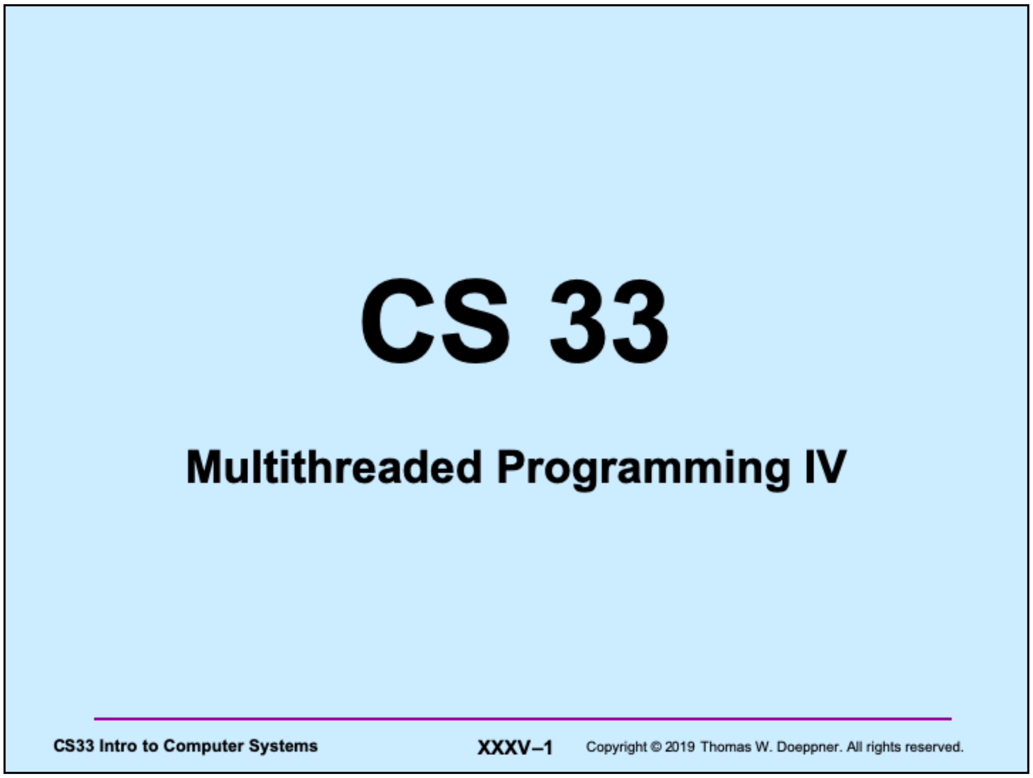
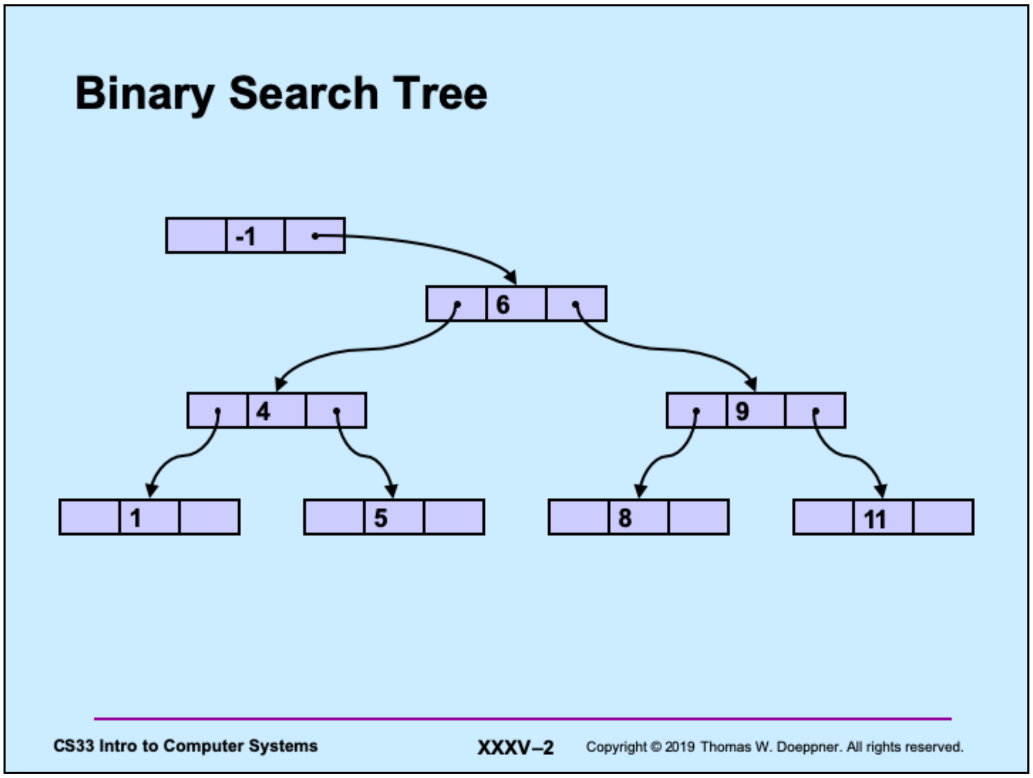


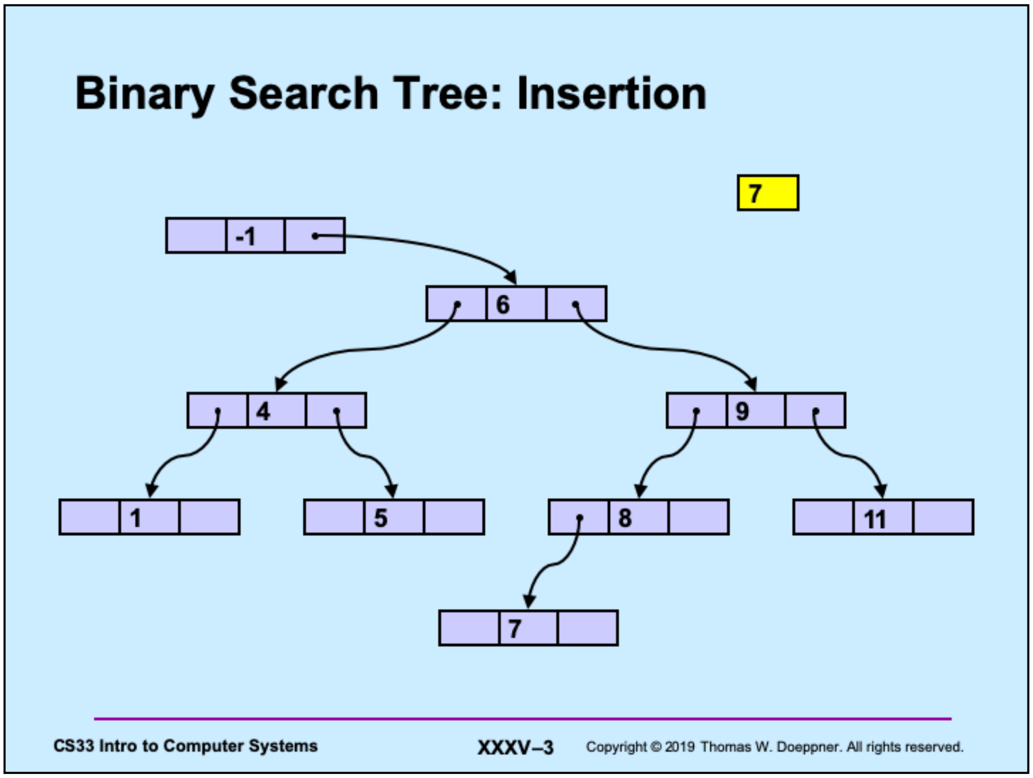
CS 33

Multithreaded Programming IV



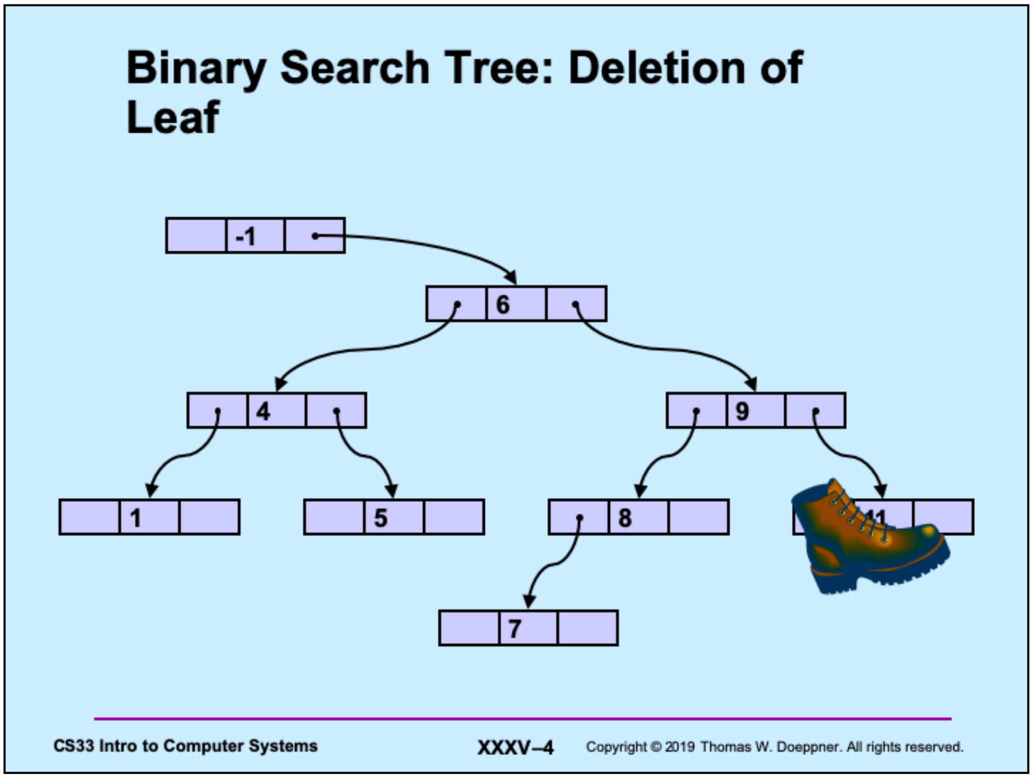
Binary Search Tree

In this sequence of slides we look at how we might take a simple (unbalanced) binary search tree and add readers-writers locks to it so that multiple threads can manipulate it concurrently. Each node of the tree consists of a pointer to a left child, a pointer to a right child, and a key (an integer value). For each node, all nodes in its left subtree have keys that are less than that of the node; all nodes in its right subtree have keys that are greater than that of the node. There are no duplicate keys. All keys are non-negative except for the special head node, which is present even for an empty tree, whose key has a value of -1.



Binary Search Tree: Insertion

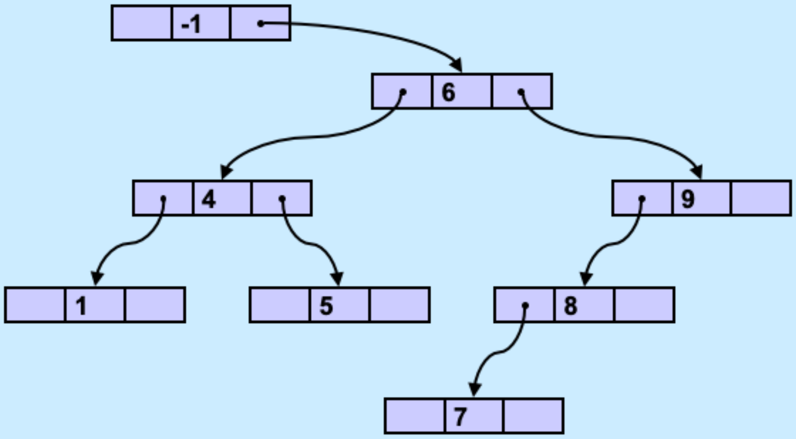
To add a new node to the tree, say one whose key will be 7, we start at the head and trace our way down the tree, comparing the new key with the keys of tree nodes, following left or right child pointers as appropriate. A new node is always inserted as a leaf.

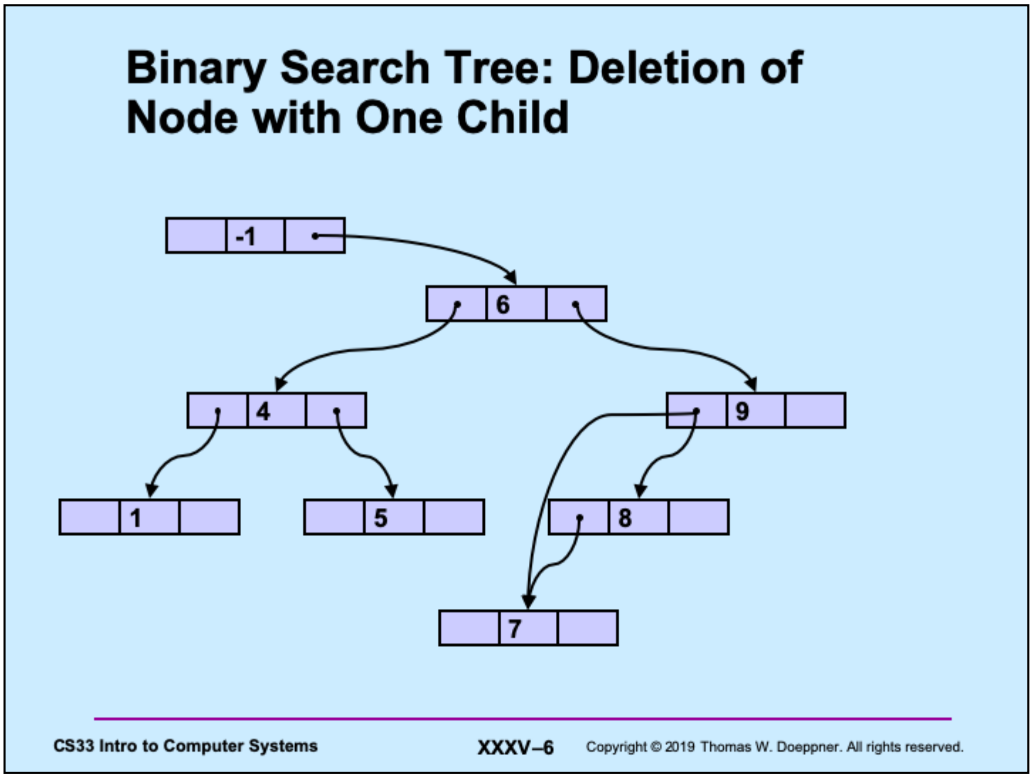


Binary Search Tree: Deletion of Leaf

Deleting a leaf node is easy — it's simply removed and the child pointer from its parent is set to null.

Binary Search Tree: Deletion of Leaf

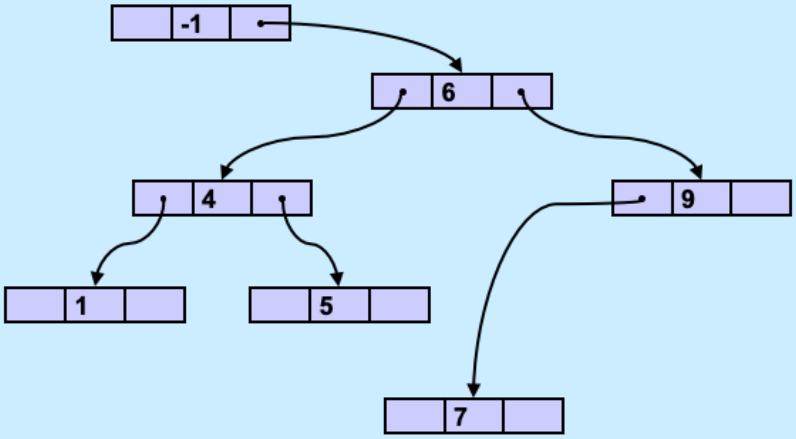


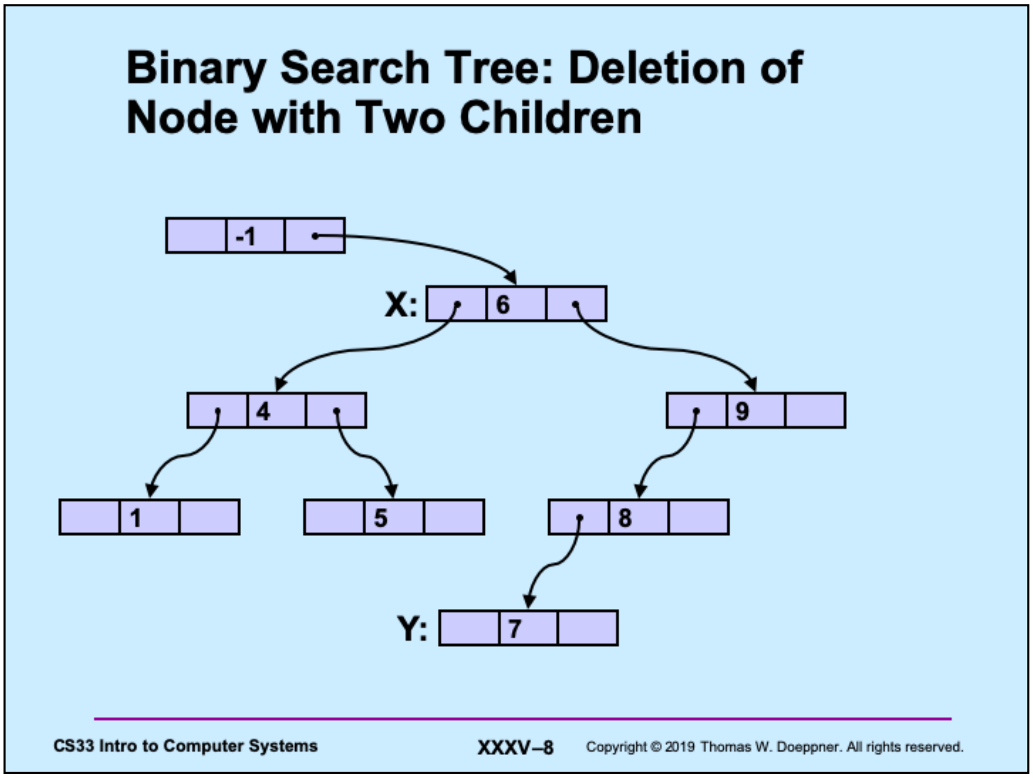


Binary Search Tree: Deletion of Node with One Child

Deleting an interior node that has just one child is almost as easy. The child pointer from its parent is changed to point to the node's child, and then the node is deleted.

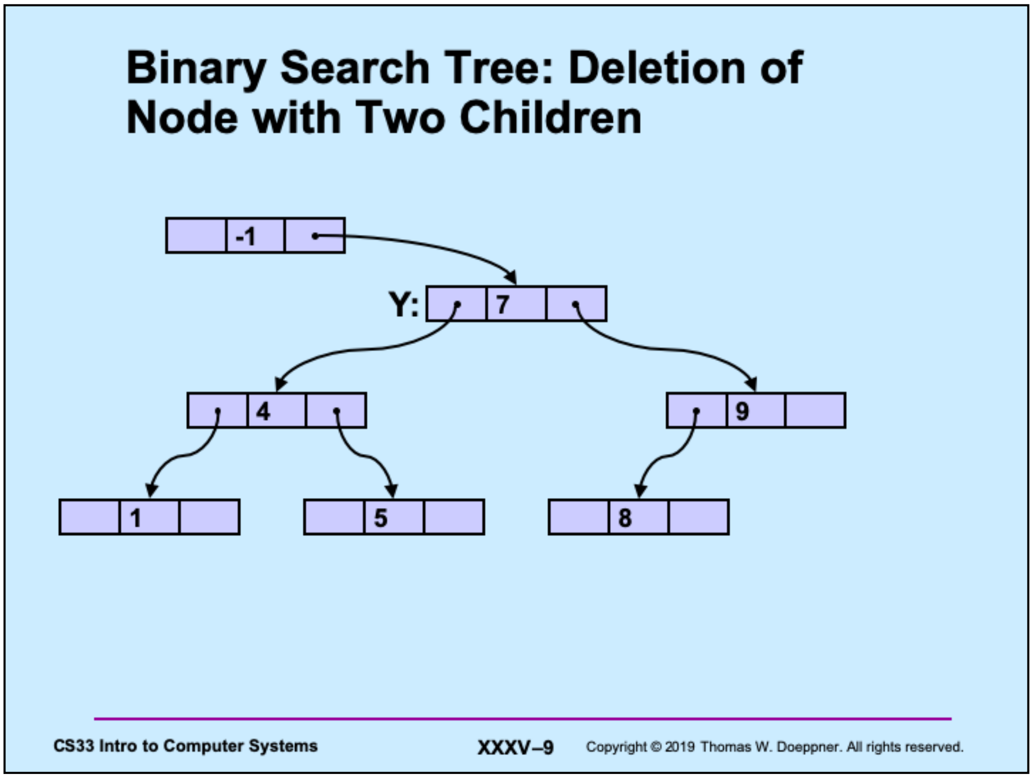
Binary Search Tree: Deletion of Node with One Child





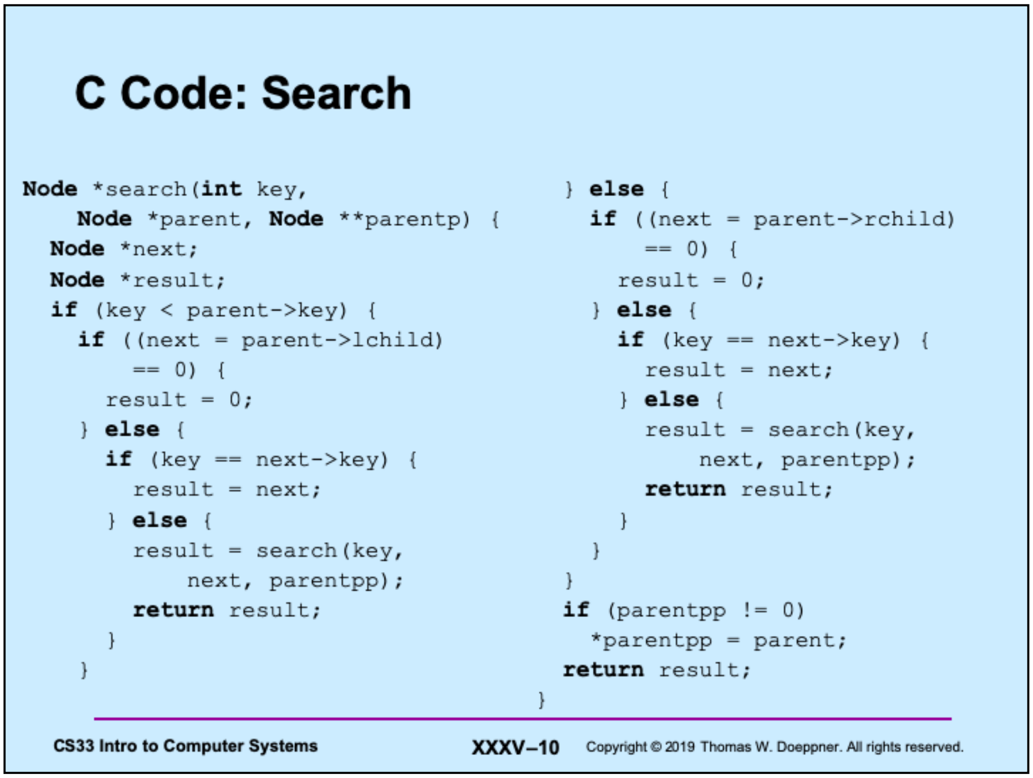
Binary Search Tree: Deletion of Node with Two Children

Deleting a node that has two children might seem tough, but it’s actually relatively easy. Consider deleting the node, X, whose value is 6. All nodes in its right subtree have values greater than its value; all nodes in its left subtree have values less than its value. Suppose we remove the node from the right subtree that has the smallest value (in this case, node Y, whose value is 7). This node thus also has a greater value than all nodes in X’s left subtree. Thus if we replace the value of node X with Y’s value, we end up with a valid binary search tree.



Binary Search Tree: Deletion of Node with Two Children

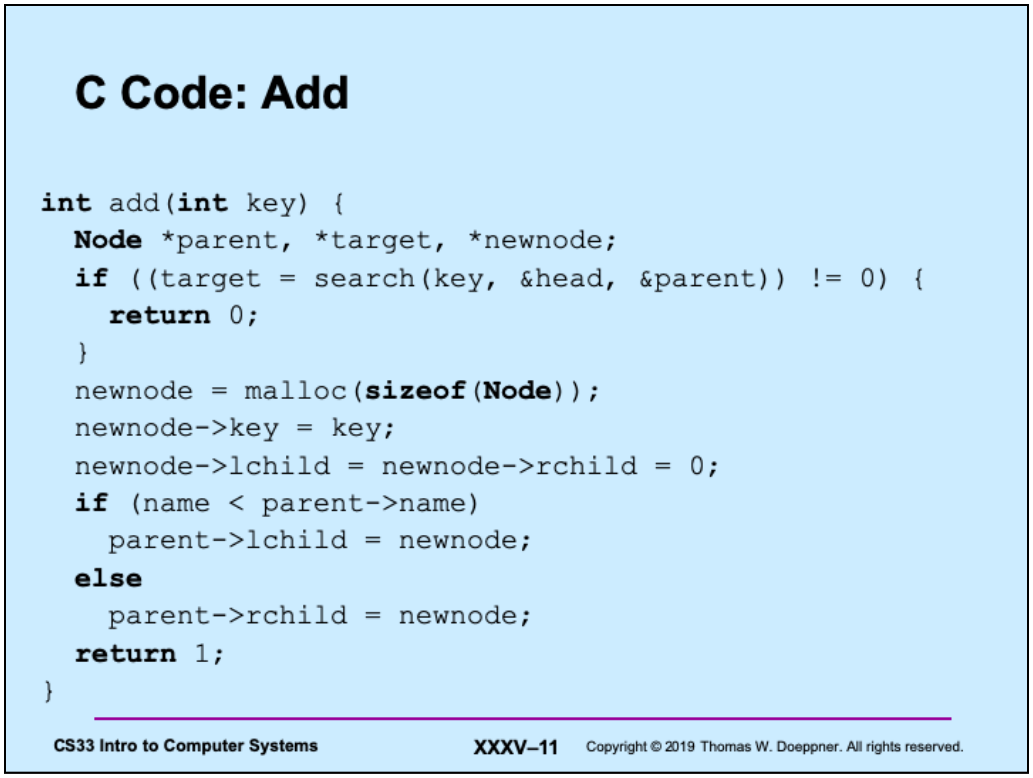
Thus effectively we've reduced the problem of deleting a node with two children to deleting a node with at most one child.



C Code: Search

```
Node *search(int key,
    Node *parent, Node **parentp) {
  Node *next;
  Node *result;
  if (key < parent->key) {
    if ((next = parent->lchild)
        == 0) {
      result = 0;
    } else {
      if (key == next->key) {
        result = next;
      } else {
        result = search(key,
            next, parentpp);
        return result;
      }
    }
  } else {
    if ((next = parent->rchild)
        == 0) {
      result = 0;
    } else {
      if (key == next->key) {
        result = next;
      } else {
        result = search(key,
            next, parentpp);
        return result;
      }
    }
  }
  if (parentpp != 0)
    *parentpp = parent;
  return result;
}
```

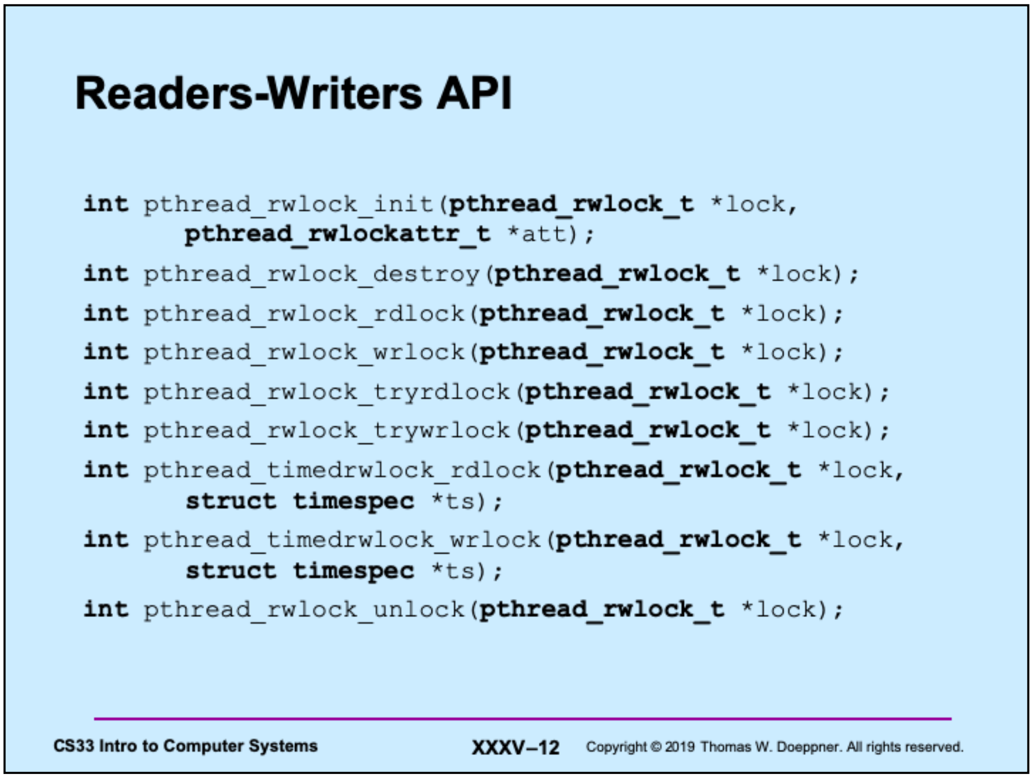
Here is the C code for searching our binary search tree, which returns either a pointer to the node containing the key or null if no such node exists. Note that search assumes that the key being searched for is not in the parent node. If the *parentp* argument is not null, then it points to a location into which the address of the returned node's parent is stored if the key is found, otherwise it returns a pointer to what would be the parent of the node containing the key if the key were in the tree.



C Code: Add

```
int add(int key) {
  Node *parent, *target, *newnode;
  if ((target = search(key, &head, &parent)) != 0) {
    return 0;
  }
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;
  if (name < parent->name)
    parent->lchild = newnode;
  else
    parent->rchild = newnode;
  return 1;
}
```

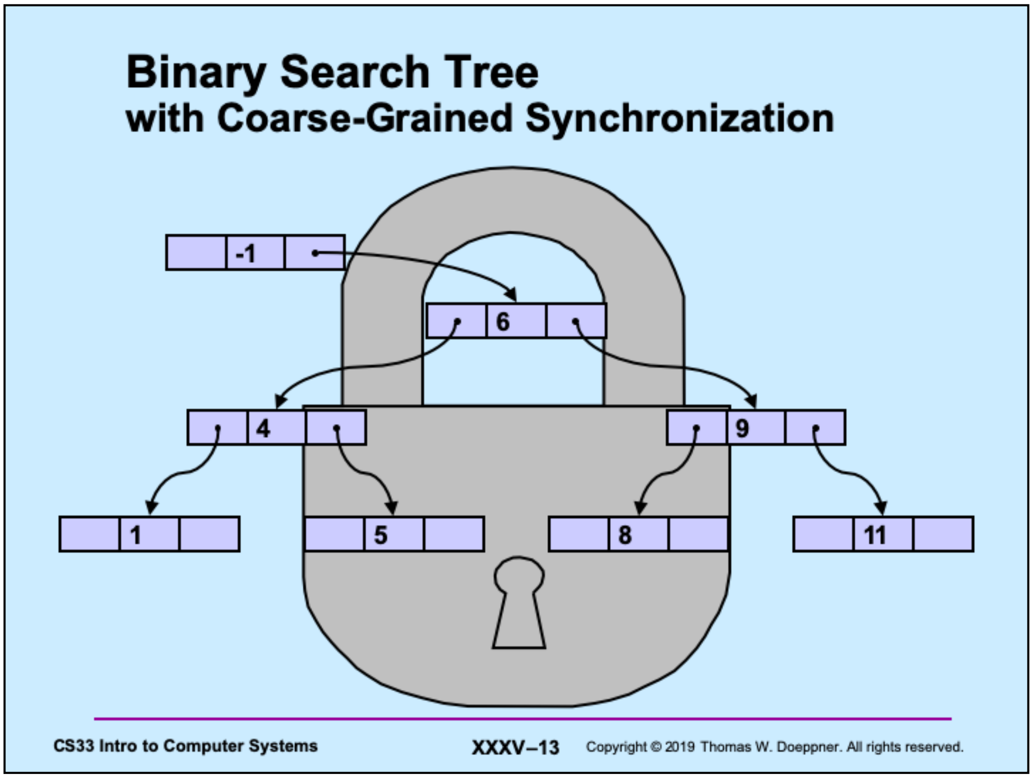
Here’s the C code for adding a node to the binary search tree.



Readers-Writers API

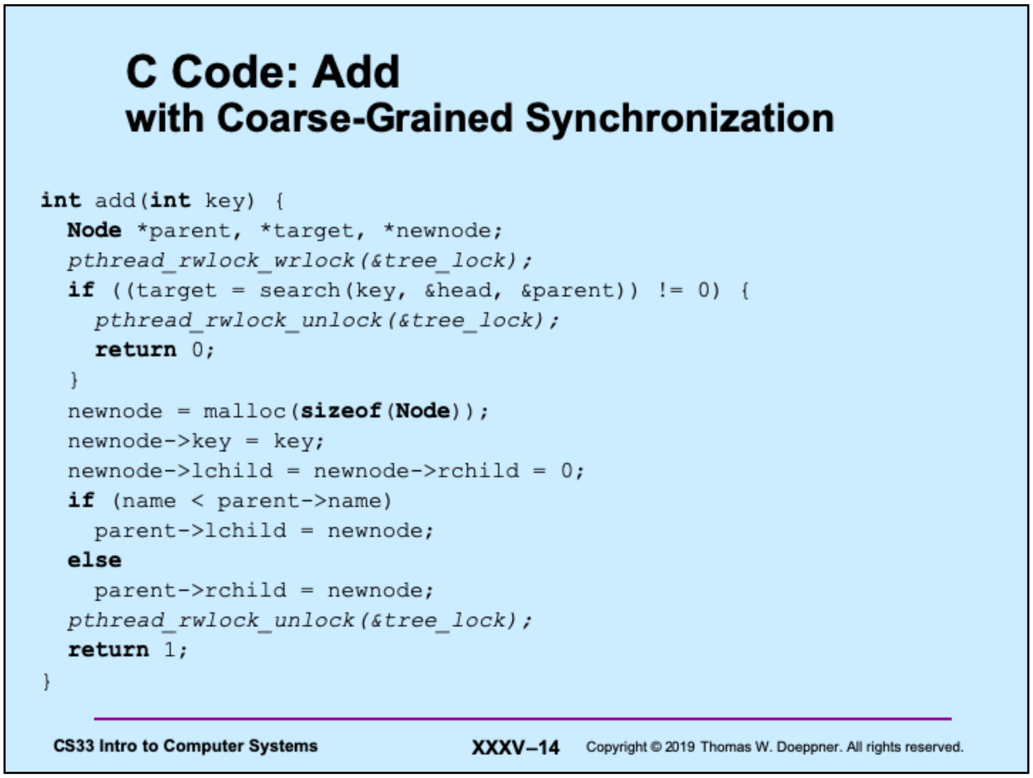
```
int pthread_rwlock_init(pthread_rwlock_t *lock,
        pthread_rwlockattr_t *att);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
int pthread_timedrwlock_rdlock(pthread_rwlock_t *lock,
        struct timespec *ts);
int pthread_timedrwlock_wrlock(pthread_rwlock_t *lock,
        struct timespec *ts);
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

With POSIX 1003.1j support for readers-writers locks was finally introduced. The almost complete API is shown in the slide (what's missing are the operations on attributes). As might be expected, readers-writers locks can be statically initialized with the constant PTHREAD_RWLOCK_INITIALIZER. The "timedrwlock" routines allow one to wait until the lock is available or a time-limit is exceeded, whichever comes first.



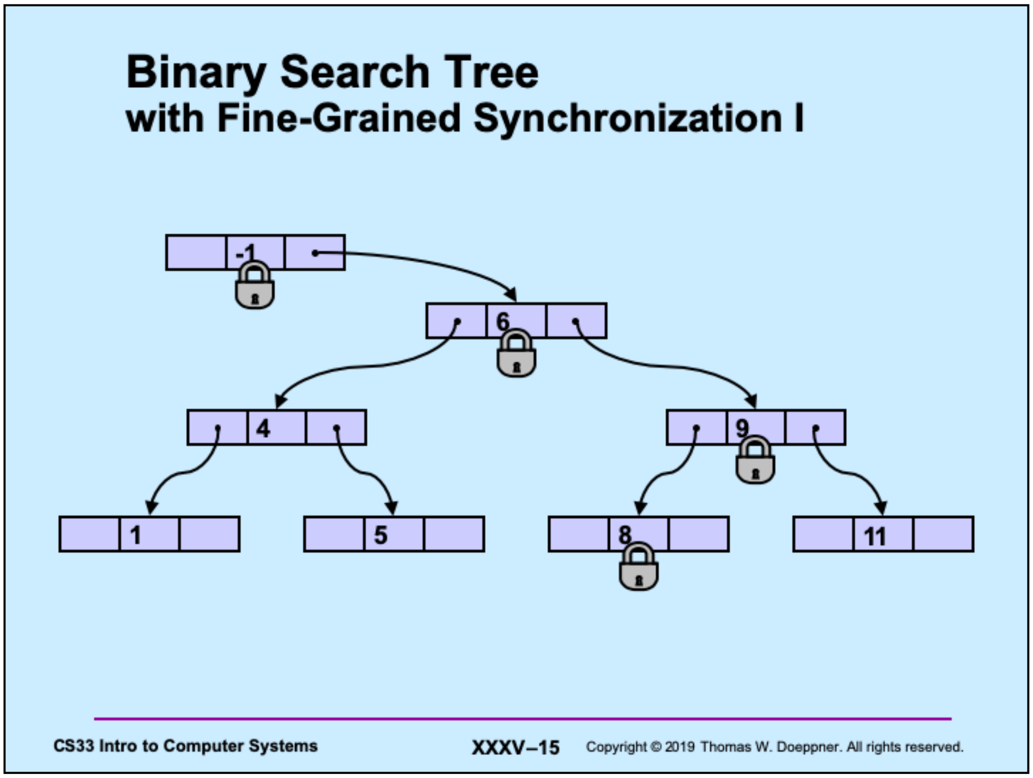
Binary Search Tree
with Coarse-Grained Synchronization

An easy way to allow multiple threads to manipulate the search tree concurrently is to employ what's known as *coarse-grained synchronization*: we associate a readers-writers lock with the entire tree. A thread that is just searching the tree for a value should take a read lock. A thread attempting to modify the tree, either adding or deleting a node, should take a write lock.



C Code: Add
with Coarse-Grained Synchronization

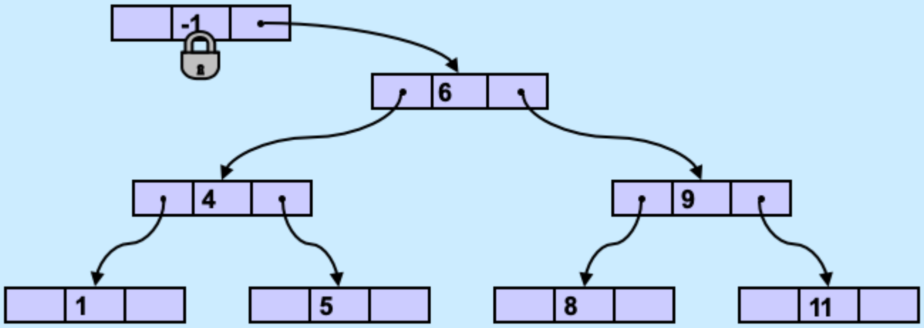
```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&tree_lock);
  if ((target = search(key, &head, &parent)) != 0) {
    pthread_rwlock_unlock(&tree_lock);
    return 0;
  }
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;
  if (name < parent->name)
    parent->lchild = newnode;
  else
    parent->rchild = newnode;
  pthread_rwlock_unlock(&tree_lock);
  return 1;
}
```

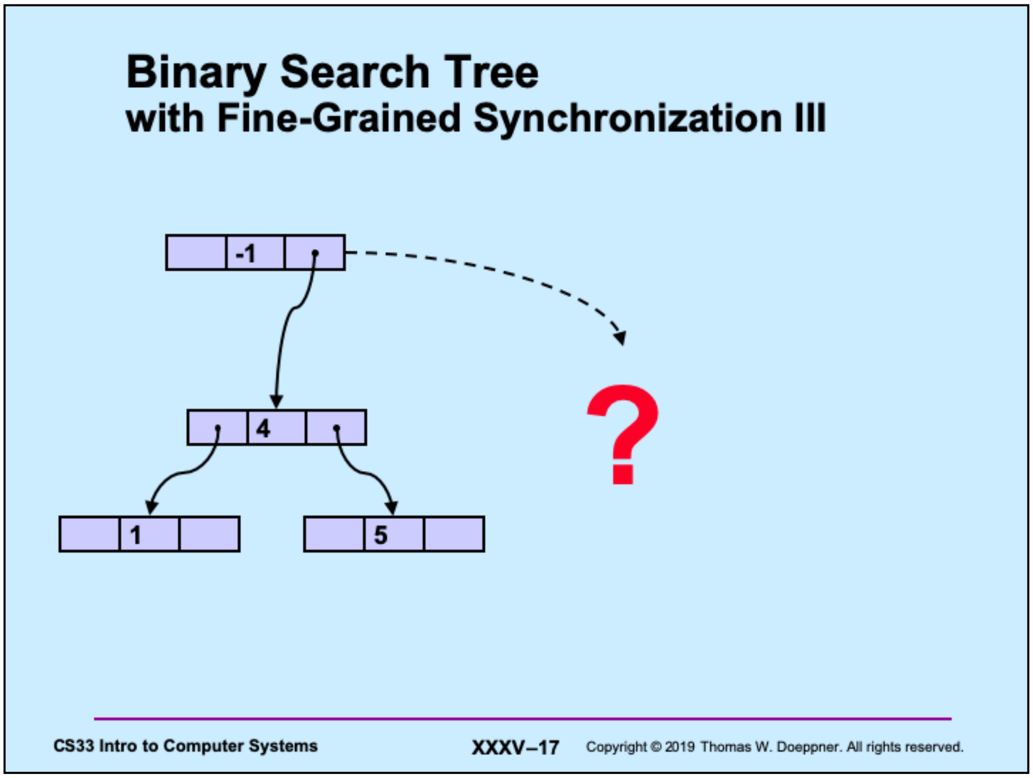


Binary Search Tree
with Fine-Grained Synchronization I

Let’s now look at what’s known as *fine-grained synchronization*, where we associate a readers-writers lock with each node of the tree. The idea is that, unlike the case for coarse-grained synchronization, we can have multiple threads working on different parts of the tree at once. The first step in making this work is to modify the search algorithm so as to lock and unlock the nodes’ rw locks appropriately. As a first attempt, we use the simple algorithm of first locking a node, then determining, based on its key’s value, which child we go to next, then unlocking the node and repeating with the child.

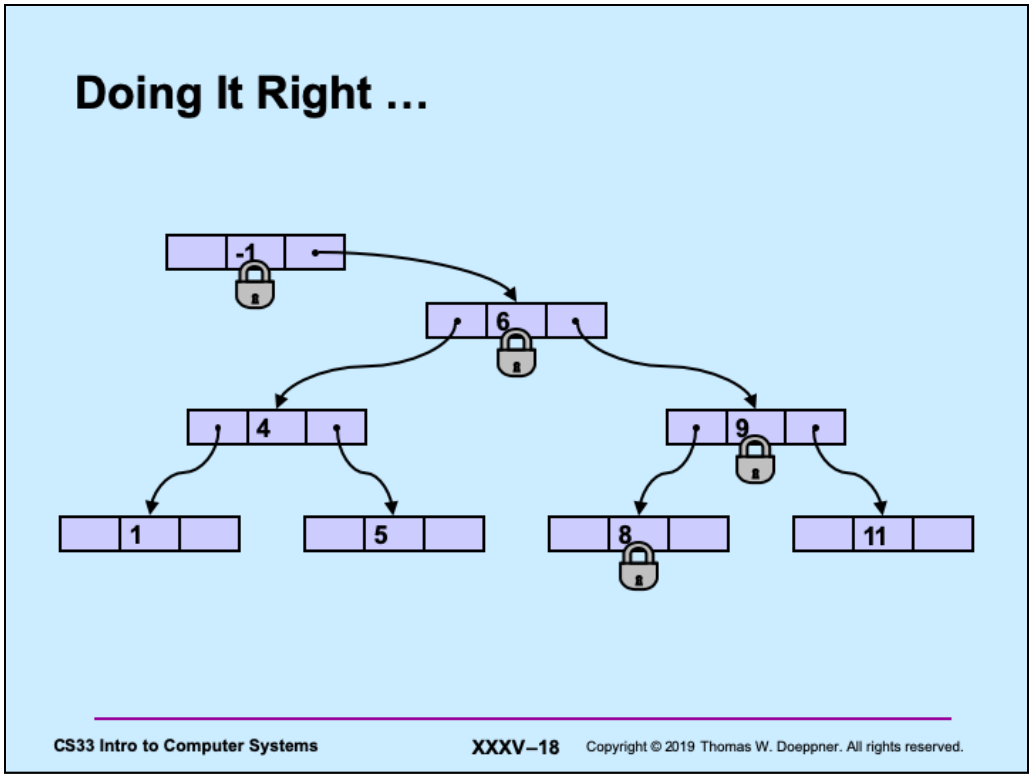
Binary Search Tree
with Fine-Grained Synchronization II





Binary Search Tree
with Fine-Grained Synchronization III

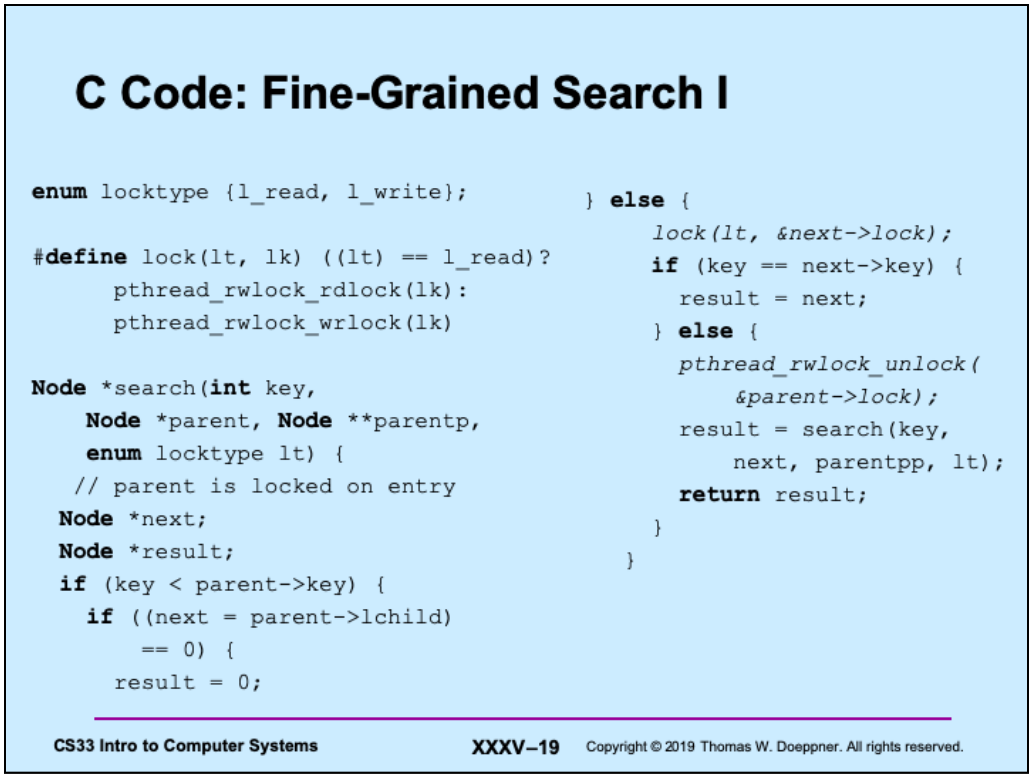
This approach could lead to trouble if after we obtain a pointer to a child and unlock a node, some other thread deletes the child (and other nodes).



Doing It Right …

CS33 Intro to Computer Systems XXXV–18

To avoid such problems, once we get a pointer to a child, we should lock the child’s rw lock, and then unlock the parent’s rw lock. This prevents other threads from deleting the child while we are using it.



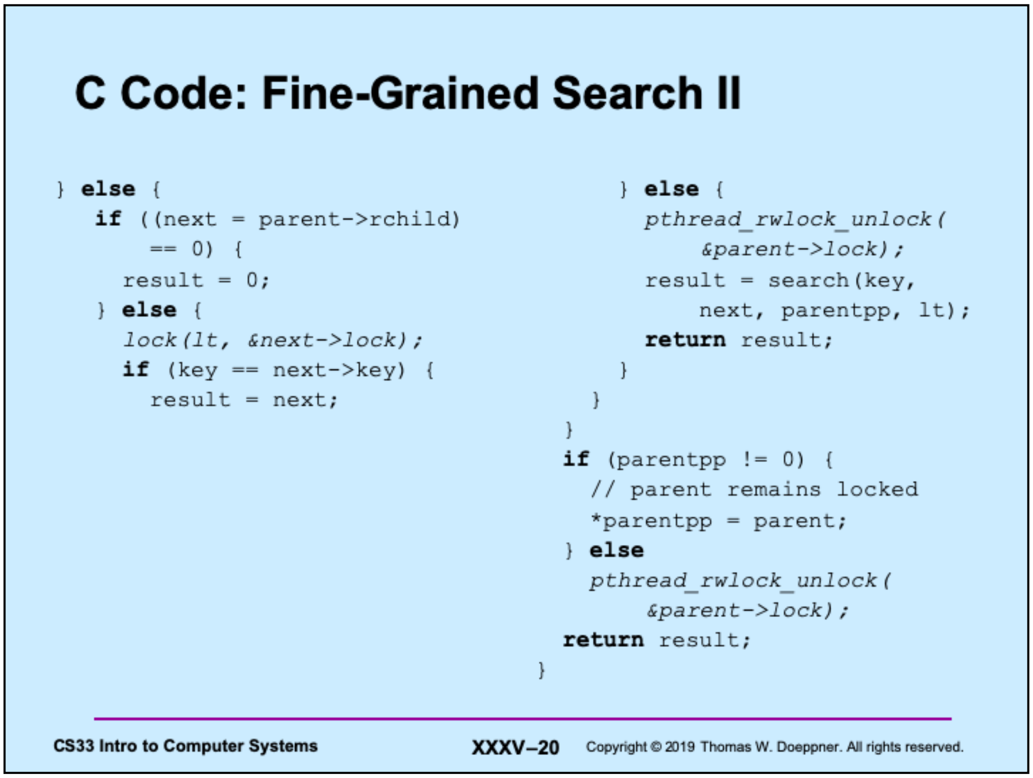
C Code: Fine-Grained Search I

```
enum locktype {l_read, l_write};

#define lock(lt, lk) ((lt) == l_read)?
      pthread_rwlock_rdlock(lk):
      pthread_rwlock_wrlock(lk)

Node *search(int key,
    Node *parent, Node **parentp,
    enum locktype lt) {
   // parent is locked on entry
  Node *next;
  Node *result;
  if (key < parent->key) {
    if ((next = parent->lchild)
        == 0) {
      result = 0;
```

```
} else {
     lock(lt, &next->lock);
     if (key == next->key) {
       result = next;
     } else {
       pthread_rwlock_unlock(
           &parent->lock);
       result = search(key,
           next, parentpp, lt);
       return result;
     }
   }
```

And here is the fine-grained search routine. Note that its last argument indicates whether it’s called by a thread that’s only searching the tree, or by a thread that intends to modify the tree. Note also that the routine assumes that the parent node is locked by the caller (and that the key being searched for is not in the parent node).

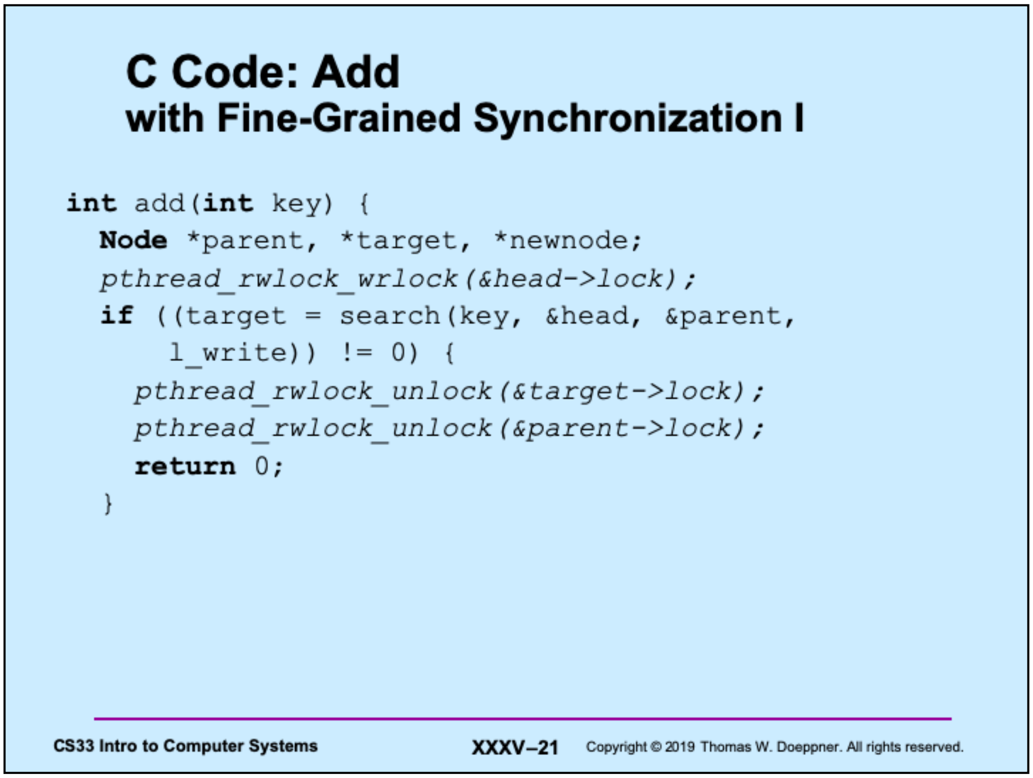
If a node containing the key is found, the found node is locked and a pointer to it is returned. If *parentp* is non-null, then the final parent node is locked and a pointer to it is stored in the location pointed to by *parentp* (the code for this is on the next slide).



C Code: Fine-Grained Search II

```
} else {
   if ((next = parent->rchild)
       == 0) {
     result = 0;
   } else {
     lock(lt, &next->lock);
     if (key == next->key) {
       result = next;
```

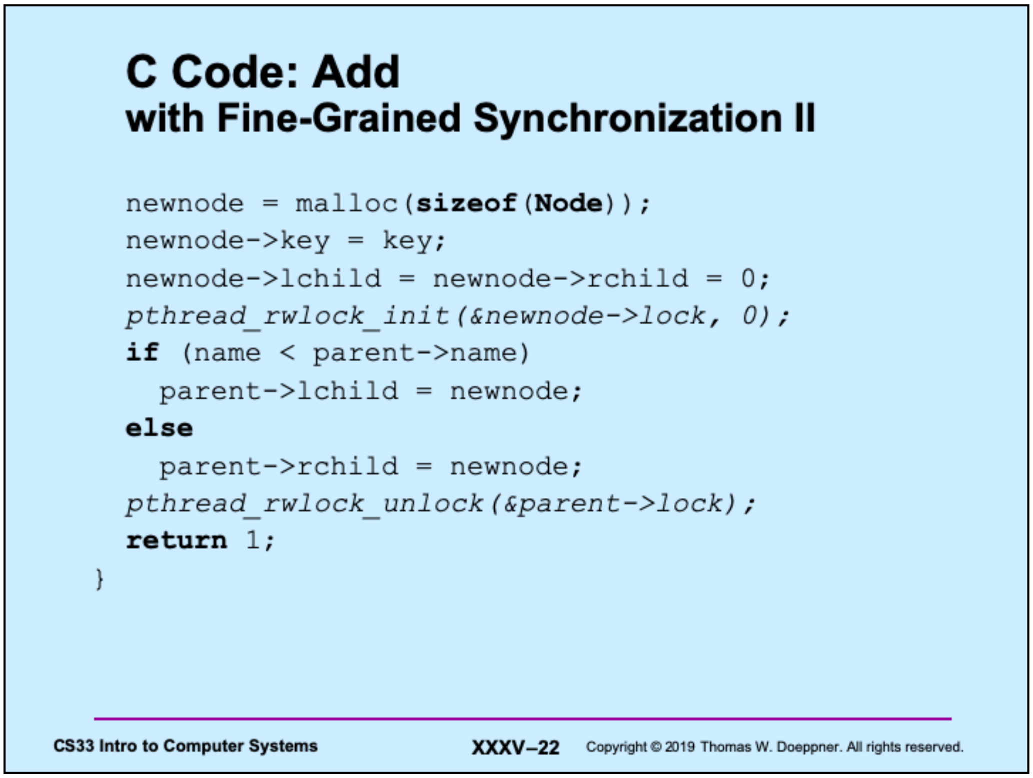
```
      } else {
        pthread_rwlock_unlock(
            &parent->lock);
        result = search(key,
            next, parentpp, lt);
        return result;
      }
    }
  }
  if (parentpp != 0) {
    // parent remains locked
    *parentpp = parent;
  } else
    pthread_rwlock_unlock(
        &parent->lock);
  return result;
}
```



C Code: Add
with Fine-Grained Synchronization I

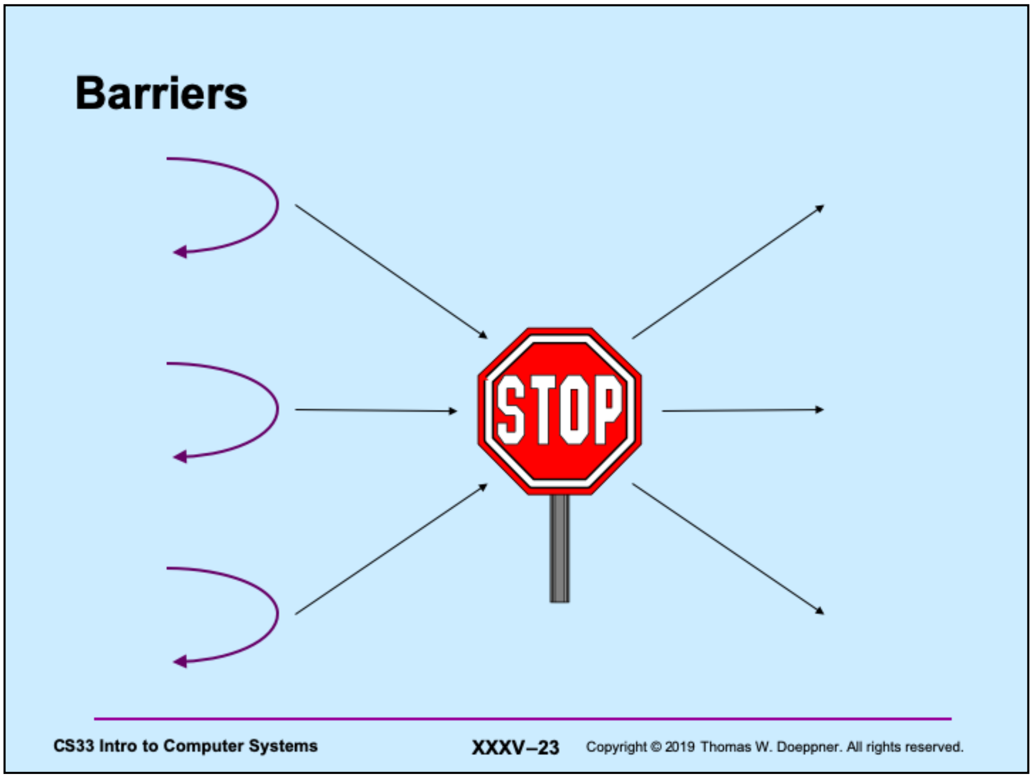
```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&head->lock);
  if ((target = search(key, &head, &parent,
      l_write)) != 0) {
    pthread_rwlock_unlock(&target->lock);
    pthread_rwlock_unlock(&parent->lock);
    return 0;
  }
```

Here is the add routine modified for fine-grained synchronization.



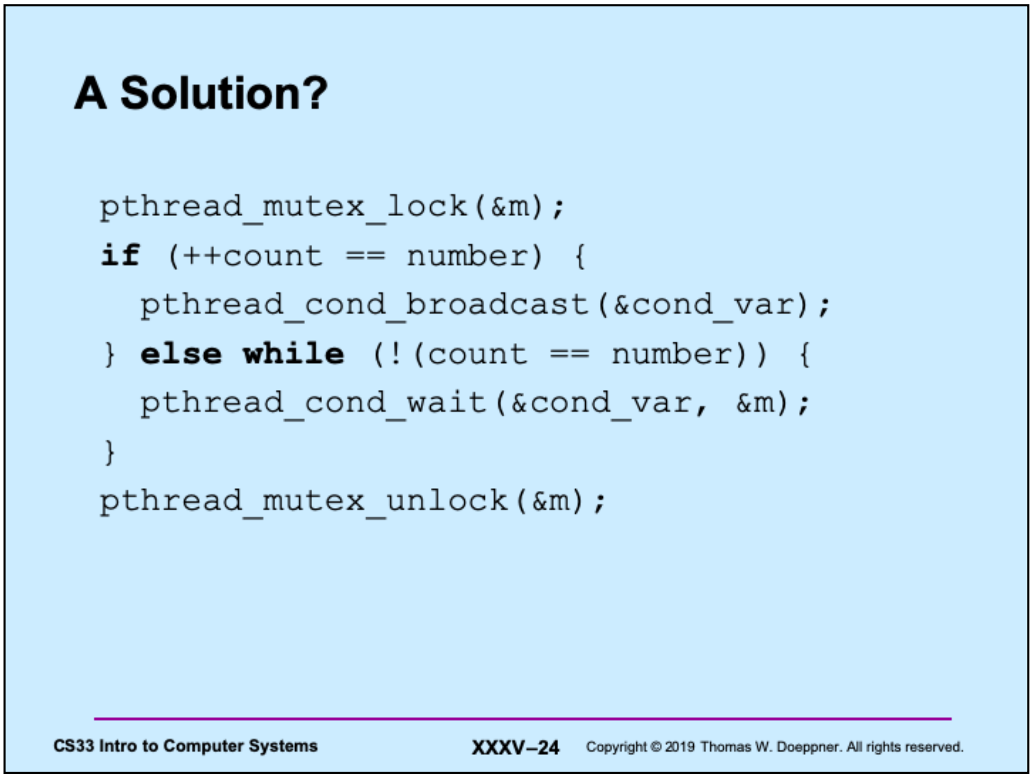
C Code: Add
with Fine-Grained Synchronization II

```
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;
  pthread_rwlock_init(&newnode->lock, 0);
  if (name < parent->name)
    parent->lchild = newnode;
  else
    parent->rchild = newnode;
  pthread_rwlock_unlock(&parent->lock);
  return 1;
}
```



Barriers

A *barrier* is a conceptually simple and very useful synchronization construct. A barrier is established for some predetermined number of threads; threads call the barrier's *wait* routine to enter it; no thread may exit the barrier until all threads have entered it.

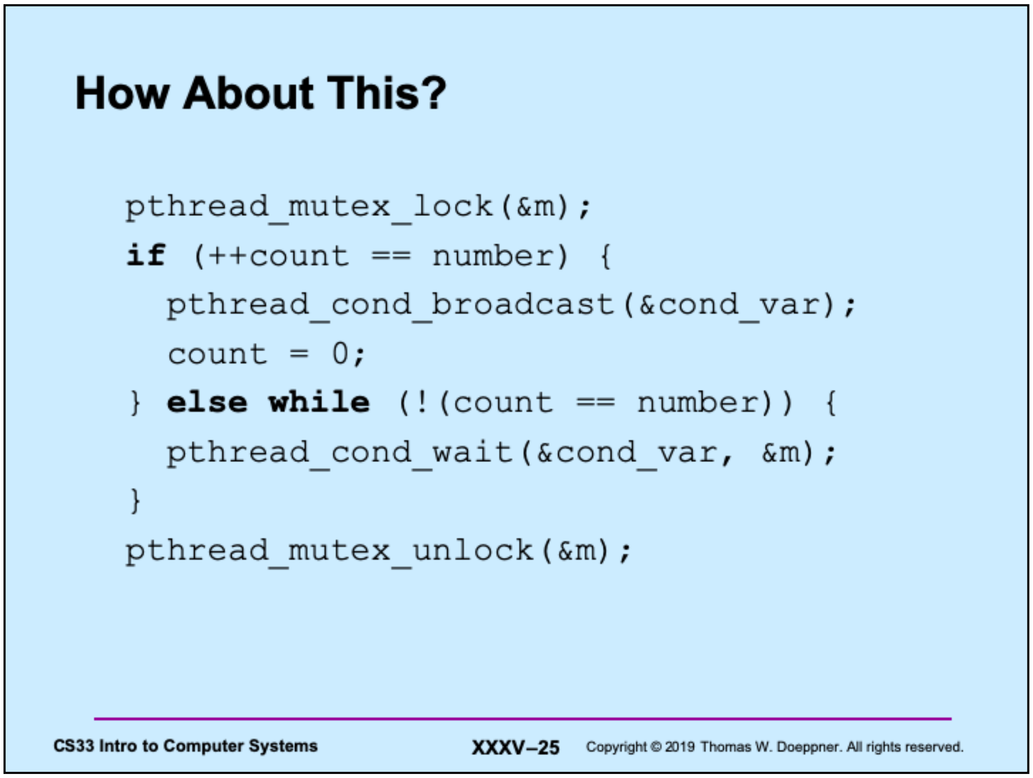


A Solution?

```
pthread_mutex_lock(&m);
if (++count == number) {
  pthread_cond_broadcast(&cond_var);
} else while (!(count == number)) {
  pthread_cond_wait(&cond_var, &m);
}
pthread_mutex_unlock(&m);
```

Is this a correct solution?

It works once, but, since it doesn't reset count to zero, it won't work more than once.

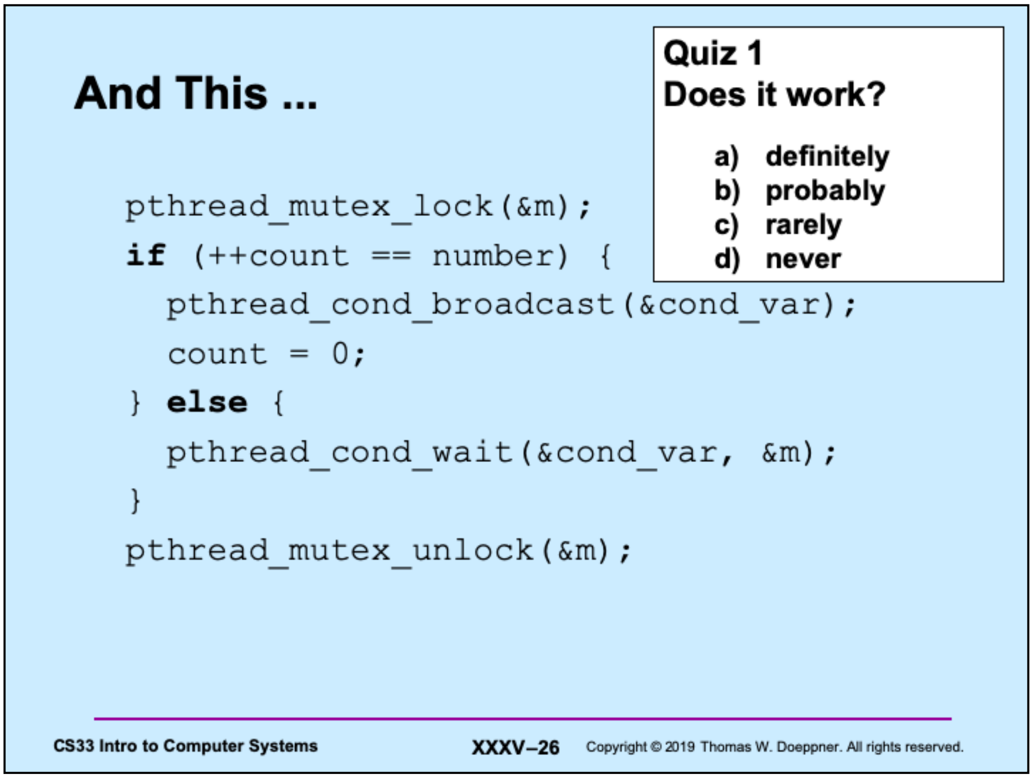


How About This?

```
pthread_mutex_lock(&m);
if (++count == number) {
  pthread_cond_broadcast(&cond_var);
  count = 0;
} else while (!(count == number)) {
  pthread_cond_wait(&cond_var, &m);
}
pthread_mutex_unlock(&m);
```

How about this?

We can try all possible places to reset count to zero – none of them work.

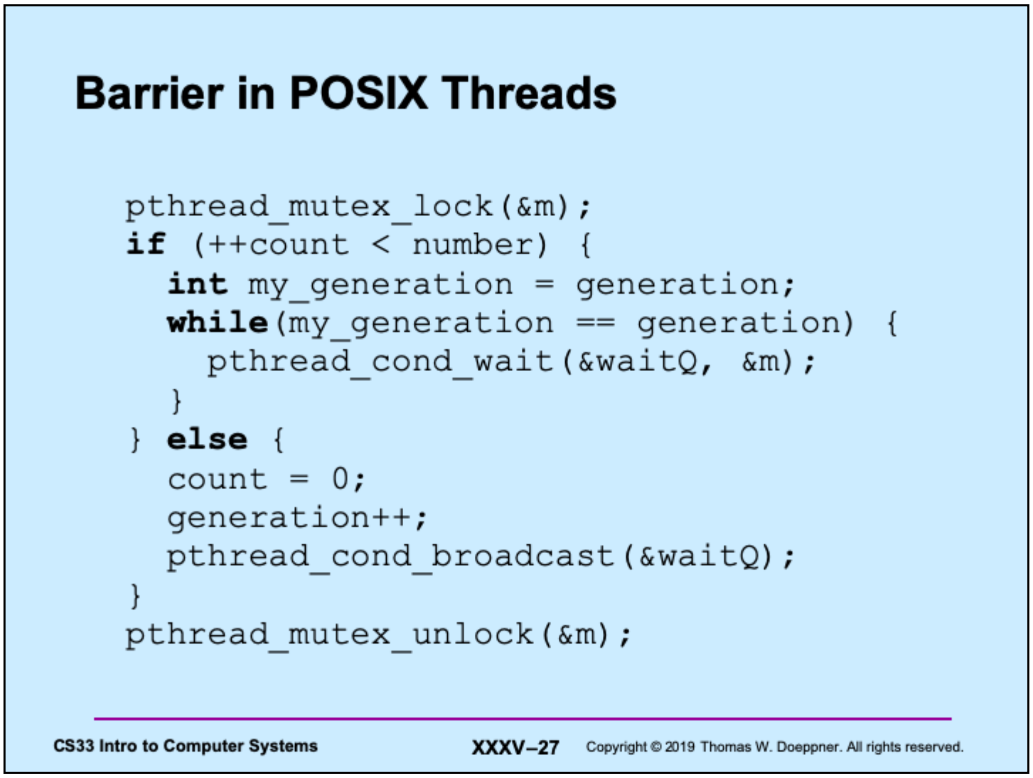


And This ...

Quiz 1
Does it work?

- a) definitely
- b) probably
- c) rarely
- d) never

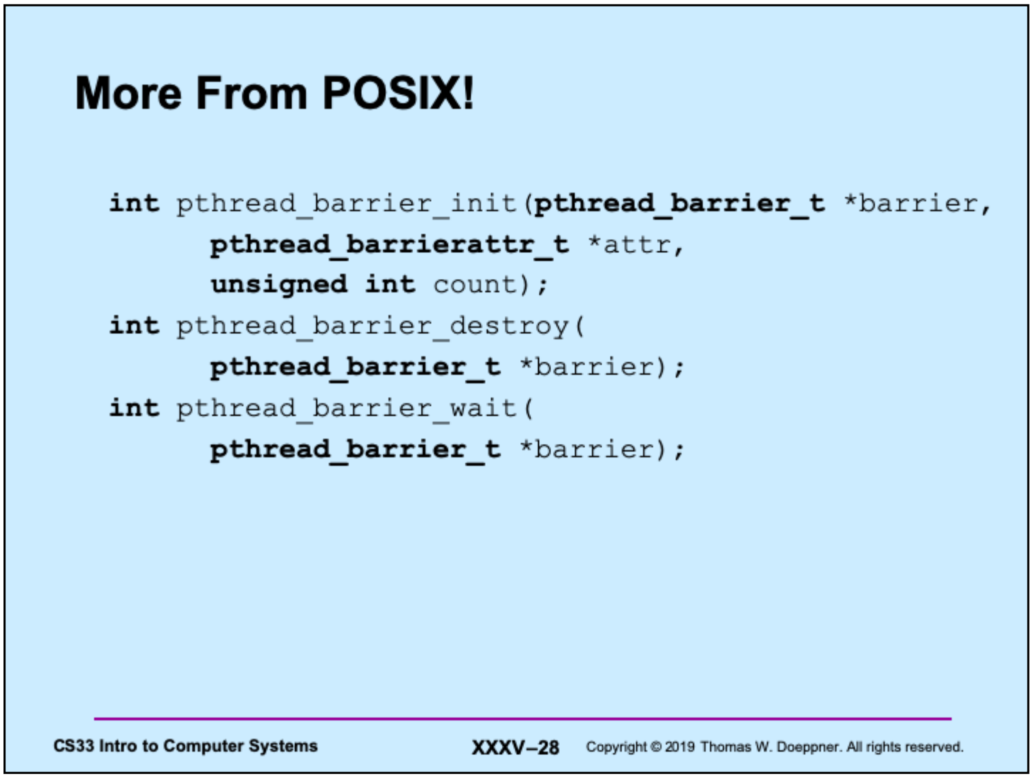
```
pthread_mutex_lock(&m);
if (++count == number) {
  pthread_cond_broadcast(&cond_var);
  count = 0;
} else {
  pthread_cond_wait(&cond_var, &m);
}
pthread_mutex_unlock(&m);
```



Barrier in POSIX Threads

```
pthread_mutex_lock(&m);
if (++count < number) {
  int my_generation = generation;
  while(my_generation == generation) {
    pthread_cond_wait(&waitQ, &m);
  }
} else {
  count = 0;
  generation++;
  pthread_cond_broadcast(&waitQ);
}
pthread_mutex_unlock(&m);
```

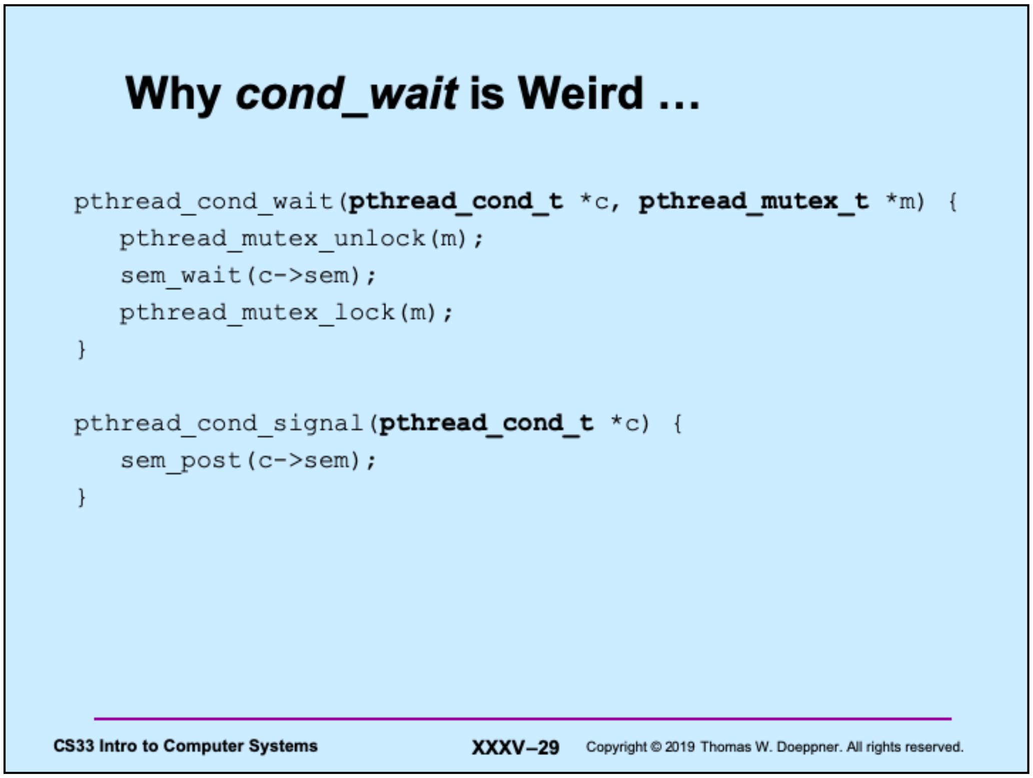
Implementing barriers in POSIX threads is not trivial. Since *count*, the number of threads that have entered the barrier, will be reset to 0 once all threads have entered, we can't use it in the guard. But, nevertheless, we still must wakeup all waiting threads as soon as the last one enters the barrier. We accomplish this with the *generation* global variable and the *my_generation* local variable. An entering thread increments count and joins the condition-variable queue if it's still less than the target number of threads. However, before it joins the queue, it copies the current value of *generation* into its local *my_generation* and then joins the queue of waiting threads, via *pthread_cond_wait*, until *my_generation* is no longer equal to *generation*. When the last thread enters the barrier, it increments generation and wakes up all waiting threads. Each of these sees that its private *my_generation* is no longer equal to *generation*, and thus the last thread must have entered the barrier.



More From POSIX!

```
int pthread_barrier_init(pthread_barrier_t *barrier,
      pthread_barrierattr_t *attr,
      unsigned int count);
int pthread_barrier_destroy(
      pthread_barrier_t *barrier);
int pthread_barrier_wait(
      pthread_barrier_t *barrier);
```

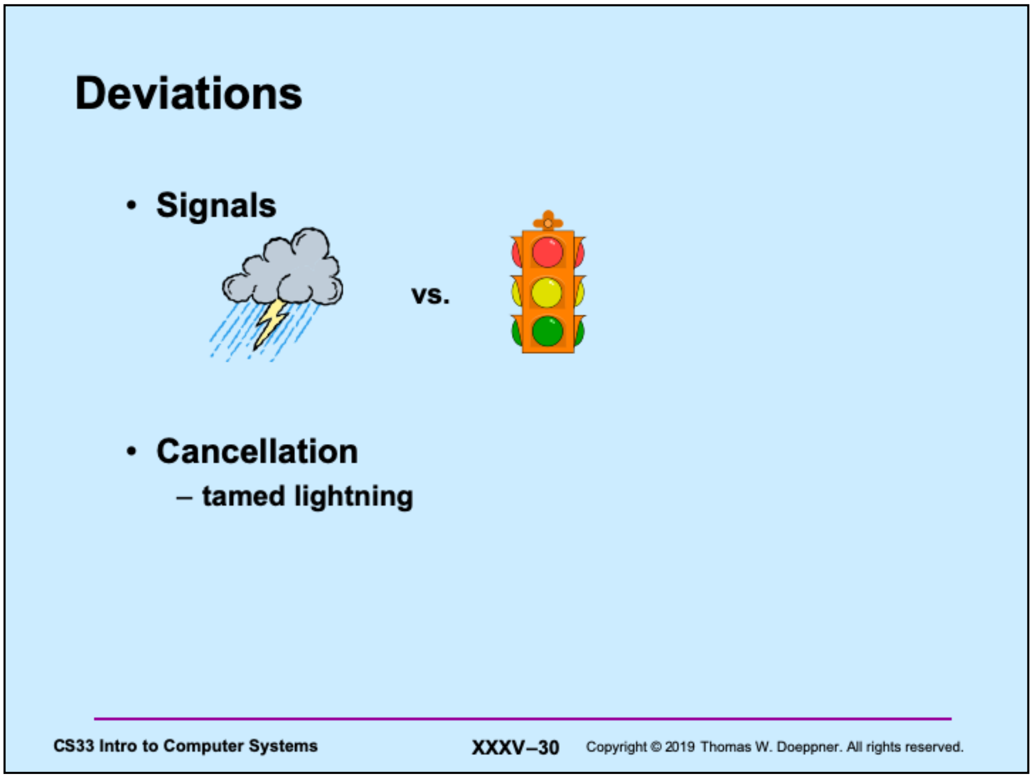
As part of POSIX 1003.1j, barriers were introduced. Unlike other POSIX-threads objects, they cannot be statically initialized; one must call *pthread_barrier_init* and specify the number of threads that must enter the barrier. In some applications it might be necessary for one thread to be designated to perform some sort function on behalf of all of them when all exit the barrier. Thus *pthread_barrier_wait* returns PTHREAD_BARRIER_SERIAL_THREAD in one thread and zero in the others on success.



Why *cond_wait* is Weird …

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m) {
   pthread_mutex_unlock(m);
   sem_wait(c->sem);
   pthread_mutex_lock(m);
}

pthread_cond_signal(pthread_cond_t *c) {
   sem_post(c->sem);
}
```

Consider the implementation of *pthread_cond_wait* and *pthread_cond_signal* shown in the slide. It has the property that calls to *pthread_cond_signal* are “remembered” if done when no threads are waiting on the condition-variable queue. While this is not a desirable property, it simplifies the implementation. To allow such implementations, the semantics of *pthread_cond_wait* are less restrictive than they should be.



Deviations

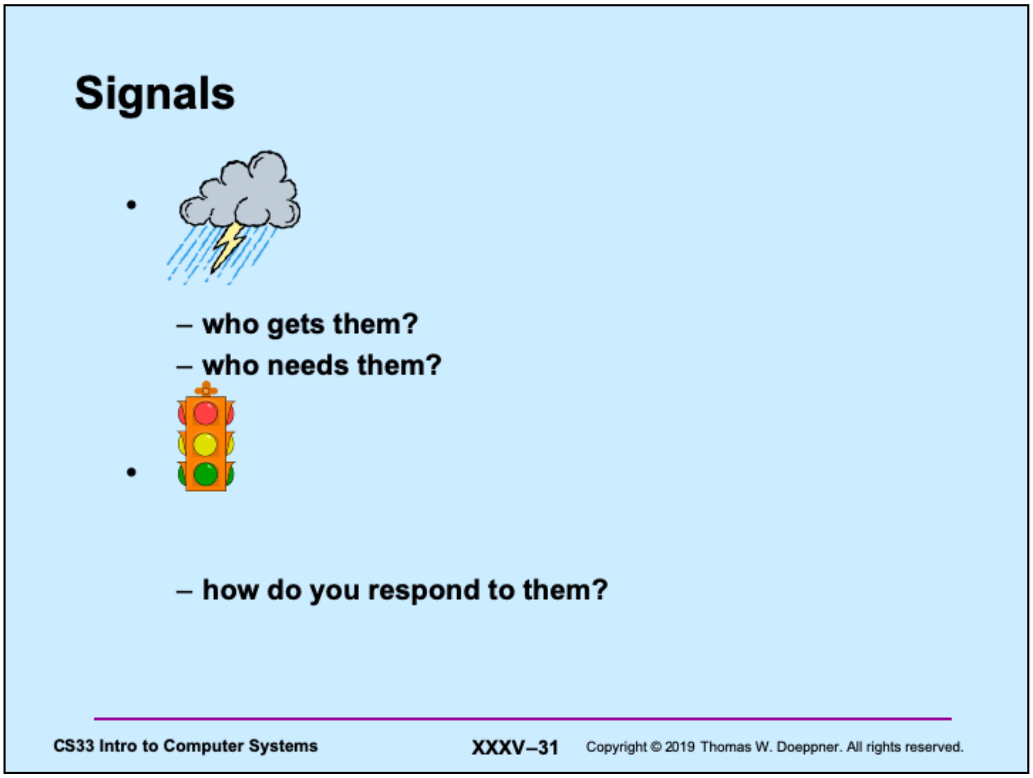
- Signals

  vs.

- Cancellation
  - tamed lightning

Deviations are things that modify a thread's normal flow of control. Unix has long had *signals*, and these must be dealt with in multithreaded improvements to Unix. There are actually two fairly different classes of signals: *asynchronous signals* and *synchronous signals*. The former are caused by events beyond the process's control, such as I/O events, clock events, system calls issued by other processes, etc. The latter are responses to what the current thread has just done, such as divide by zero, addressing exceptions, etc.

Cancellation is a new concept that pertains strictly to multithreaded programming. It is the means by which one thread can request the termination of another and provides a way for the terminating thread to terminate cleanly.



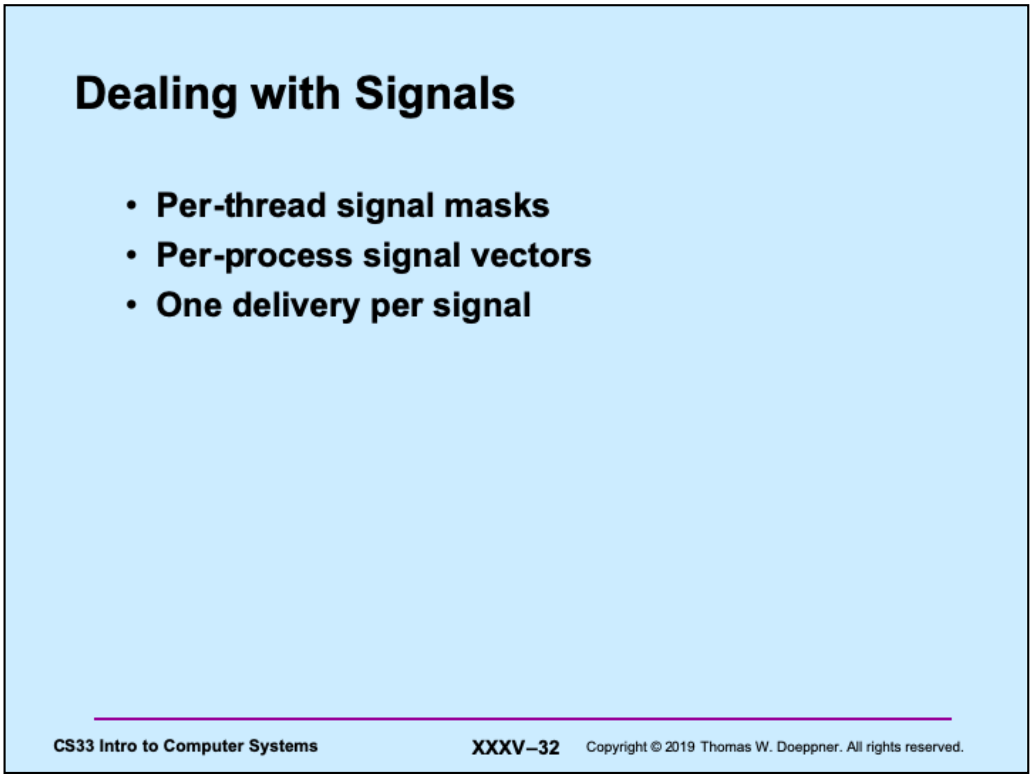
Signals

-
  - who gets them?
  - who needs them?
-
  - how do you respond to them?

Asynchronous signals were designed (like almost everything else) with single-threaded processes in mind. A signal is delivered to the process; if the signal is *caught*, the process stops whatever it is doing, deals with the signal, and then resumes normal processing. But what happens when a signal is delivered to a multithreaded process? Which thread or threads deal with it?

Asynchronous signals, by their very nature, are handled asynchronously. But one of the themes of multithreaded programming is that threads are a cure for asynchrony. Thus we should be able to use threads as a means of getting away from the “drop whatever you are doing and deal with me” approach to asynchronous signals.

Synchronous signals often are an indication that something has gone wrong: there really is no point continuing execution in this part of the program. Traditional Unix approaches for dealing with this bad news are not terribly elegant.



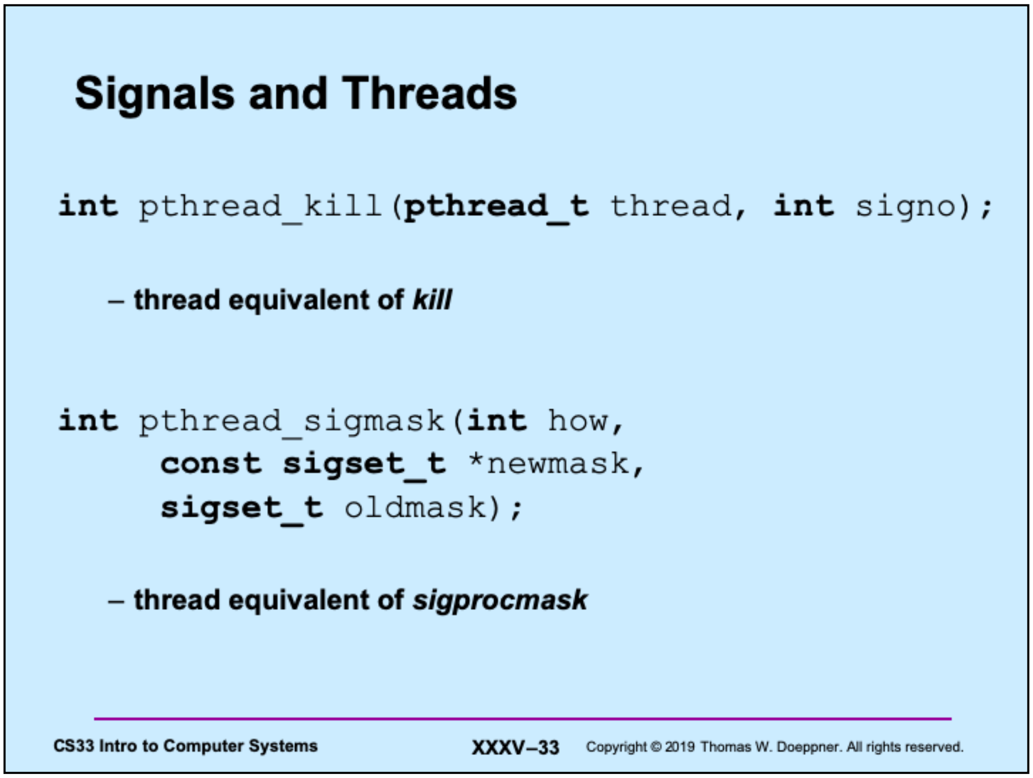
Dealing with Signals

- Per-thread signal masks
- Per-process signal vectors
- One delivery per signal

The standard Unix model has a process-wide signal mask and a vector indicating what is to be done in response to each kind of signal. When a signal is delivered to a process, an indication is made that this signal is pending. If the signal is unmasked, then the vector is examined to determine the response: to suspend the process, to resume the process, to terminate the process, to ignore the signal entirely, or to invoke a signal handler.

A number of issues arise in translating this model into a multithreaded-process model. First of all, if we invoke a signal handler, which thread or threads should execute the handler? What seems to be closest to the spirit of the original signal semantics is that exactly one thread should execute the handler. Which one? The consensus is that it really does not matter, just as long as exactly one thread executes the signal handler. But what about the signal mask? Since one sets masks depending on a thread's local behavior, it makes sense for each thread to have its own private signal mask. Thus a signal is delivered to any one thread that has the signal unmasked (if more than one thread has the signal unmasked, a thread is chosen randomly to handle the signal). If all threads have the signal masked, then the signal remains pending until some thread unmasks it.

A related issue is the vector indicating the response to each signal. Should there be one such vector per thread? If so, what if one thread specifies process termination in response to a signal, while another thread supplies a handler? For reasons such as this, it was decided that, even for multithreaded processes, there would continue to be a single, process-wide signal-disposition vector.



Signals and Threads

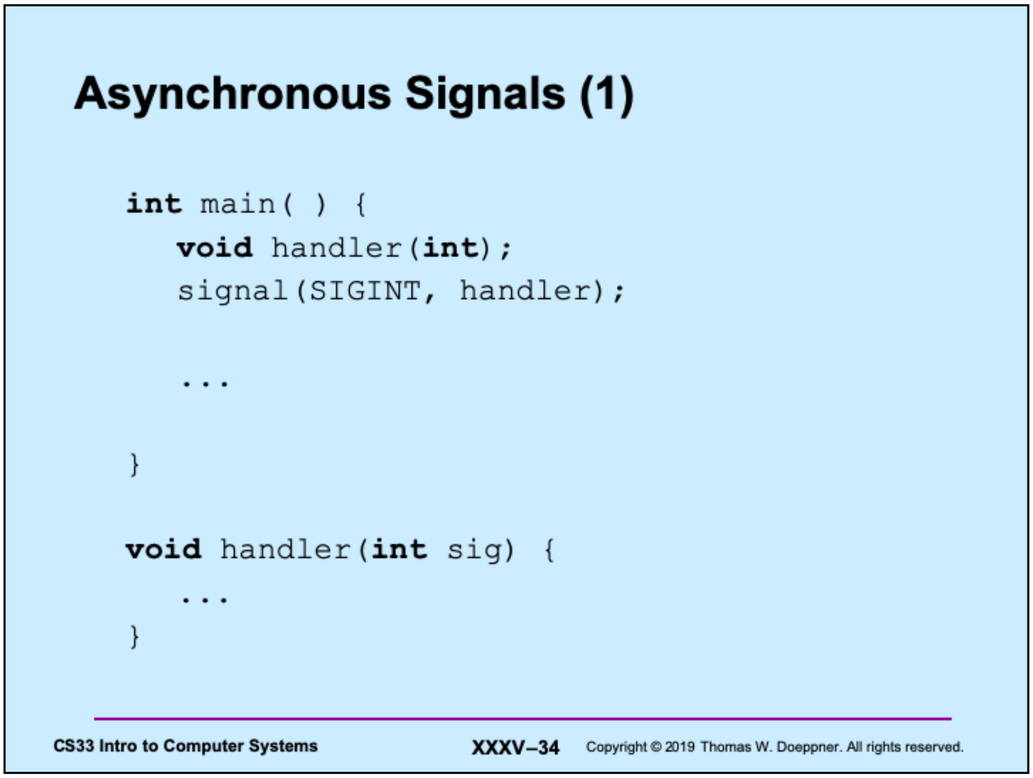
```
int pthread_kill(pthread_t thread, int signo);
```

– thread equivalent of *kill*

```
int pthread_sigmask(int how,
      const sigset_t *newmask,
      sigset_t oldmask);
```

– thread equivalent of *sigprocmask*

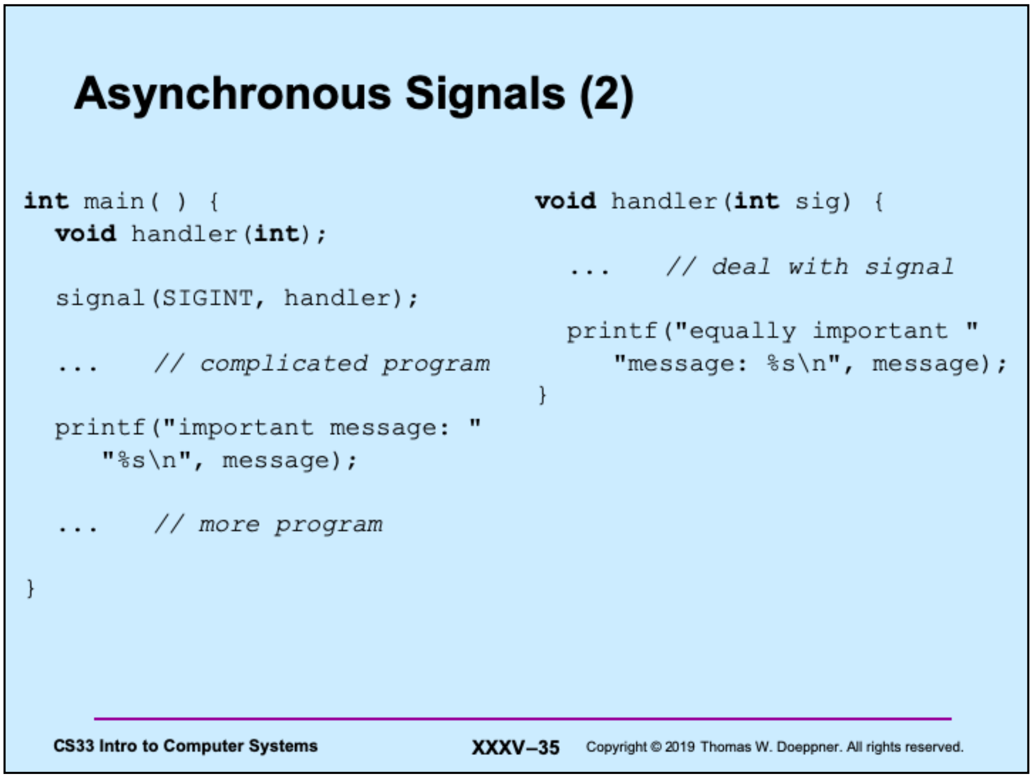
Signals may be sent to individual threads using *pthread_kill*. Though the targeted thread will handle the signal, the behavior is as set for the entire process (or clone group in Linux) using *sigaction*. Each thread may independently block and unblock signals using *pthread_sigmask*.



Asynchronous Signals (1)

```
int main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...

}

void handler(int sig) {
   ...
}
```

The slide shows the standard approach for dealing with signals: one sets up a handler that's invoked by the thread that received the signal.

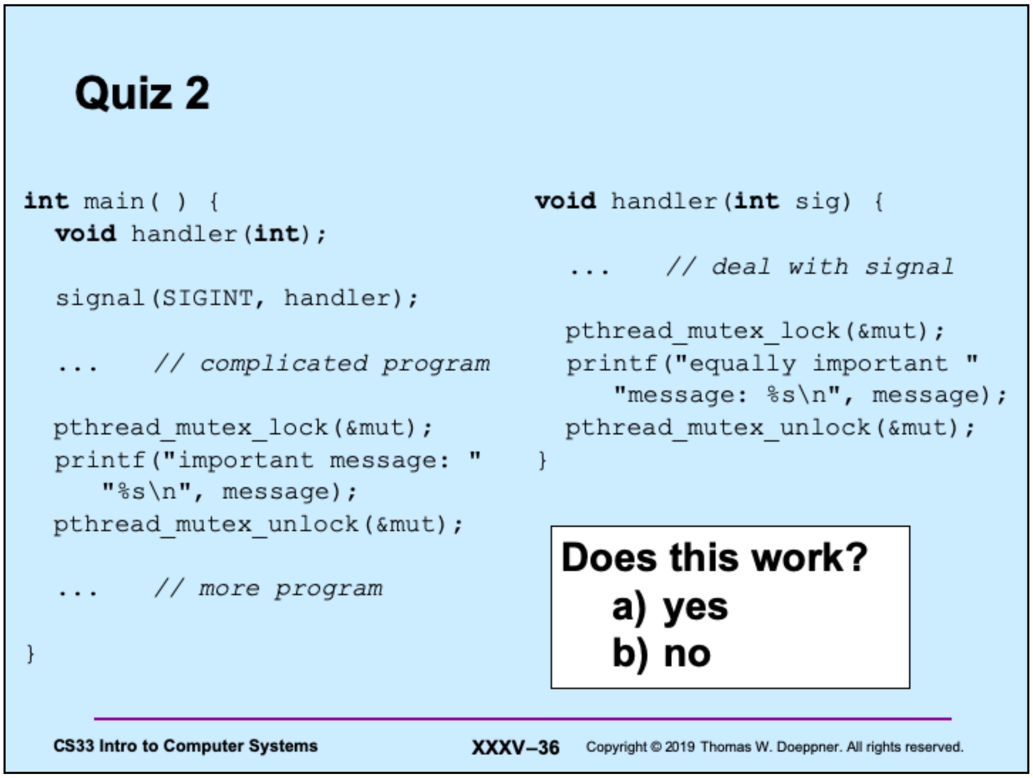


Asynchronous Signals (2)

```
int main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ...   // complicated program

  printf("important message: "
     "%s\n", message);

  ...   // more program

}
```

```
void handler(int sig) {

  ...   // deal with signal

  printf("equally important "
     "message: %s\n", message);
}
```

Here we have the example we saw a few weeks ago of the reason for requiring that signal handlers call only async-signal-safe functions.



Quiz 2

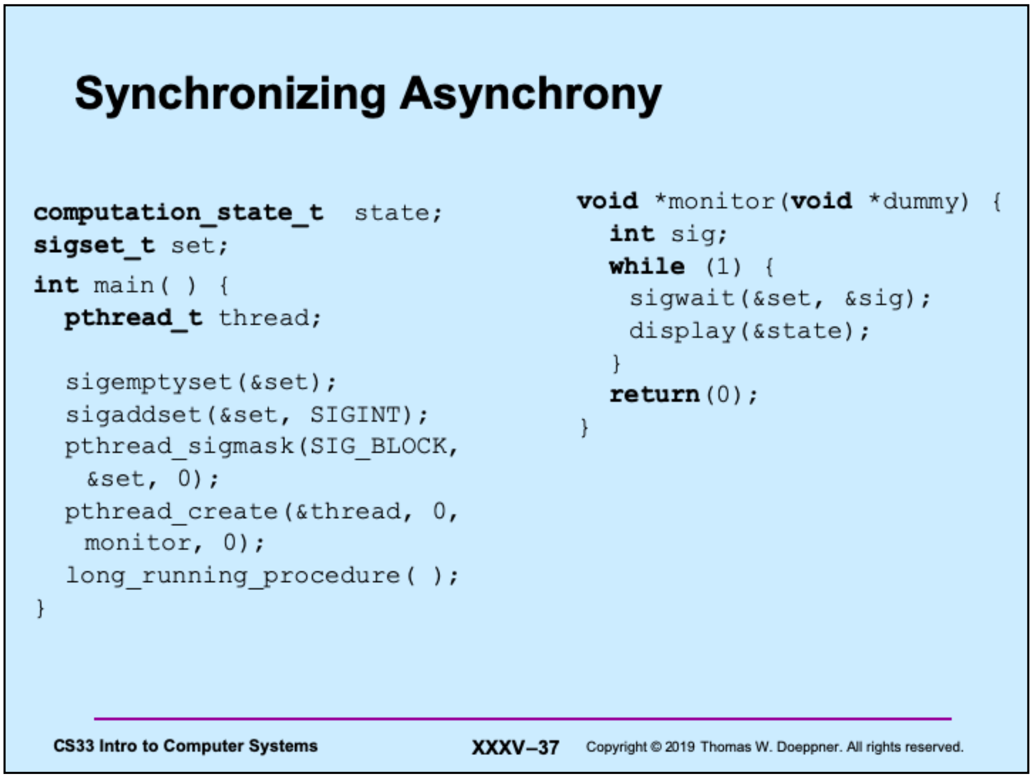
```
int main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ...    // complicated program

  pthread_mutex_lock(&mut);
  printf("important message: "
     "%s\n", message);
  pthread_mutex_unlock(&mut);

  ...    // more program

}
```

```
void handler(int sig) {

  ...    // deal with signal

  pthread_mutex_lock(&mut);
  printf("equally important "
     "message: %s\n", message);
  pthread_mutex_unlock(&mut);
}
```

Does this work?

- a) yes
- b) no

Here we have the example we saw a few weeks ago of the reason for requiring that signal handlers call only async-signal-safe functions.



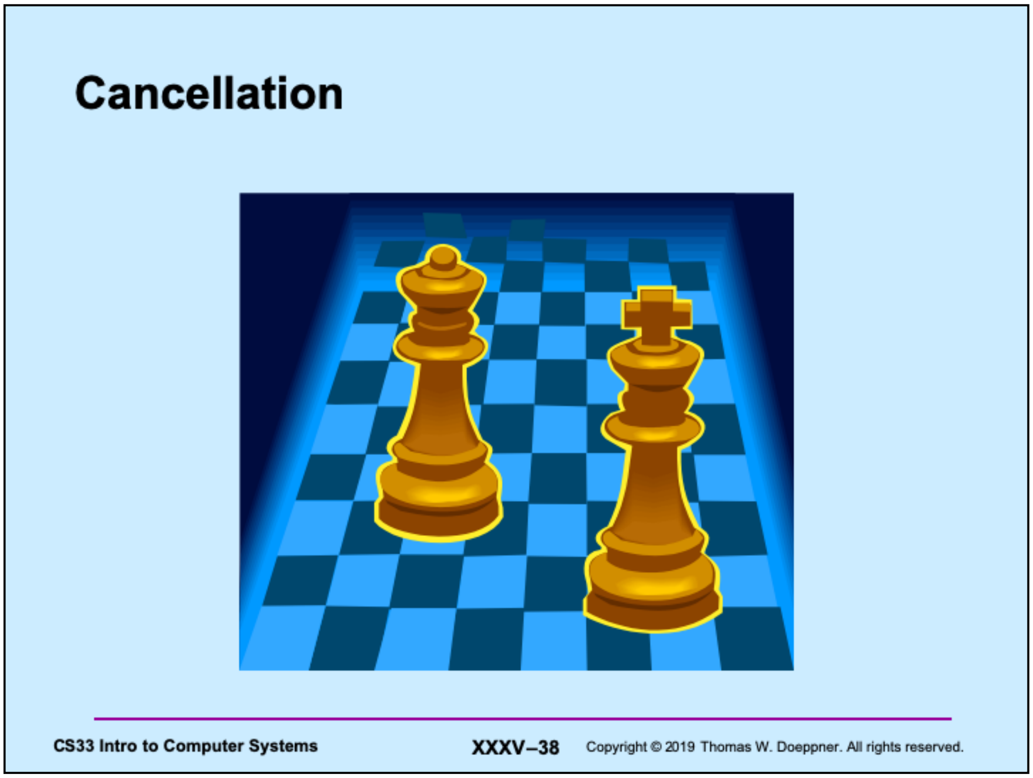
Synchronizing Asynchrony

```
computation_state_t  state;
sigset_t set;
int main( ) {
  pthread_t thread;

  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  pthread_sigmask(SIG_BLOCK,
   &set, 0);
  pthread_create(&thread, 0,
   monitor, 0);
  long_running_procedure( );
}
```

```
void *monitor(void *dummy) {
  int sig;
  while (1) {
   sigwait(&set, &sig);
   display(&state);
  }
  return(0);
}
```

Here we use a different technique for dealing with the signal. Rather than have the thread performing the long-running computation be interrupted by the signal, we dedicate a thread to dealing with the signal. We make use of a new signal-handling routine, *sigwait*. This routine puts its caller to sleep until one of the signals specified in its argument occurs, at which point the call returns and the number of the signal that occurred is stored in the location pointed to by the second argument. As is done here, *sigwait* is normally called with the signals of interest masked off; *sigwait* responds to signals even if they are masked. (Note also that a new thread inherits the signal mask of its creator.)

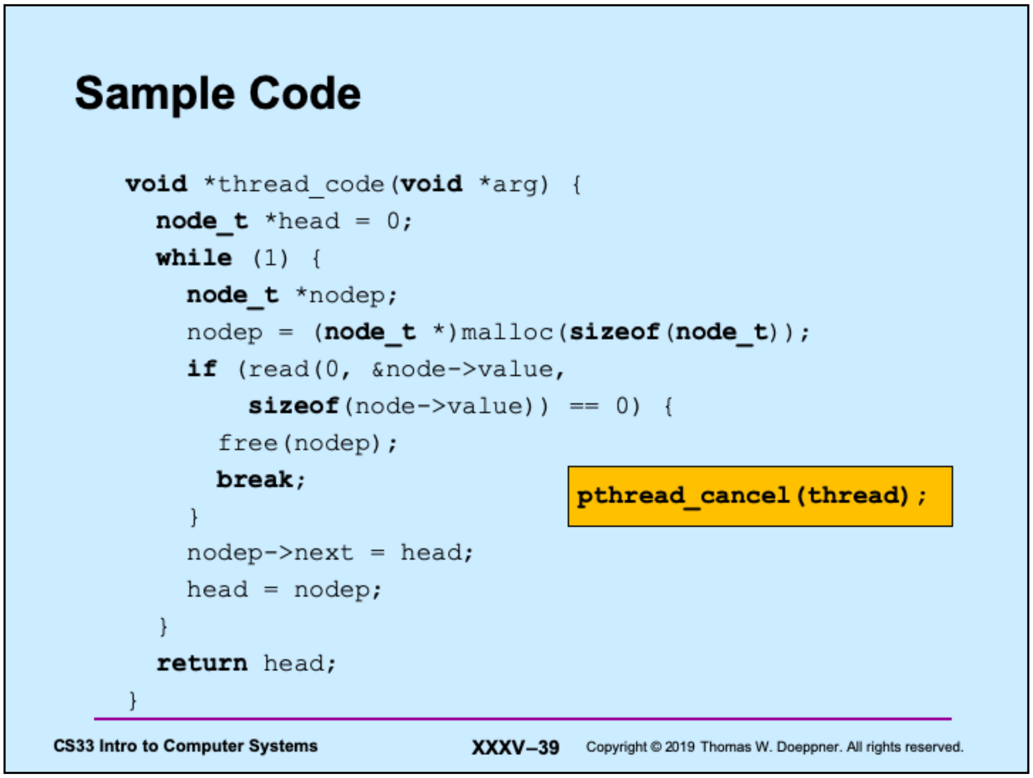
Among the advantages of this approach is that there are no concerns about async-signal safety since a signal handler is never invoked. The signal-handling thread waits for signals synchronously — it is not interrupted. Thus it is safe for it to use even mutexes, condition variables, and semaphores from inside of the *display* routine. Another advantage is that, if this program is run on a multiprocessor, the “signal handling” can run in parallel with the mainline code, which could not happen with the previous approach.



Cancellation

In a number of situations one thread must tell another to cease whatever it is doing. For example, suppose we've implemented a chess-playing program by having multiple threads search the solution space for the next move. If one thread has discovered a quick way of achieving a checkmate, it would want to notify the others that they should stop what they're doing, the game has been won.

One might think that this is an ideal use for per-thread signals, but there's a cleaner mechanism for doing this sort of thing in POSIX threads, called *cancellation*.



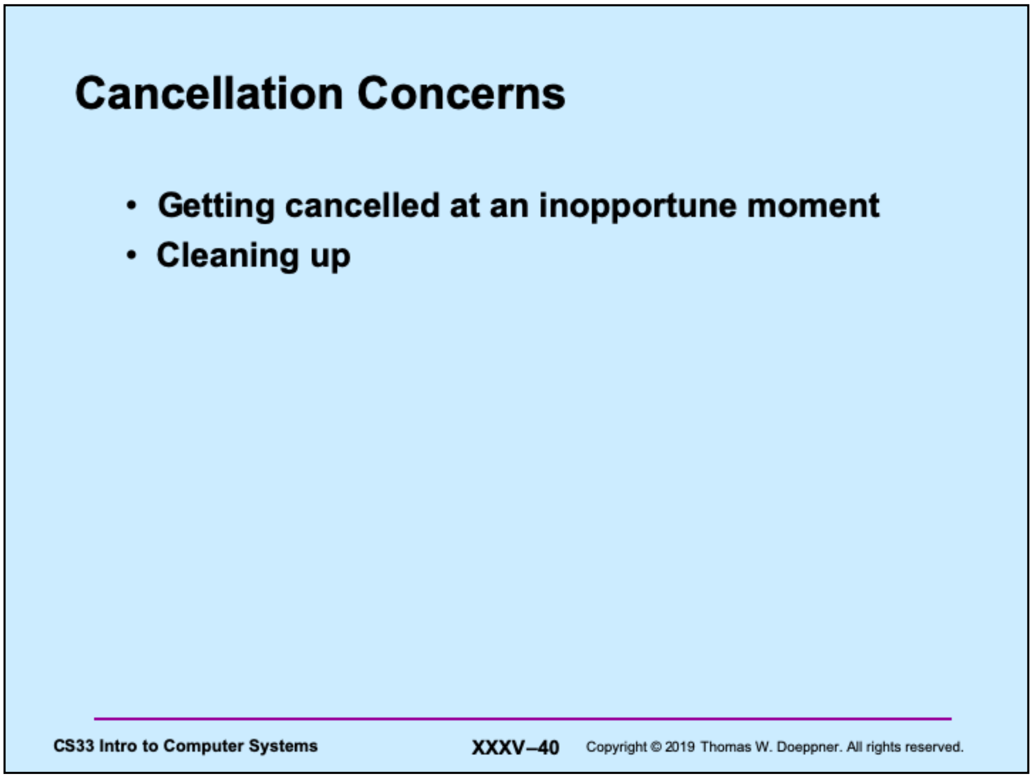
Sample Code

```
void *thread_code(void *arg) {
  node_t *head = 0;
  while (1) {
    node_t *nodep;
    nodep = (node_t *)malloc(sizeof(node_t));
    if (read(0, &node->value,
        sizeof(node->value)) == 0) {
      free(nodep);
      break;
    }
    nodep->next = head;
    head = nodep;
  }
  return head;
}
```

```
pthread_cancel(thread);
```

This code is invoked by a thread (as its first function). The thread reads values from stdin, which it then puts into a singly linked list that it allocates on the fly, and returns a pointer to the list.

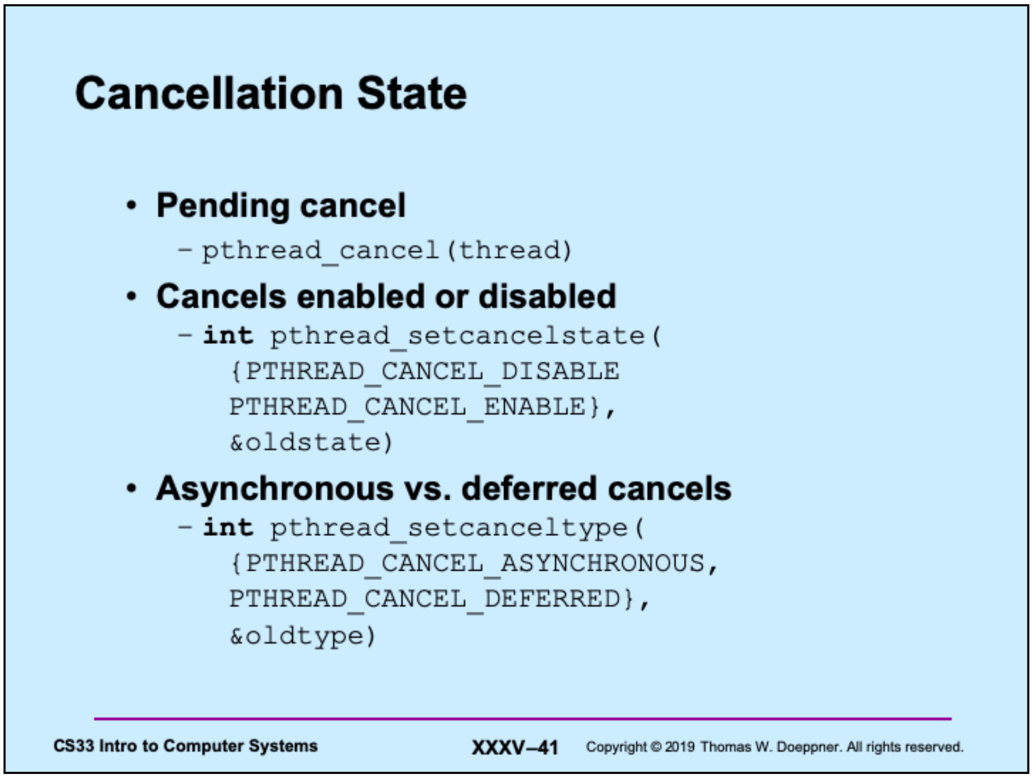
Suppose our thread is forced to terminate in the midst of its execution (some other thread invokes the operation *pthread_cancel* on it). What sort of problems might ensue?



Cancellation Concerns

- Getting cancelled at an inopportune moment
- Cleaning up

We have two concerns about the forced termination of threads resulting from cancellation: a thread might be in the middle of doing something important that it must complete before self-destructing; and a canceled thread must be given the opportunity to clean up.



Cancellation State

- **Pending cancel**
  - `pthread_cancel(thread)`
- **Cancels enabled or disabled**

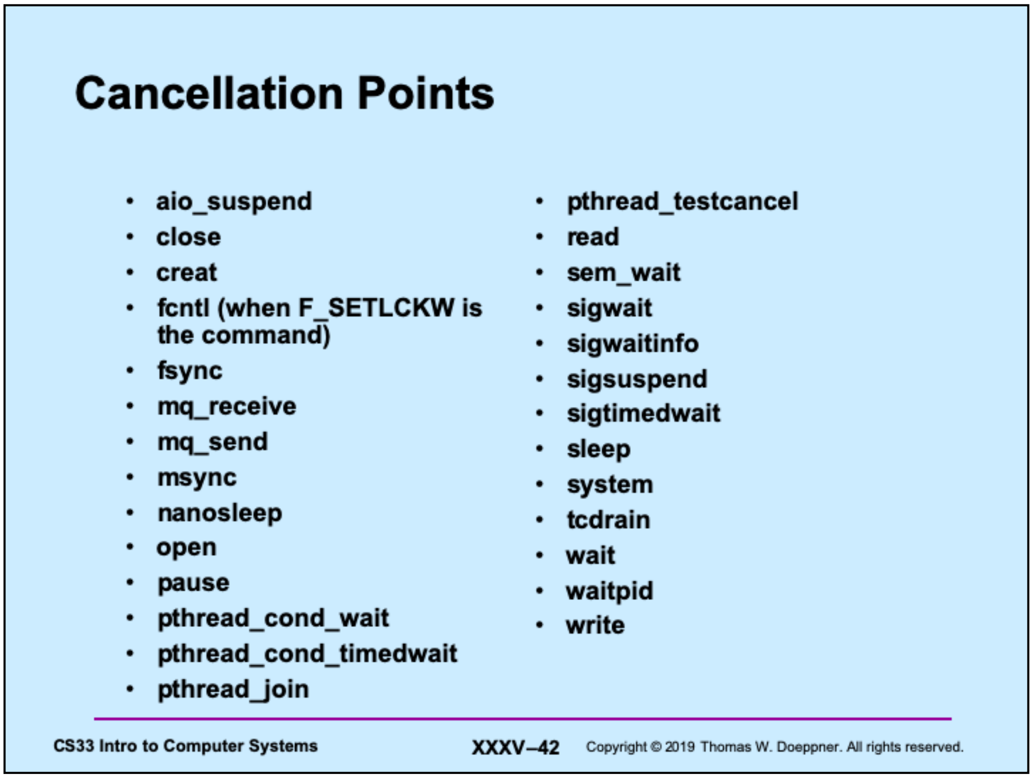
  -
    ```
    int pthread_setcancelstate(
      {PTHREAD_CANCEL_DISABLE
      PTHREAD_CANCEL_ENABLE},
      &oldstate)
    ```
- **Asynchronous vs. deferred cancels**

  -
    ```
    int pthread_setcanceltype(
      {PTHREAD_CANCEL_ASYNCHRONOUS,
      PTHREAD_CANCEL_DEFERRED},
      &oldtype)
    ```

A thread issues a cancel request by calling *pthread_cancel*, supplying the ID of the target thread as the argument. Associated with each thread is some state information known as its *cancellation state* and its *cancellation type*. When a thread receives a cancel request, it is marked indicating that it has a pending cancel. The next issue is when the thread should notice and act upon the cancel. This is governed by the cancellation state: whether cancels are *enabled* or *disabled* and by the cancellation type: whether the response to cancels is *asynchronous* or *deferred*. If cancels are *disabled*, then the cancel remains pending but is otherwise ignored until cancels are enabled. If cancels are *enabled*, they are acted on as soon as they are noticed if the cancellation type is *asynchronous*. Otherwise, i.e., if the cancellation type is *deferred*, the cancel is acted upon only when the thread reaches a *cancellation point*.

Cancellation points are intended to be well defined points in a thread's execution at which it is prepared to be canceled. They include pretty much all system and library calls in which the thread can block, with the exception of *pthread_mutex_lock*. In addition, a thread may call *pthread_testcancel*, which has no function other than being a cancellation point.

The default is that cancels are enabled and deferred. One can change the cancellation state of a thread by using the routines shown in the slide. Calls to *pthread_setcancelstate* and *pthread_setcanceltype* return the previous value of the affected portion of the cancellability state.

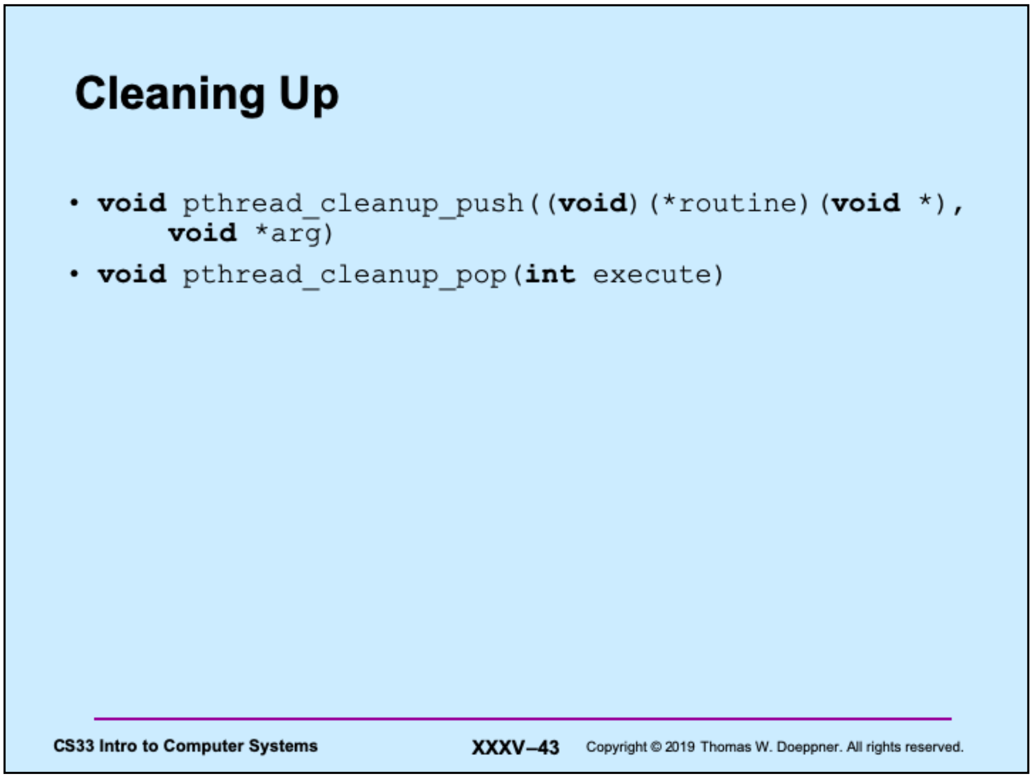


Cancellation Points

- aio_suspend
- close
- creat
- fcntl (when F_SETLCKW is the command)
- fsync
- mq_receive
- mq_send
- msync
- nanosleep
- open
- pause
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_join
- pthread_testcancel
- read
- sem_wait
- sigwait
- sigwaitinfo
- sigsuspend
- sigtimedwait
- sleep
- system
- tcdrain
- wait
- waitpid
- write

The slide lists all of the required cancellation points in POSIX.

The routine *pthread_testcancel* is strictly a cancellation point — it has no other function. If there are no pending cancels when it is called, it does nothing and simply returns.

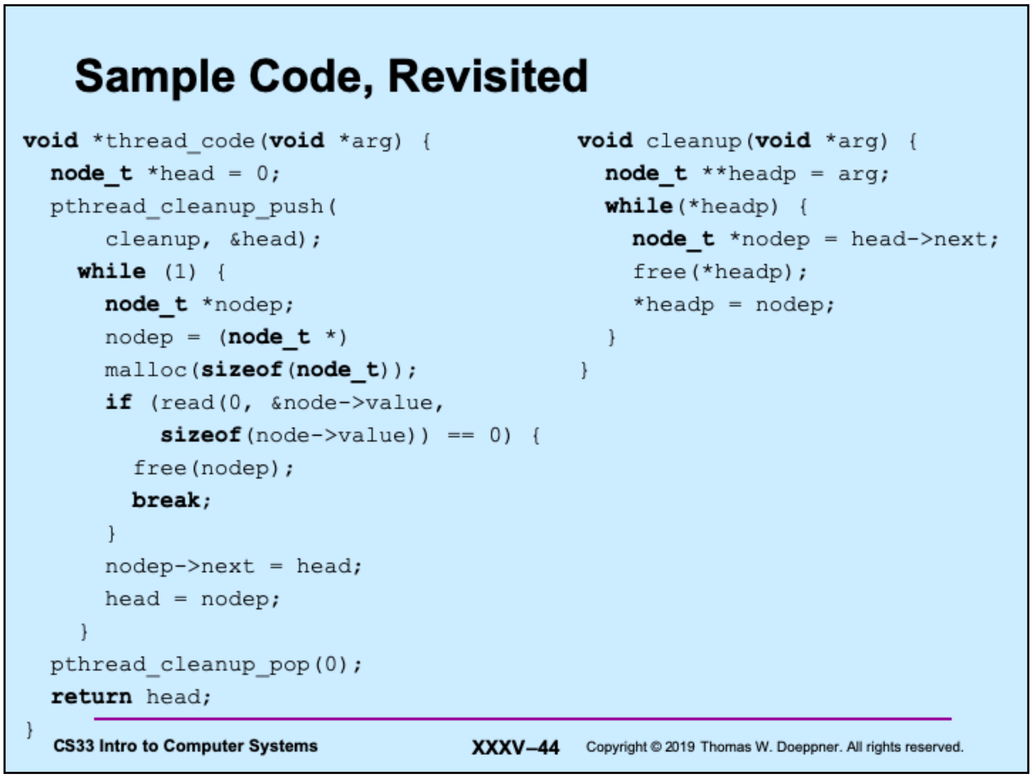


Cleaning Up

- `void pthread_cleanup_push((void)(*routine)(void *),`
  `void *arg)`
- `void pthread_cleanup_pop(int execute)`

When a thread acts upon a cancel, its ultimate fate has been established, but it first gets a chance to clean up. Associated with each thread may be a stack of *cleanup handlers*. Such handlers are pushed onto the stack via calls to *pthread_cleanup_push* and popped off the stack via calls to *pthread_cleanup_pop*. Thus when a thread acts on a cancel or when it calls *pthread_exit*, it calls each of the cleanup handlers in turn, giving the argument that was supplied as the second parameter of *pthread_cleanup_push*. Once all the cleanup handlers have been called, the thread terminates.

The two routines *pthread_cleanup_push* and *pthread_cleanup_pop* are intended to act as left and right parentheses, and thus should always be paired (in fact, they may actually be implemented as macros: the former contains an unmatched “{“, the latter an unmatched “}”). The argument to the latter routine indicates whether or not the cleanup function should be called as a side effect of calling *pthread_cleanup_pop*.

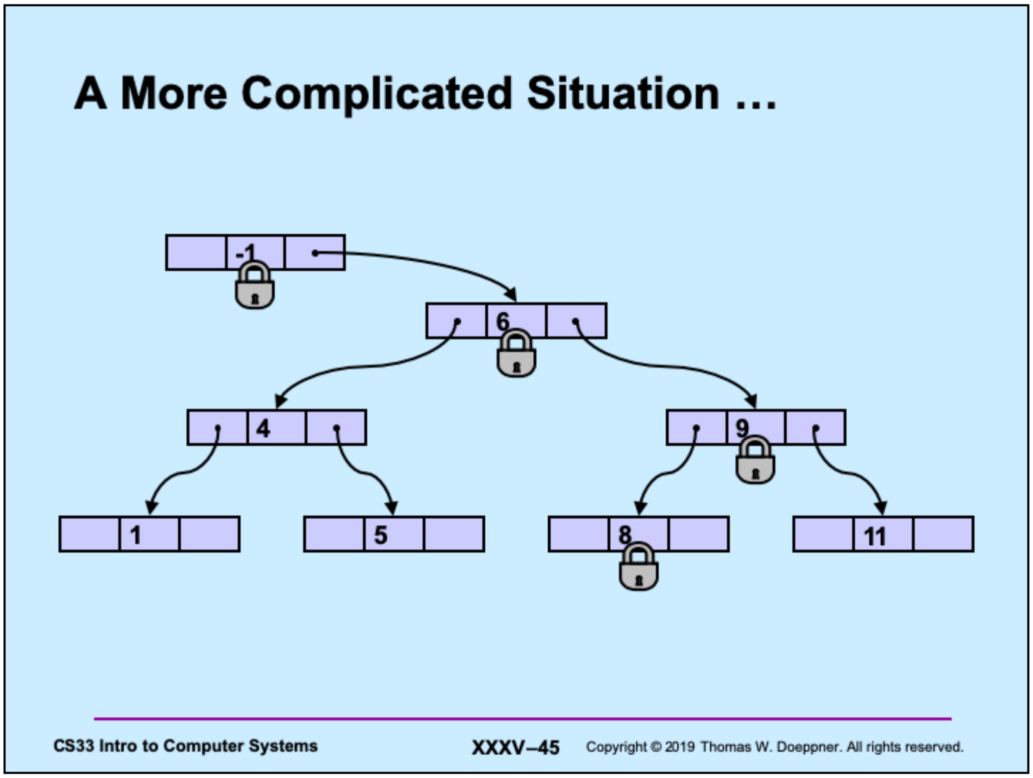


Sample Code, Revisited

```
void *thread_code(void *arg) {
  node_t *head = 0;
  pthread_cleanup_push(
      cleanup, &head);
    while (1) {
      node_t *nodep;
      nodep = (node_t *)
      malloc(sizeof(node_t));
      if (read(0, &node->value,
          sizeof(node->value)) == 0) {
        free(nodep);
        break;
      }
      nodep->next = head;
      head = nodep;
    }
  pthread_cleanup_pop(0);
  return head;
}
```

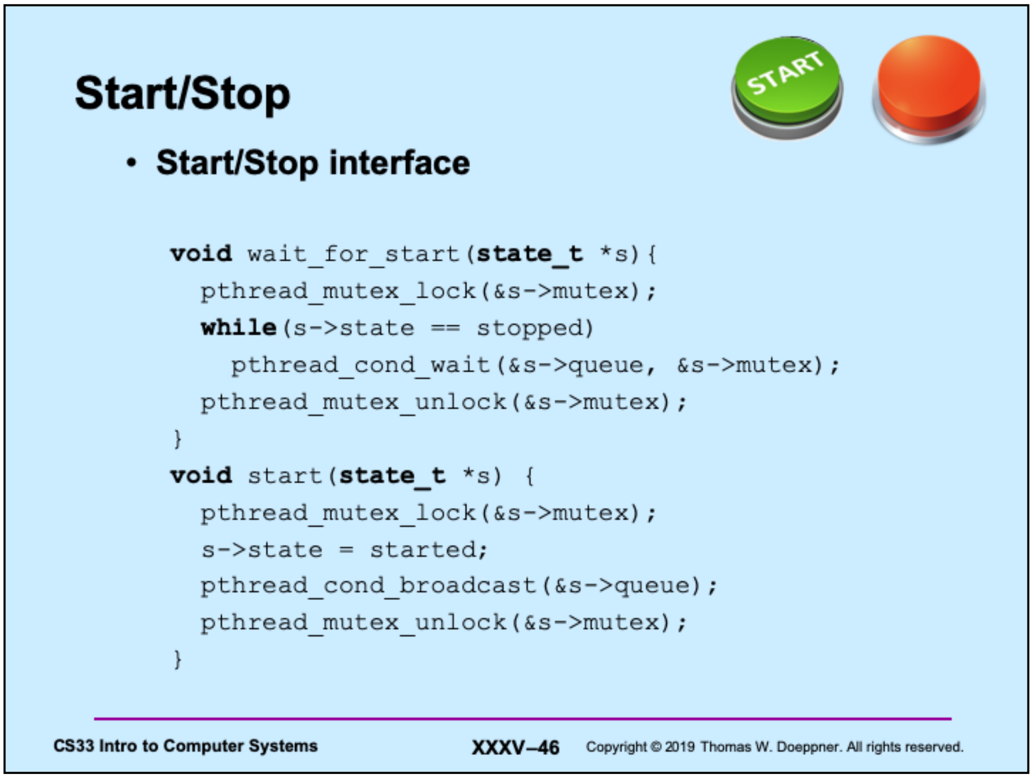
```
void cleanup(void *arg) {
  node_t **headp = arg;
  while(*headp) {
    node_t *nodep = head->next;
    free(*headp);
    *headp = nodep;
  }
}
```

Here we’ve added a cleanup handler to our sample code. Note that our example has just one cancellation point: *read*. The cleanup handler iterates through the list, deleting each element.



A More Complicated Situation …

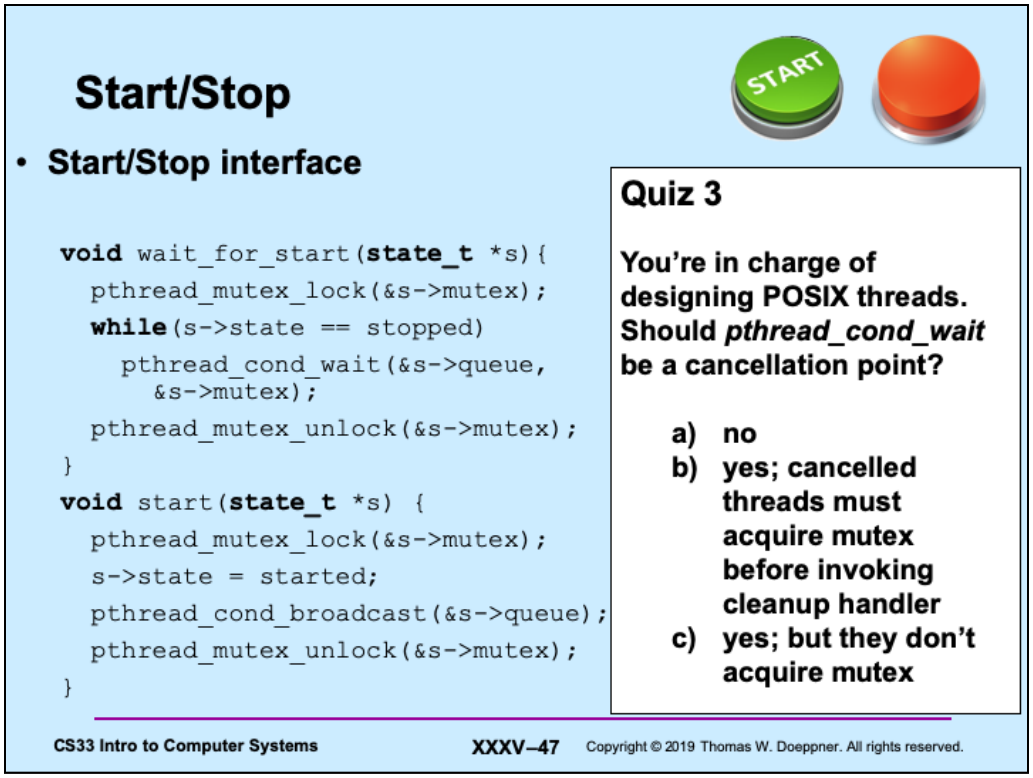
Whether threads are using mutexes or readers/writers locks when manipulating a search tree, if we have to deal with cancellation points in the middle of such operations, things can get pretty complicated and error-prone. Thus the operations to lock mutexes and readers/writers locks are not cancellation points. (Note, however, that for the case of readers/writers locks, POSIX permits waiting for readers/writers locks to be cancellation points, for the sake of vendors who have poor implementations of them. Neither Linux nor OSX implements such waiting as cancellation points.)



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);
}
```



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);
}
```

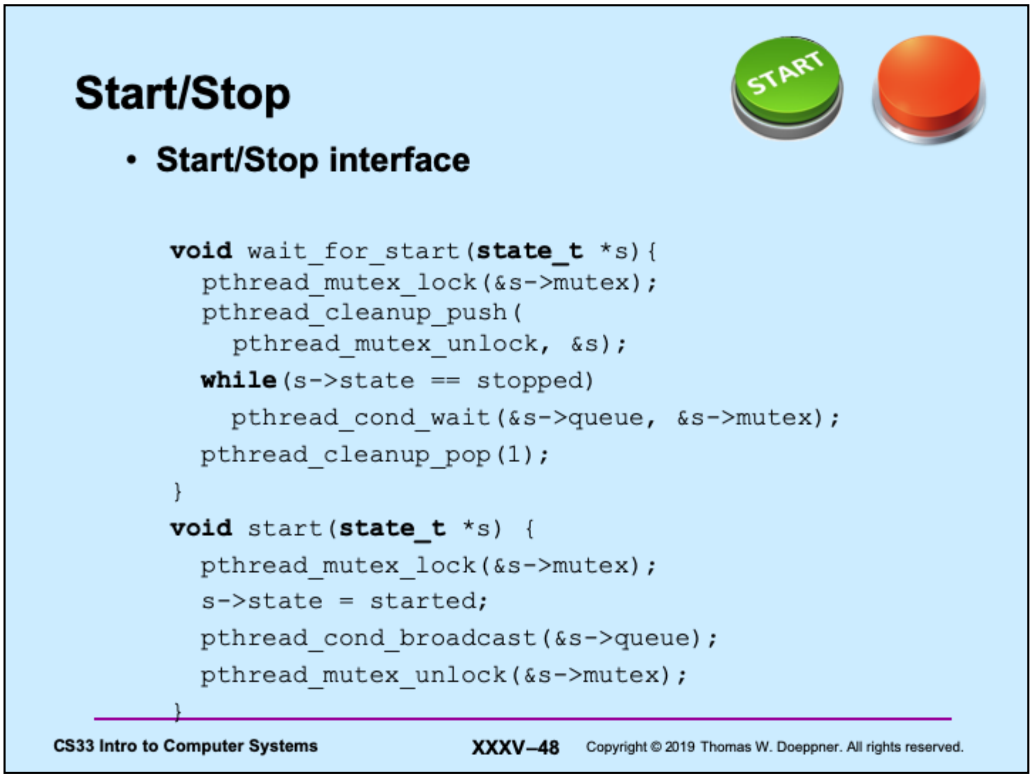
Quiz 3

You're in charge of designing POSIX threads. Should *pthread_cond_wait* be a cancellation point?

a) no

b) yes; cancelled threads must acquire mutex before invoking cleanup handler

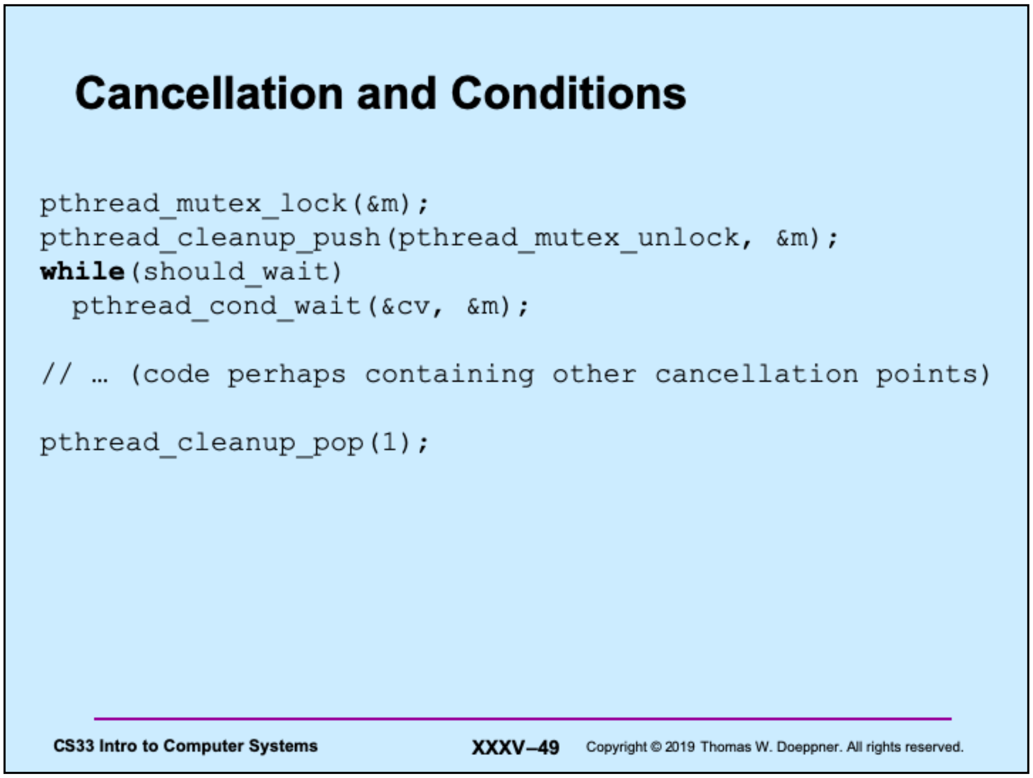
c) yes; but they don't acquire mutex



Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  pthread_cleanup_push(
    pthread_mutex_unlock, &s);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_cleanup_pop(1);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```



Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);
while(should_wait)
  pthread_cond_wait(&cv, &m);

// … (code perhaps containing other cancellation points)

pthread_cleanup_pop(1);
```

In this example we handle cancels that might occur while a thread is blocked within *pthread_cond_wait*. Again we assume the thread has cancels enabled and deferred. The thread first pushes a cleanup handler on its stack — in this case the cleanup handler unlocks the mutex. The thread then loops, calling *pthread_cond_wait*, a cancellation point. If it receives a cancel, the cleanup handler won't be called until the mutex has been reacquired. Thus we are certain that when the cleanup handler is called, the mutex is locked.

What's important here is that we make sure the thread does not terminate without releasing its lock on the mutex *m*. If the thread acts on a cancel within *pthread_cond_wait* and the cleanup handler were invoked without first taking the mutex, this would be difficult to guarantee, since we wouldn't know if the thread had the mutex locked (and thus needs to unlock it) when it's in the cleanup handler.