CODE TIME TECHNOLOGIES

Priority Inversion and Deadlock

Whitepaper

Introduction

In many systems, there are peripherals or data elements that must only be accessed by a single task at a time. The obvious way to enforce exclusive access to such resources is through a mutual exclusion semaphore ("mutex"). However, this may lead to a scenario termed *priority inversion*, whereby a higher priority task is indirectly blocked by a lower priority one.

This paper examines the problem in more depth and details the available remedies, including those unique to the Abassi real-time kernel.

Mutual Exclusion

Typically a resource that must be exclusively utilized for a period of time and not interrupted is protected by a mutex; for example, a cryptographic coprocessor. If the coprocessor was in-use and a higher priority task that also required the peripheral began to modify the control registers, any in-process results would be corrupted, and quite possibly the results for the new task as well. A mutex forces the higher priority task to wait until the lower priority task has finished using the protected resource.

Note that having different priority tasks competing for a finite resource is not recommended. Tasks requiring access to a common resource are usually placed at the same priority level, or a queuing mechanism would be put in place to sequence the access. However, as systems become more complex, this may not be enforceable, and a safety mechanism should be established[1].

Priority Inversion

While the use of a mutex resolves the basic issue of exclusive access, it creates a new problem: *priority inversion*. Priority inversion occurs when a higher priority task attempts to lock a mutex that is already locked by a lower priority task. The higher priority task is forced to wait until the lower priority task unlocks the mutex. However, if a mid-priority task begins to execute before the lower priority task has unlocked the mutex, then the high priority task is now a slave to the mid-level task; thus the inversion. The high priority task now faces an unpredictable delay before it can run, and may miss its execution deadline.

There needs to be some way for the higher priority task to essentially reserve its place in the scheduling queue for when the lower priority unlocks the mutex. The most common ways to accomplish this are through *priority inheritance* or with a *priority ceiling*.

Priority Inheritance

When priority inheritance is employed, if a lower priority task has locked a mutex, and a higher priority task attempts to lock that same mutex, the lower priority task *inherits* the priority of the higher priority task. It maintains this elevated priority until it unlocks the mutex, at which point it reverts to its original priority.

[1] Perhaps the most famous example of this is the Mars Pathfinder mission in 1997, which utilized a well established real-time kernel.

By assuming the higher priority, the task will be able to run to completion without being preempted by a mid-level task. Once the current task has completed, the blocked high priority task will immediately be scheduled to execute; whatever mid-level task was pending would need to wait for the high priority task to complete.

This mechanism works no matter how many different priority tasks attempt to lock the mutex. If a task has been promoted to a higher priority, and an even higher priority task attempts to lock the mutex, then the task owning the mutex will have its priority further promoted.

Priority Ceiling

The priority ceiling mechanism works by assigning a priority to a mutex that the task will adopt when it locks that mutex. The ceiling is usually set to the highest priority of any task that can lock the mutex, and any task locking the mutex will automatically execute at that elevated priority. This prevents a mid-level task from interrupting the task that has locked the mutex, and from delaying the execution of another task waiting for the mutex. However, since there is no guarantee that there would have been contention for the mutex, the mid-level task may have been needlessly blocked.

Simply put, priority ceiling automatically enforces the good design practice of having all tasks that require access to a common resource being at the same priority level.

Priority Propagation

In order for priority promotion to be effective, it must accommodate tasks that lock multiple mutexes simultaneously. If a task locks a mutex which causes its priority to be raised, and then attempts to lock an already owned mutex, its elevated priority must be propagated to that mutex owner, otherwise a deadlock can occur.

This is a natural extension of priority promotion, and should be part of any complete priority inversion protection mechanism.

Abassi Advantages

In addition to fully supporting priority inheritance and priority ceiling, including priority propagation, the Abassi real-time kernel corrects the shortcomings of traditional implementations, and adds many unique improvements which greatly increase reliability and robustness, and ease fault isolation.

Dynamic Priority Tracking

Most kernels statically increase the priority of the task that has locked a mutex when a higher priority task attempts to lock the same mutex. However, this simplistic approach ignores the complexity of modern systems.

The Abassi real-time kernel dynamically adjusts the priority elevation level, taking into account all tasks blocked on the mutex. Timeouts, task suspension, modifying a blocked task's priority; anything that modifies a task that is blocked on the mutex is automatically considered for the elevated task. This can result in the mutex owner's priority level being adjusted up or down, such that it always runs at the optimal promotion level.

Intelligently reacting to blocked task changes means no tasks are needlessly delayed, making your design more responsive and deterministic. And only the Abassi real-time kernel delivers this.

Optimal Priority Demotion

If a task only locks a single mutex, then the process of returning from an elevated priority level is very simple: the task reverts to its original priority when the mutex is unlocked. However, when multiple mutexes are locked, which may have different promotion priorities, and may also be unlocked in any order, the process of unwinding the task priority gets more complex.

Abassi is the only real-time kernel that properly considers the maximum priority level of all locked mutexes, as well as the priority of waiting tasks for mutexes with priority inheritance enabled. It ensures that tasks run at the optimal promotion level, even if priority aging and dynamic priority changes are used.

No other real-time kernel makes this guarantee, resulting in tasks that run at overly promoted levels, causing high priority tasks to remain needlessly blocked.

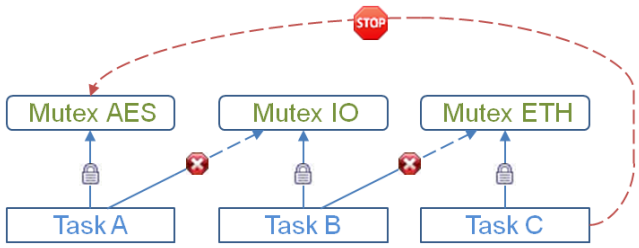
Deadlock Detection

Deadlock can occur when multiple tasks need to lock multiple mutexes. If one task locks a mutex needed by another task, and the other task has already locked a mutex needed by the first task, a deadlock occurs. Neither task is able to run and unlock a needed mutex.

While a deadlock can be avoided by judicious use of the priority ceiling, or broken by using a timeout, the Abassi real-time kernel can also automatically detect a deadlock situation. When the current task attempts to lock an already locked mutex, the kernel verifies if the mutex owner is already waiting on a mutex locked by the current task. Since this would trigger a deadlock, the Abassi kernel aborts the mutex lock request and returns an error to the user, such that corrective action can be taken.

More importantly, Abassi detects recursive mutex dependencies anywhere in the execution chain, not just directly connected tasks.

Consider a situation where 3 tasks access various coprocessors that are protected by mutexes: cryptography ("AES"), serial port ("IO"), and Ethernet ("ETH"). One task needs to decode a message and send it out the serial port. Another needs to accept characters from the serial port and send them over Ethernet. And the final task needs to encode a message and send it over Ethernet.



Task A locks the cryptography mutex, but gets blocked attempting to lock the serial port mutex. Task B locks the serial port mutex, but cannot lock the Ethernet mutex. And Task C locks the Ethernet mutex, and triggers the deadlock prevention mechanism when attempting to lock the cryptography mutex.

Deadlock detection is traditionally available only on very large general-purpose operating systems; Abassi is the only real-time kernel that provides this valuable fault detection mechanism. This inbuilt intelligence helps isolate complex logic errors that may be latent in a design, and greatly accelerates debugging.

Adaptive Priority Ceiling

Traditional priority ceiling implementations require the user to assign the mutex priority at creation time, and use that value immediately when a mutex is locked. The Abassi real-time kernel improves upon this by employing an adaptive mechanism that learns the priority ceiling value at run-time. Any time a higher priority task attempts to lock a mutex that is already locked, the kernel will track the newly found ceiling.

Abassi also defers priority elevation, such that uniquely locked mutexes do not needlessly run at an elevated priority. Only when additional tasks attempt to lock the mutex will the priority ceiling be applied.

Using the adaptive priority ceiling mechanism allows the designer flexibility in modifying task priorities within their design and having the kernel automatically adjust, eliminating a potential source of latent bugs.

Orphan Mutex Protection

If a task attempts to suspend another task that has locked one or more mutexes, then a potentially catastrophic deadlock can occur. The Abassi real-time kernel will automatically detect this error condition, and defer the task suspension until all mutexes have been unlocked.

By intelligently detecting and resolving this common design error, an extra level of robustness is added.

Conclusion

By employing a greenfield design approach, Code Time Technologies has been able to create a next generation real-time kernel that vastly improves upon any available today. The Abassi real-time kernel outperforms all existing platforms by combining code size and CPU efficiency with an unrivalled set of features.