Synchronization — Last Time

Too much milk using busy waiting:

```
   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;
```

Too much milk using locks:

```
Thread A                   Thread B

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


Implementing Mutual Exclusion — Today

- Hardware support for synchronization.
- Building higher-level synchronization abstractions using this hardware support

The Big Picture:

- The *Too Much Milk* problem illustrated that implementing a concurrent program directly with loads and stores (**load/store**) is tricky and error prone.
- If instead we use high-level operations, like semaphores, we need some low-level hardware support.

What we have and what we want:

	Concurrent Programs		
Hardware: Low-level atomic operations	load/store	interrupt disable	test&set
Software: High-level atomic operations	**locks**	semaphores send & receive	monitors

Ways to Implement Locks

Remember interrupts? When the OS is executing, no other process may interrupt execution. One implementation of this functionality sets and unsets the interrupt status bit.

There are two ways the CPU scheduler gets control:

1. **Internal Events:** the thread does something to relinquish control (e.g., I/O).
2. **External Events:** interrupts (e.g., time slice) cause the scheduler to take control away from the running thread.

On a uniprocessor, a chunk of code that does not do I/O will be atomic as long as a context switch does not occur in the middle of the operation.

1. **Internal Events:** The thread can easily prevent these by not requesting any I/O operations during a critical section.
2. **External Events:** The thread could prevent external events by disabling interrupts, in effect, telling the hardware to delay handling any external events until after the thread is finished with its atomic operation.

   Why not have the OS support these system calls?

   Lock::Acquire() // disable interrupts
   Lock::Release() // enable interrupts

(a) on a uniprocessor?

(b) on a multiprocessor?

Implementing Locks by Disabling Interrupt

- For a uniprocessor, we can disable interrupts for high-level primitives like locks, whose implementations are private to the kernel.
- The kernel thus ensures that interrupts are not disabled forever, just like it already does during interrupt handling.

  For example, the OS can implement locks as follows:

```
class Lock {
  public:
    void Acquire();
    void Release();
  private:
    int value;
    Queue Q;
}
Lock::Lock {
  // lock is free
  value = 0;
  // queue is empty
  Q = 0;
}
```

```
Lock::Acquire(T:Thread){
  disable interrupts;
  if (value == BUSY) {
     add T to Q
     T->Sleep();
  } else {
     value = BUSY;
  }
  enable interrupts; }

Lock::Release() {
  disable interrupts;
  if queue not empty {
     take thread T off Q
     put T on ready queue
  } else {
     value = FREE
  }
  enable interrupts; }
```

Wait Queues

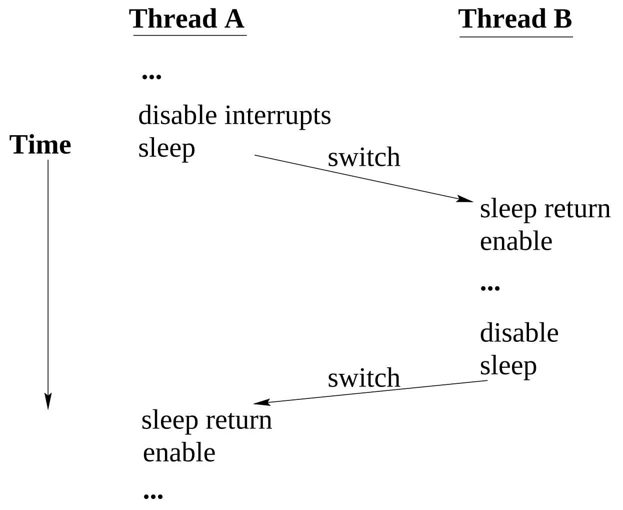
When should Acquire re-enable interrupts when going to sleep?

- **Before putting the thread on the wait queue?** No, Release could check the queue, and not wake up the thread.
- **After putting the thread on the wait queue, but before going to sleep?** No, Release could put the thread on the ready queue, but it could already be on the ready queue. When the thread wakes up, it will go to sleep, missing the wakeup from Release.
- **In the OS scheduler.** When the scheduler selects and starts the next running process, it can enable interrupts.
- **The running thread itself.** The first thing a thread does when it starts to execute is enable interrupts.

We still have a problem with multiprocessors.

Example

When the sleeping thread wakes up, it returns from Sleep back to Acquire. Interrupts are still disabled, so its ok to check the lock value, and if it is free, grab the lock and turn on interrupts.



Atomic read-modify-write Instructions

Atomic read-modify-write instructions *atomically* read a value from memory into a register and write a new value.

- Straightforward to implement simply by adding a new instruction on a uniprocessor.
- On a multiprocessor, the processor issuing the instruction must also be able to *invalidate* any copies of the value the other processes may have in their cache, i.e., the multiprocessor must support some type of *cache coherence.*

Example read-modify-write instructions:

Test&Set: (most architectures) read a value, write '1' back to memory.

Exchange: (x86) swaps value between register and memory.

Compare&Swap: (68000) read value, if value matches register value r1, exchange register r2 and value.

Load Linked and Conditional Store: (R4000,Alpha) read a value in one instruction, do an operation, when store occurs, check if value has been modified in the meantime. If not, do the store. If it has changed, abort, and jump back to the read. (This instruction is designed to fit better with load/store architectures. What's a load/store architecture?)

Implementing Locks with Test&Set

Test&Set reads a value, writes '1' to memory, and returns the old value.

```
class Lock {
  public:
    void Acquire();
    void Release();
  private:
    int value;
}
Lock::Lock {
  value = 0;
}
Lock::Acquire() {
  while (test&set(value) == 1)  //while Busy, do nothing
    ;
}
Lock::Release() {
  value = 0;
}
```

- If lock is free (value = 0), test&set reads 0, sets value to 1, and returns 0. The Lock is now busy: the test in the while fails, and Acquire is complete.
- If lock is busy (value = 1), test&set reads 1, sets value to 1, and returns 1. The while continues to loop until a Release executes.

Busy Waiting

What's wrong with

```
Lock::Acquire():
  while (test&set(value) == 1)  //while Busy, do nothing
    ;
```

- What is the CPU doing?
- What could happen to threads with different priorities?

How can we get the waiting thread to give up the processor, so the releasing thread can execute?

Locks using Test&Set with minimal busy-waiting

- Can we implement locks with test&set without any busy-waiting or disabling interrupts?
- No, but we can minimize busy-waiting time by only busy-waiting to atomically check the lock value and if the lock is busy, give up the CPU.

```
class Lock {            // same declarations as the
   ...                  // previous version, except
   private int guard;
}
Lock::Acquire(T:Thread) {
  while (test&set(guard) == 1)  ;  // busy wait
  if (value != FREE) {
     put T on Q;
     T->Sleep() & set guard = 0;
  } else {
     value = BUSY;
     guard = 0;
  } }

Lock::Release() {
  while (test&set(guard) == 1) ;  // busy wait
  if Q is not empty {
     take T off Q;
     put T on ready queue;
  } else {
     value = FREE;
  }
  guard = 0;
}
```

Summary

⟹ Without disabling interrupts, we have to do some busy waiting to do synchronization.

- In our original solution, the busy waiting occurred in the user's critical section, which could last for a long time (e.g., the critical section could be to write an entire file.)
- Our final solution moves the busy waiting into the critical section of the lock acquire and release, which we implemented in the OS as very short routines.
- Thus, we still have to do some busy waiting, but it won't be for too long.

In short, Load/store, disabling and enabling interrupts, and atomic read-modify-write instructions are all ways that we can implement higher level atomic operations.

Next Time

Semaphores: another high-level synchronization mechanism