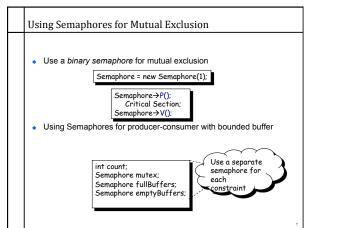
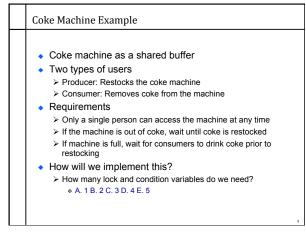
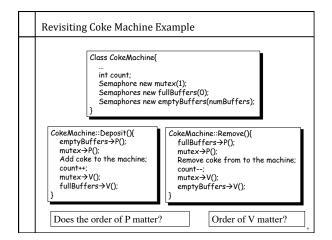
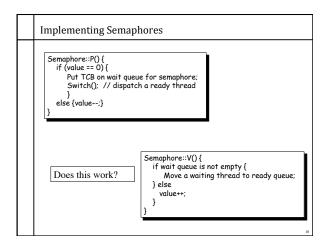


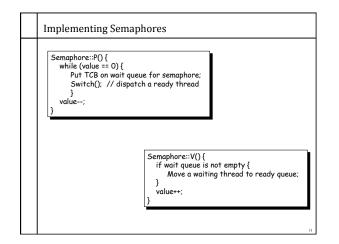
How many possible values can a binary semaphore	
take?	
> A. 0	
≻ B. 1	
≻ C. 2	
≻ D. 3	
≻ E. 4	

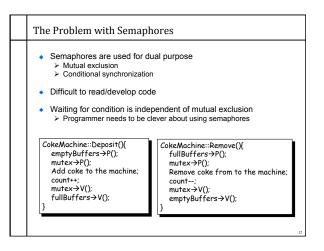












Introducing Monitors
 Separate the concerns of mutual exclusion and conditional synchronization
What is a monitor?
One lock, and
Zero or more condition variables for managing concurrent access to shared data
 General approach:
Collect related shared data into an object/module
Define methods for accessing the shared data
 Monitors first introduced as programming language construct
Calling a method defined in the monitor automatically acquires the lock
Examples: Mesa, Java (synchronized methods)

 Monitors also define a programming convention > Can be used in any language (C, C++, ...)

Critical Section: Monitors

Basic idea:

- > Restrict programming model
- > Permit access to shared variables only within a critical section

General program structure

- Entry section
 * "Lock" before entering critical section Wait if already locked, or invariant doesn't hold
 - * Key point: synchronization may involve wait
- Critical section code

- Object-oriented programming style
 - Associate a lock with each shared object

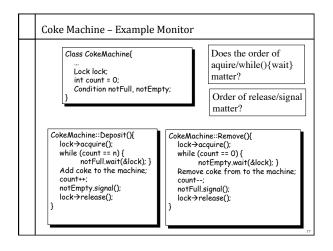
 - > Methods that access shared object are critical sections > Acquire/release locks when entering/exiting a method that
 - defines a critical section

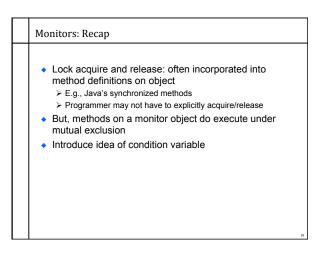
Remember Condition Variables Locks Provide mutual exclusion > Support two methods

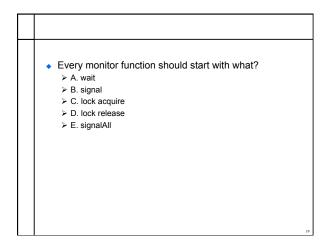
- Lock::Acquire() wait until lock is free, then grab it Lock::Release() – release the lock, waking up a waiter, if any
- Condition variables
 - > Support conditional synchronization
 - > Three operations
 - Wait(): Release lock; wait for the condition to become true; reacquire lock upon return (Java wait())
 - Signal(): Wake up a waiter, if any (Java notify())
 - * Broadcast(): Wake up all the waiters (Java notifyAll())
 - > Two semantics for implementation of wait() and signal()
 - Hoare monitor semantics
 - * Hansen (Mesa) monitor semantics

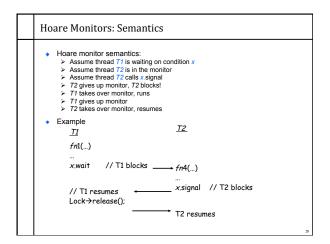
So what is the big idea?

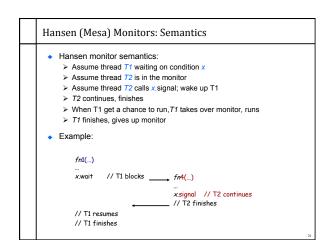
- (Editorial) Integrate idea of condition variable with language
 - Facilitate proof
 - > Avoid error-prone boiler-plate code

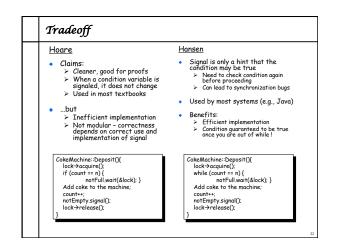


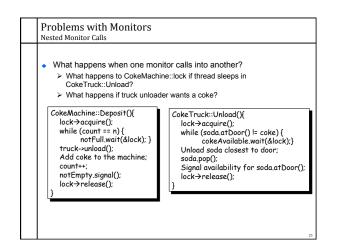




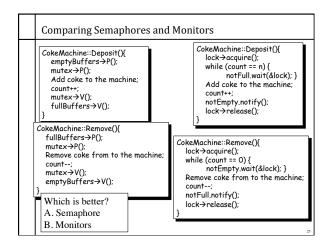


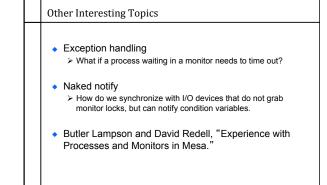






More Monitor Headaches The priority inversion problem
 Three processes (P1, P2, P3), and P1 & P3 communicate using a monitor <i>M</i>. P3 is the highest priority process, followed by P2 and P1. 1. P1 enters M. 2. P1 is preempted by P2. 3. P2 is preempted by P3. 4. P3 tries to enter the monitor, and waits for the lock. 5. P2 runs again, preventing P3 from running, subverting the priority system. A simple way to avoid this situation is to associate with each monitor the priority of the highest priority process which ever enters that monitor.
34





Summary
 Synchronization Coordinating execution of multiple threads that share data structures
 Past lectures: ≻ Locks → provide mutual exclusion ≻ Condition variables → provide conditional synchronization
 Today: > Semaphores Introduced by Dijkstra in 1960s Two types: binary semaphores and counting semaphores Supports both mutual exclusion and conditional synchronization > Monitors Separate mutual exclusion and conditional synchronization
27