Slide 1

Administrivia

- (By e-mail.)

Slide 2

Threads and Concurrency

- Most operating systems present material on concurrency abstractly, as part of discussion of processes, and mention threads only as a special type of process.

  This textbook, however, presents concurrency only in the context of threads, and includes plenty of specifics, geared toward systems that use POSIX threads (Pthreads).

- In the next few lectures I'm going to present it abstractly, in the hope that the combination of this abstract view and the lower-level view in the textbook will offer something for everyone.

- First topic is what's traditionally called the "mutual exclusion problem" and is in the textbook discussed as "locks" . . .

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

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 5

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 6

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 7

Simple Lock Variable, Continued

- Can easily fail (1).

Slide 8

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 9

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 10

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 11

Sidebar of Sidebar: Reasoning About Loops

- (I don't have time to go through these slides in much detail in class but will leave them here for anyone interested.)
- Usually want to prove two things: (1) the loop eventually terminates, and (2) 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 good enough.)

Slide 12

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, then 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 13

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 14

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 15

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();
}
```

- 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 16

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 17

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 18

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 19

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.

  Doing this formally is a bit tricky — some fiddly details — so I won't, but it's possible.

Slide 20

Peterson's Algorithm, Continued

- In principle, 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.
- In practice, writes to memory may not happen right away: C compilers (and probably other languages) don't require that, and hardware may also cache values to write. Hardware that does this has instructions to provide a "memory fence" (that guarantees all writes have completed), but a solution to mutual exclusion would need to use them.
- So, this can be made to work, but it's complicated and not very efficient because it "busy-waits".
- Hardware can help! To be continued . . .

Slide 21

Minute Essay

- Did you learn about loop invariants in CSCI 1323 (Discrete Structures)?
- Questions?