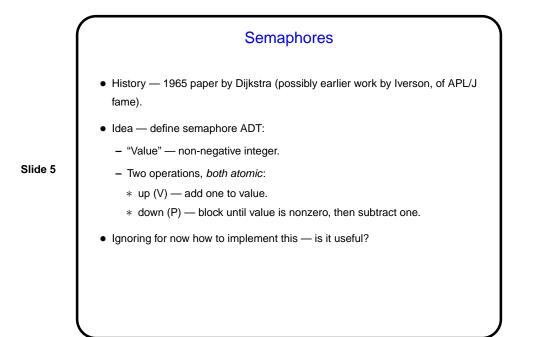
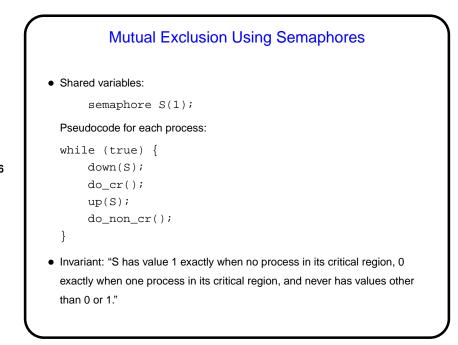
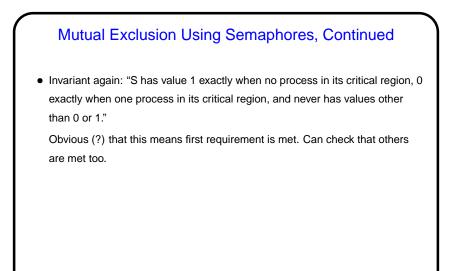
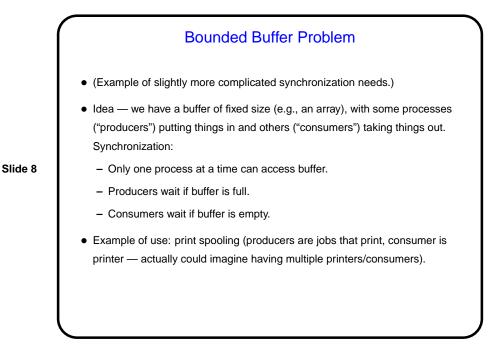


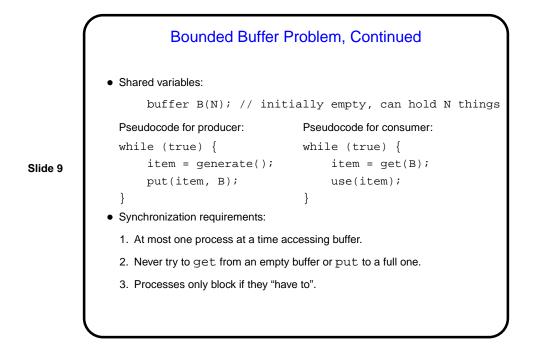
Synchronization Mechanisms — Overview
Synchronization using only shared variables seems to be tedious and inefficient.
"Synchronization mechanisms" are more-abstract ways of coordinating what processes do. A key point is providing *something* that potentially makes a process wait.

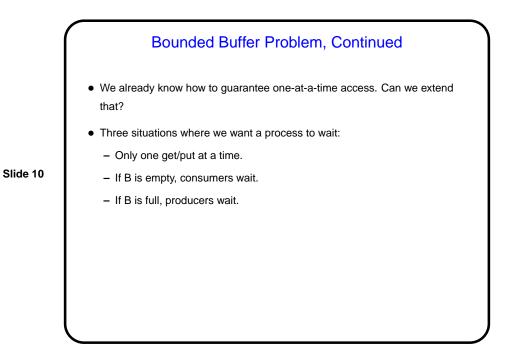


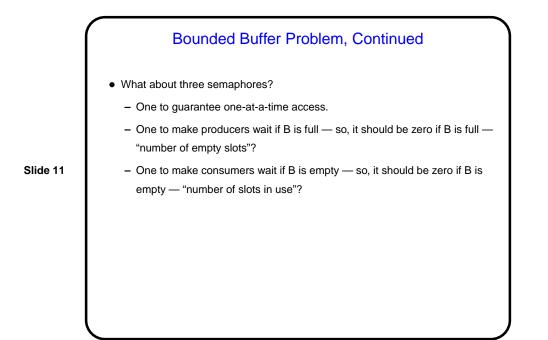




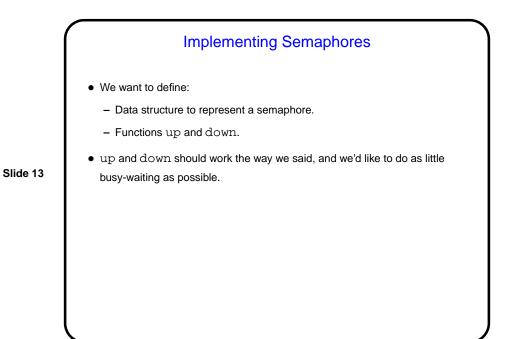


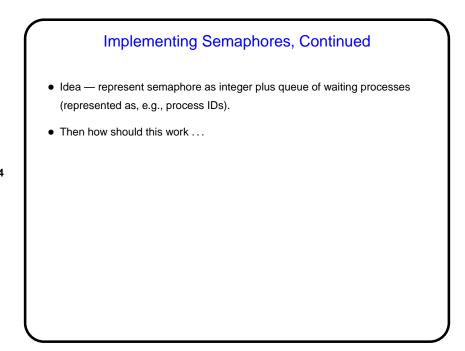


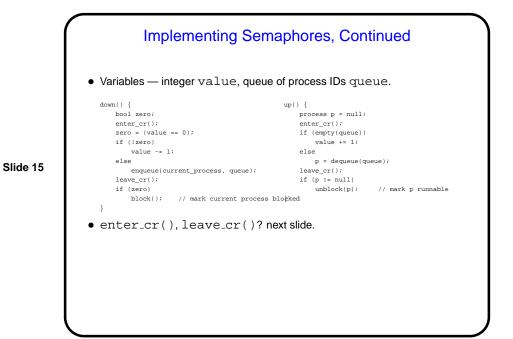




	Bounded Buffer	Problem — Solution
	• Shared variables: buffer B(N); // emp	ty, capacity N
	<pre>semaphore mutex(1); semaphore empty(N); semaphore full(0);</pre>	
Slide 12	<pre>Pseudocode for producer: while (true) { item = generate(); down(empty); down(mutex); put(item, B); up(mutex); up(full); }</pre>	<pre>Pseudocode for consumer: while (true) { down(full); down(mutex); item = get(B); up(mutex); up(empty); use(item); }</pre>







13

Sidebar: Shared Memory and Synchronization Solutions that rely on variables shared among processes assume that assigning a value to a variable actually changes its value in memory (RAM), more or less right away. Fine as a first approximation, but reality may be more complicated, because of various tricks used to deal with relative slowness of accessing memory: Optimizing compilers may keep variables' values in registers, only reading/writing memory when necessary to preserve semantics. Hardware may include cache, logically between CPU and memory, such that memory read/write goes to cache rather than RAM. Different CPUs' caches may not be in synch.

Slide 17

Sidebar: Shared Memory and Synchronization, Continued

• So, actual implementations need notion of "memory fence" — point at which all apparent reads/writes have actually been done. Some languages provide standard ways to do this; others (e.g., C!) don't. C's volatile ("may be changed by something outside this code") helps some but may not be enough.

