Sticky Bits – Powered by Feabhas
A blog looking at developing software for real-time and embedded systems

Mutex vs. Semaphores – Part 2: The Mutex

Posted on September 11, 2009 by Niall Cooling

In Part 1 of this series we looked at the history of the binary and counting semaphore, and then went on to discuss some of the associated problem areas. In this posting I aim to show how a different RTOS construct, the mutex, may overcome some, if not all, of these weaknesses.

To address the problems associated with semaphore, a new concept was developed during the late 1980's. I have struggled to find it's first clear definition, but the major use of the term *mutex* (another neologism based around MUTual EXclusion) appears to have been driven through the development of the common programming specification for UNIX based systems. In 1990 this was formalised by the IEEE as standard IEEE Std 1003.1 commonly known as POSIX.

The mutex is similar to the principles of the binary semaphore with one significant difference: **the principle of ownership**. Ownership is the simple concept that when a task locks (acquires) a mutex only it can unlock (release) it. If a task tries to unlock a mutex it hasn't locked (thus doesn't own) then an error condition is encountered and, most importantly, the mutex is not unlocked. If the mutual exclusion object doesn't have ownership then, irrelevant of what it is called, it is not a mutex.

The concept of ownership enables mutex implementations to address the problems discussed in part 1:

1. Accidental release
2. Recursive deadlock
3. Task-Death deadlock
4. Priority inversion
5. Semaphore as a signal

Accidental Release

As already stated, ownership stops accidental release of a mutex as a check is made on the release and an error is raised if current task is not owner.

Recursive Deadlock

Due to ownership, a mutex can support relocking of the same mutex by the owning task as long as it is released the same number of times.

Priority Inversion

With ownership this problem can be addressed using one of the following priority inheritance protocols:

- [Basic] Priority Inheritance Protocol
- Priority Ceiling Protocol

The *Basic Priority Inheritance Protocol* enables a low-priority task to inherit a higher-priorities task's priority if this higher-priority task becomes blocked waiting on a mutex currently owned by the low-priority task. The low priority task can now run and unlock the mutex – at this point it is returned back to its original priority.

The details of the Priority Inheritance Protocol and Priority Ceiling Protocol (a slight variant) will be covered in part 3 of this series.

Death Detection

If a task terminates for any reason, the RTOS can detect if that task current owns a mutex and signal waiting tasks of this condition. In terms of what happens to the waiting tasks, there are various models, but two doiminate:

- All tasks readied with error condition;
- Only one task readied; this task is responsible for ensuring integrity of critical region.

When *all tasks* are readied, these tasks must then assume critical region is in an undefined state. In this model no task currently has ownership of the mutex. The mutex is in an undefined state (and cannot be locked) and must be reinitialized.

When *only one task* is readied, ownership of the mutex is passed from the terminated task to the readied task. This task is now responsible for ensuring integrity of critical region, and can unlock the mutex as normal.

Mutual Exclusion / Synchronisation

Due to ownership a mutex cannot be used for synchronization due to lock/unlock pairing. This makes the code cleaner by not confusing the issues of mutual exclusion with synchronization.

Caveat

A specific Operating Systems mutex implementation **may or may not** support the following:

- Recursion
- Priority Inheritance
- Death Detection

Review of some APIs

It should be noted that many Real-Time Operating Systems (or more correctly Real-Time Kernels) do not support the concept of the mutex, only supporting the *Counting Semaphore* (e.g. MicroC/OS-II). [CORRECTION: The later versions of uC/OS-II do support the mutex, only the original version did not].

In this section we shall briefly examine three different implementations. I have chosen these as they represent the broad spectum of APIs offered (*Footnote 1*):

- VxWorks Version 5.4
- POSIX Threads (pThreads) – IEEE Std 1003.1, 2004 Edition
- Microsoft Windows Win32 – Not .NET

VxWorks from Wind River Systems is among the leading commercial Real-Time Operating System used in embedded systems today. POSIX Threads is a widely supported standard, but has become more widely used due to the growth of the use of Embedded Linux. Finally Microsoft Window's common programming API, Win32 is examined. Windows CE, targeted at embedded development, supports this API.

However, before addressing the APIs in detail we need to introduce the concept of a *Release Order Policy*. In Dijkstra's original work the concept of task priorities were not part of the problem domain. Therefore it was assumed that if more than one task was waiting on a held semaphore, when released the next task to acquire the semaphore would be chosen on a First-Come-First-Server (First-In-First-Out; FIFO) policy. However once tasks have priorities, the policy may be:

- FIFO – waiting tasks ordered by arrival time
- Priority – waiting tasks ordered by priority
- Undefined – implementation doesn't specify

VxWorks v5.4

VxWorks supports the Binary Semaphore, the Counting Semaphore and the Mutex (called the Mutual-Exclusion Semaphore in VxWorks terminology). They all support a common API for acquiring (semTake) and releasing (semGive) the particular semaphore. For all semaphore types, waiting tasks can be queued by priority or FIFO and can have a timeout specified.

The *Binary Semaphor*e has, as expected, no support for recursion or inheritance and the taker and giver do not have to be same task. Some additional points of interest are that there is no effect of releasing the semaphore again; It can be used as a signal (thus can be created empty); and supports the idea of a broadcast release (wake up all waiting tasks rather than just the first). The *Counting Semaphore*, as expected, is the same as the *Binary Semaphore* with ability to define an initial count.

The Mutual-Exclusion Semaphore is the VxWorks *mutex*. Only the owning task may successfully call semGive. The VxWorks mutex also has the ability to support both priority inheritance (basic priority inheritance protocol) and deletion safety.

POSIX

POSIX is an acronym for Portable Operating System Interface (the X has no meaning). The current POSIX standard is formally defined by IEEE Std 1003.1, 2004 Edition. The mutex is part of the core POSIX Threads (pThreads) specification (historically referred to as IEEE Std 1003.1c-1995).
POSIX also supports both semaphores and priority-inheritance mutexes as part of what are called Feature Groups. Support for these Feature Groups is optional, but when an implementation claims that a feature is provided, all of its constituent parts must be provided
and must comply with this specification. There are two main Feature Groups of interest, the Realtime Group and Realtime Threads Groups.

The semaphore is not part of the core standard but is supported as part of the Realtime Feature Group. The Realtime Semaphore is an implementation of the *Counting semaphore*.

The default POSIX *mutex* is non-recursive , has no priority inheritance support or death detection.
However, the Pthreads standard allows for non-portable extensions (as long as they are tagged with "-np"). A high proportion of programmers using POSIX threads are programming for Linux. Linux supports four different mutex types through non-portable extensions:

- Fast mutex – non-recursive and will deadlock [default]
- Error checking mutex – non-recursive but will report error
- Recursive mutex – as the name implies
- Adaptive mutex – extra fast for mutli-processor systems

These are extreamly well covered by Chris Simmonds in his posting Mutex mutandis: understanding mutex types and attributes.

Finally the Realtime Threads Feature Group adds mutex support for both priority inheritance and priority ceiling protocols.

Win32 API

Microsoft Window's common API is referred to as Win32. This API supports three different primitives:

- Semaphore – The counting semaphore
- Critical Section – Mutex between threads in the same process; Recursive, no timeout, queuing order undefined
- Mutex – As per critical sections, but can be used by threads in different processes; Recursive, timeout, queuing order undefined

The XP/Win32 mutex API does not support priority inheritance in application code, however the WinCE/Win32 API does!

Win32 mutexes do have built-in death detection; if a thread terminates when holding a mutex, then that mutex is said to be abandoned. The mutex is released (with WAIT_ABANDONED error code) and a waiting thread will take ownership. Note that Critical sections do not have any form of death detection.

Critical Sections have no timeout ability, whereas mutexes do. However Critical Sections support a separate function call TryEnterCriticalSection. A major weakness of the Win32 API is that the queuing model is undefined (i.e. neither Priority nor FIFO). According to Microsoft this is done to improve performance.

So, what can we gather from this? First and foremost the term mutex is less well defined than the semaphore. Secondly,the actual implementations from RTOS to RTOS vary massively. I urge you to go back and look at your faviourite RTOS and work out what support, if any, you have for the mutex. I'd love to hear from people regarding mutual exclusion support (both semaphores and mutexes) for their RTOS of choice. If you'd like to contact me do so at nsc(at)acm.org.

Finally, Part 3 will look at a couple of problems the mutex doesn't solve, and how these can be overcome. As part of that it will review the Basic Priority Inheritance Protcol and the Prority Ceiling Protocol.

At a later date I will also address the use of, and problems associted with, the semaphore being used for task synchronisation.

ENDNOTES

1. Please I do not want to get into the "*that's not a real-time OS*" debate here – let's save that for another day!
2. A number of people pointed out that Michael Barr (former editor of Embedded Systems Programming, now president of Netrino) has a good article about the differences between mutexes & semaphores at the following location: http://www.netrino.com/node/202. I urge you to read his posting as well.
3. Apologies about not having the atom feed sorted – this should all be working now