

## Today's Class

- Last time (Tues Oct 16)
  - Midterm
- This time
  - Synchronization (chapt 7)
- Next time (Tues Oct 23)
  - Midterm back
  - Discussion of Assignment #3 (scheduling)
  - Finish synchronization
  - Deadlock

CS414: Operating Systems

## From last time

- Atomic operation vs. mutual exclusion?
- Mutual exclusion vs. critical section?
- Critical section vs. atomic operation?

CS414: Operating Systems

## One Approach: Disabling Interrupts

shared float balance; // variable shared between threads

Code for "Credit" Thread

```
int amount; // local variable
while (1) {
    amount = getNewAmount();
    disableInterrupts();
    balance = balance + amount;
    enableInterrupts();
}
```

Code for "Debit" Thread

```
int amount; // local variable
while (1) {
    amount = getNewAmount();
    disableInterrupts();
    balance = balance - amount;
    enableInterrupts();
}
```

Evaluation of this approach?

CS414: Operating Systems

## Another Approach: Software "Solution"

shared boolean lock=FALSE; // variable shared between threads  
shared float balance; // variable shared between threads

Code for "Credit" Thread

```
int amount; // local variable
while (1) {
    amount = getNewAmount();
    // Acquire the lock
    while (lock) {} ;
    lock = TRUE;
    // Execute Critical Section
    balance = balance + amount;
    // Release the lock
    lock = FALSE;
}
```

Code for "Debit" Thread

```
int amount; // local variable
while (1) {
    amount = getNewAmount();
    // Acquire the lock
    while (lock) {} ;
    lock = TRUE;
    // Execute Critical Section
    balance = balance - amount;
    // Release the lock
    lock = FALSE;
}
```

Evaluation of this approach?

CS414: Operating Systems

## Potential Solution: Lock Manipulation as a Critical Section

shared boolean lock=FALSE; // variable shared between threads  
shared float balance; // variable shared between threads

```
acquire (boolean lock)
{
    disableInterrupts();
    while (lock) {
        enableInterrupts();
        disableInterrupts();
    }
    lock = TRUE;
    enableInterrupts();
}
```

```
release (boolean lock)
{
    disableInterrupts();
    lock = false;
    enableInterrupts();
}
```

Code for "Credit" Thread

```
int amount; // local variable
while (1) {
    amount = getNewAmount();
    acquire(lock);
    balance = balance + amount;
    release(lock);
}
```

Code for "Debit" Thread

```
int amount; // local variable
while (1) {
    amount = getNewAmount();
    acquire(lock);
    balance = balance - amount;
    release(lock);
}
```

•Evaluation of this approach?

CS414: Operating Systems

## Big Picture at this Point

- Access to shared variables can lead to race conditions (we need mutual exclusion)
- We have been looking at approaches that either [a] don't ask the OS for any help or [b] only relies on OS to enable/disable interrupts on request
  - Mutual exclusion can be ensured by disabling/enabling interrupts (forces critical section to be atomic): NOT THE BEST APPROACH
  - Mutual exclusion can be attempted through "normal, user-level software" (tends to be problematic and subject to race conditions, although it is feasible): NOT THE BEST APPROACH
  - Locks can be implemented at the user-level by disabling/enabling interrupts, but this is still NOT THE BEST APPROACH

CS414: Operating Systems

## Additional Points

- **“Deadlock”** can occur when two threads attempt to acquire two locks in **different order** (we’ll study deadlock later)
- fork, join, and quit(exit) can be used to synchronize activities; however, these operations are **very** time-consuming
- We can generalize synchronization requirements into two cases: [1] **mutual exclusion**, and [2] **scheduling constraints**

```

shared double x, y;

proc A {
  while (TRUE) {
    < compute section A1>
    write (x);
    < compute section A2>
    read (y);
  }
}

proc B {
  while (TRUE) {
    read (x);
    < compute section B1>
    write (y);
    < compute section B2>
  }
}
    
```

**Do these two threads require mutual exclusion?**

CS414: Operating Systems

## More precisely: Criteria for Acceptable Solutions to the Critical Section Problem

- Only one thread at a time in its criteria section (**mutual exclusion**)
- Once a thread attempts to enter its critical section, the decision regarding which thread will enter cannot be postponed indefinitely (**progress**)
  - “System perspective”
- Once a thread requests entry into its critical section, there must be a bound on the number of times other threads can enter their critical section before this thread is allowed to enter (**starvation – aka “bounded waiting”**)
  - “User perspective”

CS414: Operating Systems

## Semaphores

- **Semaphores**: special type of variable that supports **two atomic operations**; elegant solution to synchronization problem (Dijkstra, 1965)
- **Assumptions**: reading/writing a shared memory cell is atomic; threads are of the same priority; relative speed of processes unknown; threads are sequential and cyclic
- **Semaphore Atomic Operations** (discussed here in the context of mutual exclusion)
 

```

S->wait(); // wait until semaphore S is “free” [aka S->P();]
{ critical section code }
S->signal(); // signal to one other process that semaphore S is “free”
// [aka S->V();]
            
```
- If a thread executes **S->wait()** :
  - semaphore S is “free”, it **continues executing**.
  - If semaphore S is “not free”, process either busy-waits or blocks (depends on the implementation of semaphore in OS)
- A **S->signal()** allows **one thread** waiting on semaphore S to now be allowed to progress (if one is waiting)

CS414: Operating Systems

## Conceptual Implementation of Signal (V) and Wait (P)

```

class Semaphore {
public:
  Semaphore();
  void wait();
  void signal();
private:
  int value;
  “queue of processes” Q;
}

Semaphore::Semaphore() {
  value = 1; // default; initial value
  Q = empty; // .. depends on problem
}

Semaphore::wait() {
  value = value - 1;
  if (value < 0) {
    add this process to Q
    block
  }
}

Semaphore::signal() {
  value = value + 1;
  if (value <= 0) {
    remove P from Q
    wakeup(P)
  }
}
    
```

- Signal and Wait **must be atomic!**
- Two semaphore types: **Binary** (either 0 or 1) or **Counting**

CS414: Operating Systems

## Using Semaphores

- **For mutual exclusion**
  - Semaphore initial value is 1
  - S->wait() called **before** critical section
  - S->signal() called **after** leaving the critical section
- **Scheduling constraints**
  - General situation in which threads must wait for some event. Often the initial value of the semaphore is 0
  - **Example**: You can (roughly) implement *join* with semaphores
 

```

S->value = 0 // semaphore initialization
Thread::join
  S->wait();
Thread::finish
  S->signal();
                    
```

CS414: Operating Systems

## Readers/Writers Problem

- We’ve seen that can generalize synchronization requirements into two cases: [1] **mutual exclusion**, and [2] **scheduling constraints**
- An object is **shared among several threads**, some that only read it and some that write it
- We can allow **multiple readers** at a time, but only **one writer** at a time
- How do we **control access** to the object to permit this protocol?
  - We’ll do it with semaphores....

CS414: Operating Systems

## Readers/Writers Problem

```

class ReadWrite (
public:
    ReadWrite();
    void Read();
    void Write();
private:
    Semaphore mutex;
    int readers;
    Semaphore wrt;
}
ReadWrite::ReadWrite() {
    readers = 0;
    mutex = 1;
}

ReadWrite::Write() {
    wrt->wait();
    << perform write >>
    wrt->signal();
}

ReadWrite::Read() {
    mutex->wait();
    readers += 1;
    if (readers == 1)
        wrt->wait();
    mutex->signal();
    << perform read >>
    mutex->wait();
    readers -= 1;
    if (readers == 0)
        wrt->signal();
    mutex->signal();
}
    
```

CS414: Operating Systems

## How do we show that this is correct?

- can 1 reader be reading while a writer is writing? (no)
- can multiple readers be reading? (yes)
- can multiple writers be writing? (no)
- can all threads make progress? (yes)
  - depends on implementation of semaphores
- will any thread starve? (no)
  - depends on implementation

CS414: Operating Systems

## Implementing Semaphores

- For a uniprocessor, we can disable interrupts for primitives like semaphores, 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.

```

class Semaphore {
    int value = 1; // depends on usage
}

Semaphore::P() {
    disable interrupts;
    while (value == 0) {
        enable interrupts;
        disable interrupts;
    }
    value--;
    enable interrupts;
}

Semaphore::V() {
    disable interrupts;
    value++;
    enable interrupts;
}
    
```

CS414: Operating Systems

## 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 by simply adding a new instruction on a uniprocessor
- On a multiprocessor, the processor issuing the instruction must also be able to invalidate any value the other processors may have in their cache—the multiprocessor must support some kind of cache coherence
- Example read-modify-write instructions:
  - Test & Set (most architectures) read a value, write '1' back to memory
    - Pentium: "BTS"
  - Exchange (x86) swaps value between register and memory
  - Compare and Swap (680x0) read value, if value matches register value *r1*, exchange register *r2* and value
  - Load Linked and Conditional Store (R6000, Alpha)

CS414: Operating Systems

## Implementing Semaphores with Test and (re)Set

```

class Semaphore {
    int value = 1; // depends on problem
}

Semaphore::P() {
    while (test&reset(value) == 0) ;
}

Semaphore::V() {
    value = 1;
}
    
```

**Semaphore "free"**  
test&reset reads 1, sets value to 0, and returns 1. The test in the 'while' fails and P is complete

**Semaphore "not free"**  
test&reset reads 0, sets value to 0 (no change), and returns 0. The test in the 'while' succeeds (keeps looping) until a V executes

**Does this suffer from busy-waiting? (This is a spinlock).**

CS414: Operating Systems

## Recall: Conceptual Implementation of Signal (V) and Wait (P)

```

class Semaphore {
public:
    Semaphore();
    void wait();
    void signal();
private:
    int value;
    "queue of processes" Q;
}

Semaphore::Semaphore() {
    value = 1; // default; initial value
    Q = empty; // ... depends on problem
}

Semaphore::wait() {
    value = value - 1;
    if (value < 0) {
        add this process to Q
        block
    }
}

Semaphore::signal() {
    value = value + 1;
    if (value <= 0) {
        remove P from Q
        wakeup(P)
    }
}
    
```

- Recall: Signal and Wait must be atomic! How do we do it?

CS414: Operating Systems