Slide 1

## Administrivia

- Career networking event ("Making Connections") tomorrow, starting at 5:45pm in the Great Hall. Sounds like it might be worthwhile.

Slide 2

## Sidebar: Reasoning about Concurrent Algorithms (Review)

- 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

## Sidebar of Sidebar: Reasoning About Loops

- 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 4

## 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 5

## 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];
}
```

  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 6

## 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 7

## Review — 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) {
    // wait here if not "safe" to proceed
    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 8

## Proposed Solution — 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 p$n$ is in its critical region, `turn` has value $n$." (Might need to expand definition of "in its critical region" a bit.)

Slide 9

## Strict Alternation, Continued

- Invariant again: "If p$n$ 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 p$0$ and p$1$ 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 10

## 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. (Proposed invariant: "If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1"; similarly for p1.)

Slide 11

## Peterson's Algorithm, Continued

- Proposed invariant holds. ("If p0 is in its critical region, `interested0` is true and either `interested1` is false or `turn` is 1"; similarly for p1.) (*Caveat*: As mentioned in class, 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.)

  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".

  As before, this means first requirement is met. Others met too.
- Requires essentially no hardware support (aside from "no two simultaneous writes to memory location X" – pretty much a given).
- But complicated and not very efficient.

Slide 12

## Peterson's Algorithm Again

- 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.

Slide 13

## 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 14

## 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.")

## 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).

**Slide 15**

## Minute Essay

- The TSL-based solution seems okay, but — are there circumstances in which a process might wait forever?
- Even if this solution is correct, might it have disadvantages?

**Slide 16**

Slide 17

## Minute Essay Answer

- A process might wait forever, if the mechanism used to schedule processes somehow only gives it a turn when another process holds the lock.
- Even when the solution works, it involves processes "busy-waiting" and is thus potentially quite inefficient — though that might not matter much if critical regions are very short.

  It also relies on a hardware feature that *might* not be present, and it requires that processes cooperate.