Slide 1

Administrivia

- Reminder: Homework 1 programming problem due today.
- Homework 2 not ready yet but probably will be before next class. I'll send e-mail. You'll have at least a week to work on it.

Slide 2

Minute Essay From Last Lecture

- Responses about loop invariants varied — some people didn't remember hearing about them at all, others remembered only vaguely. I think they're a useful way of thinking about loops. I have an overview in my slides from last time.
- Most people had some exposure to multithreading. My impression is that it's often mentioned in CS2, Data Abstraction, and Algorithms.

Slide 3

Mutual Exclusion Problem — Review

- 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) {
    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

Mutual Exclusion Problem — Recap

- So far we looked at a few proposed solutions that didn’t work, for various reasons.
- Now we’ll look at some that do (yay!).

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? Yes . . .

Slide 6

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 has two parts:
  - “If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1”; similarly for p1.
  - “`turn` is either 0 or 1.”
- If we can show that, first requirement (no more than one process in critical region) is true. Other requirements are too.

  Second part is clearly okay, but for the first, a fiddly detail — the invariant can be false if p0 is in its critical region when p1 executes the lines `interested1 = true; turn = 1;`. So revise a bit . . .

Slide 7

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 (first part): “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 (I claim) works . . .

Slide 8

Peterson’s Algorithm, Continued

- Revised invariant again: “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. Invariant?
- True initially.
- Could change when either process enters its critical region. But this only happens . . . when? So okay.
- Doesn’t change when eiher process leaves its critical region (somewhat trivially).
- Changes to `intereste`n — this is where the revision comes in; if the other process is in its critical region then it’s a bit fiddly, but okay with revision.
- Changes to `turn` are okay.
- So okay!

Peterson’s Algorithm, Continued

- Requires essentially no hardware support (aside from “no two simultaneous writes to memory location X” — fairly safe assumption as long as X is a single “word”). Can be extended to more than two processes.
- But complicated and not very efficient because it “busy-waits”.

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

Slide 11

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 . . .

Slide 12

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.” (“Exactly when” here means “if and only if”.)
- If this invariant holds, that means first requirement is met. (Does it hold? Next slide.) 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. (Also depends on hardware features that *might* not be present, but these days almost all hardware has something similar.)

Slide 13

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.” (“Exactly when” here means “if and only if”.)
- True initially.
- Could change when a process enters its critical region — but notice that only happens when `lock` is 0.
- Also doesn’t change when a process leaves its critical region.
- So okay.

Slide 14

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 depends on whether values “written” to memory are actually written right away or cached. Surprisingly difficult to guarantee that!)
- Also, they’re very low-level, so might be hard to use for more complicated problems.
- So, people have proposed various “synchronization mechanisms” . . .

Slide 15

Synchronization Mechanisms — Overview

- Synchronization using only shared variables seems to be tedious and inefficient.
- “Synchronization mechanisms” are more-abstract ways of coordinating what processes do. A key point is providing *something* that potentially makes a process wait.

Slide 16

Semaphores

- History — 1965 paper by Dijkstra (possibly earlier work by Iverson, or so says a former faculty member who knows of Iverson through his work on APL/J).
- 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 17

Mutual Exclusion Using Semaphores

- Shared variables:

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

  Pseudocode for each process:

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

- Proposed 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 18

Mutual Exclusion Using Semaphores, Continued

- Proposed 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.”
- True initially.
- Could change when a process enters its critical region — but this is essentially exactly when a `down(S)` completes, so okay.
- Could change when a process leaves its critical region — but this is essentially exactly when an `up(S)` completes, so okay.

Slide 19

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 20

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 21

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 22

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 23

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 24

Minute Essay

- Does what I'm saying about using invariants to reason about concurrent algorithms make sense to you?