Slide 1

Administrivia

- Homework 2 to be on the Web soon; I will send mail.
- Next quiz a week from today.
- Midterm postponed to . . . October 26?? would allow us to finish chapter 2.

Slide 2

Minute Essay From Last Lecture

- Pretty much no one got it.
- Point is that if you never do more “down” operations than “up” operations, you never block, so what was the point . . . .

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? No — deadlock possible.

Slide 5

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? Well, it "works" w.r.t. meeting safety condition and no deadlock, but it's too restrictive.

Slide 6

Slide 7

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 8

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]);
    }
}
```

Dining Philosophers — Dijkstra Solution Works?

Slide 9

- 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?

Dining Philosophers — Chandy/Misra Solution

Slide 10

- 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.

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 . . .

Slide 11

Review — Processes and Context Switches

- Recall idea behind process abstraction — make every activity we want to manage a “process”, and run them “concurrently”.
- Apparent concurrency provided by interleaving. (Some) true concurrency provided by multiple cores/processors.
- To make this work — process table, ready/running/blocked states, context switches.
- Context switches triggered by interrupts — I/O, timer, system call, etc.
- On interrupts, interrupt handler processes interrupt, and then goes back to some process — but which one?

Slide 12

Slide 13

Which Process To Run Next?

- Deciding what process to run next — scheduler/dispatcher, using "scheduling algorithm".
- When to make scheduling decisions?
  - When a new process is created.
  - When a running process exits.
  - When a process becomes blocked (I/O, semaphore, etc.).
  - After an interrupt.
- One possible decision — "go back to interrupted process" (e.g., after I/O interrupt).

Slide 14

Scheduler Goals

- Importance of scheduler can vary; extremes are
  - Single-user system — often only one runnable process, complicated decision-making may not be necessary (though still might sometimes be a good idea).
  - Mainframe system — many runnable processes, queue of "batch" jobs waiting, "who's next?" an important question.
  - Servers / workstations somewhere in the middle.
- First step is to be clear on goals — want to make "good decisions", but what does that mean? Typical goals for any system:
  - Fairness — similar processes get similar service.
  - Policy enforcement — "important" processes get better service.
  - Balance — all parts of system (CPU, I/O devices) kept busy (assuming there is work for them).

Slide 15

Other goals depend on system type (more next time).

Slide 16

Scheduling Algorithms

- Many, many scheduling algorithms, ranging from simple to not-so-simple.
- Point of reviewing lots of them? notice how many ways there are to solve the same problem ("who should be next?"), strengths/weaknesses of each.

Slide 17

Minute Essay

- None really — sign in, unless questions?