Announcements

- Homework #2 out
- Homework #1 almost graded...
- Discussion session on Wednesday will entertain questions on the project, and will talk more about how to program with threads.
My solution to HW1.6

```c
#include <signal.h>
#include <stdio.h>

int counter = 0;

void CatchC(int sig) {
    if (++counter == 1) printf("Strike one... \n");
    else if (counter == 2) printf("Strike two... \n");
    else { printf("Out!\n"); exit(0); }
}

int main(int argc, char** argv) {
    signal(SIGINT, CatchC);
    while (1) { }
}
```
Not-so-elegant solution

```c
int counter = 0;

void CatchC(int sigint) {
    signal(SIGINT, SIG_IGN);
    if (counter == 1) signal(SIGINT, SIG_DFL);
    else {
        counter++;
        signal(SIGINT, CatchC);
    }
}

int main(void) {
    signal(SIGINT, CatchC);
    while (1) {}
}```
A make sure it's set! solution

int counter = 0;

void CatchC(int sigint) {
    signal(SIGINT, SIG_IGN);
    if (++counter == 3) exit(0);
}

int main(void) {
    while (1) {signal(SIGINT, CatchC); }
}

A *let's try this!* solution

```c
static jmp_buf env;
int counter = 0;

void CatchC(int sigint) {
    counter++;
    longjmp(env, sigint);
}

int main(void) {
    signal(SIGINT, CatchC);
    setjmp(env);
    while (counter < 3) { }
}
```
... remember the first lecture

On project 1, we're looking for simple and elegant solutions!

It's been a design goal ever since there have been systems...
Review from last lecture

- Safety/Liveness
- Peterson's Algorithm
About Requirements

There are three kinds of requirements that we'll use:

- **Safety property:** nothing bad happens
  - Mutex

- **Liveness property:** something good happens
  - Progress, Bounded waiting

- **Performance requirement**
  - Performance

- **Properties hold for each run,** while performance depends on all the runs
  - Rule of thumb: when designing a concurrent algorithm, worry about safety first (but don't forget liveness!).
Mutex with Atomic R/W: Peterson's Algorithm

```java
int turn = 1;
boolean try1 = false, try2 = false;

while (true) {
    {¬ try1 ∧ (turn == 1 ∨ turn == 2) }

1 try1 = true;
    { try1 ∧ (turn == 1 ∨ turn == 2) }

2 turn = 2;
    { try1 ∧ (turn == 1 ∨ turn == 2) }

3 while (try2 && turn != 1) ;
    { try1 ∧ (turn == 1 ∨ ¬ try2 ∨
        (try2 ∧ (yellow at 6 or at 7))) }
    critical section

4 try1 = false;
    {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
    outside of critical section
}

```

```java
while (true) {
    {¬ try2 ∧ (turn == 1 ∨ turn == 2) }

5 try2 = true;
    { try2 ∧ (turn == 1 ∨ turn == 2) }

6 turn = 1;
    { try2 ∧ (turn == 1 ∨ turn == 2) }

7 while (try1 && turn != 2) ;
    { try2 ∧ (turn == 2 ∨ ¬ try1 ∨
        (try1 ∧ (blue at 2 or at 3))) }
    critical section

8 try2 = false;
    {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
    outside of critical section
}
```

(blue at 4) ∧ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7))
    ∧ (yellow at 8) ∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (blur at 2 or at 3)))
... ⇒ (turn == 1 ∧ turn == 2)
And now...

- We return to your scheduled lecture
Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- Look at two common high-level mechanisms
  - **Semaphores**: binary (mutex) and counting
  - **Monitors**: mutexes and condition variables
- Use them to solve common synchronization problems
Semaphores

- Semaphores are instances of an abstract data type that provides mutual exclusion to critical sections
  - Block waiters with interrupts enabled within the critical section
  - Described by Dijkstra in T.H.E. operating system way back in 1968

- Semaphores can also be used as atomic counters
  - More on this later
Semaphores

- A semaphores has an integer value.
- There are exactly two supported operations:
  - `wait(semaphore)`: decrement, block until semaphore is open
    - Traditionally written `P()`, after the Dutch word for test
    - Some authors use `down()`
  - `signal(semaphore)`: increment, allow another thread to enter
    - Traditionally written `V()` after the Dutch word for increment
    - Some authors use `up()`
  - That's it! No other operations - not even just reading its value - exist.
- Semaphores satisfy one safety property: a semaphore's value is never negative.
Blocking in Semaphores

One can think of semaphores as *implementing a gate*.

- Associated with each semaphore is a queue of waiting processes
- When `wait()` is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then `signal()` opens the semaphore:
  - If threads are waiting on the queue, one thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - In other words, `signal()` has “history” (c.f. condition vars later)
    - This “history” is a counter - the integer value of the semaphore
Two Types of Semaphores

- Semaphores come in two types
  - **Mutex** semaphore (or **binary** semaphore)
    - Represents single access to a resource
    - Guarantees mutual exclusion to a critical section
  - **Counting** semaphore (or **general** semaphore)
    - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
    - Multiple threads can pass the semaphore
    - Number of threads determined by the semaphore “count”
      » mutex has count = 1, counting has count = N
Mutex, Counting Semaphores

- Let \( \langle \ldots \rangle \) mean *executed atomically* (i.e., without interruption or preemption, e.g., protected by a lock).
- Let \( \text{when } P \text{ V} \) mean *execute V only when P holds*.

**Mutex semaphore S**
- S has value of 0 or 1
- \( \text{wait}(S) \) is \( \langle \text{when } (S == 1) S = 0 \rangle \)
- \( \text{signal}(S) \) is \( \langle S = 1 \rangle \)

**Counting semaphore S**
- S has nonnegative integer value
- \( \text{wait}(S) \) is \( \langle \text{when } (S > 0) S = S - 1 \rangle \)
- \( \text{signal}(S) \) is \( \langle S = S + 1 \rangle \)
Using Semaphores

- Similar to our locks but semantics are different

```c
semaphore S = 1;

withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

It is undefined which thread runs after a signal

Threads block critical section

```
wait(S);
balance = get_balance(account);
balance = balance - amount;
wait(S);
wait(S);
put_balance(account, balance);
signal(S);
...
signal(S);
...
signal(S);
```
Semaphores in P1

- Conditions can be used as counting semaphores.
  
  \[ \text{cond } c \] is a counting semaphore with initial value 0.
  
  \[ \text{wait}(c) \] is wait(c).
  
  \[ \text{signal}(c, \text{NULL}, \text{TRUE}) \] is signal(c).

- ... but they also can be used to pass values from signaling to waiting threads.

- You implement a condition as two queues: one queue of threads blocked on c and one queue as signal values waiting to be delivered.
  
  - Will there ever be times when both queues are not empty?
Using Semaphores

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
- Now we’ll use semaphores to look at more interesting examples
  - Readers/Writers
  - Bounded Buffers
Readers/Writers Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - At any time we can allow multiple readers but only one writer
    - Let \( #r \) be the number of readers, \( #w \) be the number of writers
    - Safety: \( (#r \geq 0) \land (0 \leq #w \leq 1) \land ((#r > 0) \Rightarrow (#w = 0)) \)

- How can we use semaphores to control access to the object to implement this protocol?

- Use three variables
  - int \texttt{readcount} – number of threads reading object
  - Mutex semaphore \texttt{mutex} – control access to readcount
  - Mutex semaphore \texttt{w_or_r} – exclusive writing or reading
Readers/Writers

// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex); // unlock readcount
    Read;
    wait(mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex); // unlock readcount
}
Readers/Writers Notes

- \texttt{w\_or\_r} provides mutex between readers and writers
  - writer wait/signal
  - reader wait/signal when \texttt{readcount} goes from 0 to 1 or from 1 to 0.
- If a writer is writing, where will readers be waiting?
- Once a writer exits, all readers can start reading
  - Which reader gets to read first?
  - Is it guaranteed that all readers will eventually start reading?
- If readers and writers are waiting, and a writer exits, \texttt{who goes first}?
- Why do readers use \texttt{mutex}?
- Why don't writers use \texttt{mutex}?
- What if \texttt{signal(mutex)} is before if (\texttt{readcount} == 1)?
Bounded Buffer

- Problem: There is a set of resource buffers shared by producer and consumer threads
  - **Producer** inserts resources into the buffer set
    » Output, disk blocks, memory pages, processes, etc.
  - **Consumer** removes resources from the buffer set
    » Whatever is generated by the producer

- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff

- Safety:
  - Sequence of consumed values is prefix of sequence of produced values
  - If $nc$ is number consumed, $np$ number produced, and $N$ the size of the buffer, then $0 \leq np - nc \leq N$
Bounded Buffer (2)

- $0 \leq np - nc \leq N$
  - $np - nc \geq 0$ (number of full buffers is nonnegative)
  - $N - (np - nc) \geq 0$ (number of empty buffers is nonnegative)

- Use three semaphores:
  - empty – count of empty buffers
    » Counting semaphore
  - full – count of full buffers
    » Counting semaphore
  - mutex – mutual exclusion to shared set of buffers
    » Mutex semaphore
Bounded Buffer (3)

Mutex semaphore mutex = 1;  // mutual exclusion to shared set of buffers
Counting semaphore empty = N;  // count of empty buffers (all empty to start)
Counting semaphore full = 0;    // count of full buffers (none full to start)

producer {
    while (1) {
        Produce new resource;
        wait(empty);  // wait for empty buffer
        wait(mutex);  // lock buffer list
        Add resource to an empty buffer;
        signal(mutex);  // unlock buffer list
        signal(full);  // note a full buffer
    }
}

c consumer {
    while (1) {
        wait(full);  // wait for a full buffer
        wait(mutex);  // lock buffer list
        Remove resource from a full buffer;
        signal(mutex);  // unlock buffer list
        signal(empty);  // note an empty buffer
        Consume resource;
    }
}

Bounded Buffer (4)

- Why need the mutex at all?
- Where are the critical sections?
- What has to hold for deadlock to occur?
  - empty = 0 and full = 0
  - $N - (np - nc) = 0$ and $np - nc = 0$
  - $N = 0$
- What happens if operations on mutex and full/empty are switched around?
  - The pattern of signal/wait on full/empty is a common construct often called an \textit{interlock}
- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
  - The homework has another one.
Some Questions about Semaphores

- Does it matter **which thread is unblocked** by a signal operation?
  - Hint: consider the following three processes sharing a semaphore **mutex** that is initially 1:

```c
while (1) {
    wait(mutex);
    // in critical section
    signal(mutex);
}
```

- Are there any problems that **can be solved** with counting semaphores that **can't be solved** with mutex semaphores?
Counting Semaphores with Mutex Semaphores

```c
struct counting semaphore {
    int value = ...;          // initial value
    mutex semaphore queue = (value == 0 ? 0 : 1);
    mutex semaphore mutex = 1;
};

void wait(counting semaphore s) {
    wait(s.queue);
    wait(s.mutex);
    if (--s.value > 0) signal(s.queue);
    notify(s.mutex);
}

void signal(counting semaphore s) {
    wait(s.mutex);
    if (s.value++ == 0) signal(s.queue);
    notify(s.mutex);
}
```
Counting Semaphores with Mutex Semaphores

- How many threads can be waiting on mutex at any time?
- When is queue equal to zero?
- Can two signals to either mutex semaphore occur without an intermediate wait?

- How would you implement mutex semaphores with counting semaphores?
  - Can't do something like
    ```
    signal(S);
    if (S > 1) S = 1;
    ```
A buggy counting semaphore implementation

```c
struct counting semaphore {
    integer count = 0;
    integer s = ...;
    mutex semaphore mutex = 1, block = 0;
}

void wait(counting semaphore s) {
    wait(mutex);
    if (s == 0) {
        count = count + 1;
        notify(mutex);
        wait(block);
    } else {
        s = s - 1;
        notify(mutex);
    }
}

void notify(counting semaphore s) {
    wait(mutex);
    if (count == 0) {
        s = s + 1;
        notify(mutex);
    } else {
        count = count - 1;
        notify(block);
        notify(mutex);
    }
}
```
Semaphore Summary

- Semaphores can be used to solve any of the traditional synchronization problems.
- However, they have some drawbacks:
  - They are essentially shared global variables:
    - Can potentially be accessed anywhere in the program.
  - No connection between the semaphore and the data being controlled by the semaphore.
  - Used both for critical sections (mutual exclusion) and coordination (scheduling).
  - No control or guarantee of proper usage.
  - Sometimes hard to use and prone to bugs.

Another approach: Use programming language support.
Next Time

- Review 6.7 - 6.10.