CS 33

Multithreaded Programming IV
Binary Search Tree
Binary Search Tree: Insertion

-1 6

4 8 9

1 5 7 11

7
Binary Search Tree: Deletion of Leaf
Binary Search Tree: Deletion of Leaf
Binary Search Tree: Deletion of Node with One Child
Binary Search Tree: Deletion of Node with One Child
Binary Search Tree: Deletion of Node with Two Children

X: 6

Y: 7

-1

4

1

5

8

9
Binary Search Tree: Deletion of Node with Two Children
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;
}
C Code: Add

```c
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;
}
```
Readers-Writers API

```c
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);
```
Binary Search Tree with Coarse-Grained Synchronization
C Code: Add with Coarse-Grained Synchronization

```c
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 …
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, parentp, lt);
                return result;
            }
        }
    }
C Code: Fine-Grained Search II

```c
} 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

```c
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;
Barriers
A Solution?

```c
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?

```c
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 ...

```c
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 1
Does it work?

a) definitely  
b) probably  
c) rarely  
d) never
Barrier in POSIX Threads

```c
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!

```c
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 \textit{cond\_wait} is Weird ...

\begin{verbatim}
void pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m) {
    pthread_mutex_unlock(m);
    sem_wait(c->sem);
    pthread_mutex_lock(m);
}

void pthread_cond_signal(pthread_cond_t *c) {
    sem_post(c->sem);
}
\end{verbatim}
Deviations

• Signals

• Cancellation
  – tamed lightning
Signals

•

  – who gets them?
  – who needs them?

  ![Traffic Light]

•

  – how do you respond to them?
Dealing with Signals

- Per-thread signal masks
- Per-process signal vectors
- One delivery per signal
Signals and Threads

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

– thread equivalent of `kill`

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

– thread equivalent of `sigprocmask`
Asynchronous Signals (1)

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

    ...

}

void handler(int sig) {
    ...

}
```
Asynchronous Signals (2)

```c
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);
}
```
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
Synchronizing Asynchrony

```c
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, long_running_procedure);
}

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

```c
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;
}
```
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_CANCELASYNCHRONOUS,
     PTHREAD_CANCELDDEFERRED},
    &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

```c
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;
    }
    pthreadCleanup_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

```c
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

```c
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

```c
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

```c
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);
```