How to use priority inheritance

Kyle Renwick and Bill Renwick - May 18, 2004

Fatal embraces, deadlocks, and obscure bugs await the programmer who isn't careful about priority inversions.

A preemptive real-time operating system (RTOS) forms the backbone of most embedded systems devices, from digital cameras to life-saving medical equipment. The RTOS can schedule an application's activities so that they appear to occur simultaneously. By rapidly switching from one activity to the next, the RTOS is able to quickly respond to real-world events.

To ensure rapid response times, an embedded RTOS can use preemption, in which a higher-priority task can interrupt a low-priority task that's running. When the high-priority task finishes running, the low-priority task resumes executing from the point at which it was interrupted. The use of preemption guarantees worst-case performance times, which enable use of the application in safety-critical situations.

Unfortunately, the need to share resources between tasks operating in a preemptive multitasking environment can create conflicts. Two of the most common problems are deadlock and priority inversion, both of which can result in application failure. In 1997, the Mars Pathfinder mission nearly failed because of an undetected priority inversion. When the rover was collecting meteorological data on Mars, it began experiencing system resets, losing data. The problem was traced to priority inversion. A solution to the inversion was developed and uploaded to the rover, and the mission completed successfully. Such a situation might have been avoided had the designers of the rover accounted for the possibility of priority inversion.[1]

This article describes in detail the problem of priority inversion and indicates two common solutions. Also provided are detailed strategies for avoiding priority inversion. Avoiding priority inversion is preferable to most other solutions, which generally require more code, more memory, and more overhead when accessing shared resources.

Priority inversion

Priority inversion occurs when a high-priority task is forced to wait for the release of a shared resource owned by a lower-priority task. The two types of priority inversion, bounded and unbounded, occur when two tasks attempt to access a single shared resource. A shared resource can be anything that must be used by two or more tasks in a mutually exclusive fashion. The period of time that a task has a lock on a shared resource is called the task's critical section or critical region.

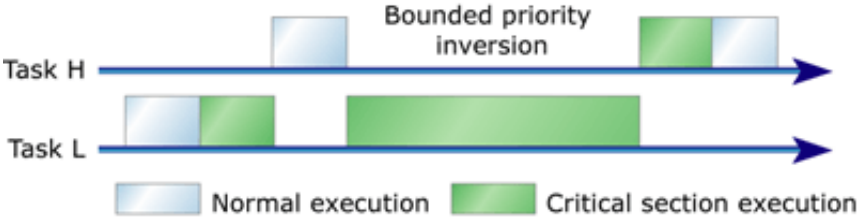

Figure 1: Bounded priority inversion

Bounded priority inversion, shown in Figure 1, occurs when low-priority Task L acquires a lock on a shared resource, but before releasing the resource is preempted by high-priority Task H.[2] Task H attempts to acquire the resource but is forced to wait for Task L to finish its critical section. Task L continues running until it releases the resource, at which point Task H acquires the resource and resumes executing. The worst-case wait time for Task H is equal to the length of the critical section of Task L. Bounded priority inversion won't generally hurt an application provided the critical section of

Task L executes in a timely manner.

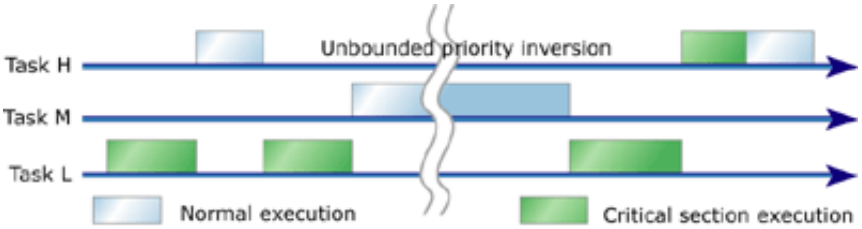

Figure 2: Unbounded priority inversion

Unbounded priority inversion, shown in Figure 2, occurs when an intervening task extends a bounded priority inversion, possibly forever.[2] In the previous example, suppose medium-priority Task M preempts Task L during the execution of Task L's critical section. Task M runs until it relinquishes control of the processor. Only when Task M turns over control can Task L finish executing its critical section and release the shared resource. This extension of the critical region leads to unbounded priority inversion. When Task L releases the resource, Task H can finally acquire the resource and resume execution. The worst-case wait time for Task H is now equal to the sum of the worst-case execution times of Task M and the critical section of Task L. Unbounded priority inversion can have much more severe consequences than a bounded priority inversion. If Task M runs indefinitely, neither Task L nor Task H will get an opportunity to resume execution.

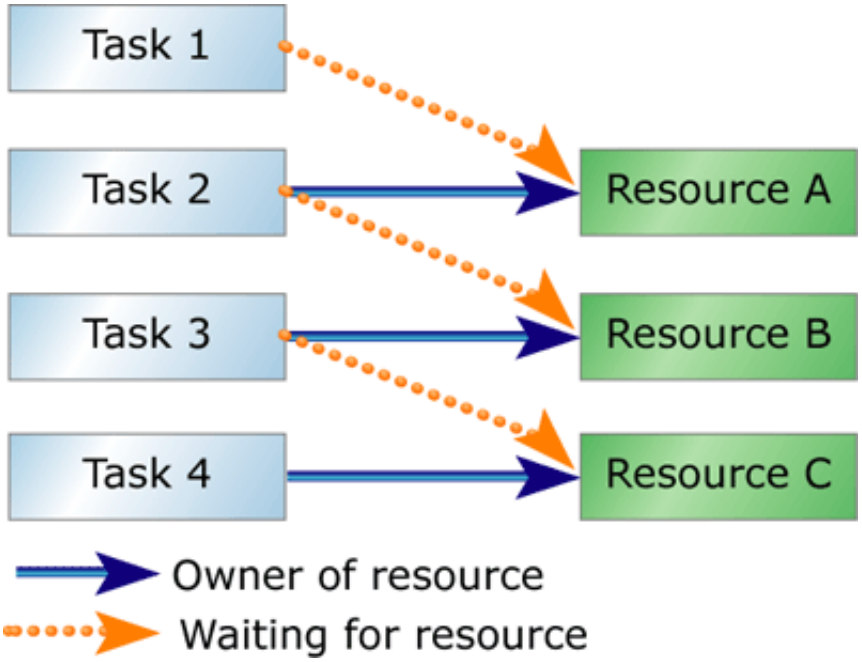

Figure 3: Chain of nested resource locks

Priority inversion can have an even more severe effect on an application when there are *nested resource locks*, as shown in Figure 3. Suppose Task 1 is waiting for Resource A. Resource A is owned by lower-priority Task 2, which is waiting for Resource B. Resource B is owned by still lower-priority Task 3, which is waiting for Resource C, which is being used by an even lower priority Task 4. Task 1 is blocked, forced to wait for tasks 4, 3, and 2 to finish their critical regions before it can begin execution. Such a chain of nested locks is difficult to resolve quickly and efficiently. The risk of an unbounded priority inversion is also high if many tasks intervene between tasks 1 and 4.

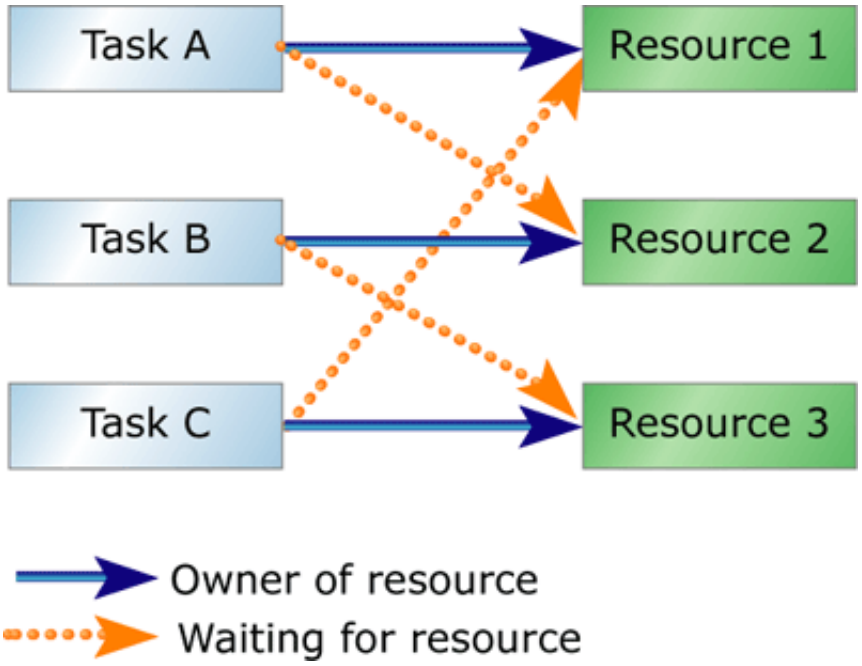

Figure 4: Deadlock

Deadlock

Deadlock, shown in Figure 4, is a special case of nested resource locks, in which a circular chain of tasks waiting for resources prevents all the tasks in the chain from executing.[2] Deadlocked tasks can have potentially fatal consequences for the application. Suppose Task A is waiting for a resource held by Task B, while Task B is waiting for a resource held by Task C, which is waiting for a resource held by Task A. None of the three tasks is able to acquire the resource it needs to resume execution, so the application is deadlocked.

Priority ceiling protocol

One way to solve priority inversion is to use the *priority ceiling protocol*, which gives each shared resource a predefined priority ceiling. When a task acquires a shared resource, the task is hoisted (has its priority temporarily raised) to the priority ceiling of that resource. The priority ceiling must be higher than the highest priority of all tasks that can access the resource, thereby ensuring that a task owning a shared resource won't be preempted by any other task attempting to access the same resource. When the hoisted task releases the resource, the task is returned to its original priority level. Any operating system that allows task priorities to change dynamically can be used to implement the priority ceiling protocol.[3]

A static analysis of the application is required to determine the priority ceiling for each shared resource, a process that is often difficult and time consuming. To perform a static analysis, every task that accesses each shared resource must be known in advance. This might be difficult, or even impossible, to determine for a complex application.

The priority ceiling protocol provides a good worst-case wait time for a high-priority task waiting for a shared resource. The worst-case wait time is limited to the longest critical section of any lower-priority task that accesses the shared resource. The priority ceiling protocol prevents deadlock by stopping chains of nested locks from developing.

On the downside, the priority ceiling protocol has poor average-case response time because of the significant overhead associated with implementing the protocol. Every time a shared resource is acquired, the acquiring task must be hoisted to the resource's priority ceiling. Conversely, every time a shared resource is released, the hoisted task's priority must be lowered to its original level. All this extra code takes time.

By hoisting the acquiring task to the priority ceiling of the resource, the priority ceiling protocol prevents locks from being contended. Because the hoisted task has a priority higher than that of any

other task that can request the resource, no task can contend the lock. A disadvantage of the priority ceiling protocol is that the priority of a task changes every time it acquires or releases a shared resource. These priority changes occur even if no other task would compete for the resource at that time.

Medium-priority tasks are often unnecessarily prevented from running by the priority ceiling protocol. Suppose a low-priority task acquires a resource that's shared with a high-priority task. The low-priority task is hoisted to the resource's priority ceiling, above that of the high-priority task. Any tasks with a priority below the resource's priority ceiling that are ready to execute will be prevented from doing so, even if they don't use the shared resource.

Priority inheritance protocol

An alternative to the priority ceiling protocol is the *priority inheritance protocol*, a variation that uses dynamic priority adjustments. When a low-priority task acquires a shared resource, the task continues running at its original priority level. If a high-priority task requests ownership of the shared resource, the low-priority task is hoisted above the requesting task. The low-priority task can then continue executing its critical section until it releases the resource. Once the resource is released, the task is dropped back to its original low-priority level, permitting the high-priority task to use the resource it has just acquired.[3]

Because the majority of locks in real-time applications aren't contended, the priority inheritance protocol has good average-case performance. When a lock isn't contended, priorities don't change; there is no additional overhead. However, the worst-case performance for the priority inheritance protocol is worse than the worst-case priority ceiling protocol, since nested resource locks increase the wait time. The maximum duration of the priority inversion is the sum of the execution times of all of the nested resource locks. Furthermore, nested resource locks can lead to deadlock when you use the priority inheritance protocol. That makes it important to design the application so that deadlock can't occur.

Nested resource locks should obviously be avoided if possible. An inadequate or incomplete understanding of the interactions between tasks can lead to nested resource locks. A well-thought-out design is the best tool a programmer can use to prevent these.

You can avoid deadlock by allowing each task to own only one shared resource at a time. When this condition is met, the worst-case wait time matches the priority ceiling protocol's worst-case wait. In order to prevent misuse, some operating systems that implement priority inheritance don't allow nested locks. It might not be possible, however, to eliminate nested resource locks in some applications without seriously complicating the application.

But remember that allowing tasks to acquire multiple priority inheritance resources can lead to deadlock and increase the worst-case wait time.

Priority inheritance is difficult to implement, with many complicated scenarios arising when two or more tasks attempt to access the same resources. The algorithm for resolving a long chain of nested resource locks is complex. It's possible to incur a lot of overhead as hoisting one task results in hoisting another task, and another, until finally some task is hoisted that has the resources needed to run. After executing its critical section, each hoisted task must then return to its original priority.

Figure 5 shows the simplest case of the priority inheritance protocol in which a low-priority task acquires a resource that's then requested by a higher priority task. Figure 6 shows a slightly more complex case, with a low-priority task owning a resource that's requested by two higher-priority tasks. Figure 7 demonstrates the potential for complexity when three tasks compete for two resources.

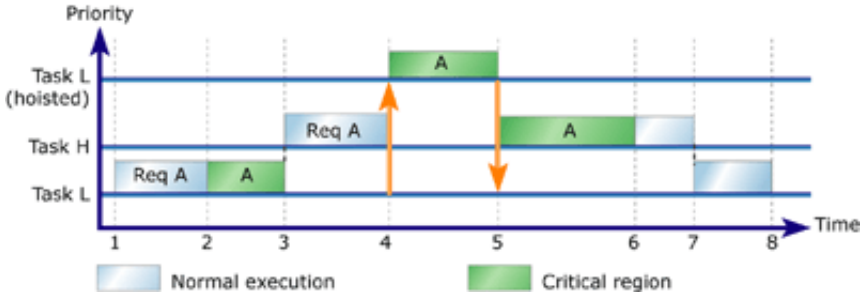

Figure 5: Simple priority inheritance

1. Task L receives control of the processor and begins executing.
    - The task makes a request for Resource A.
2. Task L is granted ownership of Resource A and enters its critical region.
3. Task L is preempted by Task H, a higher-priority task.
    - Task H begins executing and requests ownership of Resource A, which is owned by Task L.
4. Task L is hoisted to a priority above Task H and resumes executing its critical region.
5. Task L releases Resource A and is lowered back to its original priority.
    - Task H acquires ownership of Resource A and begins executing its critical region.
6. Task H releases Resource A and continues executing normally.
7. Task H finishes executing and Task L continues executing normally.
8. Task L finishes executing.

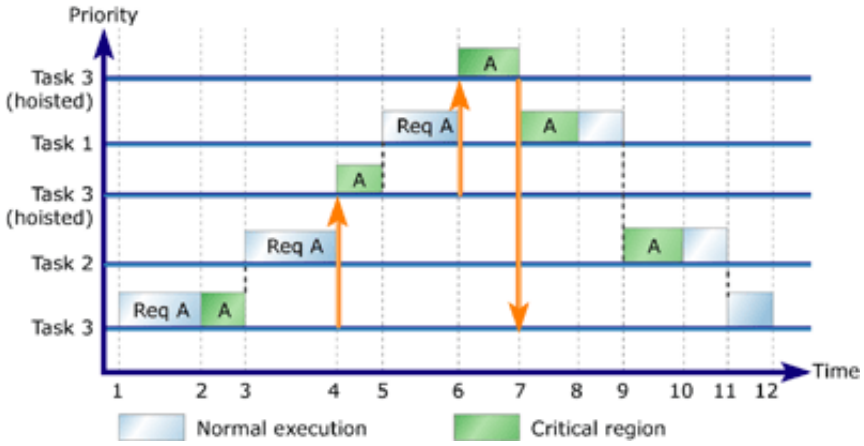

Figure 6: Three-task, one-resource priority inheritance

1. Task 3 gets control of the processor and begins executing.
    - The task requests ownership of Resource A.
2. Task 3 acquires Resource A and begins executing its critical region.
3. Task 3 is preempted by Task 2, a higher-priority task.
    - Task 2 begins executing normally and requests Resource A, which is owned by Task 3.
4. Task 3 is hoisted to a priority above Task 2 and resumes executing its critical region.
5. Task 3 is preempted by Task 1, a higher-priority task.
    - Task 1 begins executing and requests Resource A, which is owned by Task 3.
6. Task 3 is hoisted to a priority above Task 1.
    - Task 3 resumes executing its critical region.
7. Task 3 releases Resource A and is lowered back to its original priority.
    - Task 1 acquires ownership of Resource A and begins executing its critical region.
8. Task 1 releases Resource A and continues executing normally.
9. Task 1 finishes executing. Task 2 acquires Resource A and begins executing its critical region.
10. Task 2 releases Resource A and continues executing normally.
11. Task 2 finishes executing. Task 3 resumes and continues executing normally.
12. Task 3 finishes executing.

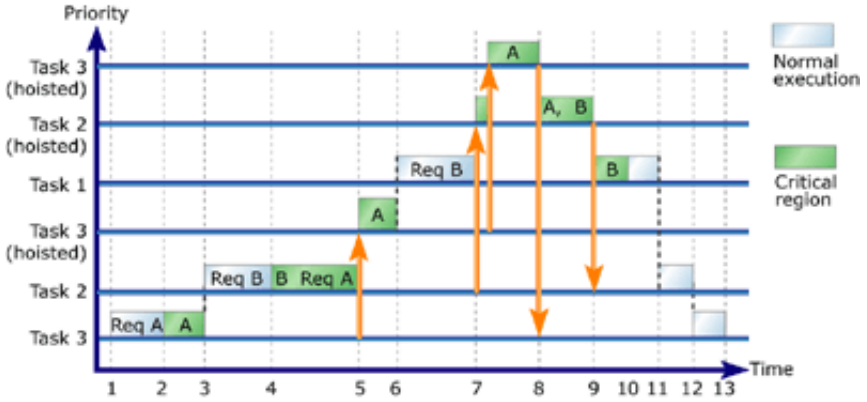

Figure 7: Three-task, two-resource priority inheritance

1. Task 3 is given control of the processor and begins executing. The task requests Resource A.
2. Task 3 acquires ownership of Resource A and begins executing its critical region.
3. Task 3 is preempted by Task 2, a higher-priority task. Task 2 requests ownership of Resource B.
4. Task 2 is granted ownership of Resource B and begins executing its critical region.
    - The task requests ownership of Resource A, which is owned by Task 3.
5. Task 3 is hoisted to a priority above Task 2 and resumes executing its critical region.
6. Task 3 is preempted by Task 1, a higher-priority task.
    - Task 1 requests Resource B, which is owned by Task 2.
7. Task 2 is hoisted to a priority above Task 1. However, Task 2 still can't execute because it must wait for Resource A, which is owned by Task 3.
    - Task 3 is hoisted to a priority above Task 2 and continues executing its critical region.
8. Task 3 releases Resource A and is lowered back to its original priority.
    - Task 2 acquires ownership of Resource A and resumes executing its critical region.
9. Task 2 releases Resource A and then releases Resource B. The task is lowered back to its original priority.
    - Task 1 acquires ownership of Resource B and begins executing its critical region.
10. Task 1 releases Resource B and continues executing normally.
11. Task 1 finishes executing. Task 2 resumes and continues executing normally.
12. Task 2 finishes executing. Task 3 resumes and continues executing normally.
13. Task 3 finishes executing.

Manage resource ownership

Most RTOSes that support priority inheritance require resource locks to be properly nested, meaning the resources must be released in the reverse order to that in which they were acquired. For example, a task that acquired Resource A and then Resource B would be required to release Resource B before releasing Resource A.

Figure 7 provides an example of priority inheritance in which two resources are released in the opposite order to that in which they were acquired. Task 2 acquired Resource A before Resource B; Resource B was then released before Resource A. In this example, Task 2 was able to release the resources in the proper order without adversely affecting the application.

Many operating systems require resources to be released in the proper order because it's difficult to implement the capability to do otherwise. However, situations occur in which releasing the resources in the proper order is neither possible nor desirable. Suppose there are two shared resources: Resource B can't be acquired without first owning Resource A. At some point during the execution of the critical region with Resource B, Resource A is no longer needed. Ideally, Resource A would now be released. Unfortunately, many operating systems don't allow that. They require Resource A to be held until Resource B is released, at which point Resource A can be released. If a higher-priority task is waiting for Resource A, the task is kept waiting unnecessarily while the resource's current owner executes.

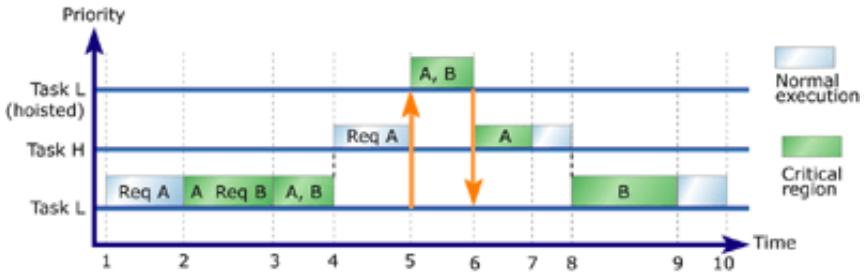

Figure 8: Managing resource ownership (example 1)

1. Task L is given control of the processor and begins executing. Task L requests Resource A.
2. Task L acquires ownership of Resource A and begins executing its critical region.
    - Task L requests Resource B.
3. Task L acquires ownership of Resource B and continues executing its critical region.
4. Task L is preempted by Task H, a higher-priority task.
    - Task H requests ownership of Resource A, which is owned by Task L.
5. Task L is hoisted above Task H and continues executing its critical region.
6. Task L releases Resource A even though it was acquired before Resource B.
    - Task L is lowered to its original priority level.
    - Task H acquires Resource A and begins executing its critical region.
    - Note that low-priority Task L no longer prevents Task H from running, even though Task L still owns Resource B.
7. Task H releases Resource A and continues executing normally.
8. Task H finishes executing and Task L continues executing its critical region.
9. Task L releases Resource B and continues executing normally.
10. Task L finishes executing.

In Figure 8, Task L releases its resources in the same order they were acquired, which most priority inheritance implementations wouldn't allow. Upon releasing Resource A, Task L drops to its original priority level. This allows Task H to acquire Resource A, ending the priority inversion. After Task H has released the resource and stopped executing, Task L can continue executing with Resource B, on which it has an uncontested lock. If Task L had been required to release Resource B before releasing Resource A, Task H would have been prevented from running until Task L had released both resources. This would have unnecessarily lengthened the duration of the bounded priority inversion.

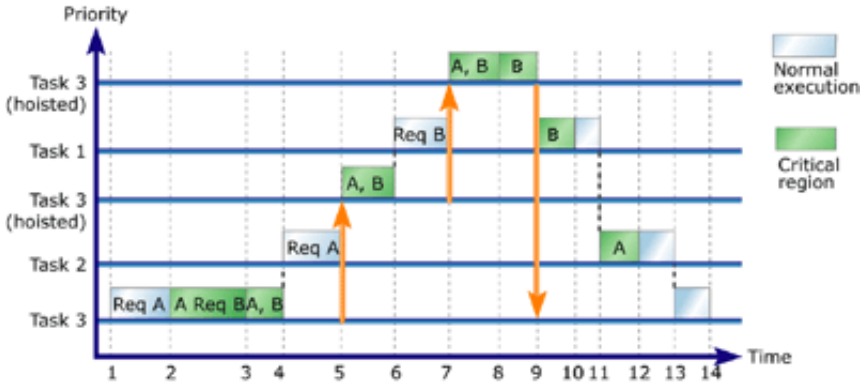

Figure 9: Managing resource ownership (example 2)

1. Task 3 is given control of the processor and begins executing. Task 3 requests Resource A.
2. Task 3 acquires ownership of Resource A and begins executing its critical region.
    - Task 3 requests Resource B.
3. Task 3 acquires ownership of Resource B and continues executing its critical region.
4. Task 3 is preempted by Task 2, a higher-priority task.
    - Task 2 requests ownership of Resource A, which is owned by Task 3.
5. Task 3 is hoisted above Task 2 and continues executing its critical region.
6. Task 3 is preempted by Task 1, a higher-priority task.
    - Task 1 requests Resource B, which is owned by Task 3.

7. Task 3 is hoisted above Task 1 and continues executing its critical region.
8. Task 3 releases Resource A and continues executing with Resource B.
    - Note that Task 3 is not dropped to either its previous priority or its original priority. To do so would immediately produce a priority inversion that requires Task 3 to be hoisted above Task 1 because Task 3 still owns Resource B.
9. Task 3 releases Resource B and is lowered to its original priority level.
    - Task 1 acquires Resource B and continues executing its critical region.
10. Task 1 releases Resource B and continues executing normally.
11. Task 1 finishes executing and Task 2 acquires Resource A. Task 2 begins executing its critical region.
12. Task 2 releases Resource A and continues executing normally.
13. Task 2 finishes executing and Task 3 continues executing normally.
14. Task 3 finishes executing.

The difficulty of implementing locks that aren't properly nested becomes apparent when three tasks compete for two resources, as seen in Figure 9. When Resource A is released at Time 8, low-priority Task 3 remains hoisted to the highest priority. An improperly designed priority inheritance protocol would lower Task 3 to its original priority level, which was the task's priority before acquiring Resource A. Task 3 would then have to be immediately hoisted above Task 1 to avoid a priority inversion, because of the contention for access to Resource B. Unbounded priority inversion could occur while Task 3 is momentarily lowered. A medium-priority task could preempt Task 3, extending the priority inversion indefinitely.

The example in Figure 8 shows why it's sometimes desirable to release nested resources "out of order," or not the reverse of the order in which they were acquired. Although such a capability is clearly advantageous, many implementations of the priority inheritance protocol only support sequentially nested resource locks.

The example in Figure 9 helps show why it's more difficult to implement priority inheritance while allowing resources to be released in any order. If a task owns multiple shared resources and has been hoisted several times, care must be taken when the task releases those resources. The task's priority must be adjusted to the appropriate level. Failure to do so may result in unbounded priority inversion.

Avoid inversion

The best strategy for solving priority inversion is to design the system so that inversion can't occur. Although priority ceilings and priority inheritance both prevent unbounded priority inversion, neither protocol prevents bounded priority inversion. Priority inversion, whether bounded or not, is inherently a contradiction. You don't want to have a high-priority task wait for a low-priority task that holds a shared resource.

Prior to implementing an application, examine its overall design. If possible, avoid sharing resources between tasks at all. If no resources are shared, priority inversion is precluded.

If several tasks do use the same resource, consider combining them into a single task. The sub-tasks can access the resource through a state machine in the combined task without fear of priority inversion. Unless the competing sub-tasks are fairly simple, however, the state machine might be too complex to justify.

Another way to prevent priority inversion is to ensure that all tasks that access a common resource have the same priority. Although one task might still wait while another task uses the resource, no priority inversion will occur because both tasks have the same priority. Of course, this only works if the RTOS provides a non-preemptive mechanism for gracefully switching between tasks of equal priority.

If you can't use any of these techniques to manage shared resources, consider giving a "server task" sole possession of the resource. The server task can then regulate access to the resource. When a "client task" needs the resource, it must call upon the server task to perform the required operations and then wait for the server to respond. The server task must be at a priority greater than that of the highest-priority client task that will access the resource. This method of controlling access to a

resource is similar to the priority ceiling protocol and requires static analysis to determine the priority of the server task. The method relies on RTOS message passing and synchronization services instead of resource locks and dynamic task-priority adjustments.

Prioritize

Priority inversion is a serious problem that, if allowed to occur, can cause a system to fail. It's generally simpler to avoid priority inversion than to solve it in software. If possible, eliminate the need for shared resources altogether, avoiding any chance of priority inversion. If you can't avoid priority inversion, at least make sure it's bounded. Unbounded priority inversions can leave high-priority tasks unable to execute, resulting in application failure. Two common methods of bounding priority inversion are the priority ceiling protocol and the priority inheritance protocol. Neither protocol is perfect for all situations. Hence, good analysis and design are always necessary to understand which solution, or combination of solutions, is needed for your particular application.

W.L. (Bill) Renwick is the founder and chief engineer of KADAK Products Ltd. He has over 40 years experience in RTOS design and real-time embedded software. He earned a BS in electrical engineering from University of Waterloo, Canada, and an MS in control theory from University of London, UK. Bill can be reached at amxtech@kadak.com.

Kyle Renwick researched and authored this article while working at KADAK as a student of Systems Engineering at University of Waterloo. Kyle can be reached at amxtech@kadak.com.

References

1. Jones, Michael. "What really happened on Mars?"Redmond, Washington: Microsoft: 1997. http://research.microsoft.com/~mbj/Mars_Pathfinder/
2. Douglass, Bruce Powel. *Design Patterns for Real-Time Systems*. Boston: Addison-Wesley Professional: 2002.
3. Sha, L., R. Rajkumar, and S. Sathaye. "Priority inheritance protocols: An approach to real-time synchronization." *IEEE Transactions on Computers*, 39 (9), 1175-1185: 1990.