CS 33

Multithreaded Programming IV

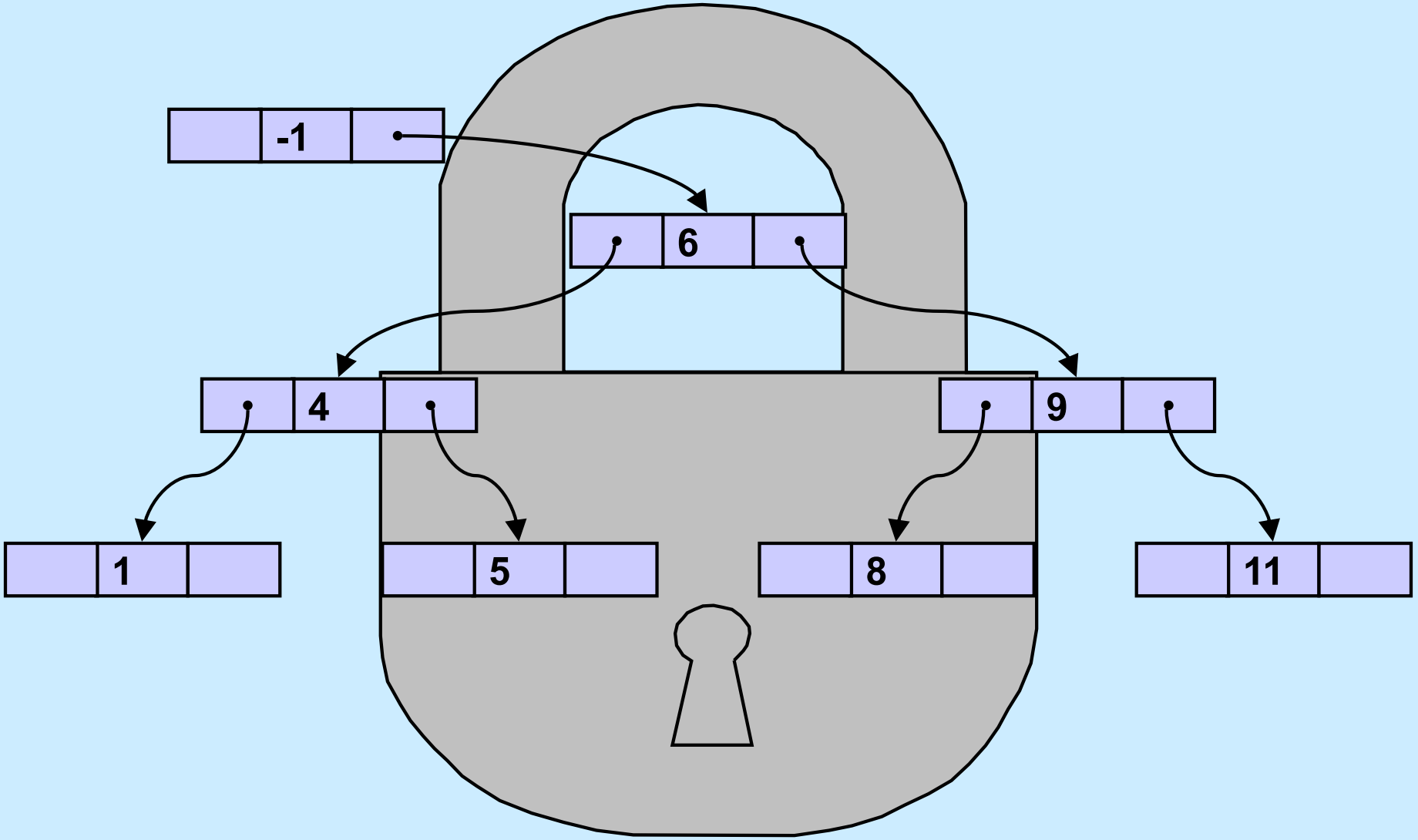
C Code: Search

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

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;
}
```

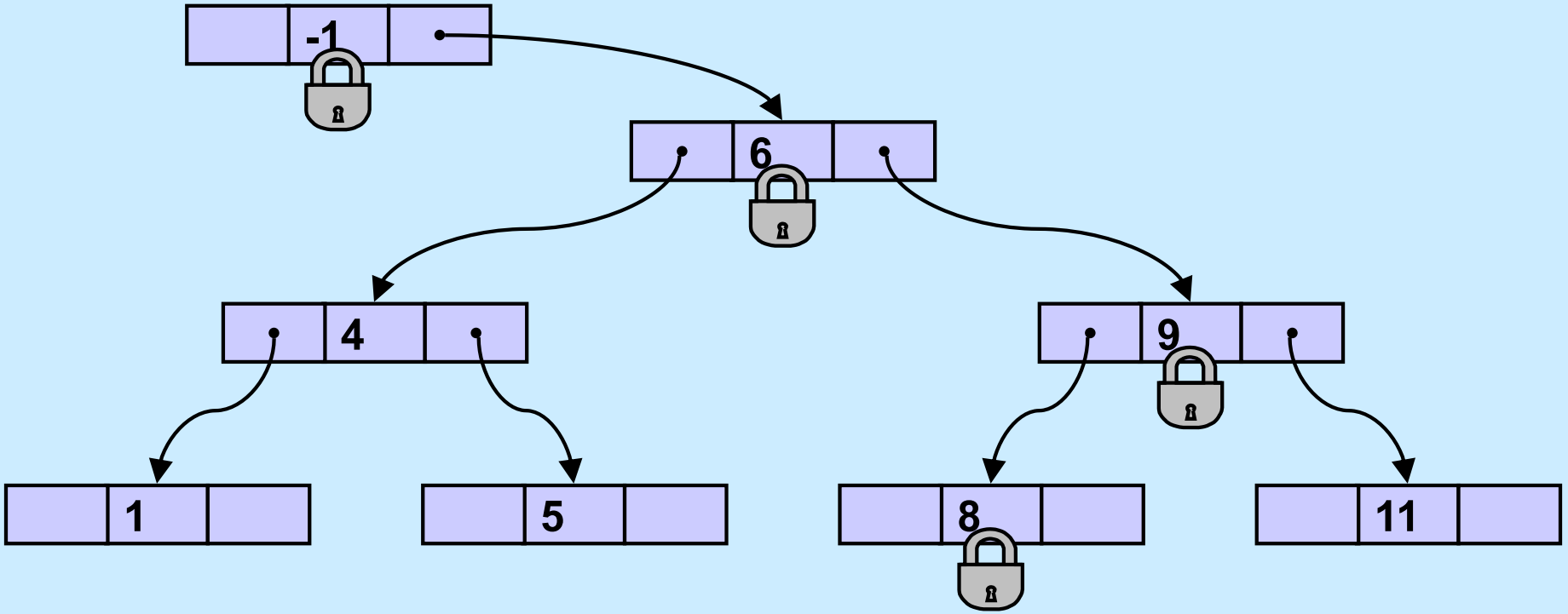

Binary Search Tree
with Coarse-Grained Synchronization



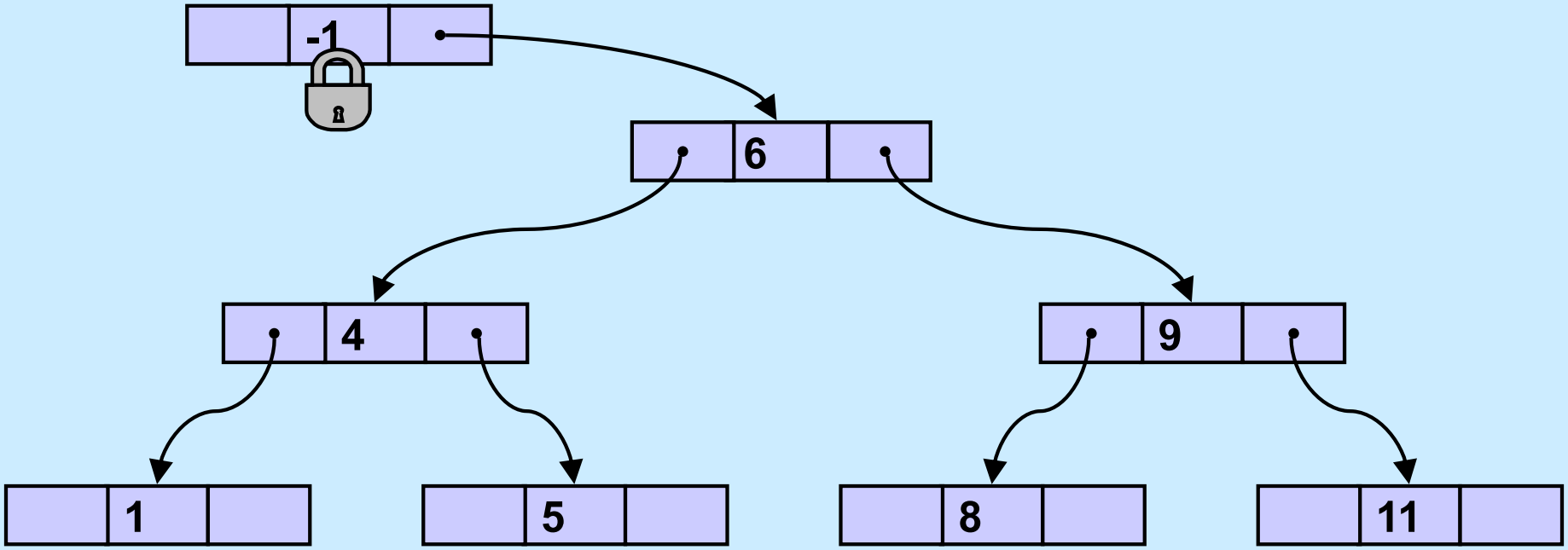
C Code: Add
with Coarse-Grained Synchronization

```
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;
}
```

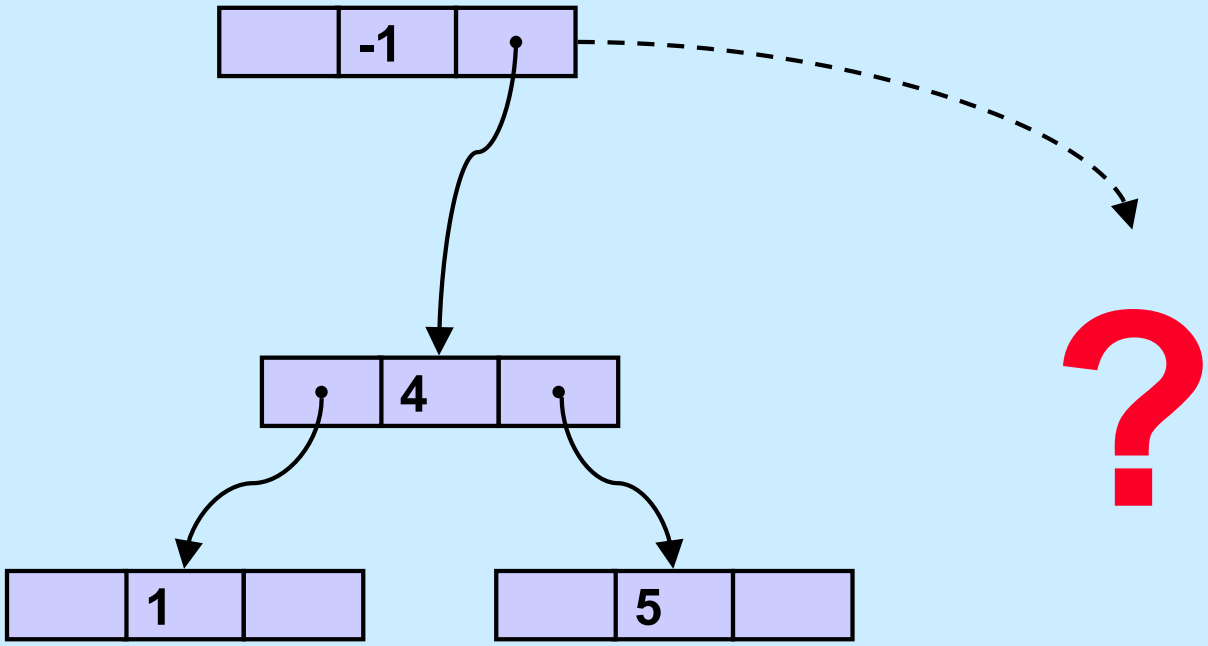
Binary Search Tree
with Fine-Grained Synchronization I



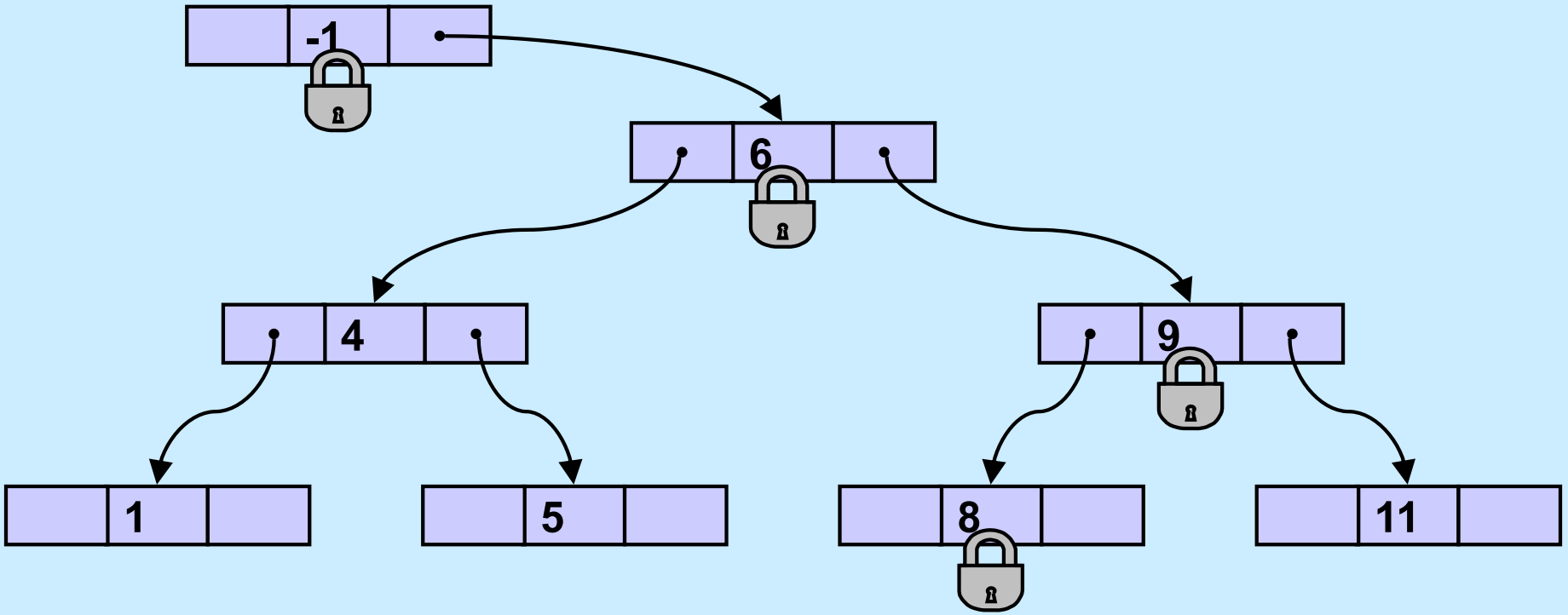
Binary Search Tree
with Fine-Grained Synchronization II



Binary Search Tree
with Fine-Grained Synchronization III



Doing It Right …



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;
      }
    }
```

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 1

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;
  }
```

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 2

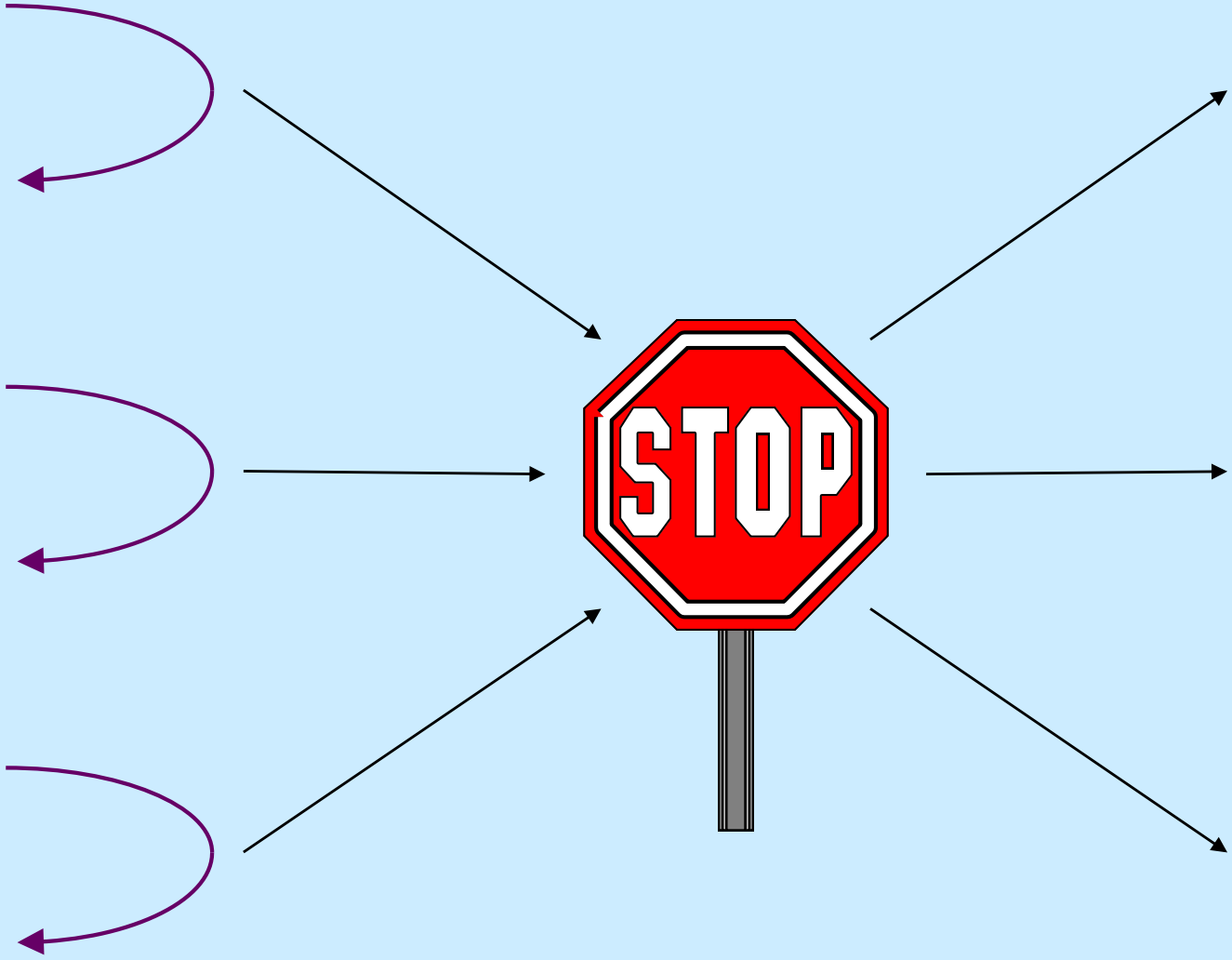
The *add* function calls *malloc*. Could we use for this the *malloc* that you'll finish by midnight, or do we need a different one that's safe for use in multithreaded programs?

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

Barriers



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);
```

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);
```

And This ...

Quiz 3
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);
```

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);
```

Why *cond_wait* is Weird …

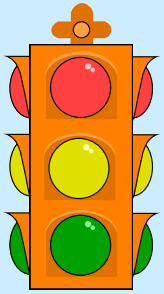
```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m) {
    pthread_mutex_unlock(m);
    sem_wait(c->sem);
    pthread_mutex_lock(m);
}

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

Deviations

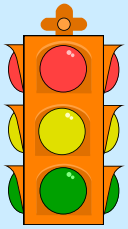
- **Signals**

vs.



- **Cancellation**
  - **tamed lightning**

Signals

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

Dealing with Signals

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

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*

Asynchronous Signals (1)

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

   ...

}

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


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);
}
```


Quiz 4

```
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) always
b) sometimes
c) never

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);
}
```

Cancellation

Sample Code

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

```
pthread_cancel(thread);
```

Cancellation Concerns

- Getting cancelled at an inopportune moment
- Cleaning 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)`

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

Cleaning Up

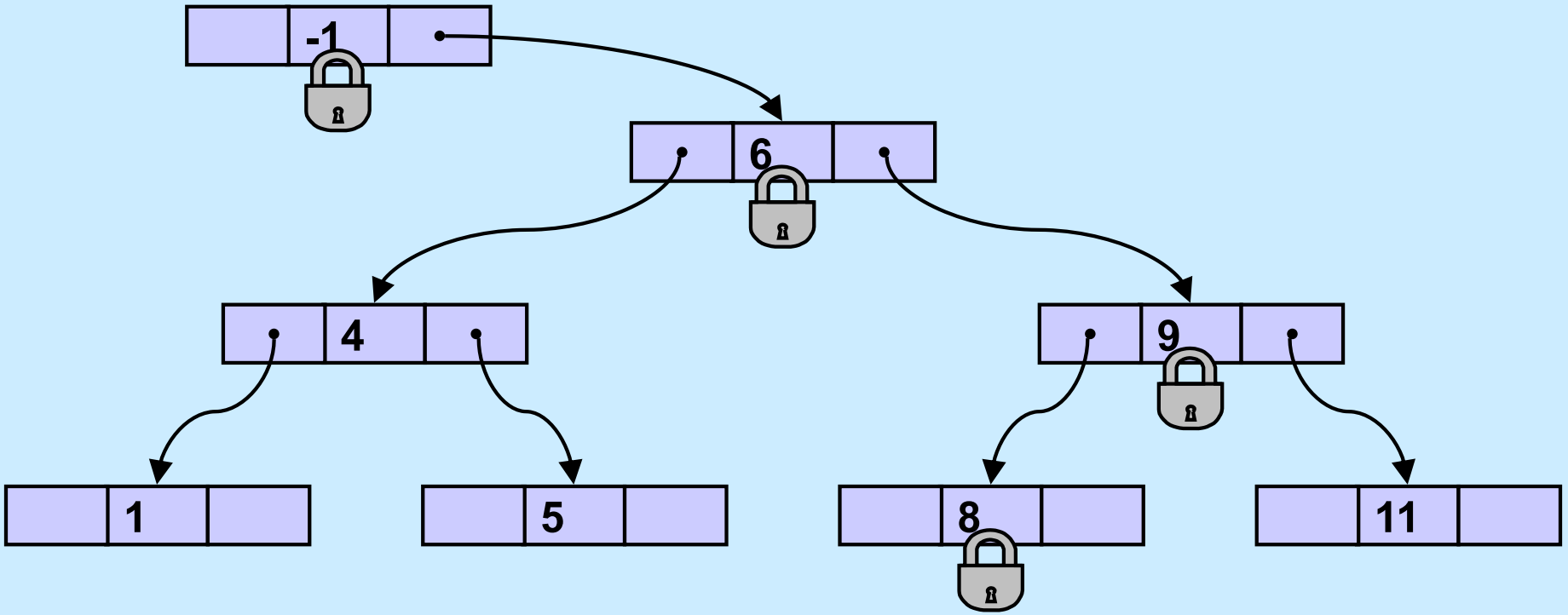
- `void pthread_cleanup_push((void)(*routine)(void *), void *arg)`
- `void pthread_cleanup_pop(int execute)`

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));
    nodep->next = head;
    head = nodep;
    if (read(0, &nodep->value,
        sizeof(nodep->value)) == 0) {
      free(nodep);
      break;
    }
  }
  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;
  }
}
```

A More Complicated Situation …



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);
}
```

Not a Quiz

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

Cancellation and Conditions

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

read(0, buffer, len);   // read is a cancellation point

pthread_cleanup_pop(1);
```

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);
}
```

