Real Time Systems

By

Dr. Rabih Ali Sbera

Department of Computer Engineering Techniques -3rd

2025-2026



tasks - interrupt
Process States
-Scheduling
Scheduling algorithms

Task

is a program under execution; it is the logical unit scheduled by the operating system.

Scheduling is the procedure of selecting which process the processor will execute at a given moment from among all the processes that are ready to run.

- Each process in the operating system is represented by a data structure (a **record**) that contains, at a minimum: the current state of the process (ready, blocked, executing, …), a process identity number, some additional descriptors (e.g., total execution time), and the set of resources (files, devices, …) allocated to that process.

A **real-time operating system (RTOS)** must provide three services for processes: **scheduling**, **dispatching**, and **intercommunication and synchronization** among processes.

Cyclic Executive

The **cyclic executive** is a system that is not driven by interrupts (i.e., it does not rely on interrupts for its operation). It gives the impression of parallel execution of several short processes on a fast processor. The system consists of an infinite loop that sequentially calls the processes, as illustrated in the following example:

```
for ( ; ; ) {   /* do forever */

    Process_1();

    Process_2();

    …

    Process_n();

}
```

Cyclic Executive

In this example, the processor time is distributed equally among all processes, since the loop cycles through them in a **round-robin** fashion. The previous loop can be easily modified to allocate longer execution times (and thus higher priority and faster response) to certain processes. For instance, in the following example, **Process_3** is executed twice, giving it twice the execution time of the other processes:

```
for ( ; ; ) {   /* do forever */

    Process_1();

    Process_2();

    Process_3();

    Process_3();

}
```

Interrupt Service Routines (ISR)

interrupt Service Routines (ISR)

It is important to understand how interrupts work when writing embedded system software, since real-time programs often need to handle interrupts generated by different system devices. The software engineer may either write a **device driver** from scratch (which, in this context, serves as the interrupt service routine), or modify and customize a **generic device driver** to suit the specific hardware for which it is written.

Interrupt Service Routines (ISR)

- There are two types of interrupts in any system:
- **Hardware Interrupts**: These are signals generated by peripheral devices and sent to the processor, which responds by executing the corresponding **Interrupt Service Routine (ISR)**. This routine, which is simply a small program, performs the necessary actions to service the interrupt. Hardware interrupts are the mechanism through which peripheral devices inform the processor that they require attention for some reason (e.g., an error occurred, an input/output operation completed, etc.).
- **Software Interrupts**: These are similar to hardware interrupts, but the interrupt is generated by special instructions invoked programmatically. This type of interrupt is usually used to call the services provided by the operating system (which are implemented as ISRs for these software interrupts).

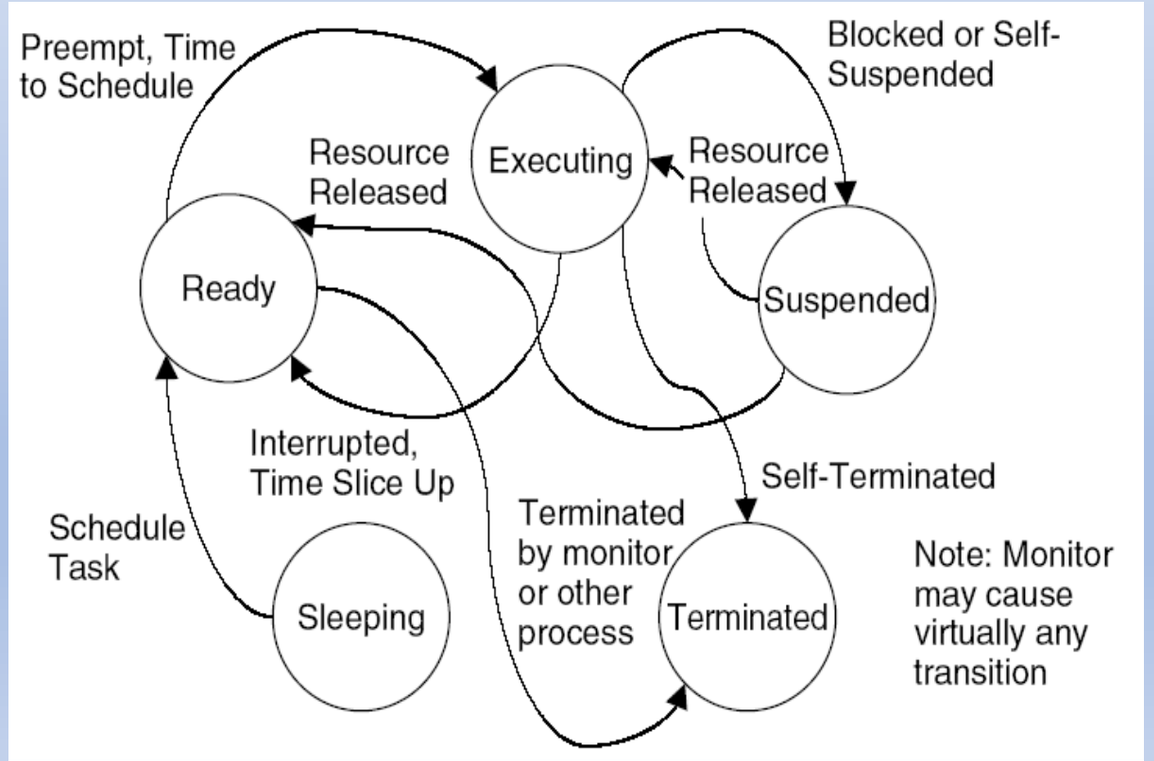
Process States

The operating system manages processes by storing their states in a control block. Typically, a process can be in one of the following four states:

- **Executing**
- **Ready**
- **Suspended (or Blocked)**
- **Dormant (or Sleeping)**

Resource Management

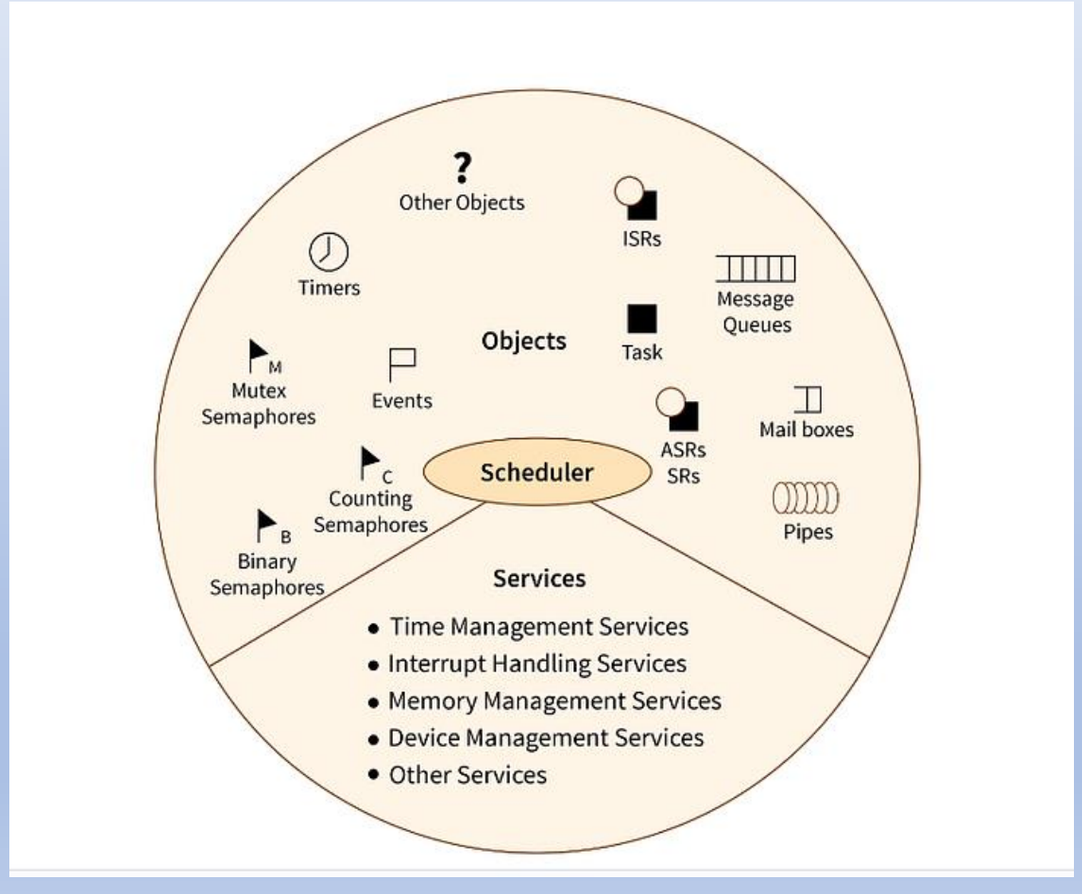
- The system periodically checks the state of processes in the **suspended queue**. If it finds a process that is suspended because it is waiting for a resource, and it determines that the resource has become available, the system changes the process state to **ready** and moves its control block to the **ready queue**.

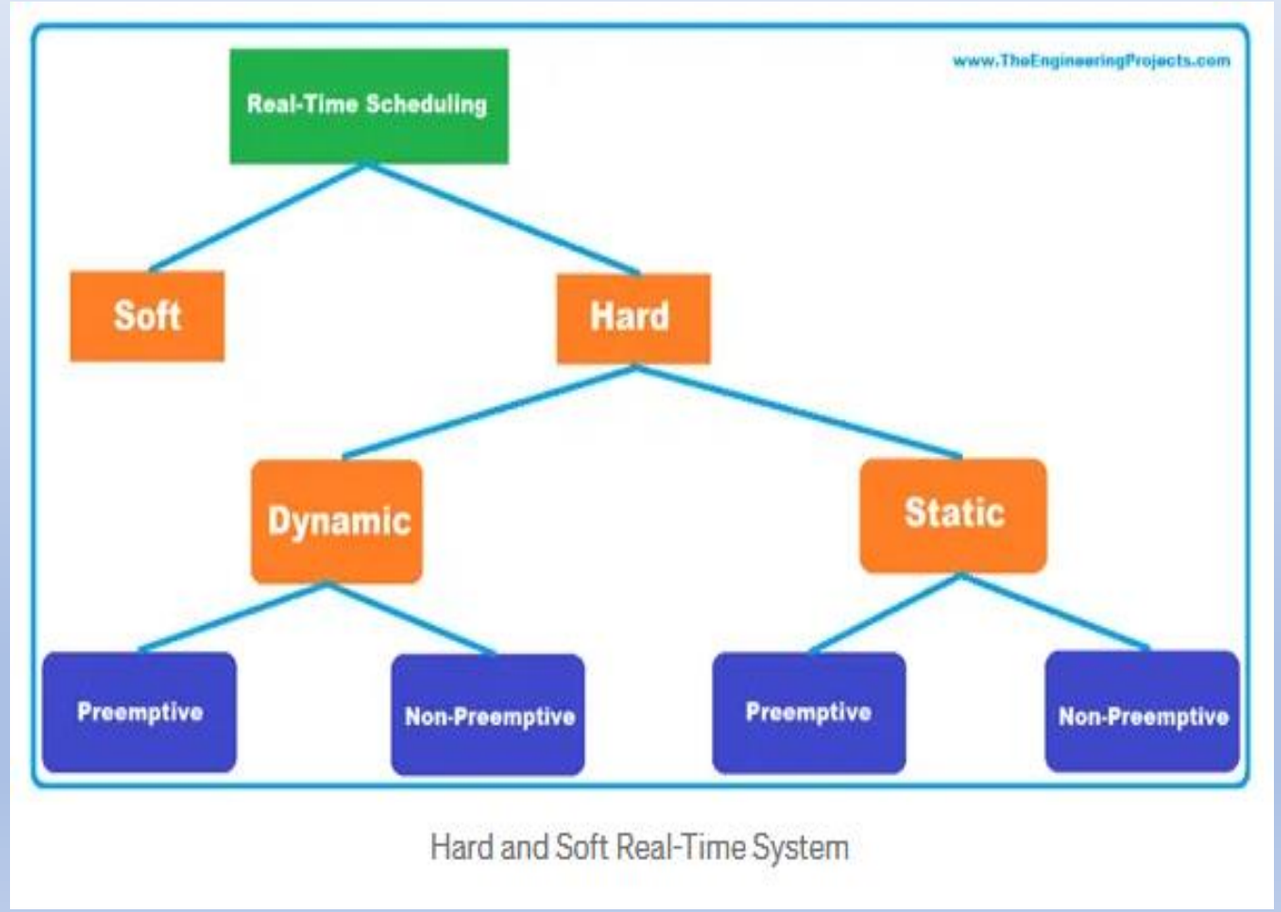


Process Scheduling

- **Scheduling** is considered one of the fundamental tasks of the operating system. To meet the timing requirements of a real-time program, it is necessary to use scheduling algorithms different from those employed in conventional operating systems. Special strategies are also required to organize the use of system resources by real-time processes, according to their priorities and timing constraints.
- This scheduling is performed **before the actual execution of the processes begins**. Such algorithms also attempt to minimize the cost of **context switching**, thereby increasing the likelihood of finding a schedule that satisfies all timing constraints.

Type of Real Time System





Hard and Soft Real-Time System

Non-Preemptive Scheduling

In non-preemptive scheduling, once a process starts using the CPU, it runs until it finishes or moves to a waiting state. The OS cannot forcibly take away the CPU.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



Non-Preemtive Scheduling

Preemptive Scheduling

✓ In preemptive scheduling, the operating system can interrupt a running process to allocate the CPU to another process usually due to priority rules or time-sharing policies. A process may be moved from Running → Ready state before it finishes.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



Preemptive Scheduling

Simplified Process Model

We will adopt a simplified model of a process in order to describe some systematic scheduling algorithms used in real-time systems. This model is based on the following assumptions:

- All processes in the system are **periodic**.
- The **relative deadline** of each process equals its period.
- All processes are **independent** of each other, i.e., there are no precedence constraints.
- No process contains a **non-preemptive section**, and the **context switching cost** is negligible.
- Each process requires **processor time only** (other requirements such as memory, I/O, etc., are neglected).
- In real-time systems, it is extremely important that the schedule generated by the scheduling algorithm is **predictable**, meaning that we can determine which process will execute at any given moment. Many real-time operating systems use a **round-robin algorithm**, which is simple and predictable.

Round-Robin Scheduling Algorithm

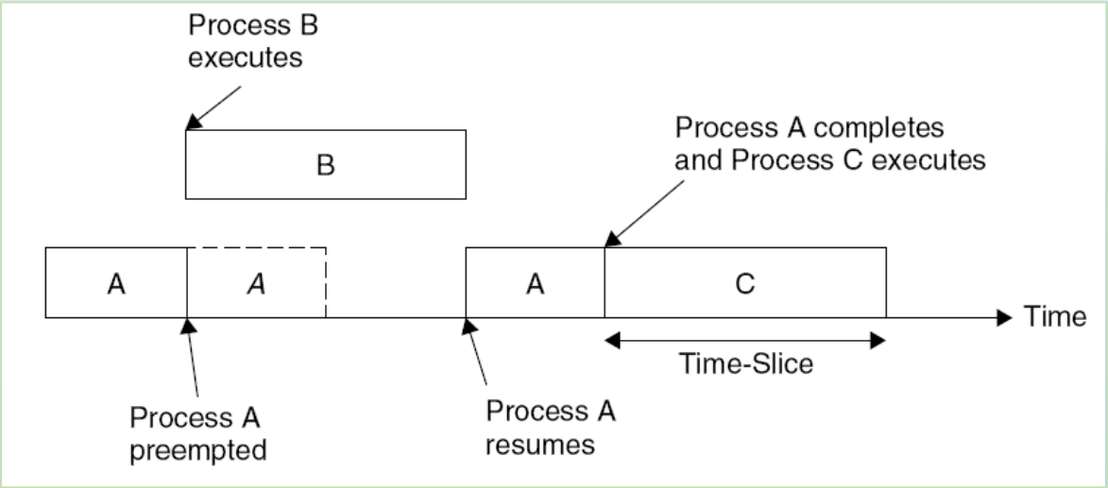
In this algorithm, processes are executed **sequentially** until completion and are often found in **cyclic executive systems**. Each process is allocated a fixed **time slice (quantum)** during which it executes. A timer with a fixed period equal to the time slice periodically interrupts the system.

A process continues executing until it either completes or its time slice expires, triggering a timer interrupt. If the process has not finished execution, its **context is saved**, and it is placed at the **end of the ready queue**. The system then retrieves the context of the next process at the **head of the queue** and continues its execution.

Thus, this algorithm provides a **fair, non-preemptive scheduling** for processes of the same priority by distributing processor time equally among them.

- The concept of **preemptive priority** can be introduced into the round-robin algorithm by allowing higher-priority processes to **preempt** the processor if it is currently executing a lower-priority process upon their arrival. The following figure illustrates this concept.

Round-Robin Scheduling Algorithm



Suppose process **A** has a higher priority. Processes **B** and **C** have the same priority. Initially, process **B** begins execution and continues until its time slice expires. Then, process **A**, having higher priority, **preempts B** and executes until the end of its time slice. Next, process **C** executes until its time slice expires, and finally execution returns to process **A**. This cycle continues in the same manner

Example: Preemptive Round-Robin Scheduling

Process	Priority	Execution Time	Time Slice (Quantum)
A	High	6 ms	2 ms
B	Medium	4 ms	2 ms
C	Medium	3 ms	2 ms

Scheduling sequence:
At **time 0 ms**, **B** starts execution (since A hasn't arrived yet) and executes for **2 ms**.
At **time 2 ms**, **A** arrives. Since A has **higher priority**, it **preempts B** and executes for **2 ms** (time slice).
At **time 4 ms**, **B** resumes execution and continues for **2 ms** (remaining time 0 ms).
At **time 6 ms**, **C** executes for **2 ms**.
At **time 8 ms**, **A** executes for the next **2 ms** (finishing its total execution time).
At **time 10 ms**, **C** executes for **1 ms** (finishing its total execution time).
Timeline (ms)
Time: 0 2 4 6 8 10 11
B | A | B | C | A | C

Rate Monotonic (RM) algorithm

Algorithm Concept

In systems with fixed priorities, each task is assigned a specific priority relative to the other tasks. It is considered one of the most important scheduling algorithms in these systems, providing optimal scheduling in the RM (Rate Monotonic) model.

In the simplified task algorithm mentioned earlier, each task is assigned a priority inversely proportional to its period: the task with a longer period is given a lower priority than the task with a shorter period.

- We will explain how this algorithm works using the following example. Suppose we have the following set of tasks:

τ_i	e_i	p_i	$u_i = e_i/p_i$
τ_1	1	4	0.25
τ_2	2	5	0.4
τ_3	5	20	0.25

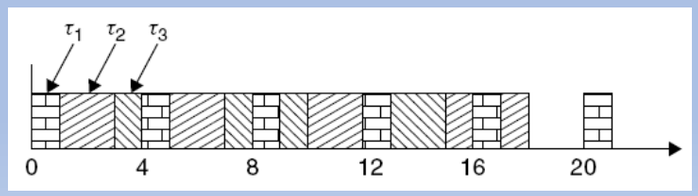
Rate Monotonic (RM) algorithm

Assume that the execution time of all three tasks is 0. The following figure illustrates the scheduling of this system using the RM (Rate Monotonic) algorithm.

- Since Task 1 (τ1) has the shortest period, its priority is higher than that of the other tasks, so it is executed first. Notice that at time 4, the second instance of Task 1 (τ1) arrives. Because it has the highest priority, it preempts Task 3 (τ3).

Rate Monotonic (RM) Scheduling
The schedulability condition of RM is:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

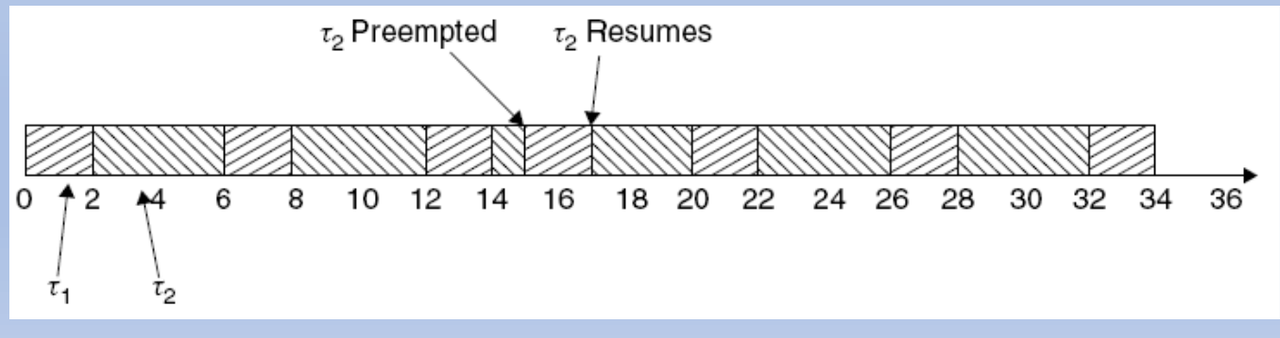


Earliest-Deadline-First Algorithm (EDF)

- Unlike static priority systems, in dynamic priority systems the priorities of tasks change during system operation. One of the most well-known algorithms of these systems is the **EDF algorithm**, which relies on dynamic priorities. EDF assigns priorities to tasks based on their **deadline** rather than their execution times. At any moment, the task with the **closest deadline** has the **highest priority**.

The following theorem provides the condition that must be satisfied for a set of tasks to be **schedulable under the EDF algorithm**

τ_i	p_i	e_i
τ_1	5	2
τ_2	7	4



Rate Monotonic (RM) algorithm

- **Earliest Deadline First (EDF) Scheduling**
- The schedulability condition of EDF is based on processor utilization:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$$

Where:

- e_i: execution time of task i
- p_i: period of task i
- n: number of tasks
- U: total CPU utilization

If U≤1, all tasks are guaranteed to meet their deadlines under EDF.