Lecture 14 – Semaphores:

[redacted]

Problems With Locks

One of the biggest problems with using locks is that they may cause a lot of busy waiting. This is because they require you to keep checking for a condition over and over again until it is met. This technique inefficiently uses system resources. You have to waste resources in checking for the condition instead of utilizing them on other tasks. Furthermore, on a uniprocessor, the other threads that must run to make this condition hold are prevented from running while the original thread is repeatedly checking this condition. Therefore other techniques, such as using condition variables, are much more efficient on uniprocessors. The tradeoffs on a multiprocessor are more complex and depend, among other factors, on how the size of the critical section compares with that of the context switching routine.

Locks vs. Condition Variables

Locks

```
mutex_lock m;
int count == 0;

Produce() {
            while (count == N);

            … ADD TO BUFFER …

            mutex_lock(m);
            count++;
            mutex_unlock (m);
}

Consume() {
            while (count == 0);

            … REMOVE FROM BUFFER …

            mutex_lock(m);
            count--;
            mutex_unlock (m);
}
```

Condition Variables

```
cond_t  not_full, not_empty;
mutex_lock m;
int count == 0;

Produce() {
            mutex_lock (m);
            if (count == N)    wait (not_full,m);

            … ADD TO BUFFER, count++ …

            signal (not_empty);
            mutex_unlock (m);
}

Consume() {
            mutex_lock (m);
            if (count == 0)    wait (not_empty,m);

            … REMOVE FROM BUFFER, count--
…



            signal (not_full);
            mutex_unlock (m);
}
```

By looking at the two pieces of code above, you can see that the left side uses mutex locks while the right side uses condition variables. The code on the right is much more efficient because it, in effect, breaks up the code into smaller critical sections. Mutex locks use a form of polling in which they keep checking to see if a certain condition is met. On the other hand, condition variables use event notification so that threads receive a notification or a signal whenever they are ready to continue. Event notification is always more efficient than polling on a uniprocessor while polling can be more efficient on a multiprocessor. This happens on a multiprocessor when critical sections are smaller than the context switching routine. You can check out the pthread example if you want to learn more about condition variables.

When Are Condition Variables Inadequate to Use?

One downside of condition variables is that signals may be lost. This happens when a signal is sent and no thread has called a wait before this. Look at the following example to see how a program can fail if signals are forgotten.

```
PROCESS A:

A1 // one or more statements
a  // single statement
signal (a_done);
A2 // one or more statements
```

```
PROCESS B:

B1 // one or more statements
wait (a_done);
b  // single statement
B2 // one or more statements
```

In this example, in executions (i.e., instruction interleavings) where *signal(a_done)* executes before *wait(a_done)*, it causes process B to starve since the signal sent by A was lost. At this point it is useful to go back to the code for Producer-Consumer that uses CVs and make sure you understand why this "lost signal" problem does not arise there.

The Invention of Semaphores

Semaphores were created by Dijkstra to solve the problem of signals being lost. Dijkstra was a Dutch mathematician and computer scientist. We will use the book, "*Little Book of Semaphores*" which can be found for free online to solve some semaphore problems. It may be useful to read the chapters that define what they are (Chap. 2) and how they can be used for creating basic synchronization constructs (Chap. 3).

What is a Semaphore?

Informally, a semaphore is a condition variable that has memory to see what waits and signals Have been done so far. This memory is provided by associating a semaphore with an integer whose value is manipulated during signal and wait operations to reflect the difference (num. signals – num. waits) so far.

Semaphores have the following properties:

1. When you create it you initialized its value to an integer
2. Operations that you can do:

   -
     ```
     Signal(S){
         S++;
     }
     ```

   -
     ```
     Wait(S){
         While S <= 0; // Do nothing
         S--;
     }
     ```
3. IMPORTANT: Wait and signal must be ATOMIC. This would be achieved by using a mutex lock. So underneath each semaphore, there is a mutex lock that causes some busy waiting. (more on this below)
4. You can never read the value of a semaphore.
5. When a thread decrements a semaphore, the thread gets blocked if the result is negative.
6. When a thread increments a semaphore, if there are threads waiting, one of those threads will get unblocked.

Whenever a thread notifies the scheduler that it cannot proceed, it will block itself. The scheduler will then prevent the thread from running until an event causes the thread to become unblocked.

Semaphores For Mutex

They can be accessed by the two atomic operations, wait and signal, that are shown above. They accomplish this by using a mutual exclusion lock inside the semaphore. It is important to note that they don't completely eliminate busy waiting, they just reduce it.

Whereas the above description of wait and signal was based on busy waiting, semaphores are usually implemented to reduce busy waiting by implementing atomic signal and wait as shown below.

```
signal (S) {
    S++;
        if (S <= 0) {
                remove a process P from queue
                add P to ready queue
}


wait (S) {
    S--;
    if (S < 0) {
            add to queue of processes
            blocked on this semaphore
            block();
    }
}
```

Consequences of the Definition of Semaphores

- There is no way to know whether a thread will become blocked before it is decremented.
- If one thread increments a semaphore and another thread gets woken up both threads will run concurrently. It will not be possible to tell which thread will continue immediately.
- When you signal a semaphore, you don't know whether or not if there are any waiting threads.

Semaphore Values

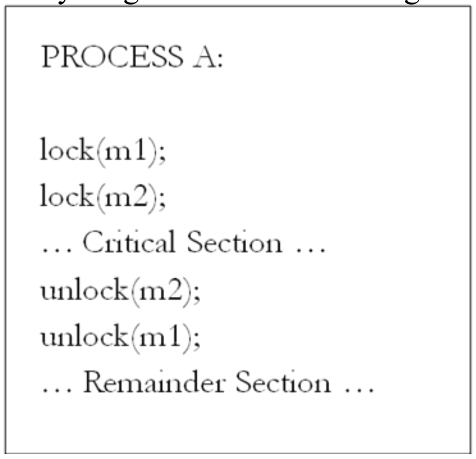
1. **Positive Value**: A positive value represents the number of threads can increase without blocking another thread.
2. **Negative Value**: A negative value represents the number of threads that are block or waiting.
    - For example: -3 represents that there are 3 blocked/waiting threads
3. **Zero**: A zero value represents that there are no blocked/waiting threads and that the next thread that tries to decrement it will become blocked.

Semaphore Types

Semaphores can have two different types. The first type is a counting semaphore. A **counting semaphore** has no restriction on the value that a semaphore can take. A **binary semaphore** can only be an integer value between 0-1. These are for mutual exclusion locks.

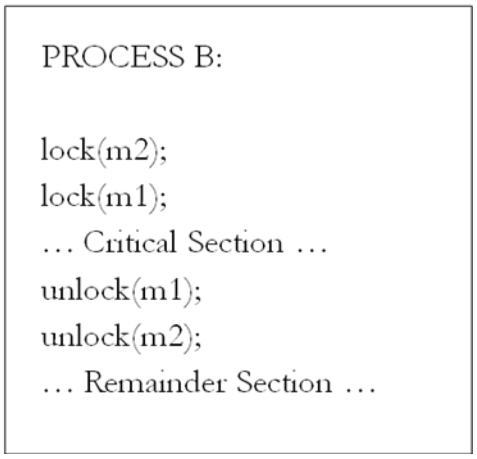
Common Pitfalls

You have to remember to lock before you unlock. Don't use locks for signaling, only use for mutual exclusion. You have to remember to unlock what you locked. Avoid locking inside of while loops because this can cause many problems. If you are using multiple locks, be sure to acquire them in the same order. If you fail to do this, you could get stuck in a deadlock where everything is locked and waiting.



```
PROCESS A:

lock(m1);
lock(m2);
… Critical Section …
unlock(m2);
unlock(m1);
… Remainder Section …
```



```
PROCESS B:

lock(m2);
lock(m1);
… Critical Section …
unlock(m1);
unlock(m2);
… Remainder Section …
```

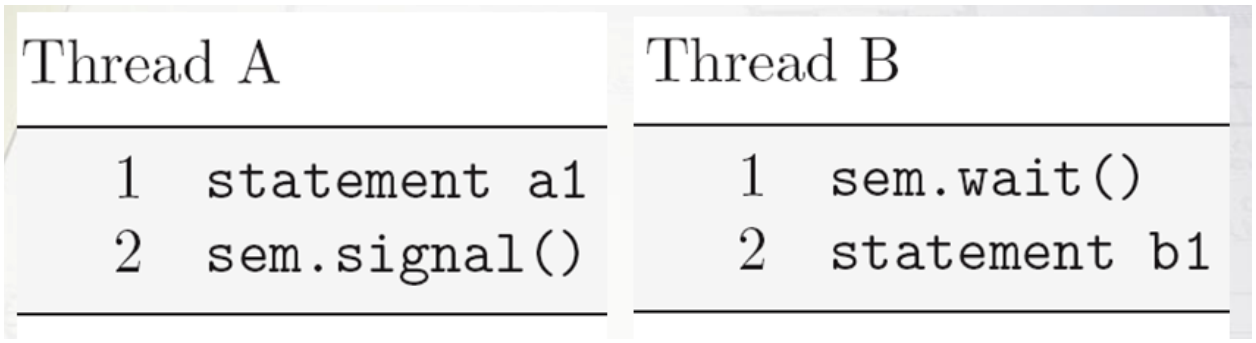
In the example above, two locks (m1 and m2) were created. However they were locked and unlocked in different orders. This causes them to be stuck in a deadlock. Whenever you are working with multiple locks, be sure to unlock/lock in the same order.

Semaphore Usage

Semaphores are not just used for mutex. They are also used in signaling, rendezvous, multiplex, barrier, re-usable barrier, queues, and FIFO queues.

Semaphores for Signal

Semaphores can be used to make a thread to wait until a signal is received before they proceed with running the thread. This is good for synchronizing statements by triggering one event from another. In the example below, thread B is waiting to receive a signal from thread A before it continues to run its routine. Threads A and B are allowed to edit the same data (given that the data is not static with respect to either threads) with a predicted result.

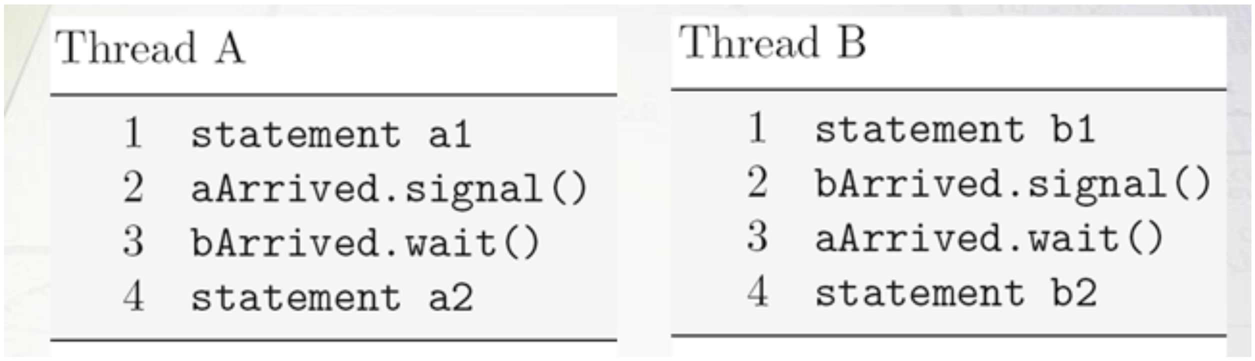


Thread A	Thread B
1 statement a1	1 sem.wait()
2 sem.signal()	2 statement b1

Semaphores for Rendezvous

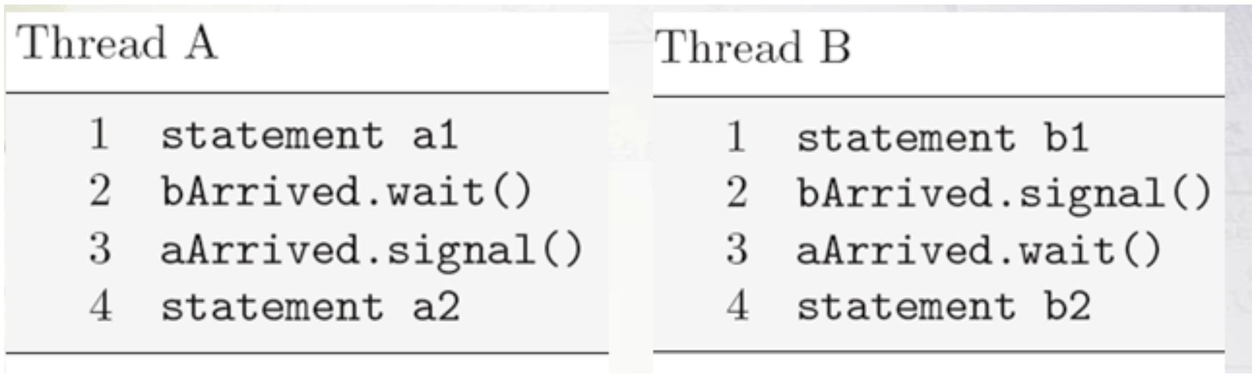
The purpose of semaphores in rendezvous is to allow threads to send and receive each other's signals before they proceed with running their statements simultaneously. This can be used to allow threads to perform different tasks at the same time, but they cannot synchronize because there are no checks in place for when they are editing the same data and the result can be undeterminable if this happens. To show an example, we have two threads waiting to rendezvous and there are two ways to go about resolving it.

Solution 1:



Thread A	Thread B
`statement a1`	`statement b1`
`aArrived.signal()`	`bArrived.signal()`
`bArrived.wait()`	`aArrived.wait()`
`statement a2`	`statement b2`

Solution 2:



Thread A	Thread B
`statement a1`	`statement b1`
`bArrived.wait()`	`bArrived.signal()`
`aArrived.signal()`	`aArrived.wait()`
`statement a2`	`statement b2`

bArrived and aArrived are two semaphores created to send signals to threads A and B, respectively. In solution 1, both threads send their signals and waited together at the same time. This ensures that both threads do not proceed with their routines until both have received the signals. In solution 2, thread A is first waiting for the signal from thread B to arrive before sending its own signal to thread B. This can cause thread A to immediately proceed with its next statement after sending the signal while B is waiting to receive thread A's signal.