Slide 1

Administrivia

- A homework on processes/synchronization is coming soon.
- A few words about the computers in front of you:
  - Checking your e-mail when you first get here is okay.
  - Taking notes online is okay.
  - Surfing the Web or playing games during lecture is not okay.
  - Remember that I can lock all screens . . .

Slide 2

Minute Essay From Last Lecture

- Which of the synchronization mechanisms we've talked about (semaphores, monitors, message passing) do you think you would prefer to use? Why?

  About equal numbers for semaphores and message passing, fewer for monitors. Seemed to depend in part on what people had experience with.

Slide 3

Synchronization Mechanisms — Recap

- Low-level ways of synchronizing — using shared variables only, using TSL instruction.
- Higher-level mechanisms — semaphores, monitors, message passing. Often built using something lower-level.

Slide 4

Classical IPC Problems

- Literature (and textbooks) on operating systems talk about "classical problems" of interprocess communication.
- Idea — each is an abstract/simplified version of problems o/s designers actually need to solve. Also a good way to compare ease-of-use of various synchronization mechanisms.
- Examples so far — mutual exclusion, bounded buffer.
- Other examples sometimes described in silly anthropomorphic terms, but underlying problem is a simplified version of something "real".

Slide 5

Dining Philosophers Problem

- Scenario (originally proposed by Dijkstra, 1972):
  - Five philosophers sitting around a table, each alternating between thinking and eating.
  - Between every pair of philosophers, a fork; philosopher must have two forks to eat.
  - So, neighbors can't eat at the same time, but non-neighbors can.
- Why is this interesting or important? It's a simple example of something more complex than mutual exclusion — multiple shared resources (forks), processes (philosophers) must obtain two resources together. (Why five? smallest number that's "interesting".)

Slide 6

Dining Philosophers — Naive Solution

- Naive approach — we have five mutual-exclusion problems to solve (one per fork), so just solve them.
- Does this work?

Slide 7

Dining Philosophers — Simple Solution

- Another approach — just use a solution to the mutual exclusion problem to let only one philosopher at a time eat.
- Does this work?

Slide 8

Dining Philosophers — Solution

- Another approach — use shared variables to track state of philosophers and semaphores to synchronize.
- I.e., variables are
  - Array of five `state` variables (`states[5]`), possible values `thinking`, `hungry`, `eating`. Initially all `thinking`.
  - Semaphore `mutex`, initial value 1, to enforce one-at-a-time access to `states`.
  - Array of five semaphores `self[5]`, initial values 0, to allow us to make philosophers wait.
- And then the code is somewhat complex . . .

Slide 9

Dining Philosophers — Code

- Shared variables as on previous slide.

Pseudocode for philosopher i:

```
while (true) {
    think();
    down(mutex);
    state[i] = hungry;
    test(i);
    up(mutex);
    down(self[i]);
    eat();
    down(mutex);
    state[i] = thinking;
    test(right(i));
    test(left(i));
    up(mutex);
}
```

Pseudocode for function:

```
void test(i)
{
    if ((state[left(i)] != eating) &&
            state[right(i) != eating) &&
            state[i] == hungry) {
        state[i] = eating;
        up(self[i]);
    }
}
```

Slide 10

Dining Philosophers — Solution Works?

- Could there be problems with access to shared `state` variables?
- Do we guarantee that neighbors don't eat at the same time?
- Do we allow non-neighbors to eat at the same time?
- Could we deadlock?
- Does a hungry philosopher always get to eat eventually?

Slide 11

Other Classical Problems

- Readers/writers.
- Sleeping barber.
- And others . . .
- Advice — if you ever have to solve problems like this "for real", read the literature . . .

Slide 12

Minute Essay

- Anything about processes or synchronization you want to hear more about? particularly unclear?