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 (link is external). 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 (link is external) (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 (link is external). 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]