Slide 1

Administrivia

- (None?)

Slide 2

Minute Essay From Last Lecture

- (Review question.)
- "Expected" answer: Caching may complicate things, especially with multiple CPUs.
- Many interesting answers.

  Aside: Registers, cache, memory, disk form a "memory hierarchy" but are distinct.

Slide 3

Review — Invariants and Concurrent Algorithms

- Last time we talked about notion of "invariant" being helpful in thinking about concurrent algorithms.
- Loosely speaking — "something about the program that's always true".

  More carefully: a statement about program variables/state such that:
  - It's true initially.
  - If it's true before any statement of the program, it's still true afterwards.

  Verify by first looking at initial state, then at everything in the program that changes variables/states mentioned in the invariant.
- Goal is to come up with an invariant that's not too difficult to verify by looking at the code and implies the property you want — as with loop invariants as (maybe) discussed in CSCI 1323.
- We're doing this informally (not very rigorously, with some hand-waving), but it can be done much more formally and rigorously.

Slide 4

Review — Peterson's Algorithm

- (Review use of invariant.)
- Worth noting that whether it works on real hardware may depend on whether values "written" to memory are actually written right away or cached.

Slide 5

Semaphores

- 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

Minute Essay

- Alleged joke (from some random Usenet person):

  A man's P should exceed his V else what's a sema for?

  Do you understand this? (Remember that P is "down" and V is "up".)

Slide 14

Minute Essay Answer

- It's a pun. The idea is roughly that if you never have a situation in which you've attempted more "down" operations than "up" operations, you didn't need a semaphore. (Or that's what I think it means. The author might have another idea!)