Administrivia

- Reminder: Homework 1 programming problem due Wednesday. Please include that "honor code statement" described in the writeup. (Fewer than half did on the written problems!)

Slide 1

Minute Essay From Last Lecture

- Most people came reasonably close. Exactly how many processes *are* in each state depends on many factors, but max/min values don't really.
- (See my answer.)

Slide 2

Slide 3

Interprocess Communication

- Processes almost always need to interact with other processes:
  - "Ordering constraints" – e.g., process B uses as input some data produced by process A.
  - Use of shared resources — files, shared memory locations, etc.
- Use of shared resources can lead to "race conditions" — output depends on details of interleaving.
- Processes must communicate to avoid race conditions and otherwise synchronize.
- "Classical IPC problems" — simplified versions of things you often want to do.

Slide 4

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) {
    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 5

Mutual Exclusion Problem, Continued

- We'll look at various solutions (some correct, some not):
  - Using only hardware features always present (some notion of shared variable).
  - Using optional hardware features.
  - Using "synchronization mechanisms" (abstractions that help solve this and other problems).
- Recall that a correct solution
  - Must work for more than one CPU.
  - Must work even in the face of unpredictable context switches — whatever we're doing, another process can pull the rug out from under us between "atomic operations" (machine instructions).

Slide 6

Sidebar: Atomic Operations

- "Atomic" operation — indivisible, executes without interference from other processes.
- Which of the following are atomic?
  - `x = 1;`
  - `x = x + 1;`
  - `++x;`
  - `if (x == 0) x = 1;`

  (Or does it depend? On what?)

Slide 7

Proposed Solution — Disable Interrupts

- Pseudocode for each process:

```
while (true) {
    disable_interrupts();
    do_cr();
    enable_interrupts();
    do_non_cr();
}
```

- Does it work? reviewing the criteria . . .

Slide 8

Disable Interrupts, Continued

- (1) okay – context switches take place only in response to interrupts, so yes if one CPU.
- (4) not okay — fails if more than one CPU (unless there is a way to disable interrupts on all CPUs).
- Also, user-level programs shouldn't be able to do this (though might be okay for O/S).

Slide 9

Proposed Solution — Simple Lock Variable

- Shared variables:

```
int lock = 0;
```

Pseudocode for each process:

```
while (true) {
    while (lock != 0);
    lock = 1;
    do_cr();
    lock = 0;
    do_non_cr();
}
```

- Does it work? reviewing the criteria . . .

Slide 10

Simple Lock Variable, Continued

- Can easily fail (1).

Slide 11

Proposed Solution — Strict Alternation

- 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();
}
```

- Does it work? reviewing the criteria . . .

Slide 12

Strict Alternation, Continued

- (Yes, we're simplifying to only two processes.)
- (1) okay.
- (2) / (3) not okay, since non-critical region need not be finite.

Slide 13

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 14

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 15

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 16

Sidebar of Sidebar: Reasoning About Loops

- (I probably won't have time to through these slides in much detail in class but will leave them here for anyone interested.)
- Usually want to prove two things — the loop eventually terminates, and it establishes some desired postcondition.
- Proving that it terminates — define a *metric* that you know decreases by some minimum amount with every trip through the loop, and when it goes below some threshold value, the loop ends.
- Proving that it establishes the postcondition — use a *loop invariant*.
- (I say "prove" here, since this can be done very rigorously, but in practical situations an informal version is usually good enough.)

Slide 17

Reasoning About Loops, Continued

- What's a loop invariant? in the context of reasoning about programs, it's a *predicate* (boolean expression using program variables) that
  - is true before the loop starts, and
  - if true before a trip through the loop, with the loop condition true, is also true after the trip through the loop.

  If you can prove that a particular predicate is a loop invariant, after the loop exits, you know it's still true, and the loop condition is not. With a well-chosen invariant, this is enough to prove useful things.
- (Might be worth noting that compiler writers have a different definition — some computation that can be moved outside the loop.)

Slide 18

Reasoning About Loops, Simple Example

- Loop to compute sum of elements of array `a` of size `n`:

```
i = 0; sum = 0;
while (i != n) {
    sum = sum + a[i];
    i = i + 1;
}
```

  At end, `sum` is sum of elements of `a`.
- Does this work? well, you probably believe it does, but you could prove it using the invariant:

  `sum` is the sum of `a[0]` through `a[i-1]`

Slide 19

Reasoning About Loops, Example

- Euclid's algorithm for computing greatest common divisor of nonnegative integers a and b:

```
i = a; j = b;
while (j != 0) {
    q = i / j; r = i % j;
    i = j; j = r;
}
```

At end, `i = gcd(a, b)`.

- Does this work? work through some examples and gain some confidence — or prove using invariant:

  `gcd(i, j) = gcd(a, b)`

  and the math fact `gcd(n, 0) = n`

Slide 20

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 21

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 22

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 23

- 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.

Sidebar: TSL Instruction

Slide 24

- 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 25

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 26

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 27

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" . . . (To be continued.)

Slide 28

Minute Essay

- Tell me about your exposure to concurrent programming (multi-threading, message passing, etc. — anything involving multiple threads of control).