Glossary

Find definitions for technical terms in our Embedded Systems Glossary.

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y
Z	Symbols			

Test Your Skills

How good are your embedded programming skills? Test yourself in the Embedded C Quiz or the Embedded C++ Quiz.

Newsletter Signup

Want to receive free how-to articles and industry news as well as announcements of free webinars and other training courses by e-mail? Signup now.

Mutexes and Semaphores Demystified

Tue, 2008-08-19 22:23 - webmaster

by Michael Barr

The question "What is the difference between a mutex and a semaphore?" is short and easily phrased. Answering it is more difficult. In this first installment of a series of articles on the proper use of a real-time operating system (RTOS), we examine the important differences between a mutex and a semaphore.

After conversations with countless embedded software developers over many years, I have concluded that even very experienced RTOS users have trouble distinguishing the proper uses of mutexes and semaphores. This is unfortunate and dangerous, as misuse of either RTOS primitive can easily lead to unintended errors in embedded systems that could turn them into life-threatening products.

Barr Group has a free 1-hour webinar on this and related topics scheduled for May 13, 2015 at 1pm EDT. Register now

In this article, I aim to distinguish these two important RTOS primitives once and for all, by debunking a popular myth about their similarity.

Myth: Mutexes and Semaphores are Interchangeable

Reality: While mutexes and semaphores have some similarities in their implementation, they should always be used differently.

The most common (but nonetheless incorrect) answer to the question posed at the top is that mutexes and semaphores are very similar, with the only significant difference being that semaphores can count higher than one. Nearly all engineers seem to properly understand that a mutex is a binary flag used to protect a shared resource by ensuring mutual exclusion inside critical sections of code. But when asked to expand on how to use a "counting semaphore," most engineers—varying only in their degree of confidence—express some flavor of the textbook opinion that these are used to protect several equivalent resources. [1]

It is easiest to explain why the "multiple resource" scenario is flawed by way of an analogy. If you think of a mutex as a key owned by the operating system, it is easy to see that an individual mutex is analogous to the bathroom key owned by an urban coffee shop. At the coffee shop, there is one bathroom and one bathroom key. If you ask to use the bathroom when the key is not available, you are asked to wait in a queue for the key. By a very similar protocol, a mutex helps multiple tasks serialize their accesses to shared global resources and gives waiting tasks a place to wait for their turn.

This simple resource protection protocol does not scale to the case of two equivalent bathrooms. If a semaphore were a generalization of a mutex able to protect two or more identical shared resources, then in our analogy, it would be a basket containing two identical keys (i.e., each of the keys would work in either bathroom door).

A semaphore cannot solve a multiple identical resource problem on its own. The visitor only knows that he has a key, not yet which bathroom is free. If you try to use a semaphore like this, you'll find you always need other state information—itself typically a shared resource protected via a separate mutex. [2] It turns out the best way to design a two-bathroom coffee shop is to offer distinct keys to distinct bathrooms (e.g., men's vs. women's), which is analogous to using two distinct mutexes.

The correct use of a semaphore is for signaling from one task to another. A mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects. By contrast, tasks that use semaphores either signal or wait—not both. For example, Task 1 may contain code to post (i.e., signal or increment) a particular semaphore when the "power" button is pressed and Task 2, which wakes the display, pends on that same semaphore. In this scenario, one task is the producer of the event signal; the other the consumer.

To summarize with an example, here's how to use a mutex:

```
/* Task 1 */
   mutexWait(mutex_mens_room);
      // Safely use shared resource
   mutexRelease(mutex_mens_room);

/* Task 2 */
   mutexWait(mutex_mens_room);
      // Safely use shared resource
   mutexRelease(mutex_mens_room);
```

By contrast, you should always use a semaphore like this:

```
/* Task 1 - Producer */
    semPost(sem_power_btn);   // Send the signal

/* Task 2 - Consumer */
    semPend(sem_power_btn);  // Wait for signal
```

Importantly, semaphores can also be used to signal from an interrupt service routine (ISR) to a task. Signaling a semaphore is a non-blocking RTOS behavior and thus ISR safe. Because this technique eliminates the error-prone need to disable interrupts at the task level, signaling from within an ISR is an excellent way to make embedded software more reliable by design.

Priority Inversion

Another important distinction between a mutex and a semaphore is that the proper use of a mutex to protect a shared resource can have a dangerous unintended side effect. Any two RTOS tasks that operate at different priorities and coordinate via a mutex, create the opportunity for **priority inversion**. The risk is that a third task that does not need that mutex —but operates at a priority between the other tasks—may from time to time interfere with the proper execution of the high priority task.

An unbounded priority inversion can spell disaster in a real-time system, as it violates one of the critical assumptions underlying the **Rate Monotonic Algorithm** (RMA). Since RMA is the optimal method of assigning relative priorities to real-time tasks and the only way to ensure multiple tasks with deadlines will always meet them, it is a very bad thing to risk breaking one of its assumptions. Additionally, a priority inversion in the field is a very difficult type of problem to debug, as it is not easily reproducible.

Fortunately, the risk of priority inversion can be eliminated by changing the operating system's internal implementation of mutexes. Of course, this adds to the overhead cost of acquiring and releasing mutexes. Fortunately, it is not necessary to change the implementation of semaphores, which do not cause priority inversion when used for signaling. This is a second important reason for having distinct APIs for these two very different RTOS primitives.

The History of Semaphores and Mutexes

The cause of the widespread modern confusion between mutexes and semaphores is historical, as it dates all the way back to the 1974 invention of the Semaphore (capital "S", in this article) by **Djikstra**. Prior to that date, none of the interrupt-safe task synchronization and signaling mechanisms known to computer scientists was efficiently scalable for use by more than two tasks. Dijkstra's revolutionary, safe-and-scalable Semaphore was applied in both critical section protection and signaling. And thus the confusion began.

However, it later became obvious to operating system developers, after the appearance of the priority-based preemptive RTOS (e.g., VRTX, ca. 1980), publication of academic papers establishing RMA and the problems caused by priority inversion, and a paper on priority inheritance protocols in 1990, [3] it became apparent that mutexes must be more than just semaphores with a binary counter.

Unfortunately, many sources of information, including textbooks, user manuals, and wikis, perpetuate the historical confusion and make matters worse by introducing the additional names "binary semaphore" (for mutex) and "counting semaphore."

I hope this article helps you understand, use, and explain mutexes and semaphores as distinct tools.

Endnotes

1. Truth be told, many "textbooks" on operating systems fail to define the term mutex (or real-time). But consider a typical snippet from the QSemaphore page in Trolltech's Qt Embedded 4.4 Reference Documentation: "A semaphore is a generalization of a mutex. While a mutex can only be locked once, it's possible to acquire a semaphore multiple times. Semaphores are typically used to protect a certain number of identical resources." The Qt documentation

then goes on to elaborate (at length and with code snippets) a solution to a theoretical problem that lacks critical details (e.g., the need for a second type of RTOS primitive) for proper implementation. [**back**]

2. Another option is to use an RTOS-provided message queue to eliminate the semaphore, the unprotected shared state information, and the mutex. [**back**]

3. Sha L., R. Rajkumar, and J.P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, September 1990, p. 1175. [**back**]

Tags:
Real-Time Operating Systems

» **webmaster's blog** | **Log in** or **register** to post comments

Comments

Thank You

Permalink Submitted by rasheedtel on Tue, 2010-01-19 04:28.

This is helpful article. Thank you very much.

» **Log in** or **register** to post comments

Thanks

Permalink Submitted by mmsreedhar on Wed, 2010-01-27 01:42.

It was very nice info. All your topics are very clear and explainatory. Thanks for the information in a nice format.

» **Log in** or **register** to post comments

good article...........but i

Permalink Submitted by venkatesh yadav on Thu, 2010-02-18 23:35.

good article...........but i have a basic doubt is that binary semapore and mutex are same? if not what is the difference.

» **Log in** or **register** to post comments

Binary Semaphore vs. Mutex

Permalink Submitted by webmaster on Thu, 2010-10-21 14:54.

Generally speaking, a binary semaphore is the same as a mutex except that only the mutex has built into its API a priority inversion workaround, such as priority inheritance protocol. Thus binary semaphores should not be used in real-time systems that are required to meet task deadlines with certainty.

» **Log in** or **register** to post comments

Very helpful.

Permalink Submitted by yuwen on Tue, 2010-03-02 21:23.

Very helpful.

» **Log in** or **register** to post comments

Mutex Vs Semaphore

Permalink Submitted by rakshith.amarnath on Tue, 2010-03-16 13:38.

Hello Mr. Barr,

Yes this is a very insightful explanation.

When I was going through one of the books which discussed RTOS'es, I found the following explanation.

Binary semaphore can be released by any task, even a task that did not originally acquire the semaphore.

But Mutexes have the task ownership property which allows only the task acquired to release it.

By the statement above, there is a fundamental difference in the way a Mutex and semaphore behaves.

The book mentions Mutex as semaphores however.

Only after perusal of this discussion I am able to rightly make a distinction amongst the two.

In broader sense use semaphore for signaling and use Mutex for locking a shared resource is what you have tried to cover. Please add more details if my understanding is not apt.

Cheers,
Rakshith

» **Log in** or **register** to post comments

Date of invention of semaphore

Permalink Submitted by pmcjones on Sun, 2010-10-24 13:11.

You say "the 1974 invention of the Semaphore", but it was much earlier than that. For example, Dijkstra's famous "Cooperating Sequential Processes (EWD123) was written in 1965, presented at a 1966 NATO Summer School, and published in a 1968 book *Programming Languages*, edited by F. Genuys. See **http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF** .

» **Log in** or **register** to post comments

Critical Sections VS Mutex & Semaphore

Permalink Submitted by azfar_khan83 on Sat, 2011-08-13 14:55.

this is a very helpful article. However I have some queries.

1. You have write this in your article that, "A semaphore cannot solve a multiple identical resource problem on its own. The visitor only knows that he has a key, not yet which bathroom is free. If you try to use a semaphore like this, you'll find you always need other state information--itself typically a shared resource protected via a separate mutex" that

means we have to use mutex with semaphores. mutexes ensures sequential access of the resources while semaphores ensures availability of the resource. Am I right? Please elaborate if I am wrong. A code snippet is preferably helpful. Secondly, does any recent framework (.NET for example) introduces a comprehensive version of Semaphore that underlying handles both issues.

2. you have also said that, "The visitor only knows that he has a key". Does the process acquire the lock of the resource but wait for semaphore to gets its turn. Am I right? If it is true, do we have to handle it ourselves?

3. What are the key differences in Critical section & Monitors as compared to semaphores and mutexes. what are the appropriate use of each.

4. Is the use or the behavior of Mutex or semaphore differs in Multi-process access and in multi-threaded access

5. In semaphore implementation, the initialization of integer value
varies with the number of shared resources.

Regards
--
Azfar Khan

» **Log in** or **register** to post comments

>Since RMA is the optimal

Permalink Submitted by halst on Tue, 2012-01-24 15:39.

>Since RMA is the optimal method of assigning relative priorities
>to real-time tasks and the only way to ensure multiple tasks with
>deadlines will always meet them

Rate-monotonic scheduling is not the *only* scheduling method that ensures that deadlines are met. Another example is deadline-monotonic scheduling (DMA), which is often better.

http://en.wikipedia.org/wiki/Deadline-monotonic_scheduling

» **Log in** or **register** to post comments

Usage case of counting semaphore

Permalink Submitted by dimonomid on Mon, 2014-10-06 09:29.

Hello, I'm a bit late to the party, but anyway, great thanks for the article.

But, you haven't mentioned the usage case of counting semaphore: when should I use it? I understand clearly what mutex is for, as well as what so-called binary semaphore is for. But, when should I use semaphore with max count > 1? I elaborated more on this on stackoverflow: **http://stackoverflow.com/questions/26217282/the-usage-case-of-counting-s...**

I would be very glad if you explain the use case of counting semaphore.

Thanks!

» **Log in** or **register** to post comments