

Sticky Bits – Powered by Feabhas

*A blog looking at developing software
for real-time and embedded systems*

Mutex vs. Semaphores – Part 3 (final part): Mutual Exclusion Problems

Posted on October 5, 2009 by Niall Cooling

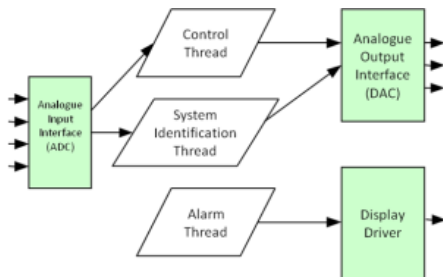
As hopefully you can see from the previous posting, the mutex is a significantly safer mechanism to use for implementing mutual exclusion around shared resources. Nevertheless, there are still a couple of problems that use of the mutex (in preference to the semaphore) will not solve. These are:

- Circular deadlock
- Non-cooperation

Circular Deadlock

Circular deadlock, often referred to as the “deadly embrace” problem is a condition where two or more tasks develop a circular dependency of mutual exclusion. Simply put, one task is blocked waiting on a mutex owned by another task. That other task is also blocked waiting on a mutex held by the first task.

So how can this happen? Take as an example a small control system. The system is made up of three tasks, a low priority Control task, a medium priority System Identification (SI) task and a high priority Alarm task. There is an analogue input shared by the Control and the SI tasks, which is protected by a mutex. There is also an analogue output protected by a different mutex.



The Control task waits for mutexes ADC and DAC:

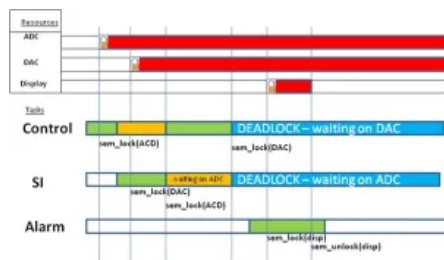
```
mutex_lock (ADC);
mutex_lock (DAC);
/* critical section */
mutex_unlock (ADC);
mutex_unlock (DAC);
```

The SI Task waits for mutexes DAC and ADC:

```
mutex_lock (DAC);
mutex_lock (ADC);
/* critical section */
mutex_unlock (DAC);
mutex_unlock (ADC);
```

Unfortunately, under certain timing conditions, this can lead to deadlock. In this example the Control task has locked the ADC, but before locking the DAC has been pre-empted by the higher priority SI task. The SI task then locks the DAC and tries to lock the ADC. The SI task is now blocked as the ADC is already owned by the Control

task. The Control task now runs and tries to lock the DAC. It is blocked as the DAC is held by the SI task. Neither task can continue until the mutex is unlocked and neither mutex can be unlocked until either task runs – classic deadlock.



For circular deadlock to occur the following conditions must all be true:

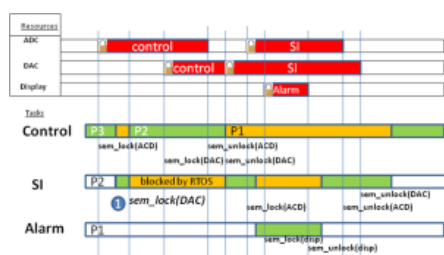
- A thread has exclusive use of resources (Mutual exclusion)
- A thread can hold on to a resource(s) whilst waiting for another resource (Hold and wait)
- A circular dependency of thread and resources is set up (Circular waiting)
- A thread never releases a resource until it is completely finished with it (No resource preemption)

These conditions can be addressed in a number of ways. For example, a design policy may stipulate that if a task needs to lock more than one mutex it must either lock all or none.

Priority Ceiling Protocol

With the *Priority Ceiling Protocol (PCP)* method each mutex has a defined priority ceiling, set to that of the highest priority task which uses the mutex. Any task using a mutex executes at its own priority – until a second task attempts to acquire the mutex. At this point it has its priority raised to the ceiling value, preventing suspension and thus eliminating the “hold and wait” condition.

In the deadlock example shown before, the significant point is when the SI task tries to lock the DAC. Before that succeeded and lead to cyclic deadlock. However with a PCP mutex, both the ADC and DAC mutex will have a ceiling priority equal to the SI’s task priority. When the SI task tries to lock the DAC, then the run-time system will detect that the SI’s task priority is not higher than the priority of the locked mutex ADC. The run-time system suspends the SI task without locking the DAC mutex. The control task now inherits the priority of the SI task and resumes execution.



Non-cooperation

The last, but most important aspect of mutual exclusion covered in these ramblings relies on one founding principle: *we have to rely on all tasks to access critical regions using mutual exclusion primitives*. Unfortunately this is dependent on the design of the software and cannot be detected by the run-time system. This final problem was addressed by Tony Hoare, called the **Monitor**.

The Monitor

The monitor is a mechanism not typically supplied by the RTOS, but something the programmer tends to build (a notable exception is Ada95’s protected object mechanism). A monitor simply encapsulates the shared resource

and the locking mechanism into a single construct (e.g. a C++ Object that encapsulates the mutex mechanism). Access to the shared resource, then, is through a controlled interface which cannot be bypassed (i.e. the application never explicitly calls the mutex, but calls upon access functions).

Finishing Off...


This goal of these initial postings is to demonstrate that common terms used in the real-time programming community are open to ambiguity and interpretation. Hopefully you should now be clear about the core differences between the Binary Semaphore, General (counting) Semaphore and the Mutex.

The underlying difference between the Semaphores and the Mutex is the **Principle of Ownership**. Given the principle of ownership a particular implementation of a mutex may support *Recursion*, *Priority inheritance* and *Death Detection*.

ENDNOTE

An aspect of the mutex I haven't covered here is that many operating systems support the concept of a *condition variable*. A condition variable allows a task to wait on a synchronization primitive within a critical region. The whole aspect Synchronization Patterns (e.g. semaphore as a signal) within the context of RTOSs will be the subject of my next posting.

AboutLatest Posts




Niall Cooling

Director at Feabhas Limited

Co-Founder and Director of Feabhas since 1995.
Niall has been designing and programming embedded systems for over 30 years. He has worked in different sectors, including aerospace, telecomms, government and banking.
His current interest lie in IoT Security and Agile for Embedded Systems.

Like (8)

Dislike (0)




Niall Cooling

Website | + posts

Co-Founder and Director of Feabhas since 1995.
Niall has been designing and programming embedded systems for over 30 years. He has worked in different sectors, including aerospace, telecomms, government and banking.
His current interest lie in IoT Security and Agile for Embedded Systems.

This entry was posted in [RTOS](#) and tagged [Deadlock](#), [Deadly Embrace](#), [Monitor](#), [Mutex](#), [Priority Ceiling Protocol](#), [RTOS](#). Bookmark the [permalink](#).

14 Responses to *Mutex vs. Semaphores – Part 3 (final part): Mutual Exclusion Problems*



venu says:

October 26, 2009 at 8:38 am

Neil,

You remind me of grad school professor, but you are better. He always taught us the theory without telling us practically where it is used.