Slide 1

## Administrivia

- Reminder: Homework 3 written problems due today; programming problem due Monday.
- Reminder: Midterm next Wednesday. Review sheet on the course Web site.
- Homework 2 written problems graded. Grades were pretty disappointing overall, mostly because of the problems involving IPC (the one about semaphores and the last one). I'm speculating about an extra-credit assignment on this topic so you can recoup lost points. (I'll ask about interest in the minute essay.)

Slide 2

## Minute Essay From Last Lecture

- (Almost everyone got it right. Yay!)

Slide 3

## Multithreading in C

- Many ways to do multithreading in C and C++. Until relatively recently, however, there was nothing about threads in the standard library for either language.
- Common (at least in UNIXworld) to use POSIX threads; widely available though not strictly standard-C. Somewhat cumbersome to use but workable.
- Support for threads added to C and C++ with C11, C++11. Both somewhat low-level (compared to higher-level ways of expressing concurrency) and based on POSIX threads. C++11 support is somewhat nicer, though a little hard to get started with. I admit I'm not sure about C11; support in `gcc` is recent.
- OpenMP is a widely-supported set of extensions to C and C++ (and Fortran!) that support multithreading somewhat more nicely.

Slide 4

## POSIX Threads — Basics

- ("Hello world" program on sample programs page.)
- Threads represented by opaque data type `pthread_t`.
- `pthread_create()` creates and starts a thread.
- `pthread_join` waits for a thread to finish.
- How do you say what code the thread is supposed to run, and how do you pass data to it? Next slide . . .

Slide 5

## POSIX Threads Basics, Continued

- How do you say what code a thread is supposed to run? via a function pointer to a function that takes one argument (a `void *`) and returns a `void *`. (The function isn't supposed to actually return anything; instead it's supposed to call `pthread_exit`.)
- How do you pass data to threads? via the single parameter to the function. But it can point to anything — an `int` in the example, or a `struct` to pass multiple values. (Yes, it's ugly! Well, it's C. In toy/demo programs it's not uncommon to just use global variables.)

Slide 6

## POSIX Threads Library Functions

- Mutual exclusion locks: Data type `pthread_mutex_t`; functions `pthread_mutex_init()`, `pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`.
- Condition variables: Data type `pthread_cond_t`; functions `pthread_cond_init()`, `pthread_cond_destroy()`, `pthread_cond_wait()`, `pthread_cond_signal()`. Also `pthread_cond_broadcast()`.
- (So you can implement monitor-based solutions to problems, though you have to put in the code yourself to ensure that only one procedure at a time runs.)
- (Examples shortly.)

Slide 7

## Other Semi-Standard IPC Library Functions

- Semaphores: Data type `sem_t`; functions `sem_init()`, `sem_destroy()`, `sem_wait()`, `sem_post()`.
- (Examples shortly.)

Slide 8

## Bounded Buffer Problem Revisited

- We didn't talk about using the invariants idea to reason about solutions to this problem. Could we? I'm not sure it makes sense to do it formally, but . . .

Slide 9

## Bounded Buffer Semaphore Solution — Informal Invariant

- The value of semaphore `empty` represents the number of empty slots in the buffer.
- The value of semaphore `full` represents the number of full slots in the buffer.
- Semaphore `mutex` is one if some process is accessing the shared buffer, zero otherwise.
- If you look at the code, all of this is true initially, remains true throughout execution, and ensures that the solution works as intended.

Slide 10

## Bounded Buffer Monitor Solution — Informal Invariant

- Names of condition variables were badly chosen. I just revised so that names represent what has to be true to not wait.
- Variable `count` represents the number of slots in use.
- Condition variable `not_full` represents producers suspended because the buffer is full.
- Condition variable `not_empty` represents consumers suspended because the buffer is empty.
- If you look at the code, all of this is true initially, remains true throughout execution, and ensures that the solution works as intended.

Slide 11

## Bounded Buffer Problem — Implementations

- Sample programs page has semaphore- and monitor-based implementations of bounded buffer problem.
- A possible complication is that I didn't want to just let the simulation run "forever"; instead I chose to specify a total number of items to produce/consume. So I need additional counters shared among threads, which means more synchronization.
- (Look at code.)

Slide 12

## Another Classical IPC Problem — Readers/Writers

- First proposed by Courtois et al in 1971.
- Problem posits a file (a database maybe) to be shared among processes. It's safe to have any number of processes read from the file as long as none of them is changing it. To write to the file, however, a process needs exclusive access.
- Textbook shows a solution using semaphores.
- I found solutions using a monitor in various online sources.

Slide 13

## Readers/Writers using Semaphores

- Shared variables:

```
semaphore reader_lock(1);
semaphore writer_lock(1);
int reader_count = 0;
```

- Routines:

```
read() {
    down(&reader_lock);
    reader_count += 1;
    if (reader_count == 1) {
        down(&writer_lock);
    }
    up(&reader_lock);
    /* do actual read */
    down(&reader_lock);
    reader_count -= 1;
    if (reader_count == 0) {
        up(&writer_lock);
    }
    up(&reader_lock);
}
```

```
write() {
    down(&writer_lock);
    /* do actual write */
    up(&writer_lock);
}
```

Slide 14

## Readers/Writers using Semaphores — Informal Invariants

- Variable `reader_count` represents the number of readers.
- The value of semaphore `reader_lock` is 0 when the number of readers is changing, 1 otherwise.
- The value of semaphore `writer_lock` is 0 when there is a writer, 1 otherwise.

Slide 15

## Readers/Writers using a Monitor

- Shared variables:

```
int readers = 0;
int writers = 0;
int readers_waiting = 0;
condition can_read;
condition can_write;
```

- (Continued on next slide.)

Slide 16

## Readers/Writers using a Monitor, Continued

- Routines:

```
begin_read() {
    if (writers == 1) {
        readers_waiting += 1;
        wait(can_read);
        readers_waiting -= 1;
    }
    readers += 1;
    signal(can_read);
}
end_read() {
    readers -= 1;
    if (readers == 0)
        signal(can_write);
}
```

```
begin_write() {
    if ((writers == 1) || (readers > 0)) {
        wait(can_write);
    }
    writers = 1;
}
end_write() {
    writers = 0;
    if (readers_waiting > 0)
        signal(can_read);
    else
        signal(can_write);
}
```

Slide 17

## Readers/Writers using a Monitor — Informal Invariants

- Variable `readers` represents the number of readers reading.
- Variable `writers` is 1 if a writer is writing, 0 otherwise.
- Variable `readers_waiting` represents the number of readers waiting.
- Condition variable `can_read` represents readers waiting because a writer is writing.
- Condition variable `can_write` represents writers waiting because at least one reader is reading.

Slide 18

## Readers/Writers Problem — Implementations

- Sample programs page has semaphore- and monitor-based implementations of readers/writers buffer problem.

Slide 19

## Readers/Writers Problem, Continued

- A weakness of these solutions is that they could block writers indefinitely.
- Fixing that — "priority readers/writers" problem.

Slide 20

## Minute Essay

- If you lost a lot of points on one or more of the Homework 2 written problems, what do you think went wrong? (Did you not understand the problem(s), had we not done enough relevant examples in class, . . . ?)
- If I do make up an extra-credit assignment on IPC, how likely would you be to try it? I was thinking a due date of a week from Friday so I can include any points in your midterm average, but if you don't care about that you could turn it in later.