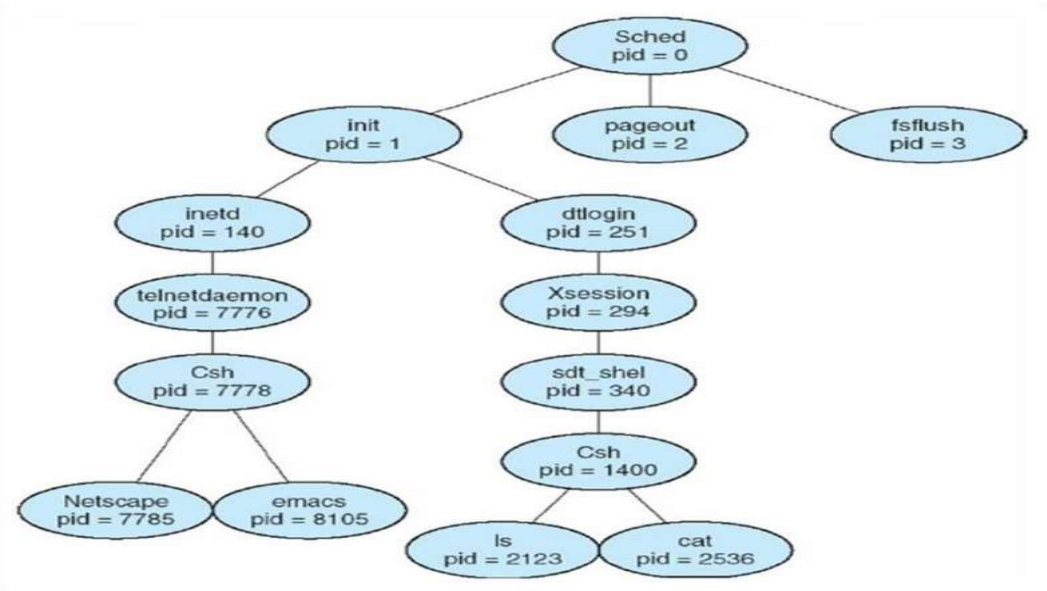
The operating system manages processes through two main operations:

- Process Creation
- Process Termination

Process Creation

Process creation is a key operation where a new process is started. The process that creates another process is called the parent, and the newly created process is called the child. A child process can also create its own child processes, forming a process hierarchy.

Each process is assigned a unique Process Identifier (PID). A process creates a child process using a system call.



In the **Solaris operating system**, the **process tree** represents processes, including background processes (**daemons**), along with their unique **Process Identifiers (PIDs)**.

In **Linux and other Unix-like systems**, a **daemon** is a background process that runs independently of any user session. These processes typically start when the system boots and continue running until shutdown. **Daemons** handle various system-level tasks such as:

- Managing hardware devices
- Handling network requests
- Scheduling jobs
- Running background services without user interaction

Daemon	Operation
init	The **init** process is the first process started by the Linux/Unix kernel and holds the process ID (PID) 1.
pageout	The **pageout** daemon is responsible for managing virtual memory,
fsflush	The **fsflush** process is a background daemon responsible for synchronizing cached data to disk.
inetd	**inetd**, short for "internet super-server", is a daemon that listens on designated ports for incoming connections and then launches the appropriate program. inetd is responsible for networking services such as telnet and ftp.
dtlogin	**Desktop login- dtlogin** is the display manager component responsible for managing user logins to the graphical environment.

In **Solaris**, the **sched** process (PID 0) is at the top of the process tree. It creates several child processes, including **pageout** and **fsflush**. It also creates the **init** process, which acts as the root parent for all user processes.

From **init**, two key child processes are created:

- **inetd** (handles network connections)
- **dtlogin** (manages graphical logins)

When a user logs in, **dtlogin** starts an **Xsession**, which then creates **sdt_shel**. This tool, part of the **Common Desktop Environment (CDE)**, provides a graphical interface for accessing system functions. Below **sdt_shel**, a **C-shell (csh)** is created, allowing users to execute commands.

For example, in the command-line shell, users can run commands like **ls** and **cat**, which are created as child processes.

Another **csh process (PID 7778)** represents a user connected via **telnet**. This user has started:

- **Netscape browser (PID 7785)**
- **Emacs editor (PID 8105)**, a powerful tool for text editing, coding, file management, email reading, web browsing, and even playing games.

OPERATING SYSTEMS

Resource Sharing Between Processes

A process requires resources such as **CPU time, memory, files, and I/O devices** to complete its tasks. When a process creates a **subprocess (child process)**, the child may:

1. **Obtain resources directly from the operating system**, or
2. **Use a portion of the parent's resources**, depending on system constraints.

Resource Allocation

- The **parent process** may either **divide its resources** among its child processes or **share certain resources** (like memory and files) among them.
- Restricting a child to a subset of the parent's resources prevents the system from **overloading** due to excessive subprocess creation.

Passing Data Between Processes

When a process is created, **initialization data (input)** can be passed from the parent to the child.

For example:

- A process that **displays an image file** (e.g., img.jpg) on a screen may receive the **file name** as input from its parent.
- The child process then **opens the file, reads its contents, and displays them on the screen**.
- Some operating systems **pass open files** to child processes, allowing them to **directly transfer data** between input (file) and output (screen) without reopening resources.

Process Execution in Operating Systems

When a process creates a new process, there are two possible execution scenarios:

1. **Concurrent Execution:** The parent continues executing alongside its child process.
2. **Sequential Execution:** The parent waits until some or all of its child processes have completed.

Process Address Space

A newly created process can have:

- **A duplicate of the parent process**, including its program and data (e.g., fork() system call).
- **A new program loaded into it**, replacing the parent's program (e.g., exec() system call).

OPERATING SYSTEMS

Process Termination

A process terminates when it completes execution and uses the exit() system call to notify the OS for deletion. At termination:

- The process may return a **status value** to the parent process via wait().
- The OS **deallocates resources** such as memory, open files, and I/O buffers.

Other Termination Scenarios

- A process can terminate another process via a system call (e.g., TerminateProcess() in Win32).
- Typically, only a **parent process** can terminate its child process.
- Reasons for termination:
  - The child **exceeds resource limits** set by the parent.
  - The child's assigned task is **no longer required**.
  - **Cascading Termination:** If a parent process exits, its children may also be terminated if the OS enforces it.

SYSTEM CALLS FOR PROCESS CREATION & TERMINATION IN LINUX

1. `fork()` System Call (Process Creation)

- Used to create a **new process (child process)** from an existing process (parent process).
- The OS duplicates the **parent's memory, file descriptors, and other attributes** for the child.
- **Return values:**
  - Parent process receives **child's PID**.
  - Child process receives **0**.
  - If unsuccessful, **-1** is returned, and no child process is created.
- Used for **process spawning**, enabling multiple child processes to run **concurrently**.

2. `exec()` System Call (Program Execution)

- Replaces the **current process** with a **new program**.
- Steps performed by `exec()`:

1. Clears the current process's memory.
2. Loads the new program into memory.
3. Sets the new program's entry point.
4. Starts execution of the new program.

- Often used after `fork()` to make the **child process execute a different program** than the parent.

3. `wait()` System Call (Parent Waiting for Child)

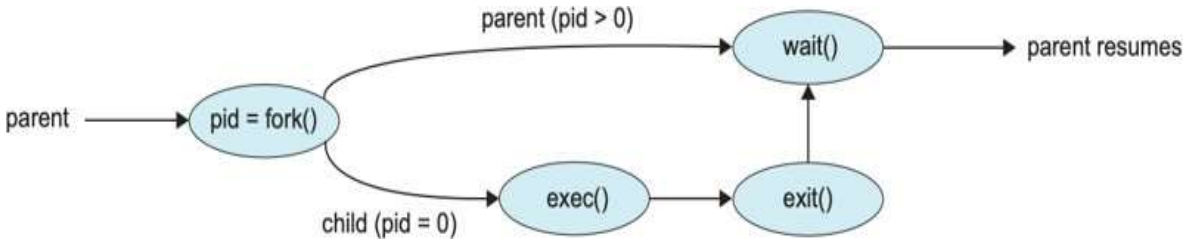
- Suspends the **parent process** until the child process finishes execution.
- If `wait()` is called:
  - The **parent halts** until the child **completes**.
  - Returns the **PID of the terminated child process** or **-1 on failure**.
  - Takes a parameter to store **child's exit status**, or NULL if not needed.

4. `exit()` System Call (Process Termination)

- Used to **terminate** a process and **release system resources**.
- After execution, the OS **retrieves allocated resources** and removes the process from the system.

Together, `fork()`, `exec()`, `wait()`, and `exit()` facilitate **process creation, execution, and termination** in Linux, enabling **efficient multitasking and resource management**.

Let's depict all the system calls in the form of a process transition diagram.



Process Lifecycle in Operating Systems

A process begins its execution when it is created and transitions through multiple states before termination.

1. Process Creation

- A process is created using the **`fork()` system call**, which duplicates an existing process (parent) to create a new process (child).
- **System calls** act as an interface between the operating system and processes, allowing user programs to request services.

2. Program Execution in a New Process

- Often, a child process needs to execute a different program than its parent.
- The **`exec()` system call** replaces the process's address space with a new program, effectively loading and running a different executable.

3. Process Termination

- A process **exits** using the **`exit()` system call**, which releases all its allocated resources except for its **Process Control Block (PCB)**.

4. Parent Monitoring Child Process

- A parent process can check the status of a **terminated child process** using the **`wait()` system call**.
- When `wait()` is used, the parent process remains **blocked** until the child process it is waiting for completes execution.

These system calls—`fork()`, `exec()`, `exit()`, and `wait()`—facilitate **process creation, execution, and termination**, ensuring efficient process management in operating systems.

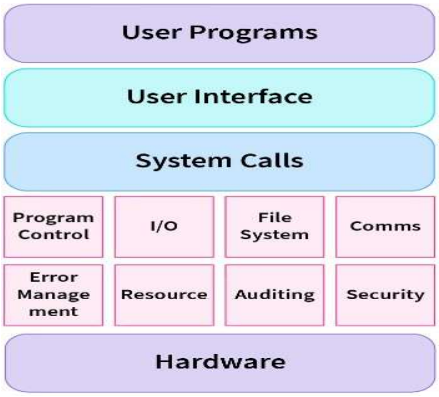
.

UNIX SYSTEM CALLS	DESCRIPTION	WINDOWS API CALLS	DESCRIPTION
Process Control			
`fork()`	Create a new process.	`CreateProcess()`	Create a new process.
`exit()`	Terminate the current process.	`ExitProcess()`	Terminate the current process.
`wait()`	Make a process wait until its child processes terminate.	`WaitForSingleObject()`	Wait for a process or thread to terminate.
`exec()`	Execute a new program in a process.	`CreateProcess()` or `ShellExecute()`	Execute a new program in a new process.
`getpid()`	Get the unique process ID.	`GetCurrentProcessId()`	Get the unique process ID.

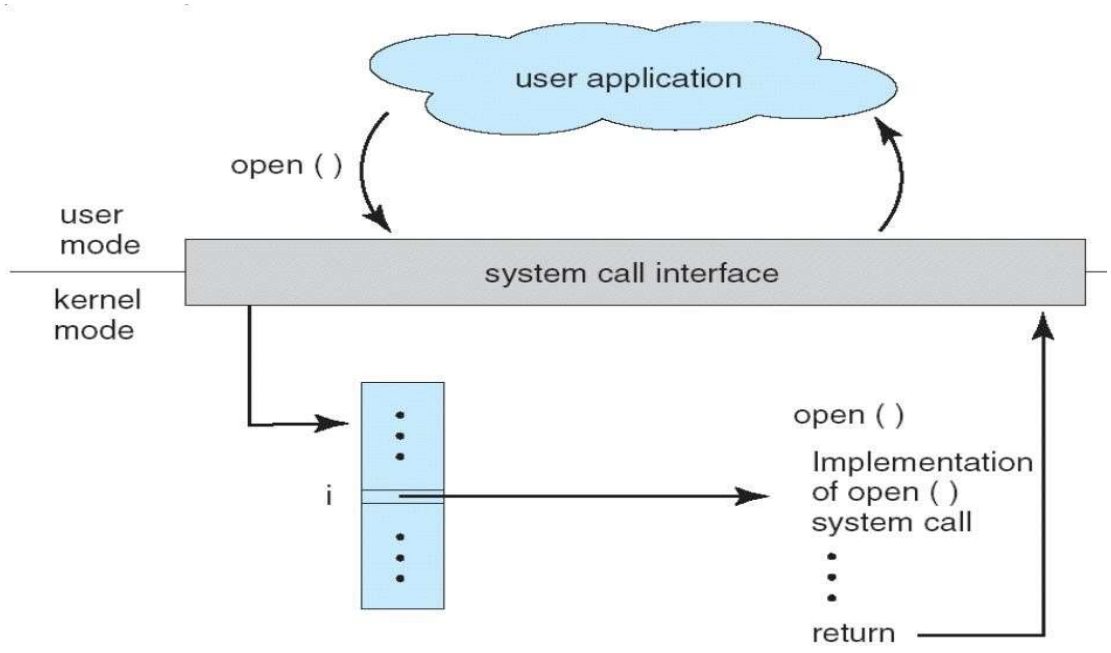
OPERATING SYSTEMS

SYSTEM CALL INTERFACE FOR PROCESS MANAGEMENT

A system call is a function that allows a user program to request services from the operating system that cannot be performed by regular functions. It acts as a bridge between the process and the operating system, enabling essential operations such as process creation, execution, and termination.



Example:



System calls are categorized into five main types:

- **Process Control**
- **File Management**
- **Device Management**
- **Information Maintenance**
- **Communication**

OPERATING SYSTEMS

Process Management System Calls

System calls in Linux	Description
fork	Create a new process.
vfork	Replace the current process with a new one.
exit	Terminate a process
wait	It makes a parent process stop its execution till the termination of the child process.
waitpid	It makes a parent process stop its execution till the termination of the specified child process.(Multiple child process)
exec	It loads a new program in child process

fork() System Call

The `fork()` system call is the primary method for process creation in Unix-like operating systems. It creates a new process, known as the **child process**, from an existing one, called the **parent process**. If the parent process terminates unexpectedly, the child process is also terminated.

Syntax:

```
pid_t pid = fork();
```

Return Values:

- **Zero (0):** Returned to the child process.
- **Positive Value:** Returned to the parent process, representing the Process ID (PID) of the child.
- **Negative Value:** Indicates an error in process creation.

Example Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
```

```
    pid_t p;
    p = fork();  // Creating a new process
    if (p == 0)
        printf("Hello from Child Process, PID=%d\n", getpid());
    else if (p > 0)
        printf("Hello from Parent Process, PID=%d\n", getpid());
    else
        printf("Error: fork() failed\n");
    return 0;
}
```

Expected Output:

```
Hello from Parent Process, PID=1601
Hello from Child Process, PID=1602
```

vfork() System Call

The `vfork()` system call is another method for creating a new process. However, unlike `fork()`, the child process shares the same address space as the parent process.

Key Characteristics:

- The **child process does not get a separate address space**; instead, it runs in the same memory space as the parent.
- The **parent process is paused** until the child process completes execution.
- Any modifications made by the child process to the code or data are **reflected in the parent process**.

This behavior makes `vfork()` more efficient than `fork()` in scenarios where the child process immediately calls `exec()` to replace itself with a new program. However, improper use can lead to **unintended behavior** due to shared memory space.

fork()	vfork()
Child process and parent process has separate address spaces.	Child process and parent process shares the same address space.
Parent and child process execute simultaneously.	Parent process remains suspended till child process completes its execution.

If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate.	If child process alters any page in the address space, it is visible to the parent process as they share the same address space.

vfork() System Call

The `vfork()` system call creates a new process, known as the **child process**, from an existing one, called the **parent process**. Unlike `fork()`, the child process **shares the same address space** as the parent and **suspends the parent process** until it terminates.

Syntax:

```
pid_t pid = vfork();
```

Example Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t p;
    p = vfork();
    if (p == 0) {
        // Child process
        printf("This is the child process. PID: %d\n", getpid());
        printf("Child process is exiting with exit()\n");
        exit(0);
    } else if (p > 0) {
        // Parent process resumes after child terminates
        printf("This is the parent process. PID: %d\n", getpid());
    } else {
        printf("Error: vfork() failed\n");
    }
    return 0;
}
```

Expected Output:

```
This is the child process. PID: 91
Child process is exiting with exit()
This is the parent process. PID: 90
```

wait() System Call

The `wait()` system call is used by a **parent process** to pause its execution **until one of its child processes terminates**.

Syntax:

```
pid_t wait(int *status);
```

Parameters & Return Value:

- `status`: A pointer to an integer where the exit status of the terminated child is stored. It can be replaced with `NULL` if the exit status is not needed.
- **Returns:**
  - **Process ID:** The PID of the terminated child process.
  - **-1:** If an error occurs during execution.

Example Program:

```
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    pid_t p, childpid;
    p = fork();
    if (p == 0) {
        // Child process
        printf("Child: I am running!\n");
        printf("Child: I have PID: %d\n", getpid());
        exit(0);
    } else {
```

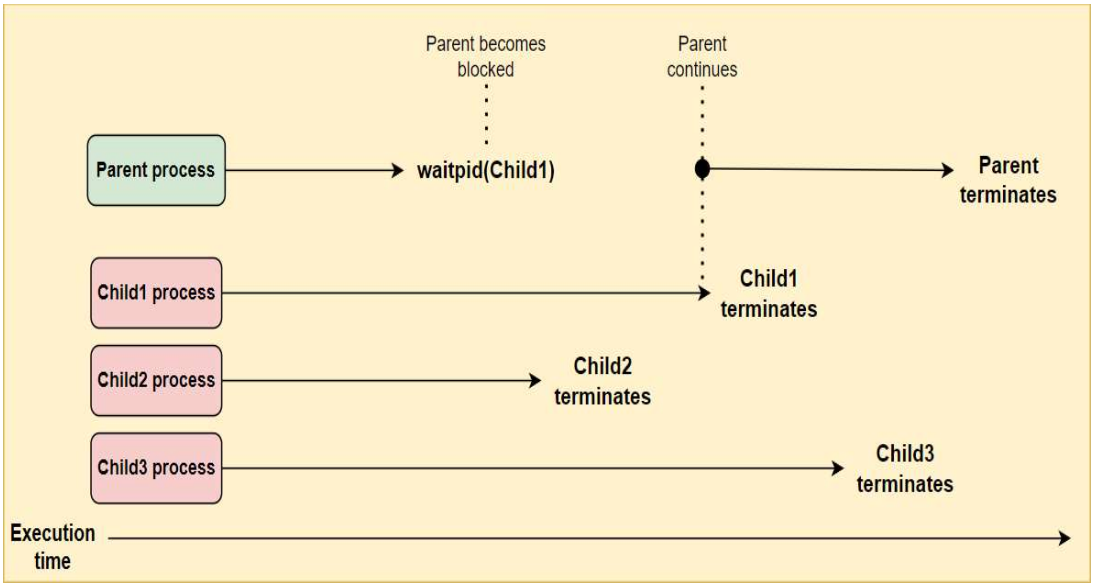
```
        // Parent process
        printf("Parent: I am running and waiting for child to
   finish!\n");
        childpid = wait(NULL); // Parent waits for child to terminate
        printf("Parent: Child finished execution! It had the PID:
   %d\n", childpid);
     }
     return 0;
}
```

Expected Output:

```
Parent: I am running and waiting for child to finish!
Child: I am running!
Child: I have PID: 102
Parent: Child finished execution! It had the PID: 102
```

waitpid() System Call

The `waitpid()` system call **allows a parent process to wait for a specific child process** to terminate, providing more control than `wait()`. It is particularly useful when dealing with multiple child processes.



vfork() System Call

The `vfork()` system call creates a new process (child process) from an existing one (parent process). Unlike `fork()`, the child process **shares the same address space** as the parent and **suspends the parent process** until it terminates.

Syntax:

```
pid_t pid = vfork();
```

Example Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t p = vfork();

    if (p == 0) {
        printf("Child process. PID: %d\n", getpid());
        printf("Child exiting\n");
        exit(0);
    } else if (p > 0) {
        printf("Parent process. PID: %d\n", getpid());
    } else {
        printf("Error: vfork() failed\n");
    }
    return 0;
}
```

wait() System Call

The `wait()` system call pauses a **parent process** until one of its child processes terminates.

Syntax:

```
pid_t wait(int *status);
```

Example Program:

```
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    pid_t p = fork();

    if (p == 0) {
        printf("Child: PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Parent waiting for child to finish\n");
        wait(NULL);
        printf("Parent: Child terminated\n");
    }
    return 0;}
```

waitpid() System Call

The `waitpid()` system call allows a parent to wait for a specific child process to terminate.

Syntax:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Example Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    pid_t pids[2], wpid;
```

```
    pids[0] = fork();
    if (pids[0] == 0) {
        printf("First child: PID = %d\n", getpid());
        exit(0);
    }
        pids[1] = fork();
    if (pids[1] == 0) {
        printf("Second child: PID = %d\n", getpid());
        exit(0);
    }
        wpid = waitpid(pids[1], NULL, 0);
    printf("Parent: Second child (PID = %d) has terminated.\n",
  pids[1]);
        return 0;
}
```

exec() System Call

The `exec()` system call replaces the current process with a new program.

Syntax:

```
int execl(const char *path, const char *arg, ..., NULL);
```

Example Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child executing '/bin/ls'\n");
        execl("/bin/ls", "ls", NULL);
    }
    return 0;
}
```

OPERATING SYSTEMS

exit() System Call

A process terminates by using the `exit()` system call.

Syntax:

```
exit();
```

ORPHAN AND ZOMBIE PROCESSES

- **Orphan Process:** A child process that continues running even after its parent terminates.
- **Zombie Process:** A process that has finished execution but still exists in the process table.

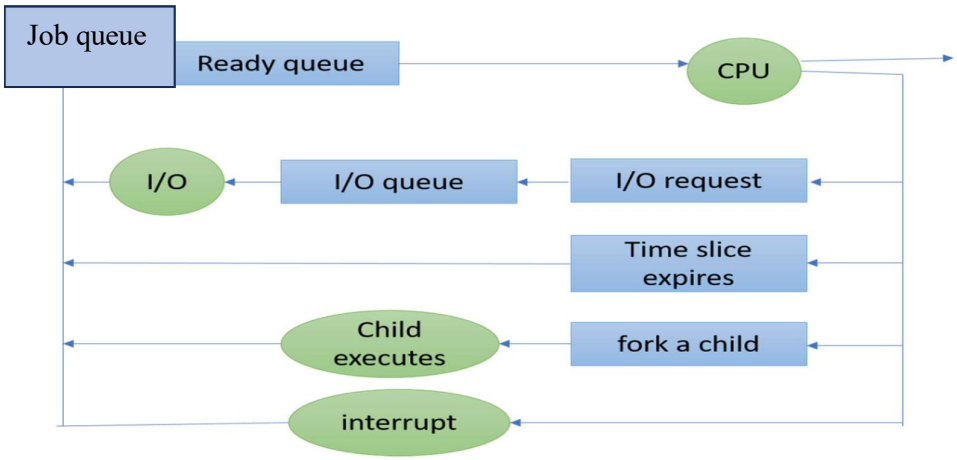
PROCESS SCHEDULING

To increase CPU utilization, multiple processes are loaded into memory. The act of selecting which process should run is called **Process Scheduling**.

Process Scheduling Queues:

1. **Job Queue** – Stores all processes initially.
2. **Ready Queue** – Holds processes that are ready to execute.
3. **Device Queue** – Stores processes waiting for I/O operations.

A process transitions between these queues based on its execution state.



OPERATING SYSTEMS

Process Termination and State Changes

A process can transition between different states based on various system conditions:

1. **Normal Termination:** The process completes execution and exits the system.
2. **I/O Request:** The process requests an I/O operation and moves to the device queue. Once the I/O operation is completed, it returns to the ready queue.
3. **Time Quantum Expiry:** In time-sharing systems, each process is allocated a fixed CPU time. When this time expires, the process is moved back to the ready queue.
4. **Child Process Creation:** When a process creates a child process and waits for its termination, it moves to the ready queue.
5. **Higher Priority Process:** If a higher-priority process arrives, it preempts the currently running process, forcing it back into the ready state.

PROCESS SCHEDULERS

Schedulers are system components responsible for managing process execution. They determine which processes should run, move between states, or be removed from execution.

Schedulers make decisions in the following situations:

- When a running process requests an I/O operation and moves to the waiting state.
- When a process terminates.
- When a process's allocated CPU time is exhausted, requiring another process to take over.
- When an I/O operation completes, making a previously waiting process ready to execute.

There are **three types of schedulers:**

1. Long-Term Scheduler (Job Scheduler)

- Controls the **degree of multiprogramming** by selecting which processes enter the system for execution.
- Chooses a balanced mix of **CPU-bound** and **I/O-bound** processes from secondary memory (new state).
    - **CPU-bound processes** require extensive CPU processing time.
    - **I/O-bound processes** perform many input/output operations and rely less on CPU time.
- Loads selected processes into main memory, moving them to the ready queue for execution.

2. Short-Term Scheduler (CPU Scheduler)

- Improves system performance by selecting which process to execute next from the ready queue.
- Works closely with the **dispatcher**, which assigns the selected process to the CPU.
- Plays a crucial role in determining the next process to run based on scheduling policies.

3. Medium-Term Scheduler (Swapper)

- Manages process swapping to optimize memory usage and ensure efficient execution.
- If memory is needed, it **swaps out** processes from main memory to secondary storage to free up space.
- When memory becomes available again, it **swaps in** suspended processes, allowing them to resume execution from where they left off.
- Helps in reducing the **degree of multiprogramming** by temporarily removing inactive processes.

The major differences between long term, medium term and short-term scheduler are as follows -

Long term scheduler	Medium term scheduler	Short term scheduler
Long term scheduler is a job scheduler.	Medium term is a process of swapping schedulers.	Short term scheduler is called a CPU scheduler.
The speed of long term is lesser than the short term.	The speed of medium term is in between short- and long-term scheduler.	The speed of short term is fastest among the other two.
Long term controls the degree of multiprogramming.	Medium term reduces the degree of multiprogramming.	The short term provides lesser control over the degree of multiprogramming.
The long term is almost nil or minimal in the time-sharing system.	The medium term is a part of the time sharing system.	Short term is also a minimal time sharing system.
The long term selects the processes from the pool and loads them into memory for execution.	Medium term can reintroduce the process into memory and execution can be continued.	Short term selects those processes that are ready to execute.

OPERATING SYSTEMS

CPU SCHEDULING

CPU scheduling is the process of managing the execution of multiple processes by allowing one process to use the CPU while others wait for necessary resources (such as I/O operations). This ensures optimal CPU utilization. Whenever the CPU becomes idle, the **short-term scheduler (CPU scheduler)** selects a process from the ready queue for execution.

Types of CPU Scheduling

CPU scheduling can be classified into two types:

1. Preemptive Scheduling

- A process can be **interrupted** if a higher-priority process enters the queue.
- The current running process **switches back** to the ready queue, allowing the high-priority process to use the CPU.
- The process can also move to the waiting state if it requires an I/O operation. Once the I/O operation is complete, it rejoins the ready queue.

Advantages:

- Prevents a single process from monopolizing the CPU.
- Improves system response time.
- Allows dynamic priority adjustments.
- Reduces the risk of deadlocks.

Disadvantages:

- Requires extra processing time for context switching.
- May lead to **starvation** (low-priority processes waiting indefinitely).

2. Non-Preemptive Scheduling

- Once a process starts execution, it **cannot be interrupted** until it completes its CPU cycle.
- New processes must wait in the queue until the current process finishes.
- If an executing process requires I/O, it moves to the waiting state, and upon completion, it **returns to the top** of the queue.

OPERATING SYSTEMS

Advantages:

- Simple and has minimal overhead.
- Easier to implement compared to preemptive scheduling.

Disadvantages:

- Cannot handle priority scheduling effectively.
- May cause **deadlock** if a long-running process blocks critical tasks.

CPU SCHEDULING CRITERIA

Different CPU scheduling algorithms have unique characteristics. The selection of an appropriate algorithm depends on various performance factors:

1. **CPU Utilization:**
    - Ensures that the CPU remains as busy as possible.
    - Typically ranges from **40% to 90%** in real-time systems.
2. **Throughput:**
    - Measures the number of processes completed per unit of time.
3. **Turnaround Time:**
    - The total time taken by a process from **arrival** to **completion**.
    - **Formula:** Turnaround Time = Completion Time – Arrival Time.
4. **Waiting Time:**
    - Time spent by a process **waiting** in the ready queue.
    - **Formula:** Waiting Time = Turnaround Time – Burst Time.
5. **Response Time:**
    - The time from process **submission** to the first CPU allocation.
    - **Formula:** Response Time = First CPU Allocation Time – Arrival Time.

Objectives of CPU Scheduling Algorithms:

Maximize:

- CPU Utilization → Ensures efficient CPU use.
- Throughput → Increases the number of completed processes per unit time.

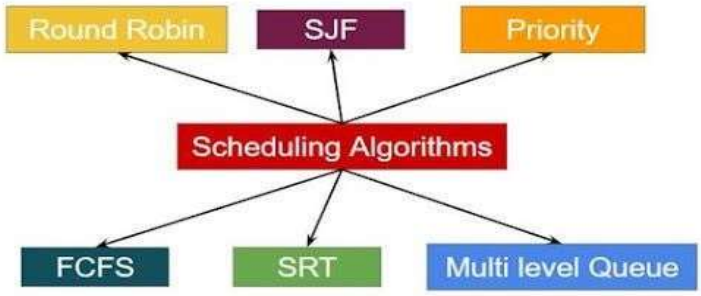
OPERATING SYSTEMS

Minimize:

- Waiting Time → Reduces idle time in the ready queue.
- Response Time → Improves interactivity by reducing initial waiting time.
- Turnaround Time → Ensures faster execution of processes.

TYPES OF CPU SCHEDULING ALGORITHMS

1. **FCFS (First Come, First Serve):**
   - Processes are executed in the order they arrive.
   - Simple but can lead to long waiting times (convoy effect).
2. **SJF (Shortest Job First):**
   - The process with the **smallest burst time** is executed first.
   - Can be **preemptive or non-preemptive**.
3. **SRT (Shortest Remaining Time):**
   - A preemptive version of SJF where the process with the **least remaining execution time** is prioritized.
4. **RR (Round Robin Scheduling):**
   - Each process gets a fixed **time quantum** for execution before moving back to the queue.
   - Ensures fairness but may increase context switching overhead.
5. **Priority Scheduling:**
   - Processes are scheduled based on their priority level.
   - Can be **preemptive** (higher-priority processes interrupt lower ones) or **non-preemptive**.



First Come First Serve (FCFS) Scheduling

FCFS is the simplest CPU scheduling algorithm, where the process that **arrives first** gets executed first. It follows the **FIFO (First-In, First-Out) queue** method.

A real-world example of FCFS is a **cash counter queue**, where customers are served in the order they arrive. Similarly, in CPU scheduling, processes are executed in the sequence they request CPU access.

Advantages of FCFS:

- **First-come, first-served** – Ensures fairness as processes are executed in the order they arrive.
- **Simple to implement** – Uses a straightforward queue structure (FIFO).
- **Easy to program** – Requires minimal complexity for scheduling.

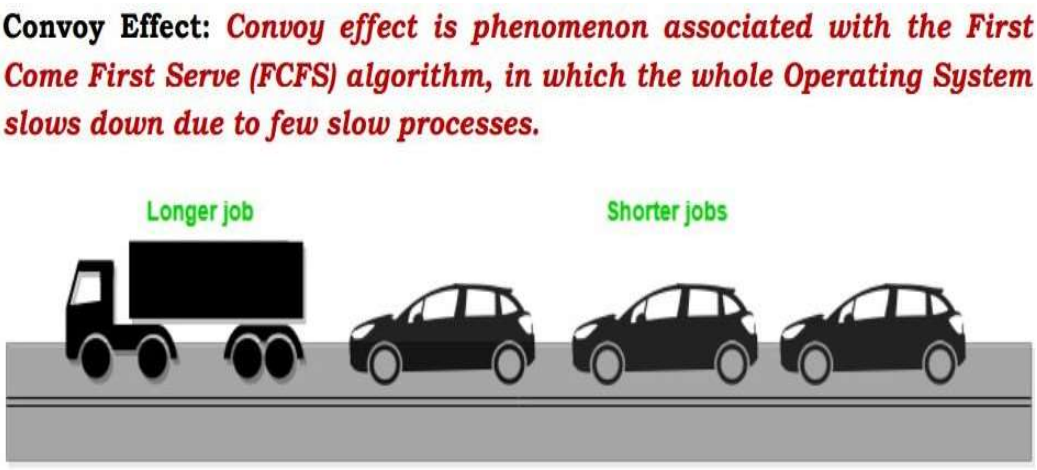
Disadvantages of FCFS:

- **Non-preemptive nature** – Does not prioritize urgent tasks.

  Example: If a **low-priority** process (such as a routine backup) is running and a **critical process** (e.g., system crash handler) arrives, the critical process must **wait**, potentially causing a system failure.

- **High average waiting time** – Processes with longer execution times **delay all subsequent tasks**, leading to inefficiency.
- **Convoy Effect** – A long-running process **blocks shorter processes**, reducing CPU and resource utilization.

  Example: Multiple small processes needing quick CPU access may be **stuck waiting** behind a single long process, leading to poor system performance.



Example 1 -FCFS Scheduling:(Without Arrival Times)

TAT=Completion Time-Arrival Time Waiting Time=Turn Around Time-Burst Time

Response Time =First Response - Arrival Time

Process	Burst time(milliseconds)
P1	5
P2	24
P3	16
P4	10
P5	3

Gantt Chart for FCFS: **(Generalized Activity Normalization Time Table (GANTT))**

A Gantt chart is a horizontal bar chart used to represent operating systems CPU scheduling in graphical view that help to plan, coordinate and track specific CPU utilization factor like throughput,waiting time, turnaround time etc.

Average turnaround time:

TAT=Completion Time-Arrival Time

Turnaround time for p1= 5=0=5. Turnaround time for p2=29-0=29 Turnaround time for p3=45-0=45 Turnaround time for p4=55-0=55

Turnaround time for p5= 55-0=58

Average turnaround time= (5+29+45+55+58/5) = 187/5 =37.5 milli seconds

Average waiting time: Waiting Time=Turn Around Time-Burst TimeWaiting time for p1=5-5=0 Waiting time for p2=29-24=5 Waiting time for p3=45-16=29

Waiting time for p4=55-10=45Waiting time for p5=58-3=55

Average waiting time= (0+5+29+45+55)/5 = 125/5 = 25 ms.

Process	Burst Time (milliseconds)	Completion Time	Turnaround Time	Waiting Time	Response time
P1	5	5	5	0	0
P2	24	29	29	5	5
P3	16	45	45	29	29
P4	10	55	55	45	45
P5	3	58	58	55	55

OPERATING SYSTEMS

First Come First Serve:(With Arrival Times)

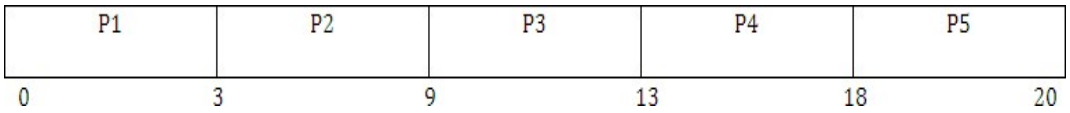
TAT=Completion Time-Arrival Time

Waiting Time=Turn Around Time-Burst Time

First Response - Arrival Time

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	6	2
P3	4	4
P4	5	6
P5	2	8

Gantt Chart



Average Turn Around Time

TAT=Completion Time-Arrival Time Turn Around Time for P1 => 3-0= 3 Turn Around Time for P2 => 9-2 = 7 Turn Around Time for P3 => 13-4=9 Turn Around Time for P4 => 18-6= 12Turn Around Time for P5 => 20-8=12

Average Turn Around Time => (3+7+9+12+12)/5 =>43/5 = 8.50 ms.

Average Response Time:

Response Time = First Response - Arrival Time

Response Time of P1 = 0 Response Time of P2 => 3-2 = 1 Response Time of P3 => 9-4 = 5 Response Time of P4 => 13-6 = 7

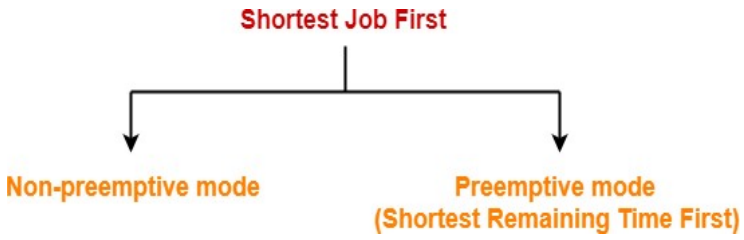
Response Time of P5 => 18-8 =10

Average Response Time => (0+1+5+7+10)/5 => 23/5 = 4.6 ms

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time	Waiting Time	Response Time
P1	3	0	3	3	0	0
P2	6	2	9	7	1	1
P3	4	4	13	9	5	5
P4	5	6	18	12	7	7
P5	2	8	20	12	10	10

Shortest Job First (SJF)

Unlike **FCFS**, where processes are scheduled based on arrival time, **SJF** prioritizes execution based on **burst time**. The process with the **shortest burst time** among the available processes in the **ready queue** is scheduled next. If two processes have the **same burst time**, the tie is resolved using **FCFS (First Come, First Serve) Scheduling**.



Non-Premptive Mode (Example)

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

Turn Around time = Exit time – Arrival time

Waiting time = Turn Around time – Burst time

Process Id	Exit time/Finishing/Completion Time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11
P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

Now,

- ***Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit***
- ***Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit***

Advantages of SJF Scheduling

- **Faster execution for shorter processes**: Short processes are prioritized, leading to quicker turnaround times.
- **Increased throughput**: Since shorter processes are completed first, more processes can be executed in less time.

Disadvantages of SJF Scheduling

- **Requires prior knowledge of burst time**: The CPU must know how long each process will take before execution, which is often impractical.
- **Starvation of longer processes**: If shorter processes keep arriving, longer processes may experience indefinite delays.

OPERATING SYSTEMS

Shortest Remaining Time First (SRTF) Scheduling

SRTF is the **preemptive version** of **Shortest Job First (SJF)** scheduling. In this approach, the CPU always selects the process that has the **smallest remaining burst time** for execution. If a new process arrives with a shorter remaining time than the currently running process, the CPU **preempts** the running process and schedules the new one.

Consider a set of **six processes** with their **arrival time** and **burst time** as follows:

Process Id	Arrival time	Burst time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

Gantt Chart-

Gantt Chart

Now, we know-

Turn Around time = Exit time – Arrival time

Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	19	19 – 0 = 19	19 – 7 = 12
P2	13	13 – 1 = 12	12 – 5 = 7
P3	6	6 – 2 = 4	4 – 3 = 1
P4	4	4 – 3 = 1	1 – 1 = 0
P5	9	9 – 4 = 5	5 – 2 = 3
P6	7	7 – 5 = 2	2 – 1 = 1

Advantages of SRTF Scheduling

- **Faster execution compared to SJF**: Since it is pre-emptive, processes are executed more efficiently based on their remaining burst time.

Disadvantages of SRTF Scheduling

- **Frequent context switching**: More interruptions lead to increased overhead, affecting system performance.
- **Starvation risk**: Like SJF, longer processes may suffer indefinite delays if shorter processes keep arriving.
- **Not suitable for interactive systems**: The exact CPU time required for each process is often unknown, making it difficult to implement in real-world applications.

Priority Scheduling Algorithm

Priority Scheduling in OS

Priority scheduling is a CPU scheduling algorithm that assigns a priority level to each process. Processes with higher priority are executed before those with lower priority.

OPERATING SYSTEMS

Types of Priority Scheduling

1. **Preemptive Priority Scheduling**
   - If a newly arrived process has a higher priority than the currently running process, the CPU stops the execution of the current process and allocates resources to the higher-priority process.
2. **Non-Preemptive Priority Scheduling**
   - If a new process with a higher priority arrives, it is placed at the front of the ready queue. However, it must wait until the current process finishes execution before it gets CPU time.

Priority Assignment

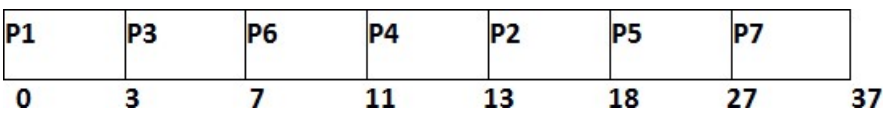
Priorities can be either **static** (fixed during execution) or **dynamic** (changing based on system conditions).

Static priority: It doesn't change priority throughout the execution of the process

Dynamic priority: In this dynamic priority, priority can be changed by the scheduler at a regular interval of time.

Non-Preemptive Priority Scheduling

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4
7	10	7	10



OPERATING SYSTEMS

processId	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
1	2	0	3	3	3	0	0
2	6	2	5	18	16	11	13
3	3	1	4	7	6	2	3
4	5	4	2	13	9	7	11
5	7	6	9	27	21	12	18
6	4	5	4	11	6	2	7
7	10	7	10	37	30	18	27

Avg Waiting Time = (0+11+2+7+12+2+18)/7 = 52/7 units

Preemptive Priority CPU Scheduling

Steps for Priority Scheduling Algorithm

1. **Start with the First Process**
   - Select the process that arrives at time **t = 0**, as it is the only one executing at the beginning.
2. **Check the Priority of the Next Available Process**
   - Compare the priority of the currently executing process with the next process in the queue:
     - If **priority(current process) > priority(previous process)** → Execute the current process.
     - If **priority(current process) < priority(previous process)** → Continue executing the previous process.
     - If **both have the same priority** → Execute the process that arrived first.
3. **Repeat Step 2 for All Processes**
   - Continue checking priorities and selecting the highest-priority process until all processes have been considered.
4. **Execute the Final Process**
   - Once the last process is reached, execute it based on priority order.
   - Continue this process until all processes are completed.

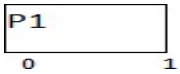
OPERATING SYSTEMS

Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

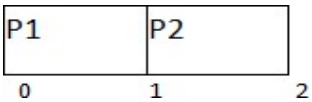
GANTT chart Preparation

At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).

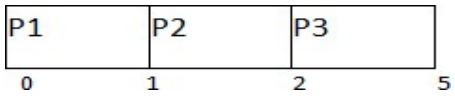
At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time



hence the Operating system has to schedule it regardless of the priority assigned to it.

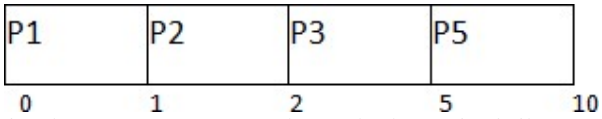


The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution ofP2 will be stopped and P3 will be scheduled on the CPU.

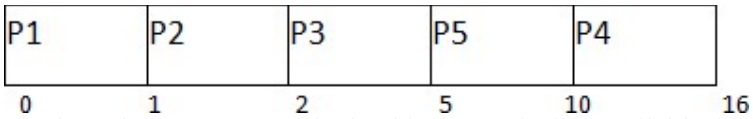


During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so P3 can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.

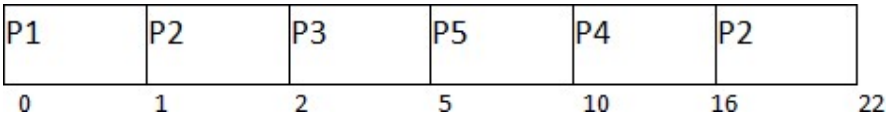
Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the



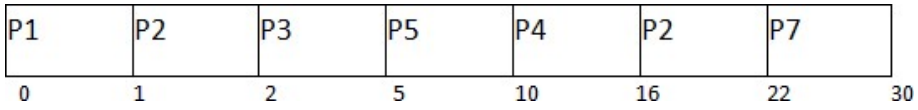
algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



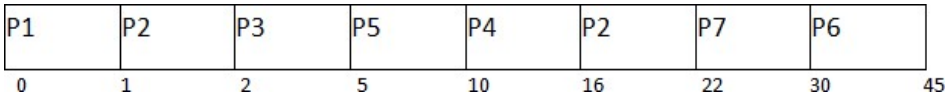
Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.



The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.



The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.

1. Turnaround Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

ProcessId	Priority	ArrivalTime	BurstTime	CompletionTime	Turn around Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

Avg Waiting Time = (0+14+0+7+1+25+16)/7 = 63/7 = 9 units

Advantages of Priority Scheduling in OS

- Simple and easy to implement.
- Ensures that high-priority processes execute first, reducing waiting time for critical tasks.
- Clearly defines the importance of each process.
- Ideal for applications with varying resource and time requirements.

Disadvantages of Priority Scheduling in OS

- Low-priority processes may be lost if the system crashes before execution.
- Can lead to **starvation**, where low-priority processes are indefinitely delayed due to continuous execution of high-priority processes.

Aging Technique

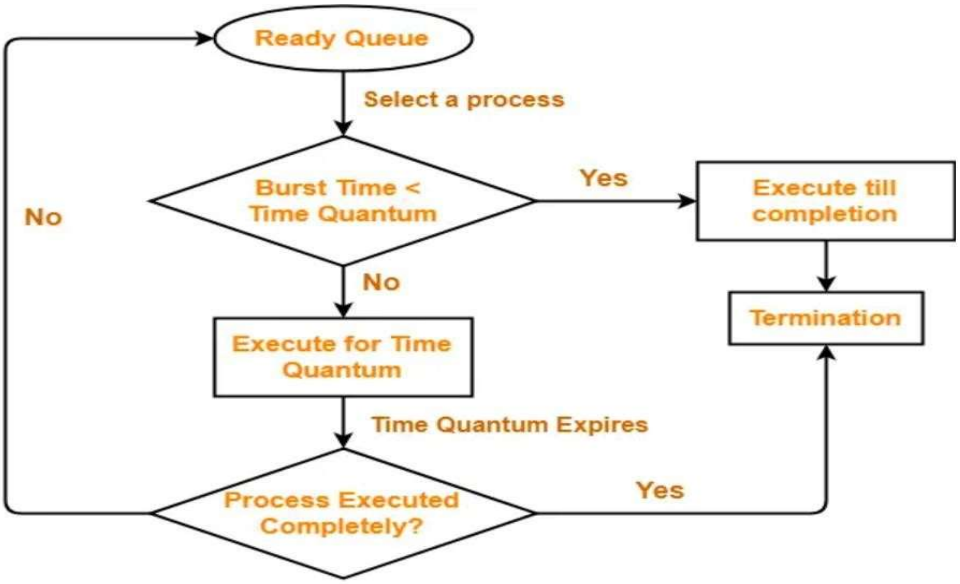
To prevent **starvation**, an aging technique is used. It gradually increases the priority of waiting processes over time, ensuring that no process is left waiting indefinitely.

OPERATING SYSTEMS

Round Robin Scheduling

- Processes are assigned CPU time in a cyclic order based on **First-Come, First-Served (FCFS)**.
- A **fixed time quantum (time slice)** is allocated to each process.
- When the time quantum expires, the running process is **preempted** and moved back to the **ready queue**.
- The CPU is then assigned to the next process in line.
- **Always preemptive**, ensuring fair allocation of CPU time among all processes.

> Round Robin Scheduling is FCFS Scheduling with preemptive mode.



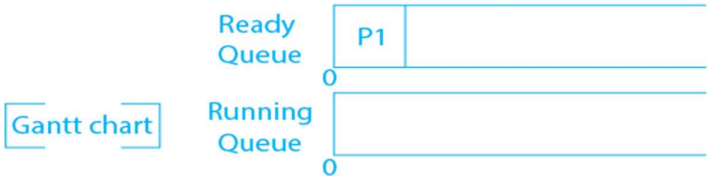
Round-robin scheduling algorithm with an example.

Let's see how the round-robin scheduling algorithm works with an example. Here, we have taken an example to understand the working of the algorithm. We will also do a dry run to understand it better.

Process No	Arrived Time	Brust Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1

In the above example, we have taken 4 processes P1, P2, P3, and P4 with an arrival time of 0,1,2,and 4 respectively. They also have burst times 5, 4, 2, and 1 respectively. Now, we need to createtwo queues the ready queue and the running queue which is also known as the **Gantt chart.**

Process No	Arrived Time	Brust Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1

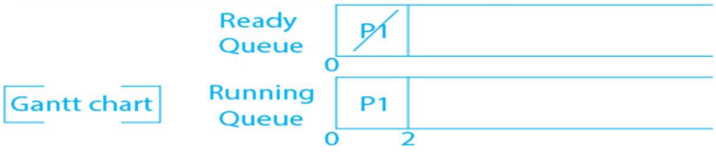


Step 1: first, we will push all the processes in the ready queue with an arrival time of 0. In thisexample, we have only P1 with an arrival time of 0.

This is how queues will look after the completion of the first step.

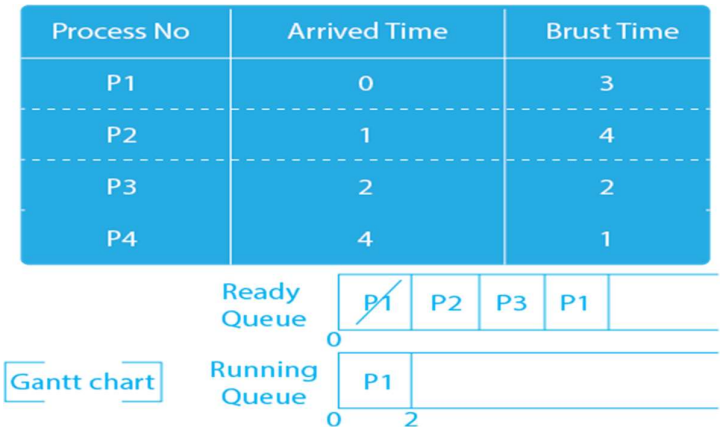
Step 2: Now, we will check in the ready queue and if any process is available in the queue then we will remove the first process from the queue and push it into the running queue. Let's see how the queuewill be after this step.

Process No	Arrived Time	Brust Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



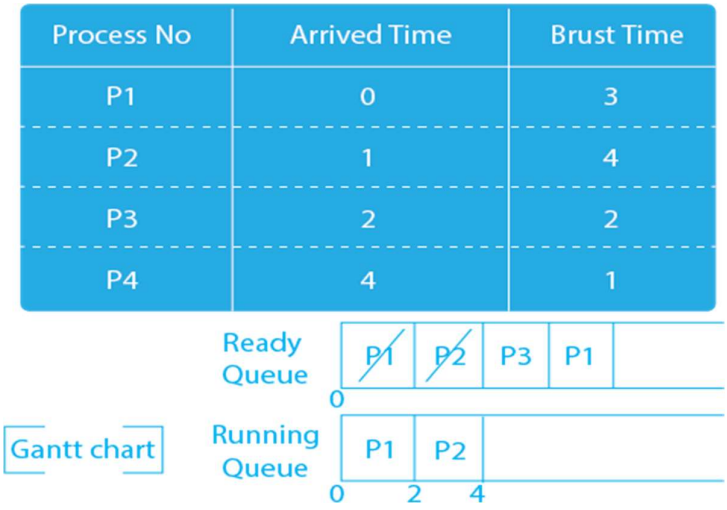
In the above image, we can see that we have pushed process P1 from the ready queue to the runningqueue. We have also decreased the burst time of process P1 by 2 units as we already executed 2 unitsof P1.

Step 3: Now we will push all the processes arrival time within 2 whose burst time is not 0.



In the above image, we can see that we have two processes with an arrival time within 2 P2 and P3 so,we will push both processes into the ready queue. Now, we can see that process P1 also has remainingburst time so we will also push process P1 into the ready queue again.

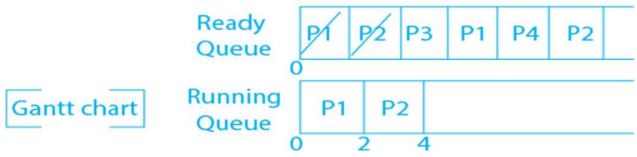
Step 4: Now we will see if there are any processes in the ready queue waiting for execution. If there isany process then we will add it to the running queue.



In the above image, we can see that we have pushed process P2 from the ready queue to the runningqueue. We also decreased the burst time of the process P2 as it already executed 2 units.

Step 5: Now we will push all the processes arrival time within 4 whose burst time is not 0.

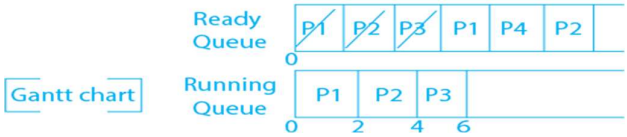
Process No	Arrived Time	Brust Time
P1	0	3
P2	1	2
P3	2	2
P4	4	1



In the above image, we can see that we have one process with an arrival time within 4 P4 so, we willpush that process into the ready queue. Now, we can see that process P2 also has remaining burst time so we will also push process P2 into the ready queue again.

Step 6: Now we will see if there are any processes in the ready queue waiting for execution. If thereis any process then we will add it to the running queue.

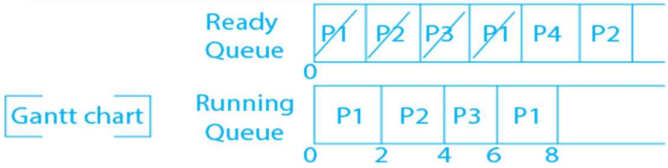
Process No	Arrived Time	Brust Time
P1	0	3
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that we have pushed process P3 from the ready queue to the running queue. We also decreased the burst time of the process P3 as it already executed 2 units. Now, process P3's burst time becomes 0 so we will not consider it further.

Step 7: Now we will see if there are any processes in the ready queue waiting for execution. If thereis any process then we will add it to the running queue.

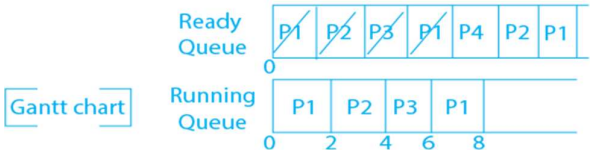
Process No	Arrived Time	Brust Time
P1	0	1
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that we have pushed process P1 from the ready queue to the runningqueue. We also decreased the burst time of the process P1 as it already executed 2 units.

Step 8: Now we will push all the processes arrival time within 8 whose burst time is not 0.

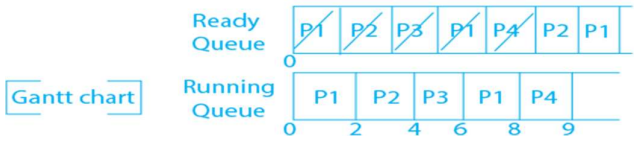
Process No	Arrived Time	Brust Time
P1	0	1
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that process P1 also has a remaining burst time so we will also pushprocess P1 into the ready queue again.

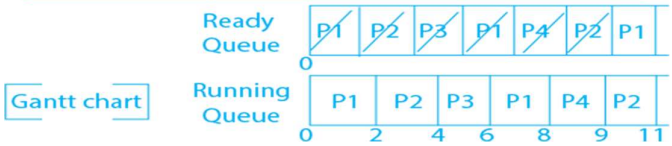
Step 9: Now we will see if there are any processes in the ready queue waiting for execution. If thereis any process then we will add it to the running queue.

Process No	Arrived Time	Brust Time
P1	0	1
P2	1	2
P3	2	0
P4	4	0



In the above image, we can see that we have pushed process P4 from the ready queue to the running queue. We also decreased the burst time of the process P4 as it already executed 1 unit. Now, process P4's burst time becomes 0 so we will not consider it further.

Process No	Arrived Time	Brust Time
P1	0	1
P2	1	0
P3	2	0
P4	4	0

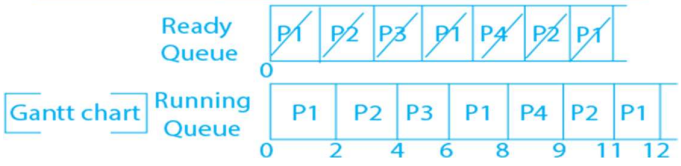


Step 10: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

In the above image, we can see that we have pushed process P2 from the ready queue to the running queue. We also decreased the burst time of the process P2 as it already executed 2 units. Now, process P2's burst time becomes 0 so we will not consider it further.

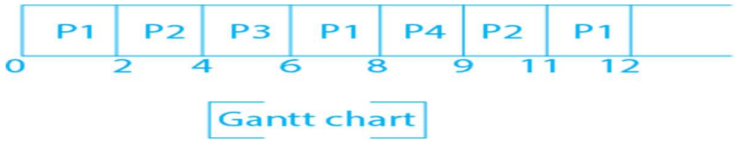
Step 11: Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

Process No	Arrived Time	Brust Time
P1	0	0
P2	1	0
P3	2	0
P4	4	0



In the above image, we can see that we have pushed process P1 from the ready queue to the runningqueue. We also decreased the burst time of the process P1 as it already executed 1 unit. Now, process P1's burst time becomes 0 so we will not consider it further. Now our ready queue is emptyso we will not perform any task now.

After performing all the operations, our running queue also known as the **Gantt chart** will look likethe below.



Let's calculate the other terms like Completion time, Turn Around Time (TAT), Waiting Time (WT), and Response Time (RT). Below are the equations to calculate the above terms.

Turn Around Time = Completion Time – Arrival Time

Waiting Time = Turn Around Time – Burst Time

Response Time = CPU first time – Arrival Time

Let's calculate all the details for the above example.

Process No	Arrived Time	Brust Time	completion Time	TAT	WT	RT
P1	0	5	12	12	7	0
P2	1	4	11	10	6	1
P3	2	2	6	4	2	2
P4	4	1	9	5	4	4

Advantages of Round Robin Scheduling

- Provides **optimal average response time** compared to other scheduling methods.
- Well-suited for **time-sharing systems, client-server architectures, and interactive environments**.
- Eliminates issues like **starvation** and **convoy effect**.
- Ensures **fair CPU allocation** for all processes.
- Treats all processes equally, without considering priority.

Disadvantages of Round Robin Scheduling

- Processes with **longer burst times** may experience delays due to repeated cycles.
- The performance is **highly dependent on the time quantum**.
- **Small time quantum** increases context switching overhead.
- **Large time quantum** makes it behave like **FCFS scheduling**.
- Finding the **optimal time quantum** is challenging.
- **Priority-based execution is not possible**.

INTRODUCTION TO THREADS

A **thread** is the smallest unit of execution within a process. It is often called a **lightweight process** because it runs independently while sharing the same memory space and resources as other threads within the same process. A single process can have multiple threads, each performing a different task while following its own execution path.

Threads enhance application performance by enabling **parallelism**. However, only **one thread is executed at a time** by the CPU, which rapidly switches between threads to create the illusion of parallel execution.

Examples of Thread Usage

- In a **web browser**, multiple tabs function as separate threads.

- When playing a **movie on a device**, different threads handle **audio and video** playback simultaneously.
- On a **web server**, multiple threads process client requests concurrently, improving efficiency without the need for separate processes for each request.

Components of a Thread in an OS

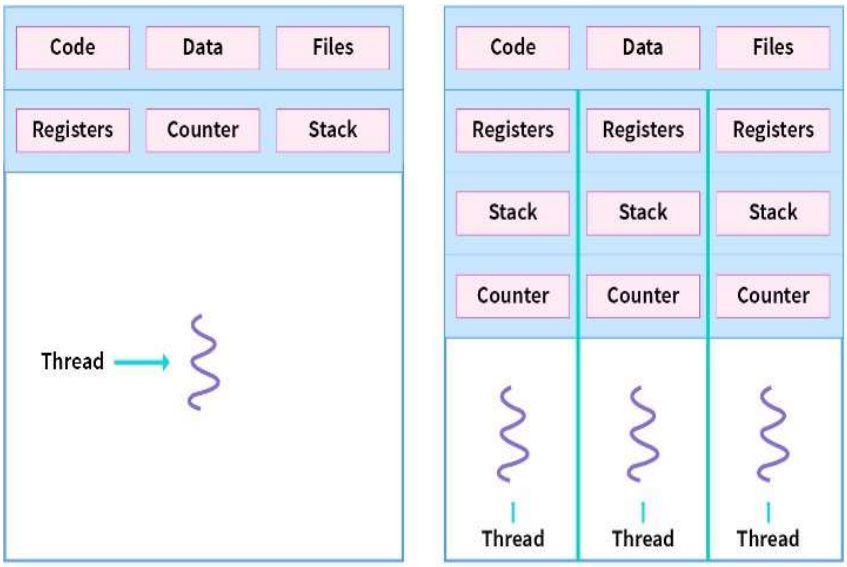
A thread consists of three main components:

1. **Stack Space** – Stores local variables and function calls specific to the thread.
2. **Register Set** – Holds temporary data and keeps track of thread execution.
3. **Program Counter** – Keeps track of the next instruction to be executed.

Single-threaded vs. Multi-threaded Processes

- **Single-threaded Process** – A process with only one thread executing its tasks sequentially.
- **Multi-threaded Process** – A process containing multiple threads, where each thread has its own **registers, stack, and counter** but shares the **code and data segments** with other threads.

Multithreading improves efficiency by allowing multiple tasks to run concurrently within the same process.



Single-threaded process Multithreaded process

Process simply means any program in execution while the thread is a segment of a process.

The main differences between process and thread are mentioned below:

Process	Thread
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.
For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.

Benefits of Multithreaded Programming

Multithreading offers several advantages, categorized into four key areas:

1. Resource Sharing

- In traditional processes, resource sharing requires explicit techniques such as **message passing** or **shared memory**.
- In contrast, **threads share memory and resources by default** within the same process.
- This enables efficient resource management, allowing multiple threads of an application to operate within the same address space.

2. Improved Responsiveness

- Multithreading allows a program to continue running even if a part of it is blocked or engaged in a long task.
- For example, in a **web browser**, one thread can handle **user interactions**, while another loads **images** simultaneously, enhancing the user experience.

3. Efficient Utilization of Multiprocessor Architectures

- In **multiprocessor systems**, multiple threads can execute in **parallel** on different processors, increasing system concurrency.
- Unlike **single-processor systems**, where only one process/thread executes at a time, multiprocessor environments benefit from simultaneous execution.

4. Cost Efficiency

- **Threads are more economical** than processes since they share resources such as memory and system files.
- Creating and managing **new processes** requires significant memory and CPU overhead, whereas **threads are lightweight** and easier to manage.

Types of Threads in an Operating System

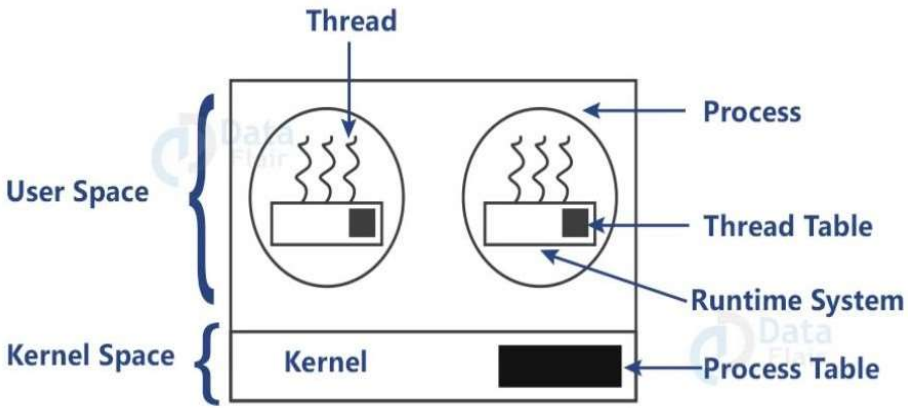
Threads are classified into two categories:

1. User-Level Threads

- These are **managed without kernel involvement** and do not require system calls for creation or management.
- They are **lightweight**, allowing efficient switching between threads without kernel intervention.
- Example: **Java Threads, POSIX Threads (Pthreads)**

2. Kernel-Level Threads

- These threads are **managed directly by the operating system's kernel**.
- They provide better system integration and take advantage of multiprocessor architectures but have a higher overhead compared to user-level threads.



Advantages of User-Level Threads

- **Easier to Implement**: User-level threads are simpler to create and manage compared to kernel-level threads.
- **Lower Context Switching Overhead**: Switching between user-level threads is faster as it does not require kernel intervention.
- **More Efficient Execution**: Since they do not require kernel-mode privileges, user-level threads execute with minimal overhead.
- **Lightweight Representation**: They consist of only essential components such as the **Program Counter, Register Set, and Stack Space**, making them efficient.

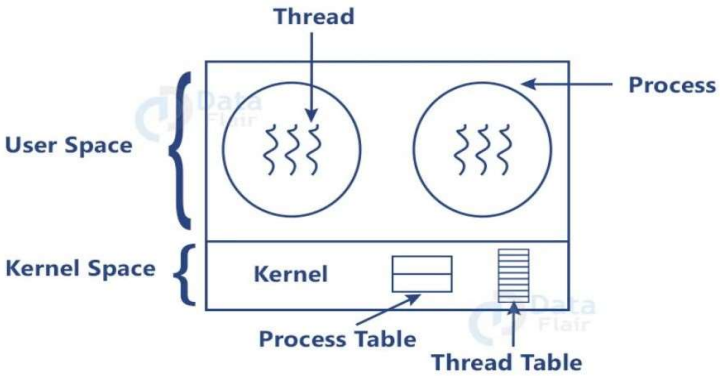
Disadvantages of User-Level Threads

- **Lack of Coordination with the Kernel**: Since the kernel is unaware of user-level threads, it cannot schedule them efficiently.
- **Process Blocking Issue**: If one thread encounters a **page fault**, the entire process, including all its threads, may get blocked.

Kernel-Level Threads

A **Kernel-Level Thread (KLT)** is managed directly by the **Operating System Kernel**. The kernel maintains a **thread table** to track and schedule these threads efficiently. However, **context switching time is higher** for kernel threads due to additional overhead.

Example: Windows, Solaris use kernel-level thread management.



Advantages of Kernel-Level Threads

- **Better Thread Management**: The kernel maintains an updated record of all threads, allowing for efficient scheduling and execution.
- **Handles Blocking Efficiently**: Kernel-level threads can manage processes that frequently block without affecting the entire process.
- **Dynamic Resource Allocation**: If a process requires more execution time, the kernel can allocate additional processing time to its threads.

Disadvantages of Kernel-Level Threads

- **Slower Execution**: Context switching for kernel-level threads involves system calls, making it slower than user-level threads.
- **Complex Implementation**: Managing kernel threads requires more resources and is more complex compared to user-level threads.

User Level threads Vs Kernel Level Threads

User level Threads	Kernel Level Threads
User thread are implemented by Users	Kernel threads are implemented by OS
OS doesn't recognize user level threads	Kernel threads are recognized by OS
Implementation is easy	Implementation is complicated
Context switch time is less	Context switch time is more
Context switch – no hardware support	hardware support is needed

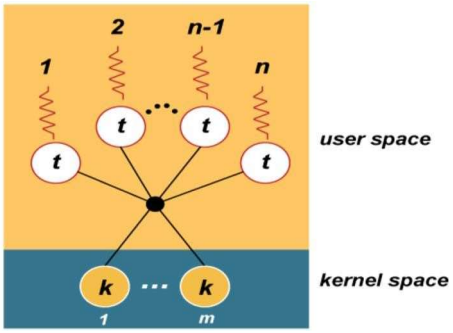
If one user level thread perform blocking operation then entire process will be blocked	If one kernel level thread perform blocking operation then another thread can continue execution.

Multithreading Models

User threads are mapped to kernel threads using different strategies, known as **threading models**. The three primary multithreading models are:

Many-to-Many Model

In this model, multiple user-level threads are mapped to the same or a lesser number of kernel-level threads. The system dynamically schedules user threads onto available kernel threads, ensuring that blocked user threads do not halt the entire process. This approach provides **efficient resource utilization and prevents system-wide blocking**, making it the most effective multithreading model.



Many-to-One Model

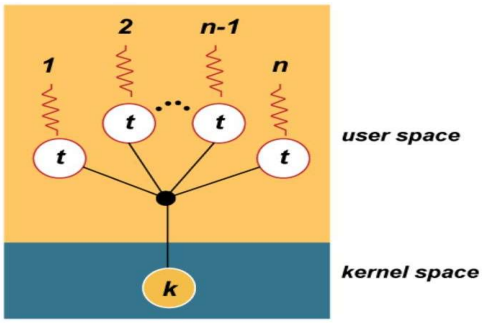
In the **Many-to-One** multithreading model, multiple user-level threads are mapped to a single kernel-level thread. This means that **thread management occurs entirely at the user level**, without kernel involvement in scheduling.

Characteristics of the Many-to-One Model

- Since only **one** kernel thread is available, multiple user threads **cannot execute in parallel on multiprocessor systems**.
- If a user thread makes a **blocking system call**, the entire process is blocked, affecting all other threads.

- Due to **user-level thread management**, this model is **more efficient** in terms of context switching and overhead.

This model is simple to implement but lacks parallelism and is not suitable for systems requiring high concurrency.



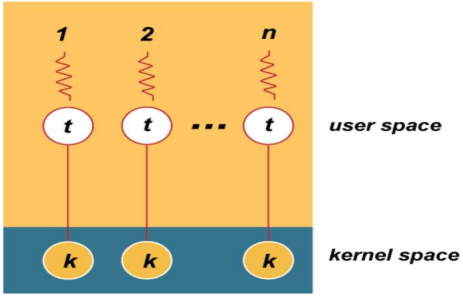
One-to-One Model

The **One-to-One** multithreading model establishes a direct **one-to-one relationship** between user threads and kernel threads. This means that each user thread is mapped to a separate kernel thread, allowing for **true parallel execution** on multiprocessor systems.

Characteristics of the One-to-One Model

- Multiple threads can run simultaneously across multiple processors, enhancing system performance.
- Since each user thread has a dedicated kernel thread, a **blocking system call in one thread does not block other threads**, ensuring better responsiveness.
- However, **creating a new user thread requires creating a corresponding kernel thread**, which increases resource consumption and limits scalability.

This model provides **better concurrency and responsiveness**, but the overhead of managing a large number of kernel threads can impact system performance.



OPERATING SYSTEMS

Threading Issues in Operating Systems

In a multithreading environment, several challenges and complexities arise. Some of the key threading issues include:

1. System Call Behaviour

- When a thread makes a **blocking system call**, it may cause the entire process to be blocked, depending on the threading model used.
- Some operating systems offer variations of **fork()**, where either all threads of a parent process are duplicated in the child process, or only the invoking thread is copied.
- The **exec()** system call replaces the current process, including all its threads, with a new program.

2. Thread Cancellation

Thread cancellation refers to terminating a thread before it has completed execution. There are two main types:

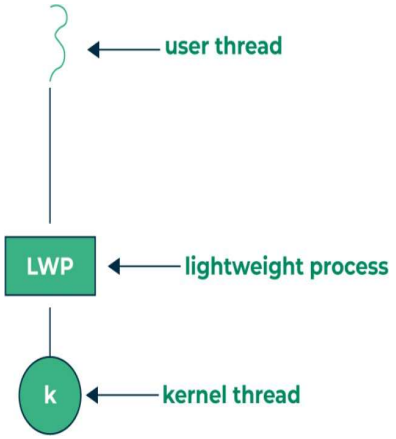
- **Asynchronous Cancellation:** The target thread is terminated immediately.
- **Deferred Cancellation:** The target thread periodically checks for cancellation and terminates itself at a safe point.

Cancelling threads improperly can lead to resource leaks or inconsistent shared data.

3. Signal Handling

Signals notify processes about system events and can be:

- **Synchronous:** Triggered by operations within the process, such as division by zero.



-

Asynchronous: Generated externally, like keyboard interrupts.

In multithreaded programs, signals can be:

- Delivered to the specific thread that caused it.
- Sent to all threads in the process.
- Handled by a dedicated thread.

4. Thread-Specific Data

While threads share process memory, some data needs to be **thread-specific** (e.g., unique transaction IDs in a banking system). Thread-local storage ensures that each thread maintains its own copy of such data.

5. Thread Pool

- Instead of creating new threads for every request, a **thread pool** maintains a set of pre-created threads.
- When a request arrives, an idle thread is assigned the task, reducing the overhead of thread creation and destruction.
- This improves efficiency and prevents resource exhaustion.

6. Scheduler Activation

- Many multithreading models (e.g., **many-to-many** and **two-level**) require coordination between the **kernel and user-level thread library**.
- **Scheduler activation** allows the kernel to notify the thread library about events like **blocked threads**, ensuring better scheduling and resource allocation.

By addressing these challenges, operating systems can optimize multithreading performance while maintaining system stability and efficiency.

• Lightweight Process (LWP) and Process Synchronization

Lightweight Process (LWP)

An **LWP (Lightweight Process)** acts as a **virtual processor** for the user-thread library, allowing applications to schedule user threads for execution. Each LWP is linked to a kernel thread, and the operating system schedules kernel threads to run on physical processors.

- If a **kernel thread blocks** (e.g., while waiting for an I/O operation), the associated **LWP also blocks**, causing the user thread linked to it to be suspended.
- The number of LWPs required depends on the application's nature:
    - A **CPU-bound application** running on a **single processor** needs only **one LWP** since only one thread can execute at a time.
    - An **I/O-intensive application** may require **multiple LWPs**, especially if multiple blocking system calls occur simultaneously.

For example, if five simultaneous file-read requests occur, **five LWPs are needed**. If only four LWPs exist, the fifth request must wait for an LWP to become available.

Upcall Mechanism

- The **kernel informs** an application about specific events using a mechanism called **upcalls**.
- **Upcall handlers**, managed by the thread library, process these notifications.
- A common event triggering an upcall is when a thread is about to **block**.
    - The kernel makes an upcall informing the application that a thread will block.
    - The kernel assigns a **new virtual processor** to run an upcall handler.
    - The upcall handler **saves the state of the blocking thread** and schedules another thread to run.
- When the blocked thread becomes ready again, the kernel sends another **upcall** to notify the thread library. The upcall handler then either **allocates a new virtual processor** or **preempts another thread** to resume execution.

PROCESS SYNCHRONIZATION

Process synchronization ensures that multiple processes execute in a controlled manner while accessing shared resources, preventing race conditions and data inconsistencies.

Producer-Consumer Problem

- A **producer** generates data that a **consumer** uses.
- Example: A compiler produces assembly code, which an assembler consumes.

Buffering Mechanisms

Two types of buffers are used:

1. **Unbounded Buffer**:
    - The buffer has no size limit.
    - The producer can always generate data, but the consumer may have to wait for items.
2. **Bounded Buffer**:
    - The buffer has a fixed size.
    - The producer must wait if the buffer is full, and the consumer must wait if it is empty.

Bounded Buffer Implementation (Inter-Process Communication - IPC)

Shared Variables:

```
#define BUFFER_SIZE 10
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```

Producer Process:

```
while (true) {
  /* produce an item in next_produced */
  while (counter == BUFFER_SIZE); /* Wait if buffer is full */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}
```

Consumer Process:

```
while (true) {
  while (counter == 0); /* Wait if buffer is empty */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
  /* consume the item */
}
```

- The buffer is a **circular array**, with `in` and `out` pointers indicating the next available slot and the next item to be consumed, respectively.
- The **counter** tracks the number of items in the buffer.
- The buffer is **full** when `counter == BUFFER_SIZE`, and **empty** when `counter == 0`.

Race Condition

A **race condition** occurs when multiple processes or threads access shared data **simultaneously**, leading to unpredictable outcomes.

Example: Producer-Consumer Race Condition

Consider a shared `counter` variable (initial value = 5) used by both **producer** and **consumer** processes.

Producer Process:

```
register1 = counter;
register1 = register1 + 1;
counter = register1;
```

Consumer Process:

```
register2 = counter;
register2 = register2 - 1;
counter = register2;
```

If executed sequentially, the output is consistent. However, if executed **concurrently**, the following interleaving could occur:

1. **Producer:** `register1 = counter` (register1 = 5)
2. **Consumer:** `register2 = counter` (register2 = 5)
3. **Producer:** `register1 = register1 + 1` (register1 = 6)
4. **Consumer:** `register2 = register2 - 1` (register2 = 4)
5. **Producer:** `counter = register1` (counter = 6)
6. **Consumer:** `counter = register2` (counter = 4)

Expected final value: 5

Actual final value: 4 (Incorrect due to race condition)

This inconsistency arises because both processes read the **same initial counter value (5)** before updating it. **Proper synchronization mechanisms** (e.g., **mutex locks**, **semaphores**) are needed to prevent such conflicts.

By handling these issues efficiently, operating systems ensure reliable process execution and synchronization in concurrent environments.

Consider this execution interleaving with "count = 5" initially:

S1: producer execute `register1 = counter`	{register1 = 5}
S2: producer execute `register1 = register1 + 1`	{register1 = 6}
S3: consumer execute `register2 = counter`	{register2 = 5}
S4: consumer execute `register2 = register2 - 1`	{register2 = 4}
S5: producer execute `counter = register1`	{counter = 6 }
S6: consumer execute `counter = register2`	{counter = 4}

Critical Section Problem and Process Synchronization

Race Condition and Incorrect Counter Value

A **race condition** occurs when multiple processes access and modify shared resources **concurrently** without proper synchronization, leading to unpredictable results.

In the **Producer-Consumer Problem**, incorrect values of `counter` may occur due to unsynchronized access.

- If the **order of execution changes**, we may end up with an incorrect count of items in the buffer.
- Example:
  - We reached an incorrect state where `counter == 4`, while the correct value should be `5`.
  - If the execution order was different, we could end up with `counter == 6`, another incorrect state.
- This happens because both the **producer and consumer modify `counter` simultaneously**, leading to inconsistencies.
- The correct **final value (counter == 5)** is only achieved when the producer and consumer **execute separately or are properly synchronized**.

OPERATING SYSTEMS

CRITICAL SECTION PROBLEM

The **Critical Section** is the part of a program where shared resources are accessed. **If multiple processes execute their critical sections concurrently, race conditions may occur.**

To avoid **race conditions**, we must ensure that **only one process at a time** can execute its **critical section**.

Process Synchronization

Process synchronization ensures that multiple processes execute **in a controlled manner**, preventing conflicts when accessing shared resources.

Processes are categorized into two types:

1. **Independent Processes**: Execution of one process **does not** affect others.
2. **Cooperative Processes**: Execution of one process **can affect** or be affected by other processes.

Since **cooperative processes share resources**, they require **proper synchronization mechanisms** to avoid conflicts.

Structure of a Process with Critical Section

A process that needs access to a shared resource follows a **standard structure** to avoid conflicts:

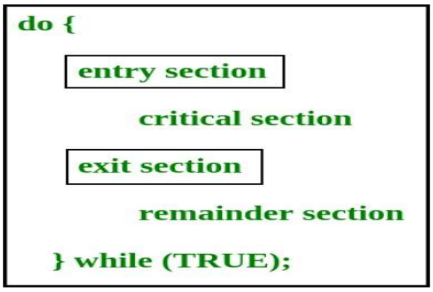
1. **Entry Section** – The process requests permission to enter the critical section.
2. **Critical Section** – The process accesses shared resources safely.
3. **Exit Section** – The process releases the critical section, allowing other processes to enter.
4. **Remainder Section** – The remaining code outside of the critical section.

General Structure of a Process (Pi):

```
while (true) {
    // Entry Section
    request_permission();

    // Critical Section
    access_shared_resource();

    // Exit Section
```

```
    release_permission();

    // Remainder Section
    execute_other_tasks();
}
```

By **implementing proper synchronization mechanisms**, such as **mutex locks, semaphores, or monitors**, we ensure that only one process can access the critical section at a time, preventing race conditions and maintaining data consistency.



Conditions to the Critical Section Problem

To ensure proper synchronization in concurrent processes, any solution to the **Critical Section Problem** must satisfy three key requirements:

1. **Mutual Exclusion** – Only one process can execute in the critical section at a time.
2. **Progress** – If no process is in the critical section and other processes wish to enter, one must be allowed to proceed without indefinite delay.
3. **Bounded Waiting** – There must be a limit on how long a process waits before entering its critical section, preventing starvation.

Peterson's Solution

Peterson's Algorithm is a **software-based** synchronization solution designed for two processes that take turns accessing the critical section.

Implementation:

- The processes are numbered **P0** and **P1**.
- The algorithm uses two shared variables:
    - `int turn` – Indicates whose turn it is to enter.
    - `boolean flag[2]` – Indicates if a process is ready to enter.

OPERATING SYSTEMS

Algorithm:

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);  // Wait if the other process wants to
enter

    // Critical Section

    flag[i] = false;  // Exit critical section

    // Remainder Section
} while (true);
```

How It Works:

- Process `Pi` sets `flag[i] = true` to indicate its intent to enter.
- It sets `turn = j` to allow the other process a chance to proceed.
- If `Pj` is not in the critical section (`flag[j] == false`), `Pi` can enter.
- If both processes try to enter at the same time, the **turn variable ensures only one gets access first**.

Satisfying the Requirements:

- **Mutual Exclusion** – Only one process can enter at a time since `turn` allows only one process to proceed.
- **Progress** – If no process is in the critical section, the turn variable ensures one of the waiting processes proceeds.
- **Bounded Waiting** – Each process waits for at most one execution of the other process before it gets a turn.

Limitations of Peterson's Solution:

- It **only works for two processes**.
- **Modern processors may optimize instructions**, breaking the algorithm's correctness.

OPERATING SYSTEMS

SYNCHRONIZATION USING HARDWARE

To address **multi-processor** synchronization issues, modern systems use **hardware-based synchronization** mechanisms.

1. Test-and-Set Locking Mechanism

The **test_and_set()** instruction executes **atomically**, ensuring that multiple processes do not modify a shared variable simultaneously.

Function Definition:

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

Algorithm for Mutual Exclusion:

```
boolean waiting[i] = false;
boolean lock = false;

do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);  // Atomic operation

    waiting[i] = false;  // Enter Critical Section

    // Critical Section

    // Find next waiting process
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
```

```
    if (j == i)
        lock = false;  // No waiting process, unlock
    else
        waiting[j] = false;  // Pass turn to next process

    // Remainder Section
} while (true);
```

How It Works:

- **First process to execute `test_and_set()` gets access.**
- **Other processes wait until the lock is released.**
- **Ensures mutual exclusion, progress, and bounded waiting.**

2. Swap Mechanism

Another **atomic** hardware operation is **swap()**, which ensures synchronization by modifying a shared variable only if its expected value matches the current value.

Function Definition:

```
int swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Algorithm for Mutual Exclusion:

```
do {
    while (swap(&lock, 0, 1) != 0); // Wait for access

    // Critical Section

    lock = 0;  // Release lock

    // Remainder Section
```

```
} while (true);
```

How It Works:

- The **first process sets `lock = 1` and enters**.
- Other processes **fail the comparison** and must wait.
- Once done, the process **resets `lock = 0`**, allowing the next process to proceed.

Problems with Hardware Solutions:

- **Complexity** – Harder to implement correctly.
- **Limited Accessibility** – Not available for high-level programming.
- **Performance Overhead** – Disabling interrupts on multiprocessors slows system performance.

SEMAPHORES

A **semaphore** is a synchronization tool used in operating systems to solve the **Critical Section problem** and coordinate process execution. It helps manage **concurrent access** to shared resources such as memory, printers, files, etc., by multiple processes.

A semaphore is basically an **integer variable** that is accessed and modified using two atomic operations: wait() and signal().

Atomic Semaphore Operations

The operations performed on semaphores are defined as follows:

wait(S) Operation

```
wait(S)
{
    while (S <= 0); // Busy wait
    S--;
}
```

- If the value of the semaphore S is **less than or equal to 0**, the process **keeps waiting** (busy waiting).
- Once the value becomes positive, the process **decreases the semaphore value by 1** and proceeds.

signal(S) Operation

```
signal(S)
{
    S++;
}
```

- This operation **increments** the value of the semaphore.
- If any processes are waiting, one of them will be allowed to proceed.

Both operations must be **atomic**, meaning they must be executed **without interruption**. No other process should be able to access or modify the semaphore value while another process is operating on it.

Types of Semaphores

Semaphores are mainly of two types:

Binary Semaphore

- Also known as a **mutex lock**.
- The value of the semaphore can only be **0 or 1**.
- Used to allow or deny access to **one process at a time**.
- If the value is 1, a process can enter the critical section. Once it enters, the value becomes 0.

Counting Semaphore

- The value of the semaphore can be any **non-negative integer**.
- Used when there are **multiple instances** of a resource.
- Initialized to the number of available resources.
    - When a process uses a resource, wait() is called, and the value is **decremented**.
    - When a process releases a resource, signal() is called, and the value is **incremented**.
- If the semaphore value becomes **zero**, it indicates that **all resources are in use**, and any further request will make the process **wait until a resource is released**.

Semaphore for Process Synchronization

Semaphores are used to ensure that **certain instructions in one process** are executed only **after some specific instructions in another process**.

Example scenario:

- Two processes P1 and P2.
- P1 contains statement S1 and P2 contains statement S2.
- Requirement: S2 should execute **only after** S1 completes.

Solution using semaphore synch initialized to 0:

In Process P1:

```
S1;
signal(synch);
```

In Process P2:

```
wait(synch);
S2;
```

Here, process P2 will wait at wait(synch) until process P1 executes signal(synch) after completing S1. This ensures the required execution order.

Busy Waiting Problem

In the original implementation of wait () using a loop (while (S <= 0) ;), the process keeps checking the condition continuously. This is known as **busy waiting**, which consumes CPU time unnecessarily.

To overcome this problem, semaphores are implemented with **blocking and wake-up mechanisms** instead of constant checking.

Improved Semaphore Implementation

To remove busy waiting, the wait () and signal () operations are redefined using a **waiting queue**.

A semaphore is now defined as a structure with:

- An integer value.
- A list of processes (queue) that are waiting.

```
typedef struct {
    int value;
```

```
    struct process *list; // Queue of waiting processes
} semaphore;
```

Updated wait () Operation:

```
wait (semaphore *S)
{
    S->value--;
    if (S->value < 0)
    {
        add this process to S->list;
        block (); // Suspend the process
    }
}
```

Updated signal () Operation:

```
signal (semaphore *S)
{
    S->value++;
    if (S->value <= 0)
    {
        remove a process P from S->list;
        wakeup(P); // Resume the process
    }
}
```

Working of Modified Implementation

- If the semaphore value is **less than 0**, the process is **blocked** and placed in the **waiting queue**.
- The CPU **scheduler selects another ready process** to run.
- When another process performs a signal () operation, **one process is removed from the waiting queue** and is moved to the **ready queue** using the wakeup () operation.
- The **block()** operation **suspends the execution** of a process.
- The **wakeup(P)** operation **resumes execution** of a specific process P.

OPERATING SYSTEMS

Data Structures Used

- A **FIFO (First-In-First-Out) queue** is maintained for each semaphore to store waiting processes.
- The **Process Control Block (PCB)** of each process contains a link field used to implement the queue.
- The queue ensures **bounded waiting**, meaning no process will wait forever.

Important Characteristics

- The wait () and signal () functions must be **atomic** to prevent race conditions.
- The improved implementation **reduces CPU waste** caused by busy waiting.
- Binary semaphores are useful for **locking**, while counting semaphores manage **multiple resource instances**.
- Semaphores help maintain **mutual exclusion**, **synchronization**, and **bounded waiting**, all of which are important properties of the **Critical Section problem**.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

In an Operating System, many processes can run at the same time. Sometimes, these processes need to share resources like files, memory, or printers. If they all try to access the same resource at the same time, it can cause problems like data corruption or inconsistency.

To avoid this, we use **process synchronization**. It ensures that shared resources are used safely when multiple processes are involved.

There are 3 famous classical synchronization problems that help us understand this concept:

1. Bounded-Buffer Problem (also called Producer–Consumer Problem)
2. Readers–Writers Problem
3. Dining Philosophers Problem

1. Bounded-Buffer Problem (Producer–Consumer Problem)

What is the Problem?

Imagine a situation:

- There is a **buffer** (a temporary storage area) that can hold a fixed number of items (e.g., size 5).
- A **Producer** process keeps creating items and adding them to the buffer.
- A **Consumer** process keeps removing items from the buffer and using them.

Now, two main conditions:

- If the buffer is **full**, the producer must wait (it cannot add more).
- If the buffer is **empty**, the consumer must wait (nothing to consume).

This requires synchronization between producer and consumer so that they do not access the buffer at the same time or in invalid conditions.

Semaphores Used:

```
int n;
semaphore empty = n; // Counts how many empty slots are available in the buffer
semaphore full = 0; // Counts how many items are in the buffer
semaphore mutex = 1; // Binary semaphore used for mutual exclusion to protect the buffer
```

Producer Process (with Explanation):

```
do {
   // 1. Produce an item and store in 'next_produced'
   wait(empty); // 2. Check if buffer has space. If not, wait until a slot is empty.
   wait(mutex); // 3. Enter critical section (lock the buffer)
// 4. Put 'next_produced' into the buffer
   signal(mutex); // 5. Exit critical section (unlock the buffer)
   signal(full); // 6. One more item is available now, inform consumer
} while (true);
```

Consumer Process (with Explanation):

```
do {
   wait(full); // 1. Wait if buffer is empty (nothing to consume)
   wait(mutex); // 2. Enter critical section (lock the buffer)
   // 3. Remove item from buffer and put into 'next_consumed'
   signal(mutex); // 4. Exit critical section (unlock the buffer)
   signal(empty); // 5. One more empty slot available, inform producer
   // 6. Use or consume the item in 'next_consumed'
```

```
} while (true);
```

2. Readers–Writers Problem

What is the Problem?

Suppose we have a shared file or database:

- **Readers** only read the data. Multiple readers can read at the same time.
- **Writers** update the data. Only one writer can write at a time, and no readers should read while writing.

We must make sure that:

- Multiple readers can read together (if no writer is writing).
- Only one writer can write at a time.
- Readers and writers must not access the data simultaneously.

Variables and Semaphores Used:

```
semaphore rw_mutex = 1; // Controls access to shared data (for writing)
semaphore mutex = 1; // Controls access to read_count variable
int read_count = 0; // Counts the number of readers currently reading
```

Reader Process (with Explanation):

```
do {
   wait(mutex); // 1. Lock 'read_count' variable
   read_count++;
   if (read_count == 1)
       wait(rw_mutex); // 2. First reader locks the shared resource
   signal(mutex); // 3. Unlock 'read_count'
   // 4. Read the shared data
   wait(mutex); // 5. Lock 'read_count' again
   read_count--;
   if (read_count == 0)
       signal(rw_mutex); // 6. Last reader unlocks the shared data
   signal(mutex); // 7. Unlock 'read_count'
```

```
} while (true);
```

Writer Process (with Explanation):

```
do {
   wait(rw_mutex); // 1. Lock the shared data for writing
   // 2. Write or update the shared data
   signal(rw_mutex); // 3. Unlock the shared data
} while (true);
```

3. Dining Philosophers Problem

What is the Problem?

- Five philosophers sit around a circular table.
- Each one needs **two chopsticks** to eat (left and right).
- After eating, they put both chopsticks back and start thinking again.

Issue:

- If all philosophers pick up the left chopstick at the same time, no one will be able to pick up the right chopstick. This leads to **deadlock** (all waiting, none eating).

Semaphores Used:

```
semaphore chopstick[5]; // One semaphore for each chopstick, initialized
to 1
```

Philosopher Process (with Explanation):

```
do {
   wait(chopstick[i]); // 1. Pick up left chopstick
   wait(chopstick[(i+1)%5]); // 2. Pick up right chopstick
   // 3. Eat food
   signal(chopstick[i]); // 4. Put down left chopstick
   signal(chopstick[(i+1)%5]); // 5. Put down right chopstick
```

```
    // 6. Think
} while (true);
```

How to Prevent Deadlock:

1. **Limit to 4 philosophers** eating at once (one always waits).
2. **Pick both chopsticks only if both are available.**
3. **Odd-even rule:**
    - Odd philosophers pick left first, then right.
    - Even philosophers pick right first, then left.
    - This breaks the circular wait condition and avoids deadlock.

PROBLEMS WITH SEMAPHORES

Semaphores are powerful tools for process synchronization. However, if used incorrectly, they can lead to **subtle and hard-to-find bugs**, especially **timing errors**. These issues may occur **only under specific execution sequences**, making them difficult to detect and debug.

Common Mistakes and Their Consequences:

1. Reversing the Order of wait() and signal()

Wrong Code:

```
signal(mutex);
// critical section
wait(mutex);
```

Problem:

- This violates **mutual exclusion**.
- Multiple processes may enter the **critical section** at the same time.
- Bug only appears when multiple processes run **simultaneously**.

2. Replacing signal() with another wait()

Wrong Code:

```
wait(mutex);
// critical section
wait(mutex);
```

Problem:

- Leads to **deadlock**.
- The process blocks itself and never releases the resource.
- Other processes waiting for mutex will also be blocked.

3. Omitting wait() or signal() (or both)

Wrong Code Example:

```
// Missing wait(mutex)
// critical section
signal(mutex);
```

Problem:

- If wait() is **omitted**, mutual exclusion is **broken**.
- If signal() is **omitted**, it causes **deadlock**.
- If **both** are omitted, synchronization **fails completely**.

Semaphores are powerful but **tricky**. When programmers use semaphores, **they need to manually manage**:

- When to use `wait()` and `signal()`
- When to lock and unlock resources
- Avoid mistakes like forgetting to unlock, using the wrong semaphore, etc.

These small mistakes can cause **big problems**, like:

1. **Deadlocks**: Two processes wait for each other and never continue.
2. **Race conditions**: Two processes change the same data at the same time, causing errors.
3. **Difficult to debug**: If one `wait()` or `signal()` is wrongly placed, it's hard to find the bug.

4. **Hard to read**: Semaphore code can be confusing for humans.

What is a Critical Section?

A **critical section** is the part of the program where a process is **accessing shared resources** like a variable, file, or database.

We must ensure that **only one process is in the critical section** at a time to avoid data corruption.

CRITICAL REGION

What is a Critical Region?

A **critical region** is a **better and safer** version of a critical section.

- It is **automatically protected** so that only one process can use the shared resource at a time.
- Unlike semaphores, we don't need to manually write `wait()` or `signal()`.

Example:

Let's say 2 processes want to update a bank account. A critical region will ensure only one of them is allowed to make the change at any moment.

But even Critical Regions are limited. They don't support **waiting** or **blocking**. For example, if a resource is **not available**, a process should **wait**, but critical regions don't support that easily.

MONITORS: A Better Solution to Synchronization

Monitors were developed to **solve the problems of semaphores** and give a **clean and safe way** to handle synchronization.

What is a Monitor?

A **Monitor** is like a **class in a programming language** that:

- Holds **shared variables**
- Has **functions (methods)** that operate on those variables
- Only **one process can be inside a monitor** at any time (like a built-in lock)

- Provides **automatic mutual exclusion** (so no need for `wait()` and `signal()` every time)

Key Features of a Monitor:

- Monitor is a **high-level synchronization tool**.
- It gives us **mutual exclusion automatically** — we don't need to do it manually.
- Only **monitor's functions** can access the shared data.
- Only **one process can be active inside the monitor at a time**.

Monitor Syntax (Structure)

```
monitor MonitorName {
    // shared variables

    function function1(...) {
        // code
    }

    function function2(...) {
        // code
    }

    initialization_code() {
        // code to initialize variables
    }
}
```

Explanation:

- `monitor MonitorName` – This defines the monitor with a name.
- Inside the monitor, you can:
  - Declare **shared variables**.
  - Write **functions** that operate on the shared data.
  - Use `initialization_code()` to set up variables at the start.
- Remember: Only **one process can execute any of these functions at a time**.

Condition Variables in Monitors

```
condition x, y;
```

Explanation:

- A **condition variable** helps a process to **wait** inside the monitor **if a condition is not satisfied**, like if a resource is not available.
- Think of it like a queue: the process goes and waits in that queue until someone tells it to resume.

`x.wait()` and `x.signal()`:

```
x.wait(); // the process sleeps
x.signal(); // wakes up one process waiting on x
```

Explanation:

- `x.wait()` – The current process will pause and **release the monitor**, so someone else can use it.
- `x.signal()` – **Wakes up one process** that was waiting on condition `x`.

Dining Philosophers Problem Using Monitors

Let's say there are **5 philosophers** sitting around a table. Each one needs **two chopsticks** to eat – the left and the right one.

We want to make sure:

- No two philosophers next to each other eat at the same time.
- No deadlock happens.

Here's how a **monitor helps**.

Monitor Code:

```
monitor DiningPhilosophers {
    enum { THINKING, HUNGRY, EATING } state[5];
    condition self[5];
```

Explanation:

- We keep track of each philosopher's state using an array `state[5]`.
  - `THINKING`: Not hungry.
  - `HUNGRY`: Wants to eat.
  - `EATING`: Eating.
- `self[5]`: Each philosopher has their own condition variable (used to wait if forks are not available).

`pickup(int i)` Method:

```
void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        self[i].wait();
}
```

Explanation:

- Philosopher `i` becomes **HUNGRY** and wants to eat.
- We call `test(i)` to check if the left and right philosophers are not eating.
- If `test(i)` fails, the philosopher **waits** using `self[i].wait()`.

`putdown(int i)` Method:

```
void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);  // left neighbor
    test((i + 1) % 5);  // right neighbor
}
```

Explanation:

- Philosopher `i` is done eating and goes back to **thinking**.
- Now check if the **left** and **right** neighbors can start eating by calling `test()` on them.

`test(int i)` Method:

```
    void test(int i) {
        if (state[i] == HUNGRY &&
            state[(i + 4) % 5] != EATING &&
            state[(i + 1) % 5] != EATING) {
            state[i] = EATING;
            self[i].signal();
        }
    }
```

Explanation:

- This function checks if philosopher `i` is **HUNGRY**, and both **left and right** are **not eating**.
- If it's safe, the philosopher starts **EATING** and we signal them to continue.

Initialization Code:

```
    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Explanation:

- At the beginning, all 5 philosophers are **THINKING**.

Usage:

```
DiningPhilosophers.pickup(i); // Philosopher tries to eat
Eat
DiningPhilosophers.putdown(i); // Philosopher finishes eating
```

Explanation:

- These are the calls from outside the monitor.

- The philosopher first **picks up** chopsticks, then **eats**, then **puts down** chopsticks.

Internal Implementation of `wait()` and `signal()` (Using Semaphores)

how monitors implement condition variables using semaphores internally:

For each condition variable `x`:

```
semaphore x_sem = 0;
int x_count = 0;
```

x.wait():

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Explanation:

- Increase the count of processes waiting.
- If someone is already waiting to signal, let them go.
- Else release the monitor.
- Then wait on `x_sem` (puts the process to sleep).
- Once woken up, reduce the count.

x.signal():

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Explanation:

- If at least one process is waiting on `x`, wake it up using `x_sem`.
- Then the **current process waits** using `next`, giving monitor access to the one who was signaled.

Example: Resource Allocator Monitor

Suppose we have a shared resource, and we want only **one process to use it at a time**.

```
monitor ResourceAllocator {
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }

    void release() {
        busy = false;
        x.signal();
    }

    initialization_code() {
        busy = false;
    }
}
```

Explanation:

- `busy` keeps track of whether the resource is in use.
- `acquire()` checks if it's free. If not, it waits.
- `release()` marks the resource as free and signals the next waiting process.
- Initialization sets `busy = false`.