Diving Deep With SEMAPHORE And MUTEX

Ankit Singh
Nov 6, 2015 · 6 min read

Lately, I've been reading about Semaphore and Mutex on the internet but surprisingly I didn't find any thorough article about it. Though, there are some good resources but it's all messed up. What happens in the end is that you get stuck with a lots of conflicting concepts in your head and then you go over all those resources again to extract some meaning out of them. So, I decided to write my introductory version on this much debated concept. Surely, the article will have references to other articles, books to help you understand more as I'm no operating systems genius. I suggest you to go through each of them at least once after finishing this article. Let's jump right into the subject.

Note — This post is not a *Semaphore versus Mutex* kind. I'll try to explain both of them separately.

Semaphore

Originally, the idea came into existence when the famous dutch scientist Sir E. W. Dijkstra in 1965 proposed a new integer variable to count the number of resources that are engaged with processes. Although, some of the details have been changed over the years but the basic idea is same. A semaphore is like an integer, with three key properties:

1. The semaphore can be initialized to any integer value. After that the only operations that are allowed on it are incrementing (increase by 1) and decrementing (decrease by 1) which are performed by processes (or threads). You are not allowed to read it's current value.

2. If the value of the semaphore becomes negative after a process decrements it, the process blocks itself (goes in the **waiting queue**). This prevents the CPU from **busy waiting** and allows it to do other useful work. The process unblocks itself (goes in **ready queue**) only after some other other process increments the semaphore.

3. When a process increments the semaphore, if there are other processes waiting, the operating system selects a process at random or according some scheduling algorithm and unblocks it.

Some of the intricacies with this definition are:

- In general, there is no way to know before a process decrements the semaphore whether it will block itself or not.
- After a process increments the semaphore and another thread gets unblocked, both processes continue running concurrently. There is no way to know which process, if either, will continue running immediately.
- When a process increments the semaphore, you don't necessarily know whether another process is waiting for that resource. So, the number of unblocked processes may be zero or more.

The Library Analogy

Here is the library analogy from wikipedia. Suppose a library has 10 identical study rooms (**resources**), to be used by one student at a time. Students (**processes**) must request a room from the font desk if they wish to use a room for study. If no rooms are free, students have to wait at the desk. When a student has finished using a room, the student must return to the desk and indicate that one room has become free. The clerk at the front desk doesn't need to keep track of which rooms are occupied and who is using them, nor does she know if any given room is actually being used, only the number of free room available (**semaphore**). When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. In this scenario, the initial value of the semaphore is 10. Once a student requests a room and granted access, the values becomes 9. After the next student comes, it drops to 8, then 7 and so on. If a student requests a room and the resulting value of the semaphore would be negative, the student is forced to wait until some student free a room. A semaphore apart from its initialization can only be accessed by two operations: wait() and signal(). A typical implementation of these operations are as follows:

Wait:

```
wait(S) {
    while (S <= 0)
        // busy wait
```

```
        S--;
    }
```

gistfile1.txt hosted with ❤ by GitHub view raw

Signal:

```
signal(S) {
    S++;
}

```

gistfile1.txt hosted with ❤ by GitHub view raw

The checking and changing of the semaphore integer (S) are all done as a single, indivisible **atomic** action. It is guaranteed that one a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

Semaphore Usage

Semaphores can be used to control access to a given number of resources. The semaphore is initialized to the available number of resources. Each process that wishes to uses a resource perform a **wait()** operation on the semaphore (thereby decrementing the count). When a process release a resource, it performs a **release()** operation (incrementing the count). When the count becomes 0, all the resources are being used. After that, process waits until the count becomes more than 0. Semaphores can also be used to solve **syncronization** problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore *synch*, initialized to 0. In process P1 , we insert the statements

In process P2 , we insert the statements

```
S1;
    signal(synch);
```

gistfile1.txt hosted with ❤ by GitHub view raw

```
wait(synch);
    S2;
```

gistfile1.txt hosted with ❤️ by GitHub view raw

Because *synch* is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

Why semaphores?

It's not a necessity that we need semaphores to solve syncronization problems, but there are some advantages in using them:

- Semaphores impose deliberate constraints that helps programmers to avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.
- Semaphores can be implemented efficiently on many systems, so solutions using semaphores are portable and usually efficient.

Mutex

The term *mutex* is short for **mutual exclusion**. A mutex is used when the semaphores ability to count is not needed. Mutexes are good only for managing mutual exclusion to some shared resource or some piece of code (**critical section**). A mutex guarantees that only one process access the shared resource at a given time. Lets take the example of *The Lord of the Flies* where a group of children (**processes**) use a conch as mutex. In order to speak, you have to hold the conch (**mutex**). As long as a child holds the conch, only he/she is allowed to speak. The mutex works on the mechanism of *locks*. A mutex is a variable that can be in one of two states: *locked* or *unlocked*. A process must acquire lock on the mutex before entering it's critical section; it releases the lock when it exits the critical section. The **acquire()** function acquires the lock and the **release()** function releases the lock. Let the mutex has a variable *available* whose value indicates that the lock is available to the process or not. A call to acquire() succeeds when the lock is available and the lock becomes unavailable. A process that attempts to acquire an unavailable lock get blocked and put in waiting queue until the lock is released. A typical implementation of the acquire() and release() functions are as follows:

acquire:

```
acquire() {
```

```
    while (!available)
        /* schedule another process */
    available = false
}
do {
    acquire lock
        critical section
    release lock
    remainder section
} while (true)
```

gistfile1.txt hosted with ❤️ by GitHub view raw

release:

```
release() {
    available = true;
}
```

gistfile1.txt hosted with ❤️ by GitHub view raw

The operations within both the functions must be performed atomically. The main advantage of mutex is that when a process fails to acquires the lock it gives up the CPU to another process. Consequently, there is no busy waiting. When the process runs the next time, it tests the lock again.

Are Semaphores and Mutexes interchangable?

The answer is **no**. While both have similarities in their implementations, they should always be used differently. Some books have gone to all the way to add specifics like **binary** semaphore and **counting** semaphore and that a mutex can sometimes be replaced by binary semaphore. I don't want to comment on that as I'm no operating system genius. A semaphore work on the mechanism **signals** while a mutex works on the mechanism of **locks**. The correct use of semaphores is for signaling from one process to another i.e., when a process finishes executing it task, it signals another process to indicate the availability of a resource. By contrast, processes that use semaphores either signal or wait — not both. For example, a process P1 increments the semaphore signaling some other process to use the resource that has just become available and then that other process decrements the semaphore. In this scenario, one is producer and the other is consumer. On the other hand, a mutex must be taken and releaseed in order by the same process that uses the shared resource. To summarize with an example, here's how the semaphore should be used:

Process P1:

```
signal(S) {
    S++;
}

```

gistfile1.txt hosted with ❤ by GitHub view raw

Process P2:

```
wait(S) {
    if(S <= 0)
        /* wait */
    S--;
}
```

gistfile1.txt hosted with ❤ by GitHub view raw

And here's how the mutex should be used:

Process P1:

```
wait(mutex)
    //safely use shared resource
    release(mutex)
```

gistfile1.txt hosted with ❤ by GitHub view raw

Process P2:

```
wait(mutex)
    //safely use shared resource
    release(mutex)
```

gistfile1.txt hosted with ❤ by GitHub view raw

That's it. That's all I could thought of and managed to write. Please write your valuable comments if you want to correct or add something.

http://www.barrgroup.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore

http://greenteapress.com/semaphores/