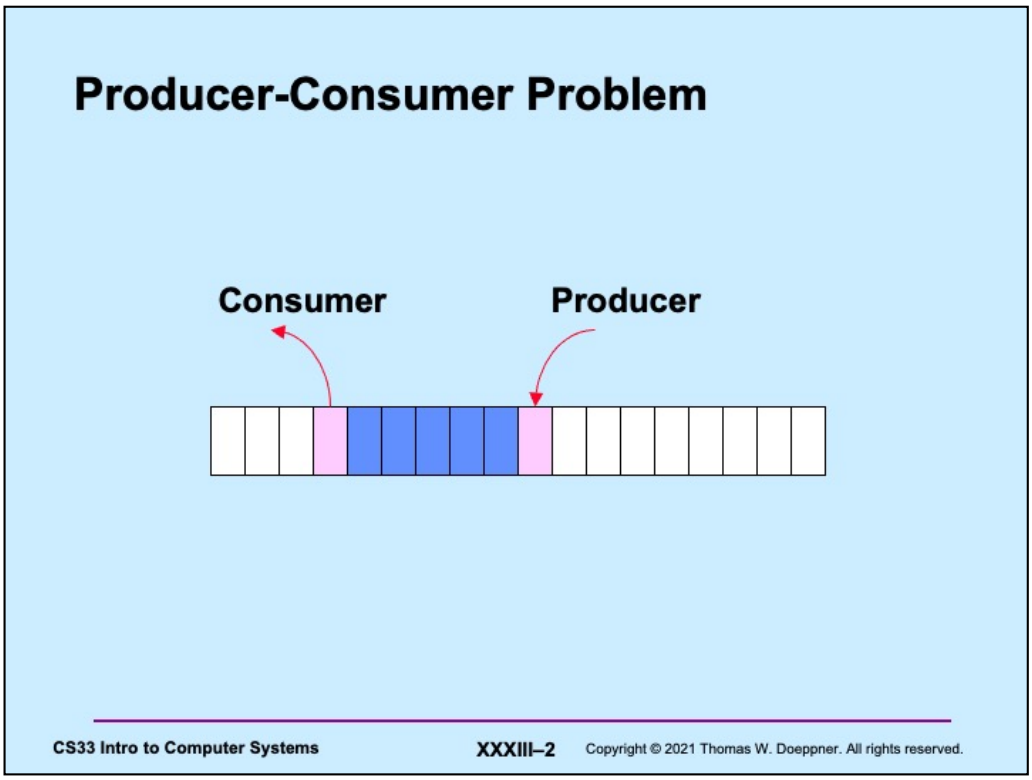
CS 33

Multithreaded Programming III



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.

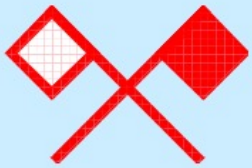
Guarded Commands

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

- **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, 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.

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**

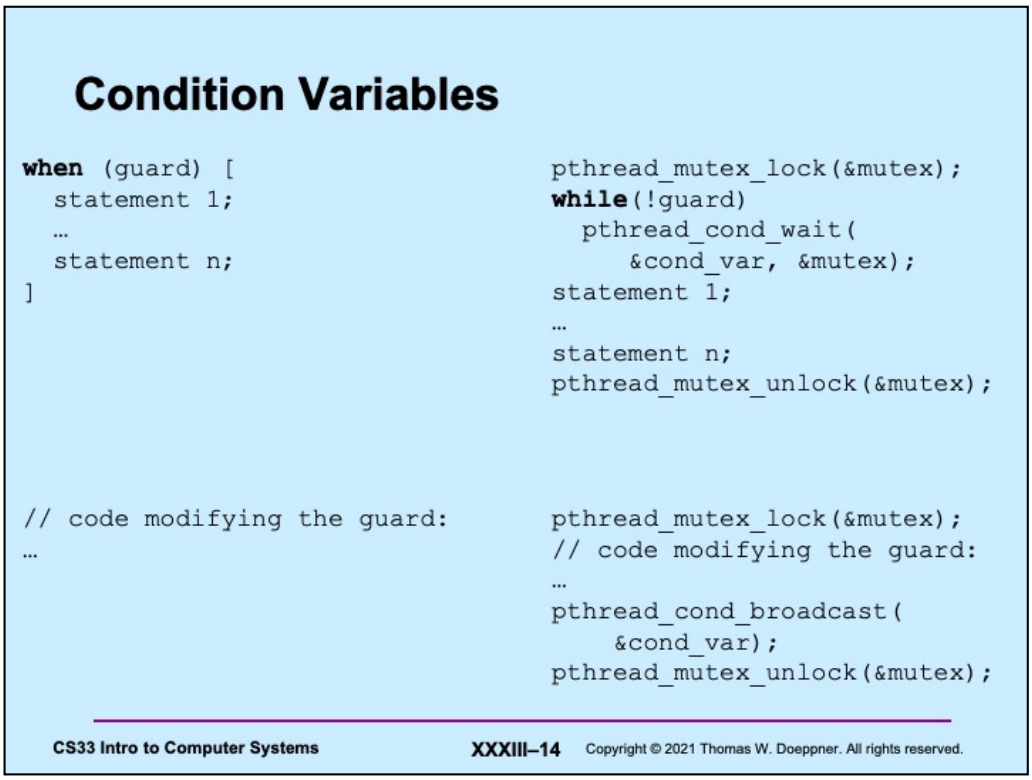
```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);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


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 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 any waiting consumers 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 to pthread_cond_wait 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( ) {
  when (writers == 0) [
   readers++;
  ]

  /* read */

  [readers--;]
}
```

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

  /* write */

  [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.

New Pseudocode

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

  /* read */

  [readers--;]
}
```

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

  /* write */

  [writers--;
  active_writers--;]
}
```


It turns out that our solution to the readers-writers problem has a flaw: writers may have to wait indefinitely before being allowed to write. This is because as long as there is a reader reading, further readers are allowed in, and writers are prevented from writing.

Though one might argue that the best solution is one that is fair to both readers and writers, what is usually preferred is one that favors writers — i.e., readers requesting permission to read must yield to writers, but writers do not yield to readers.

This slide gives pseudocode using guarded commands for a new solution to the problem, a writers-priority solution. Writers indicate their intention to write by incrementing *writers*. We use the variable **active_writers** to indicate how many writers are currently writing.

Improved Reader

```
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);
}
```


In this slide we've taken the pseudocode for the writers-priority reader and translated it into legal POSIX.

Improved Writer

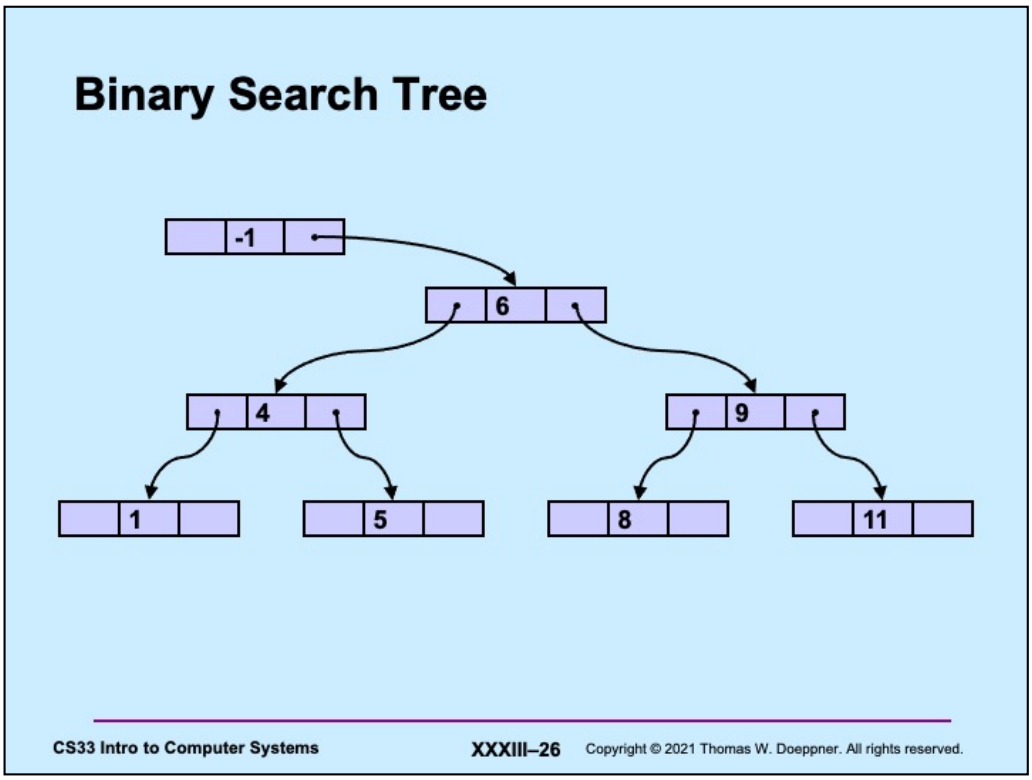
```
writer( ) {
  pthread_mutex_lock(&m);

  writers++;
  while (!((readers == 0) &&
        (active_writers == 0))) {
   pthread_cond_wait(
         &writersQ, &m);
  }
  active_writers++;

  pthread_mutex_unlock(&m);

  /* write */

  pthread_mutex_lock(&m);

  writers--;
    active_writers--;
  if (writers)
   pthread_cond_signal(
         &writersQ);
  else
   pthread_cond_broadcast(
         &readersQ);

  pthread_mutex_unlock(&m);
}
```


Here's the POSIX version of the writer code. Note the use of **pthread_cond_broadcast**: we use it to ensure that all currently waiting readers are released.

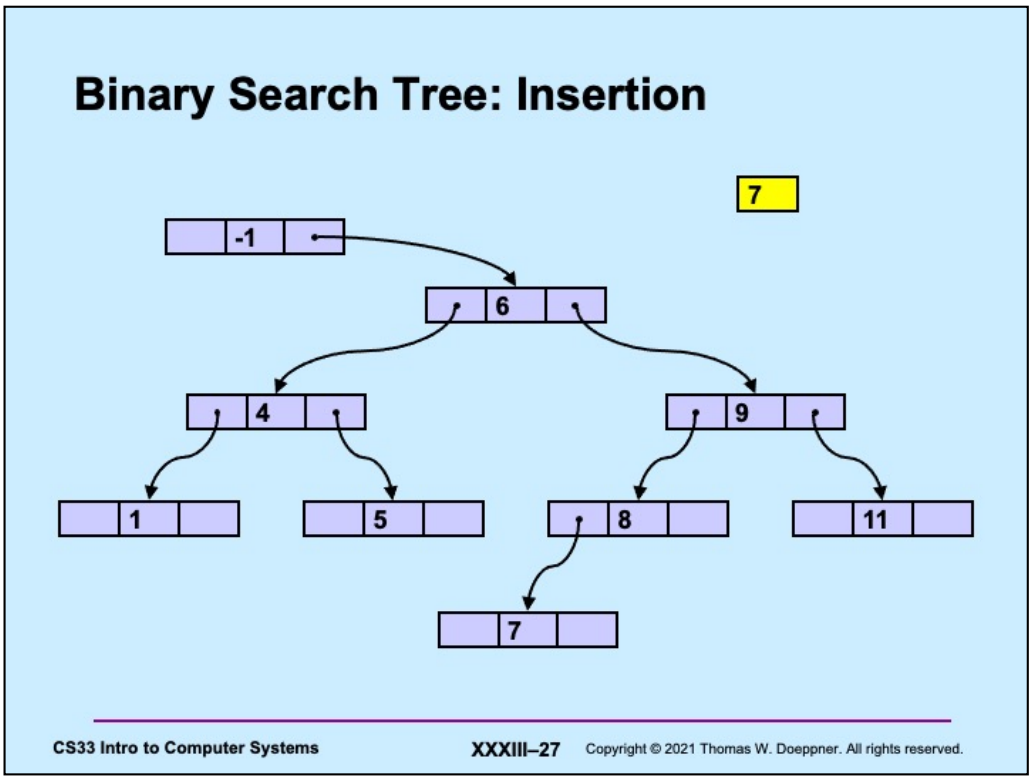
New, From POSIX!

```
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);
```

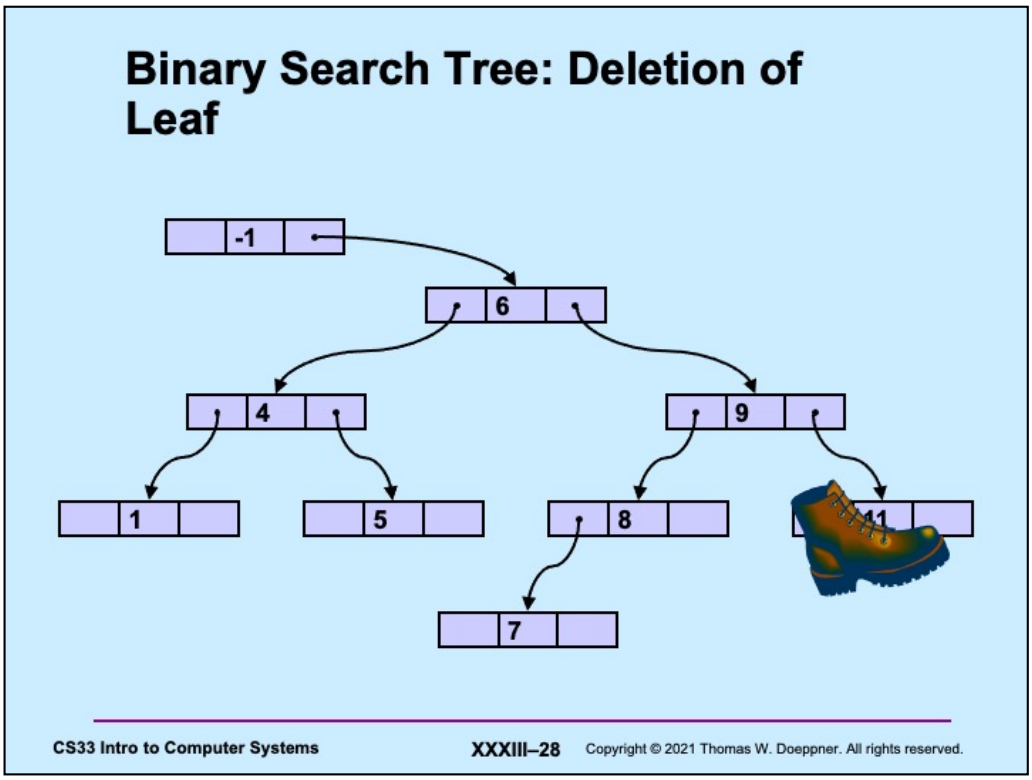

With POSIX 1003.1j support for readers-writers locks was finally introduced. The almost complete API is shown in the slide (what's missing are the operations on attributes). As might be expected, readers-writers locks can be statically initialized with the constant PTHREAD_RWLOCK_INITIALIZER. The "timedrwlock"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



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.

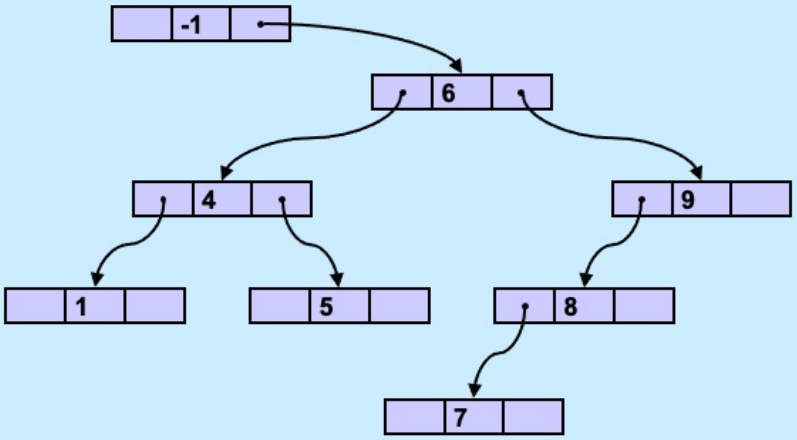


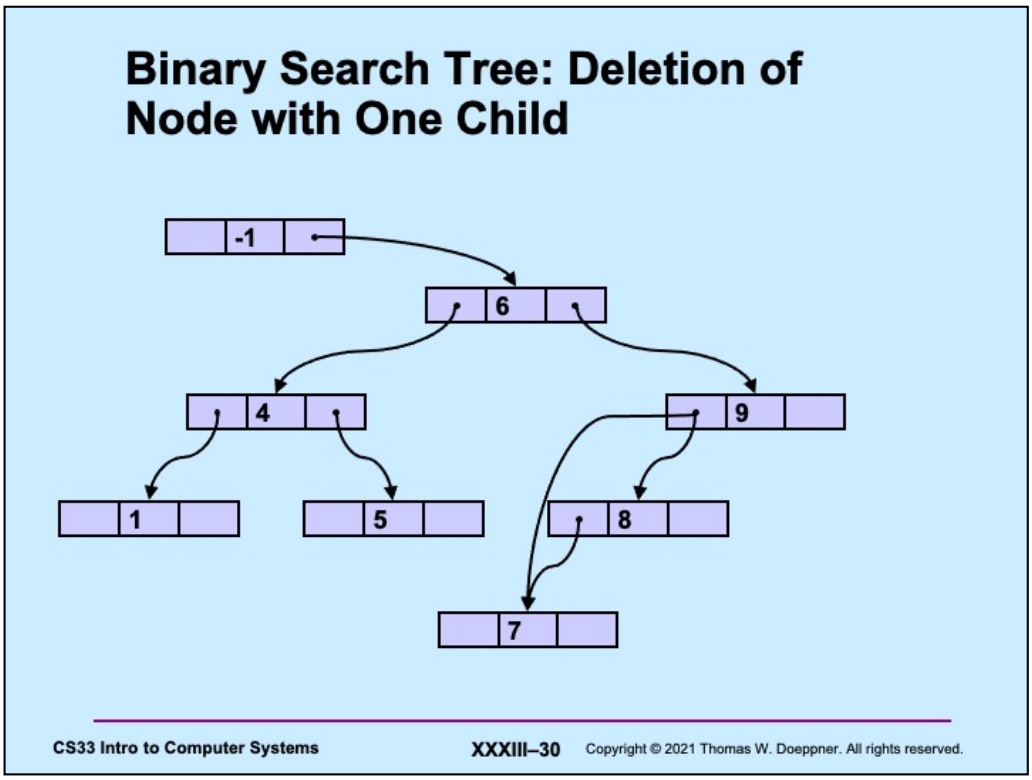
To add a new node to the tree, say one whose key will be 7, we start at the head and trace our way down the tree, comparing the new key with the keys of tree nodes, following left or right child pointers as appropriate. A new node is always inserted as a leaf.



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

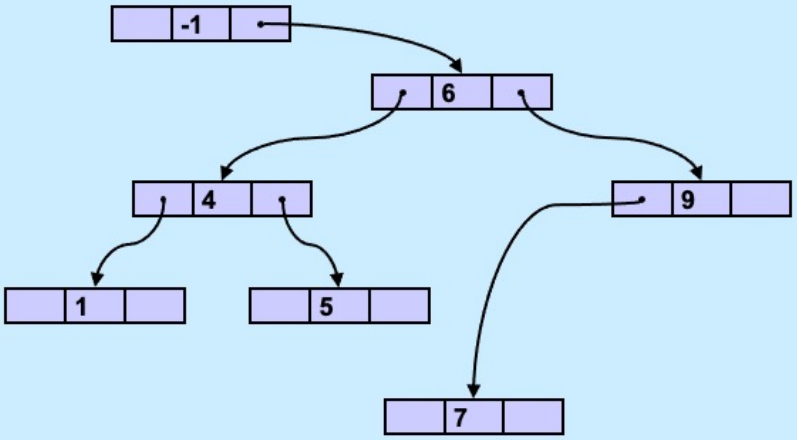
Binary Search Tree: Deletion of Leaf

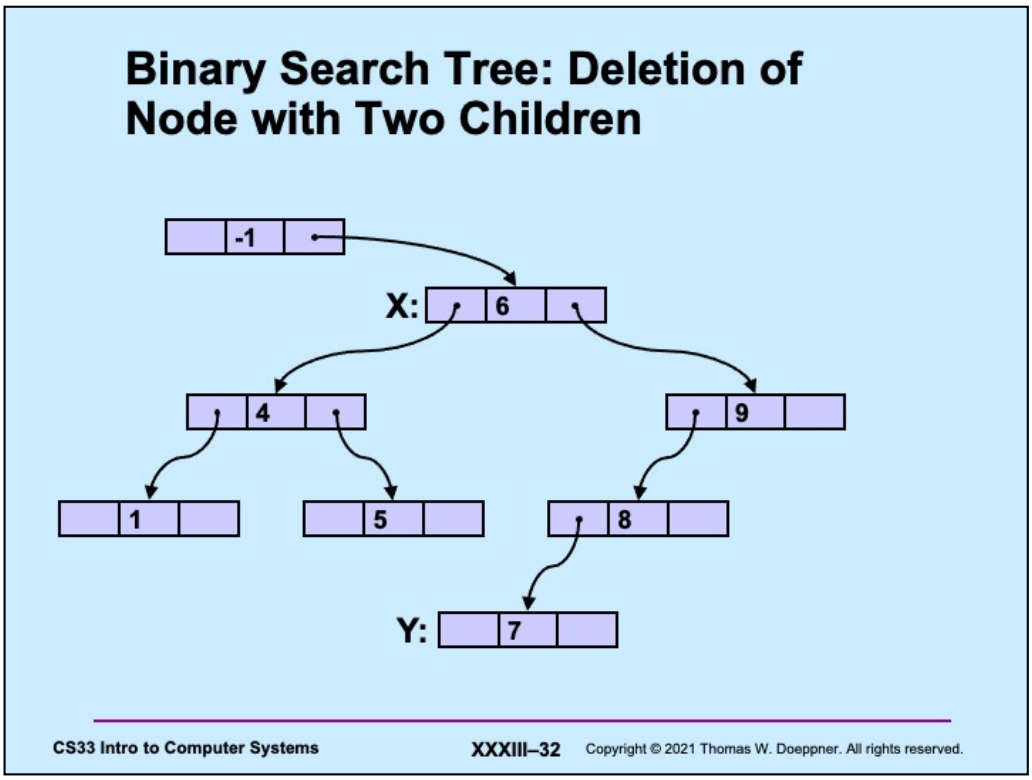




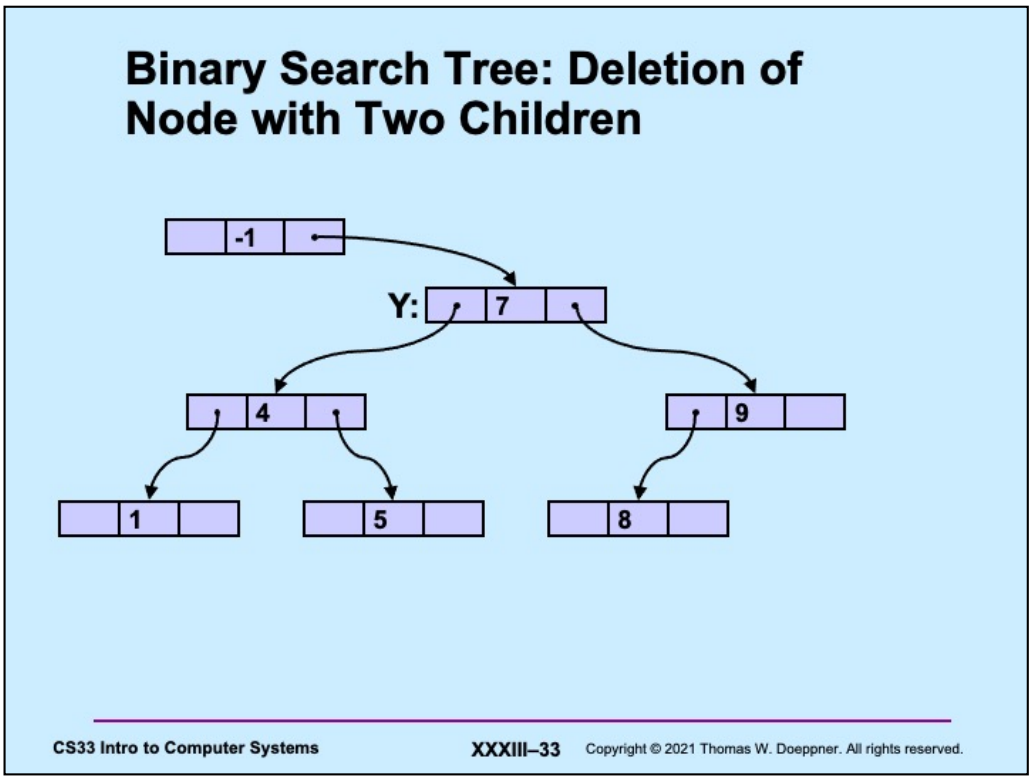
Deleting an interior node that has just one child is almost as easy. The child pointer from its parent is changed to point to the node's child, and then the node is deleted.

Binary Search Tree: Deletion of Node with One Child





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.



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

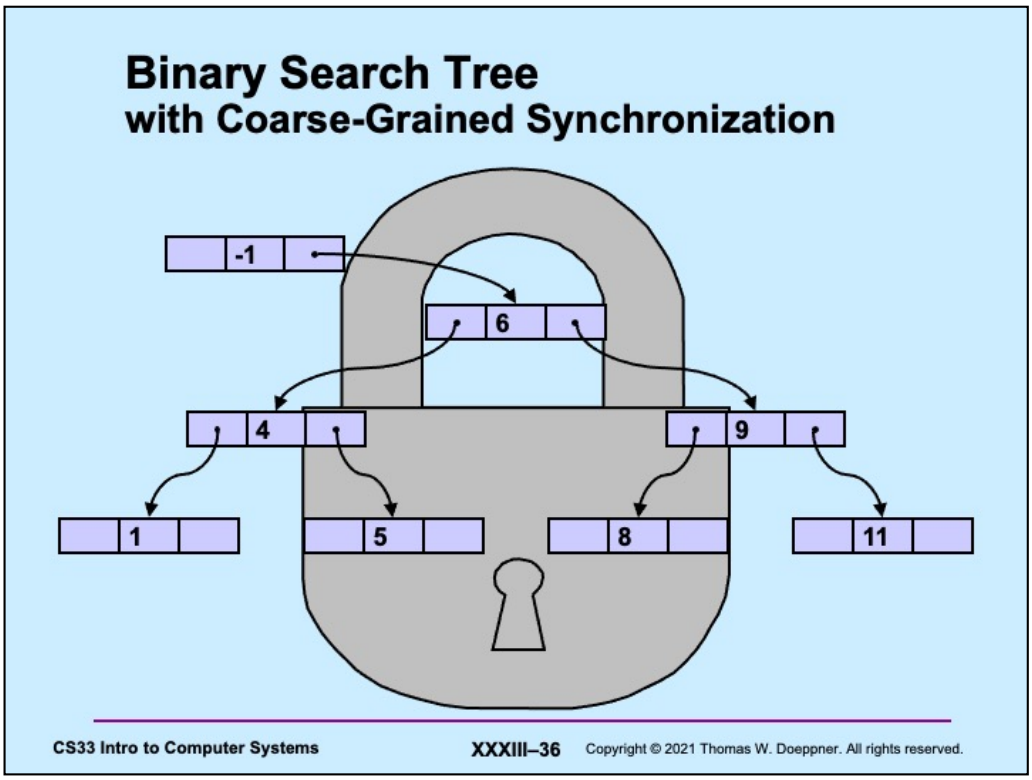
```
Node *search(int key,
    Node *parent, Node **parentp) {
  Node *next;
  Node *result;
  if (key < parent->key) {
    if ((next = parent->lchild)
        == 0) {
      result = 0;
    } else {
      if (key == next->key) {
        result = next;
      } else {
        result = search(key,
     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

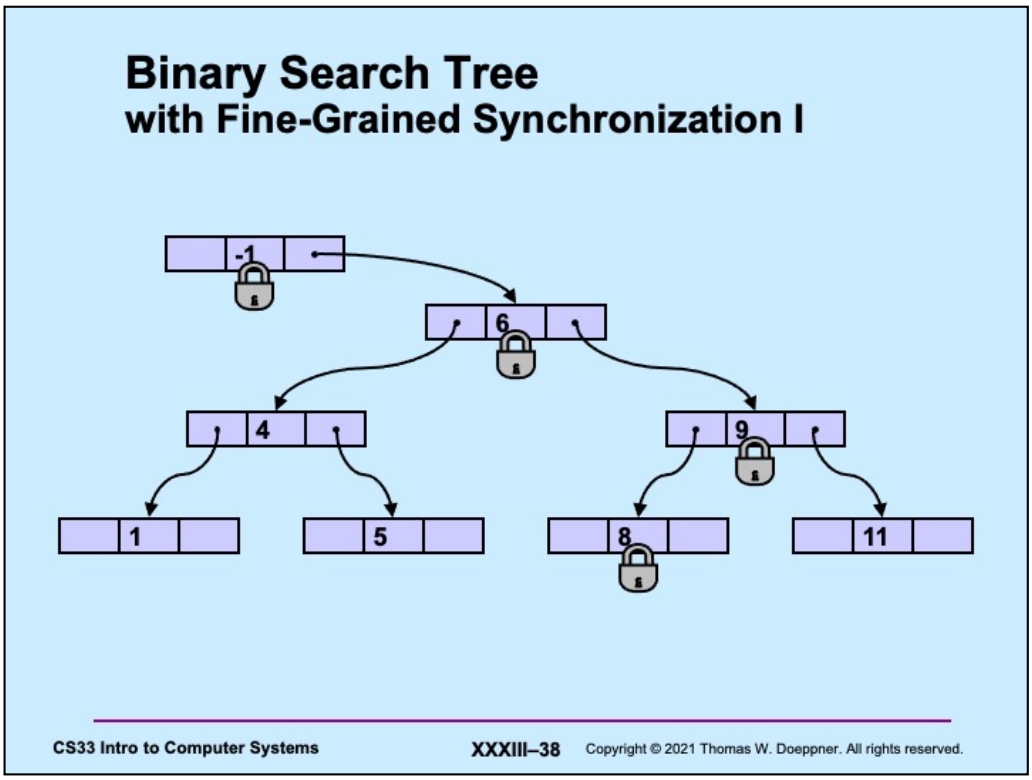

Here's the C code for adding a node to the binary search tree.



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.

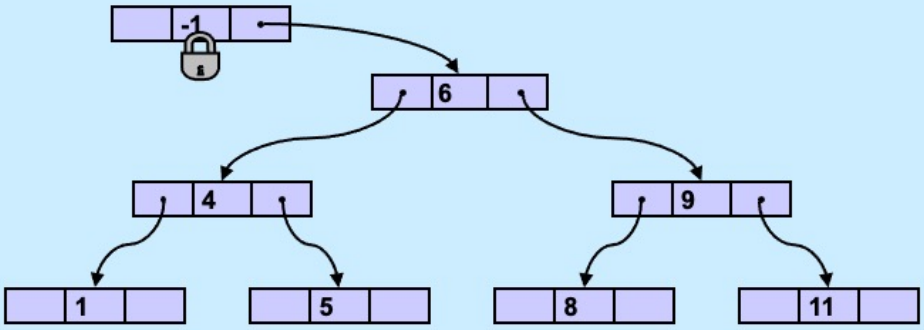
C Code: Add
with Coarse-Grained Synchronization

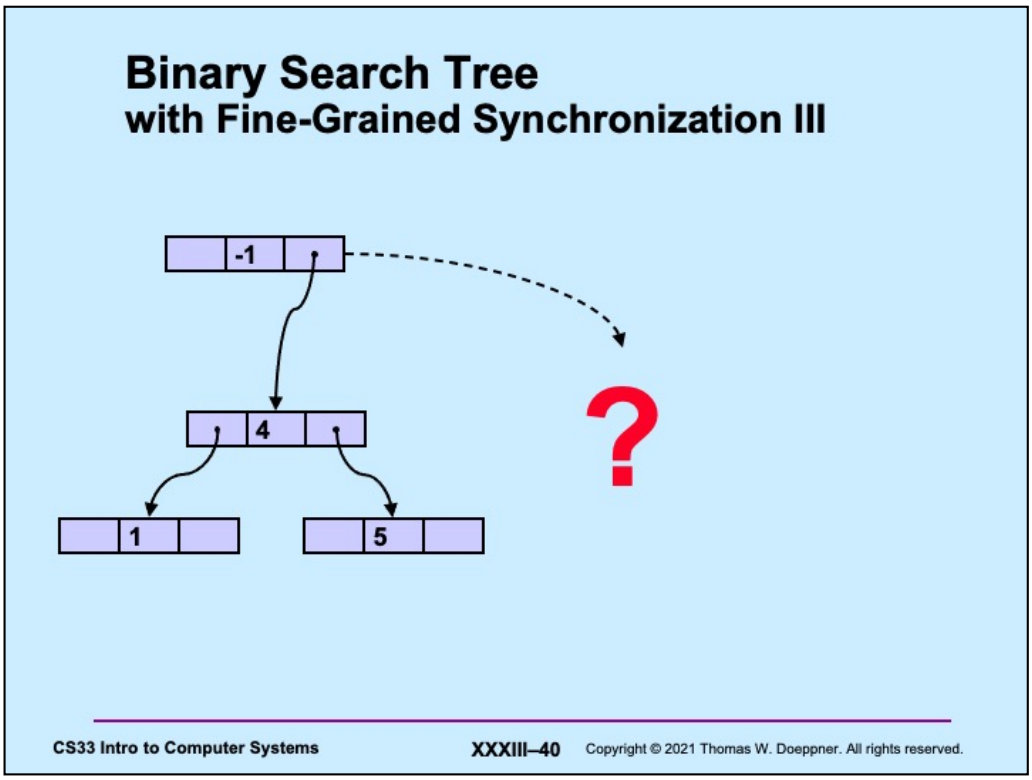
```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&tree_lock);
  if ((target = search(key, &head, &parent)) != 0) {
    pthread_rwlock_unlock(&tree_lock);
    return 0;
  }
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;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

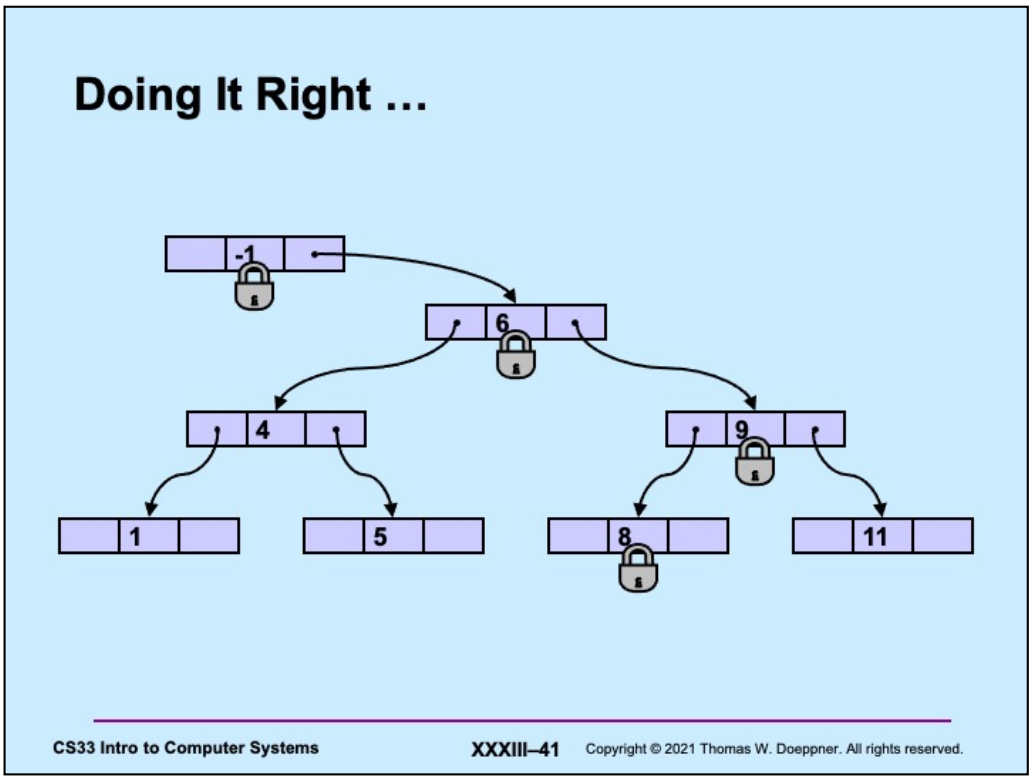
Let's now look at what's known as **fine-grained synchronization**, where we associate a readers-writers lock with each node of the tree. The idea is that, unlike the case for coarse-grained synchronization, we can have multiple threads working on different parts of the tree at once. The first step in 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





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).



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 function. 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;
}
```

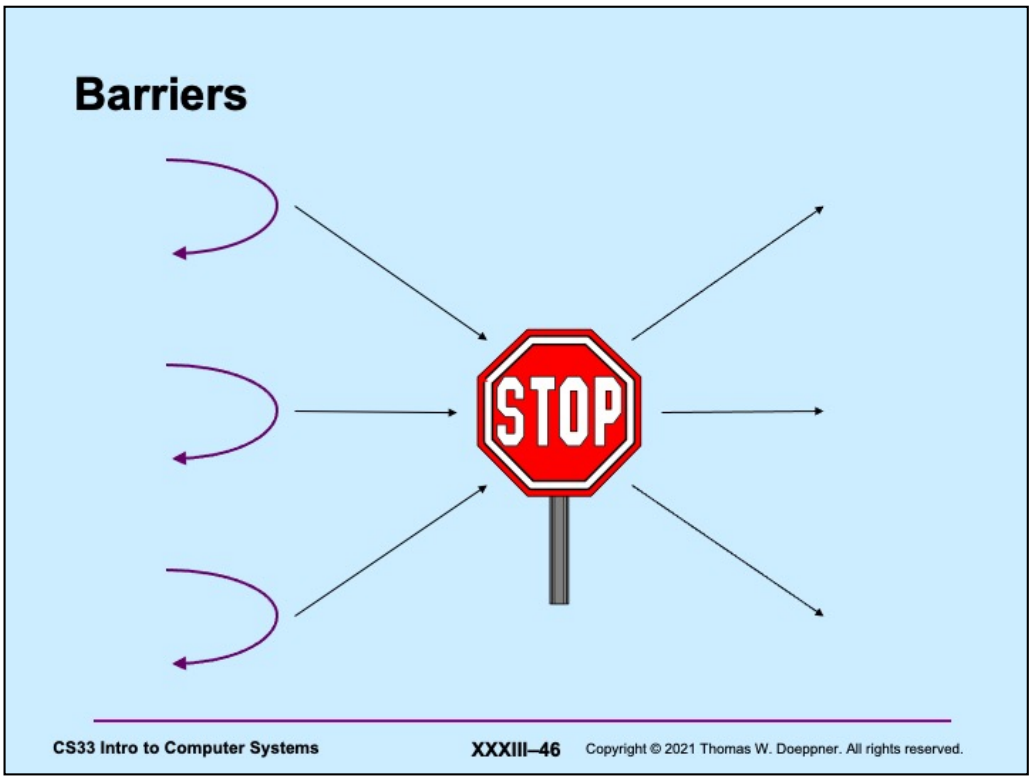

C Code: Add
with Fine-Grained Synchronization I

```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&head->lock);
  if ((target = search(key, &head, &parent,
      1_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;
}
```

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 ...

```
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);
```

Quiz 2
Does it work?

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

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.