

Bounded Buffer w/ Monitor

Synchronization 1

```

monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}

```

```

buffer::produce(item i)
{
    while ((tail+1-head)%CAPACITY==0)
        slots_avail.wait();
    buffer[head++] = i;
    items_avail.signal();
}
buffer::consume()
{
    while (head == tail)
        items_avail.wait();
    item i = buffer[tail++];
    slots_avail.signal();
    return i;
}

```

Computer Science Dept Va Tech August 2007
Operating Systems
©2003-07 McQuain

Bounded Buffer w/ Monitor

Synchronization 2

```

monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}

```

```

buffer::produce(item i)
{
    while ((tail+1-head)%CAPACITY==0)
        slots_avail.wait();
    buffer[head++] = i;
    items_avail.signal();
}
buffer::consume()
{
    while (head == tail)
        items_avail.wait();
    item i = buffer[tail++];
    slots_avail.signal();
    return i;
}

```

Q1.: How is lost update problem avoided?

Q2.: Why *while()* and not *if()*?

```

lock_release(&mlock);
block_on(items_avail);
lock_acquire(&mlock);

```

Computer Science Dept Va Tech August 2007
Operating Systems
©2003-07 McQuain

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)

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 monitor
- 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

Mesa vs Hoare Style

Synchronization 5

Mesa-style:

- Cond_signal leaves signaling thread in monitor
- so must always use “while()” when checking loop condition
- POSIX Threads & Pintos are Mesa-style (and so are C# & Java)

Alternative is “Hoare”-style where cond_signal leads to exit from monitor and immediate reentry of waiter

- Not commonly used

Monitors in Java

Synchronization 6

synchronized *block* means

- enter monitor
- *execute block*
- leave monitor

wait()/notify() use condition variable associated with receiver

- Every object in Java can function as a condition var

```
class buffer {
  private char buffer[];
  private int head, tail;
  public synchronized produce(item i) {
    while (buffer_full())
      this.wait();
    buffer[head++] = i;
    this.notify();
  }
  public synchronized item consume() {
    while (buffer_empty())
      this.wait();
    buffer[tail++] = i;
    this.notify();
  }
}
```

Per Brinch Hansen's Criticism

Synchronization 7

See *Java's Insecure Parallelism* [Brinch Hansen 1999]

Says Java abused concept of monitors because Java does not *require* all accesses to shared variables to be within monitors

Why did designers of Java not follow his lead?

- Performance: compiler can't easily decide if object is local or not - conservatively, would have to make all public methods synchronized - pay at least cost of atomic instruction on entering every time

Readers/Writer w/ Monitor

Synchronization 8

```
struct lock mlock; // protects rdrs & wrtrs
int readers = 0, writers = 0;
struct condvar canread, canwrite;
void read_lock_acquire() {
    lock_acquire(&mlock);
    while (writers > 0)
        cond_wait(&canread, &mlock);
    readers++;
    lock_release(&mlock);
}
void read_lock_release() {
    lock_acquire(&mlock);
    if (--readers == 0)
        cond_signal(&canwrite, &mlock);
    lock_release(&mlock);
}
```

```
void write_lock_acquire() {
    lock_acquire(&mlock);
    while (readers > 0 || writers > 0)
        cond_wait(&canwrite, &mlock);
    writers++;
    lock_release(&mlock);
}
void write_lock_release() {
    lock_acquire(&mlock);
    writers--;
    ASSERT(writers == 0);
    cond_signal(&canread, &mlock);
    cond_signal(&canwrite, &mlock);
    lock_release(&mlock);
}
```

Q.: does this implementation prevent starvation?

Summary Synchronization 9

Semaphores & Monitors are both higher-level constructs

Monitors can be included in a language (Mesa, Java)

- in C, however, they are just a programming pattern that involves a structured way of using mutex+condition variables

When should you use which?

Computer Science Dept Va Tech August 2007 Operating Systems ©2003-07 McQuain

High vs Low Level Synchronization Synchronization 10

As we've seen, bounded buffer can be solved with higher-level synchronization primitives

- semaphores and monitors

In Pintos kernel, one could also use `thread_block/unblock` directly

- this is not always efficiently possible in other concurrent environments

Q.: when should you use low-level synchronization (a la `thread_block/thread_unblock`) and when should you prefer higher-level synchronization?

A.: Except for the simplest scenarios, higher-level synchronization abstractions are always preferable

- They're well understood; make it possible to reason about code.

Computer Science Dept Va Tech August 2007 Operating Systems ©2003-07 McQuain