15-213 Recitation
Synchronization

Your TAs

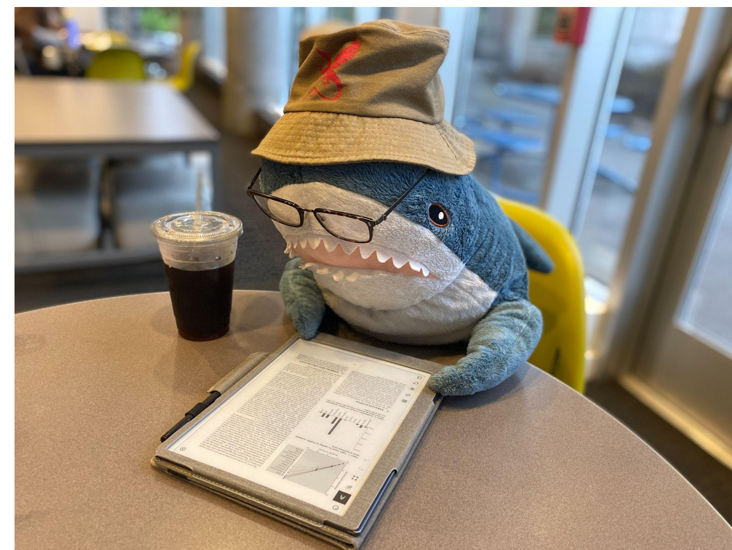
Friday, November 22th

Reminders

- **`proxylab`** is due ***Tuesday (November 26th)***
  - 2 grace days and 1 late day (runs into break)
- **`sfslab`** will be released before Thanksgiving
  - Due *December 5th*
  - Last Day to Handin: *December 6th*
- *Code Reviews for* **`tshlab`**

Apply to be a TA!

- TA Applications are open! See Piazza @1104 :-)
  - First round of interviews happening in 1-2 weeks!
- What qualifications are we looking for?
  - Decent class performance
  - Strong communication skills
  - Reasonable ability to gauge schedule and responsibilities



Agenda

- **Review:**
  - **Threading**
  - **Synchronization Errors**
  - **Locking**
- **Activity: Making Trees Thread-Safe**

Proxies and Threads

- Network connections can be handled concurrently
  - Three approaches were discussed in lecture for doing so
    - Process-based, Event-based, Thread-based
  - Your proxy should (eventually) use threads
  - Threaded echo server is a good example of how to do this

Review: Threads

- Each thread has its own logical control flow
- Each thread shares same code, data, and kernel context
- Each thread also has its own stack for local variables
  - **NOT** protected from other threads - all memory is shared
- **POSIX Threads**
  - **`pthread_create`**: starts a new thread
  - **`pthread_join`**: waits for specified thread to terminate
  - **`pthread_detach`**: marks specified thread as detached, where detached threads are cleaned-up without needing to be joined by a peer thread.

Example: Unsafe threading

```
#define NUM_THREADS 2; #define BUFFER_SIZE 50

void* print_message(void* arg) {...};

int main() {
    pthread_t threads[NUM_THREADS];
    char message[BUFFER_SIZE];
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, print_message, (void*)message);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

We launch 2 threads that each call **`print_message`**, passing in a shared constant length array

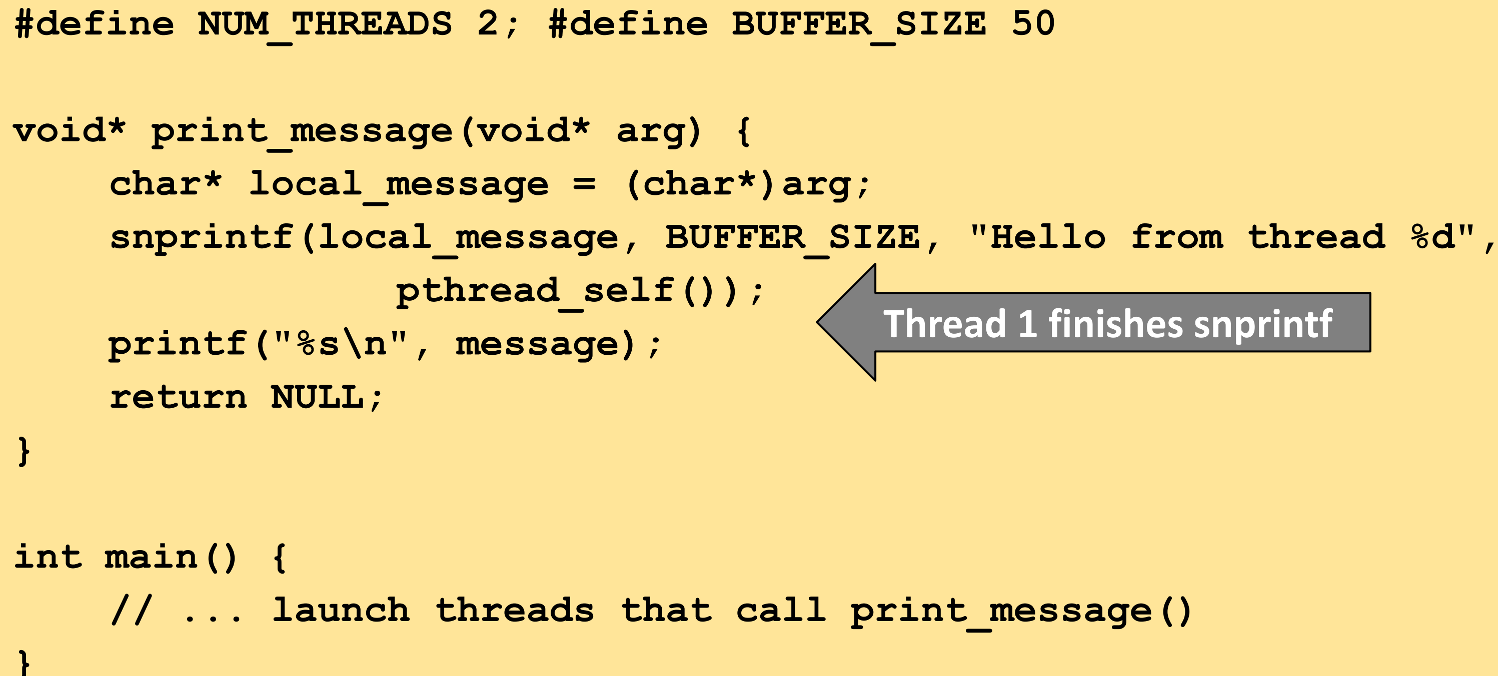
Example: Unsafe threading

```
#define NUM_THREADS 2; #define BUFFER_SIZE 50

void* print_message(void* arg) {
    char* local_message = (char*)arg;
    snprintf(local_message, BUFFER_SIZE, "Hello from thread %d",
                pthread_self());
    printf("%s\n", message);
    return NULL;
}

int main() {
    // ... launch threads that call print_message()
}
```

Stores string with thread id into `local_message` buffer

Now let's see how our threads interact with `print_message`, assuming thread 1 runs first

Example: Unsafe threading



Example: Unsafe threading

```
#define NUM_THREADS 2; #define BUFFER_SIZE 50

void* print_message(void* arg) {
    char* local_message = (char*)arg;
    snprintf(local_message, BUFFER_SIZE, "Hello from thread %d",
                 pthread_self());
    printf("%s\n", message);
    return NULL;
}

int main() {
    // ... launch threads that call print_message()
}
```

Example: Unsafe threading

```
#define NUM_THREADS 2; #define BUFFER_SIZE 50

void* print_message(void* arg) {
    char* local_message = (char*)arg;
    snprintf(local_message, BUFFER_SIZE, "Hello from thread %d",
                  pthread_self());
    printf("%s\n", message);
    return NULL;
}

int main() {
    // ... launch threads that call print_message()
}
```

Note: each local message points to the same buffer!

Example: Unsafe threading

```
#define NUM_THREADS 2; #define BUFFER_SIZE 50

void* print_message(void* arg) {
    char* local_message = (char*)arg;
    snprintf(local_message, BUFFER_SIZE, "Hello from thread %d",
                pthread_self());
    printf("%s\n", message);
    return NULL;
}

int main() {
    // ... launch threads that call print_message()
}
```

Example: Unsafe threading

```
#define NUM_THREADS 2; #define BUFFER_SIZE 50

void* print_message(void* arg) {
    char* local_message = (char*)arg;
    snprintf(local_message, BUFFER_SIZE, "Hello from thread %d",
                pthread_self());
    printf("%s\n", message);
    return NULL;
}

int main() {
    // ... launch threads that call print_message()
}
```

Various other unsafe scenarios can occur! This is only one example.

Classical Problems in Concurrency

- **Deadlock**
  - Two or more threads are unable to proceed because each is waiting for a resource that the other holds.
- **Livelock**
  - Two or more threads continuously change their state in response to each other - but with no further progress.
- **Starvation**
  - One of more threads continuously denied access to resources because other threads holds them.

Example 1: Identify problem type

- You are at a small table in a restaurant waiting to be served dinner. But, tables of 213 TAs with large group orders keep showing up, so the waiters continuously tend to these large tables and you are never served.

- **Solution**: Starvation, the TAs continually get served but you don't!

Example 2: Identify problem type

- You and your friend are at a famous restaurant but due to how busy it is, the waiter only gives the table one knife and fork. For someone to eat, they need both the knife and fork. You grab the fork and your friend grabs the knife, both of you refusing to give the other the needed utensil.

- **Solution**: Deadlock, bad resource management leads to a state of no forward progress

Example 3: Identify problem type

- When it is time to pay for the meal, both you and your friend want the check. You grab the check to pay for it, but your friend grabs the check back to place his card. The process repeats.

- **Solution**: Livelock, both you and your friend are trying to pay for the check but neither can since once you grab the check, your friend steals it, you steal it back, …

Synchronization

Locking

- We saw that all memory is shared across threads - how can we prevent unsafe behavior?
  - Use Locks! *(But correctly…)*
- There are various locks, including mutexes, semaphores, atomic operations, etc… (more to come in Tuesday's lecture!)
- Today, we'll focus on using mutexes.

Review: Mutexes

- Opaque object which is either locked or unlocked.
- **`lock(m)`**
  - If m is not locked, lock it and return
  - If locked, wait until m is unlocked, then retry
- **`unlock(m)`**
  - Should only be called when **`m`** is locked, by the locker
  - Changes **`m`**'s state to unlocked
- Now we're prepared for our activity!

Activity: Thread-Safe BSTs

The Problem

- We want to create an implementation of BSTs that supports concurrent execution across multiple threads.
- We provide code that works correctly for sequential accesses!
- Assume no lookups/inserts to the same value happen in parallel
- Note that this BST does not support removal

Starter Code: Thread Safe BSTs

- Standard tree node struct that stores the value as well as it's left and right children.
- Standard recursive lookup function. Note that a thread-safe implementation should result in consistent lookup results.

```
struct node {
    int val;
    node_t *left;
    node_t *right;
};
```

```
int lookup(node_t *t, int val){
    if(t == NULL)
        return 0;
    if(t->val == val)
        return 1;
    else if(val < t->val)
        return lookup(t->left, val);
    else if(val > t->val)
        return lookup(t->right, val);
}
```

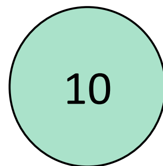
Code: Thread Safe BSTs

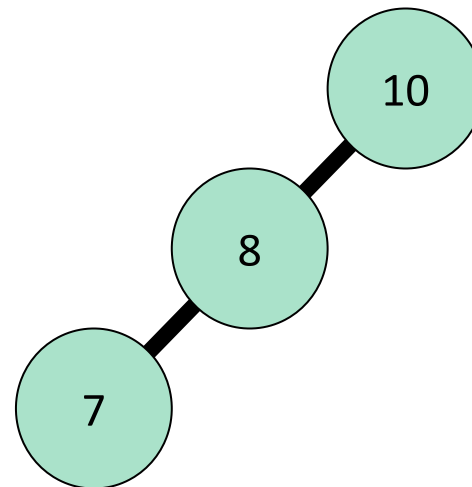
```
int insert(node_t *t, int val){
    if(t->val == val)
        return -1;
    else if(val < t->val){
        if(t->left != NULL)
            return insert(t->left, val);
        t->left = calloc(1, sizeof(node_t));
        t->left->val = val;
    }
    else if(val > t->val){
        if(t->right != NULL)
            return insert(t->right, val);
        t->right = calloc(1, sizeof(node_t));
        t->right->val = val;
    }
    return 1;
}
```

- Our main focus will be on the insert function!
- What could go wrong here?

Identifying Race Condition

- Suppose we want to do **`insert(8)`** and **`insert(7)`** on the tree below, where each call is launched in separate threads
  - Say thread 1 runs **`insert(8)`** and thread 2 runs **`insert(7)`**



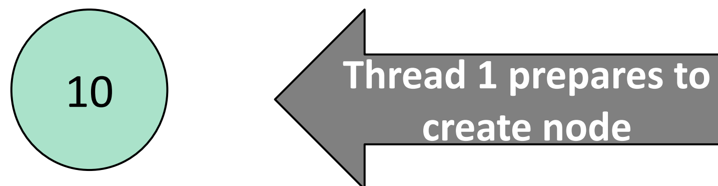


Original Tree

One Possible (correct) Tree

Identifying Race Condition

- Thread 1 sees that **`t->left == NULL`** and prepares to create the node (eg. call **`calloc`**)

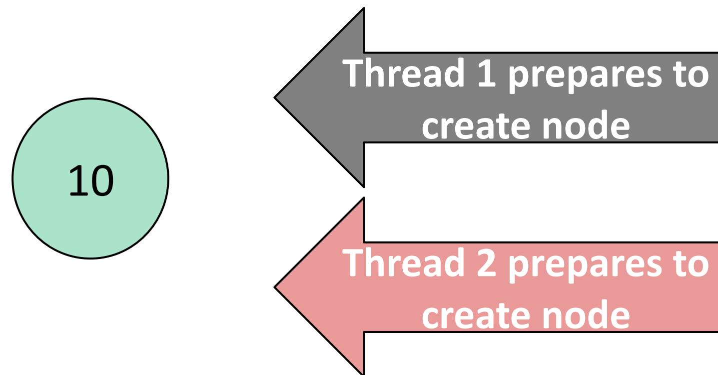


Relevant Case:

```
else if(val < t->val){
    if(t->left != NULL)
        return insert(t->left, val);
    t->left = calloc(1, sizeof(node_t));
    t->left->val = val;
}
```

Identifying Race Condition

- We then jump to thread 2, which also sees that **`t->left == NULL`** and prepares to create the node

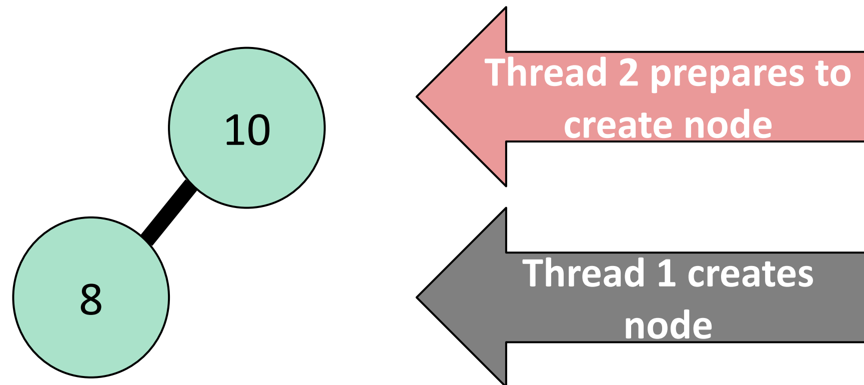


Relevant Case:

```
else if(val < t->val){
    if(t->left != NULL)
        return insert(t->left, val);
    t->left = calloc(1, sizeof(node_t));
    t->left->val = val;
}
```

Identifying Race Condition

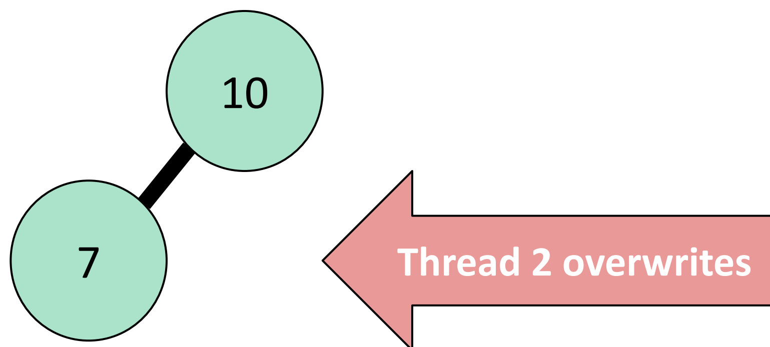
- Now thread 1 continues to run, creating the left node with `val = 8`



- However from thread 2's perspective, `t->left` is `NULL`!
  - The check has already occurred.

Identifying Race Condition

- Now thread 2 also attempts to create a left node, overwriting the node written by thread 1



- Unsafe behavior!

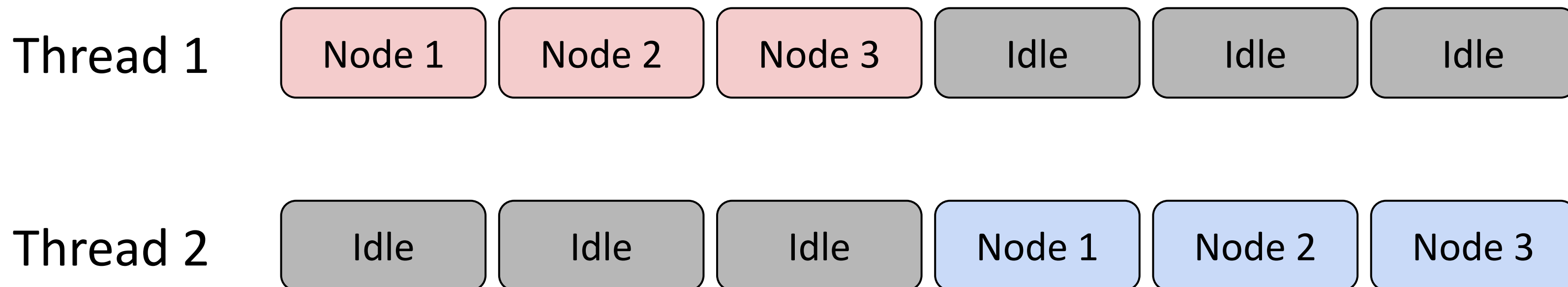
Solution 1: Coarse Grain Locking

- It is unsafe to have multiple threads accessing the tree at once
  - Let's lock away the entire tree!

```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int safe_lookup(node_t *t, int val){
    lock(&m);
    lookup(t,val);
    unlock(&m);
}

int safe_insert(node_t *t, int val){
    lock(&m);
    insert(t,val);
    unlock(&m);
}
```

Analysis: Coarse Grain Locking

- Currently, we lock the entire tree. How does this affect performance?
- Assuming each thread's call takes 3 iterations through the tree, we can see the following behavior!



- Wrapping each function call in locks makes all execution sequential. Can we make this better?

Solution 2: Partial Step towards fine grain

- We still want to lock away the tree for correctness, but can we add more granularity while still using a global lock?

```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int insert(node_t *t, int val){
    lock(&m);
    if(t->val == val)
        unlock(&m);
        return -1;
    else if(val < t->val){
        if(t->left != NULL) {
            unlock(&m);
            return insert(t->left, val);
        }
        t->left = calloc(1, sizeof(node_t));
        t->left->val = val;
    }
    else if(val > t->val){
        if(t->right != NULL){
            unlock(&m);
            return insert(t->right, val);
        }
        t->right = calloc(1, sizeof(node_t));
        t->right->val = val;
    }
    unlock(&m);
    return 1;
}
```

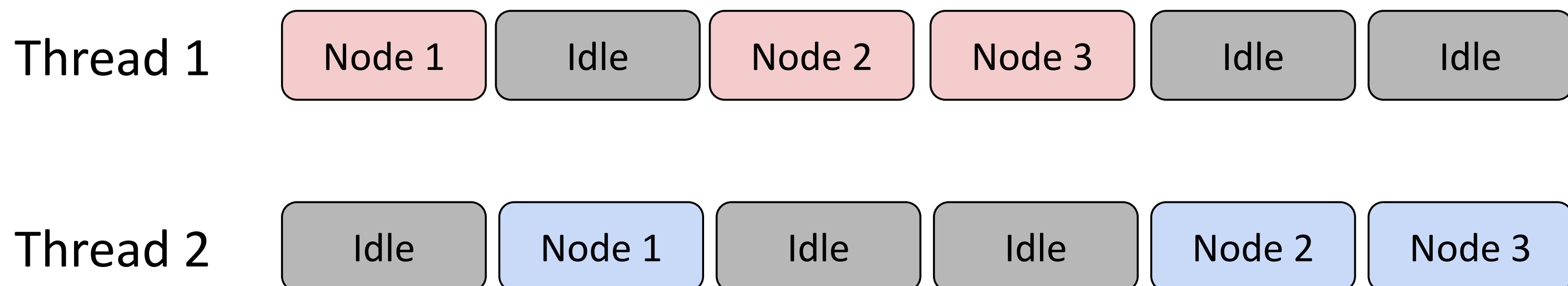
- We lock when entering and unlock before returning
  - Is this still thread-safe?

Solution 2: Partial Step towards fine grain

- Yes the function is thread-safe (under our assumptions)
  - BUT …
- Would this implementation be thread-safe if we supported removal from the tree?
- **No!** - What if we are traversing through the list and we unlock to make the recursive call, but we delete in between?
- **NOTE:** this solution only serves as an intermediary towards our final goal and is NOT a good general technique.
  - It will generally be incorrect (and also non-optimal)

Analysis: Solution 2

- Does this supposed increase in granularity actually improve performance?
- **No!** - This method only allows for the interleaving of execution, but is still inherently sequential.
  - (below is one possible example)



Analysis: Solution 2

- Another point of analysis is **lock overhead**, or the overhead from the frequency of threads attempting to acquire the lock.
- Compared to solution 1, do we see an increase or decrease in lock overhead on our global lock?
- **Increase** - previously we attempt to acquire at every call to lookup or insert, but now we make an attempt at every iteration.
- We see no improvement in performance - it actually got worse due to this overhead! Can we make this better?

Group Activity: Fine Grain Locking

- As groups, find an implementation that is thread-safe, but also doesn’t always lock the entire tree.
    - Stay within mutexes [you may modify the struct :)]

```
struct node {
    int val;
    node_t *left;
    node_t *right;
};

int lookup(node_t *t, int val){
    if(t == NULL)
        return 0;
    if(t->val == val)
        return 1;
    else if(val < t->val)
        return lookup(t->left, val);
    else if(val > t->val)
        return lookup(t->right, val);
}
```

```
int insert(node_t *t, int val){
    if(t->val == val)
        return -1;
    else if(val < t->val){
        if(t->left != NULL)
            return insert(t->left, val);
        t->left = calloc(1, sizeof(node_t));
        t->left->val = val;
    }
    else if(val > t->val){
        if(t->right != NULL)
            return insert(t->right, val);
        t->right = calloc(1, sizeof(node_t));
        t->right->val = val;
    }
    return 1;
}
```

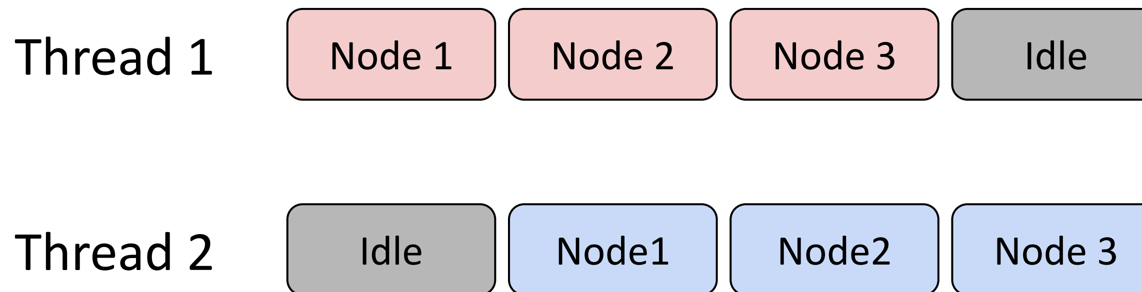
Solution 3: Fine Grain locking

- Instead of locking the entire tree, we can implement per-node locking. This ensures no two threads will try to simultaneously update the same node.
- We can adjust the node struct to include a lock (shown below)
- Similar to our second solution, we lock and unlock at each iteration, aka at each access to a node.

```
struct node {
    int val;
    node_t *left;
    node_t *right;
    pthread_mutex_t m;
};
```

Analysis: Solution 3

- How does fine-grain locking help? Let’s return to the figure from before!
  - Again, we assume each thread makes 3 iterations



- Nice! We managed to expose the potential concurrency in these iterations (*note: this requires a multiprocessor system*)

Analysis: Solution 3

- In our first coarse-grain solution, we saw a large critical section per lock (the entire function call). What about this solution?
- **Drastic Reduction!**
  - We now only block off one iteration at a time (as pointed to by the previous diagram)
- However, there is still high overhead…
  - (we try to acquire locks many times)
- Tradeoff analysis of parallelism and overhead is beyond the scope of 15-213 - look into 15-346 or 15-418!

Wrapping Up

- `proxylab` is due ***Tuesday (November 26th)***
  - 2 grace days and 1 late day (runs into break)
- `sfslab` will be released before Thanksgiving
  - Due *December 5th*
  - Last Day to Handin: *December 6th*
- *Code Reviews for* `tshlab`
- Apply to be a TA!
- Good luck on `proxylab` :-)

The End