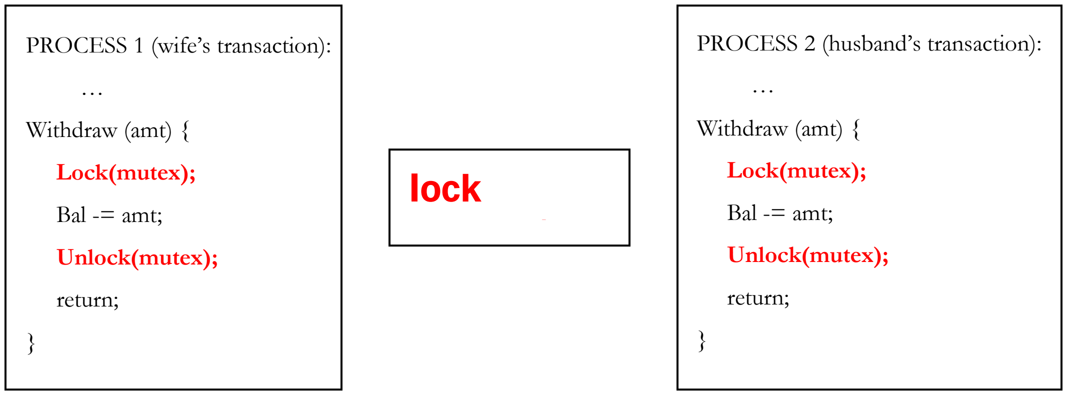
[redacted]

CPU Management, Process Synchronization

[redacted]

From previous lectures recall Mutex Locks and their uses for preventing access to “critical sections” Two concrete concurrent programs we considered were a shared bank account between a husband and wife (Figure 1) and the Producer & Consumer problem

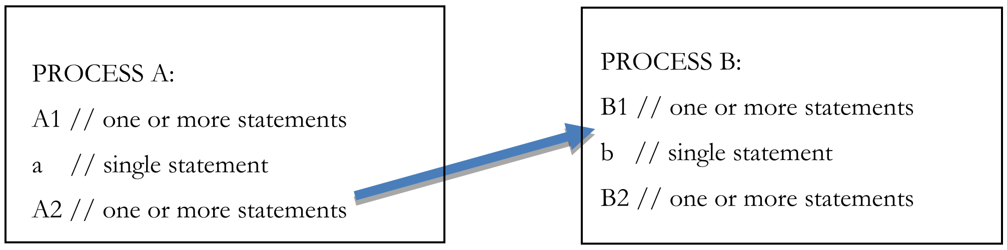


```
PROCESS 1 (wife’s transaction):
      …
Withdraw (amt) {
    Lock(mutex);
    Bal -= amt;
    Unlock(mutex);
    return;
}
```

```
PROCESS 2 (husband’s transaction):
      …
Withdraw (amt) {
    Lock(mutex);
    Bal -= amt;
    Unlock(mutex);
    return;
}
```

Figure 1

Below is another example.



```
PROCESS A:
A1 // one or more statements
a   // single statement
A2 // one or more statements
```

```
PROCESS B:
B1 // one or more statements
b   // single statement
B2 // one or more statements
```

Figure 2

For use in signaling, Figure 2 shows two processes, Process A and Process B. Each process has many lines of code. Embedded within A's code is a line referred to as "a"; B has the same layout with "b" embedded within its code. We are allowed to interleave any lines of A and B, but the condition for "a" executing before "b" must be met.

Based on the previous examples we learned in class, one may think that all we need is a lock and unlock surrounding "a" and "b." But this is the wrong way! It doesn't allow interleaving of code that ensure that "a" comes before "b."

Another approach is to have an unlock after "a" and a if-then-loop before "b" (Figure 3). This approach would cause "b" to occur after "a," The lock, m, would initially be set to locked. But this doesn't work either. When the if statement is encountered, we would enter the loop but never exit it.

```
PROCESS A:
A1 // one or more statements
a    // single statement
Unlock(m)
A2 // one or more statements
```

```
PROCESS B:
B1 // one or more statements
If(lock(m)){while 1;}
b   // single statement
B2 // one or more statements
```

lock m = Locked;

Figure 3

Instead, we can lock using Test & Set or lock using Peterson's algorithm. Recall the Test & Set Function, shown in Figure 4.

```
boolean TestAndSet (boolean *target)
 {
     boolean rv = *target;
     *target = TRUE;
     return rv:
 }
```

Figure 4

Both the Test & Set and Peterson's algorithm can be generalized into the following code:

```
PROCESS A:                              PROCESS B:
A1 // one or more statements            B1 // one or more statements
a   // single statement                 Lock(m);
Unlock(m);                              b   // single statement
A2 // one or more statements            B2 // one or more statements
                        lock m;
```

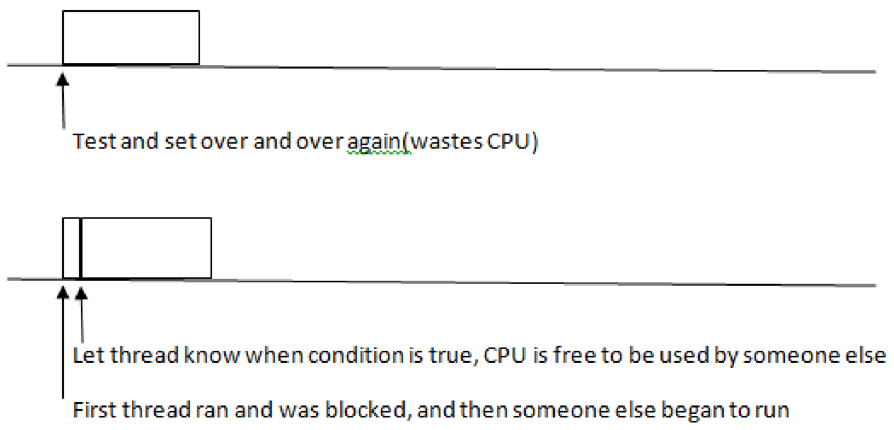
For the Test & Set and Peterson's algorithm, we would change the Unlock and Lock statements. Test & Set's Unlock(m) would be m = false and the Lock(m) would be "while(test&set(&m));" m is the Boolean variable representing the value of the mutex lock. It is important to note that m is initialized to the value of FALSE in order to ensure that test&set works to keep "b" from executing before "a." For Peterson's algorithm, we would have lock m be two variables, Flag[A] and Flag[B]. We would also need a variable called turn (to indicate whose turn it is: A or B). Flag[A] is initially set to TRUE, while Flag[B] is set to FALSE. Unlock(m) would be replaced with Flag[A] = FALSE; Lock(m) with Flag[B] = TRUE; turn =A; while (Flag[A] && turn ==A); Again note the initialization of the shared variables to ensure "a" occurs before "b."

The shortcomings of locks:

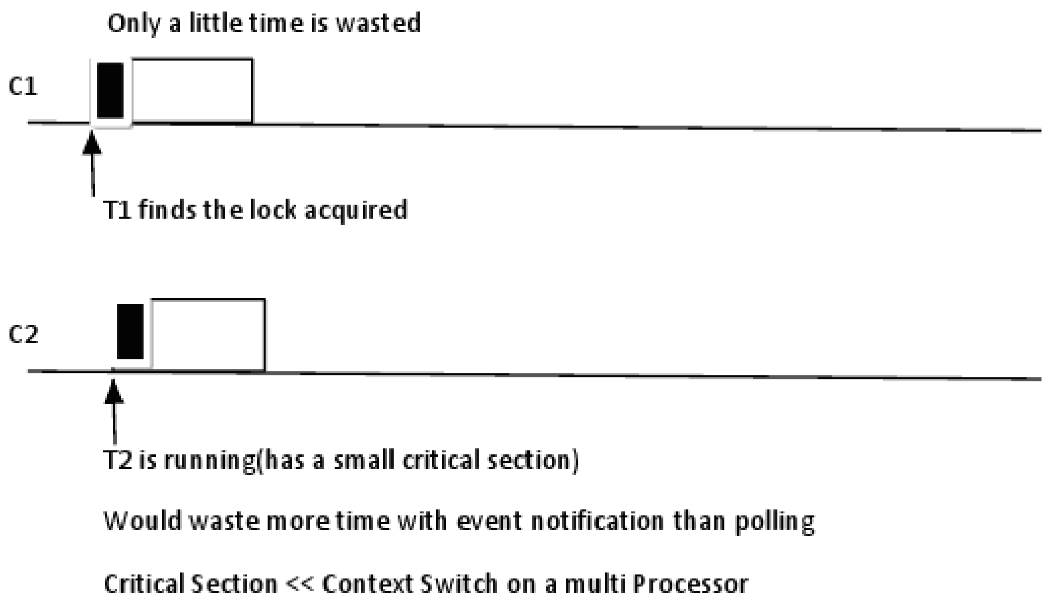
1. For certain synchronization problems, locks can be "clumsy"
2. Locks can cause wastage of CPU cycles (and hence time) due to busy waiting.
3. Can be harder to debug

Certain implementations of locks pose restriction on their initialization and usage locks. We saw in the above examples that for signaling we need to be able to initialize the mutex lock in a specific way. This may not be possible with certain implementations.

Mutex locks are also called spin locks due to the fact that if one thread has a lock, the other threads will try to acquire the lock by "spinning." in their entry sections. This manner of checking for a condition to become true is called "polling". In order to help decrease this wastage of CPU cycles, we can use event notification instead of pollings. Both have their own trade-offs to using, however. Polling is preferable in some situations, however, such as multiple CPUs in a system. Consider the following figures to understand why, when critical section is smaller than the context switching routine, on a multiprocessor polling may be preferable to event notification.


Test and set over and over again(wastes CPU)
Let thread know when condition is true, CPU is free to be used by someone else
First thread ran and was blocked, and then someone else began to run

Comparison of polling and event notification on a uniprocessor


Only a little time is wasted
C1
T1 finds the lock acquired
C2
T2 is running(has a small critical section)
Would waste more time with event notification than polling
Critical Section << Context Switch on a multi Processor

Timelines for a two-processor system, C1 and C2 refer to the two CPUs

Event Notification is similar to an alarm clock signaling you to wake up. Instead of having a thread, say T1, poll until a condition is satisfied by T2, we allow both threads to send the other thread a notification what condition we wish to have satisfied (sent by T1) and when it becomes true (sent by T2). We must ensure that T2 successfully conveys the message to T1. In order to ensure that event notification occurs correctly, we need to use condition variables.

Locks go hand in hand with using condition variables. This will be seen in the next example as we use condition variables to solve the producer-consumer problem.

Condition variables are variables to indicate that certain conditions are now true. They can be one of two options: wait operation or a signal operation. The wait operation suspends the thread instead of letting it poll. This is the notifying of what condition we want to be satisfied. The signal operation occurs after a desired event occurs (i.e. condition variable changes from false to true). If there are threads waiting on the CV, the implementation of CV (sometimes called a "monitor") will allow one thread to execute. If there is no waiting thread, the signal is treated as if it never occurred and is lost.

If we return to our signaling example and use condition variables, it would look like Figure 5.

```
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
```

```
cond_t a_done;
```

Figure 5

If instead we used condition variables for the Producer-Consumer model, we see a problem. For one, the checking of a condition followed by a wait/signal should be done atomically as count manipulations can become interleaved. Instead we add the line "mutex_unlock (m)" after each signal line. This is shown in Figure 6 The bolded code is the edited version to fix the problem.

```
cond_t  not_full, not_empty;
int count == 0;
Produce() {
            if (count == N)   wait (not_full);

            ... ADD TO BUFFER,  count++ ...
            signal (not_empty);
}
Consume() {
            if (count == 0)       wait (not_empty);
            ... REMOVE FROM BUFFER,  count-- ...
            signal (not_full);
}
```

```
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);
}
```

Figure 6

Have we completely eliminated the busy waiting that locks caused? We still need to use locks; though we have reduced busy waiting time, we cannot fully eliminate it. There is busy waiting outside and inside of a critical section.

The data structures and functions used in pthreads to emulate condition variables are: pthread_cond_t condition = PTHREAD_COND_INITIALIZER (or pthread_cont_init (condition, attr)), pthread_cond_wait(condition, mutex), and pthread_cond_signal(condition).

There are situations in which using condition variables will not work completely in the way we want it to. The signal operations can be lost as we know that a signal() is treated as it never happened if a thread isn't waiting. Other times, there isn't enough memory to remember all the signal operations that are occurring. Next lecture, we will discuss semaphores who can offer this "memory." They keep count of the number of signals and waits.