Mutex Tech Note

Mutexes provide a high level of safety for mutual exclusion

by Ralph Moore
smx Architect

Introduction

In the following sections, we will discuss the problems caused by using a *counting semaphore* or a *binary semaphore* for protection, and show how the mutual exclusion semaphore ("*mutex*") solves them.

Protection Failure

A typical counting semaphore has an internal counter, which can be counted higher than one by *signals*. This is useful when counting events or regulating access to multiple resources. However, it can backfire when using a counting semaphore for access protection for a single, non-reentrant resource. If for some reason, an extra signal (*spurious signal)* is sent to the counting semaphore, then it will allow two tasks to access such a resource, thus failing to do its job.

Both a binary semaphore and a mutex have only two states: *free* and *owned*. When in the free state, spurious signals are ignored. Hence, safety is improved, because two tasks will not accidentally be given access to a protected resource just because a spurious signal occurred.

Task Lockup

Consider the following code example, which uses a counting or binary semaphore:

```
functionA()
{
  SemWait(semA, INF);
  functionX();
  SemSignal(semA);
}

functionX()
{
  SemWait(semA, INF);
  ....
  SemSignal(semA);
}
```

In this example, both functionA and functionX need access to a resource that is protected by semA. functionA, while accessing this resource, calls functionX. When this happens the current task

(which is running functionA) is suspended indefinitely[1] on semA because functionX tests semA a second time. This happens because semA does not know that the current task already owns it.

For this, a mutex has an advantage over a counting or binary semaphore. When owned, the mutex knows which task owns it. A second test by the same task will not fail and the task will not be suspended. Hence, the above code, which could easily occur when using a *thread-safe* library, will not cause the current task to freeze.

Another kernel provides a *resource* semaphore, which is the same as a mutex, except that it gives an error if a task tests it a second time. This is intended to catch unmatched tests vs. signals. However, this is not the more likely problem – it is natural to put matched tests and signals into a routine. Non-matches can be found simply by searching for SemSignal and SemTest for the same semaphore — the two counts should be equal. The more difficult problem to detect is the one described above because the user of a library may not be aware that one function calls another and that both test the same mutex.

Premature Release

We are still not out of the woods with respect to the above example. It is not acceptable for semA to be released by the signal in functionX, because functionA, which called functionX, is still in its critical section. For every test, there must be a matching signal, before the mutex can become free. This is assured by having an internal *nesting counter* in the mutex. This counter is incremented by every test from the owner task and decremented by every signal from the owner task. Only tests and signals from the owner task can change the nesting counter. When it reaches zero, the mutex is freed.

Unbounded Priority Inversion

The next problem with using counting or binary semaphores for controlling access to critical resources is called *unbounded priority inversion*. Priority inversion occurs when a low-priority task owns a semaphore, and a high-priority task is forced to wait on the semaphore until the low-priority task releases it. If, prior to releasing the semaphore, the low priority task is preempted by one or more mid-priority tasks, then unbounded priority inversion has occurred because the delay of the high-priority task is no longer predictable. This defeats Deadline Monotonic Analysis (DMA) because it is not possible to predict if the high-priority task will meet its deadline.

Sharing a critical resource between high and low priority tasks is not a desirable design practice. It is better to share a resource only among equal priority tasks or to limit resource accesses to a single *resource server* task. Examples are a print server task and a file server task. We have long advocated this practice. However, with the layering of increasingly diverse and complicated middleware onto RTOSs, it is becoming impractical to enforce such simple strategies. Hence, in the interest of safety, it is best to implement some method of preventing unbounded priority inversion.

[1] Note that appropriate timeouts (non-infinite) should be used. In this case, the task would lock up only for the timeout period, but functionX() would always fail because it would never be able to get the semaphore.

Task Priorities

Dealing with priority inversion is not a simple matter. In the following sections, we will discuss the two principal means for implementing *priority promotion*, followed by the approach chosen for *smx*, and then we will discuss the equally difficult problem of *priority demotion*.

Priority Inheritance

The most common approach is *priority inheritance*. Since the mutex knows its current owner, it is possible to promote the priority of the owner whenever a higher-priority task starts waiting on the mutex. The current owner temporarily assumes the priority of the highest priority task waiting on the mutex. This allows the owner to resume running if it was preempted by a mid-priority task, and to continue running should a mid-priority task become ready to run. When the owner releases the mutex, it drops back to its original priority.

Priority inheritance is used by most RTOSs that offer protection against unbounded priority inversion.

Problems with Priority Inheritance

Unfortunately, priority inheritance has some problems:

1. Since promotion of low-priority tasks to higher priority levels is caused by random sequences of events, it is not possible to predict how long mid-priority tasks will be blocked by lower-priority promoted tasks. Hence, Deadline Monotonic Analysis cannot guarantee that mid-priority tasks will meet their deadlines.
2. Priority inheritance can cause extra task switching. For example, if L, M, and H are three tasks of priorities low, medium, and high, respectively, and L(x) is task L promoted to priority x, then the following task switches could occur: L → M → L(m) →H → L(h) → H → M → L = 7 task switches. (Explanation: task L runs and gets mutex, Mtx. Task M preempts L and trys to get Mtx. Since L owns Mtx, its priority is promoted to m and M waits. Then Task H preempts and tries to get Mtx. It too, must wait and task L is promoted to priority h. When Task L releases Mtx, task H gets it and immediately preempts L. When task H releases Mtx, task M gets it, but does not run until H stops running. When task M releases Mtx and stops running, task L runs.) More mid-level priorities could result in even more task switches. If L were boosted to H when it first got the mutex, then only 4 task switches would occur: L →L(h) → H → M → L.
3. If a promoted task is waiting for another mutex, then that task's owner must have its priority promoted. This is called *priority propagation* and it must be implemented for priority inheritance to be effective, if tasks can own multiple mutexes simultaneously. However it increases complexity and reduces determinacy. Thus many RTOSs do not implement it.
4. The extra task switching shown in #2 can become even worse if an even higher-priority task preempts the low-priority task and waits for the mutex. Then the low-priority task's priority must be raised again and possibly propagated again.

5. As is clear from the foregoing, complex systems consisting of many mutexes and many tasks that can own several mutexes at once, can experience significant reductions in determinacy. This is not wrong, but it should be taken into account.
6. Priority inheritance increases the likelihood of deadlocks, because priorities change unpredictably.

Priority Ceiling Promotion

An alternative to priority inheritance is *priority ceiling* promotion. With it, a priority *ceiling* is specified for a mutex, when it is created. The ceiling is set equal to the priority of the highest-priority task that is expected to get the mutex. When a lower-priority task obtains the mutex, its priority is *immediately* boosted to the mutex's ceiling. Hence, a mid-priority task cannot preempt as long as the low-priority task owns the mutex, nor can any other task preempt that wants the mutex. Interestingly, priority ceiling is a simply an automatic method for forcing tasks to be of the same priority while using a resource – i.e. it enforces a good design practice.

Priority ceiling permits Deadline Monotonic Analysis because any task may be blocked only once by a lower priority task, before it can run. Task switching is also reduced. The following task switches would occur for the previous example: L →L(H) → H → M → L = 4 task switches vs. 7, maximum, for priority inheritance.

If all mutexes, in a group, have the same ceiling, then once a task gains access to one mutex it has access to all mutexes in the group. (This is because: 1. All other tasks wanting these mutexes are blocked from running, and 2. This task could not be running if any other task already owned one of the mutexes.) This reduces task switching and improves determinacy.

Problems with Priority Ceiling Promotion

Unfortunately, priority ceiling also has some problems:

1. It causes a low-priority task to be elevated to high priority whenever it owns a resource shared by a high-priority task. If the high-priority task seldom uses the resource and the low-priority task frequently uses the resource, this needlessly blocks mid-priority tasks from running. In this situation, priority inheritance is more attractive.
2. A mutex ceiling is static. It is possible that a user task may have had its priority increased by the programmer, or increased dynamically, to a value above the mutex's ceiling. Then priority ceiling promotion would not be working fully and some priority inversion could occur.

A Combined Solution

To achieve the best results, *smx* mutexes implement a combination of both methods. This is done as follows: A mutex is created with a ceiling, *ceil,* and priority inheritance enabled or disabled by a flag in the mutex. If the priority of a new owner is less than *ceil*,

its priority is immediately promoted to *ceil*. If, another task waits on the mutex, its priority exceeds *ceil*, and priority inheritance is enabled, then the owner's priority is promoted to that of the new waiting task.

This approach permits the main advantages of ceiling priority for most mutexes, yet allows priority inheritance to be used when appropriate for best performance or protection. If the mutex ceiling is set to the priority of the highest task that can possibly get it, then inheritance is effectively disabled and the mutex operates purely in ceiling mode. Alternatively, ceiling mode may be disabled by setting the ceiling to zero, when the mutex is created. Then the mutex will operate purely in inheritance mode, if inheritance is enabled. If ceiling and inheritance are not enabled, the mutex will operate without priority promotion.

In the event that a waiting task is promoted, by some other means, to a priority above the mutex *ceil*, priority promotion will still occur for the task due to priority inheritance, if enabled. Hence, safety is improved by the judicious combination of ceiling and inheritance priority promotion.

Staggered Priority Demotion

Staggered priority demotion, when a task releases a mutex, is a non-trivial operation. Ideally the task's priority should not be higher than necessary, else high priority tasks may be needlessly blocked. But, each mutex owned by the task may require a different promotion level and mutexes can be released in any order. If the task's priority is demoted too far too soon, the protection of priority promotion will be lost for one or more of the remaining mutexes.

Further complicating demotion, *smx* supports two other mechanisms for dynamic priority change: bump_task() and message reception from a pass exchange. Either of these can occur while a task owns one or more mutexes.

In order to deal with these problems, each task maintains a list of the mutexes that it owns. When releasing a mutex, the task priority is demoted to the greatest of: (1) the highest *ceiling* among remaining owned mutexes, (2) the highest waiting task priority for owned mutexes with inheritance enabled, or (3) the task's normal priority, *normpri*. The latter stores the task's original priority, before any mutexes were acquired, as modified, subsequently, by bump_task() or message reception from a pass exchange.

Operation

The following sections discuss operational considerations.

Deadlock Prevention

Deadlocks can be a problem if two or more tasks must own two or more mutexes to complete their operations. A deadlock occurs if one task owns a mutex needed by the other task and the other task owns a mutex needed by the first task. Then, neither task can run. Deadlocks are commonly broken by task timeouts. However, when a timeout occurs, time has been lost and may result in a missed deadline, so this is not an optimum solution.

Priority ceiling provides an easy way to avoid deadlocks. If it is known that a group of mutexes is shared by a group of tasks, then deadlocks can be prevented by assigning *MaxCeil* to all mutexes in the group, where *MaxCeil* equals the highest priority of any task in the group of tasks. When one of the tasks gets any of the mutexes, its priority is promoted to *MaxCeil* and none of the other tasks in the task group can preempt it. Hence, none of them can get any of the mutexes in the mutex group. Thus, the first task can get any other mutexes in the mutex group, that it needs, without risk of deadlock. Because task priorities can be externally promoted, use of *MaxCeil* is not infallible. This technique also breaks down if the programmer increases a task priority and forgets to increase *MaxCeil*.

Another technique for deadlock prevention may be implemented in the next release of *smx*. Prior to waiting on an owned mutex, a check is made to determine if its owner is waiting on a mutex owned by the current task. (This is easily done by using the mutex owned list of the current task.) If this condition is found, the wait operation is aborted and the DEADLOCK_BROKEN error is reported. When this happens, the current task must free all mutexes it owns and start over acquiring them. This method will prevent first-level deadlocks but not second-level (mutex's owner waiting on a mutex whose owner is waiting on a mutex owned by the current task) deadlocks, and higher. However, it could be extended to do so. The question is whether this is of sufficient value to compensate for reduced performance and loss of determinism. Possibly it is, since deadlocks present a serious reliability problem.

In view of the above shortcomings, the following rules should be followed, where possible:

1. Tasks should get mutexes in the same order and release them in the reverse order.
2. Always use timeouts on MutexGet() calls.

Owner Preempted, Deleted, Suspended, or Stopped

Preemption of an owner task by a higher-priority task is allowed and is consistent with DMA. If a task is deleted, every mutex it owns is automatically freed. This guards against design errors. The situation for a task suspending or stopping to wait for another resource is more difficult to decide — see the discussion below. However a task suspending or stopping another task that owns a mutex is probably a design error. Currently, *smx* does not prevent this. Doing so will be considered for the next release.

Mutex Owner Wait Blocking

For a mutex owner to suspend or stop itself to wait for another resource is undesirable. This can cause unbounded priority inversion, with the result that a higher-priority task, needing the resource, misses its deadline. This might occur only sporadically and thus be very hard to track down. As a general rule, a task should not hold a shared resource any longer than necessary and should not perform an smx operation that could result in it waiting, if it owns a mutex.

For simple systems, it may be best to allow the programmer to decide if it is ok for a task to suspend or stop if it owns a mutex. However, this may not be acceptable in systems that are more complex. To allow either, it may be desirable to implement a MTX_NO_WAIT task flag. If set, when a task owns a mutex, any attempt by the task to suspend or stop itself fails with an error. A properly written task will detect that the operation has failed and take appropriate action. In addition, a breakpoint on the error can be used to find the badly written task. This provides safety against potentially debilitating programming errors.

When MTX_NO_WAIT is set and the task needs multiple resources and therefore must sequentially test mutexes, it is necessary to use a common *MaxCeil*, as explained above. In the case where a task needs another type of resource, e.g. a message, it should obtain that resource before getting a mutex.

Implementing a MTX_NO_WAIT flag will be considered for the next *smx* release. It may also be used to address the external suspend/stop problem, discussed above.

Summary of Methods for Mutual Exclusion

1. *Counting Semaphore*. Adequate for minimal systems, but has the following limitations: protection failure due to spurious signals, task lockup, premature release, and unbounded priority inversion.
2. *Binary Semaphore*. Better because it ignores spurious signals, but it still has these limitations: task lockup, premature release, and unbounded priority inversion.
3. *Self Promotion*. Assign a priority to a resource, which equals that of the highest priority task using it. Then, any task needing the resource temporarily boosts its own priority to this level before obtaining the resource. No mutex or semaphore is needed. This accomplishes the exclusion function of a priority ceiling mutex, without the overhead. We have long advocated this approach.
4. *Simple Mutex*. Permits nesting, but does not deal with priority inversion.
5. *Priority Inheritance Mutex*. Less blocking of mid-priority tasks than priority ceiling, but can lead to excessive task switches, thus increasing overhead.
6. *Priority Ceiling Mutex*. Less task switches than inheritance, but may block mid-priority tasks too much. Provides a good method to prevent deadlocks.

7. *Task Preemption Threshold*. Automatically accomplishes the same thing as priority ceiling or self-promotion. However, locks out mid-priority tasks for the entire time that the task is running. Hence, not a desirable alternative.
8. *Priority Ceiling and Inheritance Mutex* (*smx*). An optimum blend. Can be adjusted by the user to achieve the best results for each mutex.