Administrivia

- Homework 1 and Project 0 due today
- Homework 2 and Project 1 out
- Google form for project team => github account
- Discussion session tomorrow to go over the first two questions of project 1 and some questions from Piazza
Thread Implementations

- User-level thread implementation
- Kernel-level thread implementation
- Pros and cons?
The problem is that the execution of the two threads can be interleaved:

```c
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
```

What is the balance of the account now?
Is the bank happy with our implementation?
This problem is known as a **data race**
Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization.
- We need mechanisms to control access to these shared resources in the face of concurrency.
  - So we can reason about how the program will operate.
- Our example was updating a shared bank account.
- Also necessary for synchronizing access to any shared data structure.
  - Buffers, queues, lists, hash tables, etc.
When Are Resources Shared?

• Local variables are **not shared** (private)
  ♦ Refer to data on the stack
  ♦ Each thread has its own stack
  ♦ **Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2**

• Global variables and static objects are **shared**
  ♦ Stored in the static data segment, accessible by any thread

• Dynamic objects and other heap objects are **shared**
  ♦ Allocated from heap with malloc/free or new/delete
How Interleaved Can It Get?

How contorted can the interleavings be?
• We'll assume that all instructions are atomic (either succeed completely or fail completely)
  ♦ e.g., reads and writes of words
• We'll assume that a context switch can occur at any time
  ♦ Examples may show code
  ♦ But actually at instruction granularity
  ♦ With multiprocessor, two insts can truly execute at the same time
• We'll assume that you can delay a thread as long as you like as long as it's not delayed forever
Goal

• Make a larger chunk of code non-interleavable

• Let’s first understand the goal before discussing how to achieve it
Mutual Exclusion

• We can use **mutual exclusion** to synchronize access to shared resources
  ✷ Only one thread can access shared resources at a time
  ✷ This allows us to have larger atomic blocks

• Code block that uses mutual exclusion to synchronize its execution is called a **critical section**
  ✷ Only one thread at a time can execute in the critical section
  ✷ All other threads are forced to wait on entry
  ✷ When a thread leaves a critical section, another can enter
  ✷ Example: bathrooms on airplanes

• What requirements would you place on a critical section?
Mutual Exclusion Using Critical Sections

A enters critical region

B attempts to enter critical region

B blocked

B enters critical region

A leaves critical region

B leaves critical region

Time
Critical Section Goals

1) Mutual exclusion
   ♦ If one thread is in the critical section, then no other is

2) Progress
   ♦ If some thread T is not in the critical section, then T cannot
     prevent some other thread S from entering the critical section
   ♦ A thread in the critical section will eventually leave it

3) Bounded waiting (no starvation)
   ♦ If some thread T is waiting on the critical section, then T will
     eventually enter the critical section

4) Performance
   ♦ The overhead of entering and exiting the critical section is
     small with respect to the work being done within it
About Goals

Also expressed as three properties:

• **Safety property**: nothing bad happens
  ♦ Mutual exclusion

• **Liveness property**: something good happens
  ♦ Progress, Bounded Waiting

• **Performance property**
  ♦ Performance

• Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!). Performance is nice to have but won't affect correctness.
Mechanisms For Building Critical Sections

- **Locks**
  - Primitive, minimal semantics, used to build others

- **Semaphores**
  - Basic, easy to get the hang of, but harder to program with

- **Monitors / Conditional Variables**
  - High-level, requires language support, operations implicit

- **Messages**
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
“Too Much Milk” Problem

Roommate A

Look in fridge: out of milk
Leave for Ralphs
Arrive at Ralphs
Buy milk
Arrive home

Roommate B

Look in fridge: out of milk
Leave for Ralphs
Arrive at Ralphs
Buy milk
Arrive home

• 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 *test* (look for other person & milk) and *set* (leave note) at the same time.
Another Possible Solution?

Thread A
leave noteA
if (no NoteB) {
  if (no Milk) {
    buy milk
  }
}
remove noteA

Thread B
leave noteB
if (no NoteA) {
  if (no Milk) {
    buy milk
  }
}
remove noteB
“too much milk” Yet Another Possible Solution?

Thread A

```java
leave noteA
while (noteB)
    do nothing;
if (no Milk)
    buy milk;
remove noteA
```

Thread B

```java
leave noteB
if (no NoteA) {
    if (no Milk) {
        buy milk
    }
}
remove noteB
```

- Safe to buy
- If the other buys, quit
- Things we dislike this solution?
Remarks

• The last solution works, but
  ♦ life is too complicated
  ♦ A’s code is different from B’s
  ♦ busy waiting is a waste

• What we want is:

```c
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;
}

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?
“Too much milk” problem with locks

```java
acquire(lock);
if (no Milk)
    buy milk;
release(lock);
}
```

• What is the problem with this solution?
Deep thinking

• How can we separate “checking” from “buying milk” and only lock “checking”?

```plaintext
local_flag = FALSE;

Acquire(lock);
if (no note && noMilk){
    leave note;
    local_flag = true; }
Release(lock);

If (local_flag) buy milk;

If (local_flag){
    local_flag = FALSE;
    remove note;}
```
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);  // A context switch can occur here, causing a race condition
    lock.held = 1;
}

void release (lock) {
    lock.held = 0;
}
```
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)
  - E.g. typing -> keyboard interrupt -> handler -> kernel -> user process
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)
• In a “real” system, this is only available to the kernel
  ♦ Why?
• 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!

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

- When executing test-and-set on “flag”
  - What is the 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!)
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a uniprocessor), unless
    - Lock holder calls yield or sleep (voluntary), or
    - Involuntary context switch
Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- 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 and disabling interrupts 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
[lec 3] Process State Transition

- **Ready**
  - Scheduler dispatch
  - Resource becomes available
  - Create a process

- **Running**
  - Wait for resource
  - Terminate

- **Blocked**
  -
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;
}
```

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

    queue Q;
}
```

```c
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;
}
```
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) {
    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;
}

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\rightarrow guard)) ;
    if (lock\rightarrow held == 0) {
        lock\rightarrow held = 1;
        lock\rightarrow guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock\rightarrow Q;
    lock\rightarrow guard = 0;
    go to sleep;
    enable interrupts;
}

void release (lock) {
    disable interrupts;
    while (test-and-set(lock\rightarrow guard)) ;
    if (lock\rightarrow Q is empty)
        lock\rightarrow held = 0;
    if (lock\rightarrow Q is not empty)
        move a waiting thread to the ready queue;
    lock\rightarrow 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!