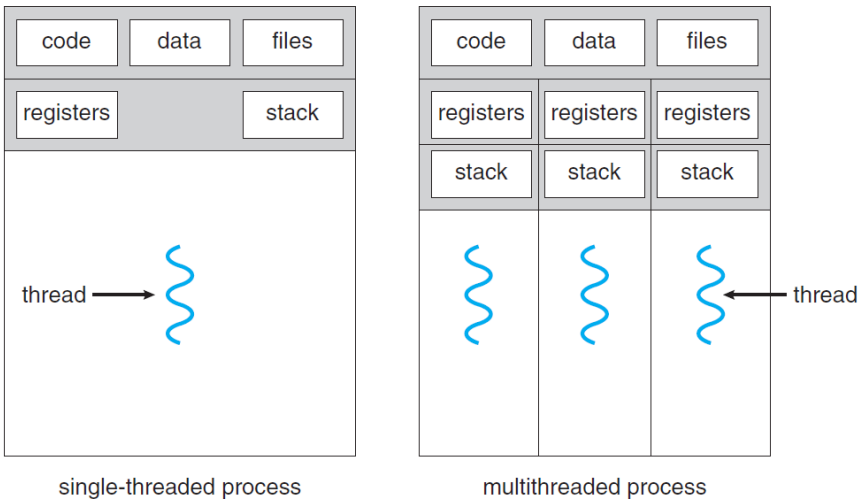
Unit 2

Multithreaded programming

1. Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set and a stack. 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. Below figure illustrates the difference between a traditional single-threaded process and a multithreaded process.

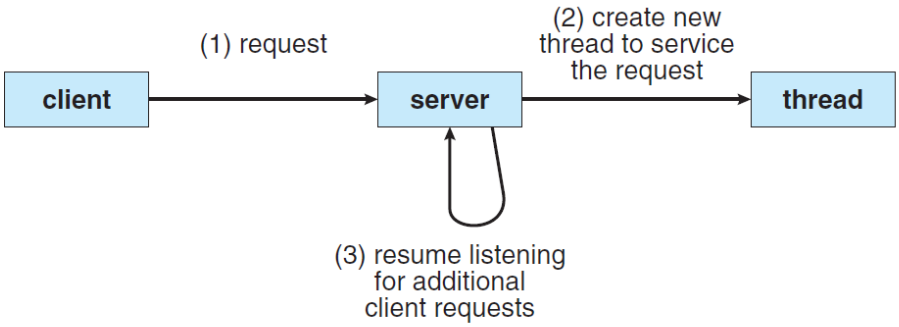


Single-threaded and multithreaded processes.

i) Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background. In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests. This is illustrated in the below figure.



Multithreaded server architecture.

ii) 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. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.

2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

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. Empirically gauging the difference in overhead can be difficult, but in

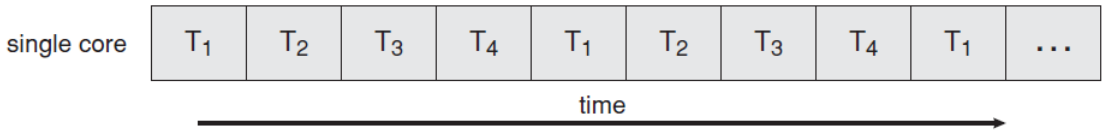
general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

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. A single-threaded process can run on only one processor, regardless how many are available.

2. Multicore programming

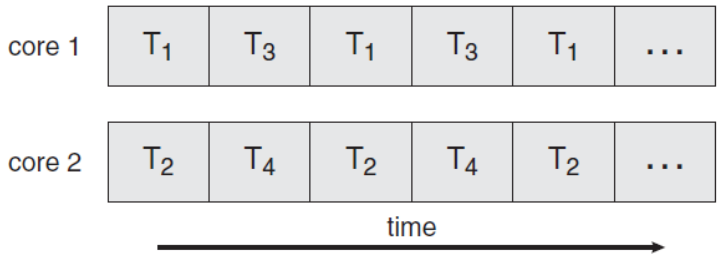
Multicore programming refers to the process of writing software that can run on systems with multiple cores in a single CPU. Each core in a multicore processor can execute instructions independently, allowing for parallel execution of tasks, which enhances performance and efficiency.

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (below figure).



Concurrent execution on a single-core system.

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 (below figure).



Parallel execution on a multicore system.

Key concepts of multicore programming:

1. **Concurrency and Parallelism**:
   - Concurrency: Executing multiple tasks in overlapping time frames.
   - Parallelism: Executing multiple tasks simultaneously on different cores.
2. **Multithreading**:
   - Using threads to break down tasks and run them concurrently on multiple cores.

- Thread libraries like **pthreads** (POSIX threads) are commonly used for this purpose.

3. **Synchronization**:
   - Managing access to shared resources among multiple threads using tools like mutexes, locks, and semaphores to prevent race conditions and ensure consistency.
4. **Load Balancing**:
   - Distributing tasks evenly across cores to optimize performance and avoid overburdening any single core.
5. **Data Sharing and Memory Access**:
   - Ensuring efficient and safe access to shared data among threads to prevent data corruption or inconsistent states.
6. **Task Decomposition**:
   - Breaking down a program into smaller tasks or processes that can be executed in parallel.
7. **Thread Safety**:
   - Writing code that ensures proper functioning when accessed by multiple threads concurrently.
8. **Scalability**:
   - Designing programs that can take advantage of increasing numbers of cores as hardware evolves.
9. **Cache Coherence**:
   - Ensuring that all cores have consistent views of memory when accessing shared data, typically handled by the hardware.
10. **Performance Optimization**:

- Minimizing the overhead of thread creation, synchronization, and data sharing to achieve maximum speed-up.

i) Programming Challenges

Five areas present challenges in programming for multicore systems:

1. Identifying tasks. This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

2. Balance. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

3. Data splitting. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

4. Data dependency. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

5. Testing and debugging. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications. Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future.

ii) Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

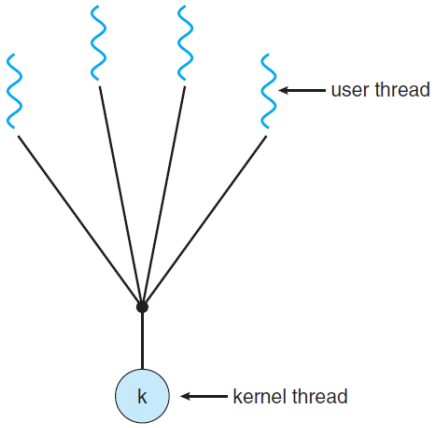
- **Task-based parallelism**: Dividing the program into independent tasks that can run concurrently on different cores.
- **Data parallelism**: Distributing data across multiple cores and performing the same operation on each partition of the data set in parallel.

3. Multithreading models

Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

(i) Many-to-One Model

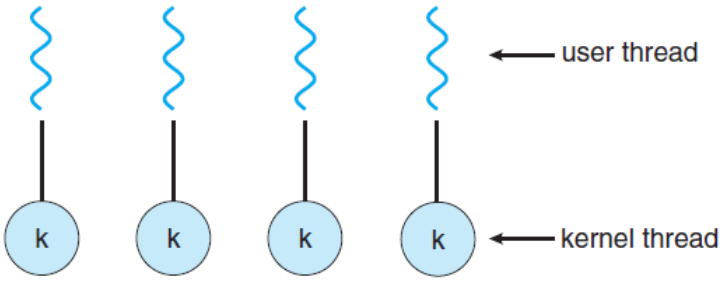
The many-to-one model (Below Figure) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model.



Many-to-one model.

(ii) One-to-One Model

The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.



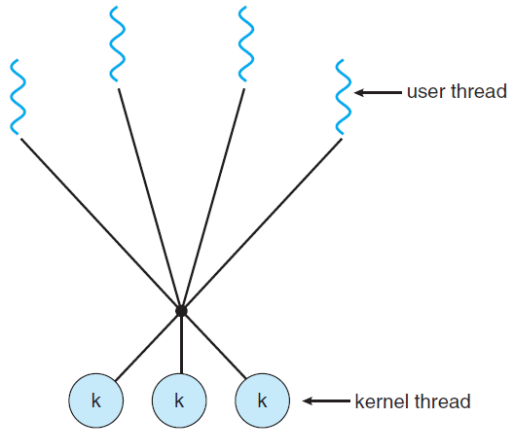
One-to-one model.

(iii) Many-to-Many Model

The many-to-many model, multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).

Let's consider the effect of this design on concurrency. Whereas the many to- one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time. The one-to-one model

allows greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many-to-many model.

4. Thread libraries – pthreads

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: POSIX Pthreads,Windows, and Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library. The Windows thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using theWindows API; UNIX and Linux systems often use Pthreads.

POSIX Pthreads:

Pthreads (POSIX threads) is a widely used multithreading library in UNIX-like operating systems such as Linux, macOS, and FreeBSD. It provides a standardized API that enables developers to write concurrent programs by creating and managing threads within a single process. The core functions include pthread_create() for spawning threads, pthread_join() for waiting for threads to complete, and pthread_exit() for terminating threads. Pthreads allows fine-grained control over thread behavior and attributes, including scheduling, detachability, and concurrency levels. Thread safety is managed through synchronization mechanisms such as mutexes, condition variables, and semaphores, which ensure that shared resources are accessed in a thread-safe manner.

Pthreads is highly portable across UNIX-based systems and provides low-level control, making it ideal for high-performance applications that require efficient multitasking. However, with this power comes complexity, as developers must explicitly manage synchronization and communication between threads. Deadlocks and race conditions can arise if shared resources are not handled carefully. Despite these challenges, Pthreads remains a go-to solution for developers building multithreaded applications in performance-critical environments, such as servers, real-time systems, and parallel processing applications.

Windows Threads:

Windows threads are a core part of the Windows operating system, allowing developers to create multithreaded applications that take advantage of modern multicore processors. The Windows API provides several functions for thread management, including CreateThread() to launch new threads and WaitForSingleObject() to wait for thread completion. Each thread is assigned its own stack and register set, and synchronization between threads is achieved using Windows-specific mechanisms such as critical sections, mutexes, semaphores, and events. Windows also supports advanced features like thread pools, which optimize thread usage by reusing threads for multiple tasks, improving system performance and resource management.

One of the key benefits of Windows threads is the ability to prioritize threads, allowing more critical tasks to execute with higher CPU time. However, like other threading libraries, developers must carefully manage synchronization to avoid issues like deadlocks or priority inversion. Windows also offers more sophisticated thread management through the use of thread affinity, which allows developers to bind threads to specific CPU cores, enabling better

performance tuning for multi-core systems. Overall, Windows threads offer extensive flexibility and control for building responsive, high-performance applications on the Windows platform.

Java Threads:

Java threads are an integral part of the Java programming language, providing a platform-independent way to implement multithreading. Threads in Java can be created either by extending the Thread class or implementing the Runnable interface. This allows developers to run multiple threads concurrently, sharing system resources while maintaining separate execution paths. Java provides built-in synchronization mechanisms such as the synchronized keyword, which ensures that only one thread can access a critical section of code at a time. In addition to these low-level controls, Java also offers higher-level concurrency utilities in the java.util.concurrent package, such as Executors and ThreadPools, simplifying the management of thread execution.

The portability of Java threads across different platforms is one of their biggest strengths, making it easier to write cross-platform, multithreaded applications. Java abstracts away the complexities of native thread management and provides robust error-handling mechanisms. Java threads are often used in server-side applications, game development, and desktop software where concurrency is critical. However, developers must still handle issues like race conditions and deadlocks when multiple threads interact with shared resources. Java's thread management tools, combined with its automatic memory management (via the garbage collector), make it a versatile option for building scalable, concurrent applications.

CPU scheduling and Process Synchronization

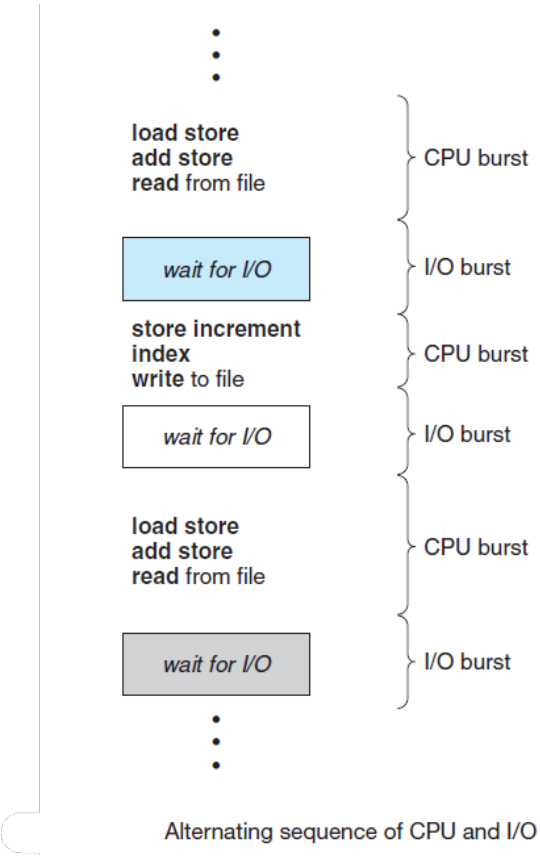
1. Basic concepts

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

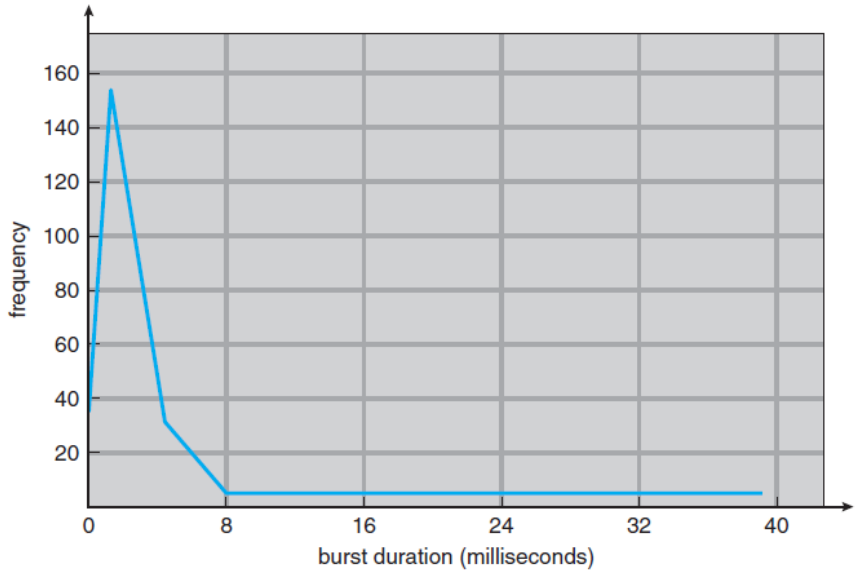
i) CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution as shown in below figure.



Alternating sequence of CPU and I/O bursts.

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that as shown in below figure. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.



Histogram of CPU-burst durations.

ii) CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler, or CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

iii) Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process) When a process switches from the running state to the ready state (for example, when an interrupt occurs).

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2

and 3. When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative. Otherwise, it is preemptive. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel maybe busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. Certain operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete or for an I/O block to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing where tasks must complete execution within a given time frame.

Another component involved in the CPU-scheduling function is the **dispatcher.** The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

2. Scheduling criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

• ***CPU utilization***. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

• ***Throughput***. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

• ***Turnaround time***. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

• ***Waiting time***. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

• ***Response time***. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Benchmark	Turnaround Time	Waiting Time
Definition	The time it takes for a process to complete from the moment it is submitted to the moment it is finished.	The total time a process spends waiting in a ready queue before reaching the CPU.
Calculation	Turnaround Time = Completion Time – Arrival Time/ Burst Time + Waiting Time	Waiting Time = Turnaround Time – Burst Time
Importance	Helps in understanding the efficiency of the CPU in processing tasks.	Can cause delays in the execution of other processes, leading to lower system performance.
Optimization	By optimizing the CPU scheduling algorithm, turnaround time can be reduced.	By minimizing the waiting time, more processes can be executed in less time, improving system efficiency.
Units	Usually measured in milliseconds, seconds or minutes.	Usually measured in the same unit as the burst time, such as milliseconds, seconds or minutes.
It is limited by the speed of the output device?	Yes.	No.

3. Scheduling algorithms

Scheduling algorithms are used by the operating system to determine which process or thread should run on the CPU at a given time. These algorithms are critical for managing system performance, responsiveness, and fairness. There are several types of scheduling algorithms, each designed for different system requirements, such as maximizing CPU utilization, reducing response time, or ensuring fairness between processes.

1. First-Come, First-Served (FCFS):

- **Description**: In FCFS, the process that arrives first is executed first. Processes are handled in the order they appear in the ready queue.
- **Advantages**: Simple to implement and understand.
- **Disadvantages**: It can lead to **convoy effect**, where shorter tasks have to wait for long tasks to finish, increasing the average waiting time.

2. Shortest Job First (SJF):

- **Description**: In SJF, the process with the shortest estimated running time is executed first.
- **Advantages**: Minimizes the average waiting time compared to FCFS.
- **Disadvantages**: It can cause **starvation** of longer processes if shorter processes keep arriving.

3. Round Robin (RR):

- **Description**: Each process is assigned a fixed **time quantum**. After each quantum, the process is moved to the end of the ready queue if it has not finished.
- **Advantages**: Fair to all processes, especially in time-sharing systems. Every process gets a chance to run within a fixed amount of time.

- **Disadvantages**: Performance depends on the choice of the time quantum. Too small quantum leads to too many context switches, while too large a quantum degenerates to FCFS.

4. Priority Scheduling:

- **Description**: Each process is assigned a priority, and the CPU is allocated to the process with the highest priority.
- **Advantages**: Important tasks are given preference, ensuring critical processes are executed first.
- **Disadvantages**: Can lead to **starvation** of low-priority processes, but can be resolved using techniques like **aging** (gradually increasing the priority of waiting processes).

5. Multilevel Queue Scheduling:

- **Description**: The ready queue is divided into multiple queues, each with its own scheduling algorithm (e.g., system processes in one queue, user processes in another). Processes are permanently assigned to one queue based on their type or priority.
- **Advantages**: Provides a good balance between different types of processes, like interactive and batch processes.
- **Disadvantages**: Once assigned to a queue, processes cannot move between queues, which may lead to inefficiencies.

6. Multilevel Feedback Queue:

- **Description**: Similar to multilevel queue scheduling, but processes can move between queues based on their behavior (e.g., a process that uses too much CPU may be moved to a lower-priority queue).
- **Advantages**: Flexible and allows for better control over process behavior. Avoids starvation by dynamically adjusting priorities.
- **Disadvantages**: Complex to implement and fine-tune.

7. Shortest Remaining Time First (SRTF):

- **Description**: A preemptive version of SJF, where the process with the shortest remaining execution time is selected for CPU.
- **Advantages**: Optimal for minimizing the average waiting time.
- **Disadvantages**: Similar to SJF, longer processes can experience starvation.

8. Real-Time Scheduling:

- **Description**: Used in real-time systems, where tasks must be completed within a strict time deadline. Algorithms include **Rate Monotonic Scheduling (RMS)** and **Earliest Deadline First (EDF)**.
- **Advantages**: Ensures that time-critical tasks are completed on time.

- **Disadvantages**: Complex to implement, and not all tasks can always be guaranteed to meet deadlines.

Each scheduling algorithm is suited for different types of systems. **FCFS** is simple but inefficient in time-sharing systems. **SJF** and **SRTF** are good for minimizing wait times but can cause starvation. **Round Robin** ensures fairness in time-sharing environments. **Priority Scheduling** and **Multilevel Feedback Queue** provide more nuanced control, balancing different types of tasks efficiently.

Real-Time CPU Scheduling

Real-Time CPU Scheduling is used in systems where tasks or processes need to be executed within strict time constraints, commonly seen in **real-time operating systems (RTOS)**. These systems are typically found in environments where timely and deterministic task execution is critical, such as embedded systems, medical devices, robotics, telecommunications, and industrial control systems.

Real-time systems are classified into two types:

- **Hard Real-Time Systems**: Missing a deadline results in a critical failure (e.g., pacemaker systems or automotive airbag systems).
- **Soft Real-Time Systems**: Missing a deadline results in degraded performance but not critical failure (e.g., video streaming or gaming systems).

Real-time tasks or processes have specific deadlines that must be met. These tasks are associated with **periodic** or **aperiodic** characteristics.

- **Periodic tasks**: Tasks that repeat at regular intervals (e.g., sensor readings).
- **Aperiodic tasks**: Tasks that occur unpredictably (e.g., emergency signals).

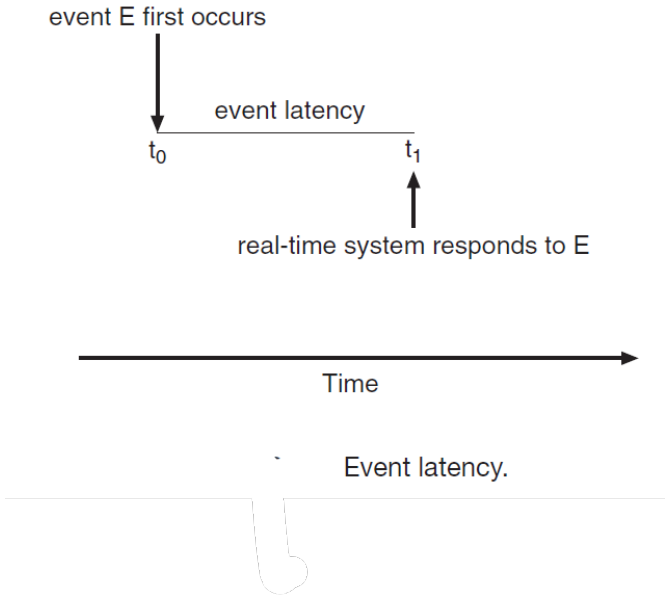
Key Metrics for Real-Time Scheduling:

- **Deadline**: The time by which a task must be completed.
- **Response Time**: The time it takes for a system to respond to an event or request.
- **Jitter**: Variation in the time it takes to complete a task, which real-time systems aim to minimize.

Several issues related to process scheduling in both soft and hard real-time operating systems are as below.

❖ *Minimizing Latency*

Consider the event-driven nature of a real-time system. The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. E**vent latency** is the amount of time that elapses from when an event occurs to when it is serviced (Figure below). Usually, different events have different latency requirements.



Event latency.

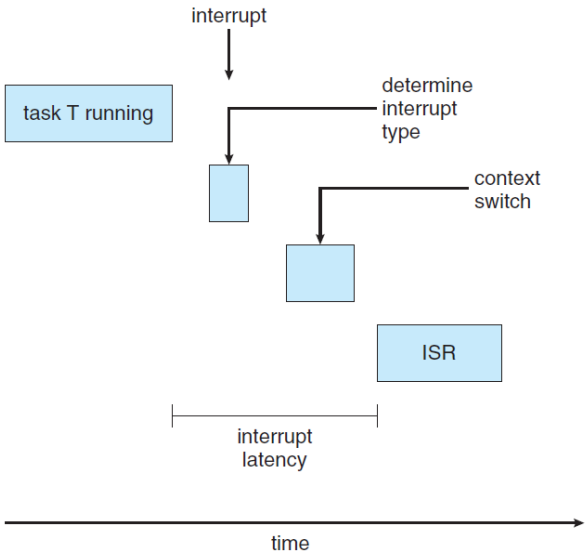
Two types of latencies affect the performance of real-time systems:

1. Interrupt latency

2. Dispatch latency

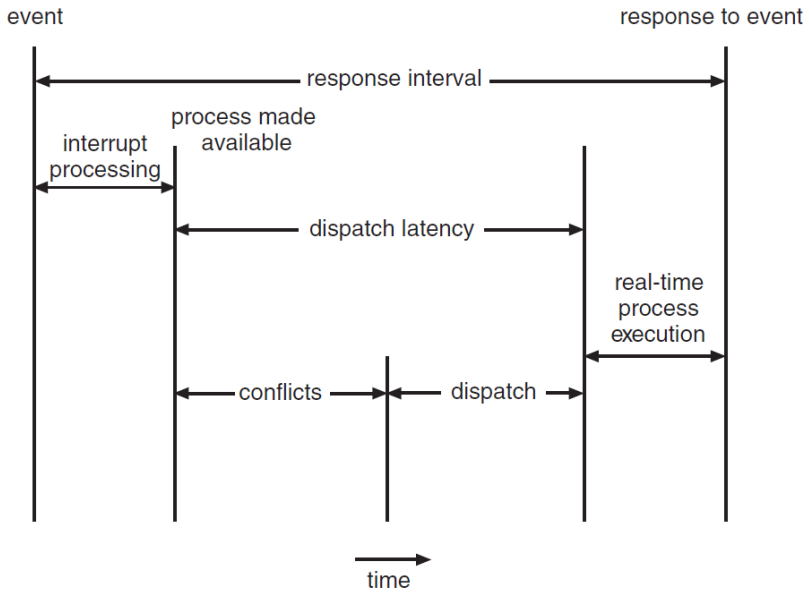
Interrupt latency refers to the period of time fromthe arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure below).

Obviously, it is crucial for real time operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention. Indeed, for hard real-time systems, interrupt latency must not simply be minimized, it must be bounded to meet the strict requirements of these systems.



Interrupt latency.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.



Dispatch latency.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as dispatch latency. Providing real-time tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well. The most effective technique for keeping dispatch latency low is to provide preemptive kernels.

The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel.

2. Release by low-priority processes of resources needed by a high-priority Process.

❖ *Priority-Based Scheduling*

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run. Providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features.

❖ *Rate-Monotonic Scheduling*

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

❖ *Earliest-Deadline-First Scheduling*

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

❖ *Proportional Share Scheduling*

Proportional share schedulers operate by allocating *T* shares among all applications. An application can receive *N* shares of time, thus ensuring that the application will have *N/T* of the total processor time. As an example, assume that a total of *T* = 100 shares is to be divided among three processes, *A*, *B*, and *C*. *A*is assigned 50 shares, *B* is assigned 15 shares, and *C* is assigned 20 shares. This scheme ensures that *A* will have 50 percent of total processor time, *B* will have 15 percent, and *C* will have 20 percent.

❖ *POSIX Real-Time Scheduling*

The POSIX standard also provides extensions for real-time computing— POSIX.1b. POSIX defines two scheduling classes for real-time threads:

• SCHED FIFO

• SCHED RR

SCHED FIFO schedules threads according to a first-come, first-served policy using a FIFO queue. However, there is no time slicing amongthreads of equal priority.Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED RR uses a round-robin policy. It is similar to SCHED FIFO except that it provides time slicing among threads of equal priority. POSIX provides an additional scheduling class—SCHED OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The POSIX API specifies the following two functions for getting and setting the scheduling policy:

• pthread attr getsched policy(pthread attr t *attr, int *policy)

• pthread attr setsched policy(pthread attr t *attr, int policy)

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either (1) a pointer to an integer that is set to the current scheduling policy (for pthread attr getsched policy()) or (2) an integer value (SCHED FIFO, SCHED RR, or SCHED OTHER) for the pthread attr setsched policy() function. Both functions return nonzero values if an error occurs.

CPU Utilization of a Process

CPU utilization refers to the percentage of CPU time a process uses during its execution. It can be calculated by comparing the time the CPU spends executing the process to the total observation time.

Formula:

$$\text{CPU Utilization} = \left(\frac{\text{CPU time used by process}}{\text{Total observation time}} \right) \times 100$$

Example 1: Single Process

Given:

- **CPU time used by the process**: 40 ms
- **Total observation time**: 100 ms

Calculation:

$$\text{CPU Utilization} = \left(\frac{40}{100} \right) \times 100 = 40\%$$

This means the process utilized 40% of the CPU during the observation period.

Example 2: Multiple Processes

Given:

- Process A: Executes for 25 ms
- Process B: Executes for 15 ms
- Process C: Executes for 10 ms
- Total observation time: 100 ms

Calculation:

1. **Total CPU Time Used**:

   25+15+10=50ms

2. **CPU Utilization**:

$$\text{CPU Utilization} = \left(\frac{50}{100} \right) \times 100 = 50\%$$

The CPU is utilized 50% of the time during the observation period.

Example 3: Real-Time System with Idle Time

Given:

- Process A: Executes for 30 ms
- Process B: Executes for 20 ms

- Idle Time: 50 ms
- Total observation time: 100 ms

Calculation:

1. **Total CPU Time Used**:

   30+20=50ms

2. **CPU Utilization**:

$$\text{CPU Utilization} = \left(\frac{50}{100}\right) \times 100 = 50\%$$

   The remaining 50% of the time, the CPU was idle.

- Idle Time: If the CPU is not processing any task, it's considered idle.
- Observation Period: This can be any interval of time, depending on what you're analyzing.
- High Utilization: Indicates better CPU usage but could lead to higher contention.
- Low Utilization: Indicates underutilization or system inefficiency.

Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

https://www.youtube.com/watch?v=xgW8VhEOpFg

https://www.youtube.com/watch?v=ERg3tnHxms0

https://www.youtube.com/watch?v=Cv_5aoKXc3g

https://www.youtube.com/watch?v=7SyX1mmcSDs

CPU utilization for scheduling *N* processes

For n tasks in RMS, the maximum allowable CPU utilization (U_{max}) is:

$$U_{max} = n \cdot (2^{1/n} - 1)$$

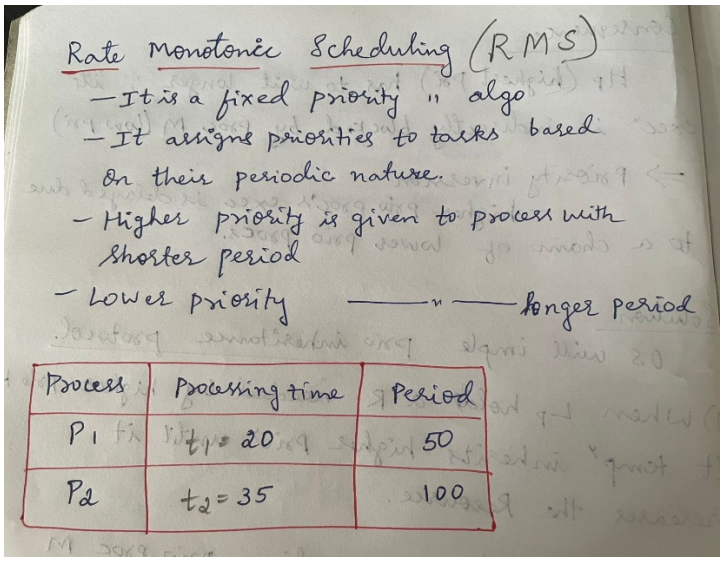
This formula ensures that the tasks meet their deadlines if the calculated CPU utilization U is less than or equal to U_{max}.

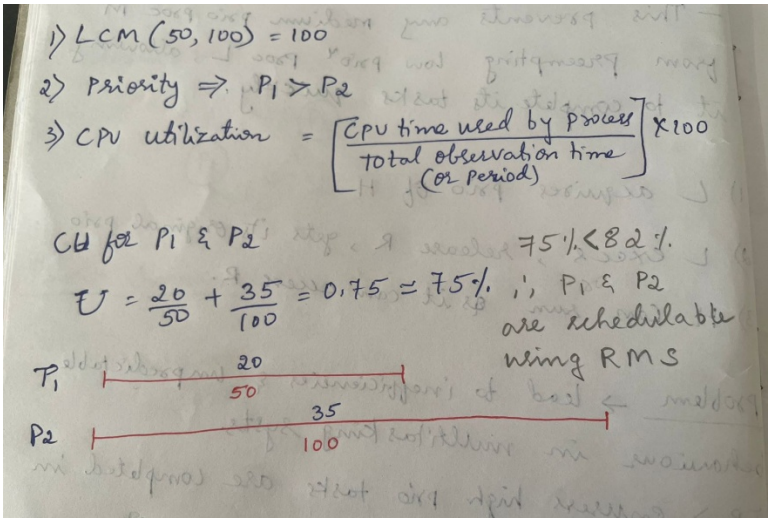
- For $n = 2$ tasks:

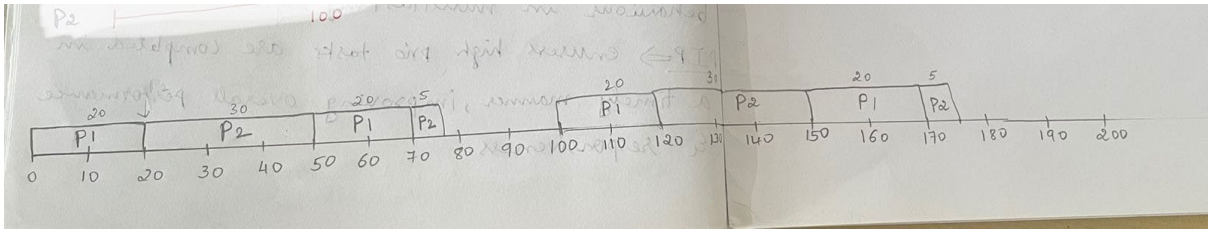
$$U_{max} = 2 \cdot (2^{1/2} - 1)$$

$$U_{max} = 2 \cdot (1.414 - 1) \approx 2 \cdot 0.414 = 0.828$$

This means the CPU utilization of the system must not exceed 82.8% to guarantee that all tasks are schedulable using RMS.







$U_{max} = n(2^{1/n} - 1)$

$= 2(2^{1/2} - 1)$

$= 0.82 \simeq 82\%$

Scheduling N procs (Formula)

(2)

Proc	Burst	Period
P_1	$t_1 = 25$	50
P_2	$t_2 = 35$	80

1) LCM (50, 80) = 400

2) Priority: $P_1 > P_2$

3) CPU utilization

$= \frac{25}{50} + \frac{35}{80} = 0.94$

94% > 82%

∴ P_1 & P_2 cannot be scheduled using RMS

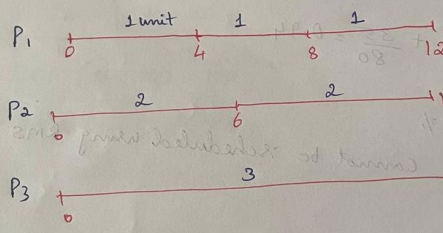
(3)

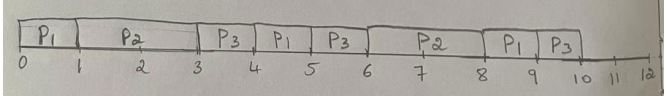
Process	Burst time	Period
P_1	1	4
P_2	2	6
P_3	3	12

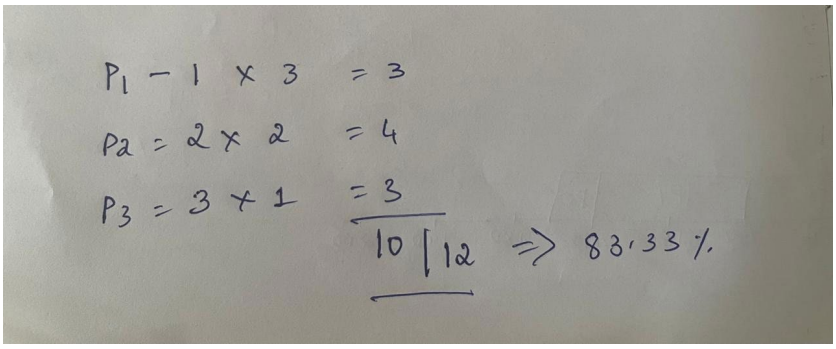
1) LCM (4, 6, 12) = 12

2) Priority ⇒ $P_1 > P_2 > P_3$

3) CPU utilization = $\frac{1}{4} + \frac{2}{6} + \frac{3}{12} = 0.83$







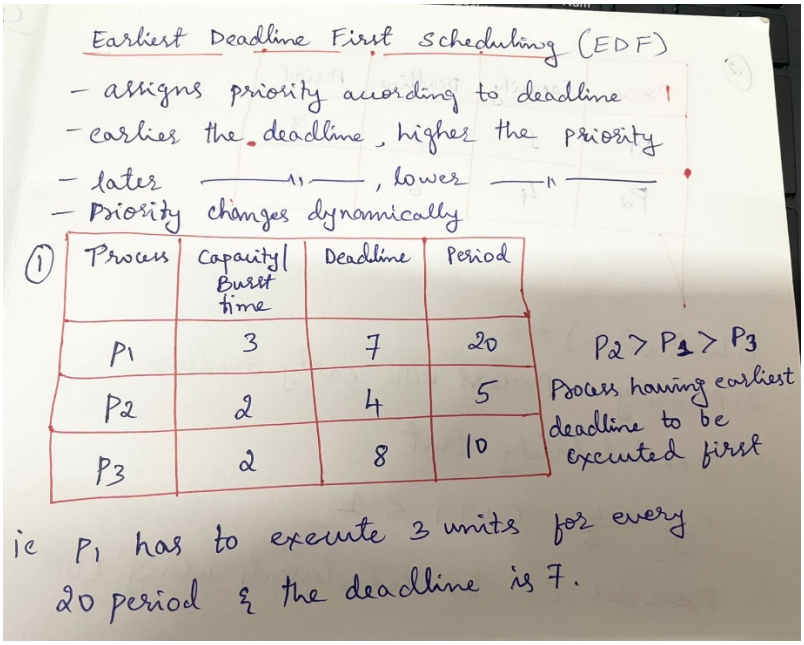
Differences Between Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF)

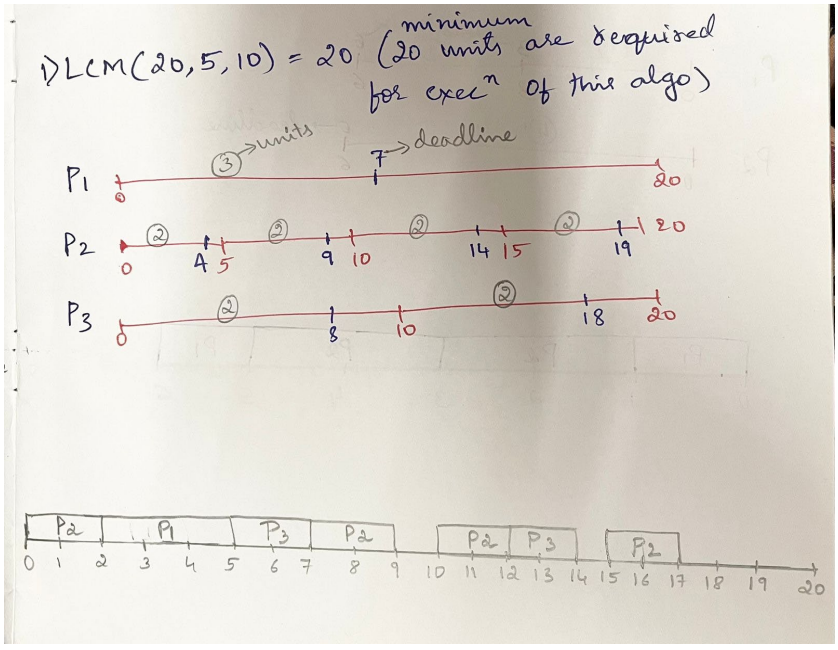
Aspect	Rate Monotonic Scheduling (RMS)	Earliest Deadline First (EDF)
Type of Scheduling	Static priority scheduling.	Dynamic priority scheduling.
Priority Assignment	Priority is assigned based on the period of tasks. Shorter periods get higher priority.	Priority is assigned based on the nearest deadline. Tasks with closer deadlines have higher priority.
Preemption	Preemptive. A higher-priority task can preempt a lower-priority task.	Preemptive. A task with a nearer deadline can preempt the running task.
Feasibility Test	Utilization bound: Scheduling is guaranteed if $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$, where C_i is execution time, T_i is period.	No fixed bound; works as long as total CPU utilization is ≤ 100%.
Flexibility	Limited flexibility. Only works well for periodic, fixed-priority tasks.	More flexible and can handle both periodic and aperiodic tasks.
Optimality	Optimal among static priority scheduling algorithms.	Optimal for preemptive, dynamic priority scheduling.
Complexity	Simpler to implement due to static priorities.	More complex due to frequent recalculation of priorities.
Overhead	Lower overhead as priorities are fixed and don't change at runtime.	Higher overhead because task priorities are dynamically adjusted.
Real-Time Guarantee	Guarantees schedulability only up to a certain utilization bound.	Guarantees schedulability for any set of tasks if total utilization is ≤ 100%.
Example Use Cases	Systems with periodic tasks and predictable workloads, e.g., embedded systems.	Systems with a mix of periodic and aperiodic tasks, e.g., multimedia or interactive systems.

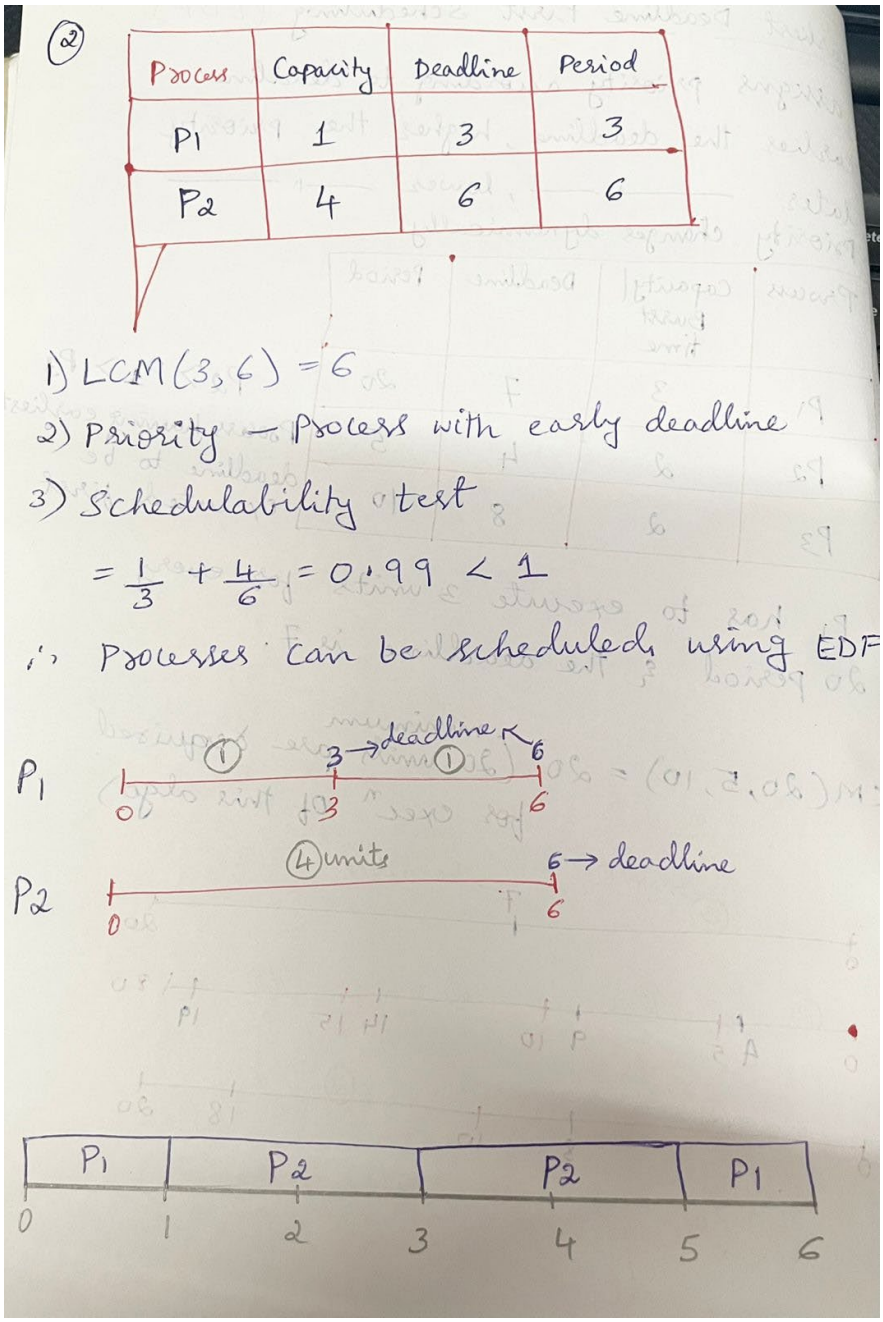
Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process.

https://www.youtube.com/watch?v=ejPXTOcMRPA






(2)
Process | Capacity | Deadline | Period
P1 | 1 | 3 | 3
P2 | 4 | 6 | 6
1) LCM(3, 6) = 6
2) Priority – Process with early deadline
3) Schedulability test
= 1/3 + 4/6 = 0.99 < 1
∴ Processes can be scheduled using EDF
P1
deadline
3
6
0
3
6
P2
4 units
6 → deadline
0
6
P1
P2
P2
P1
0
1
2
3
4
5
6