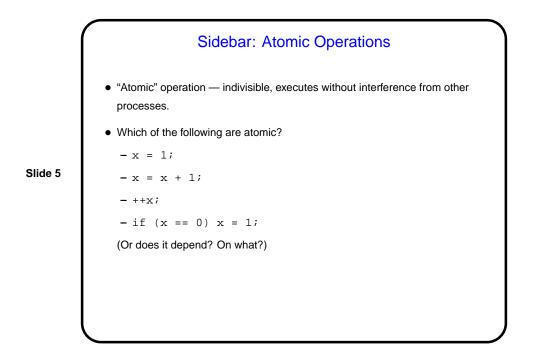


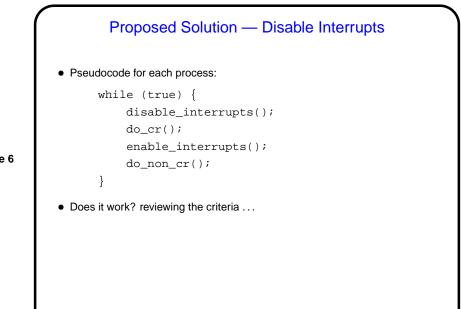
Mutual Exclusion Problem, Continued
We'll look at various solutions (some correct, some not):

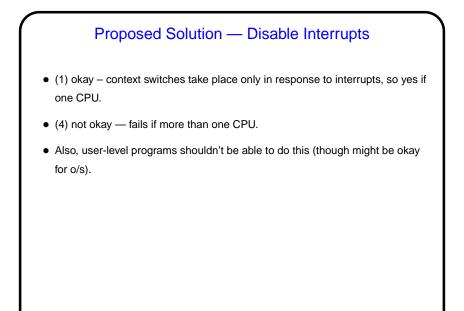
Using only hardware features always present (some notion of shared variable).
Using optional hardware features.
Using "synchronization primitives" (abstractions that help solve this and other problems).

Recall that a correct solution

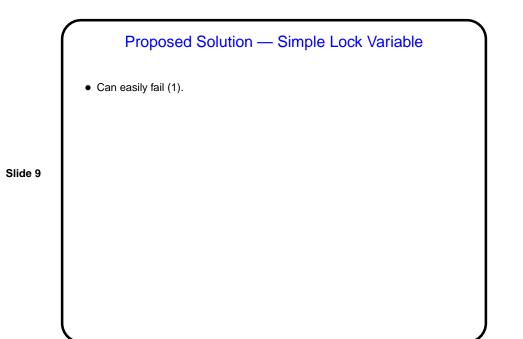
Must work for more than one CPU.
Must work even in the face of unpredictable context switches — whatever we're doing, another process can pull the rug out from under us between "atomic operations" (machine instructions).

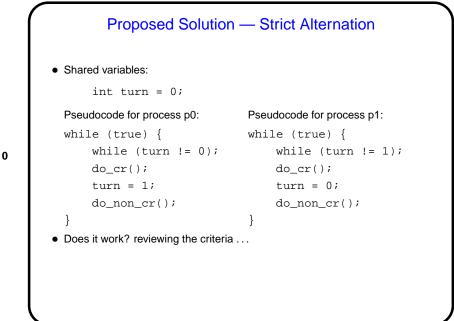


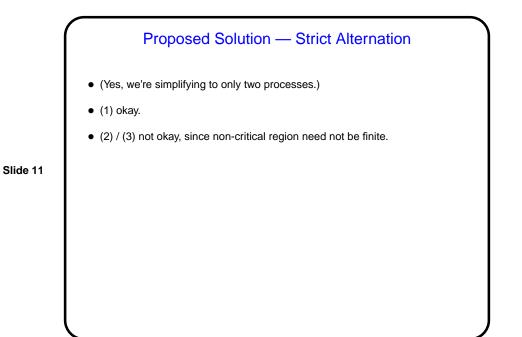




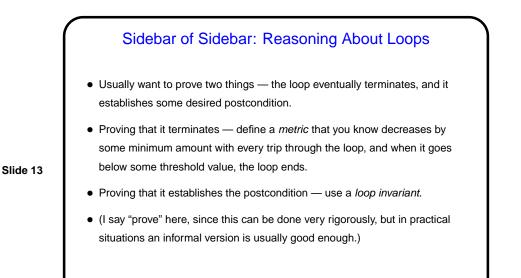
<pre>hared variables: int lock = 0; seudocode for each process: hile (true) { while (lock != 0); lock = 1; do_cr(); lock = 0; do_non_cr();</pre>		
<pre>seudocode for each process: hile (true) { while (lock != 0); lock = 1; do_cr(); lock = 0; do_non_cr();</pre>		
<pre>hile (true) { while (lock != 0); lock = 1; do_cr(); lock = 0; do_non_cr();</pre>	int lock =	0;
<pre>while (lock != 0); lock = 1; do_cr(); lock = 0; do_non_cr();</pre>	seudocode for eacl	h process:
<pre>lock = 1; do_cr(); lock = 0; do_non_cr();</pre>	hile (true) {	
<pre>do_cr(); lock = 0; do_non_cr();</pre>	while (loc	:k != 0);
<pre>lock = 0; do_non_cr();</pre>	lock = 1;	
do_non_cr();	do_cr();	
	lock = 0;	
	do_non_cr();
oes it work? reviewing the criteria	oes it work? review	ving the criteria

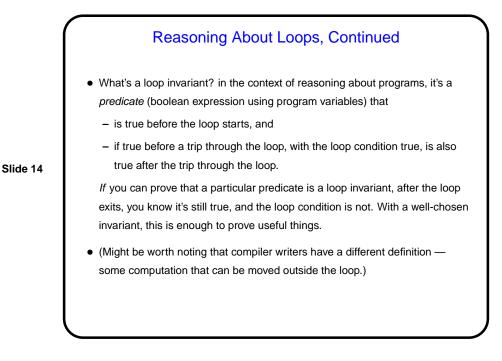






Sidebar: Reasoning about Concurrent Algorithms For concurrent algorithms (such as various solutions proposed for mutual exclusion problem), testing is less helpful than for sequential algorithms. (Why?) May be helpful, then, to try to think through whether they work. How? Idea of "invariant" may be useful: 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.





At end, sum is sum of elements of a.

}

i = 0; sum = 0; while (i != n) {

• Does this work? well, you probably believe it does, but you could prove it using the invariant:

Reasoning About Loops, Simple Example

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

• Loop to compute sum of elements of array a of size n:

sum = sum + a[i];

