Slide 1

Administrivia

- (None.)

Slide 2

Minute Essay From Last Lecture

- (Review question.)
- Several categories of answer:

  Access to shared variables.

  Access to devices — disk, screen, network card, etc.

  Access to database.

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

Sidebar: Reasoning about Concurrent Algorithms

- For concurrent algorithms (such as various solutions proposed for mutual exclusion problem), testing is less helpful than for sequential algorithms. (Why?)
- May be helpful, then, to try to think through whether they work. How? 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 5

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();
}
```

- 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 6

Strict Alternation, Continued

- Invariant again: “If pn is in its critical region, `turn` has value n.” (Might need to expand definition of “in its critical region” a bit.)
- How does this help? means that if p0 and p1 are both in their critical regions, `turn` has two different values — impossible. So the first requirement is met. Still have to think about the other three.

Slide 7

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 8

Peterson’s Algorithm, Continued

- Proposed invariant holds. (“If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1”; similarly for p1.)

  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”.

  As before, this means first requirement is met. Others met too.

- Requires essentially no hardware support (aside from “no two simultaneous writes to memory location X” – pretty much a given).

- But complicated and not very efficient.

Slide 9

Sidebar: 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 10

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? (Proposed invariant: “`lock` is 0 exactly when no processes in their critical regions, and nonzero exactly when one process in its critical region.”)

Solution Using TSL Instruction, Continued

- Proposed invariant holds. (“`lock` is 0 exactly when no processes in their critical regions, and nonzero exactly when one process in its critical region.”)

  This means first requirement is met. Others met too — well, except that it might be “unfair” (some process waits forever).

Slide 11

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 12

Slide 13

Semaphores

- History — 1965 paper by Dijkstra (possibly earlier work by Iverson).
- Idea — define semaphore ADT. Next time.

Slide 14

Minute Essay

- What problems do you think hardware designers might face in implementing a TSL instruction as described? (Think about a single processor/core versus many, cache(s), and anything else you think is relevant.)