Slide 1

Administrivia

- (None.)

Slide 2

Minute Essay From Last Lecture

- Experience with various kinds of concurrency — a few people have taken CSCI 3366, several wrote some threaded/network programs for PAD II (Lewis!), message-passing MAS (Zhang), Scala actors, message-passing, multithreading in GUI,
- Examples of situations in which mutual exclusion is needed — maintaining integrity of a database (one person had a real-world situation), stock trading (I'm a bit skeptical but maybe), collision modeling, using I/O resources (e.g., printer), simultaneous editing of files,

  One person mentioned situations in which concurrency within an application would be helpful.

Slide 3

Review — Mutual Exclusion Problem

- 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 4

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? Yes.

Slide 5

Peterson's Algorithm, Continued

- Intuitive idea — p0 can only start `do_cr()` if either p1 isn't interested, or p1 is interested but it's p0's turn; `turn` "breaks ties".
- Semi-formal proof using invariants is a bit tricky. Proposed invariant: "If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1"; similarly for p1.

  If we can show this is an invariant, first requirement is met. Others are too.

  But a fiddly detail — the invariant can be false if p0 is in its critical region when p1 executes the lines `interested1 = true; turn = 1;`. See next slide for revision.

Slide 6

Peterson's Algorithm, Continued

- Shared variables:

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

Pseudocode for process p0:

```
while (true) {
    interested0 = true; // L1
    turn = 0;           // L2
    while ((turn == 0)
        && interested1);
    do_cr();
    interested0 = false;
    do_non_cr();
}
```

Pseudocode for process p1:

```
while (true) {
    interested1 = true; // L1
    turn = 1;           // L2
    while ((turn == 1)
        && interested0);
    do_cr();
    interested1 = false;
    do_non_cr();
}
```

- Revised invariant: "If p0 is in its critical region, `interested0` is true and one of the following is true: `interested1` is false, `turn` is 1, or p1 is between L1 and L2", and similarly for p1. Ugly but works.

Peterson's Algorithm, Continued

Slide 7

- Requires essentially no hardware support (aside from "no two simultaneous writes to memory location X" – pretty much a given). Can be extended to more than two processes.
- But complicated and not very efficient.

Sidebar: TSL Instruction

Slide 8

- 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 9

Proposed Solution Using 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
```

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

Slide 10

Solution Using TSL Instruction, Continued

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

  This means first requirement is met. Others met too — well, except that it might be "unfair" (some process waits forever).
- Is this a better solution? Simpler than Peterson's algorithm, but still involves busy-waiting, and depends on hardware features that *might* not be present.

Slide 11

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 12

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 13

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 14

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 15

Minute Essay

- The textbook discusses another instruction that could be useful in implementing some sort of locking, one that swaps the contents of a register and a memory word in a single indivisible action. If you had this instruction (call it SWAP) but no TSL, how would you write the `enter_cr` and `leave_cr` routines?

Slide 16

Minute Essay Answer

- See p. 125 of the textbook. (XCHG is what I've called SWAP.)