# CS202 (003): Operating Systems
# Concurrency III

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish’s CS202

# Last time

# Condition Variables

Warning: Condition Variable is not really a Variable!

```
void cond_init(Cond *cond, ...);
void cond_wait(Cond *cond, Mutex *mutex);
void cond_signal(Cond *cond);
void cond_broadcast(Cond *cond);
```

```
mutex_lock(&mutex);
while (!condition_is_met) {
    cond_wait(&cond, &mutex);
}
// Modify shared state
mutex_unlock(&mutex);
```

Why is this a while?

# Condition Variables

Warning: Condition Variable is not really a Variable!

```
void cond_init(Cond *cond, ...);
void cond_wait(Cond *cond, Mutex *mutex);
void cond_signal(Cond *cond);
void cond_broadcast(Cond *cond);
```

```
mutex_lock(&mutex);
while (!condition_is_met) {
    cond_wait(&cond, &mutex);
}
// Modify shared state
mutex_unlock(&mutex);
```

This **MUST** be a while!

# More hypothetical questions…

Why do cond_wait releases the mutexes and goes into the waiting state in one function call (see panel 2b of handout 04)?

If those two steps were separate, could get stuck waiting.

```
Producer: while (count == BUFFER_SIZE)
Producer: release()
Consumer: acquire()
Consumer: .....
Consumer: cond_signal(&nonfull)
Producer: cond_wait(&nonfull)
```

**Producer never hears the signal!**

# More hypothetical questions…

Can we replace SIGNAL with BROADCAST, and preserve correctness*?

Yes, but it might hurt performance

Since while() checks the invariant,
Only thread satisfying the invariant will make progress

=> this does not affect correctness

But we make needlessly wakeup of threads

=> this might hurt performance

correctness*: not having race conditions, and making progress when possible

# More hypothetical questions…

Can we replace BROADCAST with SIGNAL, and preserve correctness*?

No race conditions, but may never make progress

correctness*: not having race conditions, and making progress when possible

# Monitor: Mutex + Conditional Variables (but in OOP)

**All** method calls of a class are protected by a **mutex**

Synchronization happens with condition variables whose associated mutex is the **mutex that protects the method calls**

“Monitor” can be used to refer to either a *programming convention* or a *method in certain programming languages**

* https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

# What does monitor enable us to do?

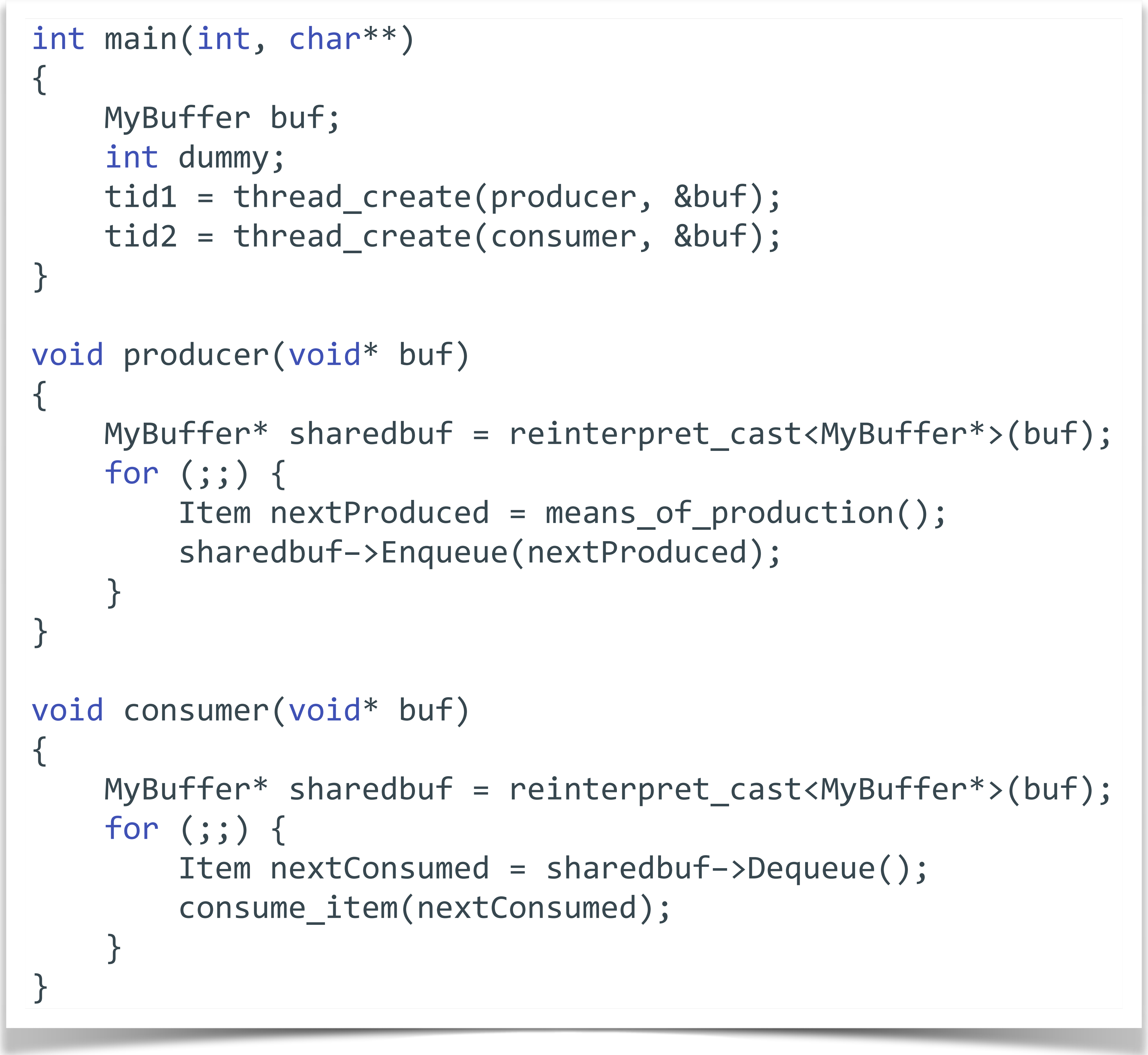
**Encapsulation!**

Separation of program logic inside threads from the shared object

The monitor handles all synchronization internally so threads don't need to worry about locking, unlocking or conditional signaling

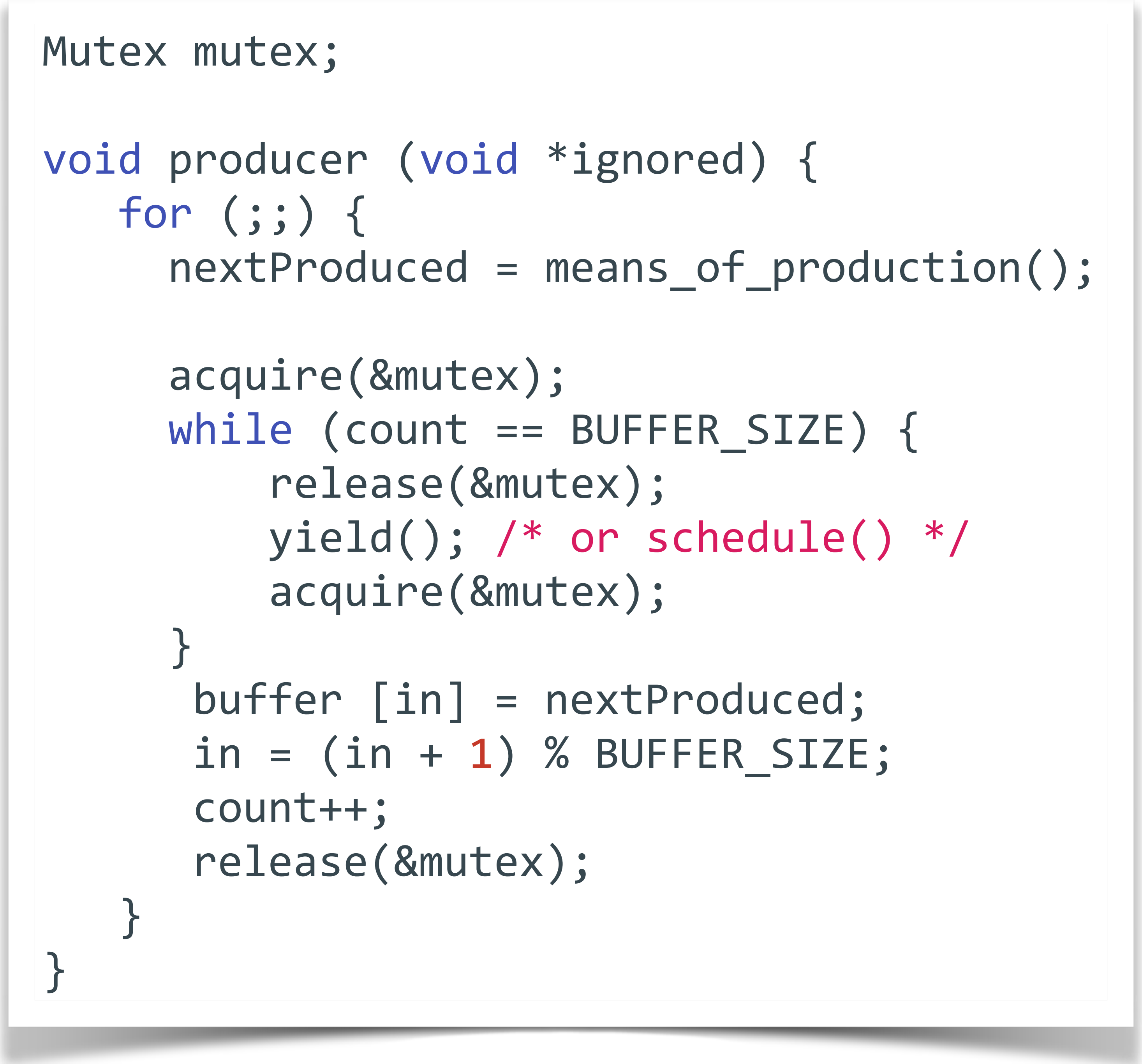
**Look at the first page of handout05!**

## Producer/Consumer w/ Monitor

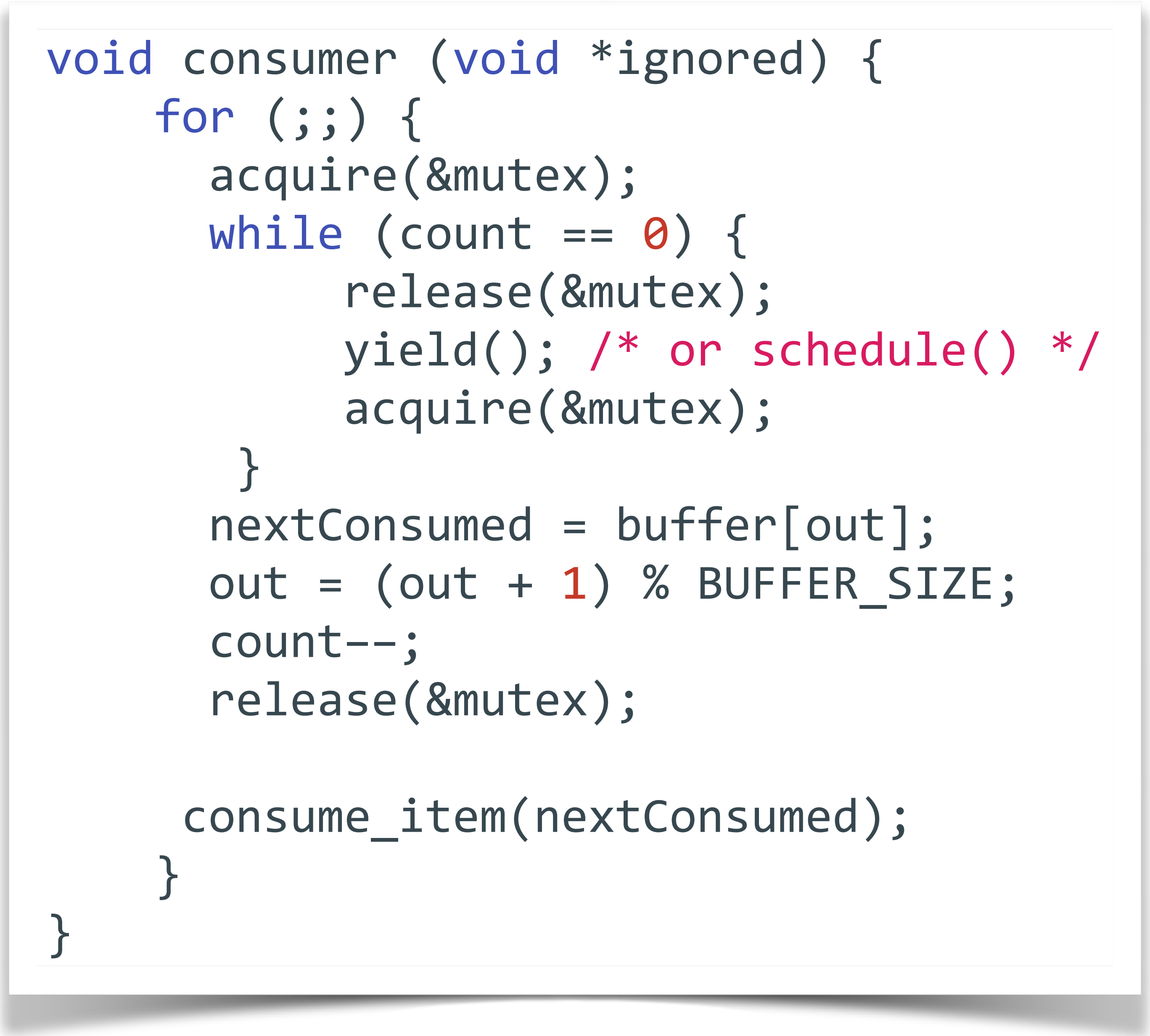


```
int main(int, char**)
{
    MyBuffer buf;
    int dummy;
    tid1 = thread_create(producer, &buf);
    tid2 = thread_create(consumer, &buf);
}

void producer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextProduced = means_of_production();
        sharedbuf->Enqueue(nextProduced);
    }
}

void consumer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextConsumed = sharedbuf->Dequeue();
        consume_item(nextConsumed);
    }
}
```

## Producer/Consumer w/ Mutex & CV



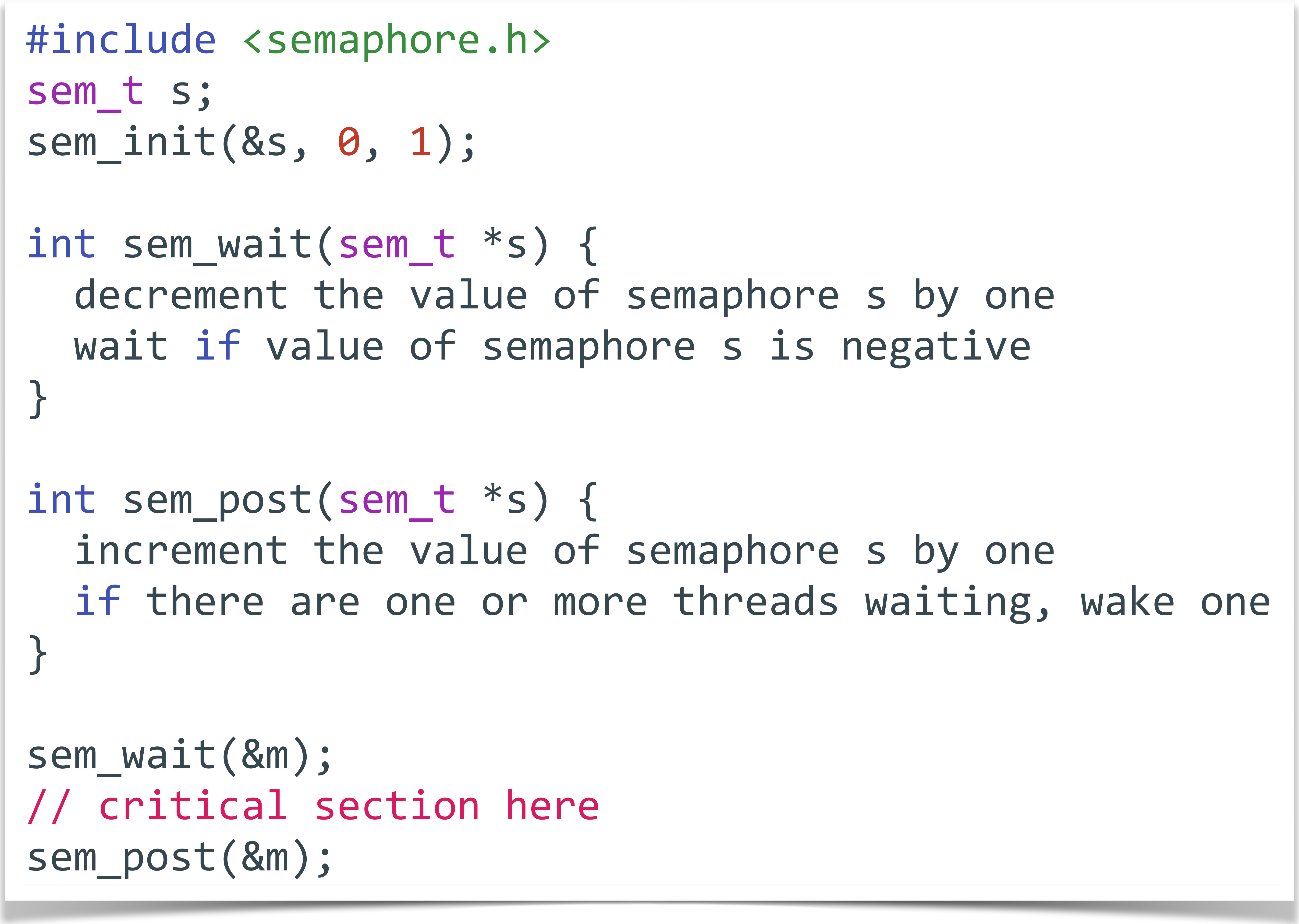
```
Mutex mutex;

void producer (void *ignored) {
   for (;;) {
     nextProduced = means_of_production();

     acquire(&mutex);
     while (count == BUFFER_SIZE) {
         release(&mutex);
         yield(); /* or schedule() */
         acquire(&mutex);
     }
      buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
      release(&mutex);
   }
}
```



```
void consumer (void *ignored) {
    for (;;) {
      acquire(&mutex);
      while (count == 0) {
           release(&mutex);
           yield(); /* or schedule() */
           acquire(&mutex);
       }
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      count--;
      release(&mutex);

     consume_item(nextConsumed);
    }
}
```

* These are pseudocode. Class is a special data type used for OOP.

# Semaphores: Mutex + Conditional Variables (but more general)



```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);

int sem_wait(sem_t *s) {
  decrement the value of semaphore s by one
  wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
  increment the value of semaphore s by one
  if there are one or more threads waiting, wake one
}

sem_wait(&m);
// critical section here
sem_post(&m);
```

# Semaphores: Mutex + Conditional Variables (but more general)

Semaphores manage a count, mutex+CV do not inherently do this

Semaphores can allow multiple threads access, unlike a basic mutex

Semaphores can be used for locking, but can also be used for other purpose

**DO NOT USE SEMAPHORE IN THIS CLASS!**

# Monitor: Mutex + Conditional Variables

**All** method calls are protected by a **mutex**

Synchronization happens with condition variables whose associated mutex is the **mutex that protects the method calls**

"Monitor" can be used to refer to either a *programming convention* or a *method in certain programming languages**

**Please follow these conventions on Lab 3!**

* https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

# Mike Dahlin’s “Programming with Threads”

You are required to follow this document
(although we don’t code in Java)

You will lose a lot of points if you don’t follow
(in labs and exams)

Do not program concurrency in other ways
unless you are a concurrency guru

# Standards for Programming w/ Threads

**Rule I:** acquire/release at beginning/end of methods

**Rule II:** hold lock when doing condition variable operations

**Rule III:** a thread that is in wait() must be prepared to be restarted at any time, not just when another thread calls "signal()"

**Rule IV:** don't call sleep()

# Advice for concurrent programming

**Top-level piece of advice: SAFETY FIRST**

Locking at coarse grain is easiest to get right, so do that

Don’t worry about performance at first

Don’t view deadlock as a disaster

**MAKE SURE YOU PROGRAM NEVER DOES THE WRONG THING**

# Advice for concurrent programming

**Getting started**

1. Identify unit of concurrency
2. Identify chunks of state
3. write down high-level main loop of each thread

**Write down the synchronization constraints, and the type**

**Create a lock or CV for each constraint**

**Implement the methods, using the locks and CVs**

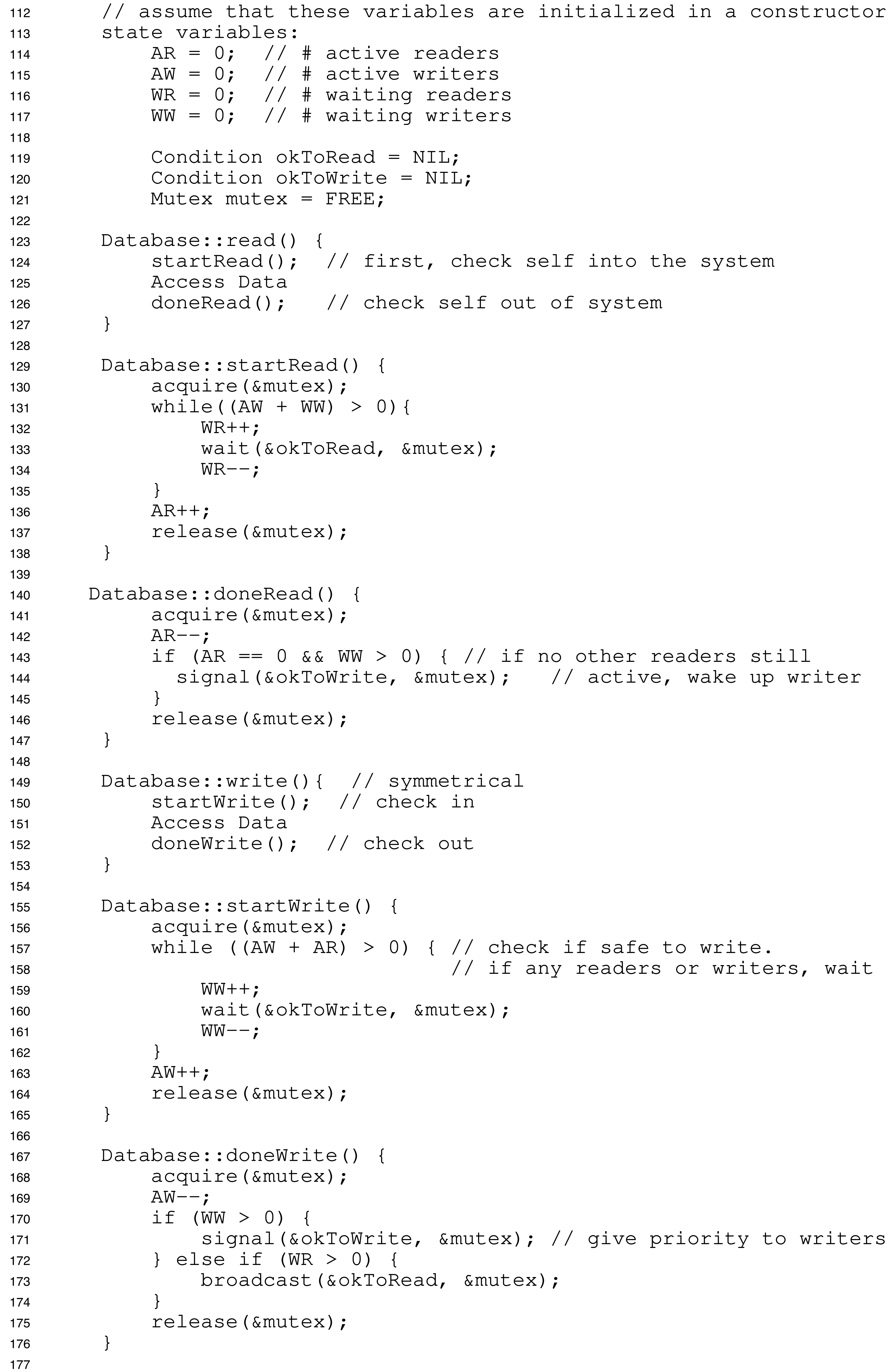
CS 202, Fall 2024
Handout 5 (Class 6)

The previous handout demonstrated the use of mutexes and condition variables. This handout demonstrates the use of monitors (which combine mutexes and condition variables).

1. The bounded buffer as a monitor

```
    // This is pseudocode that is inspired by C++.
    // Don’t take it literally.

    class MyBuffer {
      public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item = Dequeue();
      private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
    };

    void
    MyBuffer::MyBuffer()
    {
        in = out = count = 0;
        mutex = new Mutex;
        nonempty = new Cond;
        nonfull = new Cond;
    }

    void
    MyBuffer::Enqueue(Item item)
    {
        mutex.acquire();
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        ++count;
        cond_signal(&nonempty, &mutex);
        mutex.release();
    }

    Item
    MyBuffer::Dequeue()
    {
        mutex.acquire();
        while (count == 0)
            cond_wait(&nonempty, &mutex);

        Item ret = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        --count;
        cond_signal(&nonfull, &mutex);
        mutex.release();
        return ret;
    }


    int main(int, char**)
    {
        MyBuffer buf;
        int dummy;
        tid1 = thread_create(producer, &buf);
        tid2 = thread_create(consumer, &buf);

        // never reach this point
        thread_join(tid1);
        thread_join(tid2);
        return -1;
    }

    void producer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            Item nextProduced = means_of_production();
            sharedbuf->Enqueue(nextProduced);
        }
    }

    void consumer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            Item nextConsumed = sharedbuf->Dequeue();

            /* next line abstractly consumes the item */
            consume_item(nextConsumed);
        }
    }
```

Key point: *Threads* (the producer and consumer) are separate from *shared object* (MyBuffer). The synchronization happens in the shared object.



```
    // assume that these variables are initialized in a constructor
    state variables:
        AR = 0;  // # active readers
        AW = 0;  // # active writers
        WR = 0;  // # waiting readers
        WW = 0;  // # waiting writers

        Condition okToRead = NIL;
        Condition okToWrite = NIL;
        Mutex mutex = FREE;

    Database::read() {
        startRead();  // first, check self into the system
        Access Data
        doneRead();   // check self out of system
    }

    Database::startRead() {
        acquire(&mutex);
        while((AW + WW) > 0){
            WR++;
            wait(&okToRead, &mutex);
            WR--;
        }
        AR++;
        release(&mutex);
    }

   Database::doneRead() {
        acquire(&mutex);
        AR--;
        if (AR == 0 && WW > 0) { // if no other readers still
          signal(&okToWrite, &mutex);   // active, wake up writer
        }
        release(&mutex);
    }

    Database::write(){  // symmetrical
        startWrite();  // check in
        Access Data
        doneWrite();  // check out
    }

    Database::startWrite() {
        acquire(&mutex);
        while ((AW + AR) > 0) { // check if safe to write.
                                // if any readers or writers, wait
            WW++;
            wait(&okToWrite, &mutex);
            WW--;
        }
        AW++;
        release(&mutex);
    }

    Database::doneWrite() {
        acquire(&mutex);
        AW--;
        if (WW > 0) {
            signal(&okToWrite, &mutex); // give priority to writers
        } else if (WR > 0) {
            broadcast(&okToRead, &mutex);
        }
        release(&mutex);
    }
```

- workers interact with a database
- readers never modify
- writers read an modify
- allow:
    - many readers at once
      OR
    - only one writer (no reader)

Unit of concurrency?

Shared chunks of state?

What does main function looks like?

Synchronization constraints and objects?