Announcement

• Project 1 deadline extended to 5/2 (by a week)
  ♦ Don’t forget to also work for your midterm!
  ♦ Extra credit problem added (worth 5% above 100%)

• Project 2 released on 4/27

• HW1 solution and grades to be released soon

• May switch semaphore and CV lectures

• More lab hours next week
### “Too Much Milk” Problem

<table>
<thead>
<tr>
<th>Roommate A</th>
<th>Roommate B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look in fridge: out of milk</td>
<td>Look in fridge: out of milk</td>
</tr>
<tr>
<td>Leave for Ralphs</td>
<td>Leave for Ralphs</td>
</tr>
<tr>
<td>Arrive at Ralphs</td>
<td>Arrive at Ralphs</td>
</tr>
<tr>
<td>Buy milk</td>
<td>Buy milk</td>
</tr>
<tr>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
</tbody>
</table>

- How to enforce mutual exclusion?
A Possible Solution?

- Process can get context switched after checking milk and note, but before leaving note
- Why does it work for human?
Why does it work for people?

- Human can perform \textit{test} (look for other person & milk) and \textit{set} (leave note) at the same time.

- What we want is:

```
Acquire(lock);
if (noMilk)
    buy milk;
Release(lock);
```
Locks

• A lock is an object in memory providing two operations
  ♦ acquire() (or lock()): to enter a critical section
  ♦ release() (or unlock()): to leave a critical section

• Threads pair calls to acquire and release
  ♦ Between acquire/release, the thread holds the lock
  ♦ acquire does not return until any previous holder releases
  ♦ What can happen if the calls are not paired?
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

acquire(lock);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);

acquire(lock);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);

acquire(lock);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);

♦ What happens when blue tries to acquire the lock?
♦ Why is the “return” outside the critical section? Is this ok?
♦ What happens when a third thread calls acquire?
Implementing Locks

• How do we implement locks? Here is one attempt:

```c
struct lock {
    int held = 0;
};

void acquire (lock) {
    while (lock.held)
        ;
    lock.held = 1;
}

void release (lock) {
    lock.held = 0;
}
```

• This is called a **spinlock** because a thread spins waiting for the lock to be released

• Does this work?
Implementing Locks

- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```c
struct lock {
    int held = 0;
};

void acquire (lock) {
    while (lock->held)
        ;
    lock->held = 1;
}

void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition.
Implementing Locks

- The problem is that the implementation of locks has critical sections, too
- How do we stop the recursion?
- The implementation of acquire/release must be atomic
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes “all or nothing”
- How do we make them atomic?
How do we make a piece of code atomic?

- What can cause the few lines to be not atomic?

- What causes context switches?

- Recall -- only way the OS dispatcher regains control is via interrupts (including syscalls)
Disabling Interrupts

• A possible implementation of lock using interrupts

```c
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

• Can two threads disable interrupts simultaneously?
On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- But also blocks other (important) interrupts
- Disabling interrupts is insufficient on a multiprocessor
  - Interrupts are only disabled on a per-core basis
Need more help from hardware!

Why does it work for people?

- Human can perform test (look for other person & milk) and set (leave note) at the same time.
Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to true
  - Return the old value

- Hardware executes it atomically!

```cpp
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on “flag”
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?
Using Test-And-Set

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
};
void acquire (lock) {
    while (test-and-set(&lock->held))
        ;
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- What about multiprocessors?
Problems with Spinlocks

- The problem with spinlocks is that they are wasteful (busy wait!)
  - A thread spins on a lock for the entire critical section
  - The critical section can be even longer if the lock holder thread is scheduled off
Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks
- Problem: Critical sections (CS) can be long

Spinlocks:
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater chance for lock holder to be interrupted

Disabling Interrupts:
- Doesn’t work on multiprocessor
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-Level Synchronization

- Spinlocks are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are “primitive” – can’t do anything other than mutual exclusion
- Need higher-level synchronization primitives that:
  - Move waiters to the blocked queue (block waiters)
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we’ll use our “atomic” locks as primitives to implement them
[lec4] Process State Transition

Scheduler dispatch

Running

Wait for resource

Blocked

Create a process

Resource becomes available

Terminate
[lec4] State Queues

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
void acquire (lock) {
    if (lock->held == 0) {
        lock->held = 1;
        return;
    }
    put current thread on lock->Q;
    go to sleep;
}

void release (lock) {
    if (lock->Q is empty)
        lock->held = 0;
    if (lock->Q is not empty)
        move a waiting thread to the ready queue;
}
```

```c
struct lock {
    int held = 0;

    queue Q;
}
```
Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
void acquire (lock) {
    while (test-and-set(lock->guard)) ;
    if (lock->held == 0) {
        lock->held = 1;
        lock->guard = 0;
    }
    return;
}

put current thread on lock->Q;
lock->guard = 0;
go to sleep;
}
```

```c
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}
```

```c
void release (lock) {
    while (test-and-set(lock->guard)) ;
    if (lock->Q is empty)  
        lock->held = 0;
    if (lock->Q is not empty)  
        move a waiting thread to the ready queue;
    lock->guard = 0;
}
```
Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}

void acquire (lock) {
    disable interrupts;
    while (test-and-set(lock→guard)) ;
    if (lock→held == 0) {
        lock→held = 1;
        lock→guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock→Q;
    lock→guard = 0;
    go to sleep;
    enable interrupts;
}

void release (lock) {
    disable interrupts;
    while (test-and-set(lock→guard)) ;
    if (lock→Q is empty) {
        lock→held = 0;
    } else {
        move a waiting thread to the ready queue;
        lock→guard = 0;
        enable interrupts;
    }
}
```
Deep Thinking

• Why is this busy waiting (the while loop) not a concern?
  ♦ What’s our critical section here?

• Can we remove the disable/enable interrupts?
  ♦ With interrupts, when a process that gets the guard (pass the while loop) get context switched out, all other wait processes on other cores will be busy waiting
Next time...

• Read Chapters 30, 31

• Work on your project 1!