OPERATING SYSTEM

SEMESTER 5

UNIT - 2

HI COLLEGE

SYLLABUS

UNIT - 2

UNIT–II

No. of Hours: 12 **Chapter/Book Reference: TB1 [Chapters 3, 5, 6]; TB2 [Chapter 9]**

Processes: Process Concept, Process Scheduling, Operation on Processes
CPU Scheduling: Basic Concepts, Scheduling Criteria, Scheduling Algorithms
Process Synchronization: Background, The Critical-Section Problem, Semaphores solution to critical section problem
Process related commands in Linux: ps, top, pstree, nice, renice and system calls

INTRODUCTION

Process Concept

A process is a program in execution. It is a series of instructions that are being carried out by the CPU at a given point in time. Each process has its own unique memory space, which allows multiple processes to run concurrently without interfering with each other.Processes can be created in several ways, such as by executing a new program or by forking an existing process.

Process Scheduling,

1. First-come, first-served (FCFS):

- Each process arrives at the ready queue and is executed in the order in which it arrives.
- For example, let's say we have three processes, P1, P2, and P3, arriving at the ready queue at times 0, 2, and 4, respectively. If the CPU can only handle one process at a time, it will execute P1 from time 0 to completion, then P2 from time 2 to completion, and finally P3 from time 4 to completion.

2. Round-robin:

- Each process is assigned a fixed amount of CPU time (known as a time slice) and cycles through the processes in a round-robin fashion.
- For example, let's say we have three processes, P1, P2, and P3, with time slices of 5 milliseconds each. If the CPU can only handle one process at a time, it will execute P1 for 5 milliseconds at time 0, then switch to P2 for 5 milliseconds at time 5, then switch back to P1 for another 5 milliseconds at time 10, and so on. This continues until all processes have had a chance to execute for their full time slice.

3. Priority scheduling:

- Each process is assigned a priority based on its importance or urgency, and the highest-priority process is executed first.
- For example, let's say we have three processes, P1 (high priority), P2 (medium priority), and P3 (low priority), arriving at the ready queue at times 0, 2, and 4, respectively. If the CPU can only handle one process at a time, it will execute P1 from time 0 to completion (since it has the highest priority), then switch to P2 from time 2 to completion (since it has a higher priority than P3), and finally switch to P3 from time 4 to completion (since it has no higher-priority processes waiting).

4. Multilevel feedback queue (MLFQ):

- Each process starts in a low-priority queue with a large time slice and can be promoted to higher-priority queues as it demonstrates good CPU behavior.
- For example, let's say we have three processes, P1 (low priority), P2 (medium priority), and P3 (high priority), arriving at the ready queue at times 0, 2, and 4, respectively. If the CPU can only handle one process at a time and uses an MLFQ with two queues (low priority and high priority), it will execute P1 from time 0 to completion in the low-priority queue (since it has the lowest priority). After completing its execution in the low-priority queue, P1 will be moved to the high-priority queue with a smaller time slice (since it has demonstrated poor CPU behavior). Meanwhile, when P2 arrives at time 2 with medium priority, it will be executed immediately in the high-priority queue because there are no higher-priority processes waiting. Finally, when P3 arrives at time 4 with high priority, it will be executed immediately in the high-priority queue because there are no other processes waiting in that queue.

Operation on Processes

In computer science, an operating system manages multiple processes running on a computer system. These processes can interact with each other in various ways, and the operating system provides a set of operations to manipulate them. Here are some common operations on processes:

1. Creation: This operation creates a new process in the system. The operating system allocates resources such as memory, CPU time, and I/O devices to the new process.

2. Termination: This operation destroys a process and releases all the resources it has been using.

3. Suspension: This operation stops the execution of a process temporarily and places it in a waiting state. The operating system can suspend a process for various reasons, such as when it needs to perform an I/O operation or when it receives a signal from another process.

4. Resumption: This operation resumes the execution of a suspended process. The operating system selects a resumable process from the waiting queue based on its priority and CPU availability.

5. Wait: This operation causes a process to wait for an event to occur before continuing its execution. For example, a process may wait for an I/O operation to complete or for another process to release a resource it needs.

6. Signal: This operation sends a signal to another process to notify it of an event or condition. Signals can be used for various purposes, such as to terminate a process, to suspend it, or to wake it up from suspension.

7. Context Switch: This operation saves the current state of a running process (known as its context) and restores the state of another process (known as its context switch). Context switches are necessary when the operating system needs to switch between processes due to scheduling or resource allocation reasons.

CPU Scheduling: Basic Concepts

CPU scheduling is a critical function of an operating system that determines which process should be executed by the central processing unit (CPU) at any given time. The goal of CPU scheduling is to ensure that CPU resources are utilized efficiently and effectively while minimizing wait times for processes and maximizing system throughput.

Here are some basic concepts related to CPU scheduling:

1. Ready Queue: This is a list of processes that are waiting to be executed by the CPU. Processes enter the ready queue when they have been loaded into memory and are ready to run.

2. CPU Burst: This is the amount of time a process requires to execute a particular instruction sequence on the CPU. The CPU burst can vary widely from process to process, making it challenging to predict which process will require the CPU next.

3. Turnaround Time: This is the total time it takes for a process to complete execution, from the time it enters the system until it exits. Turnaround time includes both the CPU burst and any wait times for I/O operations or resource allocation.

4. Waiting Time: This is the amount of time a process spends waiting in the ready queue for the CPU to become available. Waiting time can significantly impact overall system performance, as processes that spend too much time waiting may not complete in a reasonable amount of time.

5. Response Time: This is the amount of time it takes for a process to respond after it has been submitted to the system. Response time includes both the CPU burst and any wait times for I/O operations or resource allocation, as well as any delays caused by other processes executing on the CPU.

6. Scheduling Algorithm: This is the set of rules used by the operating system to select which process should be executed next by the CPU. Scheduling algorithms can be classified into several categories, such as first-come, first-served (FCFS), round-robin, priority scheduling, and multilevel feedback queue (MLFQ). Each algorithm has its own trade-offs between CPU utilization, response time, and wait times for processes.

7. Context Switch: This is the process of saving a process's current state (known as its context) in memory and restoring another process's context when switching between processes due to scheduling or resource allocation reasons. Context switches can add overhead to CPU scheduling, as they require additional memory access and processing time.

Scheduling Criteria,

Scheduling criteria are the factors that operating systems use to select which process should be executed next by the CPU. These criteria help balance the competing demands of CPU utilization, response time, and wait times for processes. Here are some common scheduling criteria:

1. CPU Utilization: This criterion aims to maximize CPU usage by selecting processes that require the most CPU time. High CPU utilization can lead to shorter overall system response times and higher system throughput.

2. Response Time: This criterion aims to minimize the amount of time it takes for a process to respond after it has been submitted to the system. Short response times are critical for interactive processes, such as those used in graphical user interfaces (GUIs).

3. Waiting Time: This criterion aims to minimize the amount of time a process spends waiting in the ready queue for the CPU to become available. Reducing waiting times can significantly improve overall system performance, as processes that spend too much time waiting may not complete in a reasonable amount of time.

4. Priority: This criterion assigns different levels of priority to processes based on their importance or urgency. High-priority processes are executed before lower-priority processes, even if they have shorter CPU bursts or require less CPU time overall.

5. Fairness: This criterion aims to ensure that all processes receive a fair share of CPU resources over time, regardless of their initial priority or arrival time. Fairness is critical for ensuring that no single process monopolizes the CPU and prevents other processes from completing in a reasonable amount of time.

6. Deadline: This criterion is used for real-time processes that have strict timing constraints, such as those used in embedded systems or scientific computing applications. Deadline scheduling algorithms aim to ensure that all real-time processes complete within their specified deadlines, even in the presence of other processes with less stringent timing requirements.

SCHEDULING ALGORITHMS
[IMPORTANT][NUMERICAL]

CONCEPT OF PREEMPTIVE AND NON-PREEMPTIVE

- **Preemptive scheduling** allows the operating system to interrupt a running process (preempt it) if a higher-priority process becomes ready to run. This ensures that high-priority processes are executed before lower-priority processes, leading to better resource utilization. However, it results in increased overhead due to the need to save and restore process contexts during context switches.

- **Non-preemptive scheduling**, also known as cooperative scheduling, allows a running process to complete execution without interruption by the operating system, even if higher-priority processes become ready to run. This approach is simpler to implement and has lower overhead due to the lack of context switches, but it can result in longer wait times and response times for lower-priority processes that have to wait for higher-priority processes to complete before they can be executed.

FIRST-COME, FIRST-SERVED (FCFS):

Non-preemptive, This algorithm selects the process that arrives at the ready queue first to be executed next by the CPU. It's simple to implement, but can result in long wait times for processes with longer CPU bursts or arrival times.

Q. Consider the set of 5 processes whose arrival time and burst time are given below-

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

If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

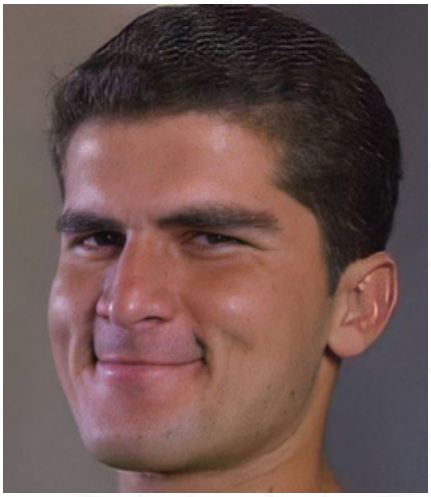
GANTT CHART

Gantt Chart

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

- Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit
- Average waiting time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit



ROUND-ROBIN

This algorithm assigns each process a fixed amount of CPU time (time quantum) and cycles through the ready queue executing each process for its assigned time quantum before moving on to the next process. If a process hasn't completed its CPU burst within its time quantum, it's placed back at the end of the ready queue and must wait for its turn again. Round-Robin ensures that all processes receive some CPU time, but can lead to frequent context switches and increased overhead due to saving and restoring process contexts.

Q. Consider the set of 5 processes whose arrival time and burst time are given below-

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

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turn around time.

P5, P1, P2, P5, P4, P1, P3, P2, P1

Gantt Chart

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

- Now,
- Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time =(8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

PRIORITY SCHEDULING

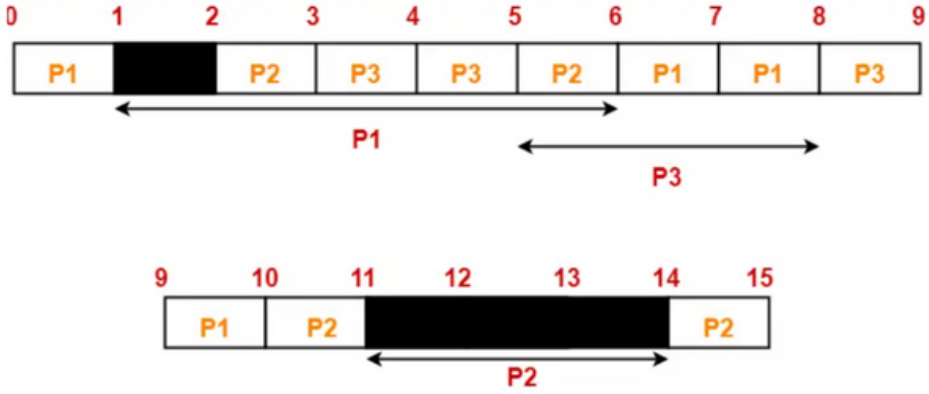
This algorithm assigns each process a priority level based on its importance or urgency, and executes the process with the highest priority first. High-priority processes are executed before lower-priority processes, even if they have shorter CPU bursts or require less CPU time overall. However, lower-priority processes may experience starv

Q. Consider the set of 3 processes whose arrival time and burst time are given below-

Process No.	Arrival Time	Priority	Burst Time		
			CPU Burst	I/O Burst	CPU Burst
P1	0	2	1	5	3
P2	2	3	3	3	1
P3	3	1	2	3	1

If the CPU scheduling policy is Priority Scheduling, calculate the average waiting time and average turn around time. (Lower number means higher priority)

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 10 10 – 0 = 10 10 – (1+3) = 6
- P2 15 15 – 2 = 13 13 – (3+1) = 9
- P3 9 9 – 3 = 6 6 – (2+1) = 3
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	10	10 – 0 = 10	10 – (1+3) = 6
P2	15	15 – 2 = 13	13 – (3+1) = 9
P3	9	9 – 3 = 6	6 – (2+1) = 3

Now,

Average Turn Around time = (10 + 13 + 6) / 3 = 29 / 3 = 9.67 units
Average waiting time = (6 + 9 + 3) / 3 = 18 / 3 = 6 units

SRTF

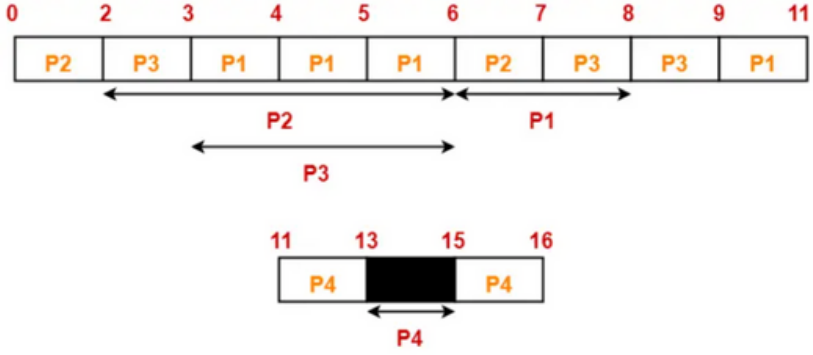
SRTF is a preemptive scheduling algorithm that selects the process with the shortest remaining CPU burst to execute next, but it's different from SJF in that it allows for preemption (interruption) of a running process if a higher-priority process becomes ready to run. SRTF is useful for minimizing wait times and response times for short jobs, while also ensuring that high-priority processes are executed before lower-priority processes. However, SRTF can lead to frequent context switches and increased overhead due to the need to save and restore process contexts.

Q. Consider the set of 4 processes whose arrival time and burst time are given below-

Process No.	Arrival Time	Burst Time		
		CPU Burst	I/O Burst	CPU Burst
P1	0	3	2	2
P2	0	2	4	1
P3	2	1	3	2
P4	5	2	2	1

If the CPU scheduling policy is Shortest Remaining Time First, calculate the average waiting time and average turn around time.

Gantt Chart-



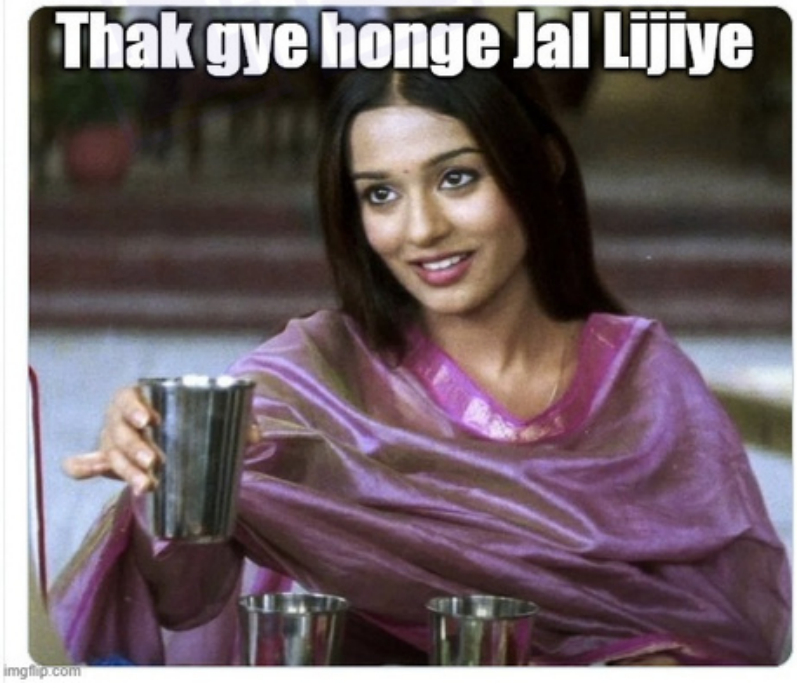
Now, we know-

Turn Around time = Exit time – Arrival time
Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	11	11 – 0 = 11	11 – (3+2) = 6
P2	7	7 – 0 = 7	7 – (2+1) = 4
P3	9	9 – 2 = 7	7 – (1+2) = 4
P4	16	16 – 5 = 11	11 – (2+1) = 8

Now,

Average Turn Around time = (11 + 7 + 7 + 11) / 4 = 36 / 4 = 9 units
Average waiting time = (6 + 4 + 4 + 8) / 4 = 22 / 5 = 4.4 units



PROCESS SYNCHRONISATION: BACKGROUND

Process synchronization is a critical concept in concurrent programming, particularly in multi-processor or multi-threaded systems. It refers to the mechanisms that ensure that multiple processes or threads access shared resources in a coordinated manner, preventing data races, deadlocks, and other synchronization issues.

In a concurrent system, multiple processes or threads can execute simultaneously, accessing shared resources such as memory, files, or network connections. Without proper synchronization, these processes can interfere with each other's access to shared resources, leading to unexpected behavior, incorrect results, or system failures.

Process synchronization aims to provide mechanisms that ensure that shared resources are accessed in a consistent and predictable manner by multiple processes or threads. These mechanisms include:

1. Mutual Exclusion: This mechanism ensures that only one process or thread can access a shared resource at a time. It prevents data races and ensures that the resource is not corrupted by multiple processes accessing it simultaneously.

2. Critical Section: A critical section is a portion of code that accesses a shared resource and must be executed serially by all processes or threads. This mechanism ensures that the resource is accessed consistently and prevents data races.

3. Synchronization Primitives: Synchronization primitives are programming constructs that provide mechanisms for process synchronization, such as semaphores, mutexes, and monitors. These primitives allow processes to wait for specific conditions to be met before accessing shared resources, preventing deadlocks and ensuring proper synchronization.

4. Deadlock Avoidance: Deadlock occurs when multiple processes are waiting for each other to release resources, resulting in a system-wide lockup. Deadlock avoidance mechanisms ensure that deadlocks are avoided by enforcing specific resource allocation policies and preventing circular waits between processes.

THE CRITICAL-SECTION PROBLEM

The Critical-Section Problem is a fundamental issue in concurrent programming that arises when multiple processes or threads share a common resource. It refers to the challenge of ensuring that only one process at a time can access the shared resource, preventing data races and ensuring consistency.

The critical section of a process is the portion of code that accesses the shared resource. To prevent multiple processes from accessing the resource simultaneously, mutual exclusion is required. Mutual exclusion ensures that only one process can enter its critical section at a time, preventing data races and ensuring consistency.

However, implementing mutual exclusion is not trivial, as it requires careful consideration of the order in which processes access the shared resource. If two processes try to enter their critical sections simultaneously, a race condition can occur, resulting in incorrect data or system failures.

To address this issue, several synchronization mechanisms have been developed, such as semaphores, mutexes, and monitors. These mechanisms provide mechanisms for mutual exclusion and ensure that only one process can access the shared resource at a time.

The Critical-Section Problem is significant because it highlights the challenges of concurrent programming and the need for careful synchronization mechanisms to ensure correctness and reliability in multi-processor or multi-threaded systems. It also underscores the importance of proper design and implementation of concurrent programs to prevent data races and ensure consistency in shared resources

SEMAPHORES SOLUTION TO CRITICAL SECTION PROBLEM

Semaphores are synchronization primitives that provide a mechanism for mutual exclusion and resource sharing in concurrent programming. They are used to solve the critical section problem by ensuring that only one process can access a shared resource at a time.

A semaphore is a variable that can take integer values, and it is used to control access to a shared resource. A semaphore is initialized with a non-negative integer value, which represents the number of processes that can access the shared resource simultaneously.

To access the shared resource, a process must first acquire the semaphore. If the semaphore value is greater than zero, the process can enter its critical section and access the shared resource. The semaphore value is then decremented to reflect that the resource is now being used by one process.

If multiple processes try to acquire the semaphore simultaneously, only one process will be able to enter its critical section at a time, as the semaphore value will be decremented to zero. The other processes will be blocked until the semaphore value becomes greater than zero again, indicating that the resource is available for use.

When a process leaves its critical section, it releases the semaphore by incrementing its value. This allows other processes that were blocked to acquire the semaphore and access the shared resource.

Semaphores are useful in solving the critical section problem because they provide a mechanism for mutual exclusion and ensure that only one process can access a shared resource at a time. They also support resource sharing by allowing multiple processes to access the shared resource simultaneously, as long as there are enough resources available.

FOR EXAMPLE: Imagine you have a shared resource, like a printer or a file, that multiple people want to use at the same time. To make sure only one person can use the resource at a time, you need a way to control access. That's where semaphores come in.

A semaphore is like a gatekeeper that controls access to the shared resource. It's like a traffic light that lets only one car go through an intersection at a time. The semaphore has a value, and when you want to use the shared resource, you ask the semaphore for permission by trying to "enter" its critical section.

If the semaphore's value is greater than zero, that means there are still resources available, and you can enter the critical section and use the shared resource. The semaphore's value is then decremented by one to reflect that there is now one less resource available.

If multiple people try to enter the critical section at the same time, only one person will be able to do so, because the semaphore's value will be decremented to zero. The other people will have to wait until the semaphore's value becomes greater than zero again, indicating that there are resources available.

When you're done using the shared resource, you release the semaphore by incrementing its value. This allows other people who were waiting to enter the critical section and use the resource.

PROCESS RELATED COMMANDS IN LINUX: PS, TOP, PSTREE, NICE, RENICE

In Linux, there are several commands that allow you to manage and monitor running processes. Here's a brief overview of some of the most commonly used ones:

1. ps (Process Status): This command displays a list of currently running processes. You can use various options with ps to customize the output, such as -u to show processes owned by a specific user, -x to show all processes (including background processes), and -f to display a full-format listing with additional details like process ID (PID), parent process ID (PPID), and CPU usage.

PROCESS RELATED COMMANDS IN LINUX: PS, TOP, PSTREE, NICE, RENICE

2. top: This command provides a dynamic, real-time view of system resource utilization, including CPU usage, memory usage, and process information. You can use the arrow keys to scroll through the list of processes, and the "k" key to kill a process by entering its PID.

3. pstree: This command displays a tree-like view of the current process hierarchy, showing parent-child relationships between processes. You can use various options with pstree to customize the output, such as -p to show process IDs, and -u to show usernames instead of UIDs.

4. nice: This command allows you to start a new process with a lower priority than normal, which can help prevent it from monopolizing system resources and causing other processes to slow down or freeze. You can use nice followed by the name of the command you want to run, followed by an optional priority value (-n) between -20 (highest priority) and 19 (lowest priority).

5. renice: This command allows you to change the nice value of an already running process, which can help prevent it from hogging resources or slowing down other processes. You can use renice followed by the PID of the process you want to modify, followed by an optional nice value (-n) between -20 (highest priority) and 19 (lowest priority).

SYSTEM CALLS

In computer systems, system calls are requests made by a running program to the operating system (OS) to perform certain tasks that are not directly supported by the processor or hardware. These tasks may include reading or writing data from a disk, sending or receiving data over a network, or managing system resources like memory and CPU time.

System calls are implemented as interrupts or traps in the processor, which cause the OS to temporarily suspend the execution of the program and switch to a special privileged mode called kernel mode. In kernel mode, the OS can access and manipulate system resources directly, without the restrictions imposed by user mode.

SYSTEM CALLS

The OS provides a standardized interface for making system calls, which is typically implemented as a set of functions or syscalls that can be invoked by the program using a specific assembly instruction or library function. The syscalls have unique numbers or names that identify their functionality, and they typically take one or more arguments that specify the parameters of the requested operation.

Some common examples of system calls include:

- read() and write(): Used for input/output (I/O) operations, such as reading data from a file or writing data to a network socket.
- open() and close(): Used for managing file descriptors, such as opening a file for reading or writing, and closing it when done.
- fork() and execve(): Used for process management, such as creating a new process (fork()), executing a new program (execve()), and waiting for a child process to complete (wait()).
- malloc() and free(): Used for memory management, such as allocating and deallocating dynamically allocated memory blocks.
- signal() and sigaction(): Used for handling signals (interrupts), such as registering a function to be called when a specific signal is received.
- waitpid() and kill(): Used for process control, such as waiting for a specific child process to complete (waitpid()), and sending a signal to another process (kill()).