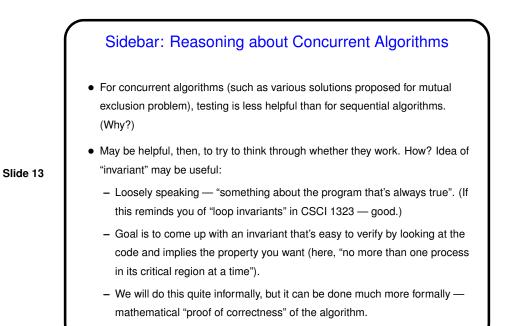
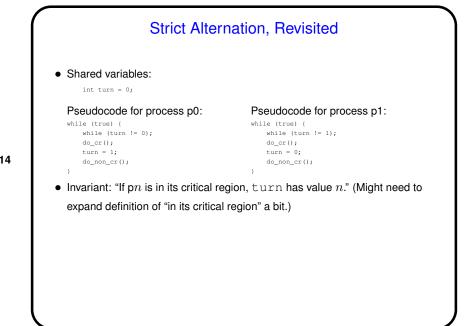
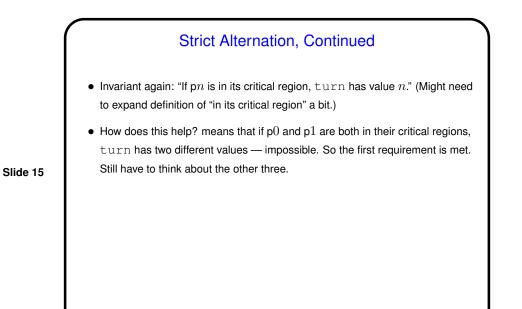


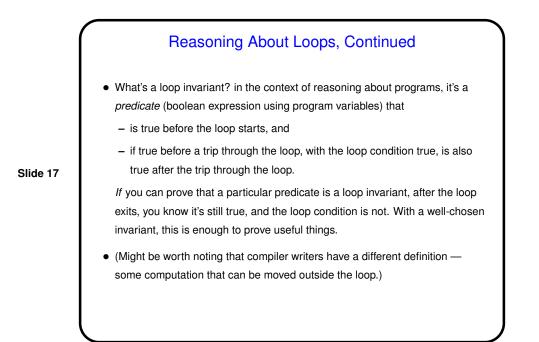
Strict Alternation, Continued
(Yes, we're simplifying to only two processes.)
(1) okay.
(2) / (3) not okay, since non-critical region need not be finite.

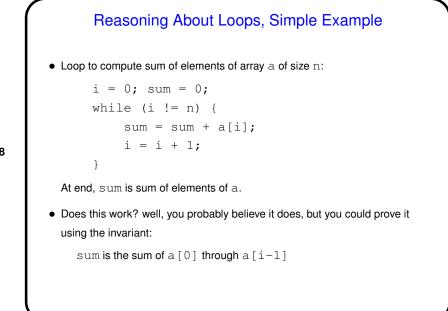


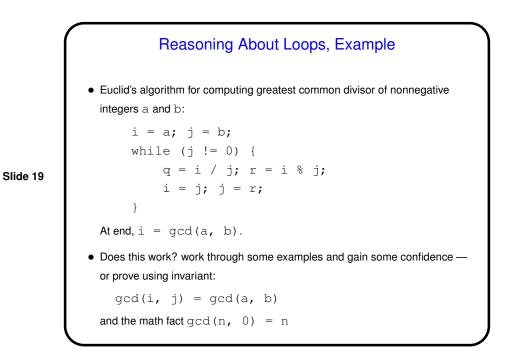




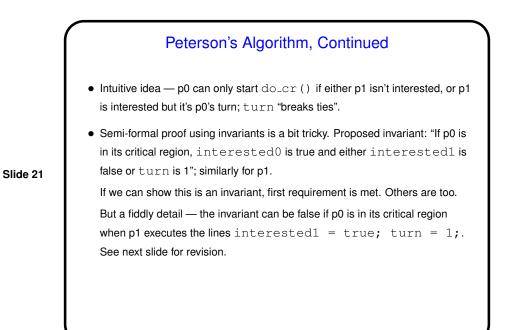
Sidebar of Sidebar: Reasoning About Loops
(I probably won't have time to through these slides in much detail in class but will leave them here for anyone interested.)
Usually want to prove two things — the loop eventually terminates, and it establishes some desired postcondition.
Proving that it terminates — 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.)

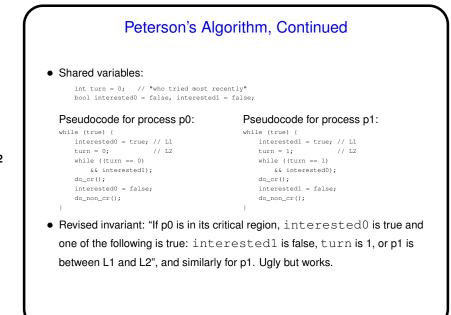


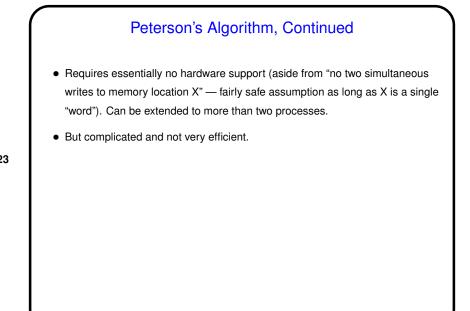


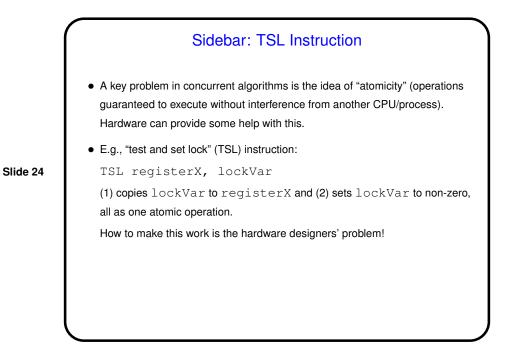


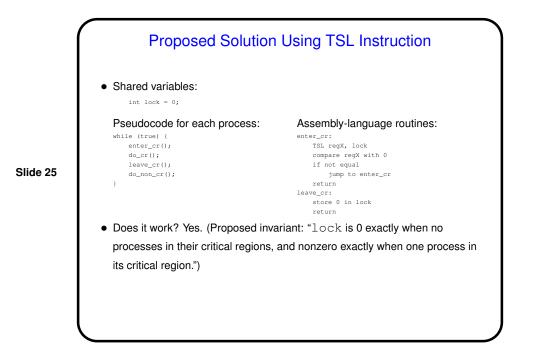
Proposed Solution — Peterson's Algorithm • Shared variables: int turn = 0; // "who tried most recently"
bool interested0 = false, interested1 = false; Pseudocode for process p0: Pseudocode for process p1: while (true) { while (true) { interested1 = true; interested0 = true; turn = 0;turn = 1; ile ((turn == 0)
&& interestedl); while ((turn == 0) while ((turn == 1) && interested0); do_cr(); do_cr(); interested0 = false; interested1 = false; do_non_cr(); do_non_cr(); • Does it work? Yes.











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.

