Slide 1

Administrivia

- None.

Slide 2

Mutual Exclusion Problem, Again

- In many situations, we want only one process at a time to have access to some shared resource.
- Generic/abstract version — multiple processes, each with a “critical region” (“critical section”):

```
while (true) {
    // wait here if not "safe" to proceed
    do_cr();        // must be "finite"
    do_non_cr();    // need not be "finite"
}
```

- 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”.
  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

Sidebar: Reasoning about Concurrent Algorithms

- Various solutions to the mutual exclusion problem (some correct, some not) — examples of "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 CSCI 1323 — 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 (Revisited)

- 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();
}
```

- Does it work? (Proposed invariant: "If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1"; similarly for p1.)

Slide 6

Proposed Solution — TSL Instruction

- A key problem in concurrent algorithms is the idea of "atomicity" (operations guaranteed to execute without interference from another CPU/process). Hardware can provide some help with this.
- E.g., "test and set lock" (TSL) instruction:

  `TSL registerX, lockVar`

  (1) copies `lockVar` to `registerX` and (2) sets `lockVar` to non-zero, all as one atomic operation.

  How to make this work is the hardware designers' problem!

Slide 7

Proposed Solution — TSL Instruction, Continued

- 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
```

- Does it work? (Proposed invariant: "`lock` is 0 exactly when no processes in their critical regions, and nonzero exactly when one process in its critical region.")

Slide 8

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 9

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 10

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 in its critical region, 0 exactly when one process in its critical region.")

Slide 11

Minute Essay

- TBA