CS 33

Multithreaded Programming III

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 sunspots. 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.

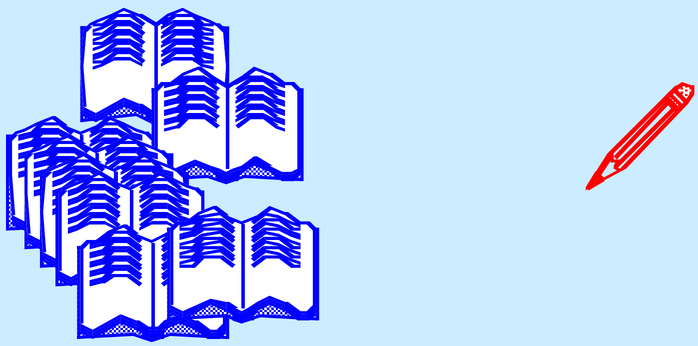
As discussed previously, this call to pthread_cond_wait has a fairly complicated effect: it releases the mutex given as the second argument and puts its caller to sleep, after queuing it on the condition variable given as the first argument. At some point in the future, a consumer should call **pthread_cond_signal**,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

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 1

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.

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.

Quiz 2

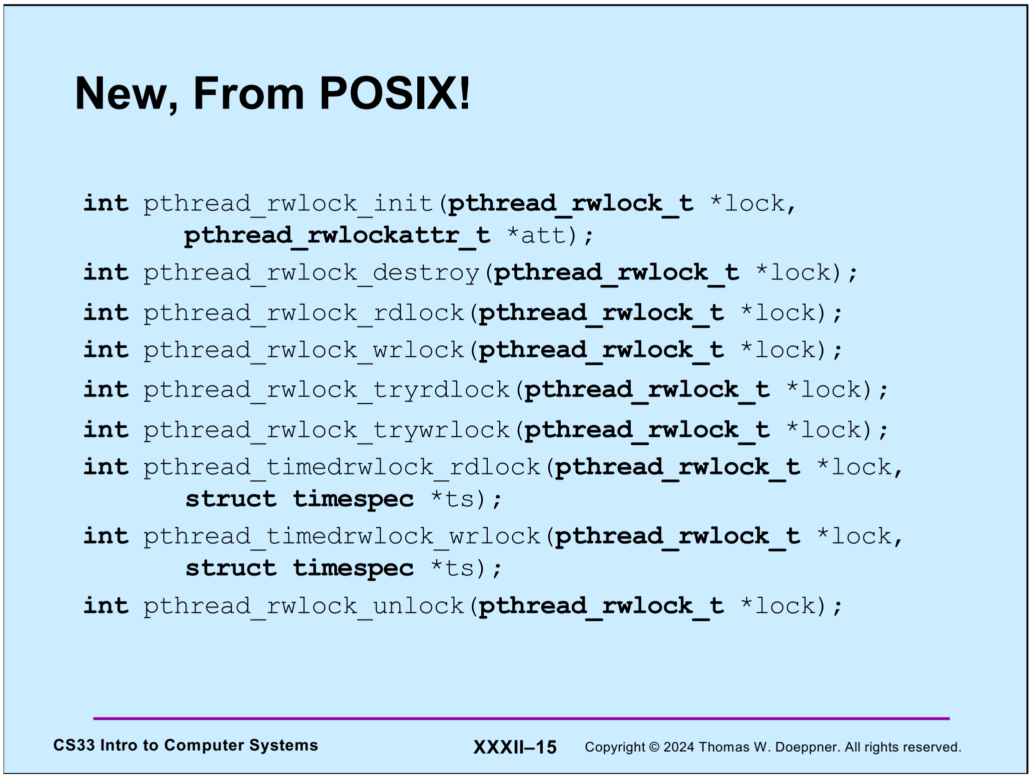
If a thread calls *reader*, will it eventually return from *reader* (assuming well behaved threads)?

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

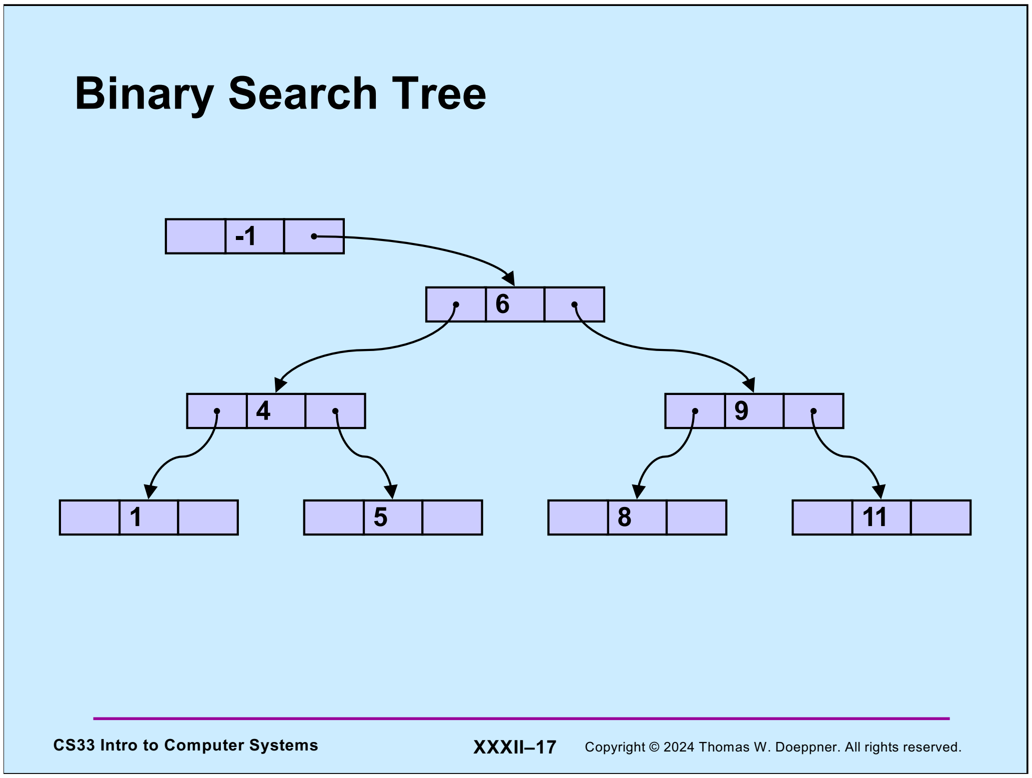
d) no, never



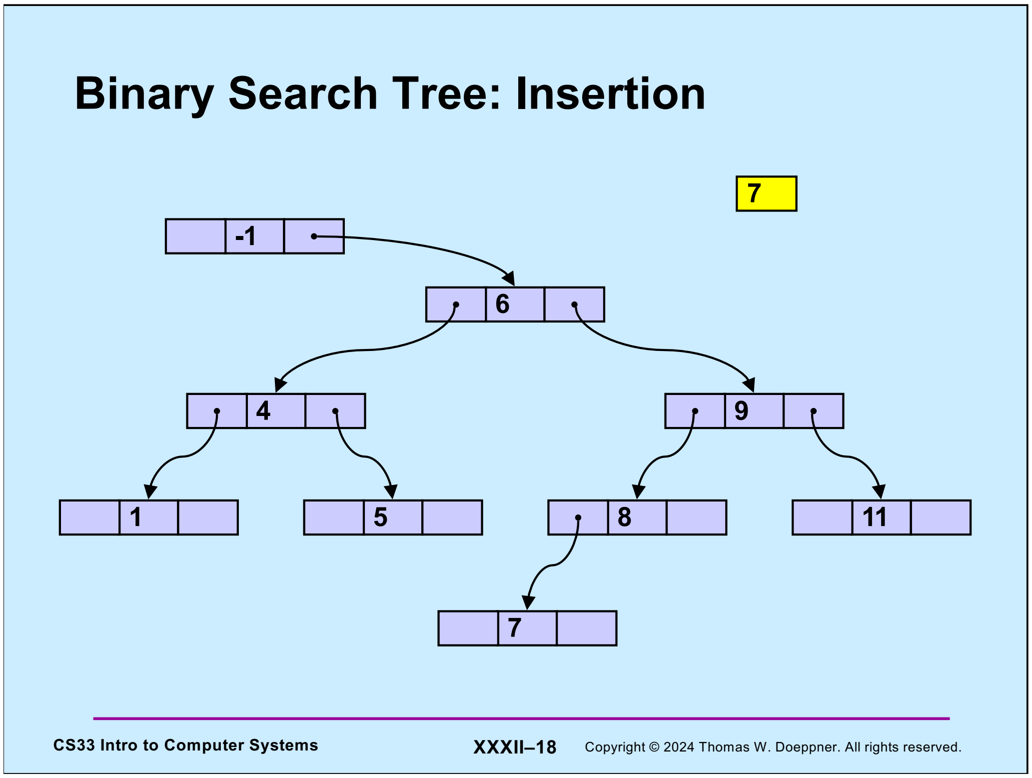
With POSIX 1003.1j support for readers-writers locks was finally introduced. The almost complete API is shown in the slide (what's missing are the operations on attributes). As might be expected, readers-writers locks can be statically initialized with the constant PTHREAD_RWLOCK_INITIALIZER. The "timedrwlock"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

Quiz 3

- Missing in the *rwlock* API is a function to "upgrade" a readers lock into a writers lock. It's not included because
  a) it's rarely needed, so there's no point to including it
  b) the same effect could be achieved by unlocking the readers lock, then taking a writers lock
  c) using such a function would likely result in deadlock

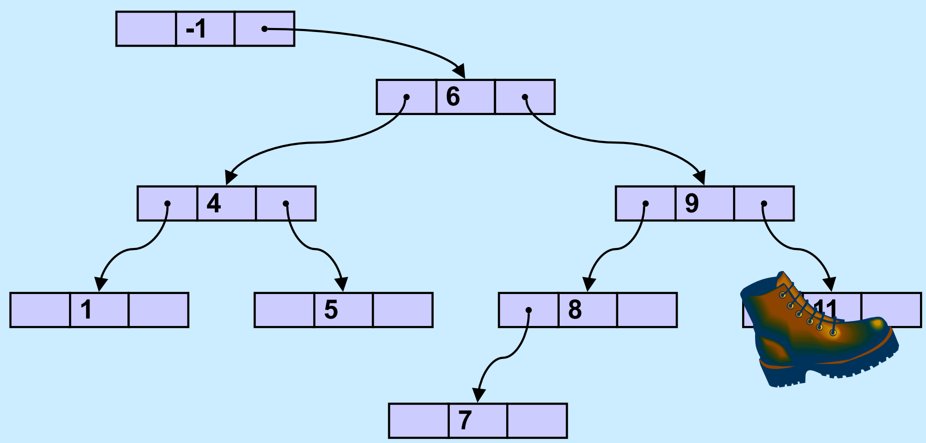


In this sequence of slides, we look at how we might take a simple (unbalanced) binary search tree and add readers-writers locks to it so that multiple threads can manipulate it concurrently. Each node of the tree consists of a pointer to a left child, a pointer to a right child, and a key (an integer 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.



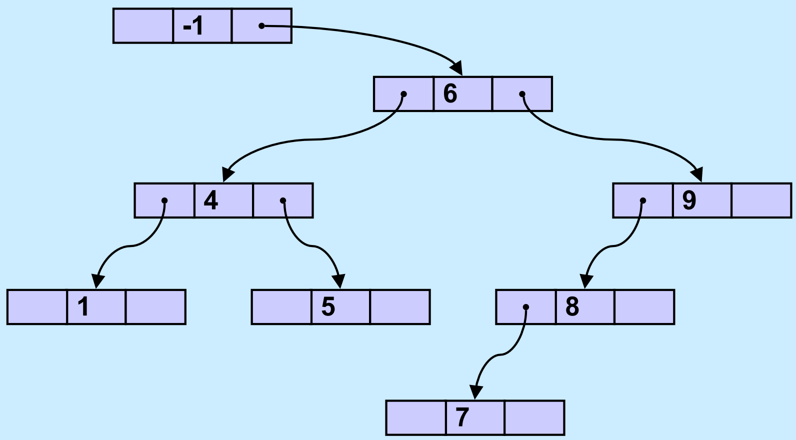
To add a new node to the tree, say one whose key will be 7, we start at the head and trace our way down the tree, comparing the new key with the keys of tree nodes, following left or right child pointers as appropriate. A new node is always inserted as a leaf.

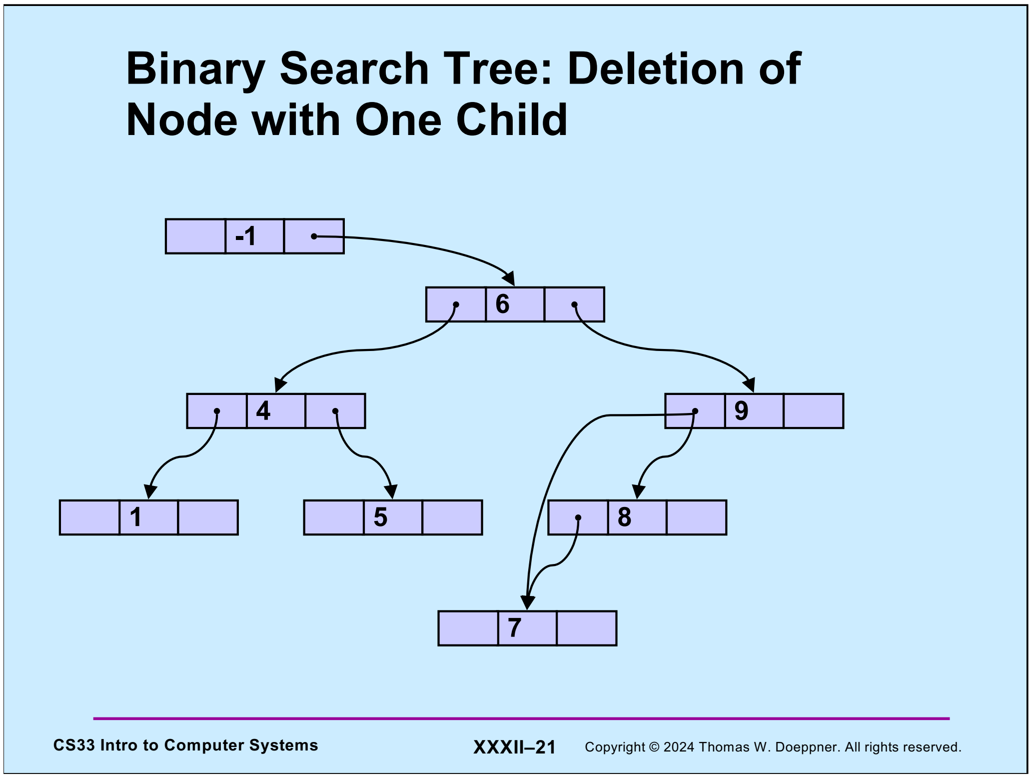
Binary Search Tree: Deletion of Leaf



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

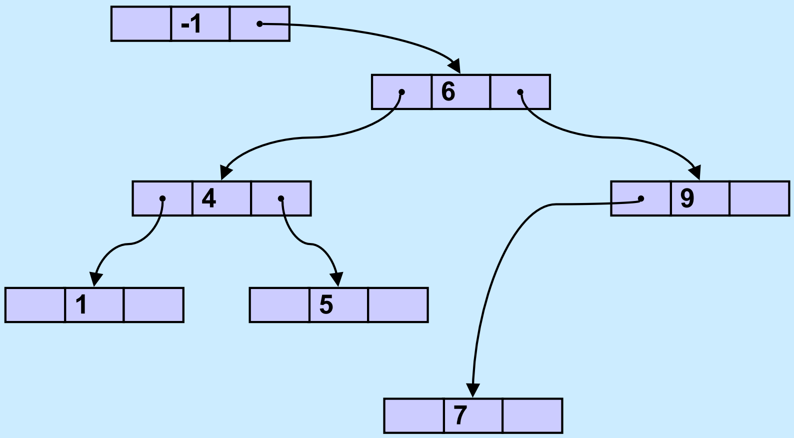
Binary Search Tree: Deletion of Leaf

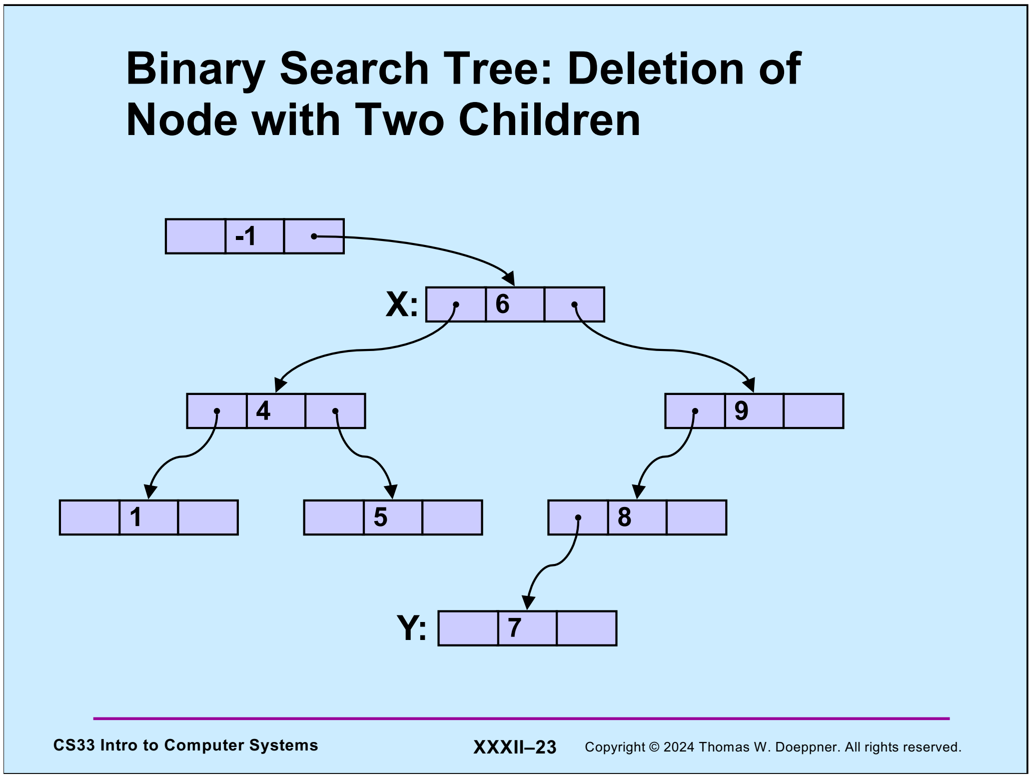




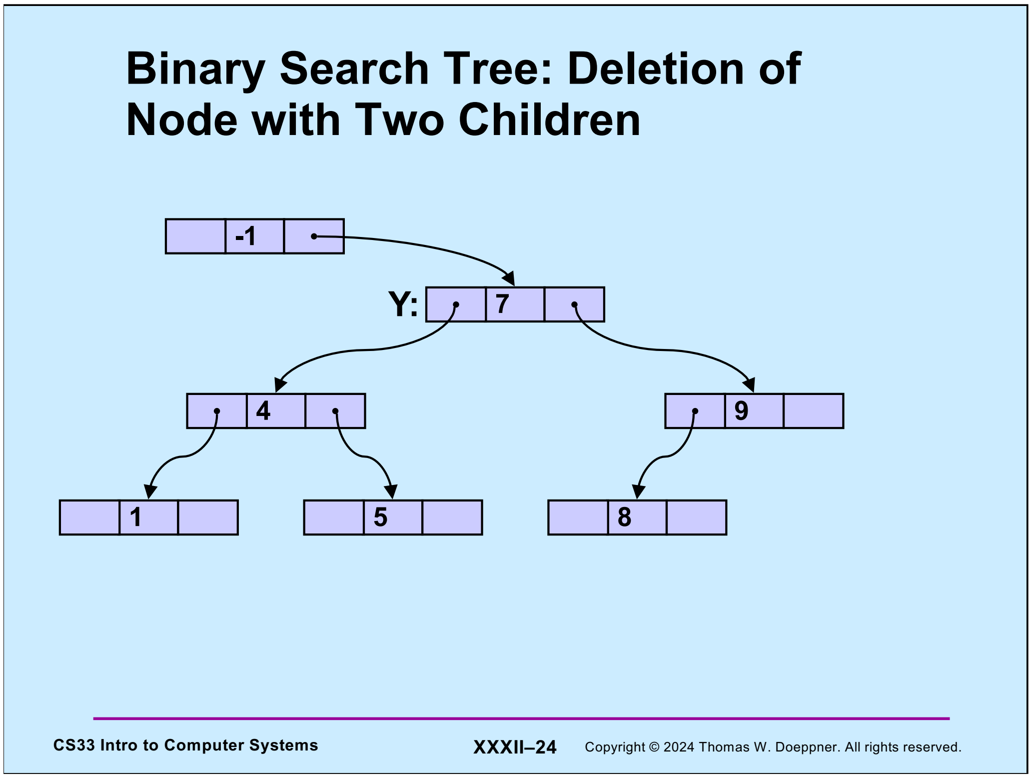
Deleting an interior node that has just one child is almost as easy. The child pointer from its parent is changed to point to the node's child, and then the node is deleted.

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

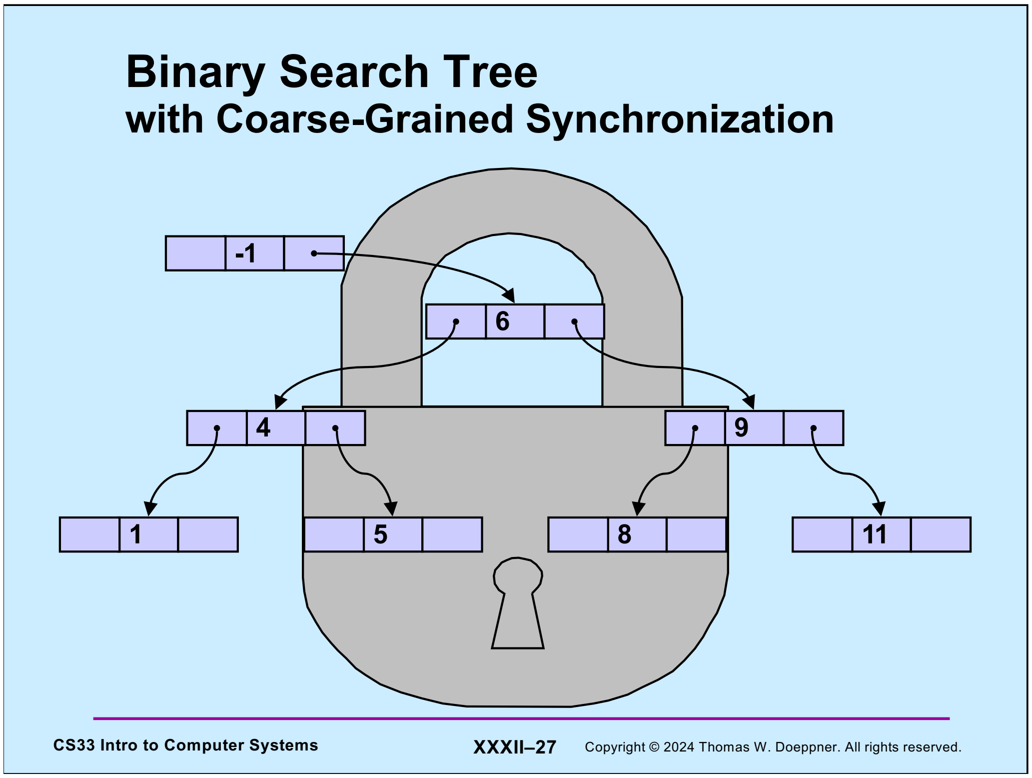
```
Node *search(int key,
    Node *parent, Node **parentp) {
  Node *next;
  Node *result;
  if (key < parent->key) {
    if ((next = parent->lchild)
        == 0) {
      result = 0;
    } else {
      if (key == next->key) {
        result = next;
      } else {
        result = search(key,
     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

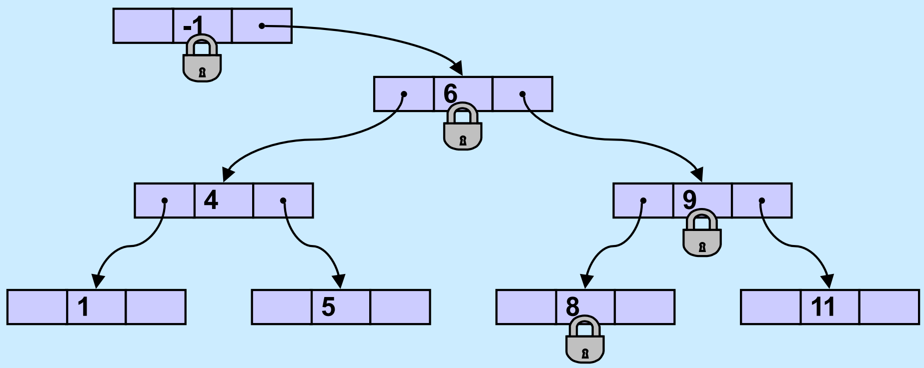


An easy way to allow multiple threads to manipulate the search tree concurrently is to employ what's known as **coarse-grained synchronization**: we associate a readers-writers lock with the entire tree. A thread that is just searching the tree for a value should take a read lock. A thread attempting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

C Code: Add
with Coarse-Grained Synchronization

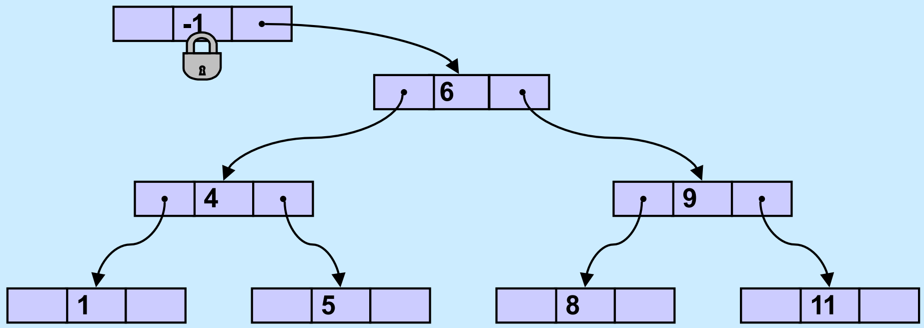
```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&tree_lock);
  if ((target = search(key, &head, &parent)) != 0) {
    pthread_rwlock_unlock(&tree_lock);
    return 0;
  }
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;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

Binary Search Tree
with Fine-Grained Synchronization I

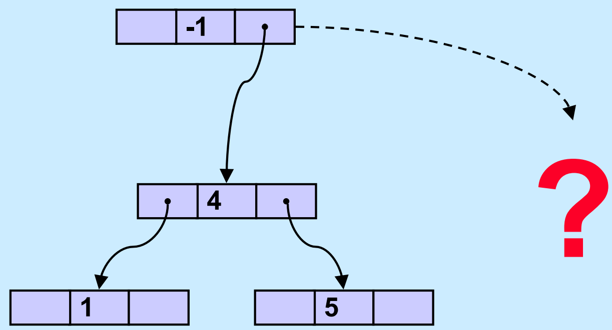


Let’s now look at what’s known as **fine-grained synchronization**, where we associate a readers-writers lock with each node of the tree. The idea is that, unlike the case for coarse-grained synchronization, we can have multiple threads working on different parts of the tree at once. The first step in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

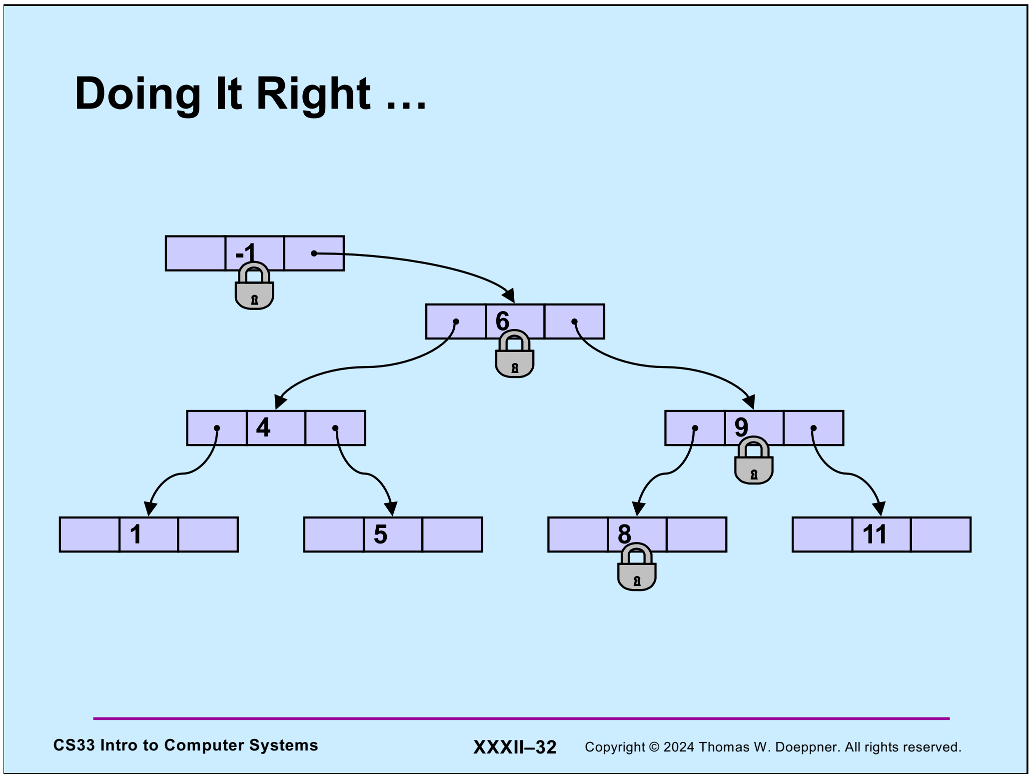
Binary Search Tree
with Fine-Grained Synchronization II



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).



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;
}
```

Quiz 4

The search function takes read locks if the purpose of the search is for a query, but takes write locks if the purpose is for an add or a delete. Would it make sense for it always to take read locks until it reaches the target of the search, then take a write lock just for that target?

a) Yes, since doing so allows more concurrency
b) No, it would work, but there would be no increase in concurrency
c) No, it would not work

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;
}
```


Quiz 5

The add function calls malloc. Could we use the malloc that you'll finish by next Monday for this, or do we need a different one that's safe for use in multithreaded programs?

a) **Since the calling thread has a write lock on the parent of the new node, it's safe to call the standard malloc**

b) **Even if the calling thread didn't have a write lock on the parent, it would be safe to call the standard malloc**

c) **We will need a new malloc, one that's safe for use in multithreaded programs**