Slide 1

Administrivia

- Reminder: Homework 1 programming problem due today.

Slide 2

Minute Essay From Last Lecture

- Everyone had had some exposure to multithreading in CS2 and (for some) in later classes, including Web Applications and Game Design. (”Hm!”?)

Slide 3

Mutual Exclusion Solutions So Far

- Solutions so far have some problems: inefficient, dependent on whether scheduler/etc. guarantees fairness.
- Also, they're very low-level, so might be hard to use for more complicated problems.
- So, people have proposed various "synchronization mechanisms" . . .

Slide 4

Synchronization Mechanisms — Overview

- Synchronization using only shared variables seems to be tedious and inefficient.
- "Synchronization mechanisms" are more-abstract ways of coordinating what processes do. A key point is providing *something* that potentially makes a process wait.

Slide 5

Semaphores

- History — 1965 paper by Dijkstra (possibly earlier work by Iverson, of APL/J fame).
- Idea — define semaphore ADT:
  - "Value" — non-negative integer.
  - Two operations, *both atomic*:
    - up (V) — add one to value.
    - down (P) — block until value is nonzero, then subtract one.
- Ignoring for now how to implement this — is it useful?

Slide 6

Mutual Exclusion Using Semaphores

- Shared variables:

```
semaphore S(1);
```

Pseudocode for each process:

```
while (true) {
    down(S);
    do_cr();
    up(S);
    do_non_cr();
}
```

- Invariant: "S has value 1 exactly when no process in its critical region, 0 exactly when one process in its critical region, and never has values other than 0 or 1."

Slide 7

Mutual Exclusion Using Semaphores, Continued

- Invariant again: "S has value 1 exactly when no process in its critical region, 0 exactly when one process in its critical region, and never has values other than 0 or 1."

  Obvious (?) that this means first requirement is met. Can check that others are met too.

Slide 8

Bounded Buffer Problem

- (Example of slightly more complicated synchronization needs.)
- Idea — we have a buffer of fixed size (e.g., an array), with some processes ("producers") putting things in and others ("consumers") taking things out. Synchronization:
  - Only one process at a time can access buffer.
  - Producers wait if buffer is full.
  - Consumers wait if buffer is empty.
- Example of use: print spooling (producers are jobs that print, consumer is printer — actually could imagine having multiple printers/consumers).

Slide 9

Bounded Buffer Problem, Continued

- Shared variables:

```
buffer B(N); // initially empty, can hold N things
```

Pseudocode for producer:

```
while (true) {
    item = generate();
    put(item, B);
}
```

Pseudocode for consumer:

```
while (true) {
    item = get(B);
    use(item);
}
```

- Synchronization requirements:
  1. At most one process at a time accessing buffer.
  2. Never try to `get` from an empty buffer or `put` to a full one.
  3. Processes only block if they "have to".

Slide 10

Bounded Buffer Problem, Continued

- We already know how to guarantee one-at-a-time access. Can we extend that?
- Three situations where we want a process to wait:
  - Only one get/put at a time.
  - If B is empty, consumers wait.
  - If B is full, producers wait.

Slide 11

Bounded Buffer Problem, Continued

- What about three semaphores?
  - One to guarantee one-at-a-time access.
  - One to make producers wait if B is full — so, it should be zero if B is full — "number of empty slots"?
  - One to make consumers wait if B is empty — so, it should be zero if B is empty — "number of slots in use"?

Slide 12

Bounded Buffer Problem — Solution

- Shared variables:

```
buffer B(N); // empty, capacity N
semaphore mutex(1);
semaphore empty(N);
semaphore full(0);
```

Pseudocode for producer:

```
while (true) {
    item = generate();
    down(empty);
    down(mutex);
    put(item, B);
    up(mutex);
    up(full);
}
```

Pseudocode for consumer:

```
while (true) {
    down(full);
    down(mutex);
    item = get(B);
    up(mutex);
    up(empty);
    use(item);
}
```

Slide 13

Implementing Semaphores

- We want to define:
  - Data structure to represent a semaphore.
  - Functions `up` and `down`.
- `up` and `down` should work the way we said, and we'd like to do as little busy-waiting as possible.

Slide 14

Implementing Semaphores, Continued

- Idea — represent semaphore as integer plus queue of waiting processes (represented as, e.g., process IDs).
- Then how should this work . . .

Slide 15

Implementing Semaphores, Continued

- Variables — integer `value`, queue of process IDs `queue`.

```
down() {
    bool zero;
    enter_cr();
    zero = (value == 0);
    if (!zero)
        value -= 1;
    else
        enqueue(current_process, queue);
    leave_cr();
    if (zero)
        block();    // mark current process blocked
}
```

```
up() {
    process p = null;
    enter_cr();
    if (empty(queue))
        value += 1;
    else
        p = dequeue(queue);
    leave_cr();
    if (p != null)
        unblock(p);     // mark p runnable
```

- `enter_cr()`, `leave_cr()` ? next slide.

Slide 16

Implementing Semaphores, Continued

- Revised functions to enter, leave critical region:

```
enter_cr:
    TSL registerX, lockVar
    compare registerX with 0
    if equal, jump to ok
    invoke scheduler # thread yields to another thread
    jump to enter_cr
ok:
    return


leave_cr:
    store 0 in lock
    return
```

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 . . .

Slide 17

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 18

Slide 19

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 20

O/S Versus Application Programs, Continued

- 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).
- What if it's not O/S code?

Slide 21

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.

Slide 22

Minute Essay

- What (if anything) did you find interesting, difficult, or otherwise noteworthy about Homework 1?