Slide 1

Administrivia

- (None.)

Slide 2

Mutual Exclusion Solutions So Far

- Solutions so far have some problems: inefficient, dependent on whether scheduler/etc. guarantees fairness.

  (It's worth noting too that for the simple ones needing no special hardware — e.g., Peterson's algorithm — whether they work on real hardware may depend on whether values "written" to memory are actually written right away or cached.)
- Also, they're very low-level, so might be hard to use for more complicated problems.
- So, people have proposed various "synchronization mechanisms" . . .

Slide 3

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 4

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 5

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 6

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 7

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 8

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 9

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 10

Bounded Buffer Problem — Solution

- Shared variables:

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

Pseudocode for producer: Pseudocode for consumer:

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

Implementing Semaphores

Slide 11

- 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.

Implementing Semaphores, Continued

Slide 12

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

Slide 13

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 14

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
```

Slide 15

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 16

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!)