15-213 Recitation
Synchronization

Your TAs

Friday, November 21st

Reminders

- `sfslab` to be released on **November 25th**
  - Due Date: ***Dec 04th***
- `proxylab` released
  - Due Date: ***November 25th***
- *Code Reviews for* `tshlab`

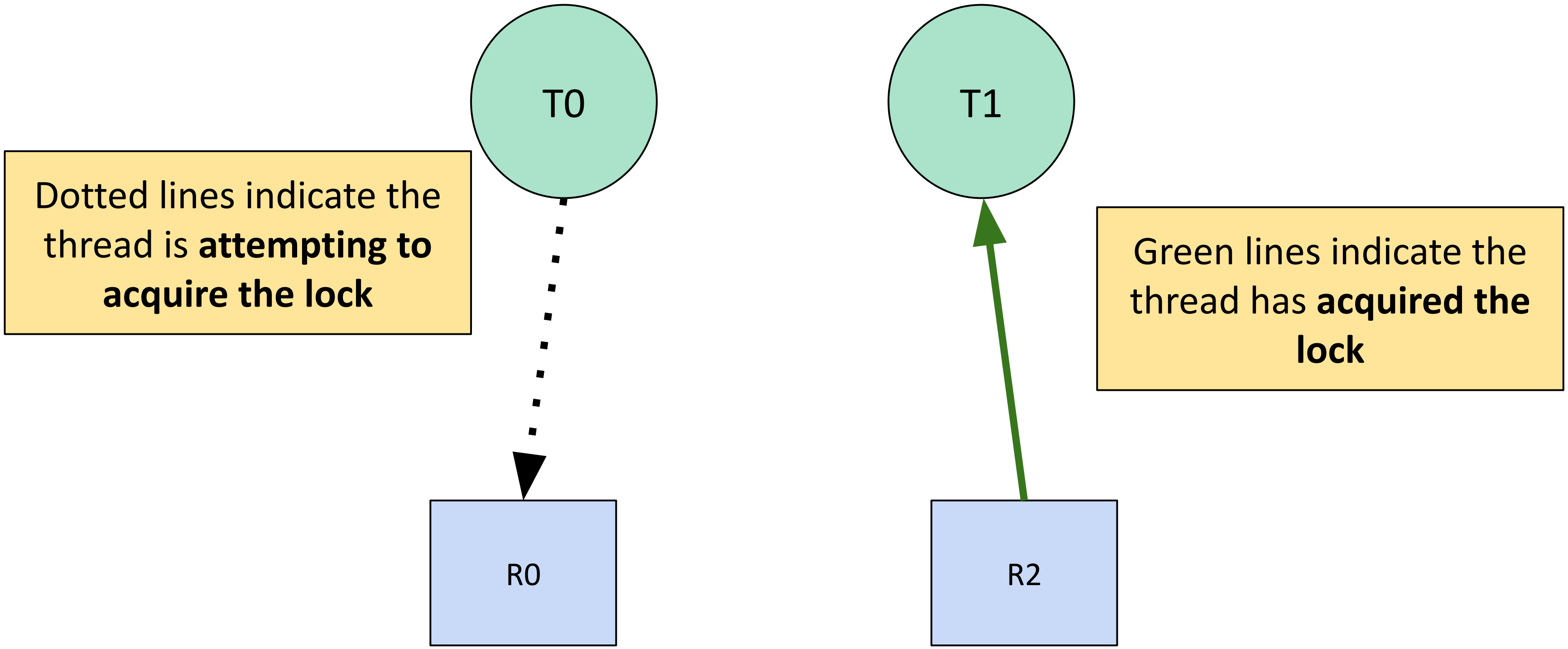
Agenda

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

Synchronizing With Locks - Deadlock

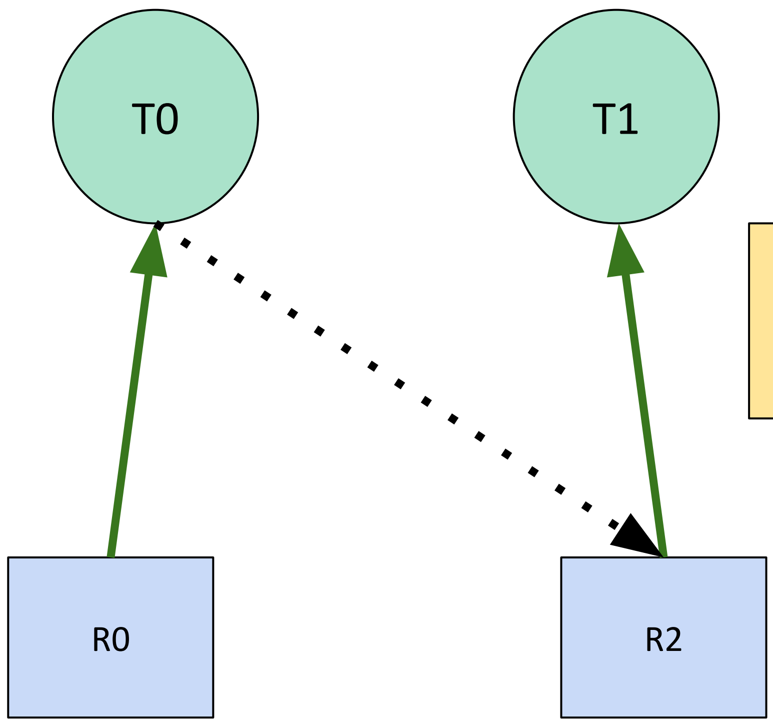
Scenario: Hold and wait

Thread *T0* needs to acquire both *R0* and *R2* to proceed.



Scenario: Hold and wait

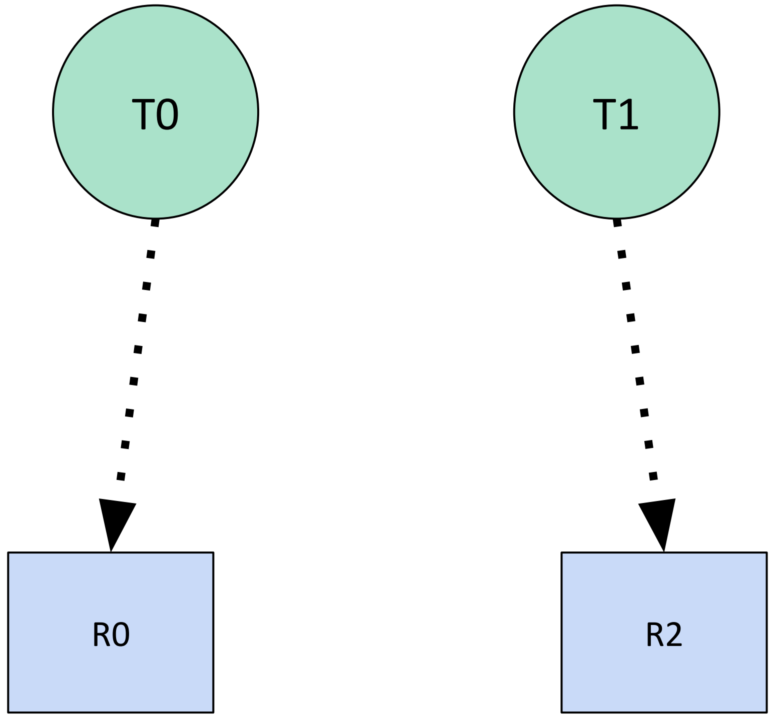
T0 waits on R2 to be released. When can T0 proceed?



Only once T1 **releases** lock R2… Will this always happen?

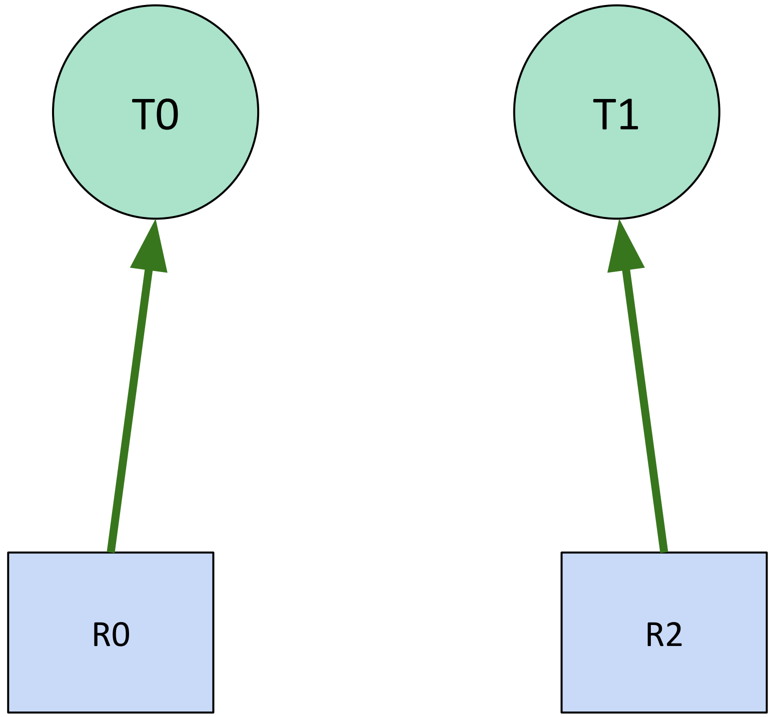
Scenario: Circular wait

T0 and *T1* try to acquire *R0* and *R1*.



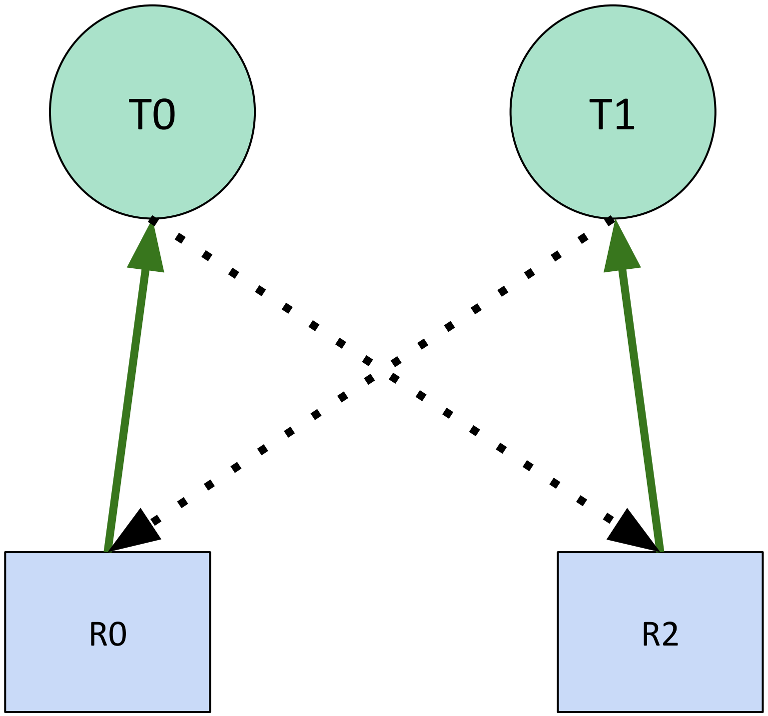
Scenario: Circular wait

T0 and T1 acquire the respective resources!



Scenario: Circular wait

But both need the other resource as well before proceeding. How do we end up here?



Circular wait

```
Thread 0

lock(&R1)
lock(&R2)

// critical section

unlock(&R2)
unlock(&R1)
```

```
Thread 1

lock(&R2)
lock(&R1)

// critical section

unlock(&R1)
unlock(&R2)
```

Circular wait

Thread 0 (*running*)	Thread 1
`lock(&R1)`	`lock(&R2)`
`lock(&R2)`	`lock(&R1)`
`// critical section`	`// critical section`
`unlock(&R2)`	`unlock(&R1)`
`unlock(&R1)`	`unlock(&R2)`

Circular wait

```
Thread 0

lock(&R1)
lock(&R2)

// critical section

unlock(&R2)
unlock(&R1)
```

```
Thread 1 (running)

lock(&R2)
lock(&R1)

// critical section

unlock(&R1)
unlock(&R2)
```

Circular wait

Thread 0 (*running*)

```
lock(&R1)
lock(&R2)
```

Stalled!

```
// critical section

unlock(&R2)
unlock(&R1)
```

Thread 1

```
lock(&R2)
lock(&R1)

// critical section

unlock(&R1)
unlock(&R2)
```

Circular wait

Thread 0

```
lock(&R1)
lock(&R2)

// critical section

unlock(&R2)
unlock(&R1)
```

Stalled!

Thread 1 (*running*)

```
lock(&R2)
lock(&R1)

// critical section

unlock(&R1)
unlock(&R2)
```

Stalled!

- *What situation are we in?* Deadlock
- *How can we avoid deadlock?* Use consistent lock orderings!

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, etc…
- 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 Binary Grow-Only Trees

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!
- The tree structure only supports an **`insert`** operation.
- Note that this BST does **not** support lookup or removal.

Starter Code: Thread Safe Trees

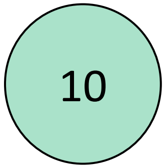
- Standard tree node struct that stores the value as well as it's left and right children.

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

```
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;
}
```

Example Trace

- Suppose we want to do **`insert(8)`** and **`insert(12)`** using two different threads on the tree below.
- Do we observe any racy behavior?



Original Tree

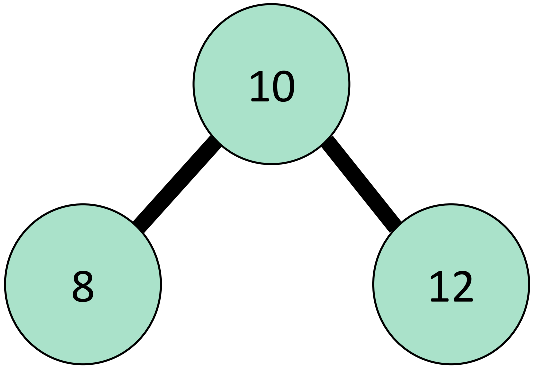
Example Trace

- Thread 1 enters the “left case” and finds that **`t->left = NULL`**
- Thread 2 enters the “right case” and finds that **`t->right = NULL`**
- Both proceed to create the new nodes.

```
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;
}
```

Example Trace

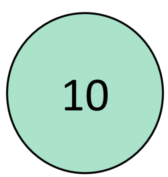
- We only get one resultant tree!



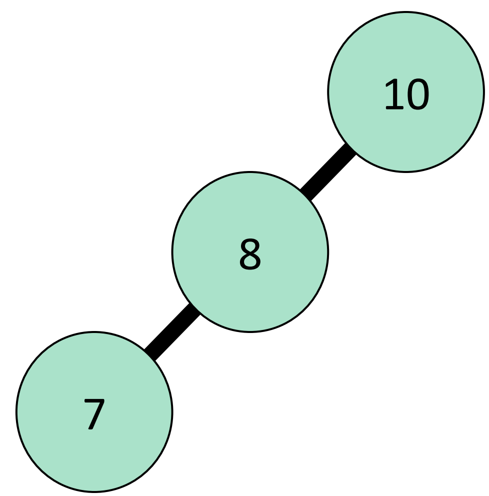
- We observed no race - there is only one possible tree.
- Is this always the case? Does this mean our code is race free?

Activity 1: Identify the Race

- Suppose we want to do **`insert(8)`** and **`insert(7)`** using two different threads on the tree below.
- Get into groups of 3-4 and try to identify the various possible outcomes. Draw out the possible resulting trees!



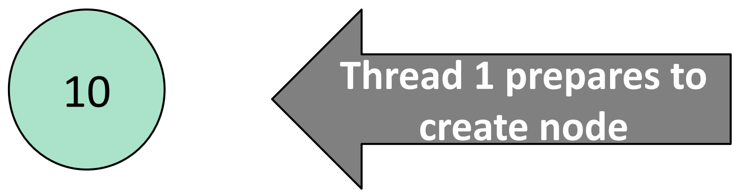
Original Tree



One Possible (correct) Tree

Identifying Race Condition

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

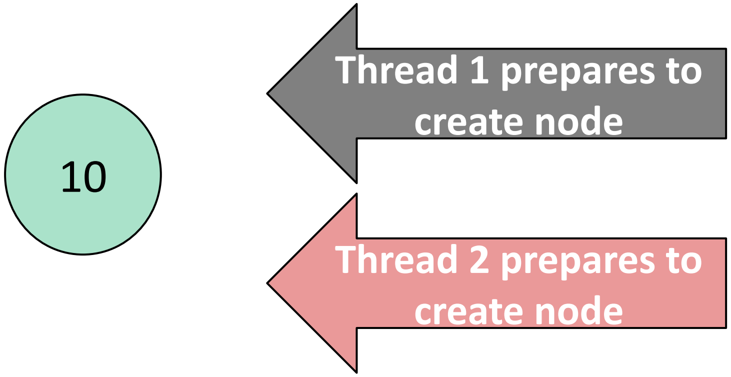


Relevant Case:

```
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

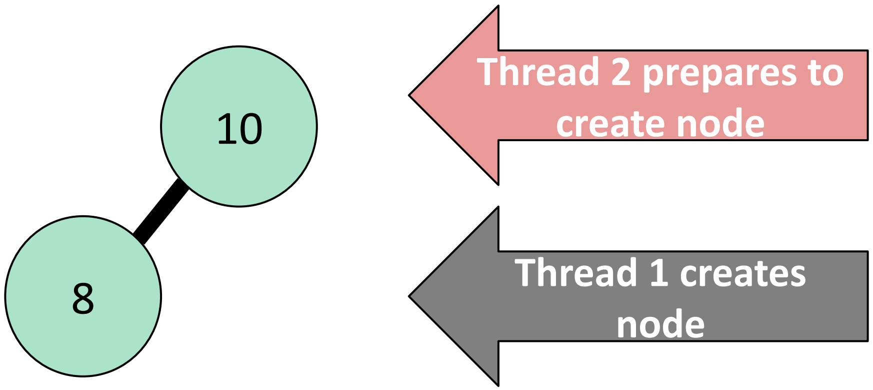


Relevant Case:

```
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

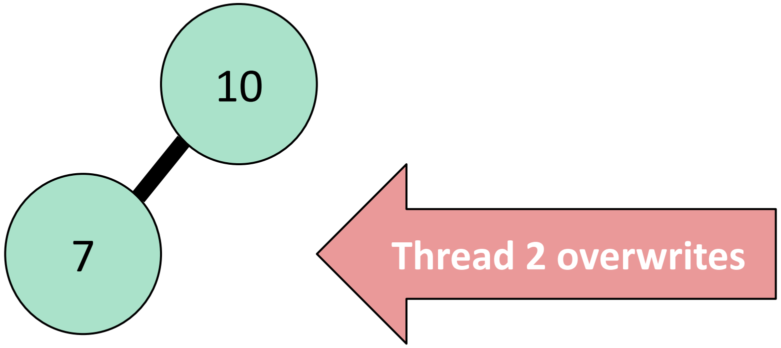
- Now thread 1 continues to run, creating the left node with
  `val = 8`



- 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 new left node, losing the node written by thread 1



- Unsafe behavior!

Why Did The Race Occur?

- What is the shared resource **in this scenario**?
  - The root of the tree - more specifically the left node field
  - Both threads attempt a **`NULL`** check on the left child, which is unsafe (*TOCTTOU*)

Disclaimer: We want to create a locking design that is thread-safe in all scenarios!

Activity 1: Creating a Simple Lock Design

- Good practice for designing + implementing a concurrent system is to **start simple** and then add levels of complexity
- What is an example of a simple design?
  - Using a single mutex to lock the entire tree!
- Get into groups of 3-4 and try to implement a **coarse grain locking design** that makes our tree structure thread-safe!

Solution 1: Coarse Grain Locking

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

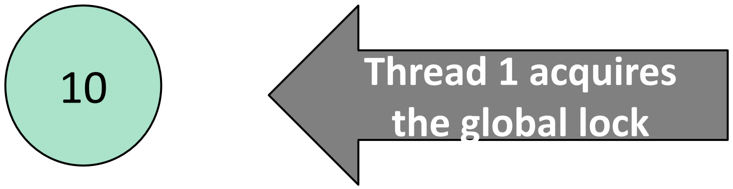
```
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

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

Activity 2: Coarse Grain Analysis

- Now that we have a locking design, let's revisit the concurrent **`insert(7)`** and **`insert(8)`**.
- Try to trace out an execution order of these instructions. What do you observe?

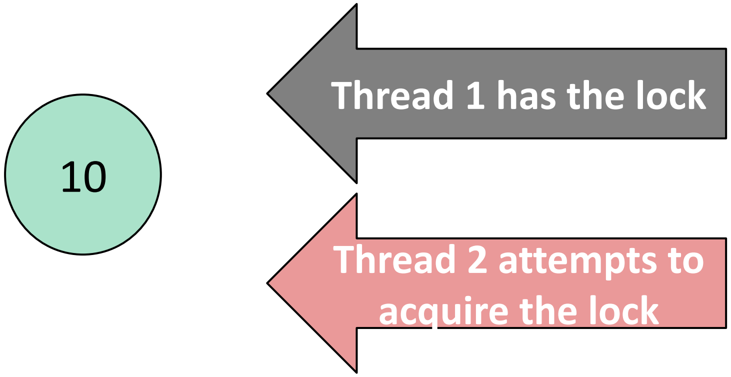
Revisiting Example

- Suppose thread 1 runs first.



Revisiting Example

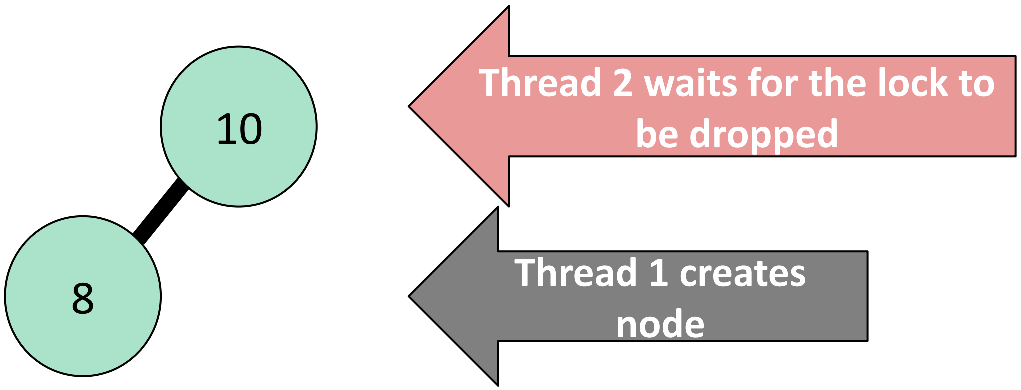
- Now suppose thread 2 runs.



Thread 2 **fails** to acquire the lock! It must wait for thread 1 to drop the lock first.

Revisiting Example

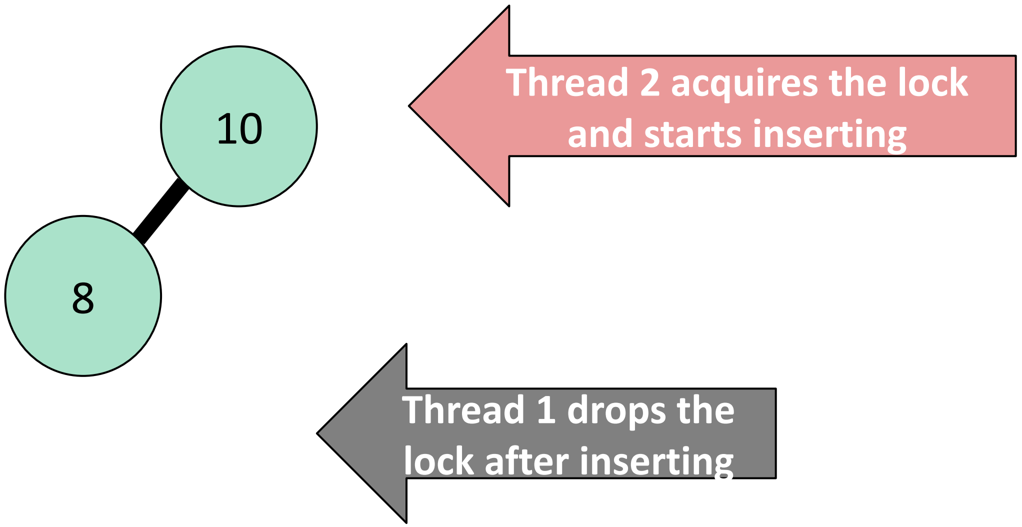
- Now thread 1 continues to run, creating the left node with **`val = 8`**



- Note that thread 2 knows nothing about **`t->left`**; it has not entered the insert routine.

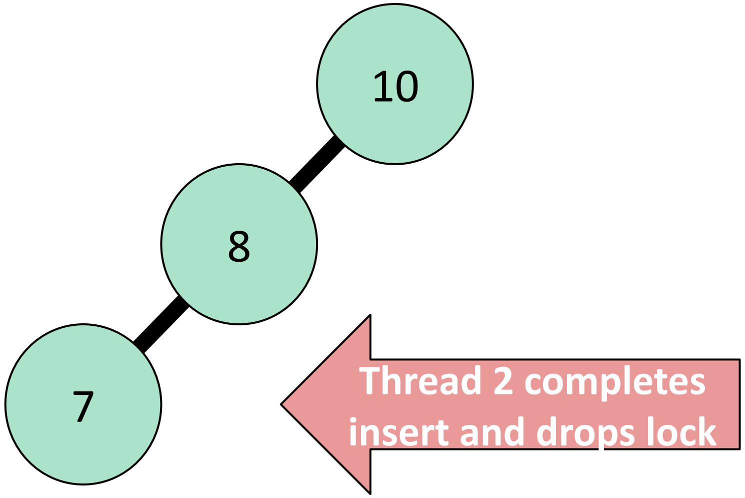
Revisiting Example

- Thread 1 completes, and now thread 2 runs!



Revisiting Example

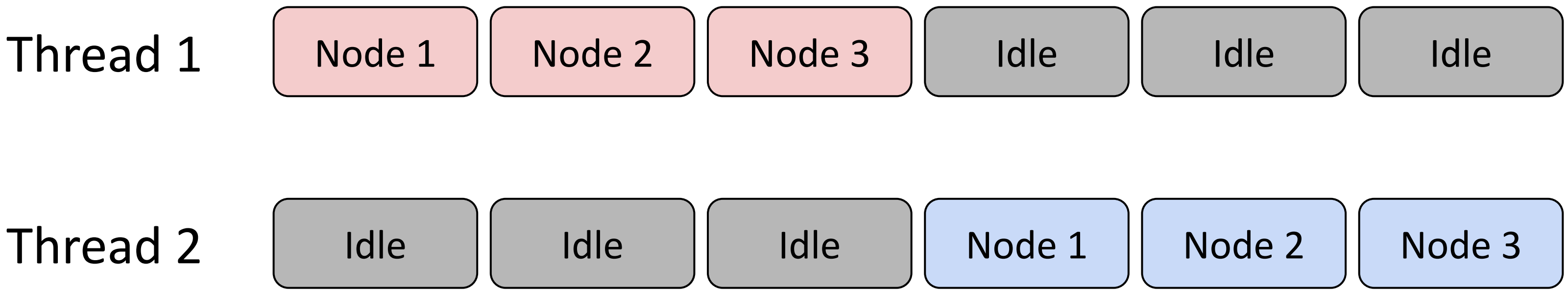
- Thread 2 continues inserting and now it sees the changes that thread 1 has made to **`root->left`**



- Now we have a correct tree!

Analysis: Coarse Grain Locking

- Wrapping each function call in locks makes all execution sequential - as we saw in the previous example.
- Looking at another example, assuming each thread's call takes 3 iterations through the tree, we can see the following behavior!

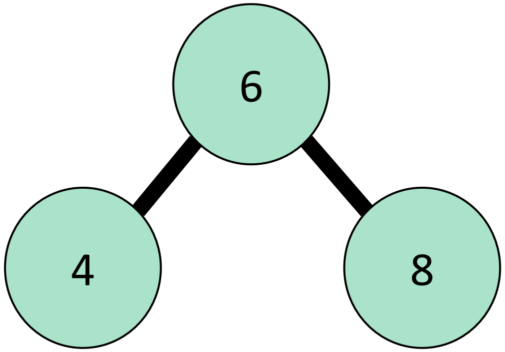


Activity 3: Trace

- Our original goal was to design a concurrent program - however, all of our accesses are sequential.
- Can we use our threads more effectively? Let's examine example traces to observe potential parallelism and whether it is utilized!

Activity 3: Trace

- Consider the following tree:



- Try tracing out the behavior of these 2 scenarios:
  - **`insert(2)`**, **`insert(3)`** in parallel
  - **`insert(3)`**, **`insert(9)`** in parallel

Activity 3: Trace

- Recall the first example, where a lock was not required to ensure correct behavior. Can we find other similar scenarios?
- **`insert(2)`**, **`insert(3)`** access the left field of node 4, meaning the accesses must be protected (TOCTTOU issue)
  - This is similar to our first racy trace!
- **`insert(3)`**, **`insert(9)`** access different branches, which means consequent checks are independent - no race will occur

Activity 3: Trace

- **`insert(2)`**, **`insert(3)`** must be protected, so they must run sequentially with respect to each other
  - Hint: How can we use locks to enforce this ordering?
- What about **`insert(3)`**, **`insert(9)`**? Do these operations also require sequential ordering?
  - **No!** (Hint: How might this be reflected in our lock design?)
- Can we put this all together to create a non-sequentially ordered locking mechanism?

Discussion: Reducing Shared Resource Size

- The previous examples showed us there is parallelism in the branches. What is a simple design that reveals branch independence?
    - Use two global locks to protect each branch!
- Does this design always perform better than the coarse grain locking design? Think about varying tree structures.
    - Balanced trees result in good concurrency
    - If all nodes are in one branch, we still run serially…

Discussion: Reducing Shared Resource Size

- We've successfully reduced the size of our shared resources, consequently reducing our critical section.
- However, we also found some cases don't perform well… Can we do better?
  - In other words, can we **further** reduce the size of our shared resource?

Activity 4: Fine Grain Locking

- As groups, brainstorm a locking design that is thread-safe, but is not always sequentially ordered.
  - Use mutexes [you may modify the struct :)]

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

```
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;
}
```

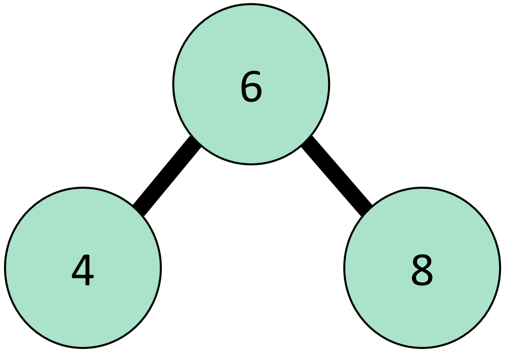
Solution 4: Fine Grain locking

- 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)

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

Example: Fine Grain locking

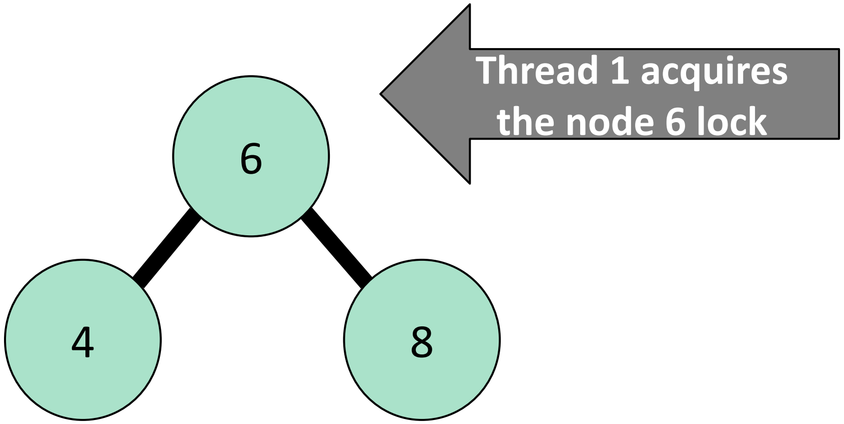
- Let's consider the **`insert(3)`**, **`insert(9)`** in parallel case.



Original Tree

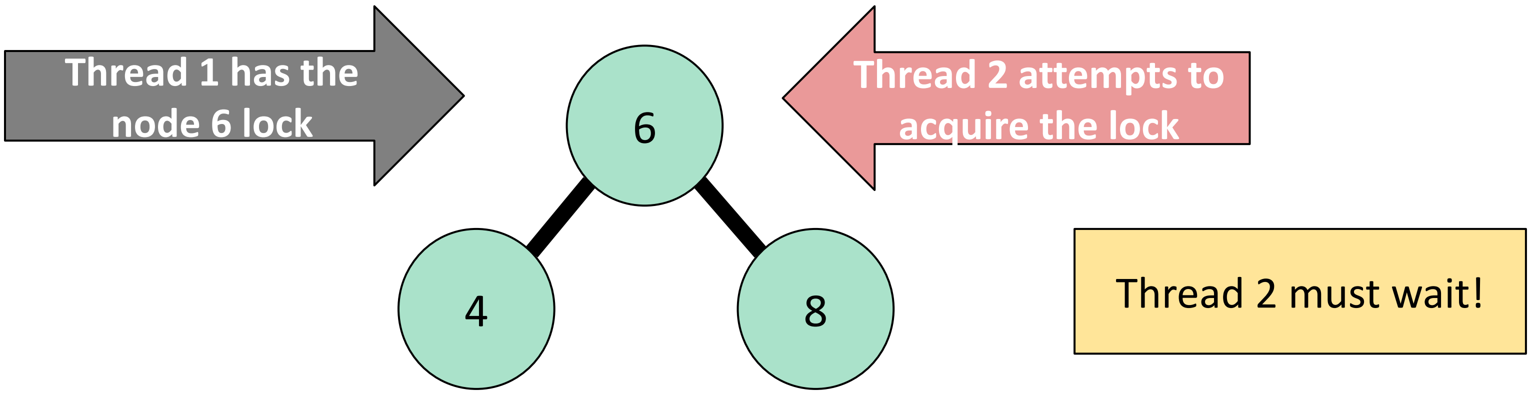
Example: Fine Grain locking

- Suppose thread 1 runs first (`insert(3)`):



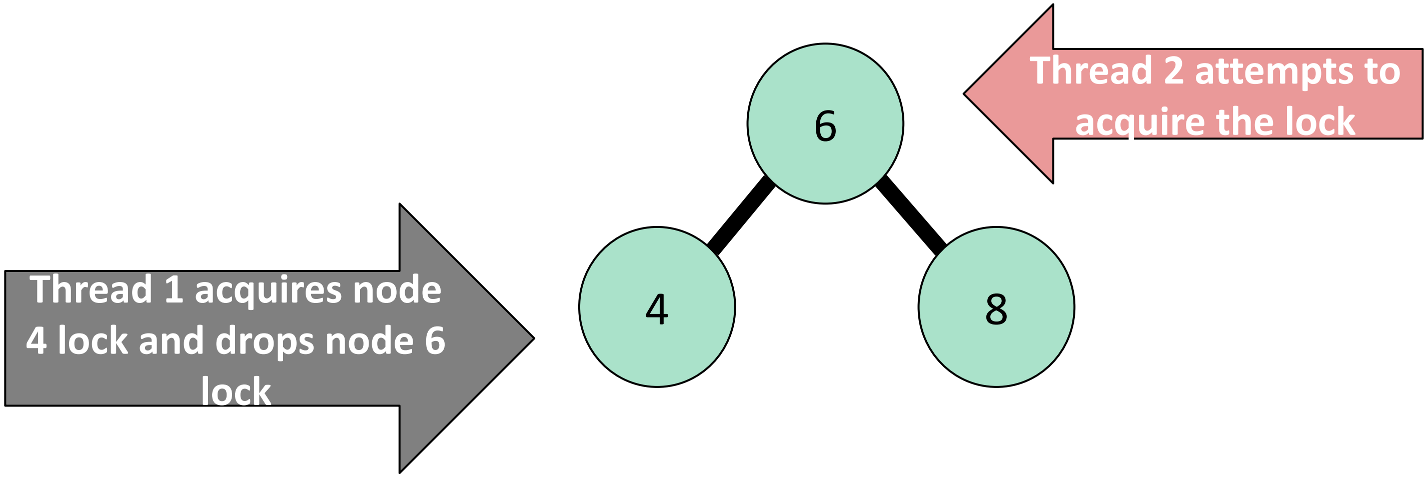
Example: Fine Grain locking

- Now suppose thread 2 runs (**`insert(9)`**):



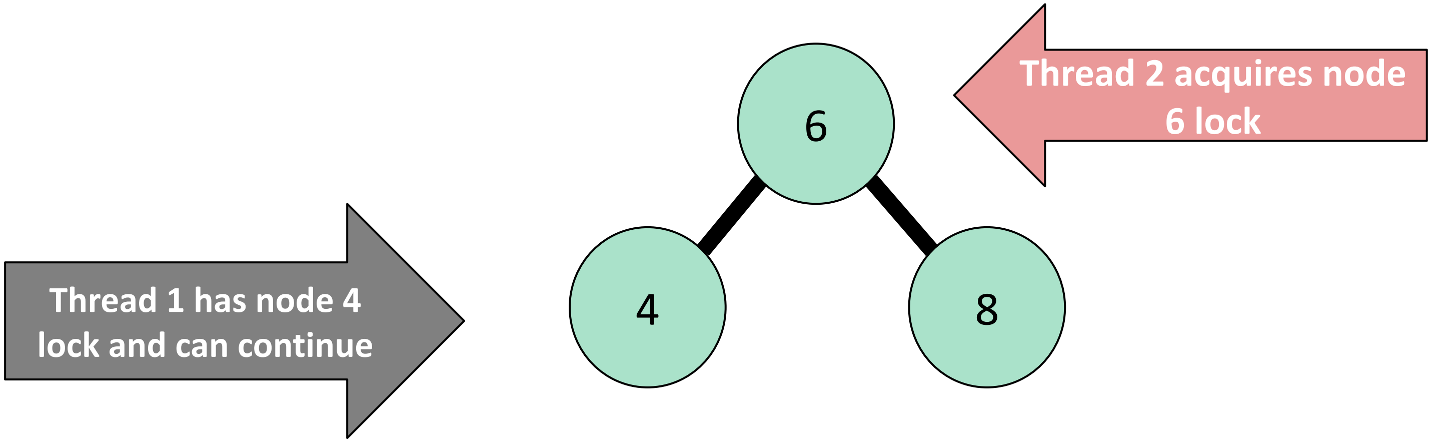
Example: Fine Grain locking

- Now thread 1 continues:



Example: Fine Grain locking

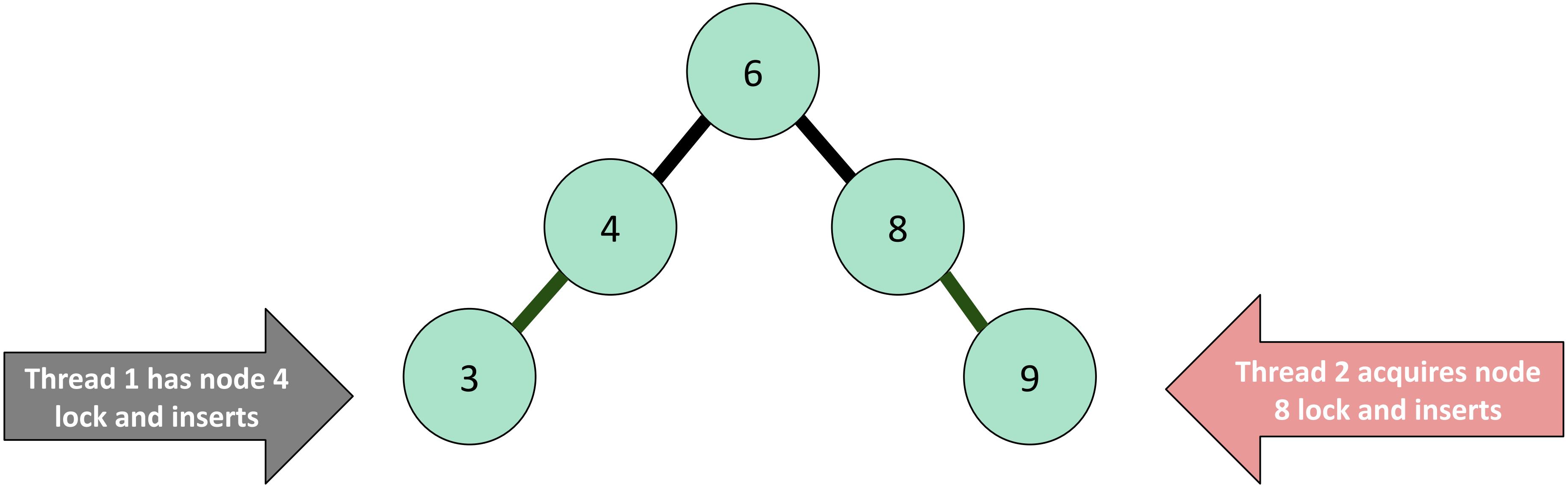
- But wait, thread 2 can now make progress!



- Note: Since there is no need for thread 2 to wait for thread 1, it is **possible** for the threads to run concurrently

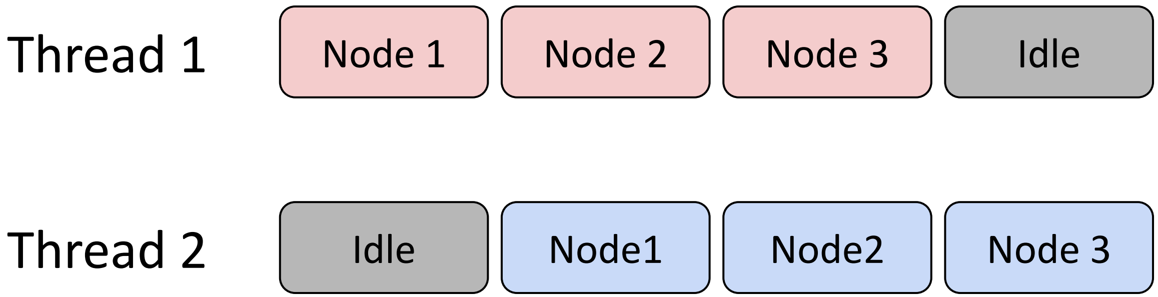
Example: Fine Grain locking

- Both threads can concurrently run to completion!



Analysis: Solution 4

- 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

Analysis: Solution 4

- In our first coarse-grain solution, any lock protected the entire tree - creating a large critical section. What about this solution?
- **Drastic Reduction!**
  - We now only block off one node access at a time (as pointed to by the previous diagram)
- A more detailed analysis of parallelism and locking is beyond the scope of 15-213 - look into 15-346 / 15-410 / 15-418!

Wrapping Up

- **`sfslab`** to be released
  - On November 25th
  - Due Date: *Dec 04th*
- **`proxylab`** released
  - Due Date: *November 25th*
- *Code Reviews for* **`tshlab`**

The End