

**CSIS0230A Principles of Operating Systems(Class A)**  
**Notes for Tutorial 10**  
**More about semaphores**

During tutorial 9, we have our first encounter with semaphores. We understood how semaphores work, and how it can be applied to solve the reader-writer problem using an algorithm that others had written. This week we will design our own algorithm, and see the intricacy involved.

### **1. Reader and writer, revisited**

There are actually 3 variants of the reader-writer problem. The algorithm that we used in tutorial 9 is the easiest of the 3. It gives **priority to readers**: once a process starts reading, other processes can join reading without restrictions, even if a writer is waiting to write. The writer may be delayed arbitrarily.

The **total ordered** variant is a little bit more difficult. In this variant, read and write must follow exactly the same order as they arrive, with the only concurrency being adjacent requests to read may be done concurrently. This can be implemented by using one more semaphore to make sure one request will not be considered until the request before it starts to be serviced.

The **writer priority** scheme requires that processes must not start reading if a write is pending. This is the most difficult scheme to implement. We need to keep counting not just the number of readers currently accessing the shared object, but also the number of writers that are currently active (i.e., waiting for write or writing), as well as the number of readers that are currently waiting for all those writers to complete.

We will implement the writer priority scheme during the tutorial. Before the tutorial, let's examine a couple of tricks involved.

### **2. Two common types of semaphores**

Semaphore is a very generic synchronization construct, allowing virtually every type of synchronization. However, they are also one of the most difficult to employ. Just like any other constructs that are difficult to use, we make them easier by identifying **patterns** of their usage.

There are two common usages of semaphores. One is to use it as a **lock** (or "**mutex**"). This is typically used to **protect shared data** from simultaneous access by multiple processes. At the beginning, a semaphore is initialized to 1. A process must decrement the semaphore before using the shared data, and must increment it afterwards. In most cases, the process which decrements a semaphore will eventually increment it, although sometimes there are exceptions, where one process locks the mutex and expects another process to unlock it (conceptually we say the first process "transfers the lock" to the second process, even though they actually do nothing to "make" that happen). All the semaphores used by the solution of tutorial 5 are locks, and the *wrt* lock is transferred among readers (i.e., locked by one reader while unlocked by another).

It is also common to use semaphores to implement **condition waiting**. In this usage, the semaphore is initially set to 0. A process which wants to wait will decrement the semaphore. Another process can then increment the semaphore to resume the waiting process. This is sometimes a bit clumsy to use: one might need to make sure that there is really somebody waiting before incrementing the semaphore (otherwise, if a process increments the semaphore when no process is waiting, the next process that wants to wait will not successfully wait). This check can be done by keeping a shared variable to count the number of waiting processes. This shared variable must, of course, be protected by a lock-typed semaphore.

### 3. Dealing with deadlocks

The possibility of deadlock is the primary reason why synchronization problems are difficult. Just like race-conditions, deadlocks are not reproducible (i.e., most of the time you cannot reproduce a deadlock by re-running your program). Furthermore, even if you can identify the exact sequence that causes the deadlock, it is usually difficult to modify the program to avoid it. Usually, when you patch the problem, you introduce another deadlock, and you end up having to redesign the scheme from ground up.

The most important tool to deal with them is simplicity of the synchronization scheme: the simpler the scheme, the less likely that it has potential deadlocks deeply hidden.

It worths to review the necessary conditions for deadlocks: mutual exclusion, hold-and-wait, no preemption and circular wait. A lock-typed semaphore has both “mutual exclusion” and “no preemption” already. If we need to deal with multiple of them, most of the time we need hold-and-wait as well. Adding in “circular wait”, all 4 conditions becomes satisfied. Recall that lock-type semaphores don’t have multiple copies. So the 4 conditions are also sufficient for deadlocks. In other words, **if your program has circular wait, it will deadlock.**

Let’s see an example. Suppose your program has two critical sections, as below:

```
WAIT(lock1); /* A */
WAIT(lock2);
... /* Critical Section 1 */
SIGNAL(lock1);
SIGNAL(lock2);
...
WAIT(lock2); /* B */
WAIT(lock1);
... /* Critical Section 2 */
SIGNAL(lock1);
SIGNAL(lock2);
```

It is possible that one process executes the instruction marked as A when another executes the instruction marked as B. If both succeeds, both will be unable to get the next lock, so a deadlock occurs. The problem is circular wait: around A, *lock1* is held when waiting for *lock2*. Around B, *lock2* is held when waiting for *lock1*. If one of the two sequences is reversed, the deadlock is resolved. Unluckily, sometimes something needs to be done between the locks, and it is difficult to reverse the lock statements. This might reveal a need for redesigning the synchronization scheme.