User versus Kernel Threads — Last Time

- What's a kernel thread?
- What's a user thread?

Uniprocessors - kernel threads are much more expensive and less flexible than user threads.

Multiprocessors - kernel threads overcome their costs by exploiting parallelism better than user threads.

$\Longrightarrow$ Multiprocessors complicate OS design considerably.

Synchronization — Today

What kind of knowledge and mechanisms do we need to get independent processes to communicate and get a consistent view of the world (computer state)?

Example: Too Much Milk

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

Synchronization Terminology

Synchronization: Using atomic operations to ensure cooperation between threads.

Mutual Exclusion: ensures that only one thread does a particular activity at a time. One thread doing an activity *excludes* all the others from doing it at that time.

Critical Section: is a piece of code that only one thread can execute at a time.

Lock: prevents another process from doing something.

1. Lock before entering a critical section, or before accessing shared data.
2. Unlock when leaving a critical section or when access to shared data is complete
3. Wait if locked

$\Longrightarrow$ All synchronization involves waiting.

Solution 1 to Too Much Milk

What are the correctness properties for the too much milk problem?

- Only one person buys milk at a time.
- Someone buys milk if you need it.

Restrict ourselves to atomic loads and stores as building blocks.

1. Leave a note (a version of lock)
2. Remove note (a version of unlock)
3. Do not buy any milk if there is note (wait)

```
Thread A                        Thread B

if (noMilk & NoNote) {          if (noMilk & NoNote) {
    leave Note;                     leave Note;
    buy milk;                       buy milk;
    remove note;                   remove note;
}                               }
```

Does this work?

Solution 2 to Too Much Milk

How about using labeled notes so we can leave a note before checking the the milk?

```
Thread A                    Thread B

leave note A                leave note B
if (noNote B) {             if (noNote A) {
   if (noMilk){                if (noMilk){
      buy milk;                   buy milk;
   }                           }
}                             }
remove note;                  remove note;
```

Does this work?

Solution 3 to Too Much Milk

```
   Thread A                      Thread B

   leave note A                  leave note B
X: while (Note B) {           Y: if (noNote A) {
      do nothing;                    if (noMilk){
   }                                    buy milk;
   if (noMilk){                      }
       buy milk;                 }
   }                             remove note B;
   remove note A;
```

Does this work?

Solution 3 to Too Much Milk

Discussion of correctness:

- At point Y, either there is a note A or not.
  1. If there is no note A, it is safe for thread B to check and buy milk, if needed. (Thread A has not started yet).
  2. If there is a note A, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.
- At point X, either there is a note B or not.
  1. If there is not a note B, it is safe for A to buy since B has either not started or quit.
  2. If there is a note B, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.
- Thus, thread B buys milk (which thread A finds) or not, but either way it removes note B. Since thread A loops, it waits for B to buy milk or not, and then if B did not buy, it buys the milk.

Is Solution 3 a good solution?

1. It is too complicated - it was hard to convince ourselves this solution works.
2. It is asymmetrical - thread A and B are different. Thus, adding more threads would require different code for each new thread and modifications to existing threads.
3. A is *busy waiting* - A is consuming CPU resources despite the fact that it is not doing any useful work.

$\Longrightarrow$ This solution relies on loads and stores being atomic.

Language Support for Synchronization

Have your programming language provide atomic routines for synchronization.

Locks - one process holds a lock at a time, does its critical section releases lock.

Semaphores - more general version of locks.

Monitors - connects shared data to synchronization primitives.

$\Longrightarrow$ All of these require some hardware support, and waiting.

Locks

Locks - provide mutual exclusion to shared data with two "atomic" routines:

Lock::Acquire - wait until lock is free, then grab it.

Lock::Release - unlock, and wake up any thread waiting in Acquire.

Rules for using a lock:

- Always acquire the lock before accessing shared data.
- Always release the lock after finishing with shared data.
- Lock is initially free.

Implementing Too Much Milk with Locks

Too Much Milk

```
Thread A                    Thread B

Lock->Acquire();            Lock->Acquire();
if (noMilk){                if (noMilk){
   buy milk;                   buy milk;
}                           }
Lock->Release();            Lock->Release();
```

This solution is clean and symmetric.

How do we make Lock::Acquire and Lock::Release atomic?

Summary

- Communication among threads is typically done through shared variables.
- Critical sections identify pieces of code that cannot be executed in parallel by multiple threads, typically code that accesses and/or modifies the values of shared variables.
- Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.

Next Time

Hardware to make Lock::Acquire and Lock::Release atomic.