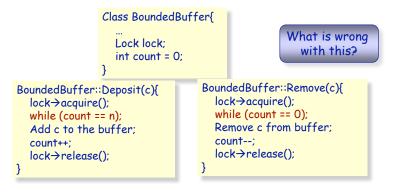
Condition Synchronization

Beyond Locks

- Locks ensure mutual exclusion
- Is this sufficient?
 - > What if you want to synchronize on a condition?
 - > Example: Producer-consumer problem



Introducing Condition Variables

- Correctness requirements for bounded buffer producerconsumer problem
 - Only one thread manipulates the buffer at any time (mutual exclusion)
 - Consumer must wait for producer when the buffer is empty (scheduling/synchronization constraint)
 - Producer must wait for the consumer when the buffer is full (scheduling/synchronization constraint)

Solution: condition variables

- > An abstraction that supports conditional synchronization
- > Key idea:
 - Enable threads to wait inside a critical section by atomically releasing lock at the same time

Condition Variables: Operations

Three operations

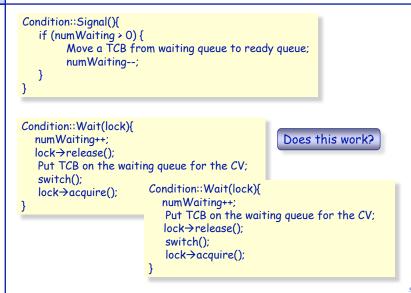
- > Wait()
- Release lock
- Go to sleep
- Reacquire lock upon return
- Signal()
 - Wake up a waiter, if any
- > Broadcast()
 - * Wake up all the waiters
- Implementation
 - > Requires a per-condition variable queue to be maintained

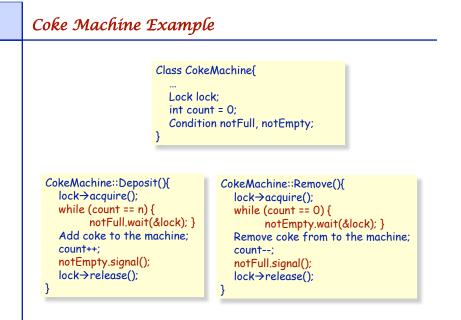
Wait() usually specifies a lock

to be released as a parameter

> Threads waiting for the condition wait for a signal()

Implementing Wait() and Signal()





Using Condition Variables: An Example

- Coke machine as a shared buffer
- Two types of users
 - Producer: Restocks the coke machine
 - > Consumer: Removes coke from the machine
- Requirements
 - > Only a single person can access the machine at any time
 - > If the machine is out of coke, wait until coke is restocked
 - > If machine is full, wait for consumers to drink coke prior to restocking
- How will we implement this?
 - > What is the class definition?
 - > How many lock and condition variables do we need?

Semaphores and Monitors: High-level Synchronization Constructs

A Historical Perspective

Synchronization Constructs

- Synchronization
 - Coordinating execution of multiple threads that share data structures

Past few lectures:

- > Locks: provide mutual exclusion
- > Condition variables: provide conditional synchronization
- Today: Historical perspective
 - > Semaphores
 - Introduced by Dijkstra in 1960s
 - * Main synchronization primitives in early operating systems
 - > Monitors
 - * Alternate high-level language constructs

Semaphores

- An abstract data type
- A non-negative integer variable with two *atomic* operations

Semaphore→P() (Passeren; wait) Atomically: If sem > 0, then decrement sem by 1 Otherwise "wait" until sem > 0

Semaphore→V() (Vrijgeven; signal) Atomically: Increment sem by 1

- We assume that a semaphore is fair
 - No thread t that is blocked on a P() operation remains blocked if the V() operation on the semaphore is invoked infinitely often
 - > In practice, FIFO is mostly used, transforming the set into a queue.

Important properties of Semaphores

- Semaphores are *non-negative* integers
- The only operations you can use to change the value of a semaphore are P() and V() (except for the initial setup)
 P() can block, but V() never blocks
- Semaphores are used both for
 - > Mutual exclusion, and
 - > Conditional synchronization
- Two types of semaphores
 - > Binary semaphores: Can either be 0 or 1
 - > General/Counting semaphores: Can take any non-negative value
 - Binary semaphores are as expressive as general semaphores (given one can implement the other)

Using Semaphores for Mutual Exclusion

Use a binary semaphore for mutual exclusion

Semaphore = new Semaphore(1);

Semaphore→P(); Critical Section; Semaphore→V();

 Using Semaphores for producer-consumer with bounded buffer

> Semaphore mutex; Semaphore fullBuffers; Semaphore emptyBuffers;

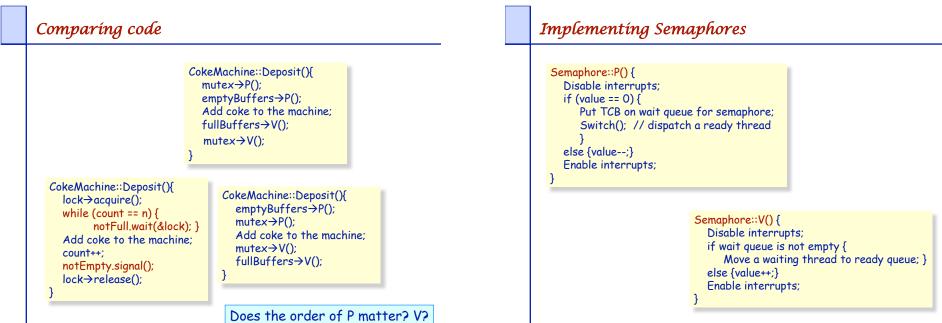
Use a separate semaphore for each constraint



Class CokeMachine{

Semaphores new fullBuffers(0); Semaphores new emptyBuffers(numBuffers);

CokeMachine::Deposit(){ emptyBuffers→P(); mutex→P(); Add coke to the machine; mutex→V(); fullBuffers→V(); CokeMachine::Remove(){ fullBuffers→P(); mutex→P(); Remove coke from to the machine; mutex→V(); emptyBuffers→V();



Implementing Semaphores

Semaphore::P() {
 Disable interrupts;
 while (value == 0) {
 Put TCB on wait queue for semaphore;
 Switch(); // dispatch a ready thread
 }
 value--;
 Enable interrupts;
}

Semaphore::V() {
 Disable interrupts;
 if wait queue is not empty {
 Move a waiting thread to ready queue; }
 value++;
 Enable interrupts;

The Problem with Semaphores

- Semaphores are used for dual purpose
 - > Mutual exclusion
 - Conditional synchronization
- Difficult to read/develop code
- Waiting for condition is independent of mutual exclusion

3

Programmer needs to be clever about using semaphores

CokeMachine::Deposit(){ emptyBuffers→P(); mutex→P(); Add coke to the machine; mutex→V(); fullBuffers→V(); }

CokeMachine::Remove(){ fullBuffers→P(); mutex→P(); Remove coke from to the machine; mutex→V(); emptyBuffers→V();