Slide 1

Administrivia

- (I guess there's not any!)

Slide 2

Recap — Mutual Exclusion Problem

- Pseudocode for all processes:

```
while (true) {
    do_cr();
    do_non_cr();
}
```

- Goal is to add something to this code such that:
  1. No more than one process at a time can be "in its critical region". Others are "blocked" — could include busy-waiting.
  2. No process not in its critical region can block another process.
  3. No process waits forever to enter its critical region.
  4. No assumptions are made about how many CPUs, their speeds.

Slide 3

Mutual Exclusion Problem, Continued

- We looked at various solutions (some correct, some not) — "concurrent algorithms".
- How to reason about concurrent algorithms (i.e., convince yourself it works)? Idea of "invariant" may be useful:
  - Loosely speaking — "something about the program that's always true". (If this reminds you of "loop invariants" in CS1323 — good.)
  - Goal is to come up with an invariant that's easy to verify by looking at the code and implies the property you want (here, "no more than one process in its critical region at a time").
  - We will do this quite informally, but it can be done much more formally — mathematical "proof of correctness" of the algorithm.

Slide 4

Proposed Solution — Strict Alternation

- Shared variables:

```
int turn = 0;
```

Pseudocode for process p0:

```
while (true) {
    while (turn != 0);
    do_cr();
    turn = 1;
    do_non_cr();
}
```

Pseudocode for process p1:

```
while (true) {
    while (turn != 1);
    do_cr();
    turn = 0;
    do_non_cr();
}
```

- Proposed invariant: "If pn is in its critical region, `turn` has value n." (Might need to expand definition of "in its critical region" a bit.)

Slide 5

Proposed Solution — Peterson's Algorithm

- Shared variables:

```
int turn = 0;   // "who tried most recently"
bool interested0 = false, interested1 = false;
```

Pseudocode for process p0:

```
while (true) {
    interested0 = true;
    turn = 0;
    while ((turn == 0)
        && interested1);
    do_cr();
    interested0 = false;
    do_non_cr();
}
```

Pseudocode for process p1:

```
while (true) {
    interested1 = true;
    turn = 1;
    while ((turn == 1)
        && interested0);
    do_cr();
    interested1 = false;
    do_non_cr();
}
```

- Proposed invariant: "If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1" and similarly for p1.

Slide 6

Proposed Solution — TSL Instruction

- Shared variables:

```
int lock = 0;
```

Pseudocode for each process:

```
while (true) {
    enter_cr();
    do_cr();
    leave_cr();
    do_non_cr();
}
```

Assembly-language routines:

```
enter_cr:
    TSL regX, lock
    compare regX with 0
    if not equal
        jump to enter_cr
    return
leave_cr:
    store 0 in lock
    return
```

- Proposed invariant: "`lock` is 0 exactly when there are no processes in their critical regions, and nonzero exactly when there is one process in its critical region."

Slide 7

Recap — Solutions So Far

- Peterson's algorithm works and requires no special hardware, but is inefficient ("busy-waiting").
- The algorithm using TSL works but requires special hardware and is inefficient.
- Both are pretty low-level. Can we do better? in terms of both efficiency and ease of use?

Slide 8

Semaphores

- History — 1965 paper by Dijkstra (possibly earlier work by Iverson).
- 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 9

Mutual Exclusion Using Semaphores

- Shared variables:

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

  Pseudocode for each process:

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

- Does it work? Proposed invariant: "S has value 1 exactly when no process is in its critical region, 0 exactly when one process is in its critical region."

Slide 10

Bounded Buffer Problem

- 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 11

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 12

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 13

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 14

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 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".)

- Anything else unclear?