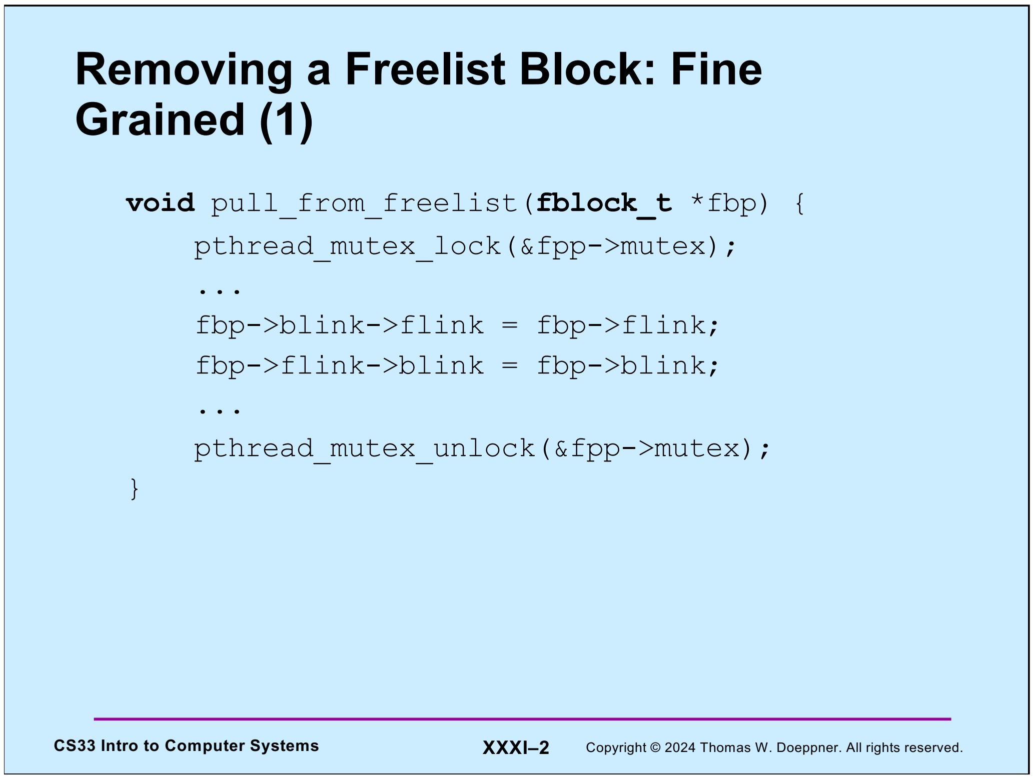
CS 33

Multithreaded Programming II

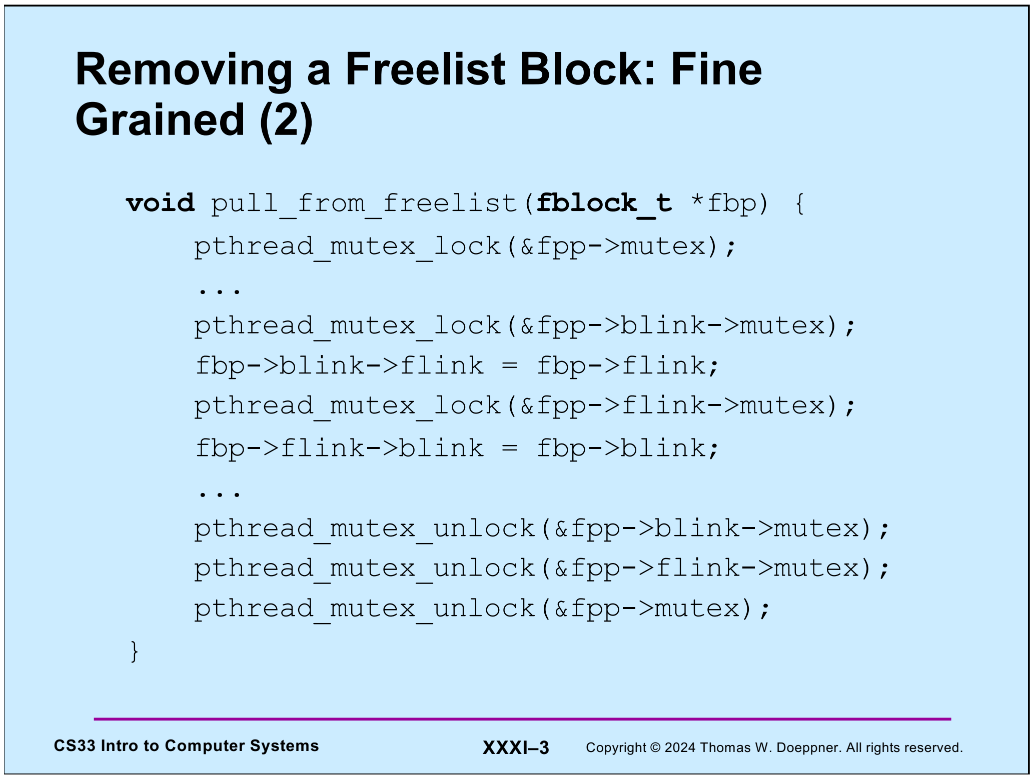


Removing a Freelist Block: Fine Grained (1)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->mutex);
}
```

Let’s try a similar approach for the fine-grained approach.

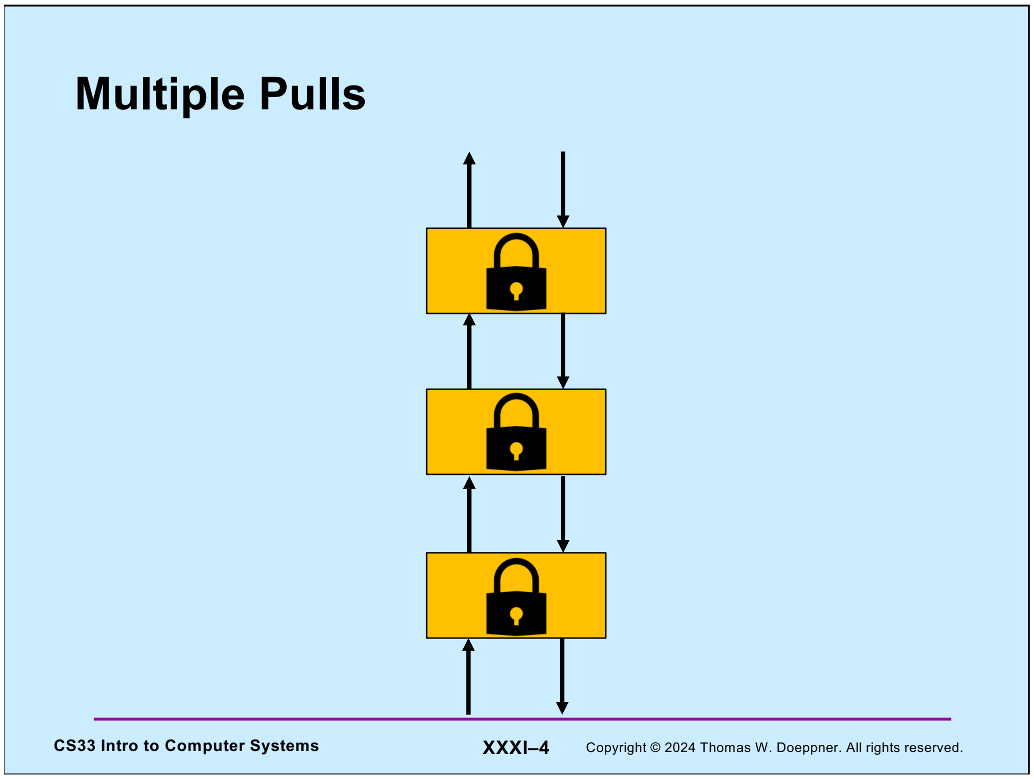
However, we can see an immediate problem: the thread executing this code will not only modify *fbp, but also its predecessor and successor blocks in the freelist.



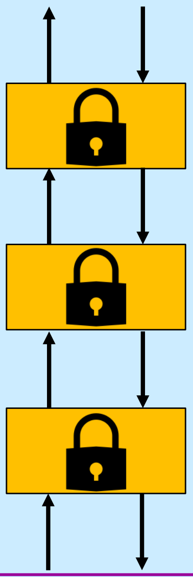
Removing a Freelist Block: Fine Grained (2)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    pthread_mutex_lock(&fpp->blink->mutex);
    fbp->blink->flink = fbp->flink;
    pthread_mutex_lock(&fpp->flink->mutex);
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->blink->mutex);
    pthread_mutex_unlock(&fpp->flink->mutex);
    pthread_mutex_unlock(&fpp->mutex);
}
```

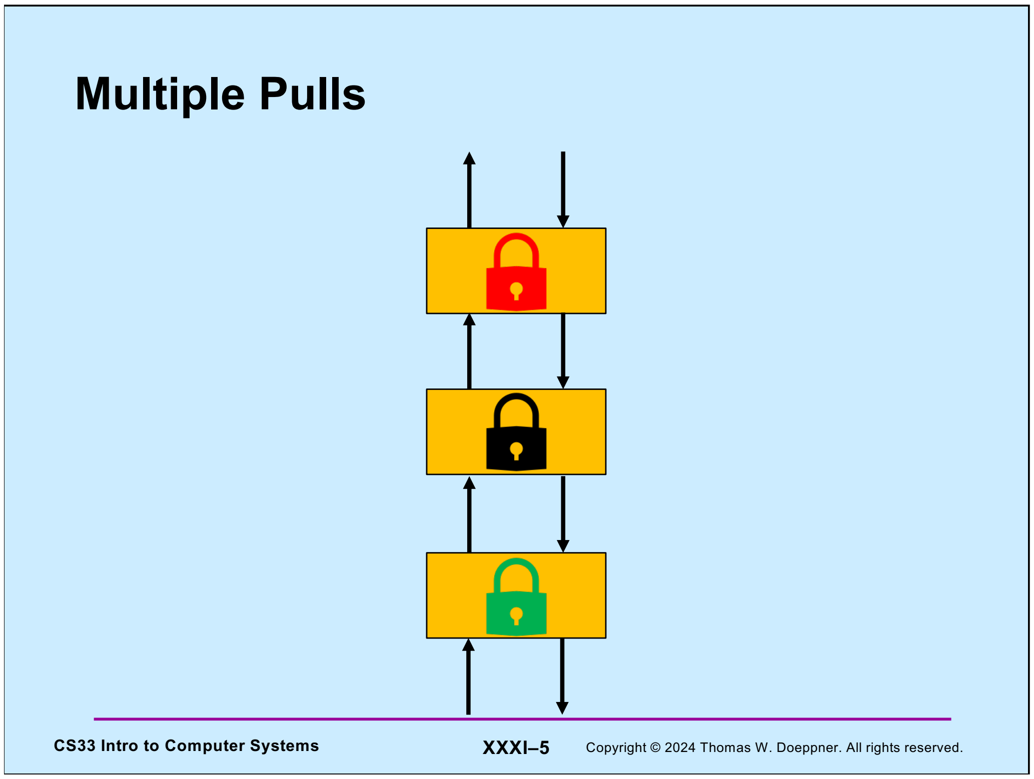
So, we add code to lock and unlock the mutexes of the adjacent blocks.



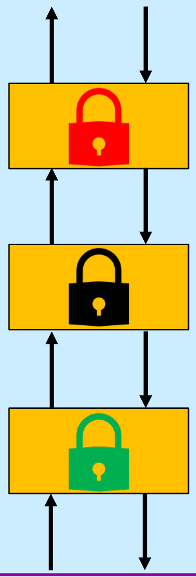
Multiple Pulls



Pictorially, our thread first locks the mutex on *fbp (the middle block), then its predecessor and then its successor.



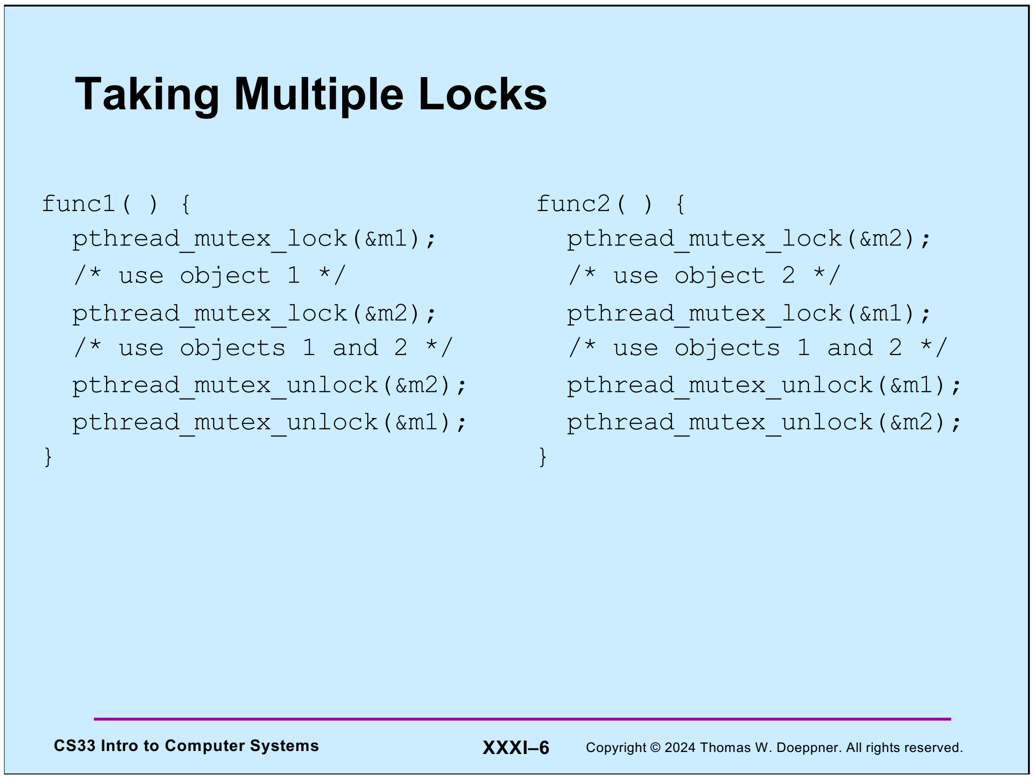
Multiple Pulls



But suppose other threads are calling pull_from_freelist at the same time. Let's say that thread 1 is pulling the middle block, thread 2 is pulling the upper block, and thread 3 is pulling the lower block.

All three threads have locked the mutex on the block they're pulling. Thread one tries to lock the mutex on the upper block (once it successfully locks it, then it will try to lock the mutex on the lower block). However, thread 2 has already locked the mutex on the upper block, and won't unlock it until after it locks the mutex on the middle block. But thread 1 won't unlock it until it locks the mutex on the upper block (not to mention the lower block).

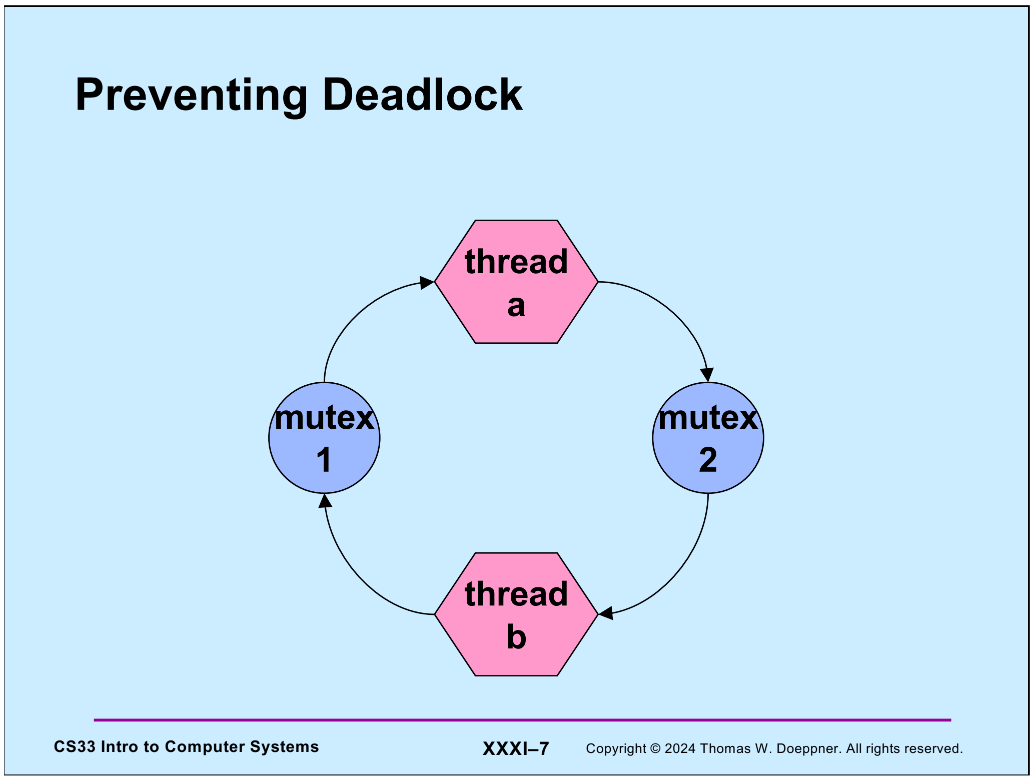
So, we're stuck. threads 1 and 2 (as well as 3) won't ever be able to lock all the mutexes they need, and, because of this, can't make any further progress. This phenomenon is called **deadlock**.



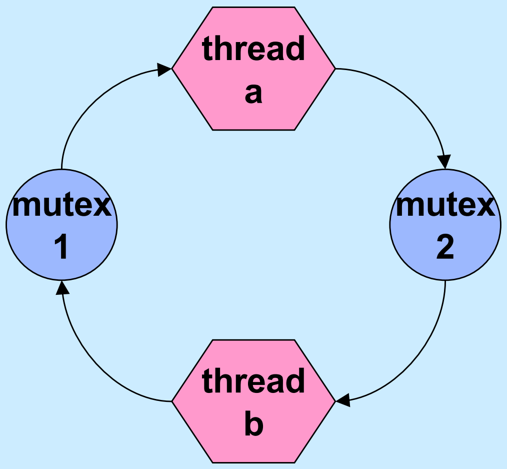
Taking Multiple Locks

```
func1( ) {                          func2( ) {
  pthread_mutex_lock(&m1);            pthread_mutex_lock(&m2);
  /* use object 1 */                  /* use object 2 */
  pthread_mutex_lock(&m2);            pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */           /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);          pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m1);          pthread_mutex_unlock(&m2);
}                                   }
```

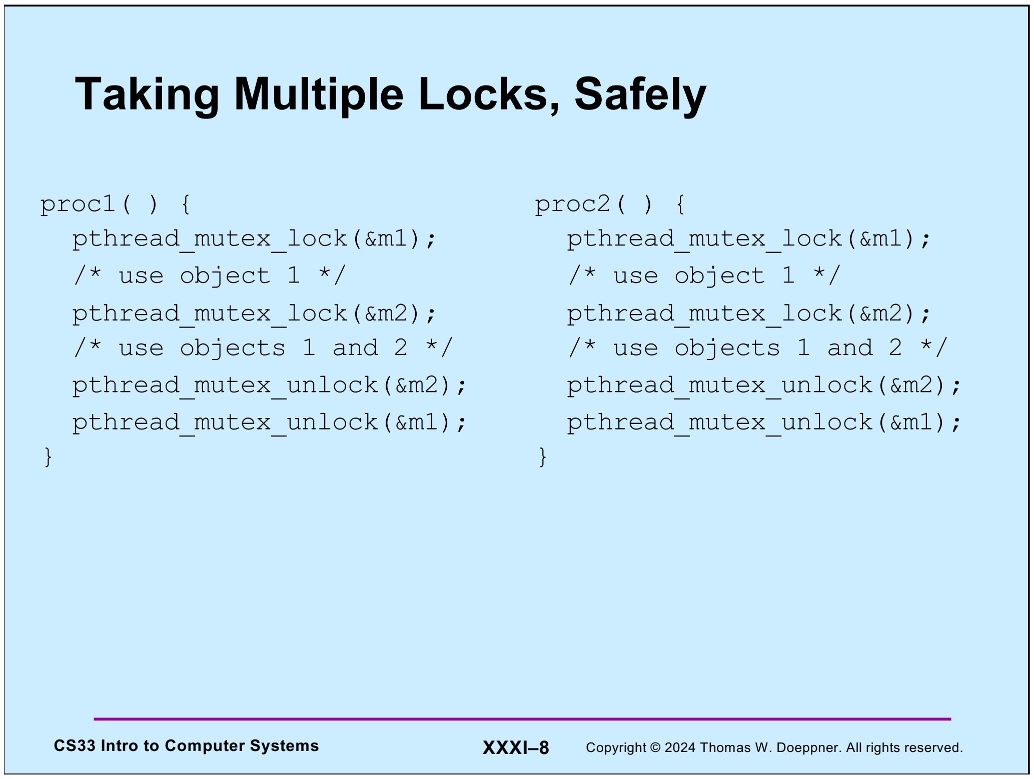
In this example our threads are using two mutexes to control access to two different objects. Thread 1, executing **func1**, first takes mutex 1, then, while still holding mutex 1, obtains mutex 2. Thread 2, executing **func2**, first takes mutex 2, then, while still holding mutex 2, obtains mutex 1. However, things do not always work out as planned. If thread 1 obtains mutex 1 and, at about the same time, thread 2 obtains mutex 2, then if thread 1 attempts to take mutex 2 and thread 2 attempts to take mutex 1, we have a **deadlock**.



Preventing Deadlock



Deadlock results when there are circularities in dependencies. In the slide, mutex 1 is held by thread a, which is waiting to take mutex 2. However, thread b is holding mutex 2, waiting to take mutex 1. If we can make certain that such circularities never happen, there can't possibly be deadlock.



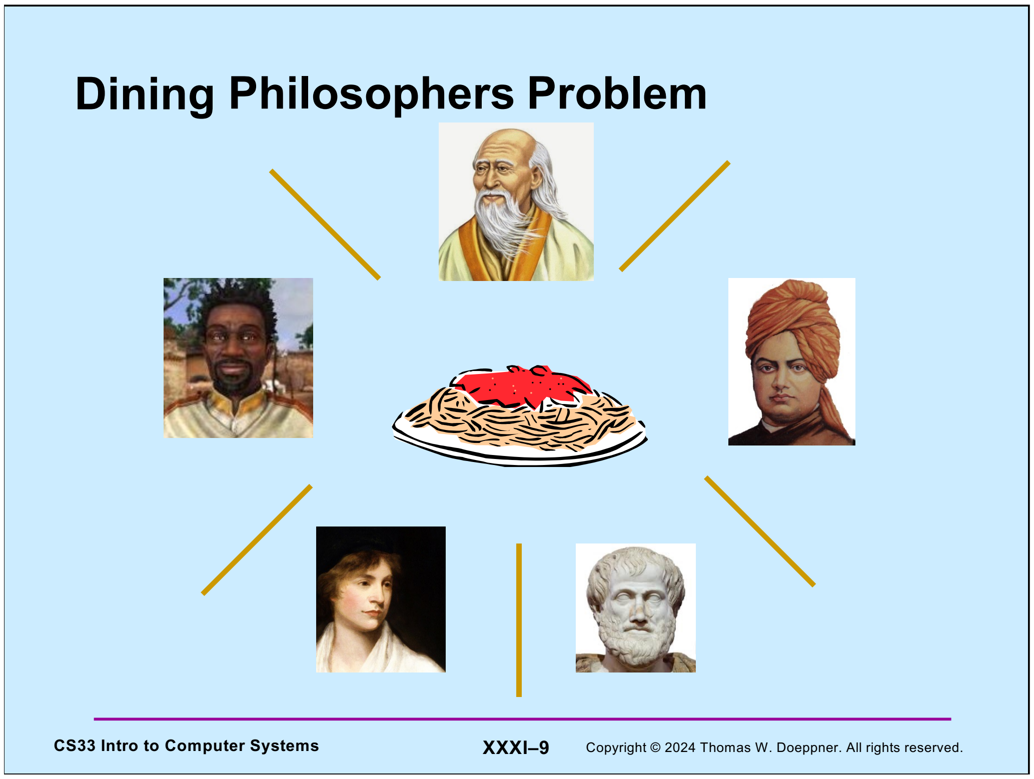
Taking Multiple Locks, Safely

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

If all threads take locks in the same order, deadlock cannot happen.

How can we modify our pull_from_freelist code to use this approach (of all threads taking locks in the same order)?



Dining Philosophers Problem

The problem we've been looking at is a special case of what's known as the "dining philosophers problem", posed by Edsger Dijkstra in EWD310, first published as Hierarchical Ordering of Sequential Processes in Operating Systems Techniques, C.A.R. Hoare and R.H. Perrot, Eds., Academic Press, New York, 1972. The idea is that we have five philosophers sitting around a table. At the center of the table is a plate of spaghetti. Between each pair of philosophers is a single chopstick (Dijkstra's original formulation used forks, but chopsticks make more sense). The algorithm of a philosopher is:

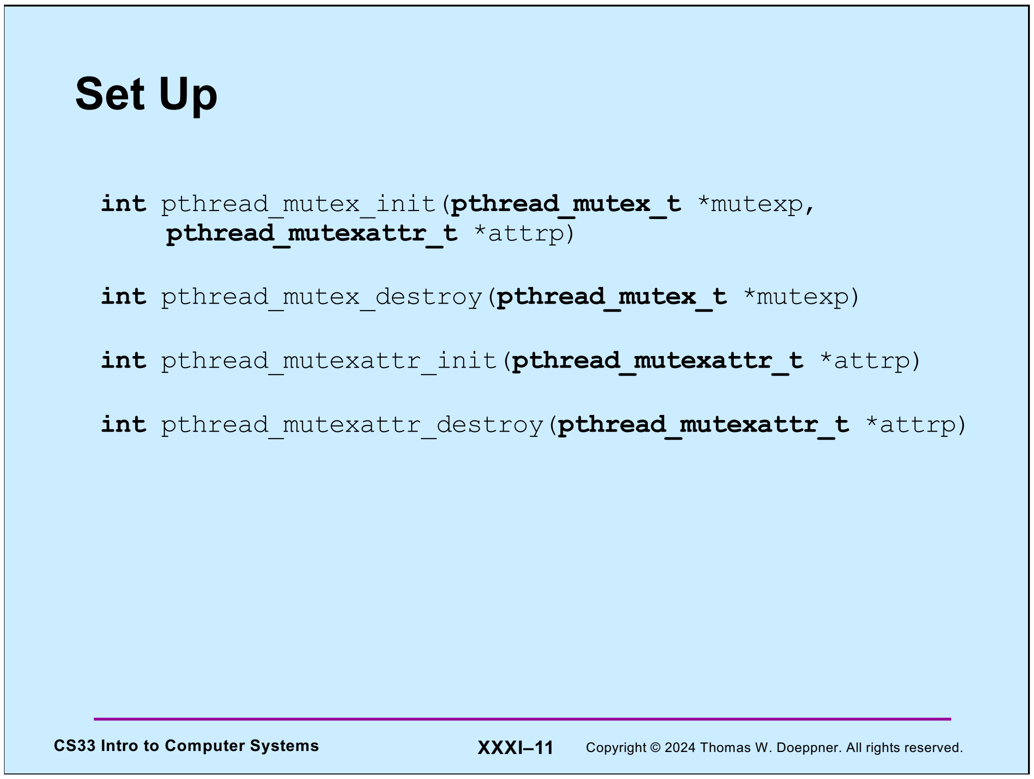
```
while (1) {
  think();
  when available
     grab chopstick from one side();
  when available
      grab chopstick from the other side();
  eat some spaghetti();
  put chopsticks down();
}
```

How long each operation takes varies. Which chopstick is grabbed first is not specified, but if each philosopher grabs their right chopstick first, they may starve to death. There are many subtle issues involved in its solution. (It has many, none of which are as interesting as the problem itself.)

Philosophers clockwise from top: Laozi, Swami Vivekananda, Aristotle, Mary Wollstonecraft, Zara Yacob.

Practical Issues with Mutexes

- Used a lot in multithreaded programs
  - speed is really important
    - shouldn't slow things down much in the success case
  - checking for errors slows things down (a lot)
    - thus errors aren't checked by default

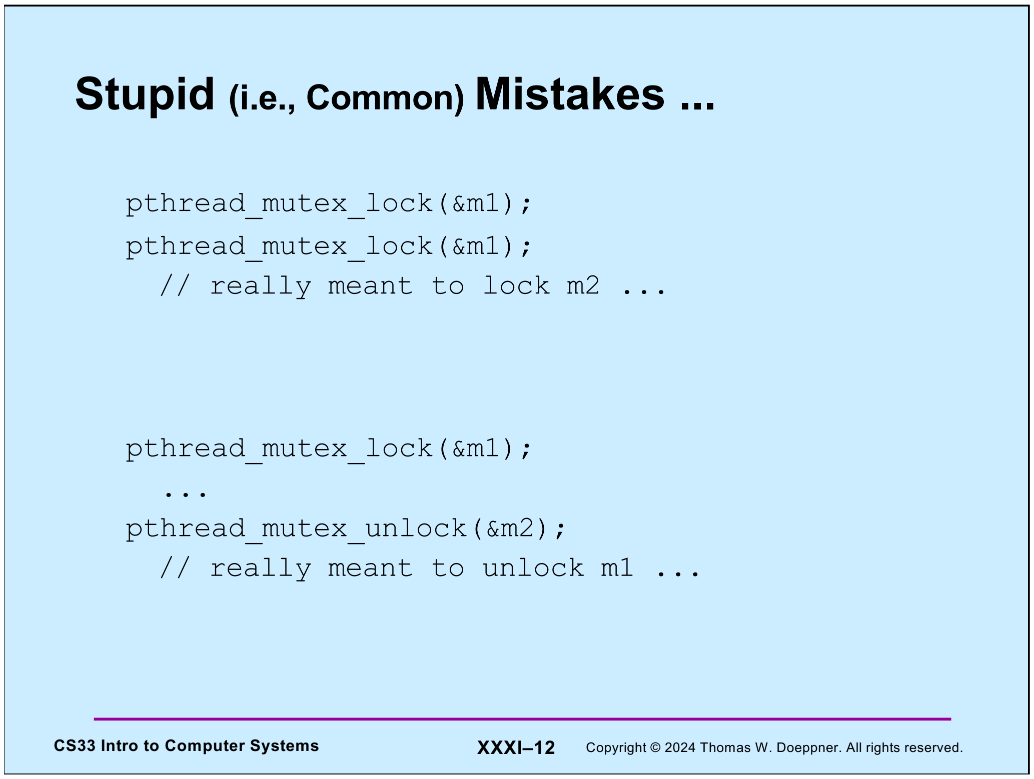


Set Up

```
int pthread_mutex_init(pthread_mutex_t *mutexp,
    pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(pthread_mutex_t *mutexp)

int pthread_mutexattr_init(pthread_mutexattr_t *attrp)

int pthread_mutexattr_destroy(pthread_mutexattr_t *attrp)
```

The functions **pthread_mutex_init** and **pthread_mutex_destroy** are supplied to initialize and to destroy a mutex. (They do not allocate or free the storage for the mutex data structure, but in some implementations they might allocate and free storage referred to by the mutex data structure.) As with threads, an attribute structure encapsulates the various parameters that might apply to the mutex. The functions **pthread_mutexattr_init** and **pthread_mutexattr_destroy** control the initialization and destruction of these attribute structures, as we see a few slides from now. For most purposes, the default attributes are fine and a NULL **attrp** can be provided to the **pthread_mutex_init** routine.

Note that, as we've already seen, a mutex that's allocated statically may be initialized with PTHREAD_MUTEX_INITIALIZER.

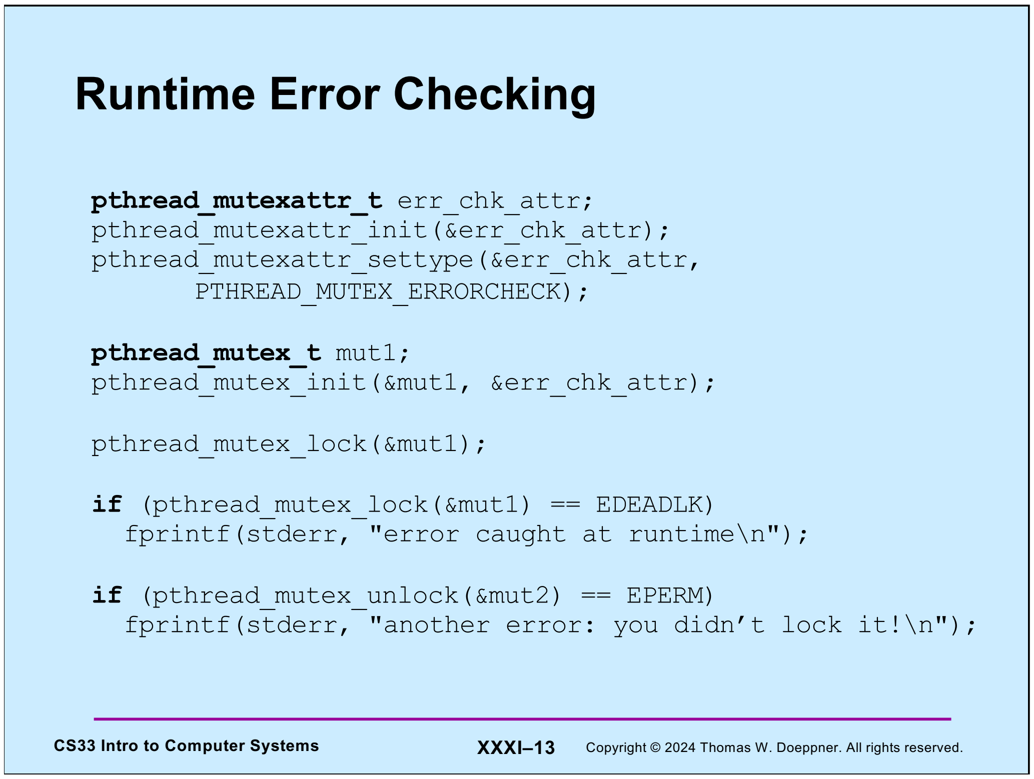


Stupid (i.e., Common) Mistakes ...

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m1);
  // really meant to lock m2 ...



pthread_mutex_lock(&m1);
  ...
pthread_mutex_unlock(&m2);
  // really meant to unlock m1 ...
```

In the example at the top of the slide, we have mistyped the name of the mutex in the second call to **pthread_mutex_lock**. The result will be that when **pthread_mutex_lock** is called for the second time, there will be immediate deadlock, since the caller is attempting to lock a mutex that is already locked, but the only thread who can unlock that mutex is the caller.

In the example at the bottom of the slide, we have again mistyped the name of a mutex, but this time for a **pthread_mutex_unlock** call. If m2 is not currently locked by some thread, unlocking will have unpredictable results, possibly fatal. If m2 is locked by some thread, again there will be unpredictable results, since a mutex that was thought to be locked (and protecting some data structure) is now unlocked. When the thread who locked it attempts to unlock it, the result will be even further unpredictability.

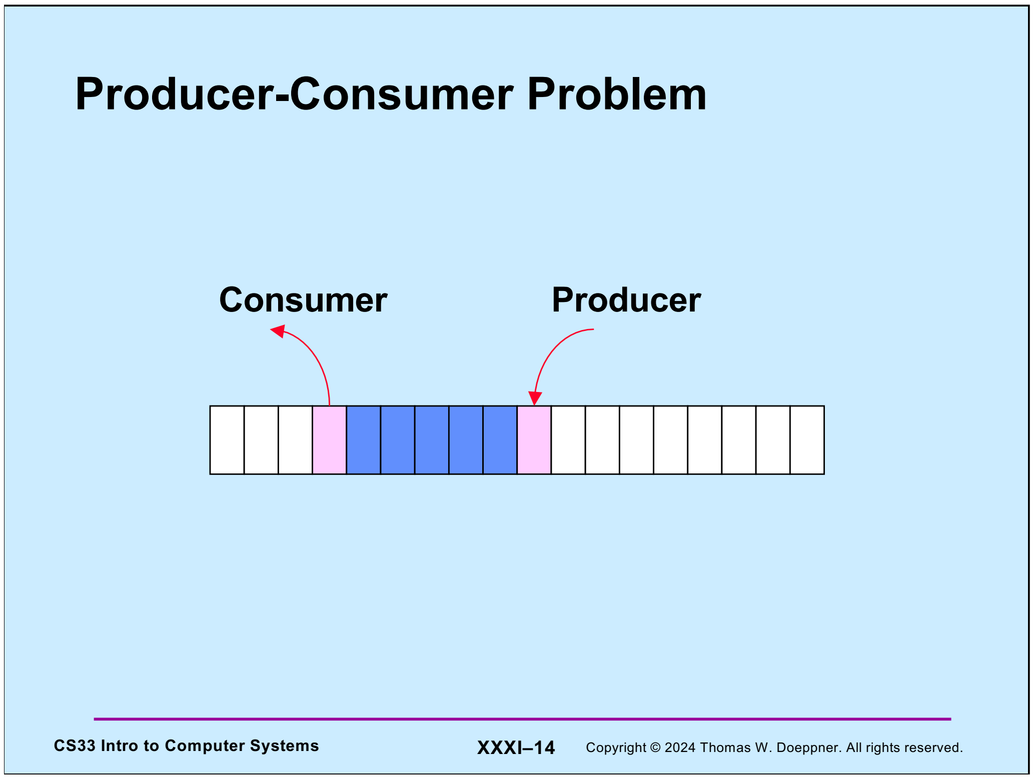


Runtime Error Checking

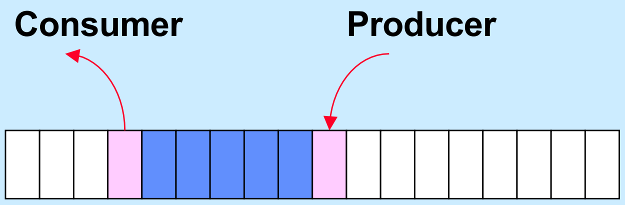
```
pthread_mutexattr_t err_chk_attr;
pthread_mutexattr_init(&err_chk_attr);
pthread_mutexattr_settype(&err_chk_attr,
       PTHREAD_MUTEX_ERRORCHECK);

pthread_mutex_t mut1;
pthread_mutex_init(&mut1, &err_chk_attr);

pthread_mutex_lock(&mut1);

if (pthread_mutex_lock(&mut1) == EDEADLK)
  fprintf(stderr, "error caught at runtime\n");

if (pthread_mutex_unlock(&mut2) == EPERM)
  fprintf(stderr, "another error: you didn’t lock it!\n");
```

Checking for some sorts of mutex-related errors is relatively easy to do at runtime (though checking for all possible forms of deadlock is prohibitively expensive). However, since mutexes are used so frequently, even a little bit of extra overhead for runtime error checking is often thought to be too much. Thus, if done at all, runtime error checking is an optional feature. One “turns on” the feature for a particular mutex by initializing it to be of type “ERRORCHECK,” as shown in the slide. For mutexes initialized in this way, **pthread_mutex_lock** checks to make certain that it is not attempting to lock a mutex that is already locked by the calling thread; **pthread_mutex_unlock** checks to make certain that the mutex being unlocked is currently locked by the calling thread.

Note that mutexes with the error-check attribute are more expensive than normal mutexes, since they must keep track of which thread, if any, has the mutex locked. (For normal mutexes, just a single bit must be maintained for the state of the mutex, which is either locked or unlocked.)

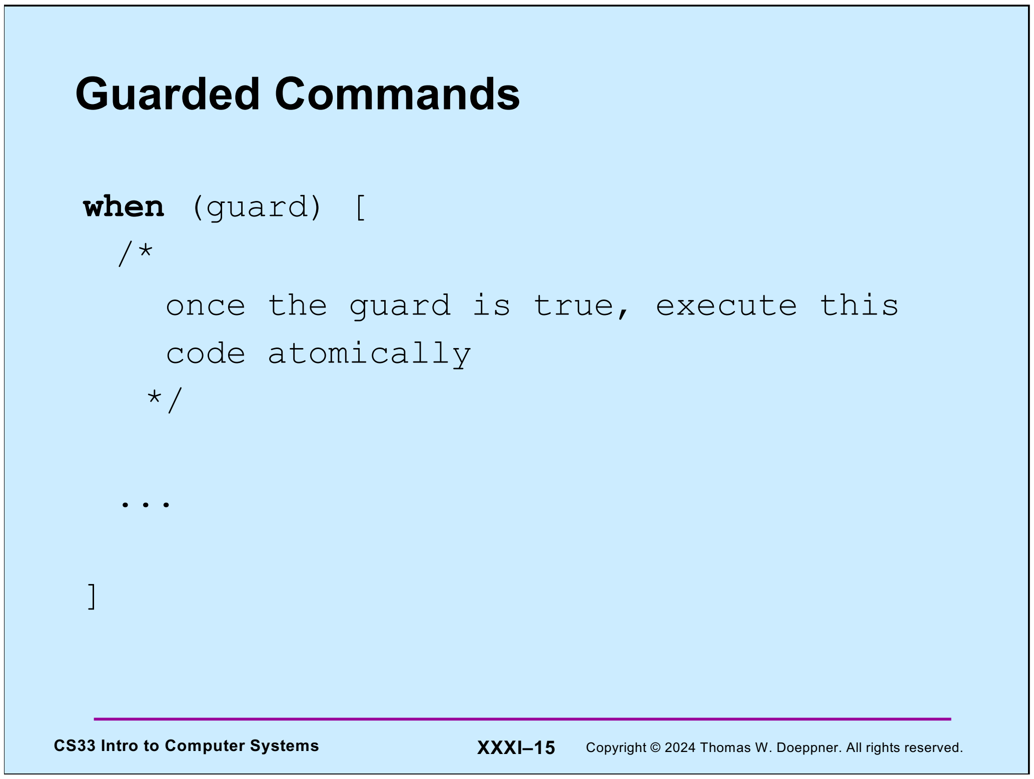


Producer-Consumer Problem



In the **producer-consumer problem** we have two classes of threads, producers and consumers, and a buffer containing a fixed number of slots. A producer thread attempts to put something into the next empty buffer slot, a consumer thread attempts to take something out of the next occupied buffer slot. The synchronization conditions are that producers cannot proceed unless there are empty slots and consumers cannot proceed unless there are occupied slots.

This is a classic, but frequently occurring synchronization problem. For example, the heart of the implementation of UNIX pipes is an instance of this problem.

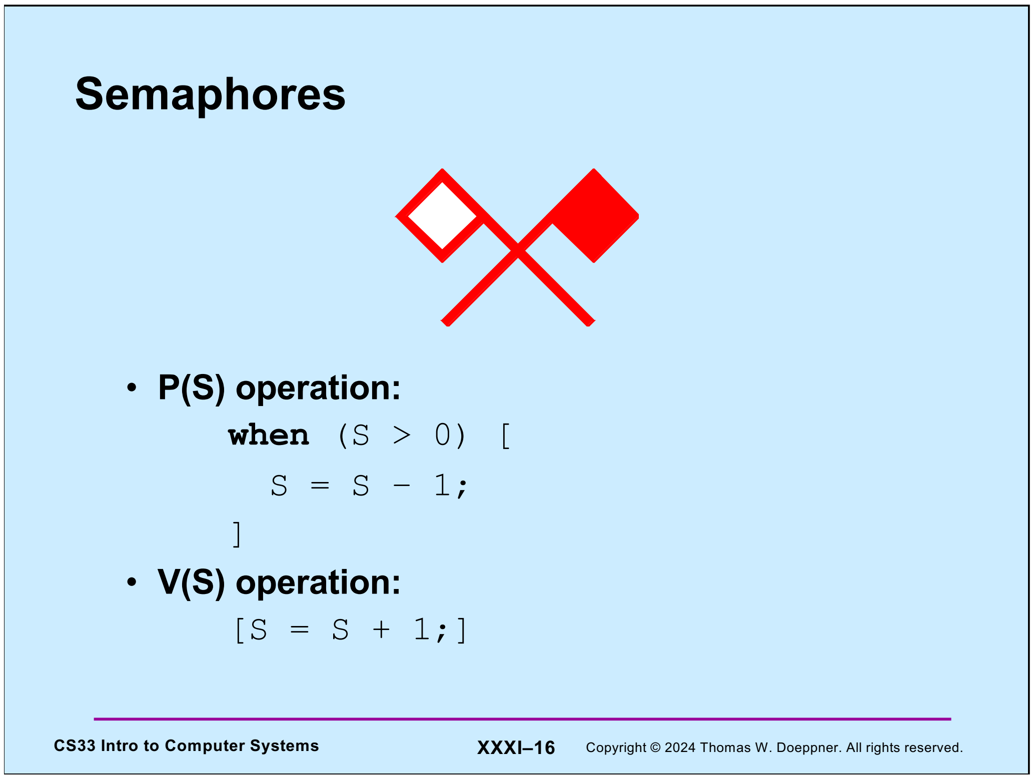


Guarded Commands

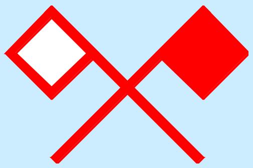
```
when (guard) [
  /*
     once the guard is true, execute this
     code atomically
    */

  ...

]
```

Illustrated in the slide is a simple pseudocode construct, the **guarded command**, that we use to describe how various synchronization operations work. The idea is that the code within the square brackets is executed only when the guard (which could be some arbitrary boolean expression) evaluates to true. Furthermore, this code within the square brackets is executed atomically, i.e., the effect is that nothing else happens in the program while the code is executed. Note that the code is not necessarily executed as soon as the guard evaluates to true: we are assured only that when execution of the code begins, the guard is true.

Keep in mind that this is strictly pseudocode: it's not part of POSIX threads and is not necessarily even implementable (at least not for the general case).



Semaphores



- **P(S) operation:**

```
when (S > 0) [
  S = S - 1;
]
```

- **V(S) operation:**

```
[S = S + 1;]
```

Another synchronization construct is the semaphore, designed by Edsger Dijkstra in the 1960s. A semaphore behaves as if it were a nonnegative integer, but it can be operated on only by the semaphore operations. Dijkstra defined two of these: P (for **prolagen**, a made-up word derived from **proberen te verlagen**, which means “try to decrease” in Dutch) and V (for **verhogen**, “increase” in Dutch). Their semantics are shown in the slide.

We think of operations on semaphores as being a special case of guarded commands — a special case that occurs frequently enough to warrant a highly optimized implementation.

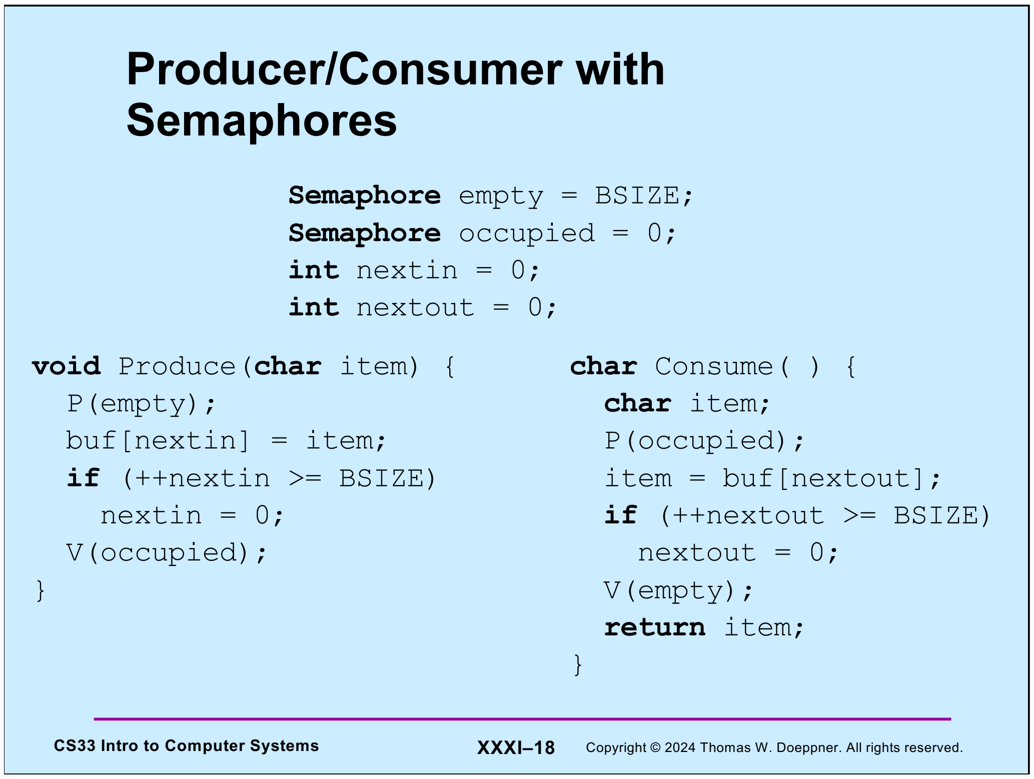
Quiz 1

```
semaphore S = 1;
int count = 0;

void func( ) {
  P(S);
  count++;
  ...
  count--;
  V(S);
}
```

The function func is called concurrently by n threads. What's the maximum value that count will take on?

a) indeterminate
b) 1
c) 2
d) n

- **P(S) operation:**
```
      when (S > 0) [
        S = S - 1;
      ]
```
- **V(S) operation:**
```
      [S = S + 1;]
```



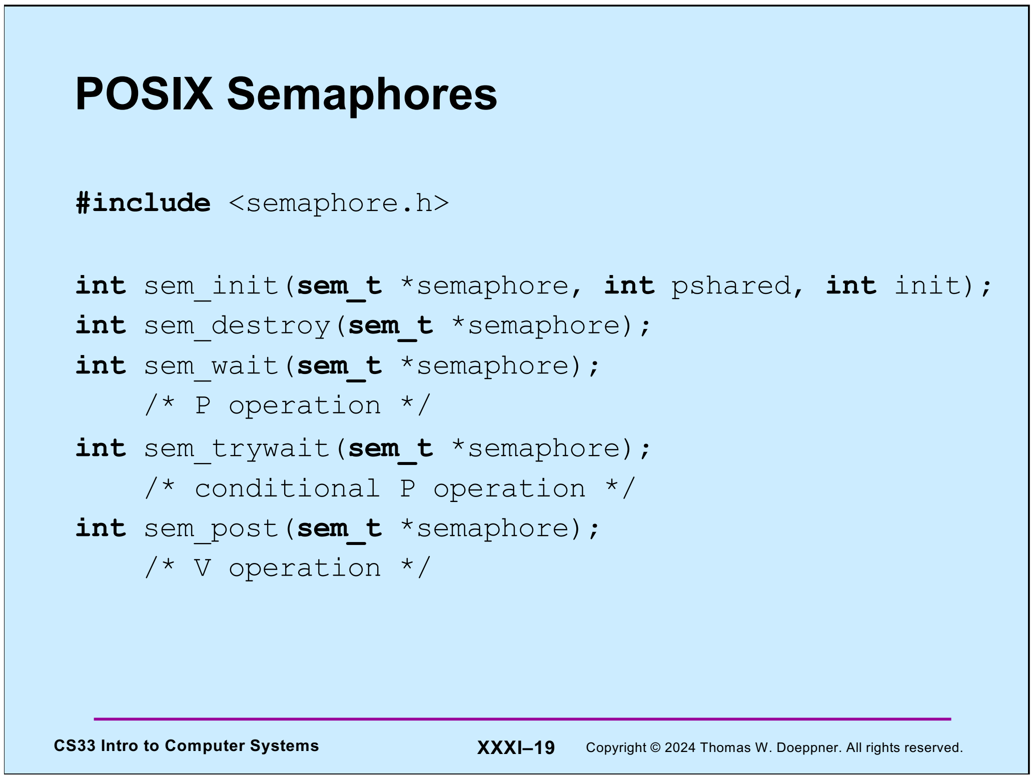
Producer/Consumer with Semaphores

```
Semaphore empty = BSIZE;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  if (++nextin >= BSIZE)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  if (++nextout >= BSIZE)
    nextout = 0;
  V(empty);
  return item;
}
```

Here’s a solution for the producer/consumer problem using semaphores — note that it works only with a single producer and a single consumer, and only one item at a time is produced or consumed, though it can be generalized to work with multiple producers and consumers.

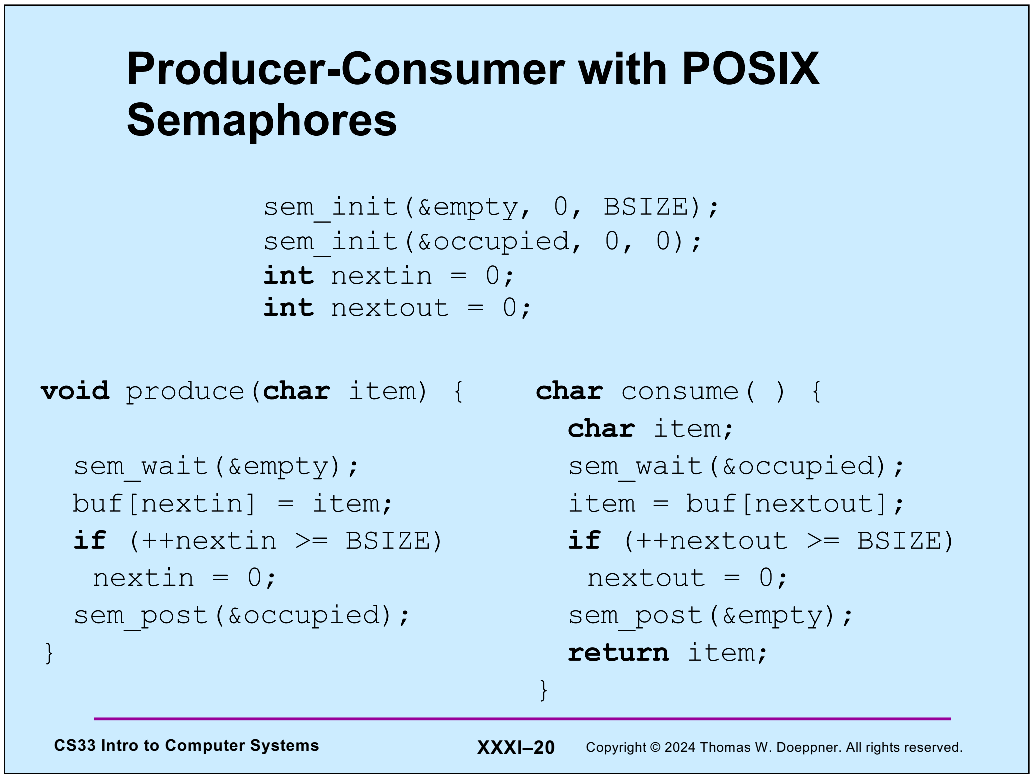


POSIX Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *semaphore, int pshared, int init);
int sem_destroy(sem_t *semaphore);
int sem_wait(sem_t *semaphore);
    /* P operation */
int sem_trywait(sem_t *semaphore);
    /* conditional P operation */
int sem_post(sem_t *semaphore);
    /* V operation */
```

Here is the POSIX interface for operations on semaphores. (These operation names are not typos — the "pthread_" prefix really is not used here, since the semaphore operations come from a different POSIX specification — 1003.1b. Note also the need for the header file, **semaphore.h**) When creating a semaphore (**sem_init**), rather than supplying an attributes structure, one supplies a single integer argument, **pshared**, which indicates whether the semaphore is to be used only by threads of one process (**pshared = 0**) or by multiple processes (**pshared = 1**). The third argument to **sem_init** is the semaphore's initial value.

All the semaphore operations return zero if successful; otherwise, they return an error code. The function **sem_trywait** is similar to **sem_wait** (and to the P operation) except that if the semaphore's value cannot be decremented immediately, then rather than wait, it returns -1 and sets errno to EAGAIN.



Producer-Consumer with POSIX Semaphores

```
            sem_init(&empty, 0, BSIZE);
            sem_init(&occupied, 0, 0);
            int nextin = 0;
            int nextout = 0;

void produce(char item) {     char consume( ) {
                                char item;
  sem_wait(&empty);             sem_wait(&occupied);
  buf[nextin] = item;           item = buf[nextout];
  if (++nextin >= BSIZE)        if (++nextout >= BSIZE)
   nextin = 0;                   nextout = 0;
  sem_post(&occupied);          sem_post(&empty);
}                               return item;
                              }
```

Here is the producer-consumer solution implemented with POSIX semaphores.

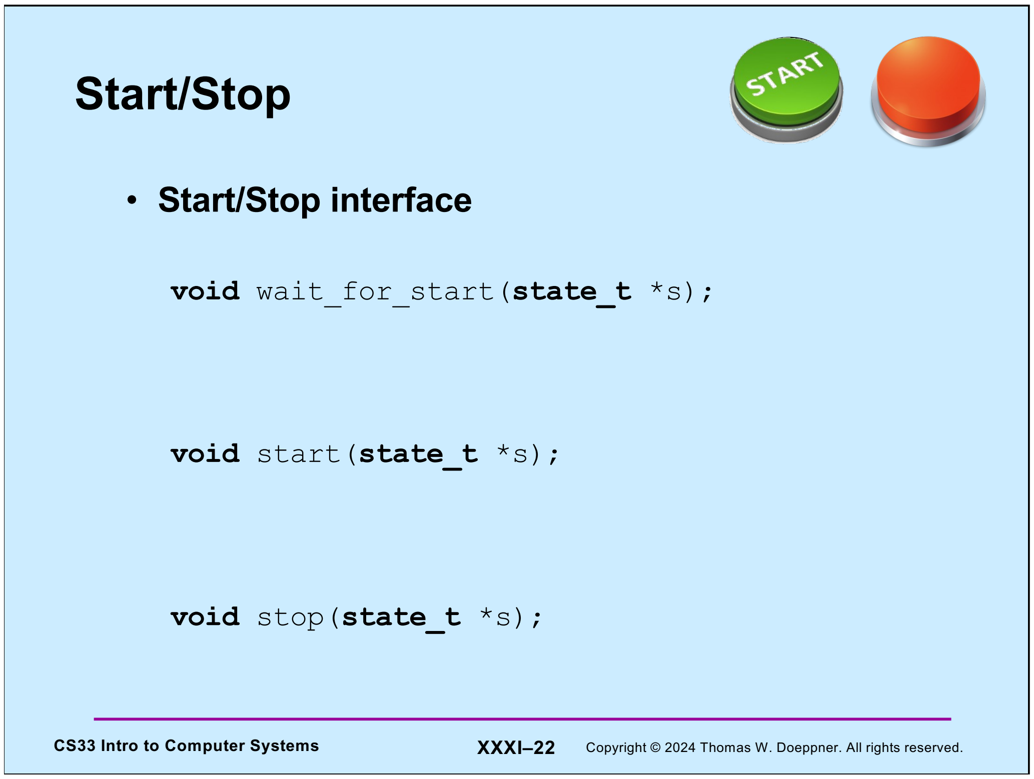
Quiz 2

Does the POSIX version of the producer-consumer solution work with multiple producers and consumers?

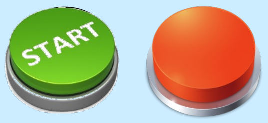
a) It can't easily be made to work

b) Yes

c) No, but it can be made to work by using mutexes to make sure that only one thread is executing the producer code at a time and only one thread is executing the consumer code at a time



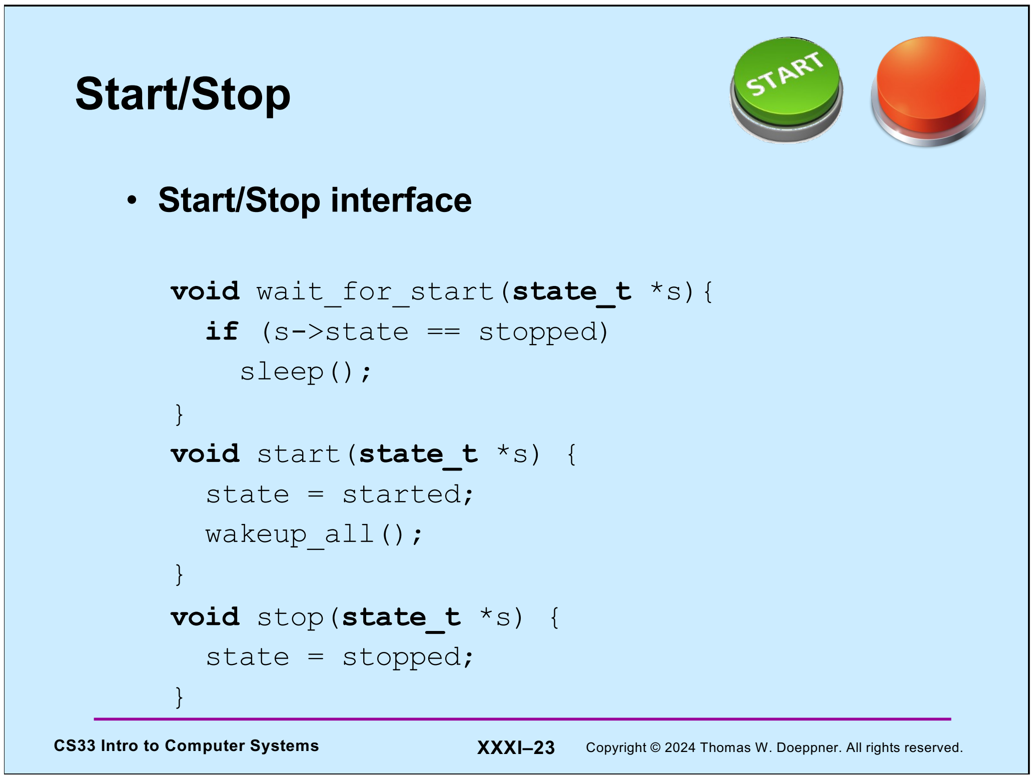
Start/Stop



- **Start/Stop interface**

```
void wait_for_start(state_t *s);



void start(state_t *s);



void stop(state_t *s);
```

We'd like to design a "start-stop" interface. A thread calling **wait_for_start** waits for the **start** button to be pressed. Once it's been pressed, those waiting will be released and subsequent threads calling **wait_for_start** will return immediately. However, once the **stop** button is pressed, then all threads calling **wait_for_start** will wait until the **start** button is pressed again.



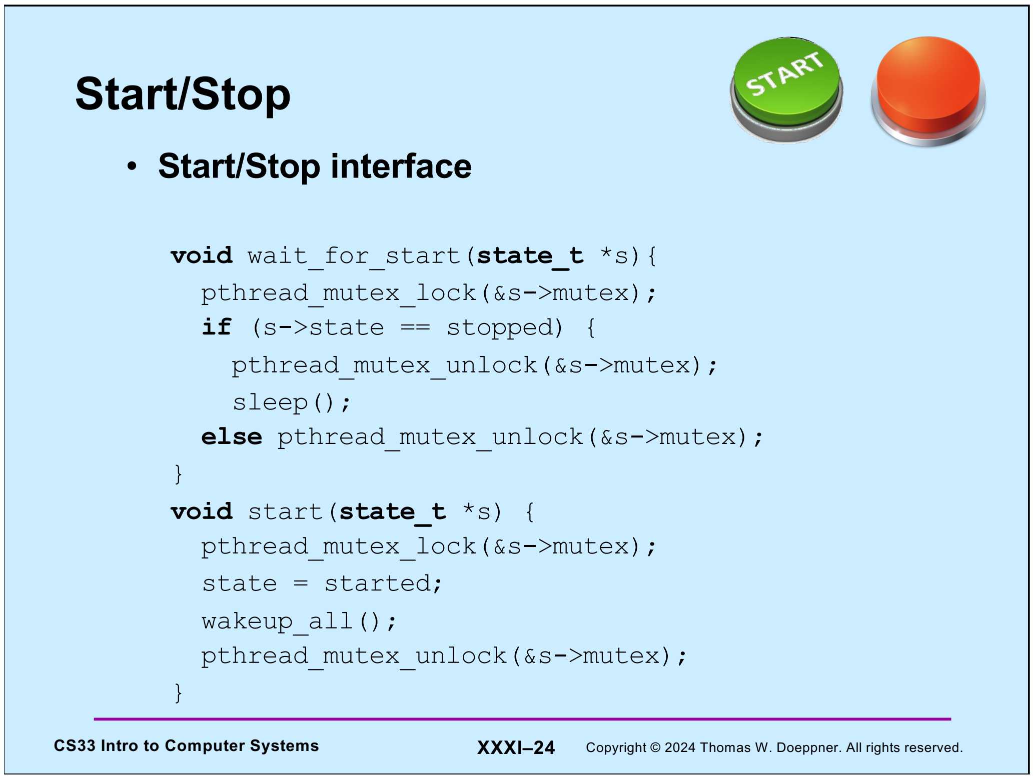
Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  if (s->state == stopped)
    sleep();
}
void start(state_t *s) {
  state = started;
  wakeup_all();
}
void stop(state_t *s) {
  state = stopped;
}
```

Here's a possible implementation. Callers of **sleep** don't return from sleep until **wakeup_all** has been called.

However, calls to **wakeup_all** merely wakeup all who are currently in **sleep**. They have no effect on subsequent calls to sleep. Thus, there could be a problem in the above code if a thread calls start while another thread has just checked the state in **wait_for_start**, but hasn't yet called **sleep**.

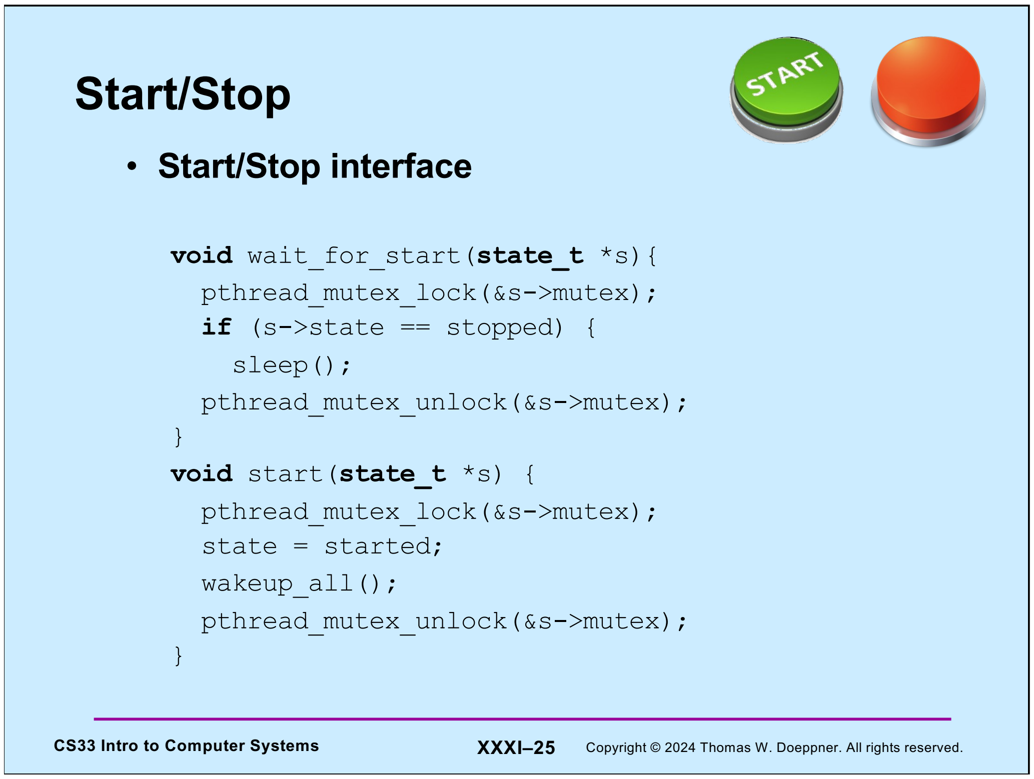


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  if (s->state == stopped) {
    pthread_mutex_unlock(&s->mutex);
    sleep();
  else pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  state = started;
  wakeup_all();
  pthread_mutex_unlock(&s->mutex);
}
```

Here's one attempt to fix the problem of the previous slide using mutexes. It clearly doesn't help – the thread calling start might get the mutex and call wakeup_all just before the other thread calls sleep.

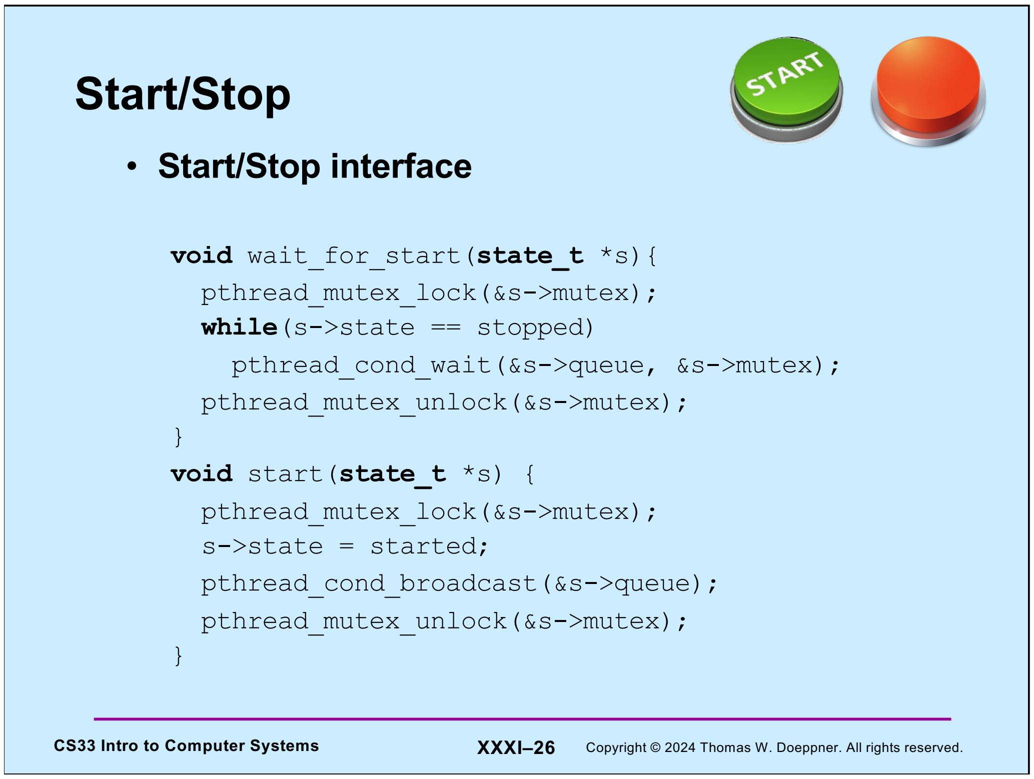


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  if (s->state == stopped) {
    sleep();
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  state = started;
  wakeup_all();
  pthread_mutex_unlock(&s->mutex);
}
```

This code is perhaps worse, the thread waits in sleep with the mutex locked, preventing any thread from calling wakeup_all.



Start/Stop

- **Start/Stop interface**

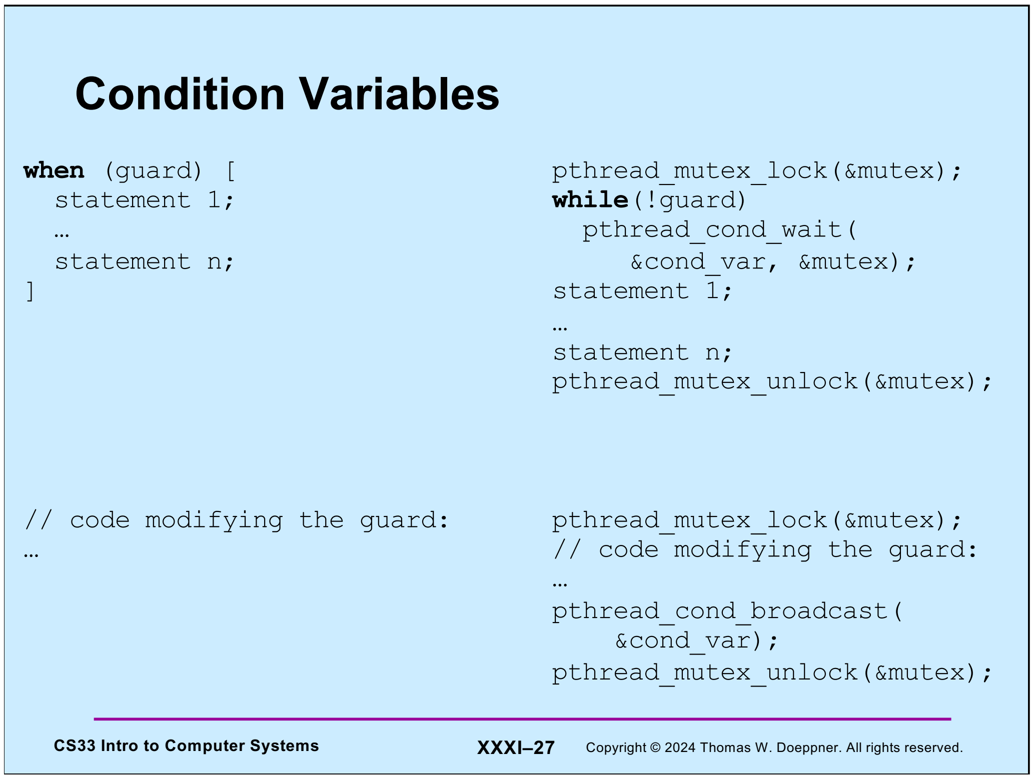
```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

This code actually works; it uses a POSIX threads construct known as the **condition variable**. The thread in wait_for_start first locks the mutex, then checks the state. If it's stopped, it calls **pthread_cond_wait**, which, all at once, put the calling thread to sleep, enqueues it on the queue (known as a condition variable, and unlocks the mutex.

A thread calling start can't proceed until it has locked the mutex, thus ensuring that no thread is in the midst of checking the state and then calling **pthread_cond_wait** in wait_for_start. Once the thread calling start has the mutex, it sets state to started and calls **pthread_broadcast**, waking up all threads who are waiting on the queue (the condition variable). It then unlocks the mutex.

The thread that was waiting within **pthread_cond_wait** is woken up, but it doesn't return from the call to **pthread_cond_wait** until it locks the mutex. Thus, it enters **pthread_cond_wait** with the mutex locked and exits it with the mutex_locked. While its inside **pthread_cond_wait**, it does not have the lock on the mutex (though some other thread might).

Thus, this code is a correct implementation of the start/stop interface.



Condition Variables

```
when (guard) [
  statement 1;
  …
  statement n;
]
```

```
pthread_mutex_lock(&mutex);
while(!guard)
  pthread_cond_wait(
     &cond_var, &mutex);
statement 1;
…
statement n;
pthread_mutex_unlock(&mutex);
```

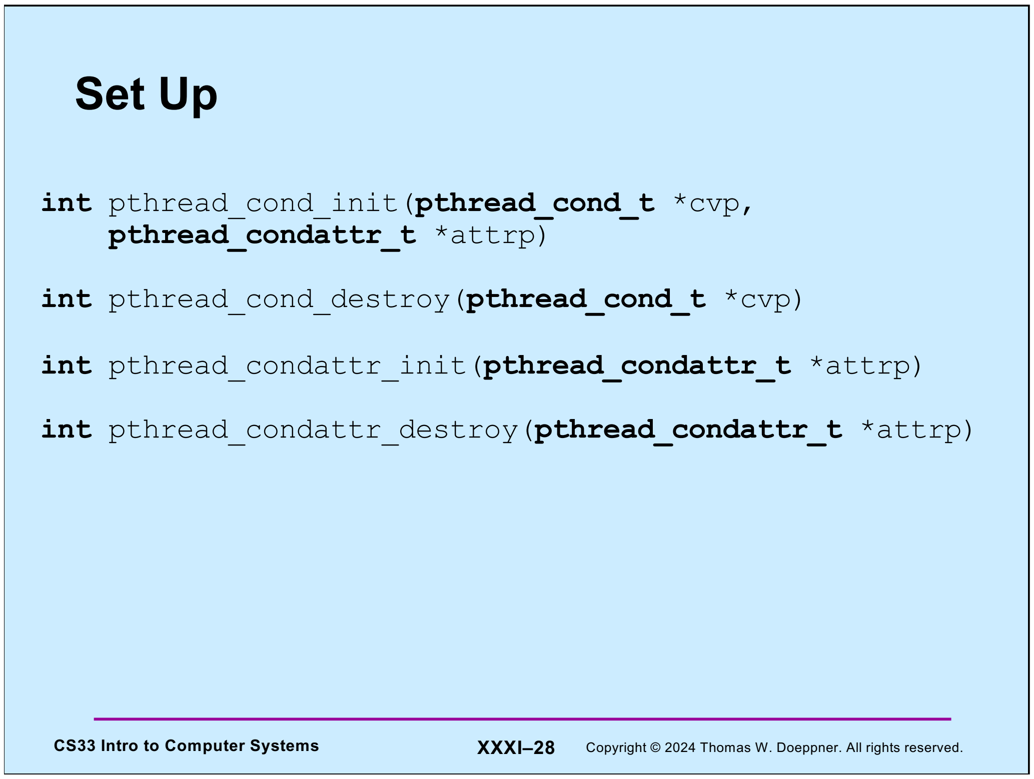
```
// code modifying the guard:
…
```

```
pthread_mutex_lock(&mutex);
// code modifying the guard:
…
pthread_cond_broadcast(
    &cond_var);
pthread_mutex_unlock(&mutex);
```

Condition variables are another means for synchronization in POSIX; they represent queues of threads waiting to be woken by other threads and can be used to implement guarded commands, as shown in the slide. Though they are rather complicated at first glance, they are even more complicated when you really get into them.

A thread puts itself to sleep and joins the queue of threads associated with a condition variable by calling **pthread_cond_wait**. When it places this call, it must have some mutex locked, and it passes the mutex as the second argument. As part of the call, the mutex is unlocked and the thread is put to sleep, **all in a single atomic step**: i.e., nothing can happen that might affect the thread between the moments when the mutex is unlocked and when the thread goes to sleep. Threads queued on a condition variable are released in first-in-first-out order. They are released in response to calls to **pthread_cond_signal** (which releases the first thread in line) and **pthread_cond_broadcast** (which releases all threads). However, before a released thread may return from **pthread_cond_wait**, it first relocks the mutex. Thus, only one thread at a time actually returns from **pthread_cond_wait**. If a call to either function is made when no threads are queued on the condition variable, nothing happens — the fact that a call had been made is not remembered.

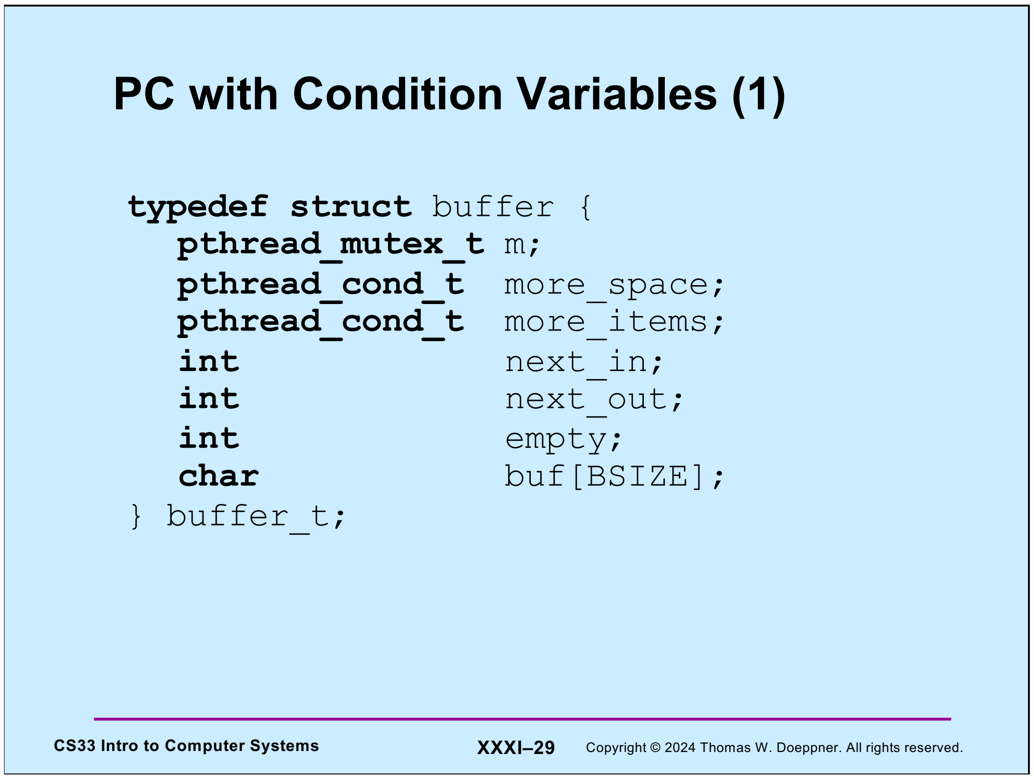
So far, though complicated, the description is rational. Now for the weird part: **a thread may be released from the condition-variable queue at any moment,** perhaps spontaneously, perhaps due to sun spots. Thus, it's extremely important that, after **pthread_cond_wait** returns, that the caller check to make sure that it really should have returned. The reason for this weirdness is that it allows a fair amount of latitude in implementations. However, the Linux implementation behaves rationally, i.e., as in the first two paragraphs. (But don’t depend on this behavior — it could change tomorrow!)



Set Up

```
int pthread_cond_init(pthread_cond_t *cvp,
    pthread_condattr_t *attrp)

int pthread_cond_destroy(pthread_cond_t *cvp)

int pthread_condattr_init(pthread_condattr_t *attrp)

int pthread_condattr_destroy(pthread_condattr_t *attrp)
```

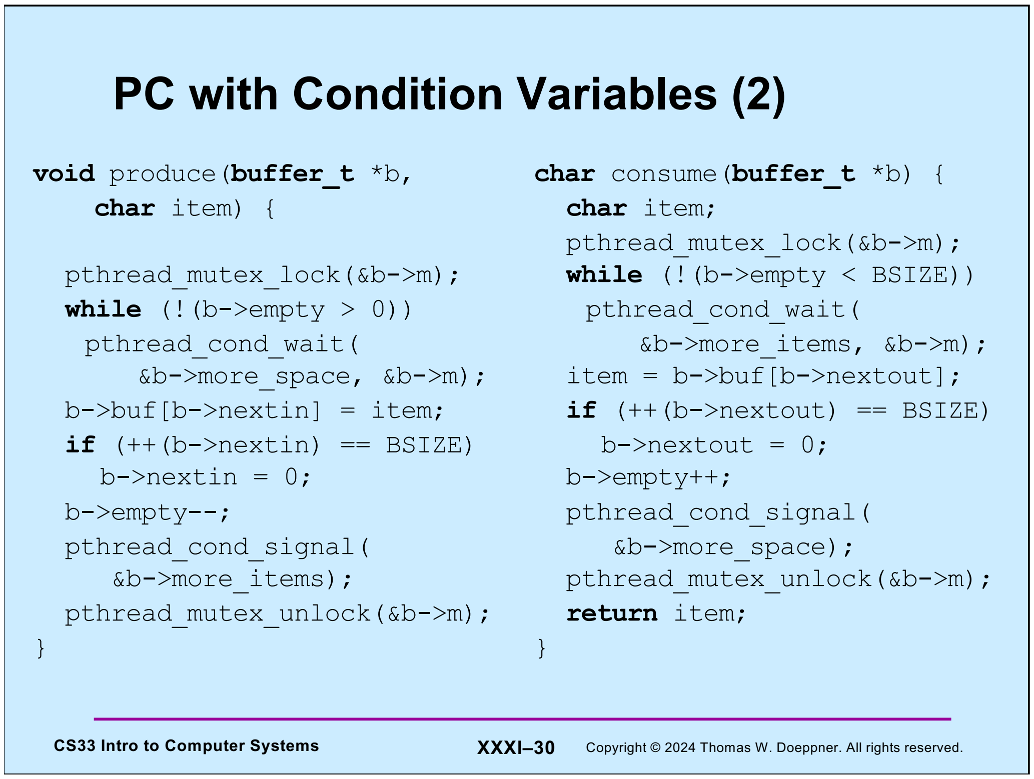
Setting up condition variables is done in a similar fashion as mutexes: The functions **pthread_cond_init** and **pthread_cond_destroy** are supplied to initialize and to destroy a condition variable. They may also be statically initialized by setting them to PTHREAD_COND_INITIALIZER in their declarations. As with mutexes and threads, default attributes may be specified by supplying a zero. The functions **pthread_condattr_init** and **pthread_condattr_destroy** control the initialization and destruction of their attribute structures.



PC with Condition Variables (1)

```
typedef struct buffer {
  pthread_mutex_t m;
  pthread_cond_t  more_space;
  pthread_cond_t  more_items;
  int             next_in;
  int             next_out;
  int             empty;
  char            buf[BSIZE];
} buffer_t;
```

Here we begin a producer-consumer solution using condition variables and mutexes; this solution, unlike the previous, allows multiple producers and consumers. We define a struct **buffer** to represent a buffer, associated synchronization variables, and other associated variables. In our example, producers wait for empty slots to become available, and consumers wait for occupied slots to become available. Waiting producers are queued on the condition variable **more_space** and waiting consumers are queued on the condition variable **more_items**.



PC with Condition Variables (2)

```
void produce(buffer_t *b,
    char item) {

  pthread_mutex_lock(&b->m);
  while (!(b->empty > 0))
    pthread_cond_wait(
        &b->more_space, &b->m);
  b->buf[b->nextin] = item;
  if (++(b->nextin) == BSIZE)
    b->nextin = 0;
  b->empty--;
  pthread_cond_signal(
     &b->more_items);
  pthread_mutex_unlock(&b->m);
}
```

```
char consume(buffer_t *b) {
  char item;
  pthread_mutex_lock(&b->m);
  while (!(b->empty < BSIZE))
    pthread_cond_wait(
        &b->more_items, &b->m);
  item = b->buf[b->nextout];
  if (++(b->nextout) == BSIZE)
    b->nextout = 0;
  b->empty++;
  pthread_cond_signal(
     &b->more_space);
  pthread_mutex_unlock(&b->m);
  return item;
}
```

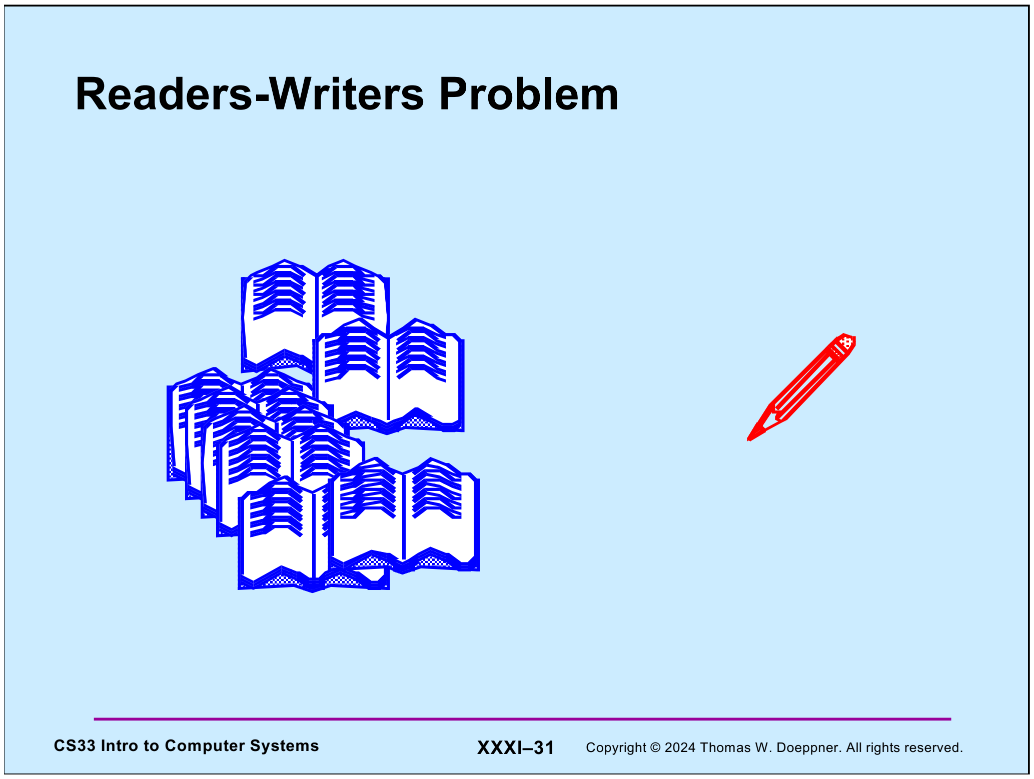
Here we have the remaining code of our solution. A producer, if there is at least one empty slot, fills the one at location **nextin**, increments **nextin** (taking wraparound into account), calls **pthread_cond_signal** to notify the first waiting consumer that there is now an occupied slot in the buffer, and releases the mutex. If there are no empty slots in the buffer, the producer calls **pthread_cond_wait** to wait for one.

As discussed previously, this call has a fairly complicated effect: it releases the mutex given as the second argument and puts its caller to sleep, after queuing it on the condition variable given as the first argument. At some point in the future, a consumer should call **pthread_cond_signal**, with **more_space** as the argument.

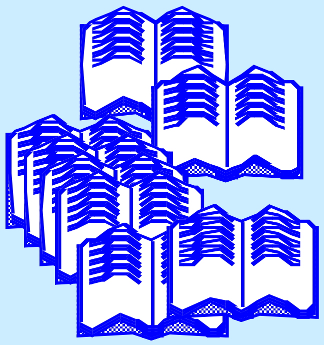
Note that we've used **pthread_cond_signal** rather than **pthread_cond_broadcast**. We can do this here since, if, for example, **n** threads are waiting within the call to **pthread_cond_wait** in the producer, then there must be **n** calls to **consume** to release them all. If we'd used **pthread_cond_broadcast** instead, the solution would still work, but would probably be less efficient, since in many cases waiting threads would return from **pthread_cond_wait**, discover that the guard is still false, and have to call **pthread_cond_wait** again.

If our producer is the first in the queue associated with **more_space**, it is released from the queue, but it does not yet return from **pthread_cond_wait**. Instead, it continues execution inside that routine, where it effectively makes a call to **pthread_mutex_lock** to reacquire the mutex it had when it entered **pthread_cond_wait** in the first place. Once it obtains the mutex, it then returns from **pthread_cond_wait**. Note that when the thread attempts to reacquire the mutex, other threads might be waiting for the mutex at the entrance of the producer code. One of these other threads might obtain the mutex first — thus there is no guarantee that callers of produce are served in FIFO order.

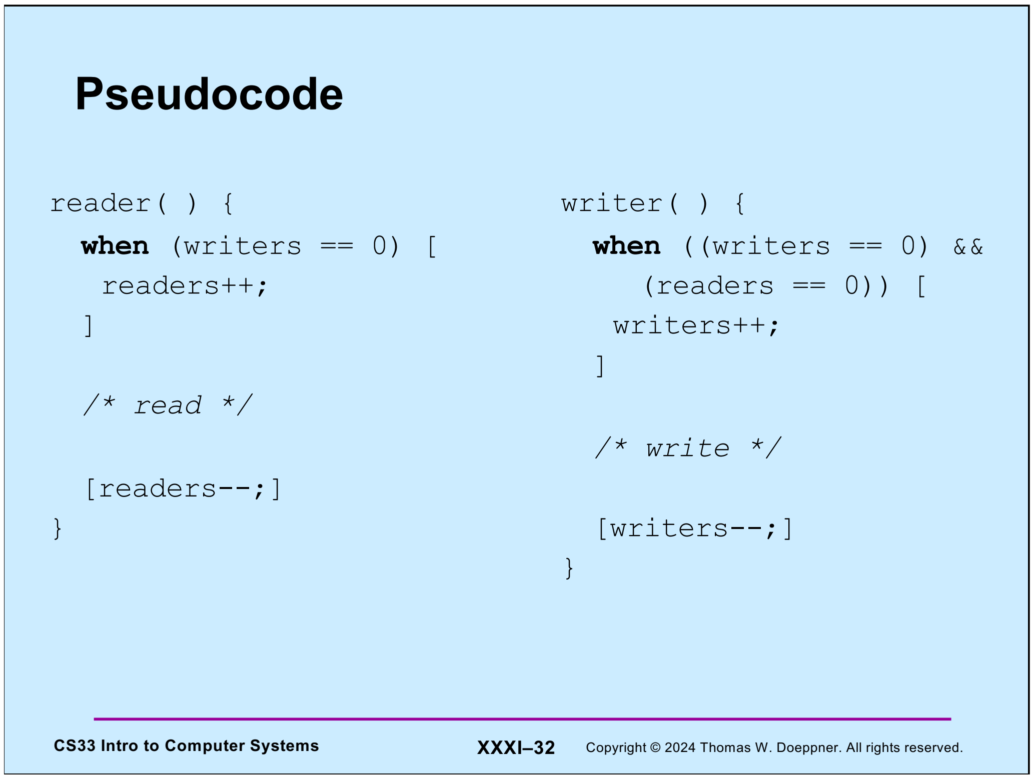
The order in which threads are released from a condition variable's queue is first-in-first-out within priority levels. Thus, waiting high-priority threads are released before waiting low-priority threads; threads of the same priority are released in the order in which they called **pthread_cond_wait**.



Readers-Writers Problem



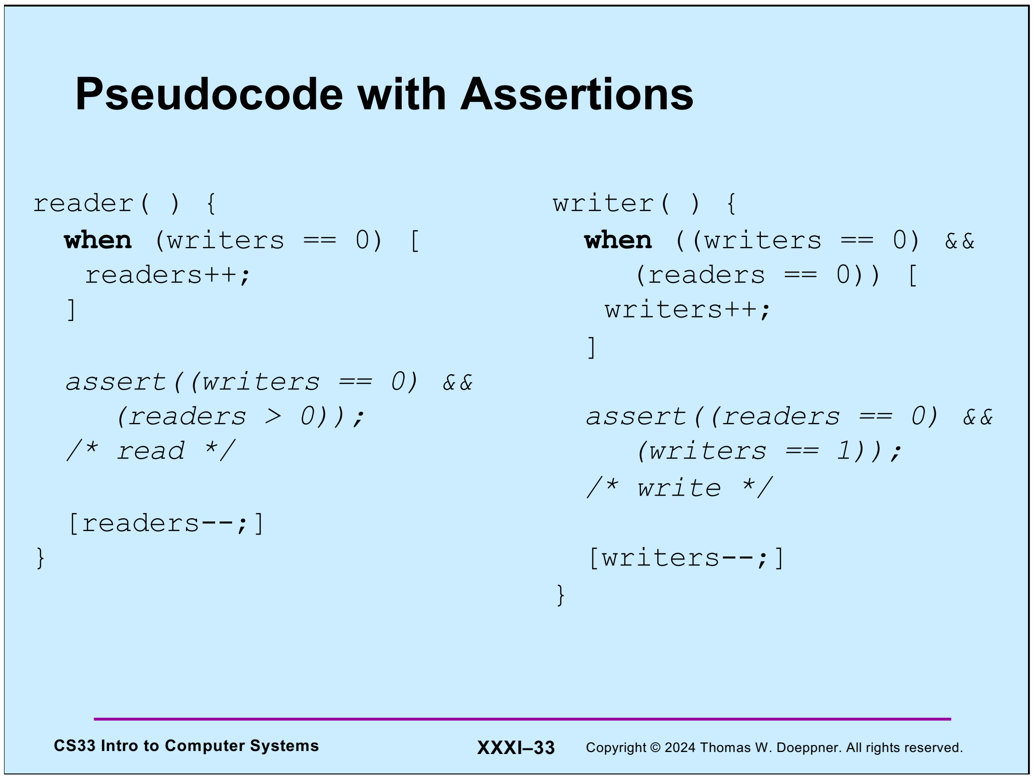
Let's look at another classic synchronization problem — the **readers-writers problem**. Here we have some sort of data structure to which any number of threads may have simultaneous access, as long as they are just reading. But if a thread is to write in the data structure, it must have exclusive access.



Pseudocode

```
reader( ) {                        writer( ) {
  when (writers == 0) [              when ((writers == 0) &&
   readers++;                           (readers == 0)) [
  ]                                    writers++;
                                     ]
  /* read */
                                     /* write */
  [readers--;]
}                                    [writers--;]
                                   }
```

Here we again use guarded commands to describe our solution.

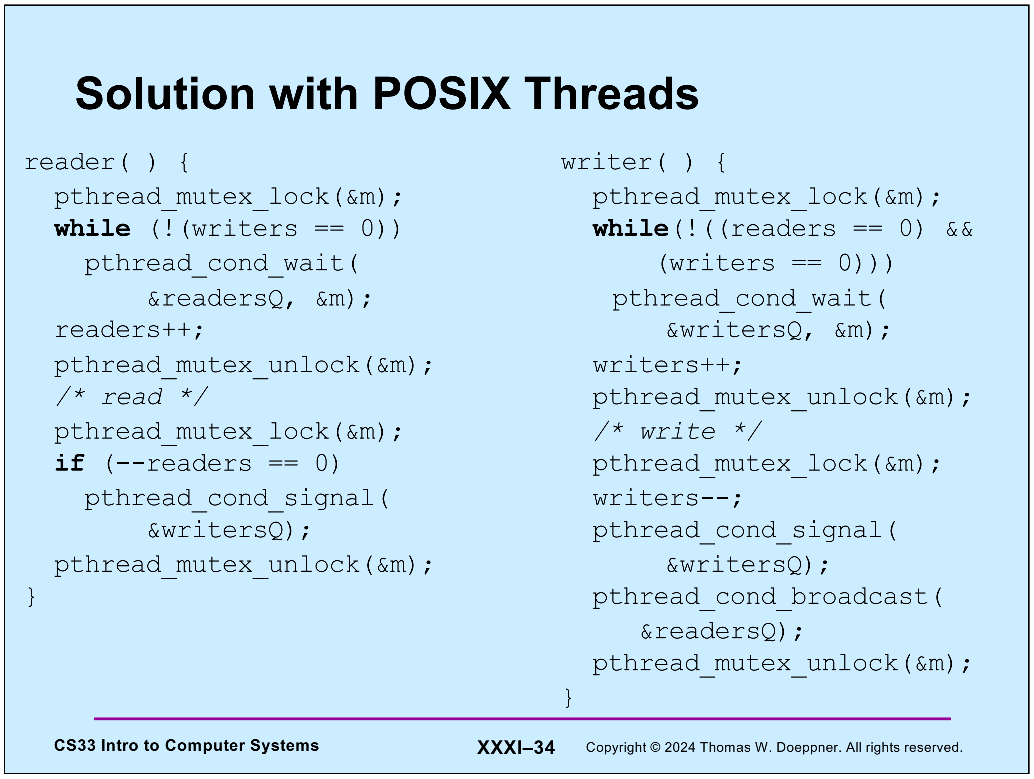


Pseudocode with Assertions

```
reader( ) {
  when (writers == 0) [
   readers++;
  ]

  assert((writers == 0) &&
     (readers > 0));
  /* read */

  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
     (readers == 0)) [
   writers++;
  ]

  assert((readers == 0) &&
     (writers == 1));
  /* write */

  [writers--;]
}
```

We’ve attached assertions to our pseudocode to help make it clearer that our code is correct. The use of assertions is a valuable technique (even in real code), particularly for multithreaded programs.



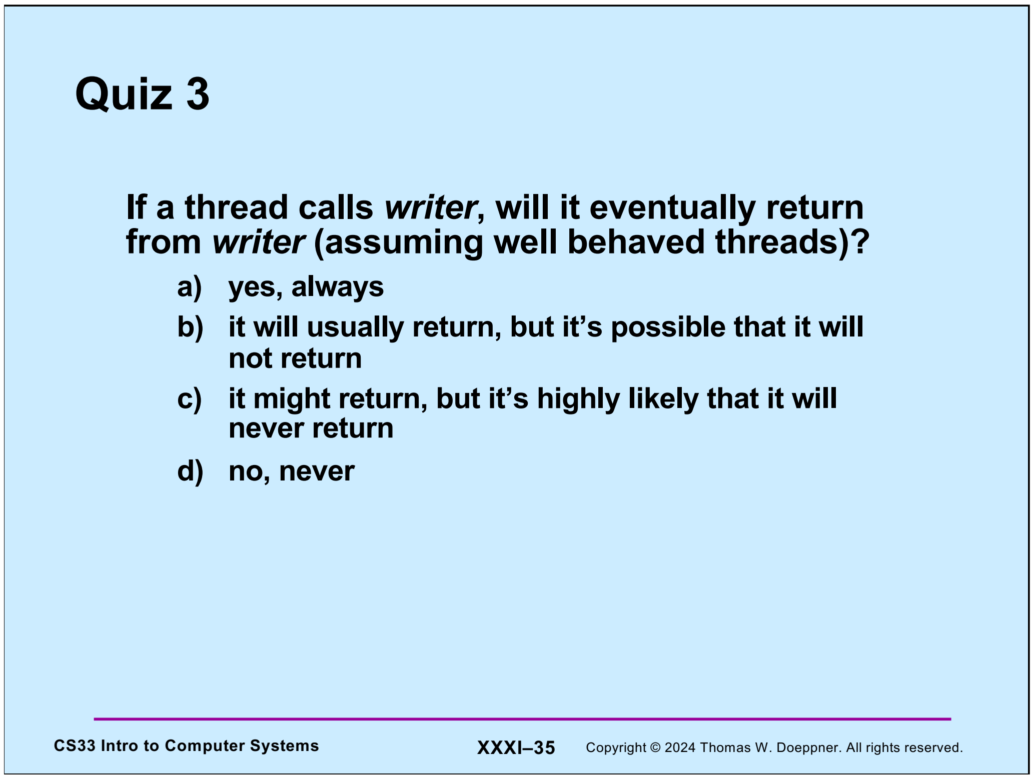
Solution with POSIX Threads

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

```
writer( ) {
  pthread_mutex_lock(&m);
  while(!((readers == 0) &&
      (writers == 0)))
   pthread_cond_wait(
       &writersQ, &m);
  writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  pthread_cond_signal(
      &writersQ);
  pthread_cond_broadcast(
     &readersQ);
  pthread_mutex_unlock(&m);
}
```

Now we convert the pseudocode to real code. We use two condition variables, **readersQ** and **writersQ**, to represent queues of readers and writers waiting for notification that their respective guards are true.

The writer calls **pthread_cond_signal** on **writersQ** so that it wakes up at most one writer, but calls **pthread_cond_broadcast** on **readersQ** to wake up all the readers.



Quiz 3

If a thread calls *writer*, will it eventually return from *writer* (assuming well behaved threads)?

a) yes, always

b) it will usually return, but it's possible that it will not return

c) it might return, but it's highly likely that it will never return

d) no, never

Well behaved threads always unlock the locks they lock.