



UNIT-II MATERIAL

SUBJECTCODE : CS4302

SUBJECT NAME: OPERATING SYSTEMS

DEPARTMENT: COMPUTER SCIENCE AND

ENGINEERING YEAR/SEM : II/III

ACADEMIC YEAR: 2025-2026

BATCH 2029

CS4302-OPERATING SYSTEMS
L T P C
3 0 2 4
**COURSE
OBJECTIVES**

- CO1: Explain the basic concepts and functions of operating system.
- CO2: Analyze various scheduling, deadlock prevention and deadlock avoidance algorithms.
- CO3: Compare and contrast various memory management schemes.
- CO4: Interpret the functionality of file systems and I/O systems.
- CO5: Compare iOS and Android Operating Systems.

UNIT II PROCESS MANAGEMENT
9

Processes - Process Concept - Process Scheduling - Operations on Processes - Inter-process Communication; CPU Scheduling - Scheduling criteria - Scheduling algorithms: Threads – Multithread Models – Multi core programming- Threading issues; Process Synchronization – The Critical-Section problem - Synchronization hardware – Semaphores – Mutex - Classical problems thread race Deadlock avoidance, Deadlock detection, Recovery from deadlock.

After Completion of the course, Students will be able:

S. No.	CO Statement	RBT
1.	Analyze various scheduling algorithms and process synchronization.	L4
2.	Explain deadlock prevention and avoidance algorithms	L2
3.	Compare and contrast various memory management schemes	L4
4.	Explain the functionality of file systems, I/O systems, and Virtualization	L2
5.	Compare iOS and Android Operating Systems	L4



S.no	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO10	PO11	PO12	PSO1	PSO2
CO1	3	2	-	-	-	-	-	-	1	-	-	3	-
CO2	3	3	2	2	-	-	-	-	1	-	-	3	1
CO3	3	3	2	2	-	-	-	-	1	-	-	3	1
CO4	3	2	-	-	-	-	-	-	1	-	-	3	
CO5	2	3	2	-	2	-	-	-	2	-	-	3	2

Course Code /

Title: UNIT

II

2.1 PROCESSES - PROCESS CONCEPT & PROCESS SCHEDULING

THE PROCESS:

- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.
- In computing, “a process is an instance of a computer program that is being executed. It contains the program code and its current activity”.
- A process is a program in execution. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc

Program vs Process:

A **process** is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

A process is an ‘active’ entity, as opposed to a program, which is considered to be a ‘passive’ entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).

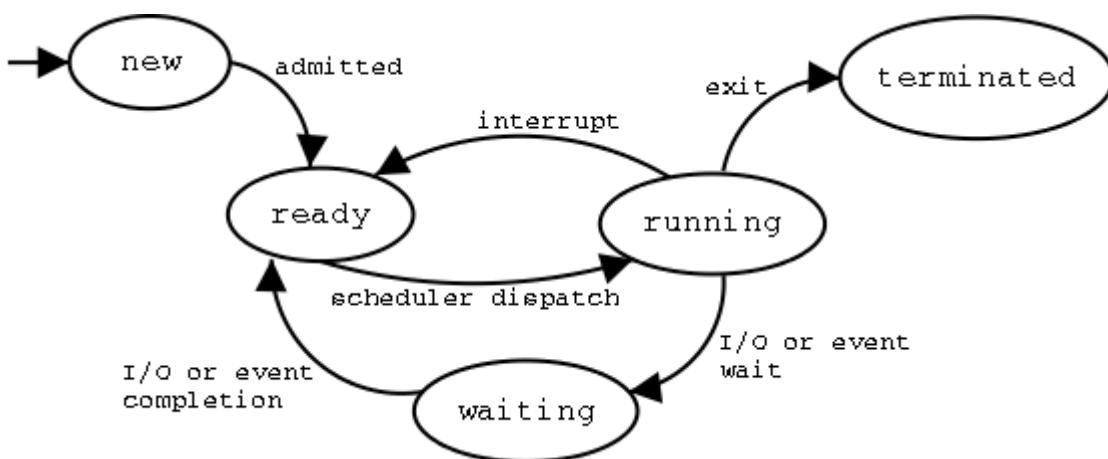
PROCESS CONCEPT:

What does a process look like in memory?

- Process memory is divided into four sections:
- The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
- The data section stores global and static variables, allocated and initialized prior to executing main.
- The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope .

PROCESS STATE:

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



1. New

A program which is going to be picked up by the OS into the main memory is called a new process.

2. Ready

Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.

The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state.

3. Running

One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system then we can have n processes running simultaneously.

4. Block or wait

From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process.

When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

5. Completion or termination

When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

6. Suspend ready

A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.

If the main memory is full and a higher priority process comes for the execution then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.

7. Suspend wait

Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.

The process has finished execution. These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting,

PROCESS CONTROL BLOCK (PCB):

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

Structure of the Process Control Block

The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given

Diagram –



Process Control Block (PCB)

➤ The following are the data items –

❖ **Process State**

This specifies the process state i.e. new, ready, running, waiting or terminated.

❖ **Process Number**

This shows the number of the particular process.

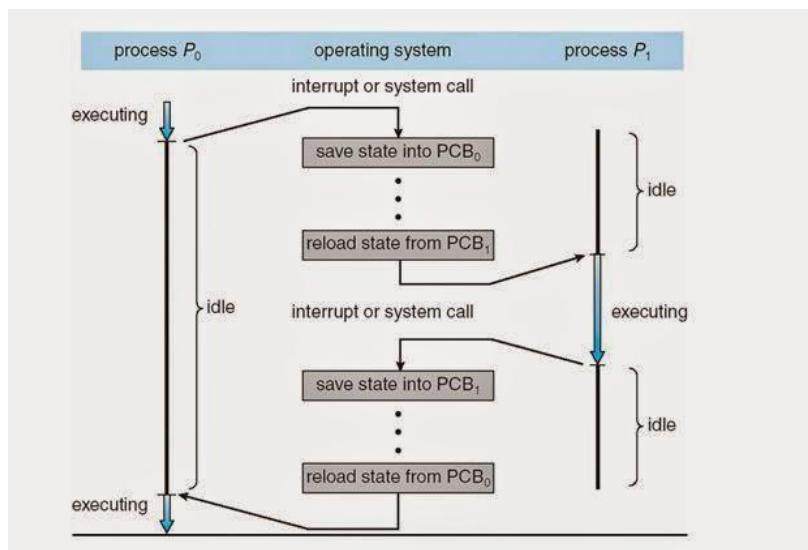
❖ **Program Counter**

This contains the address of the next instruction that needs to be executed in the process.

❖ **Registers**

This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

CPU SWITCH FROM PROCESS TO PROCESS



❖ List of Open Files

These are the different files that are associated with the process

❖ CPU Scheduling Information

The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

❖ Memory Management Information

The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

❖ I/O Status Information

This information includes the list of I/O devices used by the process, the list of files etc.

❖ Accounting information

The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

❖ Location of the Process Control Block

The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

THREADS:

THREADS-OVERVIEW

A thread is a flow of execution through the process code. It is a basic unit of CPU utilization, it comprises a thread ID, own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

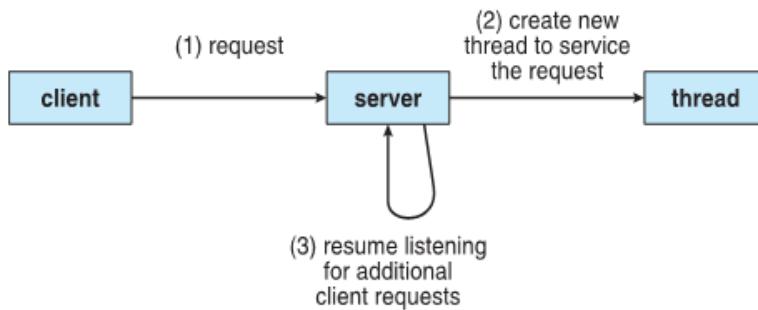
It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time

MOTIVATION:

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture:



Benefits:

The benefits of multithreaded programming can be broken down into four major categories:

1. Responsiveness:

Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

2. Resource sharing:

Processes can only share resources through techniques such as shared memory and message passing.

3. Economy:

Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context switch threads.

4. Scalability:

The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multicore Programming:

Multicore Programming Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems.

A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore or multiprocessor** systems.

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

Consider an application with four threads

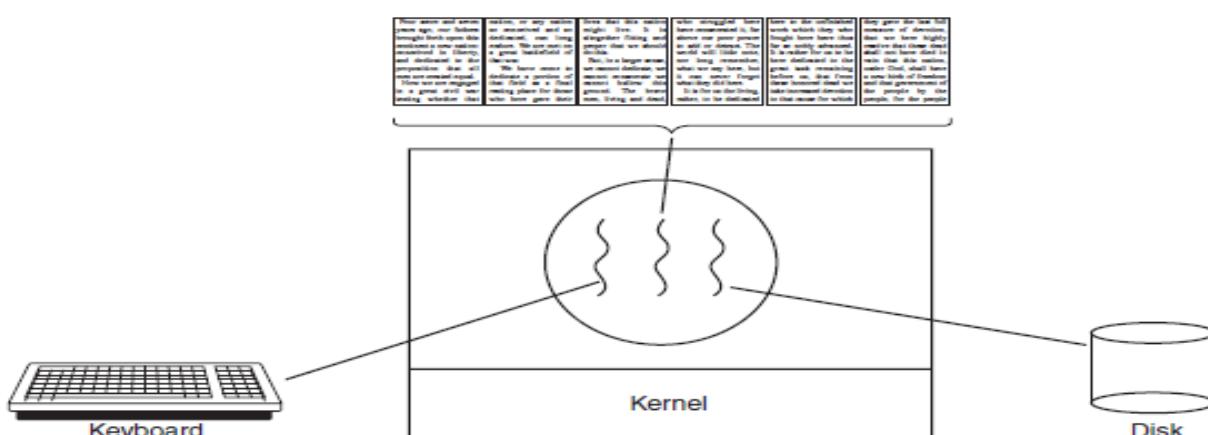


Figure 2-7. A word processor with three threads.

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, Concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core .

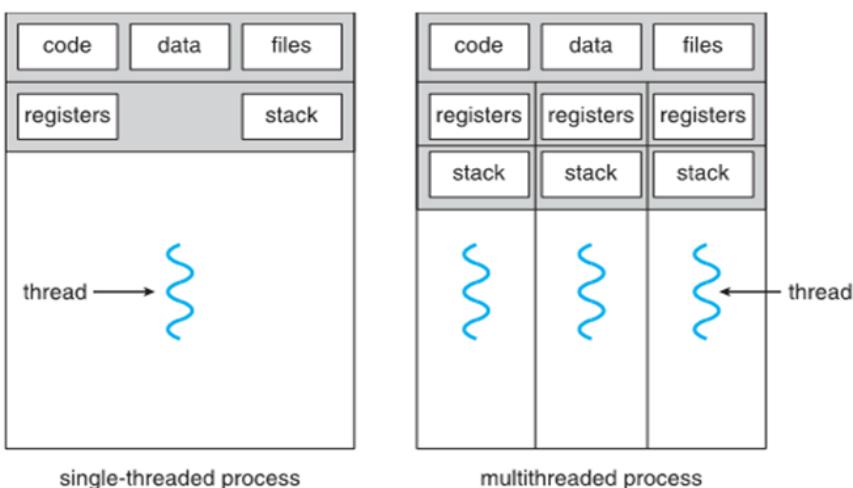
A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.

Types of parallelism

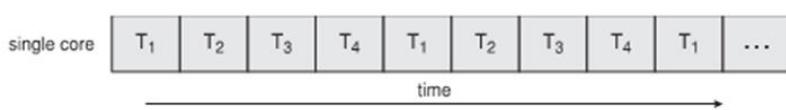
- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
- **Task parallelism** – distributing threads across cores, each thread performing unique operation

Concurrency vs. Parallelism

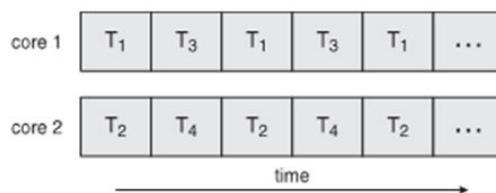
Single and Multithreaded Processes



Concurrent execution on single-core system:



Parallelism on a multi-core system:



Amdahl's Law

Identifies performance gains from adding additional cores to an application that has both serial and parallel components

S is serial portion

N processing cores

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

As N approaches infinity, speedup approaches $1 / S$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

User Threads and Kernel Threads

User threads - management done by user-level threads library

Three primary thread libraries:

POSIX Pthreads

Windows threads

Java threads

Kernel threads - Supported by the Kernel

Examples – virtually all general purpose operating systems, including:

Windows

Solaris

Linux

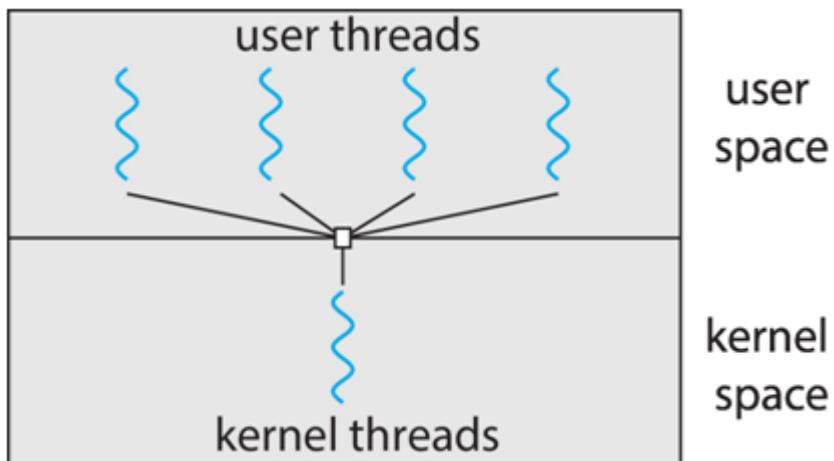
Tru64 UNIX

Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One



- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multi-core system because only one may be in kernel at a time
- Few systems currently use this model

Examples:

Solaris Green Threads

GNU Portable Threads

One-to-One

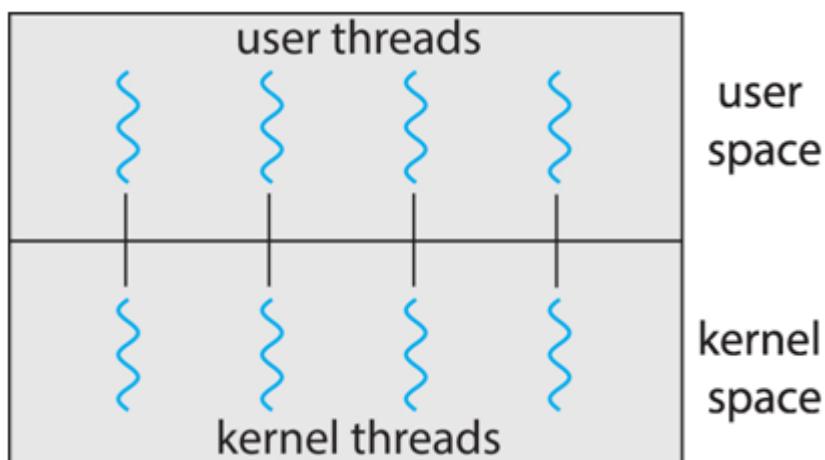
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

Examples

Windows

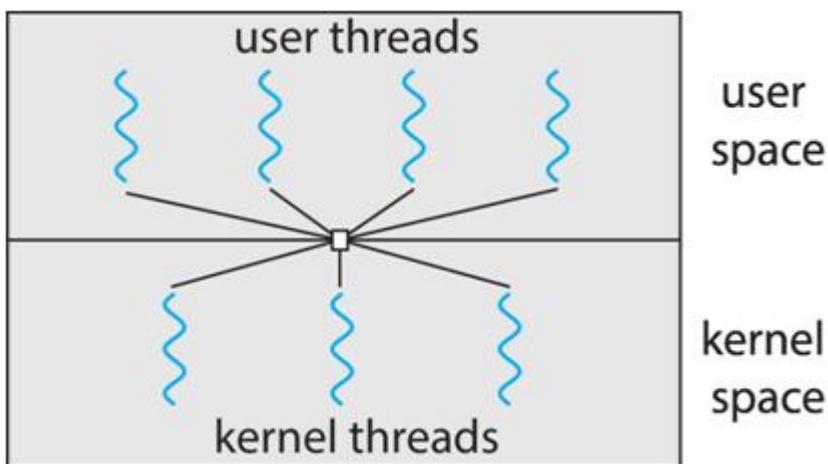
Linux

Solaris 9 and later



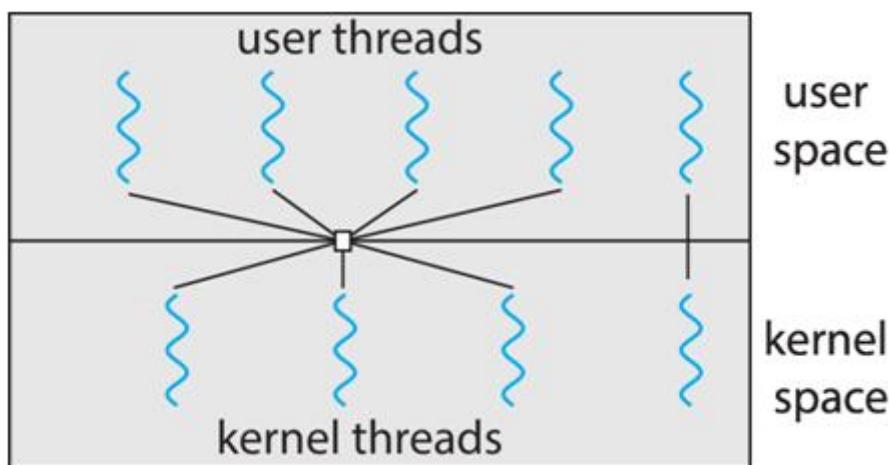
Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model**.



Thread Libraries

Thread library provides programmer with API for creating and managing threads

Two primary ways of implementing

- Library entirely in user space
- Kernel-level library supported by the OS

Three main thread libraries:

- POSIX Pthreads

Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.

- Windows

The Windows thread library is a kernel-level library available on Windows systems.

- Java

The Java thread API allows threads to be created and managed directly in Java programs.

Thread Issues

Some of the issues to consider in designing multithreaded programs

1. Semantics of fork() and exec() system calls
2. Signal handling
 - a. Synchronous and asynchronous
3. Thread cancellation of target thread Asynchronous or deferred
4. Thread-local storage
5. Scheduler Activations

1.Semantics of fork() and exec() system calls

The semantics of the fork() and exec() system calls change in a multithreaded program.

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

2.Signal handling

A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled.

All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

A signal may be handled by one of two possible handlers:

1. A default signal handler

2. A user-defined signal handler

Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-defined signal handler that is called to handle the signal. Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

3. Thread Cancellation

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.

A thread that is to be canceled is often referred to as the **target thread**.

❖ Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation: One thread immediately terminates the target thread.

2. Deferred cancellation: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

If thread has cancellation disabled, cancellation remains pending until thread enables it

Default type is deferred

Cancellation only occurs when thread reaches **cancellation point**

I.e. **pthread_testcancel()**

Then **cleanup handler** is invoked

On Linux systems, thread cancellation is handled through signals

4. Thread-Local Storage

Thread-local storage (TLS) allows each thread to have its own copy of data

Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Different from local variables

Local variables visible only during single function invocation

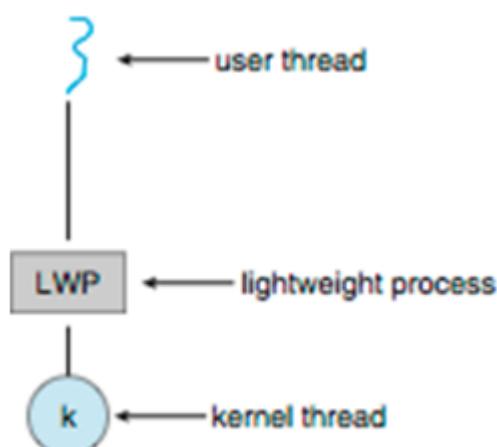
TLS visible across function invocations

Similar to static data

TLS is unique to each thread

5. Scheduler Activations

Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application



Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)

- Appears to be a virtual processor on which process can schedule user thread to run
- Each LWP attached to kernel thread
- How many LWPs to create?

Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library

This communication allows an application to maintain the correct number of kernel threads

A process is a program that performs a single thread of execution. This single thread of control allows the process to perform only one task at a time.

☞ Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

☞ This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.

☞ On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads

PROCESS SCHEDULING:

1. *What is Process Scheduling?*

- **Process Scheduling** is an OS task that schedules processes of different states like ready, waiting, and running.
- Process scheduling allows OS to allocate a time interval of CPU execution for each process. Another important reason for using a process scheduling system is that it keeps the CPU busy all the time. This allows you to get the minimum response time for programs.

2. **Process Scheduling Queues?**

Process Scheduling Queues help you to maintain a distinct queue for each and every process states and PCBs. All the processes of the same execution state are placed in the same queue. Therefore, whenever the state of a process is modified, its PCB needs to be unlinked from its existing queue, which moves back to the new state queue.

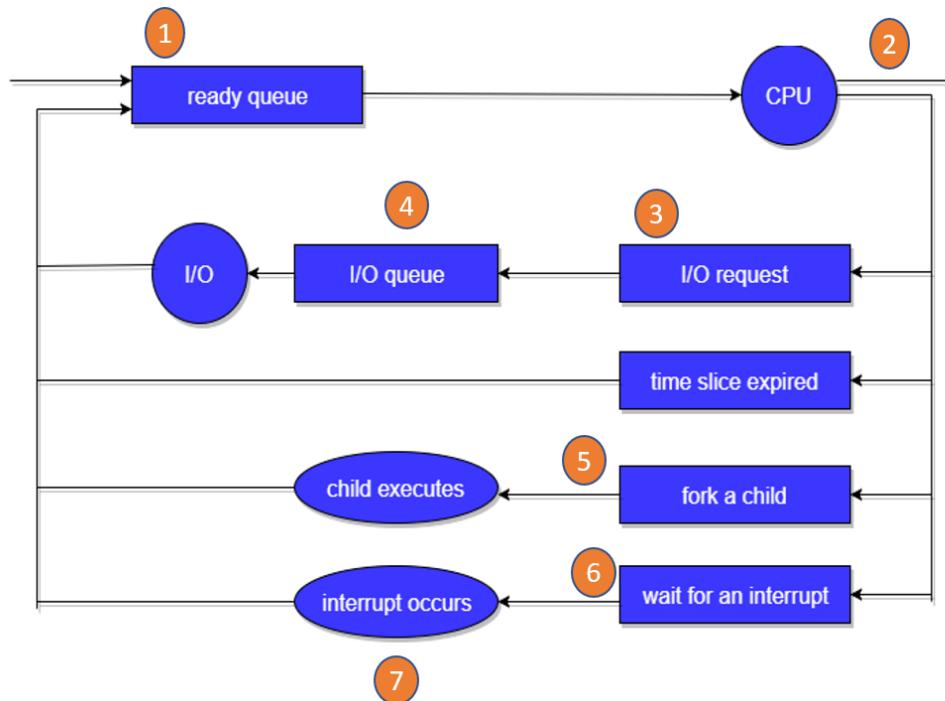
3. Why do we need Scheduling?

In Multiprogramming, if the long term scheduler picks more I/O bound processes then most of the time, the CPU remains idle. The task of Operating system is to optimize the utilization of resources.

If most of the running processes change their state from running to waiting then there may always be a possibility of deadlock in the system. Hence to reduce this overhead, the OS needs to schedule the jobs to get the optimal utilization of CPU and to avoid the possibility to deadlock.

Three types of operating system queues are:

1. **Job queue** – It helps you to store all the processes in the system.
2. **Ready queue** – This type of queue helps you to set every process residing in the main memory, which is ready and waiting to execute.
3. **Device queues** – It is a process that is blocked because of the absence of an I/O device.



In the above-given Diagram,

- Rectangle represents a queue.
- Circle denotes the resource
- Arrow indicates the flow of the process.

1. Every new process first put in the Ready queue .It waits in the ready queue until it is finally processed for execution. Here, the new process is put in the ready queue and wait until it is selected for execution or it is dispatched.
2. One of the processes is allocated the CPU and it is executing
3. The process should issue an I/O request
4. Then, it should be placed in the I/O queue.
5. The process should create a new subprocess
6. The process should be waiting for its termination.
7. It should remove forcefully from the CPU, as a result interrupt. Once interrupt is completed, it should be sent back to ready queue.

Two State Process Model

Two-state process models are:

- Running
- Not Running

RUNNING

In the Operating system, whenever a new process is built, it is entered into the system, which should be running.

Not Running

The process that are not running are kept in a queue, which is waiting for their turn to execute. Each entry in the queue is a point to a specific process.

Scheduling Objectives

Here, are important objectives of Process scheduling

- Maximize the number of interactive users within acceptable response times.
- Achieve a balance between response and utilization.
- Avoid indefinite postponement and enforce priorities.
- It also should give reference to the processes holding the key resources.

Schedulers

A scheduler is a type of system software that allows you to handle process scheduling.

There are mainly three types of Process Schedulers:

1. Long Term
2. Short Term
3. Medium Term

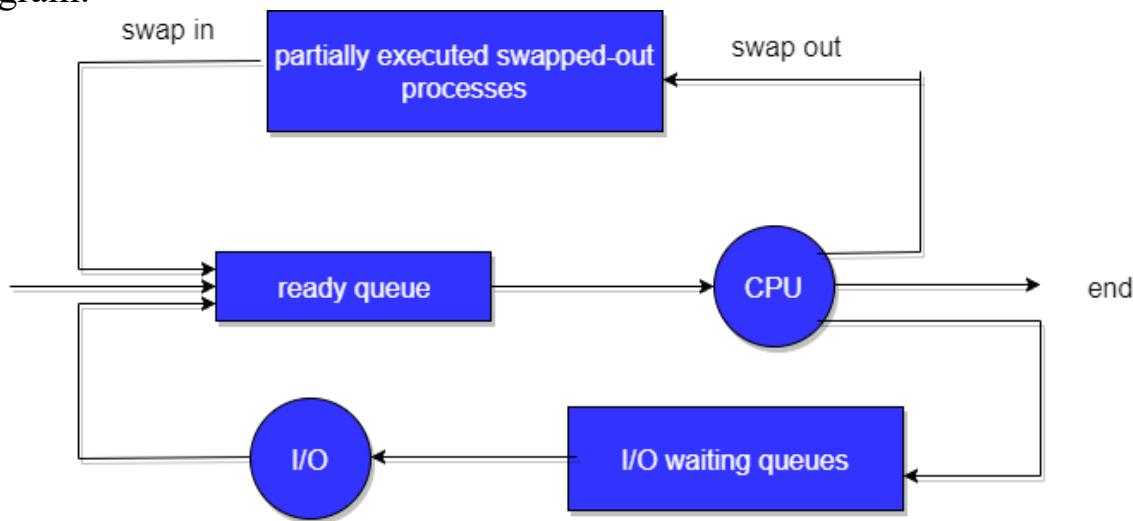
Long Term Scheduler

Long term scheduler is also known as a **job scheduler**. This scheduler regulates the program and select process from the queue and loads them into memory for execution. It also regulates the degree of multi-programing. However, the main goal of this type of scheduler is to offer a balanced mix of jobs, like Processor, I/O jobs., that allows managing multiprogramming.

Medium Term Scheduler

This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution van be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium term scheduler. Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. This complete process is descripted in the below

diagram:



Addition of Medium-term scheduling to the queueing diagram.

Short Term Scheduler

Short term scheduling is also known as **CPU scheduler**. The main goal of this scheduler is to boost the system performance according to set criteria. This helps you to select from a group of processes that are ready to execute and allocates CPU to one of them. The dispatcher gives control of the CPU to the process selected by the short term scheduler.

Multicore programming

Multicore programming involves designing and implementing algorithms that utilize multiple processing cores within a single processor or across multiple processors to achieve parallel execution and improved performance. This approach enables the division of a computational task into smaller, independent subtasks that can be executed concurrently on different cores, thereby significantly reducing overall processing time, especially for computationally intensive applications.

Definition:

Multicore programming in an operating system refers to the techniques and strategies used to execute multiple tasks concurrently on a computer system that has multiple processing cores within a single chip or on multiple chips. This approach leverages the parallel processing capabilities of multicore processors to enhance system performance and responsiveness by distributing workloads across different cores.

Example (Conceptual):

Imagine you need to process a large image. A naive approach would be to process it pixel by pixel, one at a time. However, on a multicore processor, you can split the image into multiple regions and assign each region to a different core. Each core can then process its assigned region simultaneously, resulting in much faster overall processing. Multicore programming helps you create concurrent systems for deployment on multicore processors and multiprocessor systems

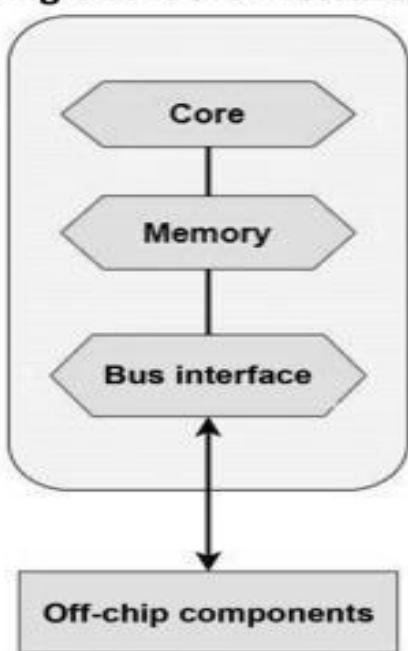
A *multicore processor system* is a single processor with multiple execution cores on one chip. By contrast, a *multiprocessor system* has multiple processors on the motherboard or chip. A multiprocessor system might include a Field-Programmable Gate Array (FPGA). An FPGA is an integrated circuit containing an array of programmable logic blocks and a hierarchy of reconfigurable interconnects. A *processing node* processes input data to produce outputs. It can be a processor in a multicore or multiprocessor system, or an FPGA.

Multicore programming is commonly used in signal processing and control systems for plants. In signal processing, you can have a concurrent system that processes multiple frames in parallel. In plant-control systems, the controller and the plant can execute as two separate tasks. Using multicore programming helps to split your system into multiple parallel tasks that run simultaneously. These are processors with multiple independent processing units (cores) on a single integrated circuit (IC) chip. Each core can execute instructions independently, allowing for parallel processing of tasks. Multicore

Programming:

This involves designing and implementing software that can utilize these multiple cores to achieve parallelism. The goal is to break down complex tasks into smaller parts that can be executed concurrently on different cores, thereby speeding up overall processing.

Single core architecture



Multi-core architecture

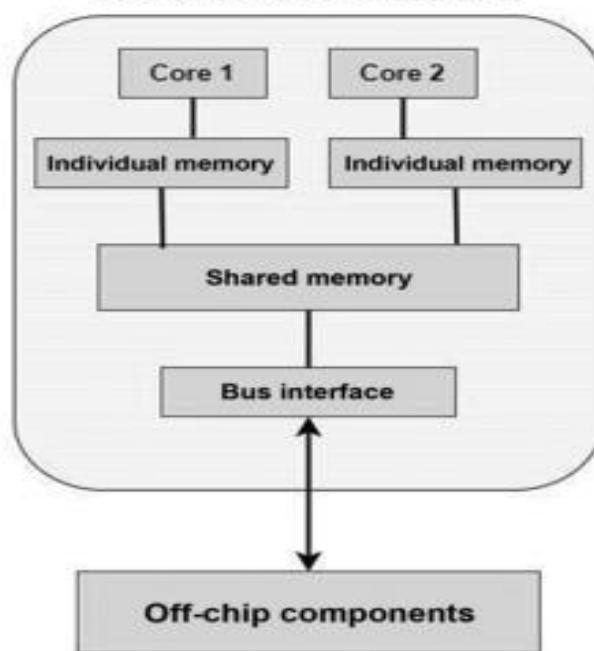


Fig. 1 Multicore processor architecture

Key Concepts:

- **Parallelism:**

The ability to execute multiple tasks simultaneously. Multicore programming leverages parallelism to reduce execution time.

System Partitioning for Parallelism

Partitioning methods help you to designate areas of your system for concurrent execution. Partitioning allows you to create tasks independently of the specifics of the target system on which the application is deployed.

Consider this system. F1–F6 are functions of the system that can be executed independently. An arrow between two functions indicates a data dependency. For example, the execution of F5 has a data dependency on F3.

Execution of these functions is assigned to the different processor nodes in the target system. The gray arrows indicate assignment of the functions to be deployed on the CPU or the FPGA. The CPU scheduler determines when individual tasks run. The CPU and FPGA communicate via a common communication bus.

The figure shows one possible configuration for partitioning. In general, you test different configurations and iteratively improve until you get the optimal distribution of tasks for your application.

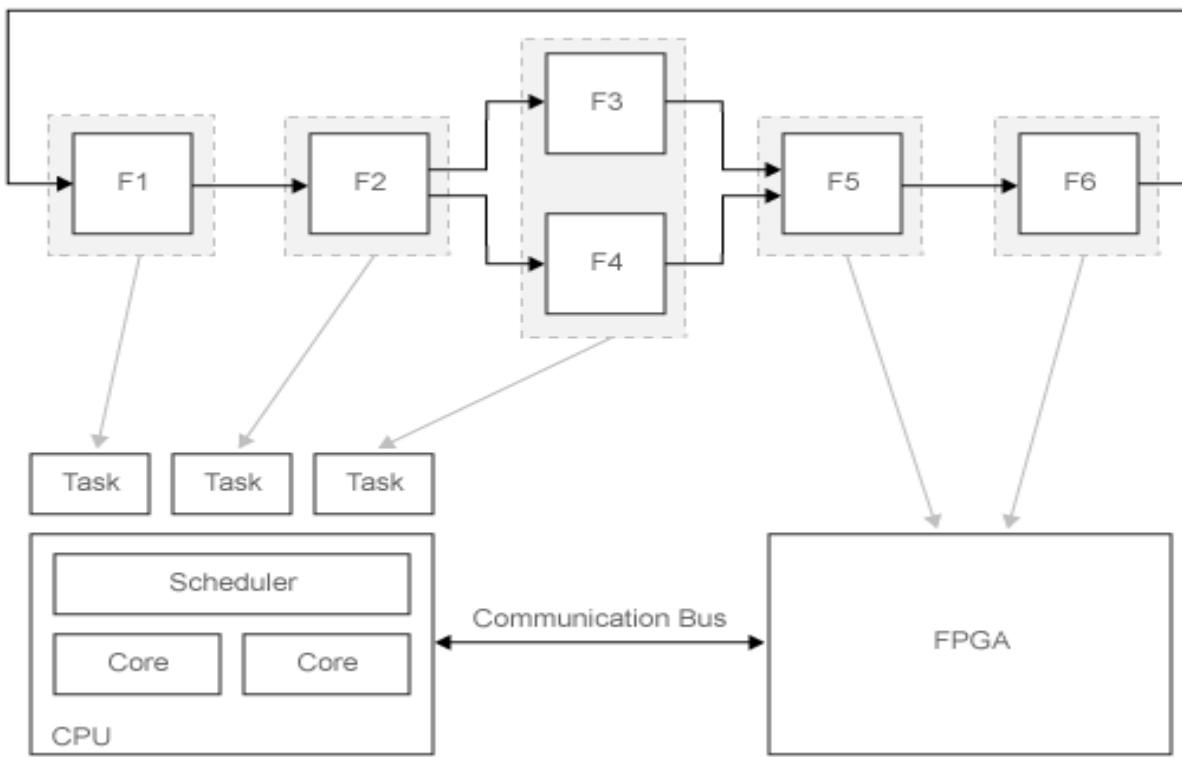


Fig. 2 Multicore processor partitioning

Types of Parallelism

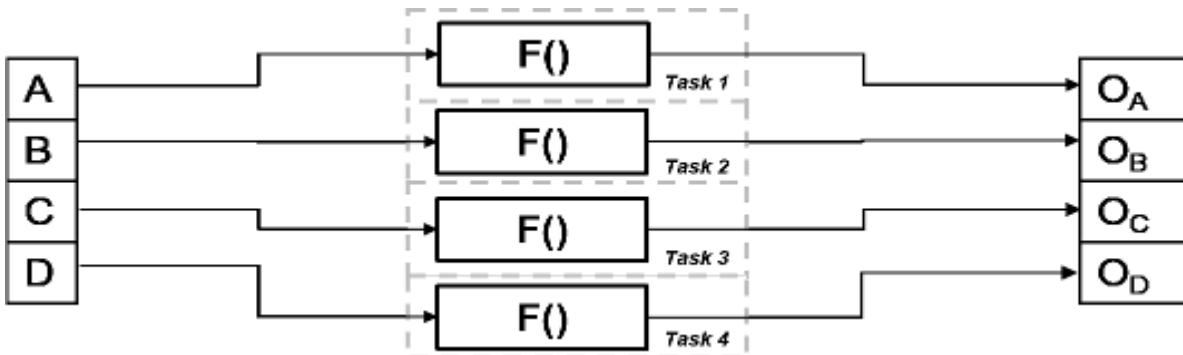
The concept of multicore programming is to have multiple system tasks executing in parallel. Types of parallelism include:

- Data parallelism
- Task parallelism
- Pipelining

Data Parallelism

Data parallelism involves processing multiple pieces of data independently in parallel. The processor performs the same operation on each piece of data. You achieve parallelism by feeding the data in parallel.

The figure shows the timing diagram for this parallelism. The input is divided into four chunks, A, B, C, and D. The same operation $F()$ is applied to each of these pieces and the output is O_A , O_B , O_C , and O_D respectively. All four tasks are identical, and they run in parallel



**Fig. 3 Multicore processor parallel processing
Task Parallelism**

In contrast to data parallelism, task parallelism doesn't split up the input data. Instead, it achieves parallelism by splitting up an application into multiple tasks. Task parallelism involves distributing tasks within an application across multiple processing nodes. Some tasks can have data dependency on others, so all tasks do not run at exactly the same time.

Consider the **below figure** a system that involves four functions. Functions F2a() and F2b() are in parallel, that is, they can run simultaneously. In task parallelism, you can divide your computation into two tasks. Function F2b() runs on a separate processing node after it gets data Out1 from Task 1, and it outputs back to F3() in Task 1.

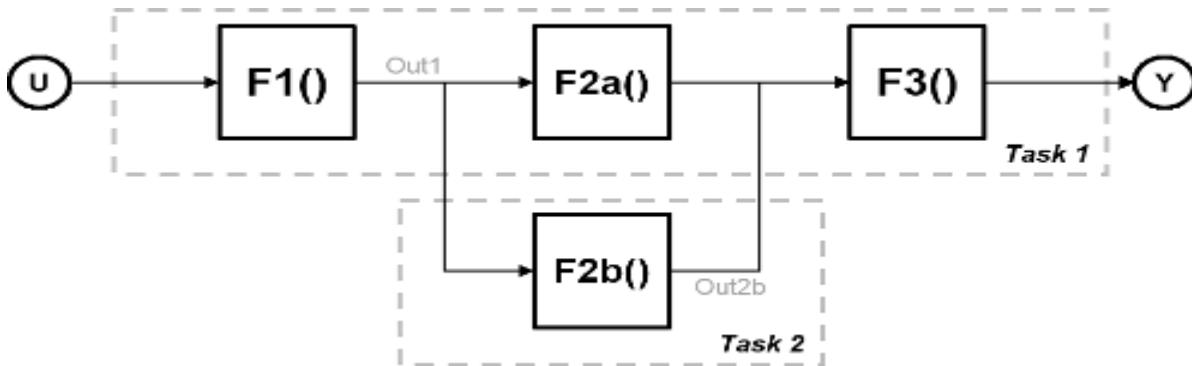


Fig. 4 Multicore processor task parallelism

Model Pipeline Execution (Pipelining)

Use model pipeline execution, or pipelining, to work around the problem of task parallelism, where threads do not run completely in parallel. This approach involves modifying your system model to introduce delays between tasks where there is a data dependency.

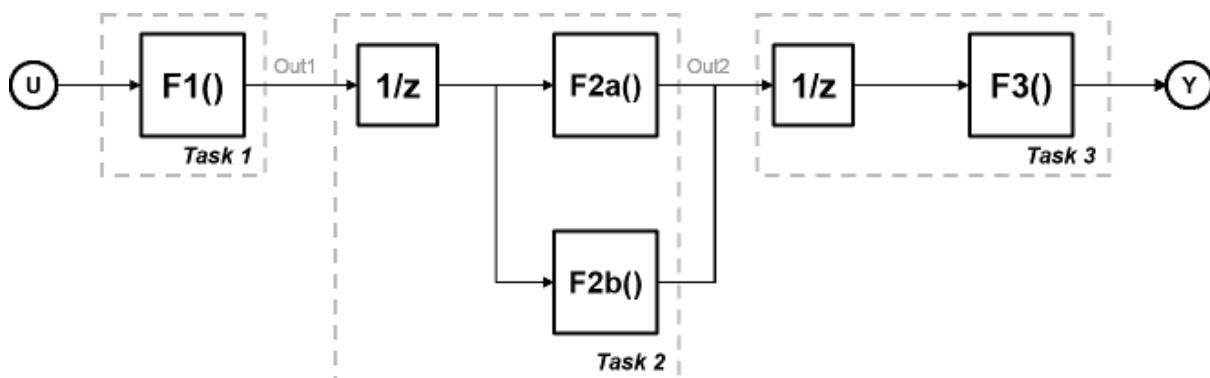


Fig. 5 Multicore processor pipeline processing

In this figure, the system is divided into three tasks to run on three different processing nodes, with delays introduced between functions. At each time step, each task takes in the value from the previous time step by way of the delay.

Concurrency:

While often used interchangeably, concurrency refers to the ability to handle multiple tasks at the same time, potentially by rapidly switching between them on a single core (time-slicing). Multicore programming often involves true parallelism, where tasks execute truly simultaneously.

Threads:

Lightweight units of execution that can be managed by the operating system and run concurrently. Multicore programming often uses threads to distribute tasks across cores.

Shared Memory:

A memory space accessible by all cores. Multicore programming often uses shared memory for communication and data exchange between threads.

Synchronization:

Mechanisms (like mutexes, semaphores) to manage access to shared resources and prevent data races or inconsistencies when multiple threads access the same data.

Example Scenario:

Consider an image processing application that needs to apply a filter (e.g., Gaussian blur) and then perform edge detection (e.g., using the Sobel operator) on a large image.

Sequential Approach:

- The application would process the image sequentially: first apply the blur, then perform edge detection.
- This approach limits performance to the speed of a single core.

Multicore Approach:

1. Identify Parallelizable Tasks:

The blur and edge detection operations can be done independently on different parts of the image.

2. Thread Creation:

Divide the image into smaller blocks (e.g., rows or tiles). Create a thread for each block.

3. Parallel Execution:

Each thread applies the blur and edge detection to its assigned block. The threads execute concurrently on different cores.

4. Result Combination:

Once all threads complete, the processed blocks are combined to form the final image.

5. Synchronization:

If necessary, synchronization mechanisms can be used to ensure that the threads don't interfere with each other when accessing shared data or resources.

❖ Advantages of Multicore Programming:

- **Improved Performance:** Faster execution of computationally intensive tasks.
- **Increased Throughput:** Ability to handle more tasks concurrently.
- **Better Responsiveness:** UI elements can remain responsive while other tasks are running in the background.
- **Resource Optimization:** Efficiently utilize the available processing cores.

SCENARIO-BASED QUESTIONS WITH ANSWERS:

1. Scenario:

You need to process a large dataset (e.g., an image) in parallel. Each core can work on a separate chunk of the data. How would you divide the work and synchronize access to shared resources?

Problem Context:

In multicore systems:

- **Each core runs threads in parallel.**
- **If threads work on shared data (e.g., writing processed results), it can cause race conditions.**

- Need to divide work efficiently and synchronize shared resource access.

Step-by-Step Solution

a. Divide Work – Data Partitioning (Workload Distribution)

Split the large dataset (e.g., an image) into **equal-sized chunks** and assign each to a different thread/core.

For example, if the image has 1000 rows and 4 cores:

Thread 0 → rows 0–249
Thread 1 → rows 250–499
Thread 2 → rows 500–749
Thread 3 → rows 750–999

This is called **static partitioning**. You can also use **dynamic scheduling** for load balancing if chunks vary in complexity.

b. Thread Creation and Execution

Example in C++ using std::thread:

```
std::vector<std::thread> threads;  
for (int i = 0; i < num_threads; ++i) {  
    threads.emplace_back(process_chunk, start_row[i], end_row[i]);  
}  
for (auto& t : threads) t.join();
```

c. Synchronization of Shared Resources

If threads only read their chunk and write to **independent output locations**, no synchronization is needed. But if threads **access shared resources** (e.g., global result matrix, counters, logs), you must protect these with synchronization:

Use Mutexes or Locks:

```
std::mutex result_lock;
```

```
void process_chunk(int start, int end) {  
    for (int i = start; i < end; ++i) {  
        // process data
```

```

std::lock_guard<std::mutex> lock(result_lock);
// safely write to shared result
}

```

} Use Atomic Variables for Counters:

```

std::atomic<int> processed_chunks = 0;
processed_chunks++;

```

d. Barrier Synchronization (if needed)

If all threads must finish before a final step (e.g., combining results), use **barrier synchronization**:

- In POSIX: use pthread_barrier_t
- In C++20: use std::barrier

```
std::barrier sync_barrier(num_threads);
```

```

void process_chunk(...) {
    // process data
    sync_barrier.arrive_and_wait(); // wait for all threads
    // final combine step
}

```

Summary:

Task	OS-based Approach
Divide work	Static/dynamic chunking
Run on multiple cores	Use threads (pthreads, std::thread, etc.)
Synchronize shared resources	Mutexes, atomic operations, barriers
Avoid race conditions	Protect critical sections
Ensure completion	Use thread join or barriers

2. Scenario: A video streaming platform needs to transcode (convert) uploaded videos into multiple formats (e.g., 1080p, 720p, 480p) for compatibility with various devices. To speed up the process, the platform uses a multicore server where each core can handle a part of the video or a specific resolution.

Problem:

As multiple threads work on encoding different parts or formats of the same video simultaneously, they also need to update a shared job status dashboard and write to a shared output directory.

Question:

How would you design the multicore program to:

1. Efficiently divide the transcoding work across multiple cores?
2. Ensure thread safety when updating shared resources like the dashboard or output directory?
3. Prevent data corruption and ensure correct completion of the entire transcoding job?

Solution:

a. Efficient Workload Division (Parallelism Strategy)

To utilize all cores effectively, the video transcoding task can be divided using one of the following strategies:

- **Task Parallelism:** Assign each thread (core) to a specific output format (e.g., one thread for 1080p, another for 720p). This is suitable when each format takes a different amount of time.
- **Data Parallelism:** Split the video into segments (chunks of frames) and assign each segment to a thread, where all threads transcode the same format in parallel. This is efficient when working with large video files.

Work is divided using **static** partitioning for simplicity or **dynamic task queues** for load balancing.

b. Thread Creation and Execution

Multiple threads are created to execute transcoding jobs. In C++ using std::thread, or in Java using ExecutorService, the threads are responsible for processing their assigned video chunks or formats.

Example (pseudocode in C++):

```
std::thread t1(transcode_1080p, video_chunk1);
std::thread t2(transcode_720p, video_chunk2);
std::thread t3(transcode_480p, video_chunk3);
...
t1.join();
t2.join();
t3.join();
```

c. Synchronization of Shared Resources

To update shared resources like a job status dashboard or an output directory, proper synchronization must be used:

- **Mutex Locks:** Protect shared structures (e.g., status updates) using mutual exclusion to avoid race conditions.

```
std::mutex status_mutex;

void update_status(std::string status) {
    std::lock_guard<std::mutex> lock(status_mutex);
    // safely update dashboard
}
```

- **Atomic Variables:** For simple shared counters (e.g., number of completed tasks), use atomic types.

```
std::atomic<int> completed_tasks = 0;
completed_tasks++;
```

- **Barriers:** Ensure all threads complete their transcoding before merging results or notifying the user.

```
std::barrier sync_point(num_threads);
```

```
void thread_task(...) {
    transcode(...);
    sync_point.arrive_and_wait(); // wait for all threads
    if (is_last_thread)
        merge_output_files();
}
```

d. Avoiding Data Corruption and Ensuring Consistency

- **Output File Management:** Assign unique file names or directories per thread to prevent simultaneous writes to the same file.
- **Thread-safe Logging:** If all threads log status to a shared log file, use mutexes or concurrent logging queues.
- **Validation and Merging:** Once all threads finish, a final step should verify and merge outputs if needed.

Conclusion:

By combining efficient workload partitioning with proper synchronization mechanisms (mutexes, atomic operations, and barriers), the multicore video transcoding system ensures:

- Optimal use of CPU cores,
- Thread-safe access to shared resources,
- Consistent and correct processing of video data.

This approach reflects core operating system principles such as **process/thread management**, **concurrency control**, and **synchronization**, enabling robust parallel program design in real-world applications.

3. Scenario: An e-commerce platform handles thousands of concurrent transactions during sales events. Each transaction involves checking inventory, processing payment, and updating stock levels in the database. The system runs on a multicore processor and must ensure correctness and performance under heavy load.

Question:

How would you use multicore programming techniques to efficiently handle concurrent transactions while ensuring data consistency, especially when multiple threads may attempt to update the same product's inventory simultaneously?

a. Workload Distribution Across Cores

To utilize all cores effectively, the transaction processing workload is divided among multiple worker threads. A **thread pool** can be used to handle incoming transactions in parallel. Each thread is assigned a task from a **transaction queue**.

```
// Pseudocode for thread pool usage
while (true) {
    Transaction txn = task_queue.get();
    thread_pool.execute(process_transaction, txn);
}
```

This allows multiple transactions to be processed simultaneously, leveraging all available cores.

b. Ensuring Thread Safety with Critical Sections

A critical challenge is to ensure that two threads do not simultaneously update the inventory of the same product, leading to **data inconsistency** or **race conditions**.

To avoid this, the following mechanisms are used:

i. Fine-Grained Locks (Per-Product Locking)

Each product has an associated mutex. Only one thread can modify the inventory of a product at a time.

```
std::mutex product_mutex[MAX_PRODUCTS];
```

```
void process_transaction(Transaction txn) {
    std::lock_guard<std::mutex> lock(product_mutex[txn.product_id]);

    if (inventory[txn.product_id] >= txn.quantity) {
        inventory[txn.product_id] -= txn.quantity;
        log_success(txn);
    } else {
        log_failure(txn, "Out of stock");
    }
}
```

This avoids the bottleneck of global locking and reduces contention between threads working on different products.

ii. Atomic Operations (Optional)

For simple inventory decrement operations where exact synchronization isn't critical, **atomic integers** can be used:

```
std::atomic<int> inventory[MAX_PRODUCTS];
```

```
if (inventory[txn.product_id].fetch_sub(txn.quantity) >= txn.quantity) {
    // success
}
```

However, this is not ideal if rollback or validation is needed.

c Handling Deadlocks and Starvation

- **Deadlock Prevention:** Always acquire locks in a consistent order if multiple locks are needed.
- **Starvation Avoidance:** Use **fair mutexes** or priority-aware task queues.

d. Final Consistency Check and Logging

After processing all transactions:

- A **synchronization barrier** or `thread.join()` ensures all threads complete.
- The final inventory state can be verified before committing to the main database.

Conclusion:

This multicore solution achieves:

- Efficient parallel transaction processing,
- Data consistency across all concurrent operations, and
- High system throughput during peak load.

By carefully managing **thread synchronization**, **lock granularity**, and **shared resource access**, the system balances performance with correctness—illustrating real-world application of **multicore operating system principles**.

Drawbacks:

- **Software complexity:** Developing software that effectively utilizes multiple cores requires a shift in programming paradigms, from sequential to parallel processing.
- **Data synchronization:** Ensuring data consistency and preventing race conditions when multiple cores access shared resources is crucial and can be difficult to manage.
- **Debugging multithreaded code:** Identifying and fixing bugs in multithreaded applications can be more challenging due to the complex interactions between threads.
- **Cost:** Multicore processors can be more expensive than single-core processors, although the price difference is becoming less significant over time.

Advantages:

➤ **Faster Task Execution:**

By dividing tasks into smaller parts and assigning them to different cores, multicore processors can execute them concurrently, significantly reducing overall processing time for computationally intensive operations.

➤ **Enhanced Multitasking:**

With multiple cores, a computer can handle numerous applications simultaneously without significant performance degradation. This allows for smoother transitions between tasks and a more responsive user experience.

➤ **Improved Application Performance:**

Applications designed to leverage multicore processors can achieve much higher performance levels compared to single-core systems. This is especially true for applications that can utilize parallel processing, such as video editing software, scientific simulations, and complex databases.

➤ Better Resource Utilization:

Multicore processors allow for more efficient sharing of resources like caches and memory buses, leading to better overall system performance and potentially lower power consumption.

➤ Fault Tolerance:

In some cases, multicore processors can offer a degree of fault tolerance. If one core fails, other cores can potentially take over its tasks, preventing complete system failure.

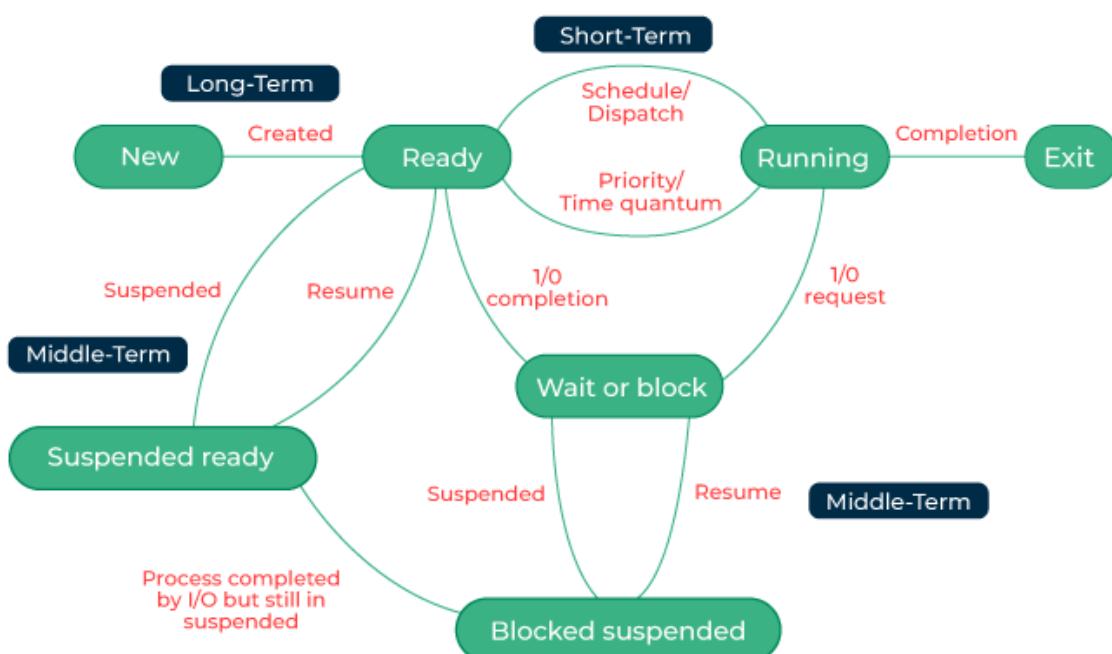
➤ Support for complex workloads:

Multicore processors can handle complex tasks like video encoding/decoding, real-time audio processing, and advanced simulations that would be challenging or impossible on single-core systems.

➤ Future-proofing:

As software development increasingly embraces parallel programming, multicore processors are well-positioned to take advantage of these advancements, offering a path to continued performance improvements.

Categories of Scheduling



Categories of Scheduling

Scheduling falls into one of two categories:

- ❖ **Non-Preemptive:** In this case, a process's resource cannot be taken before the process has finished running. When a running process finishes and transitions to a waiting state, resources are switched.

❖ **Preemptive:** In this case, the OS can switch a process from running state to ready state. This switching happens because the CPU may give other processes priority and substitute the currently active process for the higher priority process.

Types of Process Schedulers

There are three types of process schedulers:

1. Long Term or Job Scheduler

Long Term Scheduler loads a process from disk to main memory for execution. The new process to the ‘Ready State’.

- It mainly moves processes from [Job Queue](#) to [Ready Queue](#).
- It controls the Degree of [Multi-programming](#), i.e., the number of processes present in a ready state or in main memory at any point in time.
- It is important that the long-term scheduler make a careful selection of both I/O and CPU-bound processes. I/O-bound tasks are which use much of their time in input and output operations while CPU-bound processes are which spend their time on the CPU. The job scheduler increases efficiency by maintaining a balance between the two.
- In some systems, the long-term scheduler might not even exist. For example, in time-sharing systems like Microsoft Windows, there is usually no long-term scheduler. Instead, every new process is directly added to memory for the short-term scheduler to handle.
- Slowest among the three (that is why called long term).

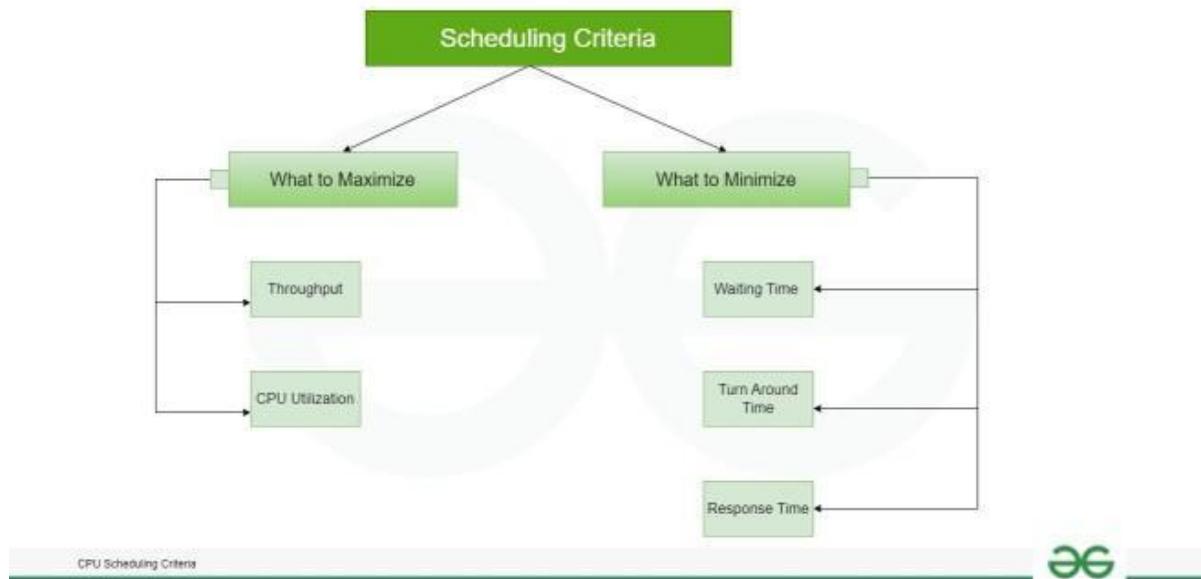
2. Short-Term or CPU Scheduler

CPU Scheduler is responsible for selecting one process from the ready state for running (or assigning CPU to it).

- STS (Short Term Scheduler) must select a new process for the CPU frequently to avoid starvation.
- The CPU scheduler uses different [scheduling algorithms](#) to balance the allocation of CPU time.
- It picks a process from ready queue.
- Its main objective is to make the best use of CPU.
- It mainly calls [dispatcher](#).
- Fastest among the three (that is why called Short Term).

CPU Scheduling Criteria

CPU scheduling is essential for the system’s performance and ensures that processes are executed correctly and on time. Different CPU scheduling algorithms have other properties and the choice of a particular algorithm depends on various factors. Many criteria have been suggested for comparing CPU scheduling algorithms.



Criteria of CPU Scheduling

Importance of Selecting the Right CPU Scheduling Algorithm for Specific Situations

It is important to choose the correct CPU scheduling algorithm because different algorithms have different priorities for different CPU scheduling criteria. Different algorithms have different strengths and weaknesses. Choosing the wrong CPU scheduling algorithm in a given situation can result in suboptimal performance of the system.

Example: Here are some examples of CPU scheduling algorithms that work well in different situations.

Round Robin scheduling algorithm works well in a time-sharing system where tasks have to be completed in a short period of time. SJF scheduling algorithm works best in a batch processing system where shorter jobs have to be completed first in order to increase throughput. Priority scheduling algorithm works better in a real-time system where certain tasks have to be prioritized so that they can be completed in a timely manner.

CPU Scheduling Algorithms:

There are several CPU Scheduling Algorithms, that are listed below.

- [First Come First Served \(FCFS\)](#)
- [Shortest Job First \(SJF\)](#)
- [Longest Job First \(LJF\)](#)
- [Priority Scheduling](#)
- [Round Robin \(RR\)](#)
- [Shortest Remaining Time First \(SRTF\)](#)
- [Longest Remaining Time First \(LRTF\)](#)

Program for FCFS CPU Scheduling

First come – First served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes according to the order they arrive in the ready queue. In this algorithm, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

Terminologies Used in CPU Scheduling

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Turn Around Time:** Time Difference between completion time and arrival time. Turn Around Time = (Completion Time – Arrival Time)
- **Waiting Time (W. T):** Time Difference between turnaround time and burst time. CPU Burst time is the overall CPU time a process needs.
Waiting Time = (Turn Around Time – Burst Time).

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :



P1 waiting time : 0

P2 waiting time : 24

P3 waiting time : 27

The Average waiting time :

$$(0+24+27)/3 = 17$$

In this example, we have assumed the arrival time of all processes is 0, so turnaround and completion times are the same.

Implementation

Input : Processes along with their burst time (bt).

Output : Waiting time, turnaround time for all processes.

1. As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. $wt[0] = 0$.

2. Find **waiting time** for all other processes i.e. for process i:
 $wt[i] = bt[i-1] + wt[i-1]$.
3. Find **turnaround time** = **waiting_time** + **burst_time** for all processes.
4. Find **average waiting time** = **total_waiting_time** / **no_of_processes**.
5. Similarly, find **average turnaround time** =
total_turn_around_time / **no_of_processes**.

Program for Shortest Job First (or SJF)

The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN), can be [preemptive or non-preemptive](#).

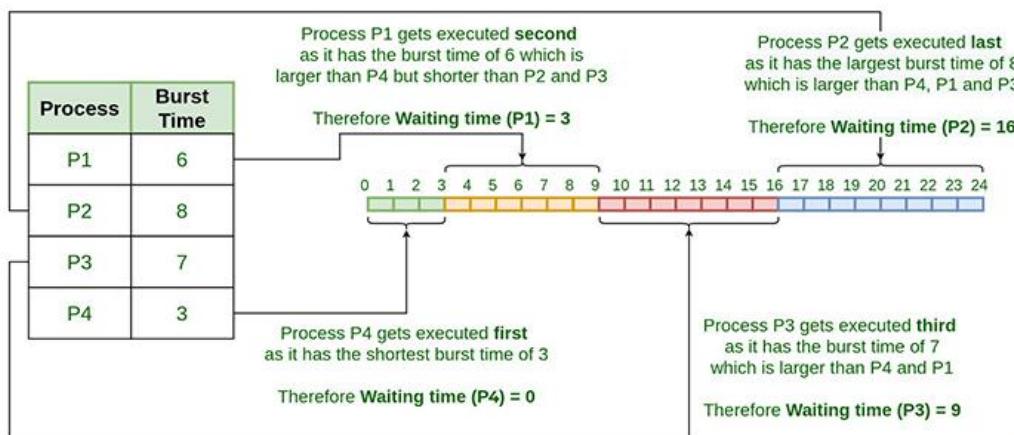
Characteristics of SJF Scheduling:

- Shortest Job first has the advantage of having a minimum average waiting time among all [scheduling algorithms](#).
- It is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
- SJF can be used in specialized environments where accurate estimates of running time are available.

Algorithm:

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

Shortest Job First (SJF) Scheduling Algorithm



How to compute below times in SJF using a program?

- **Completion Time:** Time at which process completes its execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

Turn Around Time = Completion Time – Arrival Time

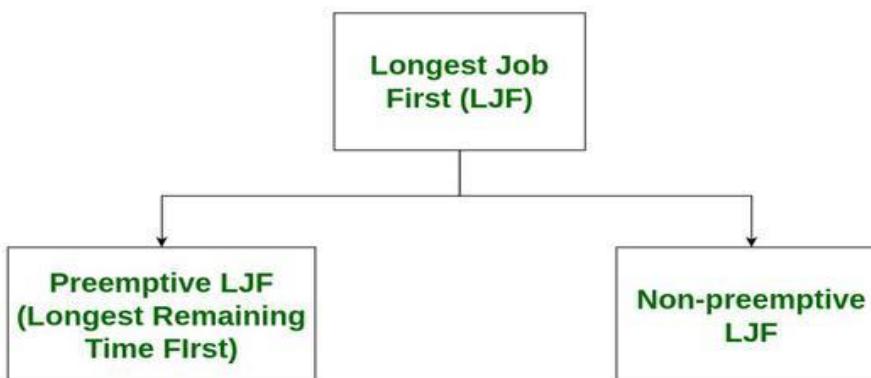
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time

Longest Job First (LJF)

Longest Job First (LJF) is a non-preemptive scheduling algorithm. This algorithm is based on the burst time of the processes. The processes are put into the ready queue based on their burst times i.e., in descending order of the burst times. As the name suggests this algorithm is based on the fact that the process with the largest burst time is processed first. The burst time of only those processes is considered that have arrived in the system until that time. Its preemptive version is called Longest Remaining Time First (LRTF) algorithm.

Types of LJF Scheduling Algorithms



Characteristics of Longest Job First(Non-Preemptive)

- Among all the processes waiting in a waiting queue, the CPU is always assigned to the process having the largest burst time.
- If two processes have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages of Longest Job First(LJF)

- No other process can execute until the longest job or process executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages of Longest Job First CPU Scheduling Algorithm

- This algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to a convoy effect.
- It may happen that a short process may never get executed and the system keeps on executing the longer processes.
- It reduces the processing speed and thus reduces the efficiency and utilization of the system.

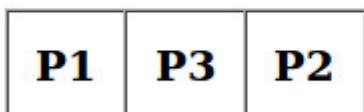
Program for Priority CPU Scheduling

Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. The process with the highest priority is to be executed first and so on. Processes with the same priority are executed on a first-come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement. Also priority can be decided on the ratio of average I/O to average CPU burst time.

Implementation:

- 1- First input the processes with their burst time and priority.
- 2- Sort the processes, burst time and priority according to the priority.
- 3- Now simply apply FCFS algorithm.

Process	Burst Time	Priority
P1	10	2
P2	5	0
P3	8	1



0 10 18 23

Program for Round Robin Scheduling for the Same Arrival Time

Inter Process Communication (IPC)

Processes need to communicate with each other in many situations, for example, to count occurrences of a word in text file, output of grep command needs to be given to wc command, something like grep -o -i <word> <file> | wc -l. **Inter-Process Communication or IPC** is a mechanism that allows processes to communicate. It helps processes synchronize their activities, share information, and avoid conflicts while accessing shared resources.

Types of Process

Let us first talk about types of types of processes.

- **Independent process:**

An independent process is not affected by the execution of other processes. Independent processes are processes that do not share any data or resources with other processes. No inter-process communication required here.

- **Co-operating process:**

Interact with each other and share data or resources. A co-operating process can be affected by other executing processes. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them.

- **Inter Process Communication**

Inter process communication (IPC) allows different programs or processes running on a computer to share information with each other. IPC allows processes to communicate by using different techniques like sharing memory, sending messages, or using files. It ensures that processes can work together without interfering with each other. Cooperating processes require an Inter Process Communication (IPC) mechanism that will allow them to exchange data and information.

The two fundamental models of Inter Process Communication are:

- **Shared Memory**
- **Message Passing**

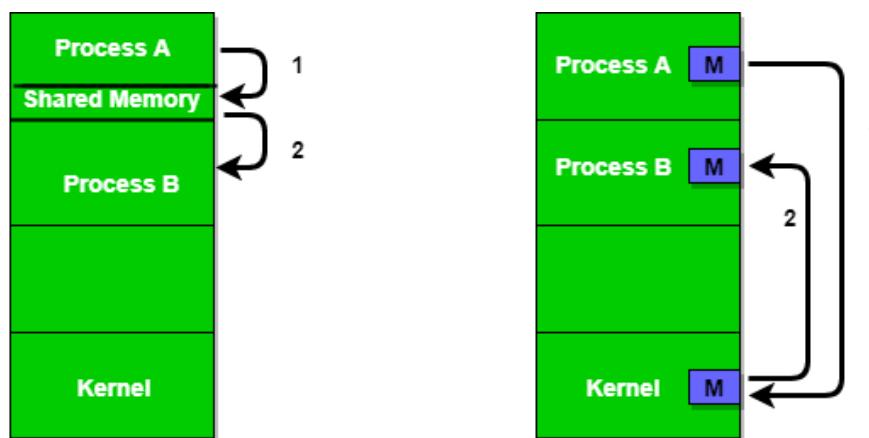


Figure 1 - Shared Memory and Message Passing

- Figure 1 above shows a basic structure of communication between processes via the shared memory method and via the message passing method.
- An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One

way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

- Figure 1 above shows a basic structure of communication between processes via the shared memory method and via the message passing method.

Methods in Inter process Communication

Inter-Process Communication refers to the techniques and methods that allow processes to exchange data and coordinate their activities. Since processes typically operate independently in a multitasking environment, IPC is essential for them to communicate effectively without interfering with one another. There are several methods of IPC, each designed to suit different scenarios and requirements. These methods include shared memory, message passing, semaphores, and signals, etc.

To read more refer – [methods of Inter Process Communication](#)

Role of Synchronization in IPC

In IPC, synchronization is essential for controlling access to shared resources and guaranteeing that processes do not conflict with one another. Data consistency is ensured and problems like race situations are avoided with proper synchronization.

Advantages of IPC

- Enables processes to communicate with each other and share resources, leading to increased efficiency and flexibility.
- Facilitates coordination between multiple processes, leading to better overall system performance.
- Allows for the creation of distributed systems that can span multiple computers or networks.
- Can be used to implement various [synchronization](#) and communication protocols, such as semaphores, pipes, and sockets.

Disadvantages of IPC

- Increases system complexity, making it harder to design, implement, and debug.
- Can introduce security vulnerabilities, as processes may be able to access or modify data belonging to other processes.
- Requires careful management of system resources, such as memory and CPU time, to ensure that IPC operations do not degrade overall system performance.
Can lead to data inconsistencies if multiple processes try to access or modify the same data at the same time.
- Overall, the advantages of IPC outweigh the disadvantages, as it is a necessary mechanism for modern operating systems and enables processes to work together and share resources in a flexible and efficient manner. However, care must be taken to design and implement IPC systems carefully, in order to avoid potential security vulnerabilities and performance issues.

DEADLOCK

A deadlock is a situation where a set of processes is blocked because each process is holding a resource and waiting for another resource acquired by some other process. In this article, we will discuss deadlock, its necessary conditions, etc. in detail.

- **Deadlock** is a situation in computing where two or more processes are unable to proceed because each is waiting for the other to release resources.
- Key concepts include mutual exclusion, resource holding, circular wait, and no preemption.

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. This is a practical example of deadlock.

How Does Deadlock occur in the Operating System?

Before going into detail about how deadlock occurs in the Operating System, let's first discuss how the Operating System uses the resources present. A process in an operating system uses resources in the following way.

- **Requests a resource**
- **Use the resource**
- **Releases the resource**

A situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). **For example**, in the below diagram, Process 1 is holding Resource 1 and waiting

for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

Necessary Conditions for Deadlock in OS

Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)

- **Mutual Exclusion:** Only one process can use a resource at any given time i.e. the resources are non-sharable.
- **Hold and Wait:** A process is holding at least one resource at a time and is waiting to acquire other resources held by some other process.
- **No Preemption:** A resource cannot be taken from a process unless the process releases the resource.
- **Circular Wait:** set of processes are waiting for each other in a circular fashion. For example, lets say there are a set of processes {P0P0,P1P1 ,P2P2,P3P3} such that P0P0 depends on P1P1, P1P1 depends on P2P2 , P2P2 depends on P3P3 and P3P3 depends on P0P0. This creates a circular relation between all these processes and they have to wait forever to be executed.

Methods of Handling Deadlocks in Operating System

There are three ways to handle deadlock:

1. Deadlock Prevention or Avoidance
2. Deadlock Detection and Recovery
3. Deadlock Ignorance

Deadlock Prevention or Avoidance

Deadlock Prevention and Avoidance is the one of the methods for handling deadlock. First, we will discuss Deadlock Prevention, then Deadlock Avoidance.

Deadlock Prevention

In deadlock prevention the aim is to not let full-fill one of the required condition of the deadlock. This can be done by this method:

(i) Mutual Exclusion

We only use the Lock for the non-share-able resources and if the resource is share- able (like read only file) then we not use the locks here. That ensure that in case of share -able resource , multiple process can access it at same time. Problem- Here the problem is that we can only do it in case of share-able resources but in case of no-share-able resources like printer , we have to use Mutual exclusion.

(ii) Hold and Wait

To ensure that Hold and wait never occurs in the system, we must guarantee that whenever process request for resource , it does not hold any other resources.

- we can provide the all resources to the process that is required for it's execution before starting it's execution . **problem** – for example if there are three resource that is required by a process and we have given all that resource before starting execution of process then there might be a situation that initially we required only two resource and after one hour we want third resources and this will cause starvation for the another process that wants this resources and in that waiting time that resource can allocated to other process and complete their execution.
- We can ensure that when a process request for any resources that time the process does not hold any other resources. Ex- Let there are three resources DVD, File and Printer . First the process request for DVD and File for the copying data into the file and let suppose it is going to take 1 hour and after it the process free all resources then again request for File and Printer to print that file.

(iii) No Preemption

If a process is holding some resource and request other resources that are acquired and these resource are not available immediately then the resources that current process is holding are preempted. After some time process again request for the old resources and other required resources to restart.

For example – Process p1 have resource r1 and requesting for r2 that is hold by process p2. then process p1 preempt r1 and after some time it try to restart by requesting both r1 and r2 resources.

Problem – This can cause the Live Lock Problem .

Live Lock: Live lock is the situation where two or more processes continuously changing their state in response to each other without making any real progress.

Example:

- suppose there are two processes p1 and p2 and two resources r1 and r2.
- Now, p1 acquired r1 and need r2 & p2 acquired r2 and need r1.
- so according to above method- Both p1 and p2 detect that they can't acquire second resource, so they release resource that they are holding and then try again.
- continuous cycle- p1 again acquired r1 and requesting to r2 p2 again acquired r2 and requesting to r1 so there is no overall progress still process are changing there state as they preempt resources and then again holding them. This the situation of Live Lock.

(iv) Circular Wait:

To remove the circular wait in system we can give the ordering of resources in which a process needs to acquire.

Ex: If there are process p1 and p2 and resources r1 and r2 then we can fix the resource acquiring order like the process first need to acquire resource r1 and then resource r2. so the process that acquired r1 will be allowed to acquire r2 , other process needs to wait until r1 is free.

This is the Deadlock prevention methods but practically only fourth method is used as all other three condition removal method have some disadvantages with them.

Deadlock Avoidance

Avoidance is kind of futuristic. By using the strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources that the process will need is known to us before the execution of the process. We use Banker’s algorithm to avoid deadlock.

In prevention and avoidance, we get the correctness of data but performance decreases.

Deadlock Detection and Recovery

If [Deadlock prevention or avoidance](#) is not applied to the software then we can handle this by deadlock detection and recovery. which consist of two phases:

1. In the first phase, we examine the state of the process and check whether there is a deadlock or not in the system.
2. If found deadlock in the first phase then we apply the algorithm for recovery of the deadlock.

In Deadlock detection and recovery, we get the correctness of data but performance decreases.

Deadlock Detection

Deadlock detection is a process in computing where the system checks if there are any sets of processes that are stuck waiting for each other indefinitely, preventing them from moving forward. In simple words, deadlock detection is the process of finding out whether any process are stuck in loop or not.

There are several algorithms like:

- [Resource Allocation Graph](#)
- [Banker's Algorithm](#)

These algorithms helps in detection of deadlock in Operating System.

Deadlock Recovery

There are several Deadlock Recovery Techniques:

- Manual Intervention
- Automatic Recovery
- Process Termination

- Resource Preemption

1. Manual Intervention

When a deadlock is detected, one option is to inform the operator and let them handle the situation manually. While this approach allows for human judgment and decision-making, it can be time-consuming and may not be feasible in large-scale systems.

2. Automatic Recovery

An alternative approach is to enable the system to recover from deadlock automatically. This method involves breaking the deadlock cycle by either aborting processes or preempting resources. Let's delve into these strategies in more detail.

3. Process Termination

- Abort all Deadlocked Processes

This approach breaks the deadlock cycle, but it comes at a significant cost. The processes that were aborted may have executed for a considerable amount of time, resulting in the loss of partial computations. These computations may need to be recomputed later.

- Abort one process at a time

Instead of aborting all deadlocked processes simultaneously, this strategy involves selectively aborting one process at a time until the deadlock cycle is eliminated. However, this incurs overhead as a deadlock-detection algorithm must be invoked after each process termination to determine if any processes are still deadlocked.

- Factors for choosing the termination order:
The process's priority

- 1.Completion time and the progress made so far
- 2.Resources consumed by the process
- 3.Resources required to complete the process
- 4.Number of processes to be terminated
- 5.Process type (interactive or batch)

4. Resource Preemption

- Selecting a Victim

Resource preemption involves choosing which resources and processes should be preempted to break the deadlock. The selection order aims to minimize the overall cost of recovery. Factors considered for victim

selection may include the number of resources held by a deadlocked process and the amount of time the process has consumed.

- **Rollback**

If a resource is preempted from a process, the process cannot continue its normal execution as it lacks the required resource. Rolling back the process to a safe state and restarting it is a common approach. Determining a safe state can be challenging, leading to the use of total rollback, where the process is aborted and restarted from scratch.

- **Starvation Prevention**

To prevent resource starvation, it is essential to ensure that the same process is not always chosen as a victim. If victim selection is solely based on cost factors, one process might repeatedly lose its resources and never complete its designated task. To address this, it is advisable to limit the number of times a process can be chosen as a victim, including the number of rollbacks in the cost factor.

Deadlock Ignorance

If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take. we use the ostrich algorithm for deadlock ignorance.

“In Deadlock, ignorance performance is better than the above two methods but the correctness of data is not there.”

Safe State

A safe state can be defined as a state in which there is no deadlock. It is achievable if:

- If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. if such a sequence does not exist, it is an unsafe state.
- All the requested resources are allocated to the process.

Difference between Starvation and Deadlocks

Aspect	Deadlock	Starvation
Definition	A condition where two or more processes are blocked forever, each waiting for a resource held by another.	A condition where a process is perpetually denied necessary resources, despite resources being available.
Resource Availability	Resources are held by processes involved in the deadlock.	Resources are available but are continuously allocated to other processes.
Cause	Circular dependency between processes, where each process is waiting for a resource from another.	Continuous preference or priority given to other processes, causing a process to wait indefinitely.
Resolution	Requires intervention, such as aborting processes or preempting resources to break the cycle.	Can be mitigated by adjusting scheduling policies to ensure fair resource allocation.