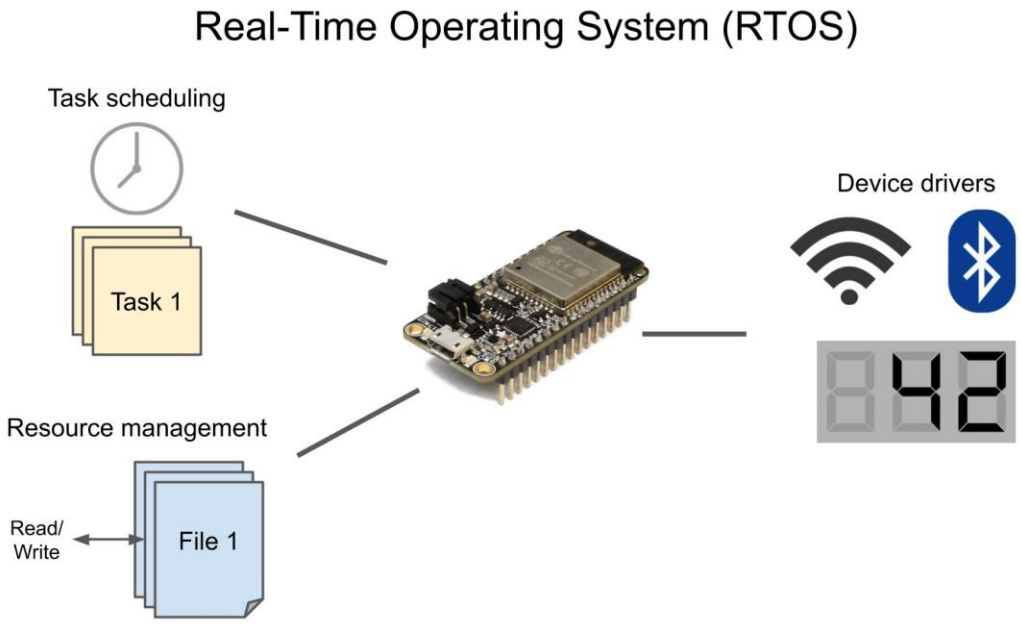
RTOS Scheduling for Embedded Systems

From Theory to Practice

CONTENTS

INTRODUCTION TO REAL-TIME OPERATING SYSTEMS AND SCHEDULING

Real-time operating systems (RTOS) are specialized software systems that provide predictable and deterministic response times for critical applications in real-time systems. These systems are designed to execute tasks with strict timing requirements and provide guaranteed execution times for mission-critical applications.



Task Scheduling

Task scheduling is a critical component of RTOS that allows the system to manage and execute multiple tasks concurrently. In RTOS, tasks are usually assigned priorities based on their importance and time sensitivity. The RTOS scheduler uses these priorities to decide which task to execute next. When a task with a higher priority becomes ready to run, it pre-empts the current task and starts executing immediately.

For example, in a flight control system, the task of monitoring and adjusting the altitude of the aircraft would have a higher priority than the task of adjusting the cabin temperature. The RTOS scheduler ensures that critical tasks have guaranteed execution times, which is important in real-time systems where timing is critical.

Interrupt Handling

RTOS also provides specialized interrupt handling mechanisms that allow the system to respond quickly to external events. Interrupts are used to signal the occurrence of an event that requires immediate attention, such as an incoming data packet or a sensor reading. The RTOS interrupt handler saves the current state of the system and executes an interrupt service routine (ISR) to manage the event. Once the ISR completes, the RTOS returns to the interrupted task and resumes execution.

For example, in an autonomous vehicle, the vehicle's sensors may send interrupts to the RTOS to alert it to changes in the environment, such as the presence of another vehicle or a pedestrian. The RTOS can respond quickly to these events and take appropriate action to ensure the safety of the vehicle and its occupants.

Real-time Communication

Real-time communication is another critical feature of RTOS that allows tasks to communicate with each other and share resources efficiently. RTOS provides several communication mechanisms, including message passing, shared memory, and semaphores, which allow tasks to synchronize and exchange data in a real-time environment.

For example, in a smart home system, the RTOS can use message passing to allow the temperature control task to communicate with the lighting control task to ensure that the temperature and lighting settings are coordinated. This type of communication ensures that the smart home system functions smoothly and efficiently.

Conclusion

In summary, RTOS is a specialized software system that provides predictable and deterministic response times for critical applications in real-time systems. Task scheduling, interrupt handling, and real-time communication are critical features of RTOS that allow the system to manage and execute multiple tasks concurrently while ensuring that the system meets its real-time requirements. These features are essential for a wide range of applications, from medical devices and flight control systems to autonomous vehicles and smart homes.

FUNDAMENTALS OF TASK SCHEDULING IN RTOS

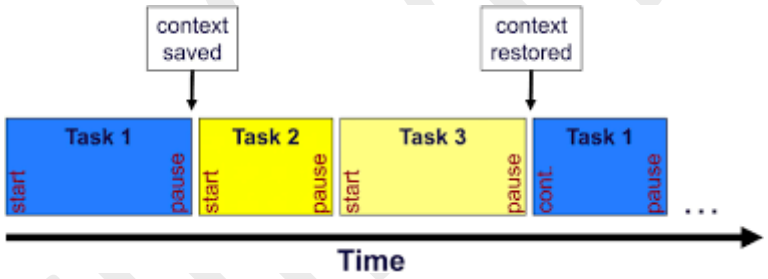
Do you ever wonder how your phone can play music, send messages, and run apps all at the same time without crashing? It is all thanks to real-time operating systems (RTOS) and task scheduling!

In the world of RTOS, tasks are like tiny workers that each have their own job to do. But how does the system decide which task to run first? That is where task scheduling comes in.

Types of Task Scheduling in RTOS

There are two main types of task scheduling in RTOS, which are pre-emptive and non-pre-emptive scheduling.

- *Pre-emptive scheduling* allows a higher priority task to interrupt a lower priority task that is currently running.
- In contrast, *non-pre-emptive scheduling* does not allow interruption and only switches tasks when the running task completes or waits for input/output (I/O) operations.



Priorities in Task Scheduling in RTOS

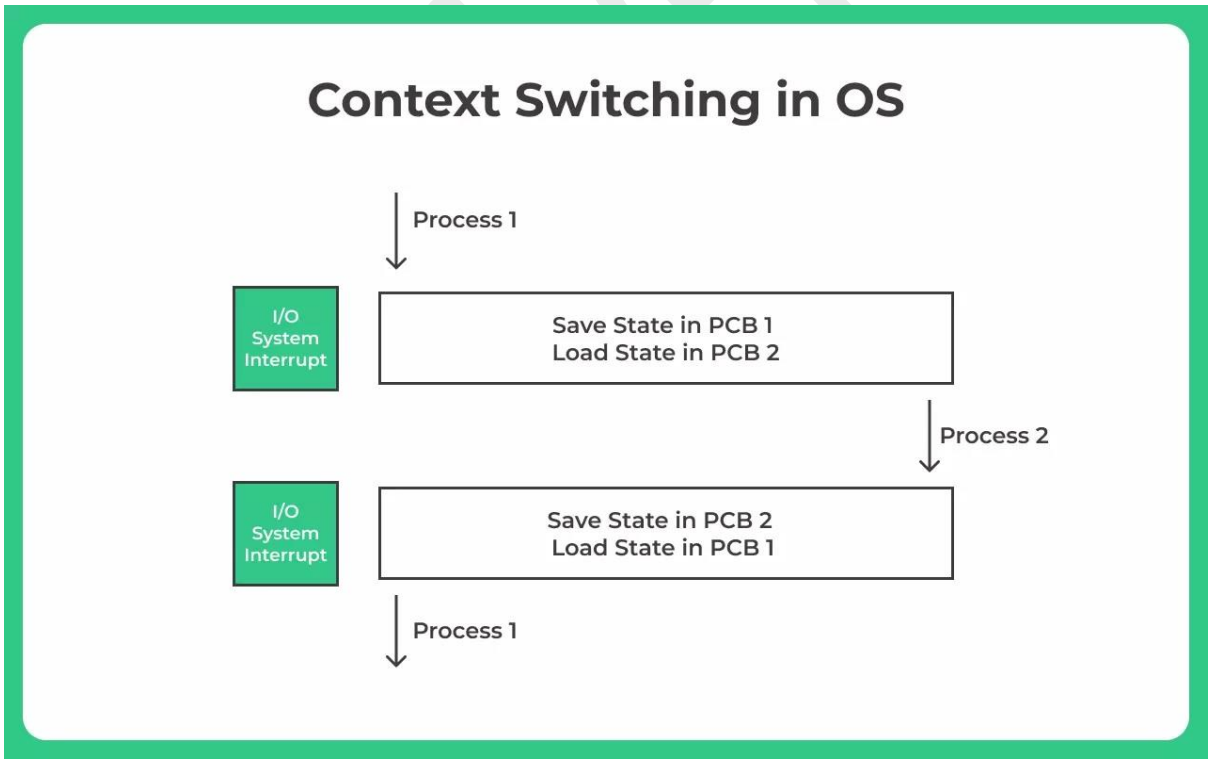
In RTOS, tasks are usually assigned priorities based on their importance and time sensitivity. The RTOS scheduler uses these priorities to decide which task to execute next. When a task with a higher priority becomes ready to run, it pre-empts the current task and starts executing immediately. The priority levels are typically defined in a range from 0 (lowest priority) to N-1 (highest priority), where N is the maximum number of priorities supported by the RTOS.

Scheduling Algorithms in Task Scheduling in RTOS

The scheduling algorithm used by the RTOS determines how the system selects which task to run next. There are several types of scheduling algorithms, including round-robin, rate-monotonic, and earliest deadline first (EDF). The round-robin technique allocates a fixed time slice to each task and switches to the next task after the time slice expires. The rate-monotonic technique assigns priorities based on task periods, with the shortest period tasks given the highest priority. The EDF technique assigns priorities based on deadlines, with tasks having the earliest deadline given the highest priority.

Context Switching in Task Scheduling in RTOS

Context switching is the process of saving the state of a currently running task and restoring the state of a new task. It occurs when the RTOS scheduler decides to switch from one task to another. The context switch time is a critical factor in real-time systems as it affects the system's responsiveness and performance. The RTOS must minimize the context switch time to ensure that the system can respond quickly to external events.



Task Scheduling Performance Metrics in RTOS

Task scheduling performance metrics are used to evaluate the efficiency and effectiveness of the RTOS scheduling algorithm. Some of the commonly used performance metrics include response time, throughput, and deadline compliance. Response time is the time taken by the system to respond to an event, while throughput is the number of tasks completed per unit time. Deadline compliance measures the percentage of tasks that are completed before their deadline.

Conclusion

In conclusion, task scheduling is a fundamental mechanism used by RTOS to manage and execute multiple tasks concurrently. The scheduling algorithm used by the RTOS determines how the system selects which task to run next. Task scheduling performance metrics are used to evaluate the efficiency and effectiveness of the RTOS scheduling algorithm. By managing the execution of multiple tasks, the RTOS can ensure that critical tasks are executed with minimal delay and that the system responds quickly to external events.

MUTEX AND SEMAPHORE

Mutex and semaphore are two commonly used synchronization mechanisms in concurrent programming and operating systems. They play a critical role in coordinating access to shared resources and ensuring the correctness and efficiency of concurrent programs.

What is a Mutex?

- A mutex, short for mutual exclusion, is a synchronization object that is used to protect shared resources, ensuring that only one thread or process can access the resource at a time.
- Mutexes are typically used in situations where a resource can only be accessed by one thread or process at a time.

For example, when multiple threads or processes need to access a shared variable or a critical section of code, a mutex can be used to prevent conflicts and ensure that only one thread or process can access the shared resource at a time.

How Mutex Works?

Mutexes are typically implemented using a lock, which is acquired when the mutex is requested and released when the mutex is released. When a thread acquires a mutex, it gains exclusive access to the shared resource, and no other thread can access the resource until the mutex is released. When the thread releases the mutex, the lock is released, and other threads can then request and acquire the mutex.

Advantages of Using Mutex:

One of the key advantages of mutexes is that they can be used to implement critical sections of code, which are sections of code that must be executed atomically to avoid race conditions. Race conditions occur when multiple threads or processes try to access shared resources simultaneously, leading to unexpected or incorrect behaviour. By using a mutex to protect a critical section of code, developers can ensure that only one thread or process can execute the critical section at a time, avoiding race conditions and ensuring correct behaviour.

What is a Semaphore?

A semaphore is another synchronization mechanism used to control access to shared resources by multiple threads or processes. Unlike a mutex, a semaphore can allow multiple threads to access a shared resource simultaneously, up to a specified limit. Semaphores are often used in situations where a limited number of threads or processes can access a shared resource at a time.

How Semaphore works?

A semaphore maintains a counter that indicates the number of threads currently accessing the shared resource. When a thread requests access to the resource, the semaphore's value is decremented, and the thread is granted access if the counter is greater than or equal to zero. When a thread releases the resource, the semaphore's value is incremented, allowing other threads to access the resource.

Semaphores can be used to implement a variety of synchronization patterns, such as producer-consumer and reader-writer. In a producer-consumer scenario, multiple threads may be producing data for a shared buffer, while other threads may be consuming data from the same buffer. A semaphore can be used to limit the number of producers and consumers accessing the buffer at a time, ensuring that the buffer is not overwhelmed with too much data. A semaphore can have a binary (0 or 1) or non-binary value, depending on the application's requirements.

Similarly, in a reader-writer scenario, multiple threads may be reading data from a shared resource, while one thread may be writing data to the resource. A semaphore can be used to ensure that only one thread is writing to the resource at a time, while multiple threads can read from the resource simultaneously.

Advantages of using Semaphore?

One of the key advantages of semaphores is that they can be used to implement priority scheduling, where higher-priority threads are given access to the resource before lower-priority threads. This can be useful in situations where certain threads or processes require more urgent access to a shared resource than others.

S.No	Mutex	Semaphore
1	A program object that allows multiple processes to take turns to share the same resource.	A variable that is used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system
2	Locking Mechanism	Signalling Mechanism
3	If the mutex is locked, the process that requests the lock waits until the system releases the lock	If the semaphore value is 0, the process perform wait operation until the semaphore becomes greater than 0
4	No categorization	Categorized as binary and counting semaphores

Conclusion:

In conclusion, mutexes and semaphores are essential synchronization mechanisms in concurrent programming and operating systems. They play a critical role in coordinating access to shared resources, ensuring correct behaviour, and improving the efficiency of concurrent programs. Developers must carefully choose between mutexes and semaphores depending on the synchronization pattern and requirements of their application.

NON-PRE-EMPTIVE SCHEDULING IN RTOS

Introduction

Non-pre-emptive scheduling is a scheduling technique used in Real-Time Operating Systems (RTOS) that allows a task to complete its execution before switching to another task. In non-pre-emptive scheduling, the scheduler does not interrupt a running task unless the task voluntarily releases the CPU.

Example 1: Infotainment System

One example of how non-pre-emptive scheduling can be used in the automotive industry is in an infotainment system. The infotainment system is responsible for managing entertainment and navigation functions, such as playing music, displaying maps, and providing voice commands.

In non-pre-emptive scheduling, the infotainment system can allocate the CPU to a task responsible for playing music until the music is complete or the user requests to change the song. The task responsible for playing music can block on a semaphore until it receives a signal to play the next song. This way, the task voluntarily releases the CPU, allowing other tasks to run until it is ready to play the next song.

Example 2: Climate Control System

Another example of non-pre-emptive scheduling in the automotive industry is in a climate control system. The climate control system is responsible for regulating the temperature and air conditioning of the vehicle's cabin.

In non-pre-emptive scheduling, the climate control system can allocate the CPU to a task responsible for monitoring the temperature and controlling the air conditioning until the desired temperature is achieved. The task responsible for monitoring the temperature can wait for a signal to change the temperature or turn on the air conditioning. This way, the task voluntarily releases the CPU, allowing other tasks to run until the temperature changes or a new signal is received.

Drawbacks of Non-Pre-emptive Scheduling

While non-pre-emptive scheduling can provide a better user experience and reduce context switching overhead, it has some drawbacks. One of the main disadvantages of non-pre-emptive scheduling is that it can lead to priority inversion.

Priority inversion occurs when a higher priority task is blocked by a lower priority task holding a shared resource, resulting in unpredictable behaviour, and potentially compromising the system's safety and reliability. In the automotive industry, priority inversion can occur when a lower priority task responsible for managing the infotainment system holds a shared resource, such as a display, while a higher priority task responsible for monitoring the engine or brakes is waiting for the resource.

Conclusion

In summary, non-pre-emptive scheduling is a scheduling technique used in RTOS that allows a task to complete its execution before switching to another task. In the automotive industry, non-pre-emptive scheduling can be used in infotainment and climate control systems to provide a better user experience. However, it is important to be aware of the drawbacks of non-pre-emptive scheduling, such as priority inversion, and choose the appropriate scheduling technique based on the system's requirements and priorities.

PRE-EMPTIVE SCHEDULING IN RTOS

What is pre-emptive scheduling and why is it important in RTOS?

Pre-emptive scheduling is a technique used in Real-Time Operating Systems (RTOS) to manage the execution of multiple tasks or processes. It ensures that critical tasks are executed in a timely manner and the system's responsiveness is maintained. It is important in RTOS because it guarantees that the system is responsive to critical events, ensuring the safety and reliability of the system.

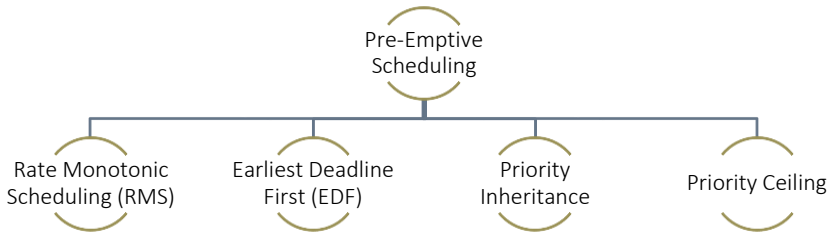
How does pre-emptive scheduling work in RTOS?

In pre-emptive scheduling, the scheduler is responsible for determining which task to run next. The scheduler uses various algorithms to prioritize the tasks based on their importance and urgency. In pre-emptive scheduling, the scheduler ensures that a higher priority task always pre-empts a lower priority task. This means that the system can respond to critical events quickly, even if a lower priority task is currently running.

Pre-emptive scheduling uses interrupts to switch between tasks. An interrupt is a signal to the processor that an event has occurred, such as a timer reaching its limit or an input/output operation completing. When an interrupt occurs, the processor suspends the current task and switches to the interrupt service routine (ISR) associated with the interrupt. The ISR performs the necessary actions, such as saving the current task's context, and then returns control to the scheduler. The scheduler then determines the next task to execute based on its priority and schedules it for execution.

Types of Pre-emptive scheduling in RTOS?

There are different types of pre-emptive scheduling, each with its advantages and disadvantages.



1. **Rate-Monotonic Scheduling (RMS):** Rate-Monotonic Scheduling (RMS) is a pre-emptive scheduling algorithm that assigns priorities to tasks based on their period or frequency and is widely used in industries such as avionics, where critical tasks require timely execution.

2. **Earliest Deadline First (EDF):** Earliest Deadline First (EDF) is another pre-emptive scheduling algorithm that assigns priorities based on the deadline of each task and is effective for systems with tasks that have varying execution times and deadlines, such as real-time video processing.

3. **Priority Inheritance:** Priority Inheritance is a pre-emptive scheduling technique used to prevent priority inversion in systems with shared resources.

4. **Priority Ceiling:** Priority Ceiling is another pre-emptive scheduling technique used to prevent priority inversion. In this technique, each shared resource is assigned a priority ceiling, which is the maximum priority of all tasks that can access the resource.

Benefits and Drawbacks of Pre-emptive Scheduling

Pre-emptive scheduling can help prevent priority inversion, a problem that can occur when a low-priority task holds a resource required by a high-priority task. In this case, the high-priority task is unable to execute, even though it has a higher priority than the low-priority task. Pre-emptive scheduling can solve this problem by allowing the high-priority task to pre-empt the low-priority task and access the resource it needs.

However, pre-emptive scheduling also has some disadvantages. It can cause context switching overhead, which is the time required to save and restore the context of a task. This overhead can reduce the system's overall performance, especially if the system has many tasks with similar priorities. Additionally, pre-emptive scheduling can lead to unpredictable behaviour if tasks have varying execution times or if there are multiple tasks with the same priority.

Example of how pre-emptive scheduling can be used in RTOS?

One such example is in the braking system of a car, where tasks responsible for reading wheel speed sensors and activating the brakes need to be executed in a specific order to ensure safe operation. If the task responsible for reading the wheel speed sensor is delayed or blocked by a lower priority task, the system may not detect a wheel locking up and fail to activate the brakes, resulting in a potential safety hazard. In this case, pre-emptive scheduling ensures that the wheel speed sensor task pre-empts the lower priority task, ensuring that the braking system responds to critical events quickly and safely.

Conclusion

In conclusion, pre-emptive scheduling is a critical technique used in real-time systems to ensure timely and predictable responses. It allows the operating system to switch between tasks based on their priorities and guarantees that critical tasks are executed first. However, pre-emptive scheduling also has some disadvantages, such as context switching overhead and potential for unpredictable behaviour, which must be considered when designing real-time systems.

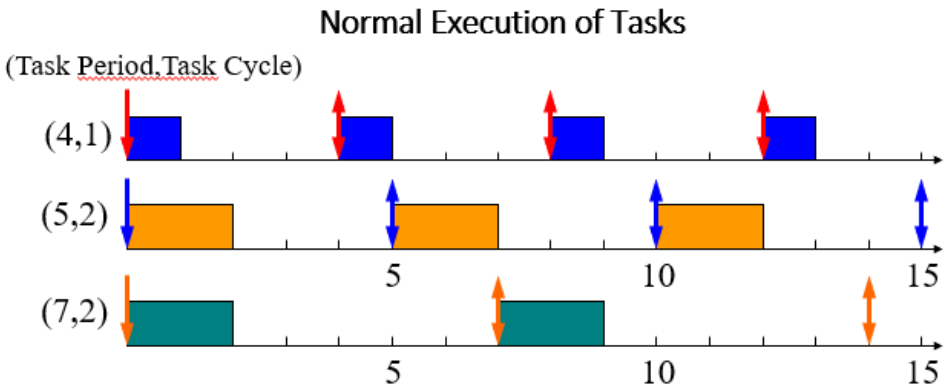
RATE-MONOTONIC SCHEDULING

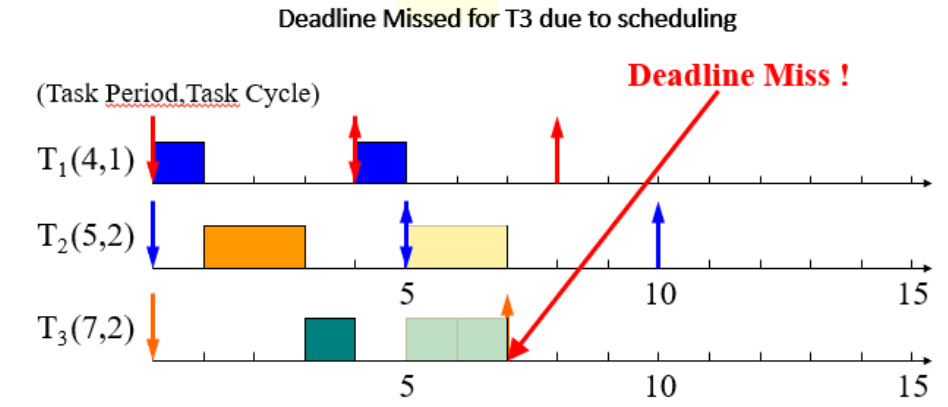
Introduction to Rate-Monotonic Scheduling (RMS)

Rate-Monotonic Scheduling (RMS) is a widely used scheduling algorithm in real-time operating systems (RTOS). It assigns task priorities based on their relative periods or execution rates. In RMS, tasks with shorter periods or higher execution rates are assigned higher priorities.

Working of RMS

The basic principle behind RMS is that tasks with shorter periods have higher temporal requirements and need to be executed more frequently to meet their deadlines. By assigning higher priority to tasks with shorter periods, the system ensures that they are executed in a timely manner, and their deadlines are not missed. In RMS, tasks are prioritized based on a mathematical formula that considers their period.





Priority Assignment in RMS

The priority levels assigned to tasks in RMS follow a strict mathematical formula based on the task's period. The task with the shortest period is assigned the highest priority, and priority levels decrease as the period increases. This ensures that the highest-priority task is always the one with the shortest period or highest execution rate.

Assumptions of RMS

RMS assumes that tasks are periodic and have fixed execution times. This makes it easy to calculate the task priorities and simplifies the scheduling process. However, this assumption may not always hold true in practice and may lead to inaccuracies in the scheduling of non-periodic or variable execution time tasks.

Benefits of RMS

Despite its limitations, RMS is a popular scheduling algorithm used in a wide range of embedded applications such as aerospace, automotive, and medical devices. Its simplicity and ease of implementation make it an attractive option for real-time systems with strict timing requirements. One of the key benefits of RMS is that it provides a simple and efficient way to ensure that real-time tasks are executed on time, and their deadlines are not missed. By assigning task priorities based on their periods or execution rates, the system can ensure that high-priority tasks are executed first, and that lower-priority tasks are executed when system resources are available.

Conclusion

RMS is a popular scheduling algorithm used in RTOS that assigns task priorities based on their relative periods or execution rates. Tasks with shorter periods are assigned higher priorities to ensure that they are executed in a timely manner, and their deadlines are not missed. Although RMS makes certain assumptions about task execution times, it is still widely used in real-time systems with strict timing requirements. By understanding the basics of RMS, engineers can design and develop reliable real-time systems for a wide range of embedded applications.

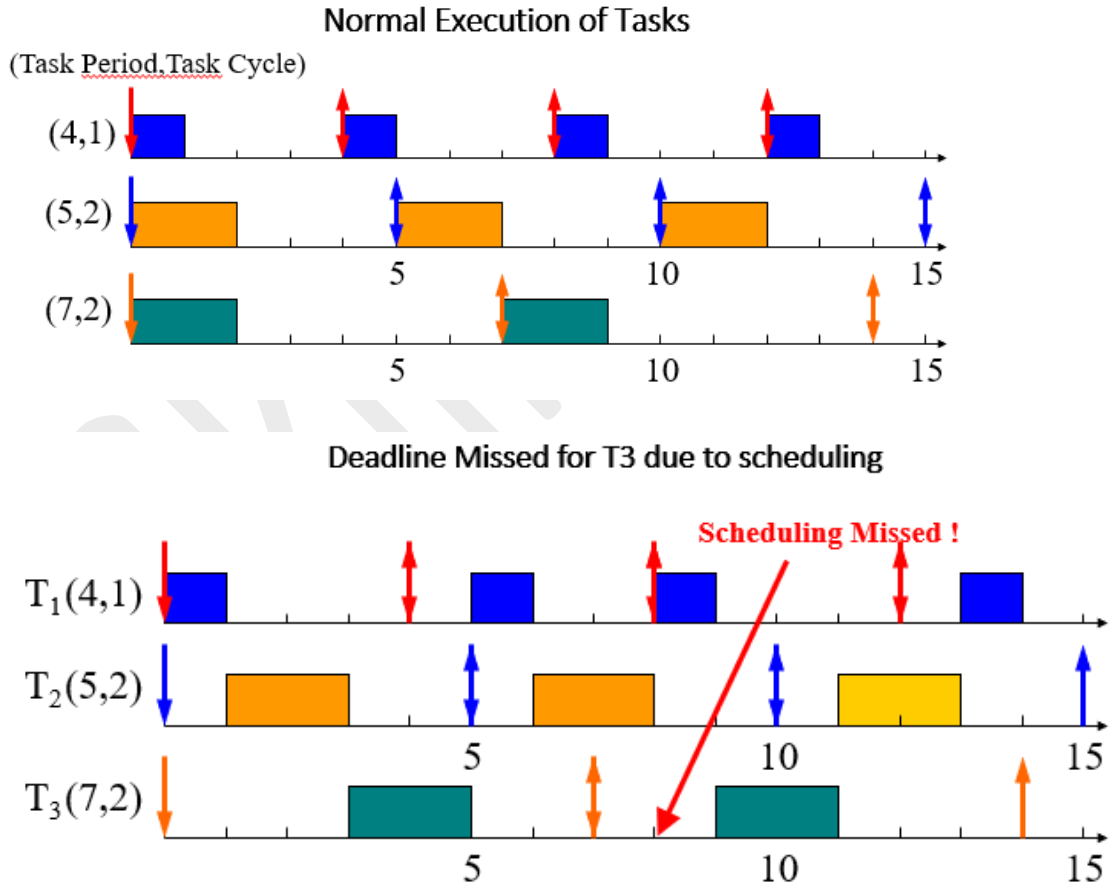
EARLIEST DEADLINE FIRST (EDF) SCHEDULING IN RTOS

Introduction to EDF Scheduling in RTOS

Earliest Deadline First (EDF) is a scheduling algorithm used in real-time operating systems that assigns priorities to tasks based on their relative deadlines. In this algorithm, tasks with earlier deadlines are given higher priorities so that they can be executed first and meet their deadlines.

How EDF works in RTOS

EDF works by maintaining a dynamic priority queue of tasks, where the task with the earliest deadline is always at the top of the queue. Whenever a task completes its execution or a new task is added, the priority queue is updated to ensure that the task with the earliest deadline is always at the top of the queue. This way, tasks are executed in the order of their deadlines.



Comparison with To-Do List

Think of it like a to-do list, where tasks that have earlier deadlines are given priority to ensure they are completed on time. Similarly, EDF assigns higher priorities to tasks with earlier deadlines so that they are executed first and meet their deadlines.

Benefits of EDF

One of the key benefits of EDF is its ability to handle tasks with varying execution times and deadlines, unlike the RMS algorithm. This makes it suitable for real-time systems where tasks may have unpredictable execution times or deadlines. EDF is a powerful scheduling algorithm for real-time systems with unpredictable workloads.

Limitations of EDF

EDF has some limitations, including the high computational overhead required to maintain the priority queue, which makes it less efficient than simpler algorithms like RMS. Also, it can be challenging to analyse the worst-case response times of tasks, making it difficult to guarantee that all tasks will meet their deadlines.

EDF in Automotive Industry

In the automotive industry, EDF can be used to schedule tasks in a vehicle's embedded system, such as the engine control unit. For example, tasks such as monitoring sensors and controlling actuators can be assigned priorities based on their deadlines to ensure that the engine operates smoothly and safely.

Conclusion

In summary, EDF is a scheduling algorithm used in RTOS that assigns priorities to tasks based on their relative deadlines. Tasks with earlier deadlines are given higher priorities to ensure that they are executed first and meet their deadlines. Despite its limitations, EDF is widely used in real-time systems where tasks have varying execution times and deadlines. Its ability to manage non-periodic and sporadic tasks makes it a powerful scheduling algorithm for real-time systems with unpredictable workloads.

PRIORITY INHERITANCE

Priority inheritance is a technique used in real-time operating systems (RTOS) to address the problem of priority inversion, where a lower-priority task holds a resource that a higher-priority task needs, resulting in the higher-priority task being blocked and unable to execute. In priority inheritance, the priority of a task is temporarily raised to prevent priority inversion and ensure that critical tasks are executed in a timely manner.

How Priority Inheritance Works?

When a higher-priority task requires a resource that is currently held by a lower-priority task, priority inheritance temporarily raises the priority of the lower-priority task to that of the higher-priority task. This allows the lower-priority task to complete its work quickly and release the resource, allowing the higher-priority task to proceed without delay.

Examples of Priority Inheritance

For example, suppose that a high-priority task needs to access a shared resource that is currently being used by a low-priority task. Without priority inheritance, the low-priority task would continue to use the resource, blocking the high-priority task from executing. However, with priority inheritance, the priority of the low-priority task is temporarily raised to the level of the high-priority task, allowing it to complete its work quickly and release the resource. Once the resource is released, the priority of the low-priority task returns to its original level.

In the context of automotive embedded systems, priority inheritance can play a critical role in ensuring the safe and reliable operation of a vehicle. For example, the anti-lock braking system (ABS) is a safety-critical system that helps prevent the wheels of a car from locking up during braking, which can lead to loss of control and accidents.

To ensure that the ABS operates reliably and efficiently, an RTOS with priority inheritance can be used. The ABS controller task may require access to a resource that is being held by a lower-priority task. Without priority inheritance, the ABS controller task could be blocked, potentially leading to a safety-critical situation. However, with priority inheritance, the RTOS will temporarily raise the priority of the lower-priority task to that of the ABS controller task, allowing the ABS controller task to execute and ensure the safe operation of the vehicle.

Benefits of Priority Inheritance

Priority inheritance is a useful technique in real-time systems because it helps to prevent priority inversion and ensure that critical tasks are executed in a timely manner. By temporarily raising the priority of a lower-priority task, priority inheritance can prevent a higher-priority task from being blocked and improve the overall responsiveness of the system.

In addition to preventing priority inversion, priority inheritance can also improve the predictability and reliability of the system. By ensuring that critical tasks are executed on time, priority inheritance can help to prevent system failures and reduce the risk of catastrophic events in safety-critical systems.

Potential Issues with Priority Inheritance

While priority inheritance can be an effective technique for preventing priority inversion, it can also introduce new problems if not implemented correctly. One potential issue is the possibility of priority inversion cycles, where multiple tasks hold resources that are required by other tasks, leading to a chain of priority inheritance that can result in deadlock or other system failures.

Another issue is the potential for priority inversion to occur even with priority inheritance in place if the system is not carefully designed. For example, if the priority levels of tasks are not properly assigned or if there are too many high-priority tasks, priority inversion can still occur despite the use of priority inheritance.

Conclusion

Priority inheritance is a useful technique in real-time systems for preventing priority inversion and ensuring the timely execution of critical tasks. By temporarily raising the priority of a lower-priority task, priority inheritance can help to prevent a higher-priority task from being blocked and improve the overall responsiveness of the system. However, it is important to carefully design the system to avoid potential issues such as priority inversion cycles and to ensure that priority levels are properly assigned to prevent priority inversion from occurring in the first place.

PRIORITY CEILING

Priority Ceiling is a synchronization mechanism used in Real-Time Operating Systems (RTOS) to prevent priority inversion problems. In this mechanism, each resource in the system is assigned a priority ceiling, which is the maximum priority of any task that can access the resource.

Definition and Explanation:

Priority ceiling is a protocol used to prevent priority inversion in Real-time operating systems (RTOS). When multiple tasks share a resource such as a memory location or an input/output device, priority inversion can occur, and a higher-priority task may be blocked by a lower-priority task that holds the resource.

Priority Ceiling Protocol

The priority ceiling protocol is a synchronization mechanism that prevents priority inversion by temporarily raising the priority of the task that holds a shared resource to the highest priority of any task that can access the resource. When a task requests access to a shared resource, it checks the resource's priority ceiling. If the task's priority is higher than the resource's ceiling, the task is blocked until the resource is released. If the task's priority is lower than or equal to the resource's ceiling, the task temporarily inherits the ceiling priority, allowing it to execute while the resource is held.

Example of Priority Ceiling Protocol

Consider an example where a high-priority task requires access to a shared resource held by a low-priority task. Without priority ceiling protocol, the low-priority task could hold the resource for an extended period, blocking the high-priority task and leading to a priority inversion problem.

With the priority ceiling protocol, when the high-priority task requests the shared resource, the RTOS will temporarily raise the priority of the low-priority task to the ceiling priority of the resource. This prevents the low-priority task from blocking the high-priority task, ensuring that the high-priority task can execute without delay and prevent any critical issues.

Benefits and Drawbacks of Priority Ceiling Protocol

Priority ceiling protocol offers several benefits, including preventing priority inversion problems and ensuring timely execution of high-priority tasks. However, it also has some drawbacks, including increased overhead due to the need to check the priority ceiling of each

shared resource and potential deadlocks if two tasks hold resources that have the same ceiling priority.

Applications of Priority Ceiling Protocol

Priority ceiling protocol is widely used in safety-critical systems such as avionics, medical equipment, and industrial control systems. It is used to ensure the safe and reliable operation of these systems by preventing priority inversion problems and ensuring that critical tasks are executed in a timely and efficient manner.

Conclusion

Priority ceiling is an essential mechanism used in Real-Time Operating Systems (RTOS) to prevent priority inversion problems. It ensures the timely execution of high-priority tasks and prevents lower-priority tasks from blocking them by temporarily raising their priority to the ceiling priority of the shared resource. This mechanism is widely used in safety-critical systems, including aviation, medical equipment, and industrial control systems, to ensure the safe and reliable operation of these systems.

MULTICORE SCHEDULING STRATEGIES IN RTOS

Multicore processors are a powerful tool for real-time systems, but to utilize their resources effectively, it is critical to choose the right scheduling strategy. Let us dive into some of the commonly used scheduling strategies in real-time operating systems (RTOS) for multicore processors.

Static Partitioning:

In this strategy, each task is assigned to a specific core during system start-up, and the core remains dedicated to executing that task throughout the system's operation. This approach is simple to implement, but it may lead to uneven resource utilization. For example, if one task has a much lower workload than another, the core dedicated to the former may remain idle while the latter overloads its core.

Dynamic Partitioning:

This strategy involves assigning tasks to cores based on their current workload. The RTOS periodically checks the workloads of each task and reassigns them to cores with lower utilization. This approach can maximize resource utilization and minimize the risk of uneven resource allocation, but it can be challenging to implement and requires significant computational overhead to reassign tasks between cores. Dynamic partitioning would be the best choice in scenarios where task workloads vary significantly.

Global Scheduling:

In this strategy, all tasks are scheduled across all available cores, with each core assigned a priority level. The RTOS dynamically schedules tasks across all cores based on their priority level. This approach can maximize resource utilization and improve system performance, but it can also be complex and challenging to implement. A potential issue with global scheduling is that if a single task requires a lot of resources, it may monopolize a core, leading to inferior performance for other tasks.

Gang Scheduling:

This global scheduling approach schedules a group of related tasks together across all available cores. It is useful for tasks that require synchronized execution or intertrack communication. For example, in a video rendering application, one core may manage the rendering of frames, while another core handles encoding and compression.

Load Balancing:

This strategy involves periodically checking the workload of each core and redistributing tasks to balance the workload across all cores. While this approach can improve resource utilization and reduce the risk of overloading a single core, it can be challenging to implement and may lead to increased system overhead.

Hybrid Scheduling:

This strategy combines elements of static and dynamic partitioning and global scheduling to balance resource utilization and system complexity. It can be beneficial in scenarios where task workloads vary significantly, and some tasks require synchronized execution.

Conclusion:

In conclusion, the efficient utilization of multicore processors in real-time systems is a critical factor in ensuring high performance and timely execution of tasks. The choice of scheduling strategy plays a crucial role in achieving this goal, and each strategy has its strengths and weaknesses, and the choice of the appropriate scheduling strategy depends on the specific requirements of the system, including task workloads, intertrack communication requirements, and available computational resources. By carefully selecting and implementing the right scheduling strategy, real-time systems can maximize their efficiency, performance, and responsiveness while ensuring timely execution of critical tasks

RTOS SCHEDULING FOR EMBEDDED SYSTEMS: CASE STUDIES AND BEST PRACTICES

RTOS scheduling plays a critical role in ensuring the timely execution of critical tasks in embedded systems. Here, we will explore some case studies and best practices in RTOS scheduling for embedded systems.

Aviation Industry

One of the most popular RTOS scheduling strategies is Rate-Monotonic Scheduling (RMS), which assigns priorities based on the inverse of the task period. The avionics industry is an example of an embedded system that utilizes RMS, where critical flight control tasks require timely execution to ensure passenger safety. The use of RMS in avionics systems has been shown to provide prominent levels of reliability and predictability, ensuring the timely execution of critical tasks.

Automotive Industry

Many modern vehicles utilize dozens of electronic control units (ECUs) to control various systems such as engine control, transmission control, and infotainment. In this context, scheduling becomes particularly important as the ECUs must cooperate and communicate with each other to ensure smooth and safe operation of the vehicle. One best practice in this scenario is the use of global scheduling to balance the workload across all ECUs and ensure timely execution of critical tasks such as engine control. For example, if the engine control ECU is not given priority, it could lead to a malfunction in the engine, putting the passengers at risk.

Real-time Video Processing

Real-time video processing is another area where RTOS scheduling plays a critical role. For example, video processing in surveillance systems requires timely execution of tasks such as object detection, tracking, and recognition. In this context, Earliest Deadline First (EDF) scheduling has been shown to be an effective approach for ensuring that critical tasks are executed on time while maximizing resource utilization.

Medical Industry

In the medical industry, RTOS scheduling is used in various medical devices such as implantable devices, monitoring devices, and drug delivery systems. For example, in implantable devices such as pacemakers, the use of dynamic partitioning can ensure that tasks are assigned to the appropriate core based on their workload, maximizing battery life while ensuring the timely execution of critical tasks.

Best Practices in RTOS Scheduling for Embedded Systems

Ensure optimal RTOS scheduling for embedded systems, some best practices should be followed:

1. Choose the appropriate scheduling strategy based on the specific requirements of the system.
2. Prioritize critical tasks to ensure their timely execution.
3. Use global scheduling to balance workload and improve resource utilization.
4. Implement load balancing techniques to ensure efficient use of all available computational resources.
5. Use tools such as profiling and monitoring to detect and resolve scheduling issues.
6. Ensure that the RTOS is correctly configured and optimized for the specific hardware platform.

In summary, RTOS scheduling is critical for ensuring the timely and reliable execution of critical tasks in embedded systems. By choosing the appropriate scheduling strategy and implementing best practices, system designers can maximize performance and efficiency while ensuring that critical tasks are executed on time.