Slide 1

Administrivia

- Reminder: Homework 1 programming problem due today.

  If you can't finish completely by the due date/time, but you have something that represents at least a good start, *send me what you have* and submit a revised/improved version as soon as you can. You lose fewer points that way, I think you learn more, and I'd rather grade code that works than code that doesn't!

  Please remember to mention the course and the assignment in the subject line. No Google-Drive shares please! If working remotely, consider using `mail-files` script (see "sample programs" page) to send mail from command line (so attaching a file is easy even if it's on the remote system).

- At least one copy of textbook on reserve in the library. 1-day reserve, which I hope will give those without their own copies a reasonable chance . . . ?

Slide 2

Reasoning about Concurrent Algorithms — Review/Recap

- 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 3

Strict Alternation (Review)

- 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, and `turn` is either 0 or 1” (interpreting “in its critical region” as “from just after the `while` to the line after `do_cr()`”.

Slide 4

Strict Alternation, Continued

- Proposed invariant again: “If pn is in its critical region, `turn` has value n, and `turn` is either 0 or 1”.
- How would this help? would mean that if p0 and p1 are both in their critical regions, `turn` has two different values — impossible. So the first requirement would be met. (Still have to think about the other three.)
- Is it an invariant? check whether true initially and remains true even when one process changes something it mentions. Fairly obvious that it’s initially true, so check . . .

Slide 5

Strict Alternation, Continued

- Proposed invariant: “If pn is in its critical region, `turn` has value n, and `turn` is either 0 or 1”. True initially. When could it become false?
- When either process enters its critical region. But this happens for pn only when `turn` is n, so invariant stays true (okay).
- When either process leaves its critical region. Also okay.
- When either process changes `turn`. Only happens after process leaves its critical region. So also okay.

Slide 6

Proposed Solution — Peterson’s Algorithm (Review)

- 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 7

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 for revision.

Slide 8

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 9

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 `interested`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!

Slide 10

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 11

Proposed Solution Using TSL Instruction (Review)

- 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, and depends on hardware features that *might* not be present.

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 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 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

- What do you remember about loop invariants from CSCI 1323?
- The discussion of invariants in concurrent algorithms — useful? interesting? inscrutable?