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

Mutex vs. Semaphores – Part 1: Semaphores

Posted on September 7, 2009 by Niall Cooling

It never ceases to amaze me how often I see postings in newsgroups, etc. asking the difference between a semaphore and a mutex. Probably what baffles me more is that over 90% of the time the responses given are either incorrect or missing the key differences. The most often quoted response is that of the "The Toilet Example (c) Copyright 2005, Niclas Winquist" . This summarises the differences as:

- **A mutex is really a semaphore with value 1**

No, no and no again. Unfortunately this kind of talk leads to all sorts of confusion and misunderstanding (not to mention companies like Wind River Systems redefining a mutex as a "Mutual-Exclusion Semaphore" – now where is that wall to bang my head against?).

Firstly we need to clarify some terms and this is best done by revisiting the roots of the semaphore. Back in 1965, Edsger Dijkstra, a Dutch computer scientist, introduced the concept of a binary semaphore into modern programming to address possible race conditions in concurrent programs. His very simple idea was to use a pair of function calls to the operating system to indicate entering and leaving a critical region. This was achieved through the acquisition and release of an operating system resource called a semaphore. In his original work, Dijkstra used the notation of P & V, from the Dutch words *Prolagen* (P), a neologism coming from *To try and lower*, and *Verhogen* (V) *To raise, To increase*.

```
/* task 1 */
   ...
   P(S);
   /* critical region */
   V(S);
   ...
```

```
/* task 2 */
   ...
   P(S);
   /* critical region */
   V(S);
   ...
```

With the model the first task arriving at the **P(S)** [where S is the semaphore] call gains access to the critical region. If a context switch happens while that task is in the critical region, and another task also calls on **P(S)**, then that second task (and any subsequent tasks) will be blocked from entering the critical region by being put in a waiting state by the operating system. At a later point the first task is rescheduled and calls **V(S)** to indicate it has left the critical region. The second task will now be allowed access to the critical region.

A variant of Dijkstra's semaphore was put forward by another Dutchman, Dr. Carel S. Scholten. In his proposal the semaphore can have an initial value (or count) greater than one. This enables building programs where more than one resource is being managed in a given critical region. For example, a counting semaphore could be used to manage the parking spaces in a robotic parking system. The initial count would be set to the initial free parking places. Each time a place is used the count is decremented. If the count reaches zero then the next task trying to acquire the semaphore would be blocked (i.e. it must wait until a parking space is available). Upon releasing the semaphore (A car leaving the parking system) the count is incremented by one.

Scholten's semaphore is referred to as the **General or Counting Semaphore**, Dijkstra's being known as the **Binary Semaphore**.

Pretty much all modern Real-Time Operating Systems (RTOS) support the semaphore. For the majority, the actual implementation is based around the **counting** semaphore concept. Programmers using these RTOSs may use an initial count of 1 (one) to approximate to the binary semaphore. One of the most notable exceptions is probably the leading commercial RTOS VxWorks from Wind River Systems. This has two separate APIs for semaphore creation, one for the Binary semaphore (*semBCreate*) and another for the Counting semaphore (*semCCreate*).

Hopefully we now have a clear understanding of the difference between the binary semaphore and the counting semaphore. Before moving onto the mutex we need to understand the inherent dangers associated with using the semaphore. These include:

- Accidental release
- Recursive deadlock
- Task-Death deadlock
- Priority inversion
- Semaphore as a signal

All these problems occur at run-time and can be very difficult to reproduce; making technical support very difficult.

Accidental release

This problem arises mainly due to a bug fix, product enhancement or cut-and-paste mistake. In this case, through a simple programming mistake, a semaphore isn't correctly acquired but is then released.

```
/* Apps thread code */
   ...
   P(S);
   if (count > 0) --count;
   /* read data from buffer */
   V(S);
   ...
```

```
/* Comms thread code */
   ...
   /* OOPS forgot
      P(S);
   */
   ++count;
   /* write data to buffer */
   V(S);
   ...
```

When the counting semaphore is being used as a binary semaphore (initial count of 1 – the most common case) this then allows two tasks into the critical region. Each time the buggy code is executed the count is increment and yet another task can enter. This is an inherent weakness of using the counting semaphore as a binary semaphore.

Deadlock

Deadlock occurs when tasks are blocked waiting on some condition that can never become true, e.g. waiting to acquire a semaphore that never becomes free. There are three possible deadlock situations associated with the semaphore:

- Recursive Deadlock
- Deadlock through Death
- Cyclic Deadlock (Deadly Embrace)

Here we shall address the first two, but shall return to the cyclic deadlock in a later posting.

Recursive Deadlock

Recursive deadlock can occur if a task tries to lock a semaphore it has already locked. This can typically occur in libraries or recursive functions; for example, the simple locking of malloc being called twice within the framework of a library. An example of this appeared in the MySQL database bug reporting system: *Bug #24745 InnoDB semaphore wait timeout/crash – deadlock waiting for itself*

Deadlock through Task Death

What if a task that is holding a semaphore dies or is terminated? If you can't detect this condition then all tasks waiting (or may wait in the future) will never acquire the semaphore and deadlock. To partially address this, it is common for the function call that acquires the semaphore to specify an optional timeout value.

Priority Inversion

The majority of RTOSs use a priority-driven pre-emptive scheduling scheme. In this scheme each task has its own assigned priority. The pre-emptive scheme ensures that a higher priority task will force a lower priority task to release the processor so it can run. This is a core concept to building real-time systems using an RTOS. Priority inversion is the case where a high priority task becomes blocked for an indefinite period by a low priority task. As an example:

- An embedded system contains an "information bus"
- Sequential access to the bus is protected with a semaphore.
- A bus management task runs frequently with a **high priority** to move certain kinds of data in and out of the information bus.
- A meteorological data gathering task runs as an infrequent, **low priority** task, using the information bus to publish its data. When publishing its data, it acquires the semaphore, writes to the bus, and release the semaphore.
- The system also contains a communications task which runs with **medium priority**.
- Very infrequently it is possible for an interrupt to occur that causes the (medium priority) communications task to be sch
  eduled while the (high priority) information bus task is blocked waiting for the (low priority) meteorological data task.
- In this case, the long-running communications task, having higher priority than the meteorological task, prevents it from running, consequently preventing the blocked information bus task from running.
- After some time has passed, a **watchdog timer** goes off, notices that the data bus task has not been executed for some time, concludes that something has gone drastically wrong, and initiate a total system reset.

This well reported event actual sequence of events happened on NASA JPL's Mars Pathfinder spacecraft.

Semaphore as a Signal

Unfortunately, the term synchronization is often misused in the context of mutual exclusion. Synchronization is, by definition "To occur at the same time; be simultaneous". Synchronization between tasks is where, typically, one task waits to be notified by another task before it can continue execution (*unilateral rendezvous*). A variant of this is either task may wait, called the bidirectional rendezvous. This is quite different to mutual exclusion, which is a protection mechanism. However, this misuse has arisen as the counting semaphore can be used for unidirectional synchronization. For this to work, the semaphore is created with a count of 0 (zero).

```
...
/* WAIT */
P(S);
...
```

```
...
/* SIGNAL */
V(S);
...
```

Note that the P and V calls are not used as a pair in the same task. In the example, assuming Task1 calls the **P(S)** it will block. When Task 2 later calls the **V(S)** then the unilateral synchronization takes place and both task are ready to run (with the higher priority task actually running). Unfortunately "misusing" the semaphore as synchronization primitive can be problematic in that it makes debugging harder and increase the potential to miss "accidental release" type problems, as an **V(S)** on its own (i.e. not paired with a **P(S)**) is now considered legal code.