Sticky Bits

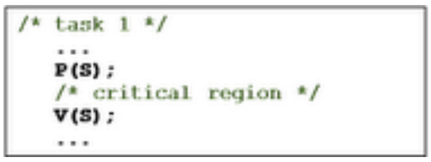
Mutex vs. Semaphores – Part 1: Semaphores

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:

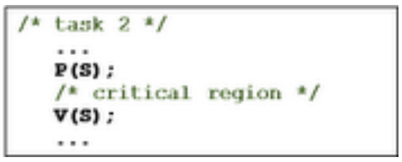
- **A mutex is really a semaphore with value 1**

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 this 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*).

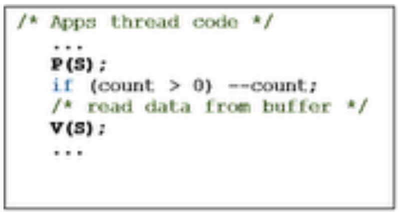
Hopefully we now have a clear understanding of the difference between the binary semaphore and the counting semaphore. Before moving onto the mutex we need to understand the inherent dangers associated with using the semaphore. These include:

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

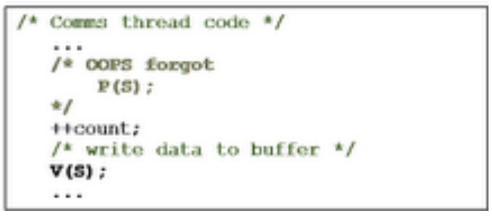
All these problems occur at run-time and can be very difficult to reproduce; making technical support very difficult.

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

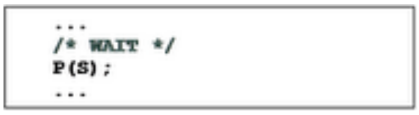
The majority of RTOSs use a priority-driven pre-emptive scheduling scheme. In this scheme each task has its own assigned priority. The pre-emptive scheme ensures that a higher priority task will force a lower priority task to release the processor so it can run. This is a core concept to building real-time 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.

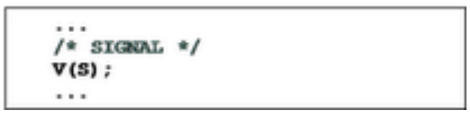
This well reported event actual sequence of events happened on NASA JPL's Mars Pathfinder spacecraft.

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.

In the next posting I shall look at how the mutex address most of the weaknesses of the semaphore.

About Latest Posts

Admin

Share this:

LinkedIn Twitter Reddit Google Email

Like this:

★ Like

Be the first to like this.

Posted on September 7th, 2009
» Feed to this thread
» Trackback

24 Comments a "Mutex vs. Semaphores – Part 1: Semaphores"

1. *Anonymous* says:
   September 8th, 2009 at 3:21 pm

   Can you please enable the rss or atom feed for your blog? Thanks!

2. *Michael Barr* says:
   September 8th, 2009 at 6:38 pm

   Niall,

   Welcome to the blogosphere! As I would expect from you, you've done a nice job above explaining some rather complex (and often misunderstood) issues.

However, I have a problem with one of the examples. The robotic parking garage implementation suffers from implementation by a counting semaphore. Only a mutex is truly needed. Here's why…

Each of the parking spaces is an individually identifiable object. In a computer, it is analogous to a fixed-sized memory buffer. If the buffer contains data, some part of the application code has a pointer to it. If the buffer is empty, it must be tracked in a "free list" data structure. One suitable data structure is a linked list maintained within the empty buffers plus a head pointer. (But the important general point is there is always at least one piece of metadata.)

The free list data structure must be protected via a mutual exclusion primitive (preferably a mutex). Gaining access to that data structure will tell the caller if there are any free parking spots.

The suggested use of a counting semaphore neither (a) gets you deep enough into the implementation to pick a specific parking spot nor (b) is an alternative to the mutex, which must always be used. Thus the counting semaphore is a waste of space in the solution to that problem.

Generalizing, I recommend avoiding use of the term "counting semaphore" altogether. Only mutexes and semaphores are of practical use, for data protection and signaling respectively.

I've blogged a bunch on mutexes and semaphores at http://www.embeddedgurus.net/barr-code and, on the points you're discussing, highly recommend the article Mutexes and Semaphores Demystified at http://www.netrino.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore

Cheers,
Michael

3. *Rennie Allen* says:
September 8th, 2009 at 9:39 pm

While I think I agree on where you are going with this, the argument is difficult to grasp because you seem to be comparing an implementation with an abstract functional concept.

From the functional concept perspective, a conceptual mutex (rather than a specific implementation such as pthreads) could be considered to be a semaphore with a count of one. Because the generic concept of the function of a mutex is ill-defined, I am convinced that I could write a set of cover functions for a mutex, implemented in terms of a semaphore with a count of one; and that you'd be hard pressed to show me why that doesn't provide all the attributes of a conceptual mutex function (i.e. conceptually, they are the same).

Certainly, if we consider this question within the constraints of the SysV behavioral model of the semaphore and the pthreads behavioral model of a mutex, then I think we can agree that there are many behavioral details that make a SysV semaphore with a count of 1, and a pthread mutex significantly different.

I think it is important to make clear though, that you are talking about SysV semaphores and pthread mutexes (you are, aren't you?) rather than the conceptual model of semaphores and mutexes.

4. *Dan* says:
September 8th, 2009 at 11:01 pm

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

Also another article discusses the "Perils of Preemption" (including issues you covered such as deadlock & priority inversion) at this link:

http://www.netrino.com/Embedded-Systems/How-To/Preemption-Perils

Keep up the good work. More people need to learn about such topics.

5. *Anonymous* says:
September 8th, 2009 at 11:05 pm

This is a great start for a blog. Good luck with it.

6. *Bill Dittmann* says:
September 9th, 2009 at 3:16 pm

Niall,
Good article. In the RTXC Quadros RTOS mutex implementation, nesting locks on the same mutex are allowed, supported, and safe.

We do 2 things to make it safe. First, we remember who locked (owns) the mutex and keep a counter as it is locked and locked. The counter scheme properly tracks the scope of the mutex lock.

We also make sure that only the owner can release (unlock) the mutex.

In my experiences, during development phases, it is not uncommon for a task to inadvertently release when it is doesn't even own the lock. RTXC Quadros detects this lock underflow or release by non-owner condition, e.g., a program design flaw, during runtime.

7. *Anonymous* says:
September 10th, 2009 at 8:36 am

Funny how people still seem to want to assign code blocks that need to be executed serially, to different tasks. Why not assign the resource to one and only one task, and let client tasks send requests and receive replies? This is easily understood by laymen and easy to debug.

Alas not many so-called "modern" operating systems support this methodology very well.

8. *steve* says:

September 10th, 2009 at 3:32 pm

Am enjoying the blog so far. There is a shortage of decent blogs that deal specifically with embedded systems.

Can I echo previous comments, and request that ths RSS/Atom feeds be enabled? Thanks.

Steve.

9. *Anonymous* says:
September 28th, 2009 at 3:17 pm

Thank u very much man. This is very helpful for me.

Waiting for information on MUTEX.

10. *Anonymous* says:
October 15th, 2009 at 9:33 pm

Thanks. All the information in this discussion were useful for my assignments and exams. I should give you the credit. And would be more than happy if you can provide some information about how ISRs work with reference to device drivers in linux.

Thanks in advance,
Jayaraman Baskar,
Student,MS in computer engineering

11. *Solti* says:
February 1st, 2010 at 3:11 am

Hi Niall,

Can I say when we are dealing with mutual exclusion problems, a binary semaphore is the same to a mutex?

Suppose we use them correctly, and there is a resource which allows only one user at every time instant. There must be a get/return pair in 'user' if we use mutex, and there must also be a wait/signal pair if we use binary semaphore. So in this case, can I say they are totally the same?

Thank you very much, and I really appreciate your articles.

Solti

12. *Shreshtha* says:
May 20th, 2010 at 7:18 am

Hi All,

With such a great quality of information and discussion from industry's eminent people, this page should come in top every time mutex semaphore is searched. Wonderful discussion.

From Linux perspective here is my understanding about difference of Mutex and Sem –
(please correct me if I am wrong anywhere)

Note – I find its very important that we are clear about the part of Linux we are discussing this topic – Kernel Space or User Space.

i) Scope – The scope of mutex is within a process address space which has created it and is used for synchronization of common resource access. Whereas semaphore can be used across processes space and hence it can be used for interprocess synchronization/signaling. Hence mutex must be released by same thread which is taking it.

ii) Mutex is lightweight and faster than semaphore

(only valid for userspace)
iii) Mutex can be acquired by same thread successfully multiple times with condition that it should release it same number of times. Other thread trying to acquire will block. Whereas in case of semaphore if same process tries to acquire it again it blocks as it can be acquired only once.

I found this tread very informative in this context – http://alinux.tv/Kernel-2.6.29/mutex-design.txt

Cheers,
@Shreshtha19

13. *Shreshtha* says:
July 12th, 2010 at 8:06 am

above link is not working. here is working one –
http://www.kernel.org/doc/Documentation/mutex-design.txt

14. *Mutex Vs Semaphore « Roshan Singh* says:
November 17th, 2010 at 6:38 am

[…] these links to understand the difference: 1. http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%e2%80%93-part-1-semaphores/ 2. http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%e2%80%93-part-2-the-mutex/ 3. […]

15. *Priya* says:
August 14th, 2012 at 1:11 am

Very useful article. . .Thanks a lot:)

16. *Multithreading, mutex, semaphore | Agnihotri* says:
July 11th, 2013 at 12:23 am

[…] http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%e2%80%93-part-1-semaphores/ 2. http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%e2%80%93-part-2-the-mutex/ 3. […]

17. *nice one* says:
July 11th, 2013 at 12:24 am

 Nice explaination.

18. *Sara* says:
July 11th, 2013 at 2:51 pm

 Great article!!

 I'm a little confused w/ the priority inversion. if tL (low priority task) was running in the critical section, how can tM preempted tL? This mutex locking is to disallow preemption during critical region.

19. *admin* says:
July 12th, 2013 at 8:59 am

 Preemption is not disabled in critical sections, there are times in which you really want to allow preemption within critical sections, especially between unrelated tasks. Many RTOSs support disabling preemption, but this is through an explicit API. If critical sections disabled preemption you would end up with a very unresponsive system.

20. *mutex vs semaphore | technoless* says:
August 21st, 2013 at 6:14 pm

 […] – http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%e2%80%93-part-1-semaphores/ – http://blog.feabhas.com/2009/09/mutex-vs-semaphores-%e2%80%93-part-2-the-mutex/ – […]

21. *Akil* says:
December 3rd, 2013 at 4:49 am

 Nice post

22. *Sticky Bits » Blog Archive » Templates and polymorphism* says:
July 10th, 2014 at 9:40 am

 […] conditions within the queue (both threads trying to modify the queue 'simultaneously' – see here for a more detailed description) we should protect it with a mutual exclusion mechanism. For the […]

23. *bhupesh* says:
November 21st, 2014 at 2:56 pm

 http://www.writeulearn.com/binary-semaphore-mutex-semaphore/

24. *shashank* says:
[January 26th, 2015 at 4:55 pm](#)

Really really useful 🙂 thanks!

Leave a Reply

Name

E-mail (will not be published)

Website

Submit Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.

Powered by [WordPress](#) - [StupidGenius](#) theme by [Cristiano M. Gaston](#)