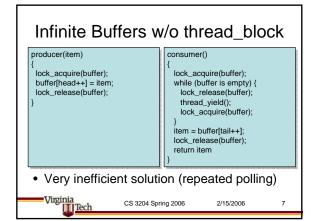
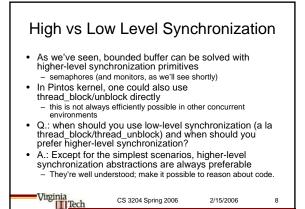


# Infinite Buffer Problem (revisited) This is a correct solution; in fact, we've just reinvented "monitors" – topic of this lecture. Q1: What if we hadn't had direct access to thread\_block/thread\_unblock? Q2: Even if we have, should we use them?





### **Monitors**

- A monitor combines a set of shared variables & operations to access them
- Think of an enhanced C++ class with no public fields
   A monitor provides implicit synchronization (only
- A monitor provides implicit synchronization (only one thread can access private variables simultaneously)
  - Single lock is used to ensure all code associated with monitor is within critical section
- · A monitor provides a general signaling facility
  - Wait/Signal pattern (similar to, but different from semaphores)
  - May declare & maintain multiple signaling queues



### Monitors (cont'd)

- Classic monitors are embedded in programming language
  - Invented by Hoare & Brinch-Hansen 1972/73
  - First used in Mesa/Cedar System @ Xerox PARC 1978
  - Limited version available in Java/C#
- (Classic) Monitors are safer than semaphores
- can't forget to lock data compiler checks this
- In contemporary C, monitors are a synchronization pattern that is achieved using locks & condition variables
  - Must understand monitor abstraction to use it



2/15/2006

# Infinite Buffer w/ Monitor monitor buffer { | buffer::produce(item i) |

/\* implied: struct lock mlock;\*/
private:
 char buffer[];
 int head, tail;
public:
 produce(item);
 item consume();
}

buffer::consum { /\* try { lock buffer[head-/\* } finally {lc} }

buffer::consum { /\* try { lock buffer::consum { /\* try { lock return buffer /\* } finally {lc} }

{ /\* try { lock\_acquire(&mlock); \*/
buffer[head++] = i;
 /\* } finally {lock\_release(&mlock);} \*/
}
buffer::consume()
{ /\* try { lock\_acquire(&mlock); \*/
 return buffer[tail++];
 /\* } finally {lock\_release(&mlock); \*/

11

Monitors provide implicit protection for their internal variables

- Still need to add the signaling part

CS 3204 Spring 2006 2/15/2006

### **Condition Variables**

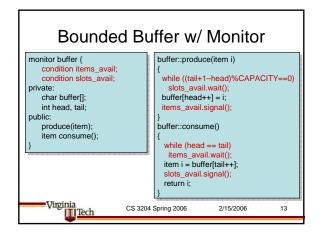
- Variables used by a monitor for signaling a condition
  - a general (programmer-defined) condition, not just integer increment as with semaphores
- Monitor can have more than one condition variable
- · Three operations:
  - Wait(): leave monitor, wait for condition to be signaled, reenter monitor
  - Signal(): signal one thread waiting on condition
  - Broadcast(): signal all threads waiting on condition

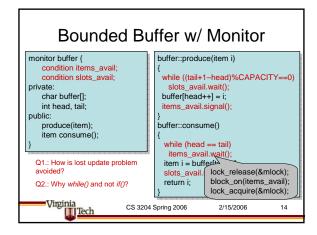
Virginia Tech

CS 3204 Spring 2006

2/15/2006

12





### Implementing Condition Variables • State is just a queue of waiters: - Wait(): adds current thread to (end of queue) & block - Signal(): pick one thread from queue & unblock it Hoare-style Monitors: gives lock directly to waiter Mesa-style monitors (C, Pintos, Java): signaler keeps lock – waiter gets READY, but can't enter until signaler gives up lock

- Broadcast(): unblock all threads · Compare to semaphores:
- Condition variable signals are lost if nobody's on the queue (semaphore's V() are remembered)
- Condition variable wait() always blocks (semaphore's P() may or may not block)



### Monitors in C

- POSIX Threads & Pintos
- No compiler support, must do it manually
  - must declare locks & condition vars
- must call lock\_acquire/lock\_release when entering&leaving the
  - must use cond\_wait/cond\_signal to wait for/signal condition
- Note: cond\_wait(&c, &m) takes monitor lock as
- parameter
- Necessary so monitor can be left & reentered without losing signals
- Pintos cond\_signal() takes lock as well
  - only as debugging help/assertion to check lock is held when signaling
- pthread\_cond\_signal() does not

Virginia 11 Tech CS 3204 Spring 2006

2/15/2006 16

## Summary

- · Semaphores & Monitors are both higherlevel constructs
- Monitors in C is just a programming pattern that involves mutex+condition variables
- When should you use which?

Virginia Tech 17 CS 3204 Spring 2006 2/15/2006