Slide 1

Administrivia

- Homework 2 due date/time: Written problems due at class time; not accepted late except by prior agreement. Normal rules for programming problems apply.
- (Review minute essay from last time. Is average turnaround time the same for all algorithms?)

Slide 2

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 3

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 4

Dining Philosophers — Naive Solution

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

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 5

Dining Philosophers — Dijkstra 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 6

Slide 7

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 8

Dining Philosophers — Dijkstra 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 9

Dining Philosophers — Chandy/Misra Solution

- Original solution allows for scenarios in which one philosopher "starves" because its neighbors alternate eating while it remains hungry.
- Briefly, we could improve this by maintaining a notion of "priority" between neighbors, and only allow a philosopher to eat if (1) neither neighbor is eating, *and* (2) it doesn't have a higher-priority neighbor that's hungry. After a philosopher eats, it lowers its priority relative to its neighbors.

Slide 10

Other Classical Problems

- Readers/writers (in textbook).
- Sleeping barber, drinking philosophers, . . .
- Advice — if you ever have to solve problems like this "for real", read the literature . . .

Minute Essay

- Any questions about material in chapter 2 (processes, threads, synchronization, scheduling)?

Slide 11