Slide 1

Administrivia

- (Return quizzes. Solutions in hardcopy.)

Slide 2

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 3

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 primitives" (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 4

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 5

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 6

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 7

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 8

Simple Lock Variable, Continued

- Can easily fail (1).

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 . . .

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 11

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 12

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 13

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 14

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 15

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 16

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]`

Reasoning About Loops, Example

Slide 17

- 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`

Minute Essay

Slide 18

- Tell me about your experience (if any!) with writing programs that involve concurrency — multithreaded, message-passing, communicating over sockets, etc.
- Anything today (or last time) that was particularly unclear, or that you want to know more about?