Slide 1

Administrivia

- Reminder: Homework 2 written problems due Monday.
- For programming problems, no need to make the name of your program unique — my semi-automated grading makes a directory for each student, so no problem with name collisions. Just sayin'!

Slide 2

Minute Essay From Last Lecture

- (Review "answer" slide.)
- Some people got the point, others didn't.

  Might be worth mentioning that of course(?) at any point in the program you can't have *completed* more `down`s than the number of completed `up`s, plus the semaphore's initial value, but if there's no time when you called `down` on a semaphore with value 0 then maybe you didn't need one? (As with so many things, too much attention to details takes some of the fun out of the alleged joke?)
- "Alleged" — because no one was amused. Ah well. (And why do we groan at puns? I do too, and I *like* them!)

Slide 3

Homework 1 Programming Problem

- Most people did well. Almost everyone included the needed `#include`s. Yay! in previous years many did not.
- Most common error was apparently not completely understanding what happens if `execve` does not succeed: If that happens, now you have two processes running your shell program!

Slide 4

O/S Versus Application Programs — Recap/Review

- Should seem reasonable to make distinction between what O/S can do and what application programs can do.
- But how to enforce that? i.e., how to make it as difficult as possible for buggy or malicious application programs to do what they shouldn't?

  Can this problem be solved completely by clever programming? Consider that most current systems can be asked to load and execute machine-level application code . . .

O/S Versus Application Programs, Continued

- If you don't allow that — how do you decide what's okay?
- If you do allow loading and executing arbitrary code, then some sort of hardware mechanism for limiting what it can do seems like the only way. This is the problem "dual-mode operation" is intended to solve.

Slide 5

O/S Versus Application Programs, Continued

- At hardware level, then, need to keep track of which mode we're in and use that information to allow/disallow certain operations (and maybe memory accesses — though that could be a separate problem/solution).
- To do this efficiently — single bit in a register somewhere, probably a special-purpose one, checked by "privileged" instructions.
- What happens if unprivileged program tries . . . ? Hardware version of exception — interrupt.
- How to set this bit? privileged operation, or no?

Slide 6

Slide 7

O/S Versus Application Programs, Continued

- Setting the "privileged okay" bit has to be privileged, no? or what's the point? But then how do you ever get from unprivileged to privileged?
- A solution: Include instruction to generate interrupt, and have hardware, on interrupt, transfer control to a fixed location *and* set the "privileged" bit. If what's at the fixed location is O/S code, then it can do more checking (e.g., passwords). (This is what's behind "system calls".)
- Now, if what's at that fixed location is not O/S code . . . (So you probably don't want that!)

Slide 8

O/S Versus Application Programs, Continued

- So maybe we need memory protection too? but we probably needed that anyway.
- How to make memory protection work? more about that later, but for now — again, seems like the only way to do this reliably and efficiently is with help from hardware.
- Most (many?) schemes for memory protection involve some special-purposes registers. Access to these registers — privileged mode or not?

O/S Versus Application Programs, Continued

- How about general-purpose registers? and the PC? should accessing them be privileged, or not?
- (Consider what the processor is actually doing — executing instructions.)

Slide 9

System Calls

- In Homework 1 I ask you to run `strace` and look at its output. The functions it lists are “wrappers” for system calls, and while there should be `man` pages for all of them, sometimes the description isn’t that helpful without more background than you have.
- (Look at a few that many people looked at?)

Slide 10

Slide 11

Classical IPC Problems — Review

- 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 12

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”.)

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 13

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 14

Slide 15

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 16

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 17

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 18

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.

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 19

Minute Essay

- Any questions about IPC (synchronization, classical problems) before we move on?

Slide 20