Module 5 : Deadlock & Multicore Systems

Module 5 : Deadlock & Multicore Systems

Deadlock: Understanding and Prevention

Deadlock is a situation where two or more processes or tasks are indefinitely blocked, waiting for each other to release some resources needed to proceed. Deadlock can occur in any system involving concurrency and resource sharing, such as real-time systems. In FreeRTOS, deadlock can happen when two or more tasks attempt to access resources protected by mutual exclusion mechanisms like mutexes or semaphores.

Example Scenario

Consider two tasks on an ESP32: Task A and Task B. Both tasks need to access two resources: Resource 1 and Resource 2. These resources are protected by mutexes: Mutex 1 and Mutex 2. The following sequence of events can lead to deadlock:

1. Task A acquires Mutex 1 and locks Resource 1.
2. Task B acquires Mutex 2 and locks Resource 2.
3. Task A tries to acquire Mutex 2 to access Resource 2 but is blocked because Mutex 2 is held by Task B.
4. Task B tries to acquire Mutex 1 to access Resource 1 but is blocked because Mutex 1 is held by Task A.

Both tasks are now deadlocked as each is waiting for the other to release a mutex.

Detecting and Avoiding Deadlock

There are two main approaches to handle deadlock: **prevention** and **detection/recovery**.

Prevention

To prevent deadlock, we need to ensure that at least one of the four conditions necessary for deadlock is not met. These conditions are:

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode by a task.
2. **Hold and Wait:** A task must hold at least one resource and wait for additional resources currently held by other tasks.
3. **No Preemption:** Resources cannot be forcibly taken from tasks holding them; they must be released voluntarily.
4. **Circular Wait:** A set of tasks must exist such that each task is waiting for a resource held by the next task in the set, forming a circular chain.

Strategies to Avoid Deadlock:

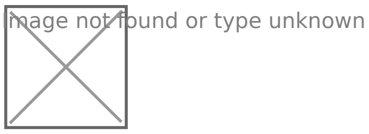
- **Avoid Mutual Exclusion:** Make resources shareable or use virtualization techniques to create multiple copies of the same resource. For example, use read/write locks instead of mutexes to allow multiple tasks to access the resource in read-only mode, or use memory mapping to create virtual copies of physical memory.
- **Avoid Hold and Wait:** Ensure tasks request all the resources they need at once or release resources they hold before requesting new ones. For example, use a nested monitor pattern to acquire all required mutexes in a tiered manner or check for mutex availability before acquisition.
- **Avoid No Preemption:** Allow tasks to release resources they hold when they are blocked or preempted by higher-priority tasks. For example, use priority inheritance protocols to ensure that tasks holding mutexes inherit the priority of the highest-priority task blocked on the mutex, or use priority ceiling protocols to ensure that tasks acquiring mutexes have the highest priority among all tasks that can access the mutex.
- **Avoid Circular Wait:** Enforce an order on resources and ensure tasks request resources in increasing order of their identifiers. For example, assign unique numbers to each resource and make tasks acquire resources in ascending numerical order.

Detection and Recovery

To detect and recover from deadlock, we need mechanisms to monitor the system’s status and identify the tasks involved in deadlock. Then, we need strategies to handle deadlock by releasing some resources or terminating some tasks.

Detection Method:

- **Wait-for Graph:** A wait-for graph is a directed graph representing dependencies between tasks and resources. Each node in the graph represents a task or resource. An edge from a task node to a resource node indicates that the task holds the resource. An edge from a resource node to a task node indicates that the task is waiting for the resource. Deadlock exists if and only if there is a cycle in the graph.



Recovering from Deadlock

There are several methods for recovering from deadlock:

1. **Preemption:** This involves forcibly taking resources from some tasks and reallocating them to other tasks. Preemption can be used to break the circular wait condition by reallocating resources from tasks involved in deadlock to those that can proceed.
2. **Rollback:** This method involves rolling back some tasks to a previous state and retrying their operations. Rollback is used to return tasks to a state before they entered the deadlock, allowing them to attempt their operations again without causing deadlock.
3. **Termination:** This approach involves terminating some tasks and releasing their resources. Termination can break the deadlock by stopping certain tasks, thus freeing up resources that can be reallocated to other tasks.

Each of these methods has its own trade-offs and suitability depending on the specific system requirements and constraints.

Starvation

Starvation is a situation where one or more tasks are unable to make progress because they are continuously denied access to some resources needed to complete their work. Starvation can occur in any system involving concurrency and resource allocation, such as real-time systems. In FreeRTOS, starvation can happen when one or more tasks have lower priority compared to other tasks and are continuously preempted by them.

Strategies to Address Starvation

Several strategies can be used to address starvation in real-time systems. Some of them include:

1. **Priority Inheritance:** This technique allows a lower-priority task to inherit the priority of a higher-priority task that is blocked on a resource held by the lower-priority task. This way, the lower-priority task can complete its work more quickly and release the resource for the higher-priority task. For example, if Task E is blocked on a mutex held by Task G, Task G inherits the priority of Task E until it releases the mutex.
2. **Priority Ceiling:** This technique assigns a priority ceiling to each resource, which is equal to the highest priority of all tasks that can access that resource. Each task acquiring the

resource must raise its priority to the priority ceiling of the resource until it releases it. This way, no other task can preempt the current task while it holds the resource. For example, if Task E and Task F can access Resource 1, Resource 1 has a priority ceiling of 3. If Task F acquires Resource 1, it raises its priority to 3 until it releases Resource 1.

3. **Aging:** This technique increases the priority of waiting tasks over time until it reaches a maximum value. This way, tasks that have been waiting can eventually gain access to resources or the processor. For example, if Task G is waiting for an available core, its priority will increase over time until it reaches 3 and gets an available core.
4. **Fair Scheduling:** This technique allows tasks to dynamically adjust their priorities based on system conditions or workload. This way, important or urgent tasks can be given higher priority when needed. For example, if Task G is an urgent task, it can dynamically increase its priority to get an available core more quickly.

Priority Inversion

Priority inversion is a phenomenon that occurs in multitasking systems when a higher-priority task is blocked by a lower-priority task, causing priority inversion. This can result in unwanted effects such as reduced performance, missed deadlines, or system failures.

Priority inversion can occur when tasks share resources protected by mutual exclusion mechanisms such as mutexes or semaphores. Mutexes or semaphores are locks that ensure only one task can access the shared resource at a time. When a task takes a lock, it enters a critical section where it performs operations on the shared resource. When finished, it releases the lock and exits the critical section.

However, if a higher-priority task tries to acquire the same lock while it is held by a lower-priority task, the higher-priority task will be blocked until the lower-priority task releases the lock. This means the lower-priority task is running when the higher-priority task should be running, which is the essence of priority inversion.

There are two types of priority inversion: bounded and unbounded. Bounded priority inversion occurs when the blocking time of the higher-priority task is limited by the length of the critical section of the lower-priority task. Unbounded priority inversion occurs when the blocking time of the higher-priority task is not limited and can potentially be unbounded. This can happen when a medium-priority task preempts the lower-priority task while it holds the lock, preventing it from releasing the lock and unblocking the higher-priority task.

Priority inversion can cause serious problems in real-time systems, where tasks have strict constraints and deadlines. For example, in 1997, NASA's Mars Pathfinder mission experienced several system resets due to unbounded priority inversion caused by a priority inversion bug that triggered a watchdog timer activated by a low-priority meteorological data collection task, a high-priority bus management task, and a medium-priority communication task. The low-priority and high-priority tasks shared a mutex to access the information bus, while the medium-priority task did not use the shared resource. Occasionally, the medium-priority task would run while the low-priority task held the mutex, blocking both the low and high-priority tasks for several seconds. The watchdog timer was set to reset the system if the high-priority task did not run for a certain period, assuming something was wrong. This resulted in losing one day of operation every few days until NASA engineers identified and fixed the bug using the priority inheritance protocol.

Priority Inheritance Protocol

One potential solution to prevent unbounded priority inversion is to use the priority inheritance protocol. The priority inheritance protocol is a technique that allows a lower-priority task to temporarily inherit the priority of a higher-priority task that is blocked by it. This way, the lower-priority task can complete its critical section more quickly and release the lock sooner, reducing the blocking time of the higher-priority task.

The priority inheritance protocol works as follows:

1. When a lower-priority task acquires a lock, it retains its original priority.
2. When a higher-priority task attempts to acquire the same lock, it is blocked, and the lower-priority task inherits the higher-priority task's priority.
3. When the lower-priority task releases the lock, it returns to its original priority, and the higher-priority task is no longer blocked and continues execution.
4. If multiple higher-priority tasks are blocked by the same lower-priority task, the lower-priority task inherits the highest priority among them.
5. If the lower-priority task is preempted by another task while holding the lock, it retains the inherited priority until it releases the lock.

Priority Ceiling Protocol

Another possible solution to prevent unbounded priority inversion is to use the priority ceiling protocol. The priority ceiling protocol is a technique that assigns a priority ceiling to each lock, which is the highest priority of all tasks that can acquire that lock. When a task acquires a lock, it raises its priority to the lock's priority ceiling, preventing other tasks from preempting while the task holds the lock. This way, the task can complete its critical section more quickly and release the lock sooner, reducing the blocking time of other tasks.

The priority ceiling protocol works as follows:

1. When a task acquires a lock, it raises its priority to the lock's priority ceiling.
2. When a task releases the lock, it returns to its original priority.
3. A task can only acquire a lock if its current priority is higher than the priority ceiling of any other lock held by other tasks.
4. If multiple tasks try to acquire the same lock simultaneously, the one with the highest original priority will get it first.

Multicore Systems

A multicore system is a hardware platform that has two or more processing units (cores) that can execute instructions simultaneously. Multicore systems can offer higher performance, lower power consumption, and better scalability compared to single-core systems. Multicore systems are increasingly used in embedded systems, especially for applications involving complex computations such as machine learning, image processing, or wireless communications.

Using multicore systems with an RTOS can bring many benefits, such as:

- **Increased Responsiveness**: Tasks can be distributed among cores to reduce response time and increase system throughput.
- **Increased Reliability**: Tasks can be isolated on different cores to prevent interference and enhance fault tolerance.
- **Increased Flexibility**: Tasks can be dynamically allocated to different cores based on their needs and availability.

However, using multicore systems with an RTOS also presents some challenges, such as:

- **Increased Complexity**: System design and implementation become more complex due to the need for inter-core communication, synchronization, and load balancing.
- **Reduced Predictability**: System behavior becomes less deterministic due to potential contention and interference between cores.
- **Limited Compatibility**: RTOS may not natively support multicore execution or may have different features and limitations for different architectures.

AMP vs SMP

Asymmetric Multi-Processing (AMP) and Symmetric Multi-Processing (SMP) are two common approaches to utilizing multicore systems.

AMP is an older and simpler approach involving two or more cores or processors that can communicate with each other. In AMP, one core is responsible for running the operating system and dispatching tasks or jobs to other cores. Secondary cores may have their own operating systems or minimal firmware to receive and execute tasks. AMP allows for different architectures or even different devices for the cores. For example, a microcontroller might communicate with a separate microprocessor or computer through serial interfaces or networks.

SMP is a more modern and sophisticated approach where every core or processor is treated equally and runs the same operating system across all cores. In SMP, a shared task list can be accessed by all cores, and each core takes tasks from this list to decide what work to perform. SMP requires the same architecture for each core since they need to be closely related to share resources. Cores or processors have shared memory and input/output buses where they can access shared memory and hardware.

Key Advantages of AMP:

- **Simplicity**: System design and implementation are relatively simple, as each core has a clear role and responsibility.
- **Efficiency**: The system can utilize the strengths of each core or device for different tasks, such as using low-power cores for simple tasks and high-performance cores for complex tasks.
- **Scalability**: The system can easily add more cores or devices as needed without affecting existing ones.

Key Disadvantages of AMP:

- **Overhead**: The system needs to manage communication and coordination between cores or devices, which can introduce latency and complexity.
- **Imbalance**: The system may experience load imbalance if some cores or devices are overloaded while others are idle.
- **Compatibility**: The system may face compatibility issues if different cores or devices use different protocols or standards.

Key Advantages of SMP:

- **Uniformity**: System design and implementation are consistent and cohesive since every core has the same capabilities and features.
- **Responsiveness**: The system can achieve higher performance and lower latency by distributing the workload among cores.
- **Flexibility**: The system can dynamically adjust task assignments based on task requirements and core availability.

Key Disadvantages of SMP:

- **Complexity**: System design and implementation are more complex due to the need for inter-core synchronization, contention management, and cache coherence.
- **Interference**: System behavior may become less predictable due to potential interference between cores, such as cache misses, bus contention, or interrupt conflicts.
- **Constraints**: System performance can be limited by shared resources or bottlenecks in interconnections.

ESP32 Multicore Architecture

ESP-IDF is a software development framework for ESP32 that includes a port of FreeRTOS, a popular RTOS for embedded systems. ESP-IDF provides several options for creating and scheduling tasks for multicore execution.

The first option is to let the scheduler on each core choose the highest-priority task from a shared ready list. This option is enabled by passing `tskNO_AFFINITY` as the last argument to the `xTaskCreate` or `xTaskCreatePinnedToCore` function. This option offers the highest flexibility and responsiveness, as each core can run any available and suitable task. However, it also introduces the highest complexity and non-determinism, as it is difficult to predict which core will run which task and when.

The second option is to pin tasks to specific cores at creation. This option is enabled by passing 0 or 1 as the last argument to the `xTaskCreate` or `xTaskCreatePinnedToCore` function. This option offers the highest simplicity and predictability, as each task will always run on the same core. However, it also introduces the highest overhead and imbalance, as it can lead to some cores being overloaded while others are idle.

The third option is to use inter-task communication mechanisms, such as queues, semaphores, or events, to coordinate and synchronize tasks across different cores. This option can be combined with either of the previous options to achieve smoother multicore execution control and optimization. However, it also requires careful design and implementation to avoid deadlock, starvation, or race conditions.

Challenges & Trade-offs in Multicore Systems

Using multicore systems with an RTOS involves several challenges and trade-offs that need to be considered and addressed to achieve optimal performance and reliability.

One challenge is how to balance the workload among cores. Ideally, each core should have a similar amount of work, so no core is idle or overloaded. However, this may not be easy to achieve in practice, as different tasks may have different requirements, priorities, deadlines, or dependencies. Additionally, some tasks may be better suited to certain cores than others, due to their affinity with specific hardware or protocols.

One potential solution is to use dynamic load-balancing algorithms that can monitor and adjust task assignments based on the current workload and core availability. However, this may introduce additional overhead and complexity in system design and implementation.

Another challenge is how to ensure data correctness and consistency among different cores. Since each core has its own memory cache and interrupts, certain data may become stale or inconsistent if modified by one core but not updated or removed in other cores. Additionally, some data may be shared or accessed by multiple tasks on different cores simultaneously, which can lead to data corruption or inconsistency if proper synchronization mechanisms are not used.

One potential solution is to use cache coherence protocols that can automatically update or invalidate cache entries when modified by other cores. However, this may add latency and additional contention in system performance.

Another potential solution is to use mutual exclusion mechanisms that can prevent multiple tasks from accessing or modifying data simultaneously. However, this may introduce additional overhead and complexity in system design and implementation.

Another challenge is how to minimize interference and contention between different cores. Since each core shares the same bus and memory with other cores, some cores may experience delays or conflicts when trying to access the same resources. For example, if two cores try to read or write to the same memory location simultaneously, one core may have to wait until the other core completes its operation. Additionally, some interrupts may have higher priority than others, which can prioritize the execution of certain tasks over others.

One potential solution is to use priority-based arbitration protocols that can provide different priorities for different cores or tasks when accessing shared resources. However, this may introduce additional complexity and overhead in system design and implementation.

Another potential solution is to use partitioning or isolation techniques that can allocate different resources for different cores or tasks exclusively. However, this may introduce additional waste and inefficiency in system performance.

As we can see, using multicore systems with an RTOS involves various challenges and trade-offs that need to be carefully considered and addressed to achieve optimal performance and reliability. There is no one-size-fits-all solution, as different applications may have different requirements and constraints. Therefore, it is important to understand the characteristics and limitations of multicore systems and RTOS, as well as the design goals and trade-offs of the application.

External Reference

Check out the external reference by digikey: Deadlock & Multicore