Real-Time Operating System (RTOS)

Unit 3

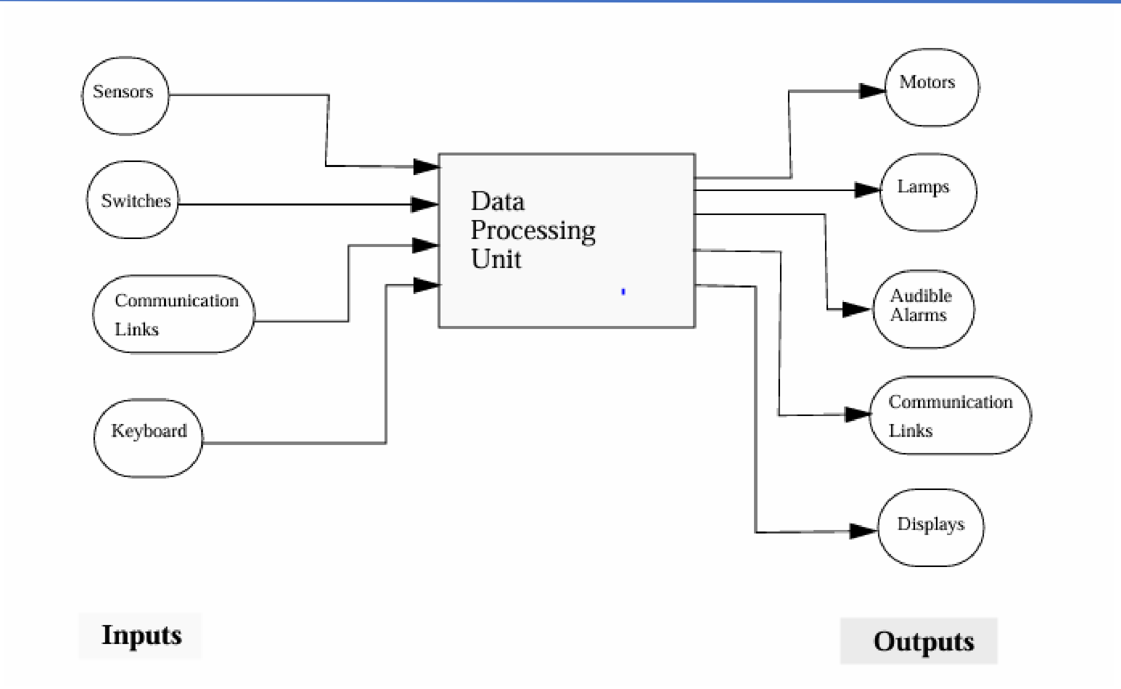
3 Concepts of Real-Time Systems

In today's advanced computing systems, where many operations run under strict time constraints, even the slightest delay can have serious consequences. From autonomous vehicles to medical devices monitoring a patient's vital signs, and industrial robots performing precise tasks on an assembly line, these systems cannot afford to be slow or unreliable. They must operate with precision, responding swiftly and accurately in real-time to ensure safe and efficient performance. But what enables these systems to function so seamlessly in high-pressure environments? This is where real-time systems come into play. Although the term real-time has been well defined, it is still being misused and misinterpreted. One of the most frequent misrepresentations is that "real-time" simply implies that the system operates quickly. While speed may be a component of some real-time systems, it is not the defining factor. A system is considered real-time if it can respond to events or stimuli within a predetermined deadline, which could be microseconds, seconds, or even minutes depending on the system.

Definition: *In the simplest form, a real-time system is defined as one where the system output's correctness is dependent not only on producing the right result but also on delivering it at the right time. If a system generates the correct result either too early or too late relative to a specified time limit, it may no longer be considered a real-time system.*

For a system to be considered real-time, it must respond to external events promptly, with the response time guaranteed to meet strict deadlines.

To better understand the principles of real-time systems let's consider the example of an air traffic control system, where precise timing and immediate response to incoming data are critical for maintaining safety and efficient flight operation. *Figure below show a simplified block diagram of an example of air-traffic control system. Inputs may come from various sensors like radar sensors, communication inputs and weather sensors whereas outputs may be in the form of flight commands and alerts.*



Inputs:

- **Radar Sensors**: Track the positions and speeds of aircraft.
- **Communication Inputs**: Receive flight plans and updates from pilots.
- **Weather Sensors**: Provide real-time weather information.

System Operation

An Air Traffic Control (ATC) system is crucial for ensuring the safety and efficiency of air travel. It monitors the real-time positions of all aircraft in a specific airspace using radar sensors. The ATC personnel analyze this data alongside communication inputs from pilots, who report their status and any changes in their flight plans.

When an aircraft deviates from its assigned path, encounters unexpected weather, or approaches another aircraft too closely, the ATC system must respond immediately:

- The system provides timely flight commands to pilots to ensure safe distances between aircraft.
- If adverse weather is detected, alerts are issued to reroute flights or delay takeoffs.

Time Constraints:

- **Response Time**: The ATC system must process incoming data from radar and communications and provide flight commands within a few seconds to ensure safety. For instance, if two aircraft are on a collision course, the system might have only 10-15 seconds to issue a corrective command to avoid an accident.

- **Deadline**: The critical deadlines in this system are based on the aircraft's altitude changes, landing sequences, and adherence to air traffic regulations. Each command issued to pilots must be completed before the aircraft reaches specific waypoints or altitude levels.

If a plane's radar indicates it's too close to another aircraft, the ATC system must respond within a strict deadline (e.g., 10 seconds) to reroute the aircraft and maintain safe separation.

Outputs:

- **Flight Commands**: Instructions to pilots for altitude changes, course adjustments, and landing sequences.
- **Alerts**: Warnings for potential collisions or adverse weather conditions.

This example exemplifies a real-time system where timely responses to inputs—such as aircraft position data and weather conditions—are crucial for maintaining safety and efficiency; the system must produce outputs, like flight instructions and alerts, within strict deadlines to ensure the seamless management of air traffic.

3.1 Classification of Real-time System

Based on timing constraints and operational requirements, real-time systems can be classified into three types: hard real-time systems, soft real-time systems, and firm real-time systems.

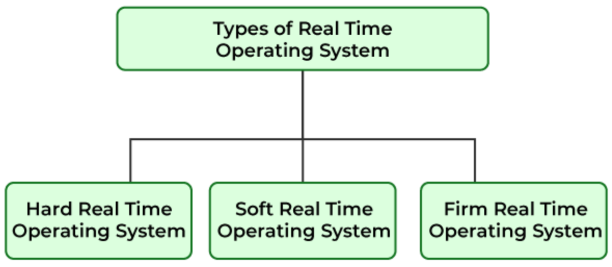

Types of Real Time Operating System
Hard Real Time Operating System
Soft Real Time Operating System
Firm Real Time Operating System

Figure 3-1: Types of Real-time operating systems

1. Hard Real-Time Systems

Hard real-time systems are those in which failing to meet a deadline can lead to catastrophic consequences, such as loss of life, significant property damage, or mission failure. These systems require absolute adherence to timing constraints, and even a single missed deadline is unacceptable.

Examples:

- **Aircraft control systems**: Ensuring safe flight operations.
- **Medical devices**: Such as pacemakers and life support systems.
- **Industrial control systems**: Managing critical infrastructure like power plants.

2. Soft Real-Time Systems

Soft real-time systems can tolerate some deadline misses without severe consequences. The system's performance degrades gradually with each missed deadline, rather than failing completely. The primary goal is to optimize performance and ensure most deadlines are met.

Examples:

- **Multimedia systems**: Streaming audio and video where occasional delays or quality degradation are acceptable.
- **Online transaction systems**: Like booking and reservation systems where slight delays are tolerable.
- **Telecommunications**: Network traffic management where delays can affect service quality but not critically.

3. Firm Real-Time Systems

Firm real-time systems lie between hard and soft real-time systems. Missing a deadline in these systems does not cause catastrophic failure but makes the result useless. Repeated misses, however, may lead to unacceptable performance degradation.

Examples:

- **Financial trading systems**: Where outdated information can lead to losses.

- **Inventory management systems**: Where timely updates are crucial but not life-threatening.
- **Networked data acquisition systems**: Collecting data where delayed information loses its relevance.

3.2 Introduction to Real-Time Operating Systems

A Real-Time Operating System (RTOS) is essential to the functioning of many modern embedded systems, providing a structured software platform to develop, manage, and schedule tasks effectively. While an RTOS ensures that time-critical operations occur within specific deadlines, not every embedded system requires one. Systems with simple hardware configurations or minimal software code—such as basic sensors or control units—can often function without the complexity of an RTOS. However, as the size and complexity of embedded software increase, so does the need for advanced task scheduling, resource sharing, and real-time responsiveness. In such systems, an RTOS is vital to managing multiple tasks simultaneously, ensuring that real-time constraints are met, and efficiently handling the interaction between hardware and software components. This is especially true in embedded applications like automotive systems, industrial control, and healthcare devices, where precise timing and system stability are critical for success.

Definition: *A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code.*

Application code designed on an RTOS can be quite diverse, ranging from a simple application for a digital stopwatch to a much more complex application for aircraft navigation. Good RTOSes, therefore, are scalable in order to meet different sets of requirements for different applications.

3.2.1 Key Characteristics of RTOS

In the world of embedded systems, where precise timing and reliability are critical, the need for an operating system that can handle multiple tasks with guaranteed performance becomes paramount. This is where Real-Time Operating Systems (RTOS) shine. RTOS provides the backbone for systems that cannot afford delays or unpredictability, ensuring tasks are completed within strict time constraints. To truly understand why RTOS is so crucial in these environments, it's important to explore its key characteristics that make it stand out. Some of the more common key characteristics are discussed below

1. **Deterministic Timing**: An RTOS guarantees that tasks are completed within specific time constraints, ensuring predictable and consistent behavior. This is crucial for systems that must meet strict deadlines, such as in medical devices or automotive control systems.
2. **Multitasking**: RTOS enables the concurrent execution of multiple tasks by effectively managing system resources like CPU, memory, and I/O devices. It ensures that tasks are prioritized based on their urgency.
3. **Minimal Latency**: RTOS minimizes the delay between the occurrence of an external event and the system's response. This responsiveness is vital for applications like robotics, where quick reaction times are essential.
4. **Real-Time Priority Levels**: It supports multiple levels of task priorities, allowing more urgent tasks to preempt less important ones. This ensures that critical tasks meet their deadlines without being delayed by non-essential tasks.
5. **Memory Management**: Efficient memory management in RTOS ensures that resources are allocated and freed dynamically, with minimal fragmentation, allowing real-time tasks to execute without memory-related delays.
6. **Reliability and Fault Tolerance**: RTOS is designed for high reliability and often includes mechanisms for error handling, fault recovery, and ensuring continuous operation even under failure conditions, crucial for mission-critical systems.
7. **Scalability:** An RTOS should adapt to various embedded systems by scaling up or down. It must support modular additions or deletions, such as file systems or protocol stacks. A scalable RTOS allows reuse across different projects, saving development time and costs.

3.2.2 Popular RTOS Platform in Embedded Domain

Originally, real-time operating systems (RTOSs) were predominantly used in military, aerospace, and high-end industrial control applications. However, with the advancement of affordable and powerful computing hardware, RTOS usage has expanded significantly into traditional embedded systems and IoT applications. Today’s RTOS platforms are more user-friendly, offering extensive libraries of development tools and application-specific stacks. Modern RTOS platforms have also evolved to include specialized architectures and features, tailored to specific applications or platforms, making them indispensable in fields such as automotive, healthcare, and consumer electronics. Some of the most popular RTOS platforms widely utilized in the embedded domain are discussed below.

1. FreeRTOS

FreeRTOS is a widely acclaimed real-time operating system (RTOS) kernel designed for microcontrollers and small microprocessors in embedded devices, focusing on reliability and user-friendliness. Written primarily in the C programming language, it utilizes minimal assembly language instructions, mainly for architecture-specific scheduler routines. This design choice simplifies the process of porting and maintaining FreeRTOS across various processor architectures, making it a versatile option for developers.

Currently, FreeRTOS supports over 15 toolchains and more than 40 microcontroller architectures, including modern options like RISC-V and ARMv8-M (such as the Arm Cortex-M33 series). This extensive compatibility ensures that developers can implement FreeRTOS in a wide range of applications, from simple consumer electronics to complex industrial systems.

To manage multiple threads, FreeRTOS employs a thread tick method that the host program invokes at regular short intervals, typically between 1 and 10 milliseconds. This method enables task switching based on priority and a round-robin scheduling scheme. The timing interval can be adjusted to meet the specific needs of different applications, providing flexibility in system design.

Distributed freely under the MIT open-source license, FreeRTOS not only includes a robust kernel but also boasts an expanding library of resources tailored for IoT applications. These resources encompass drivers, security updates, and application-specific add-ons, all maintained by Amazon Web Services (AWS) to benefit the FreeRTOS community.

For developers seeking commercial support, two licensed variants of FreeRTOS are also available:

- **OpenRTOS**: This commercially licensed version of the FreeRTOS kernel includes indemnification and dedicated support, while sharing the same code base as FreeRTOS.

- **SAFERTOS**: This derivative version of FreeRTOS has undergone rigorous analysis, documentation, and testing to comply with stringent industrial (IEC 61508 SIL 3), medical (IEC 62304 and FDA 510(K)), automotive (ISO 26262), and other international safety standards. SAFERTOS includes independently audited safety lifecycle documentation artifacts, making it an excellent choice for safety-critical applications.

With its rich feature set, strong community support, and emphasis on safety and reliability, FreeRTOS continues to be a preferred choice for developers working on embedded systems across various industries.

2. VxWorks

VxWorks, developed by Wind River and first released in 1987, is recognized as one of the earliest real-time operating systems (RTOS) on the market. Originally designed for military and aerospace applications, VxWorks has evolved to provide robust board support packages (BSPs) and software stacks that meet stringent industry certification standards, including DO-178C, IEC 61508, IEC 62304, and ISO 26262. Its reliability and compliance have made it a preferred choice for critical applications in various sectors.

VxWorks has been instrumental in automating systems for the Air Force, including drone operations and in-air refueling systems. Its legacy extends to space exploration, where it has powered missions like the Mars Insight Lander and the Mars Curiosity Rover, showcasing its ability to perform under extreme conditions.

Designed for deterministic, priority-based preemptive scheduling, VxWorks delivers high reliability with low latency and minimal jitter. Its scheduling mechanisms include:

- **Priority-Based Preemption**: Allows for immediate response to critical tasks, with optional round-robin scheduling for lower-priority tasks.
- **Time and Space Partitioning**: Ensures resource allocation and isolation between different tasks.
- **Adaptive Scheduling**: Offers both foreground and background threading to efficiently manage tasks based on their urgency.

- **POSIX PSE52 Thread Scheduling Extensions**: Supports FIFO (First In, First Out) and sporadic scheduling to accommodate various application requirements.

VxWorks is compatible with a wide range of microcontroller units (MCUs) based on AMD/Intel, POWER, Arm, and RISC-V architectures. It can be utilized in multicore asymmetric multiprocessing (AMP), symmetric multiprocessing (SMP), and mixed-mode applications, providing flexibility for developers.

To enhance reliability and security, VxWorks incorporates several architectural features, including:

- **Secure Boot**: Ensures that only digitally signed images are executed during the boot process.
- **Secure ELF Loader**: Loads only digitally signed applications, preventing unauthorized code execution.
- **Kernel Hardening**: Protects against various attack vectors with measures like non-executable pages and stack smashing protection.
- **Secure Storage**: Provides encrypted containers and full disk encryption for sensitive data.
- **Address Sanitizer (ASan)**: Helps detect memory corruption bugs, improving software reliability.

The latest version of VxWorks supports modern development languages, including C++, Boost, Rust, Python, and pandas.

3. Other RTOS Options (RTLinux, Micrium, QNX)

In addition to FreeRTOS and VxWorks, several other real-time operating systems (RTOS) have gained recognition in the embedded systems domain, each offering unique features and capabilities tailored to specific application needs. Below, we explore three notable alternatives: RTLinux, Micrium, and QNX.

RTLinux

RTLinux is an extension of the Linux kernel that allows for real-time processing alongside traditional non-real-time tasks. By implementing a dual-kernel approach, RTLinux separates real-time tasks from standard Linux processes, ensuring that critical tasks can be executed with guaranteed response times. This makes RTLinux an attractive option for applications that require both real-time performance and the extensive capabilities of the Linux ecosystem. It is commonly used in telecommunications, robotics, and industrial automation where multitasking and real-time responsiveness are essential.

Micrium

Micrium is a commercial RTOS designed specifically for microcontrollers and small embedded systems. Known for its simplicity and modularity, Micrium provides a lightweight footprint, making it ideal for resource-constrained devices. Its features include a powerful multitasking kernel, a comprehensive set of communication protocols, and a graphical user interface (GUI) library, which facilitate rapid application development. Micrium is particularly popular in industries like automotive, medical devices, and consumer electronics, where reliable performance and ease of integration are critical.

QNX

QNX is a commercial RTOS known for its microkernel architecture, which enhances system reliability and security. By minimizing the amount of code running in the kernel, QNX reduces the potential for system failures. It supports advanced features such as symmetric multiprocessing (SMP), fault tolerance, and real-time scheduling, making it suitable for high-stakes environments like automotive (e.g., in-car infotainment systems), medical devices, and industrial automation. QNX is highly regarded for its robustness and is often employed in mission-critical applications where system uptime is paramount.

3.2.3 General Structure of RTOS

A Real-Time Operating System (RTOS) is designed to manage hardware resources, run multiple tasks concurrently, and ensure that critical tasks meet their deadlines. The general structure of an RTOS can be divided into several key components: the application layer, the RTOS kernel, board support packages (BSP), and hardware.

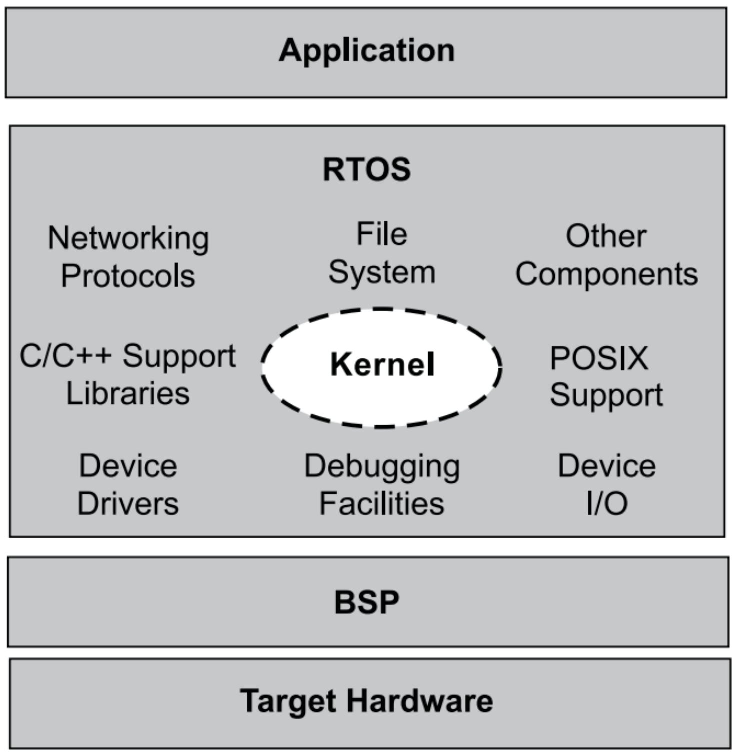


Figure 3-2:High-level view of an RTOS

1. **Application Layer**

The application layer consists of the user-defined tasks, processes, and functions that perform the actual work of the embedded system. This is where the developer writes code specific to the application's requirements. The application layer interacts with the RTOS kernel to schedule tasks, handle events, and manage resources.

2. **RTOS Kernel**

The kernel is the core component of an RTOS, responsible for task management, scheduling, and resource allocation. It ensures that tasks are executed in a deterministic manner according to their priorities and timing requirements.

3. Board Support Packages (BSP)

A Board Support Package (BSP) is a collection of software components that provide an interface between the RTOS kernel and the specific hardware of the target board. The BSP is crucial for abstracting hardware details and ensuring the RTOS can run on different hardware platforms. BSP includes Low level routines like device drivers, Initialization code and Hardware Abstraction layers.

4. Hardware

The hardware consists of the physical components on which the RTOS and applications run. It includes the central processing unit (CPU), memory, and various peripherals.

4 Common Components in an RTOS kernel

RTOS can be a combination of various modules, including the kernel, a file system, networking protocol stacks, and other components required for a particular application. Although many RTOSes can scale up or down to meet application requirements, the common element at the heart of all RTOSes contain the following components:

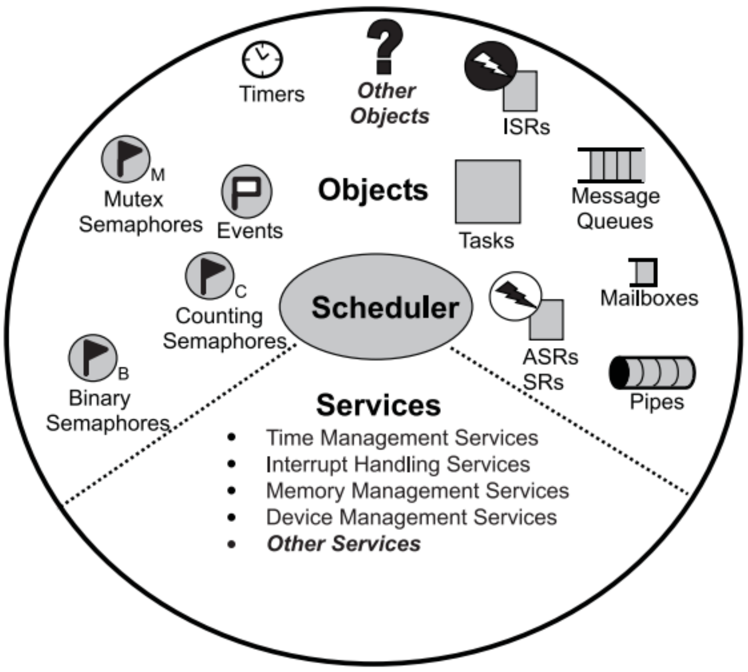


Figure 4-1: Common Components in an RTOS kernel

- **Scheduler**—is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling.

- **Objects**—are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.
- **Services**—are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.

5 Managing Tasks in RTOS: Scheduling, Switching, and Synchronization

In the realm of real-time operating systems (RTOS), effective task management is essential for ensuring that tasks are completed on time and in the right sequence. Consider an air traffic control system, where numerous flights are constantly monitored, each needing immediate attention to ensure safety. Here, tasks such as radar scanning, flight communication, and emergency handling must be executed precisely when needed. In a similar fashion, embedded systems demand the seamless execution, switching, and synchronization of tasks to meet their stringent real-time constraints. This is where **Task Scheduling**, **Context Switching**, and **Task Synchronization** come into play, forming the backbone of efficient RTOS operation. Without these mechanisms, the smooth and predictable performance of embedded systems would be compromised, making them unreliable for critical applications.

5.1 Task Scheduling: Deciding What Happens and When

Task scheduling in a real-time operating system (RTOS) is a critical process that determines which task should run next based on its priority, deadlines, and the availability of system resources. The primary goal of scheduling is to ensure that tasks meet their timing requirements, as even a minor delay in executing a time-sensitive task can lead to serious consequences in real-world applications, such as industrial automation or medical devices.

Scheduling in RTOS is governed by algorithms designed to manage task execution efficiently. Broadly, these algorithms fall into two categories: Preemptive and Non-preemptive scheduling.

1. **Preemptive Scheduling**

   In preemptive scheduling, the RTOS can interrupt a currently running task if a higher-priority task becomes ready to execute. This ensures that critical tasks are given immediate attention, allowing them to meet their deadlines, even if lower-priority tasks are already running. For example, in an automotive system, a task monitoring engine temperature (high priority) can preempt a task managing the car's infotainment system (lower priority).

   Preemptive scheduling is particularly suitable for systems where tasks have strict timing constraints and require rapid responsiveness. However, frequent task switching can introduce context-switching overhead, which needs to be minimized for optimal system performance.

2. **Non-Preemptive Scheduling**

   Non-preemptive scheduling allows tasks to run to completion once they have started. In this approach, the RTOS does not interrupt a running task, even if a higher-priority task becomes ready. This method is useful for tasks that need uninterrupted execution, such as writing critical data to non-volatile memory.

While non-preemptive scheduling ensures that a task completes without interference, it can lead to problems if a lower-priority task monopolizes the CPU for too long. In such cases, higher-priority tasks may miss their deadlines, making this approach less suitable for systems with strict real-time requirements.

Moreover, RTOS task scheduling algorithms are complex and must be carefully designed to meet the specific needs of the real-time system. Some common RTOS task scheduling algorithms include:

1. Preemptive Priority-Based Scheduling

This is one of the most widely used scheduling algorithms in real-time kernels. Each task is assigned a priority, and the task with the highest priority is executed first. If a higher-priority task becomes ready while a lower-priority task is running, the kernel immediately preempts the lower-priority task, saves its context in its Task Control Block (TCB), and switches to the higher-priority task.

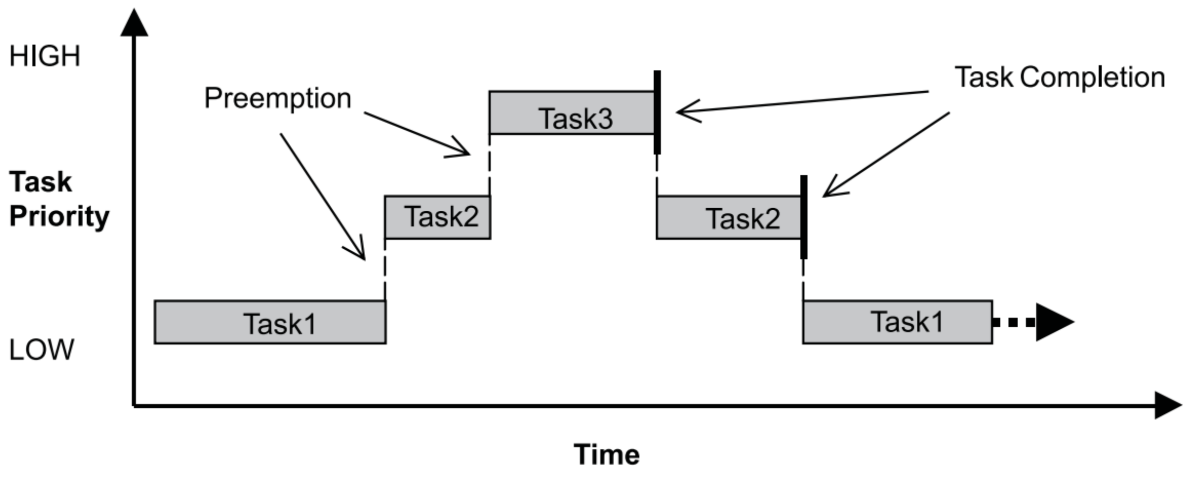


Figure 5-1: Preemptive priority-based scheduling

For example, as shown in Figure 3-5, task 1 is interrupted by task 2 (with a higher priority), which is further preempted by task 3 (the highest priority). After task 3 completes, task 2 resumes execution, followed by task 1.

2. Preempt-Round-Robin Scheduling

Round-robin scheduling ensures that tasks share the CPU time equally. In this approach, each task is given a fixed time slice to execute before the CPU switches to the next task in the queue. While pure round-robin scheduling is unsuitable for real-time systems due to its lack of prioritization, it can be effectively combined with preemptive priority-based scheduling. This hybrid approach applies time slicing among tasks of the same priority, ensuring fairness without compromising the responsiveness of higher-priority tasks.

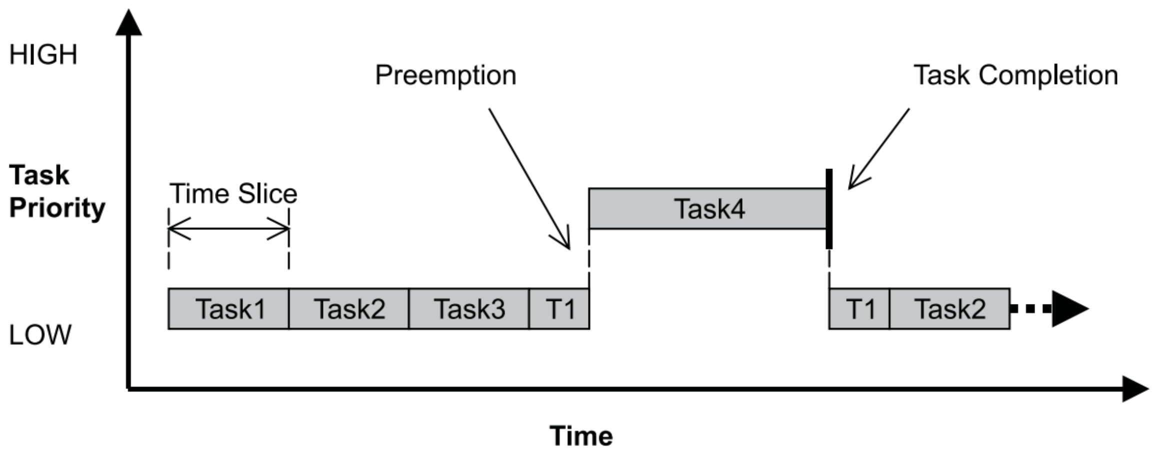


Figure 5-2: Round-robin with augmented preemptive scheduling.

For instance, higher-priority tasks are always executed first. If multiple tasks share the same priority level, they are scheduled in a round-robin manner, allowing equal CPU allocation. This method balances priority-based responsiveness with equitable resource sharing.

5.2 Context Switching: Ensuring Seamless Transitions

In a multitasking system, context switching is a vital process that allows the system to alternate between tasks without losing the progress of any individual task. This ensures that real-time systems maintain responsiveness while managing multiple concurrent operations. At its core, context switching involves saving the state of the currently running task and restoring the state of the next task scheduled to run.

A context switch occurs when the scheduler decides to stop running one task and start executing another. Each task in an RTOS operates with its own context, which includes:

- The **program counter** (indicating the next instruction to execute).
- **CPU register values** (representing the current state of computation).
- Other state information critical for resuming the task seamlessly.

When a context switch is triggered by the kernel's scheduler, the following steps unfold:

1. **Saving the Current Task's State**
   The RTOS saves the context of the currently active task into its **Task Control Block (TCB)**—a dedicated data structure that maintains all task-specific information. This context includes critical data such as CPU registers, the program counter, and stack pointers, ensuring that the task can be resumed later without any disruption.
2. **Loading the Next Task's Context**
   The scheduler retrieves the context of the next task from its TCB and loads it into the CPU. This task now becomes the active thread of execution, ready to carry out its assigned operations.
3. **Task Resumption**
   The previously active task's context is temporarily "frozen" within its TCB. When the scheduler selects it to run again, the task resumes from the exact point where it was interrupted, ensuring no loss of data or execution flow.

A typical context switch scenario is illustrated below.

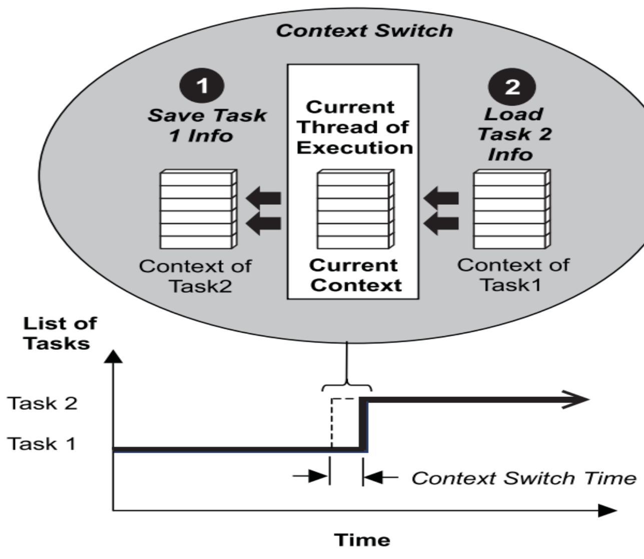


Figure 5-3:Multitasking using a context Switch

As shown in Figure, when the kernel's scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:

1. The kernel saves task 1's context information in its TCB.

2. It loads task 2's context information from its TCB, which becomes the current thread of execution.
3. The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch.

The time it takes for the scheduler to switch from one task to another is the **context switch time** and the **dispatcher** is the part of the scheduler that performs context switching and changes the flow of execution.

5.3 Task Synchronization: Coordinating Task Operations

In a real-time operating system (RTOS), task synchronization is a crucial concept that ensures the smooth and coordinated execution of tasks. *A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time*. Tasks may need to execute concurrently to meet real-time system requirements, but when multiple tasks share resources or need to operate in a particular order, synchronization becomes essential.

5.3.1 The Challenge of Task Synchronization

When tasks run concurrently, they often need to access shared resources or execute in a specific sequence to achieve the desired outcomes. Without proper synchronization, tasks may interfere with each other, leading to issues such as:

Data corruption: When two tasks access the same resource simultaneously, one task might overwrite the data that another task is working on, causing inconsistency.
Race conditions: A race condition occurs when the outcome of task execution depends on the order in which tasks are executed. If the order is unpredictable, the result may vary, leading to unintended behavior or errors.
Deadlock: Deadlock happens when tasks wait indefinitely for resources that are locked by other tasks. This results in a system freeze or a significant performance bottleneck.
Priority inversion: This occurs when a lower-priority task holds a resource needed by a higher-priority task, causing the higher-priority task to wait unnecessarily.

To address these synchronization problems, synchronization mechanisms are used to ensure that tasks can safely share resources and execute in the correct order. One of the most common synchronization objects in RTOS is the semaphore.

6 Managing Resource Access: Resource Sharing, Deadlock and Priority Inversion

In real-time operating systems (RTOS), resource access management is crucial for ensuring smooth operation when multiple tasks or processes share common resources. Without proper synchronization, these tasks can interfere with each other, causing issues such as deadlock, priority inversion, and resource contention.

6.1 Resource Sharing: The Foundation of Task Coordination

Resource sharing is an essential aspect of multitasking systems, where tasks or processes need to access common system resources like memory, devices, and data structures. While resource sharing is efficient, it also introduces the risk of **resource contention** and **conflicts** if multiple tasks try to access the same resource at the same time.

6.1.1 Resource Sharing challenge without synchronization

Consider the case where two tasks, Task A and Task B, need to write data to the same UART interface. Task A sends the message "11111," while Task B sends the message "2222."

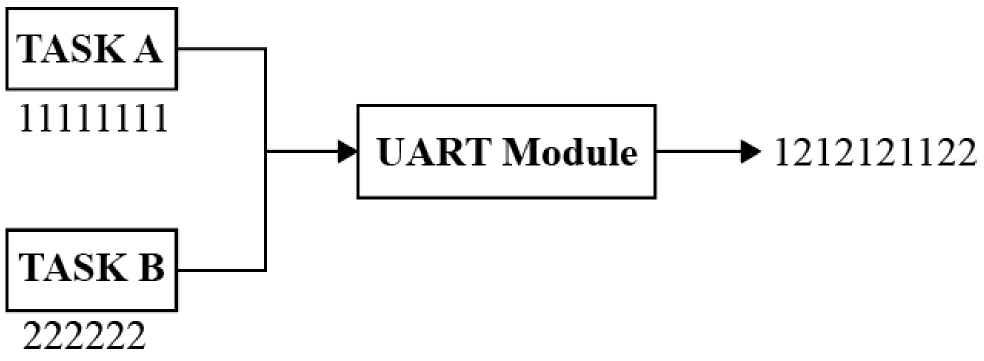


Figure 6-1: Corrupted Data

Without proper synchronization, both tasks may attempt to write to the UART interface at the same time. Since the UART can only handle one operation at a time, this results in data corruption. The messages from Task A and Task B would overlap, leading to garbled output, as illustrated above. In this figure, simultaneous writes by Task A and Task B cause the UART to transmit mixed or corrupted data, which can lead to communication failures or misinterpretation of the messages.

6.1.2 Solution to Resource Contention

To ensure safe resource sharing, synchronization mechanisms are necessary. Some of the primary mechanisms are:

- **Semaphores**: A semaphore is a signaling mechanism that controls access to shared resources. It uses a counter to allow or block tasks from accessing the critical section. If the counter is positive, tasks can access the resource; if it's zero, tasks must wait. In simple language semaphore can be thought of as a token or permit; tasks must acquire this token before accessing the resource. If the semaphore is already in use, subsequent tasks are blocked until it becomes available. This ensures orderly and mutually exclusive access.
- **Mutexes (Mutual Exclusion)**: A mutex is similar to a semaphore but is specifically designed to provide exclusive access to a resource. Only one task can lock a mutex at a time, ensuring no other task can access the shared resource simultaneously

6.1.3 Semaphore as a Solution to Resource Contention

In multitasking environments, shared resources such as memory, data structures, or I/O devices often become points of contention when multiple tasks attempt to access them concurrently. Without proper synchronization, resource contention can lead to data corruption, inconsistent system states, and unreliable application behavior. Semaphores are a robust solution to this problem, providing controlled access to shared resources and ensuring system stability.

Imagine two tasks, Task A and Task B, both attempting to write data to the same UART interface.

Task A tries to send the message 11111. Simultaneously, **Task B** tries to send 2222.

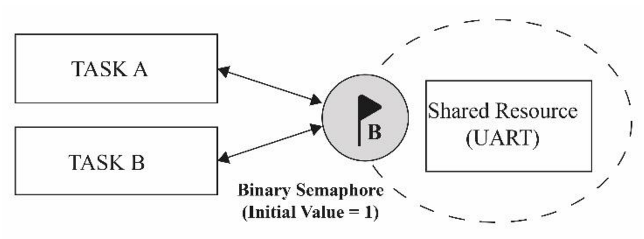
Without synchronization, both tasks may write to the UART at the same time, resulting in garbled data like 112212211.

The Problem

Data Corruption: Concurrent access leads to scrambled messages.

Unreliable Communication: Neither task's message is transmitted correctly.

By introducing a semaphore, the tasks can coordinate their access to the UART as shown below.



1. **Task A** requests and acquires the semaphore (if available).
   - It writes `11111` to the UART.
   - During this time, Task B is blocked because the semaphore is held by Task A.
2. When Task A completes, it releases the semaphore.
3. **Task B** then acquires the semaphore and writes `2222` to the UART.

The result is clear, sequential communication through the UART:

- Task A sends `11111`, followed by Task B sending `2222`.

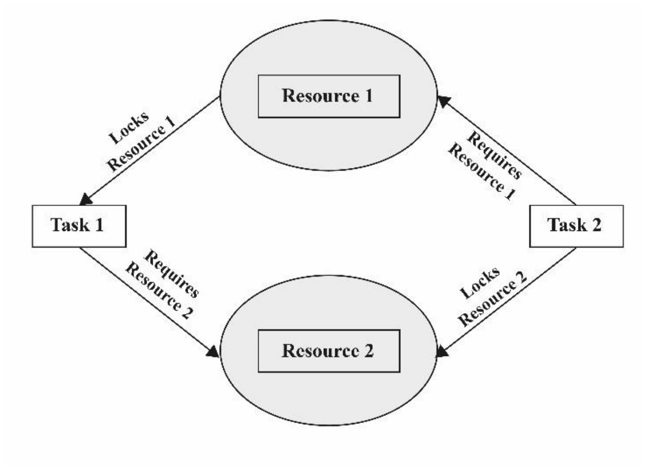
Using a **semaphore** ensures that Task A and Task B write their messages sequentially rather than simultaneously. This prevents data corruption and ensures reliable and clear communication through the UART interface.

6.2 Deadlock: A Standstill in Task Execution

Deadlock is a critical issue in multitasking environments, including real-time operating systems (RTOS), where multiple tasks or threads compete for limited resources. Deadlock occurs when two or more tasks are indefinitely blocked because each task is waiting for a resource held by another. This results in a complete standstill, where none of the tasks involved can proceed.

In an RTOS, tasks often need to access shared resources such as memory, files, or I/O devices. While synchronization mechanisms like semaphores help manage resource access, improper implementation can lead to deadlocks.

Consider two tasks, **Task A** and **Task B**, and two shared resources, **Resource 1** and **Resource 2**, in an RTOS environment. The following sequence highlights how deadlock can occur:



1. **Task Initialization**:
   - Task A starts execution and successfully locks **Resource 1**.
   - Simultaneously, Task B starts execution and locks **Resource 2**.

   At this stage:

   - **Task A holds Resource 1.**
   - **Task B holds Resource 2.**
2. **Resource Requests**:
   - Task A requires **Resource 2** to proceed, but it is currently held by Task B.
   - Task B requires **Resource 1** to proceed, but it is currently held by Task A.
3. **Circular Wait**:
   - Task A waits indefinitely for Task B to release **Resource 2**.
   - Task B waits indefinitely for Task A to release **Resource 1**.

   The system is now in a state of **deadlock**, where:

   - **Task A is waiting for Task B.**
   - **Task B is waiting for Task A.**

This situation is depicted as a circular wait, a hallmark of deadlock.

6.2.1.1 Solution to Deadlock

Deadlocks can be prevented through several strategies that ensure tasks do not get stuck in a circular waiting pattern. Here are some common methods used to avoid deadlock in real-time systems:

1. Resource Ordering

If tasks always request resources in a fixed order (for example, Resource 1 before Resource 2), the possibility of circular waiting is eliminated. This ensures that there is no scenario where Task 1 holds Resource 1 and waits for Resource 2 while Task 2 holds Resource 2 and waits for Resource 1.

Example:

- Task 1 must acquire Resource 1 first, and then Resource 2.
- Task 2 must acquire Resource 1 first, then Resource 2.
- By following this order, tasks will never end up waiting for each other.

2. **Timeouts**

   Instead of letting tasks wait indefinitely for a resource, we set a timeout. If a task cannot acquire the resource within a specific time frame, it is either retried or aborted, preventing the system from getting locked in a deadlock.

Example:

- Task 1 tries to acquire Resource 1, and then Resource 2. If it cannot acquire Resource 2 within a specified time, it releases Resource 1, retries, or aborts its operation.
- This approach ensures that no task is left waiting forever, thereby breaking potential deadlocks before they can form.

6.3 Priority Inversion: Low-Priority Tasks Blocking High-Priority Ones

Priority inversion happens when a high-priority task is waiting for a resource that is currently held by a lower-priority task. In a well-designed system, higher-priority tasks should always preempt lower-priority tasks, ensuring they execute first. However, during priority inversion, the execution order is reversed, and the high-priority task has to wait for the low-priority task to release the resource it needs.

Example of Priority Inversion:

1. **Task Setup**: We have three tasks with different priorities:
   - Task A: High priority
   - Task B: Low priority
   - Task C: Medium priority
2. **Execution Flow:**

- Task A starts executing and needs Resource 1.
- Task B, a low-priority task, holds Resource 1 and continues executing, preventing Task A from acquiring the resource.
- Task C, a medium-priority task, starts and executes. However, Task A is still waiting for Task B to release Resource 1.
- In this situation, Task A, despite being higher priority, is indirectly blocked by Task B through Task C, leading to priority inversion.

The key issue here is that Task A (high priority) is blocked by Task B (low priority) because Task B is holding a resource that Task A needs. In the meantime, Task C, which has a medium priority, executes, further delaying Task A.

6.3.1 Solution to Priority Inversion

Two main solutions are commonly used to address priority inversion:

- **Priority Inheritance Protocol**: When a low-priority task holds a resource needed by a higher-priority task, the low-priority task temporarily inherits the priority of the high-priority task. This allows the low-priority task to finish quickly and release the resource, allowing the high-priority task to proceed.
- **Priority Ceiling Protocol**: Each resource is assigned a "ceiling" priority, which is the highest priority of any task that could access it. A task must have a priority equal to or higher than the ceiling of the resource before it can access it, preventing lower-priority tasks from blocking higher-priority ones.

These protocols ensure that high-priority tasks are not unduly blocked by lower-priority tasks, maintaining the real-time guarantees of the system.

7 Multithreading and Multi-tasking in RTOS: Efficient Task Management and Scheduling

In a **Real-Time Operating System (RTOS)**, multithreading and multi-tasking are key concepts that allow the efficient execution of multiple tasks or threads within the system, especially in time-critical applications. Let’s break down these concepts:

7.1 Multithreading in RTOS

Multithreading refers to the ability of an RTOS to manage and execute multiple threads within a single process. A thread is the smallest unit of execution in a program, and an RTOS can run multiple threads concurrently, making the system more efficient in handling concurrent operations.

- **Thread**: A thread is a lightweight process with its own execution path but shares the same resources (such as memory) with other threads within the same process. Threads within an RTOS can execute independently, but they may need to synchronize or share resources.
- **Context Switching**: In a multithreaded RTOS, the operating system can switch between threads quickly to ensure that high-priority tasks are executed when needed. This process is called **context switching**, where the RTOS saves the current state of a running thread and restores the state of the next thread to run.
- **Preemptive Scheduling**: RTOS often uses preemptive scheduling for multithreading. This means the RTOS can forcibly suspend the execution of a currently running thread (even if it has not finished) in favor of a higher-priority thread.

7.2 Multi-tasking in RTOS

Multi-tasking is the ability of an RTOS to manage and execute multiple tasks concurrently, even though there may be fewer processors than tasks. The RTOS switches between tasks so quickly that they appear to run simultaneously, improving efficiency and responsiveness.

- **Task**: A task is a block of code or a program that is executed by the system. In an RTOS, tasks can have different priorities, and the RTOS scheduler decides which task to execute based on the priority and timing constraints.
- **Task Scheduling**: RTOS uses a task scheduler to decide which task should run at any given time. The task scheduler can implement various algorithms such as round-robin, priority-based scheduling, and time-slicing to ensure tasks meet their deadlines.
- **Time-slicing**: In some RTOS configurations, time-slicing is used for task scheduling. Time-slicing ensures that each task gets a fixed amount of time to execute, after which

the RTOS switches to another task. This ensures fairness and responsiveness, especially when tasks have equal priority.

7.3 Key Differences Between Multithreading and Multi-tasking:

- **Granularity**: In multithreading, multiple threads within the same process run concurrently, whereas multi-tasking typically refers to multiple independent tasks that run concurrently.
- **Resource Sharing**: Threads in a multithreading environment share the same process resources, while tasks in a multi-tasking environment may have separate resources.
- **Scheduling**: Multithreading focuses on managing the execution of threads within a process, while multi-tasking deals with the management of multiple independent tasks that may run concurrently.

Review Questions

Q.1 What are real-time systems, and how are they different from non-real-time systems?

Q.2 Explain the difference between hard real-time and soft real-time systems with examples.

Q.3 What is a Real-Time Operating System (RTOS), and how does it differ from a general-purpose operating system?

Q.4 Why is RTOS widely used in embedded systems, and what are its major characteristics?

Q.5 What is task scheduling in RTOS, and what are the common scheduling algorithms used?

Q.6 Discuss the role of task synchronization mechanisms, such as semaphores and mutexes, in RTOS

Q.7 Explain the issues that arise during resource sharing in a multitasking environment and How such problem can be addressed in RTOS.

Q.8 Explain priority inversion, Discuss the methods to address this problem.

Q.9 Define multithreading and multitasking. How do they differ in the context of RTOS?

Q.10 Define Task and Explain different Task State with the help of diagram.