# COL380

## Introduction to
## Parallel & Distributed Programming

# Agenda

- Synchronization properties and examples

Quiz on 13th at 12 [LH 503/504]

# Atomics

```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
  // t = var.load();
  // if(t == expected) {
  //      var.store(newval);
  //      return true
  // } else {
  //      return false
  // }
```

```
#pragma atomic
    var++;
```

```
#pragma omp atomic capture compare
{
   old = svar;
   if (old == expected) svar = newval;
}
   // old == expected ⇒ success
```

# Atomics

```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
```

#pragma atomic

```
std::atomic<node<T>*> top;
…
void push(const T& data) {
      node<T>* new_node = new node<T>(data);

      // put the current value of top into new_node->next
      new_node->next = top.load();

     // Update top to point to the new node
          top.store(new_node);
}
```

# Atomics

top

```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
```

#pragma atomic

```
std::atomic<node<T>*> top;
…
void push(const T& data) {
      node<T>* new_node = new node<T>(data);

      // put the current value of top into new_node->next
      new_node->next = top.load();

     // Update top to point to the new node
          top.store(new_node);
}
```

# Atomics


```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
```

#pragma atomic

```
std::atomic<node<T>*> top;
...
void push(const T& data) {
    node<T>* new_node = new node<T>(data);

    // put the current value of top into new_node->next
    new_node->next = top.load();

    // Update top to point to the new node
        top.store(new_node);
}
```

# Atomics


```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
```

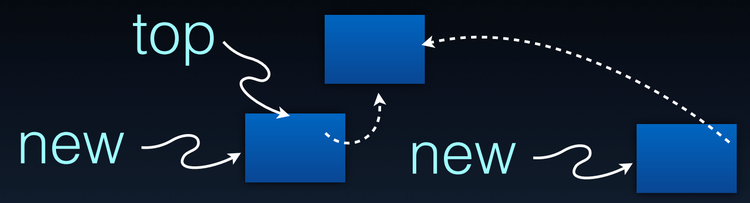
#pragma atomic

```
std::atomic<node<T>*> top;
...
void push(const T& data) {
    node<T>* new_node = new node<T>(data);

    // put the current value of top into new_node->next
    new_node->next = top.load();

    // Update top to point to the new node
        top.store(new_node);
}
```

# Atomics




```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
```

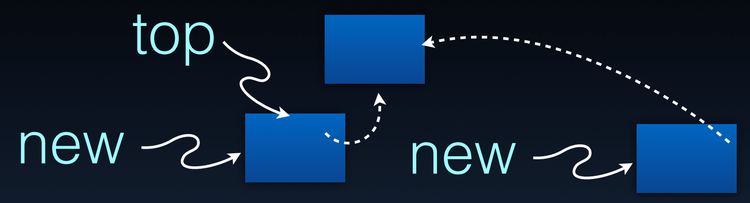
#pragma atomic

```
std::atomic<node<T>*> top;
...
void push(const T& data) {
    node<T>* new_node = new node<T>(data);

    // put the current value of top into new_node->next
    new_node->next = top.load();

    // Update top to point to the new node
        top.store(new_node);
}
```

# Atomics




```
std::atomic<int> var(0);

var.compare_exchange_strong(expected, newval);
  // Atomically:
```

#pragma atomic

```
std::atomic<node<T>*> top;
...
void push(const T& data) {
      node<T>* new_node = new node<T>(data);

      // put the current value of top into new_node->next
      do new_node->next = top.load();

     // make new_node the top, as long as top still equals new_node->next
      while(!top.compare_exchange_strong(new_node->next, new_node));
}
```

# Synchronization

- Events should happen together
  - ➡ Barrier
- Events should NOT happen together
  - ➡ Mutual Exclusion, Critical Section
- A should happen before B
  - ➡ Conditions
- Lower level Primitives
  - ➡ Locks, Semaphores, Registers, Transactional memory

    Overhead?

# Synchronization

- Events should happen together
  - ➡ Barrier
- Events should NOT happen together
  - ➡ Mutual Exclusion, Critical Section
- A should happen before B
  - ➡ Conditions
- Lower level Primitives
  - ➡ Locks, Semaphores, Registers, Transactional memory

Overhead?

Progress
- ➡ Starvation
- ➡ Deadlock

Safety

Liveness

# Synchronization

- Events should happen together
  - ➡ Barrier
- Events should NOT happen together
  - ➡ Mutual Exclusion, Critical Section
- A should happen before B
  - ➡ Conditions
- Lower level Primitives
  - ➡ Locks, Semaphores, Registers, Transactional memory

    Overhead?

Progress
- ➡ Starvation
- ➡ Deadlock

Blocking vs Non-blocking

Safety

Liveness

Busy-wait vs OS-scheduled

Fairness

Subodh Kumar

# Fairness

- Strong Fairness
  - ➡ If any synchronizer is ready infinitely often, it should be executed infinitely often
- Weak Fairness
  - ➡ If any synchronizer is ready, it should be executed eventually

# Progress

| | Not lock-based<br>Independent of Scheduler | Lock-based<br>Depends on Scheduler |
|---|---|---|
| Everyone Progresses | Wait Free | Starvation Free |
| Someone Progresses | Lock Free | Deadlock Free |

# Progress

| | Not lock-based<br>Independent of Scheduler | Lock-based<br>Depends on Scheduler |
|---|---|---|
| Everyone Progresses | Wait Free | Starvation Free |
| Someone Progresses | Lock Free | Deadlock Free |

All synchronizers succeed in a finite time

# Progress

| | Not lock-based<br>Independent of Scheduler | Lock-based<br>Depends on Scheduler |
|---|---|---|
| Everyone Progresses | Wait Free | Starvation Free |
| Someone Progresses | Lock Free | Deadlock Free |

Some synchronizer succeeds in a finite time

# Progress

| | Not lock-based<br>Independent of Scheduler | Lock-based<br>Depends on Scheduler |
|---|---|---|
| Everyone Progresses | Wait Free | Starvation Free |
| Someone Progresses | Lock Free | Deadlock Free |

All waiting synchronizers are scheduled

# Progress

| | Not lock-based<br>Independent of Scheduler | Lock-based<br>Depends on Scheduler |
|---|---|---|
| Everyone Progresses | Wait Free | Starvation Free |
| Someone Progresses | Lock Free | Deadlock Free |

Not all synchronizers are blocked

Progress

Liveness

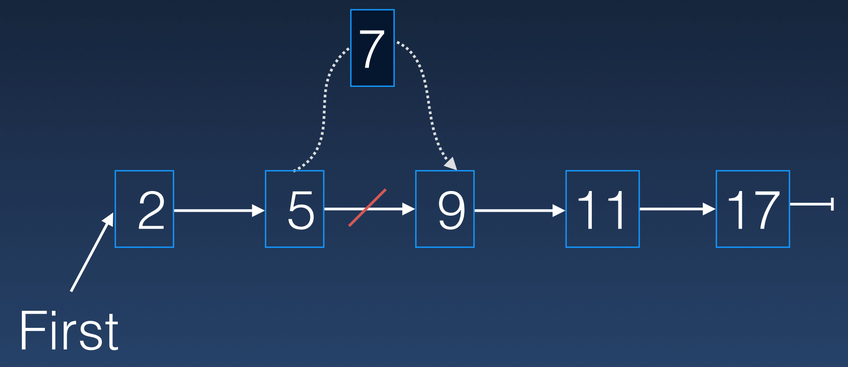
| | Not lock-based<br>Independent of Scheduler | Lock-based<br>Depends on Scheduler |
|---|---|---|
| Everyone Progresses | Wait Free | Starvation Free |
| Someone Progresses | Lock Free | Deadlock Free |

# Lock

- Lock “resources”
- Process
- Unlock “resources”

# Lock

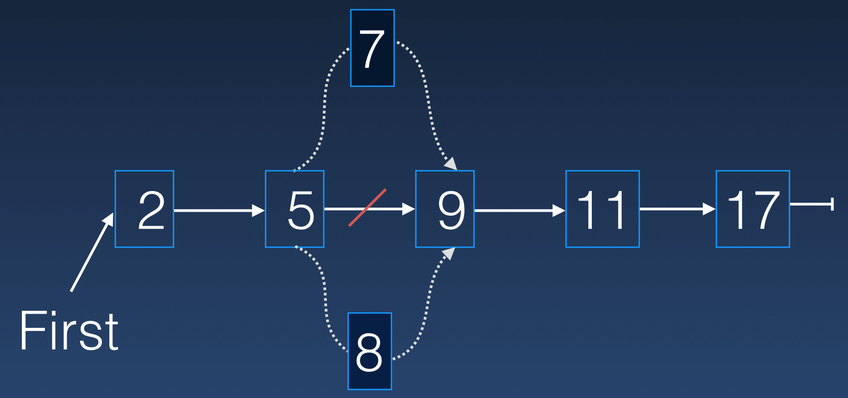
- Lock “resources”
- Process
- Unlock “resources”



# Lock

- Lock “resources”
- Process
- Unlock “resources”

Correctness?

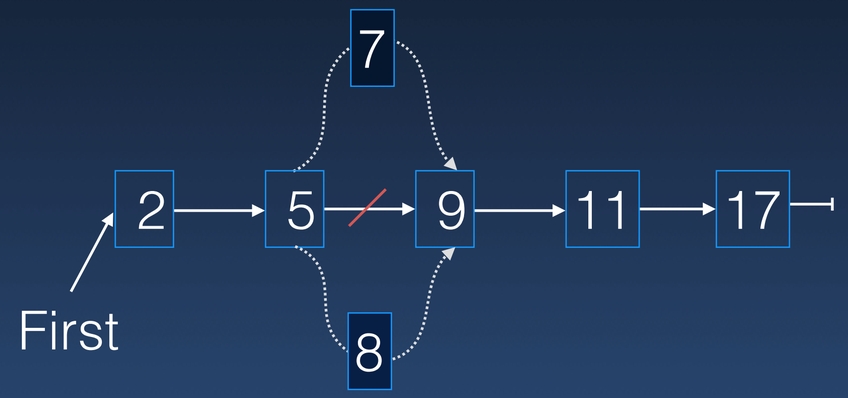


# Lock

- Lock “resources”
- Process
- Unlock “resources”

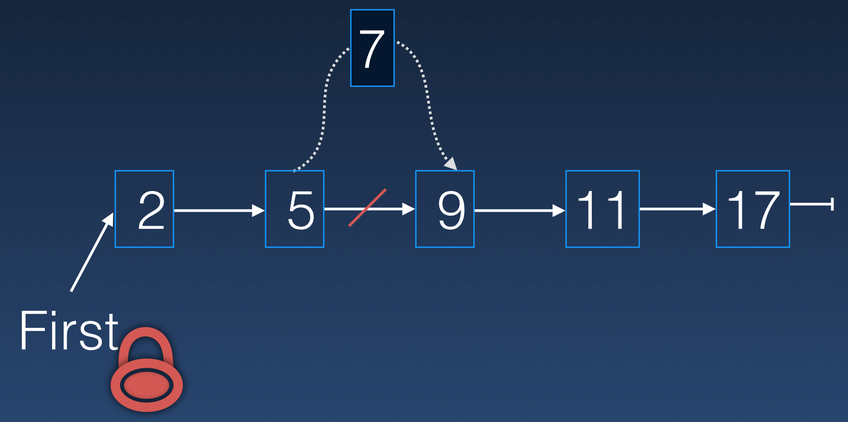
Correctness?

“Sequential Equivalence”



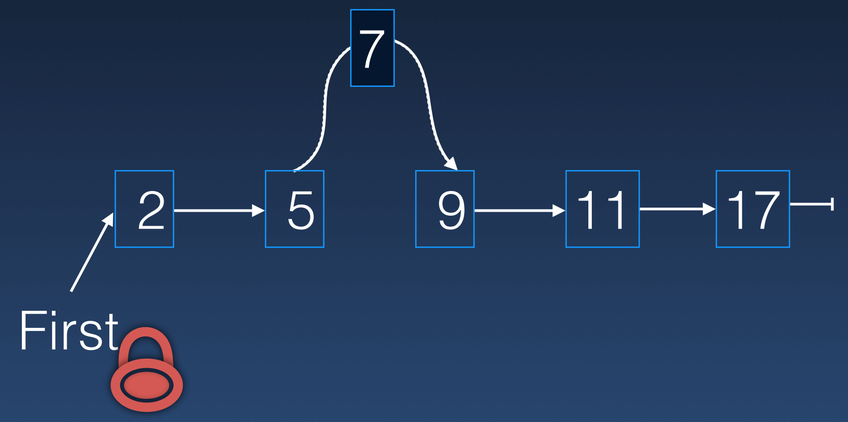
# Lock

- Lock “resources”
- Process
- Unlock “resources”



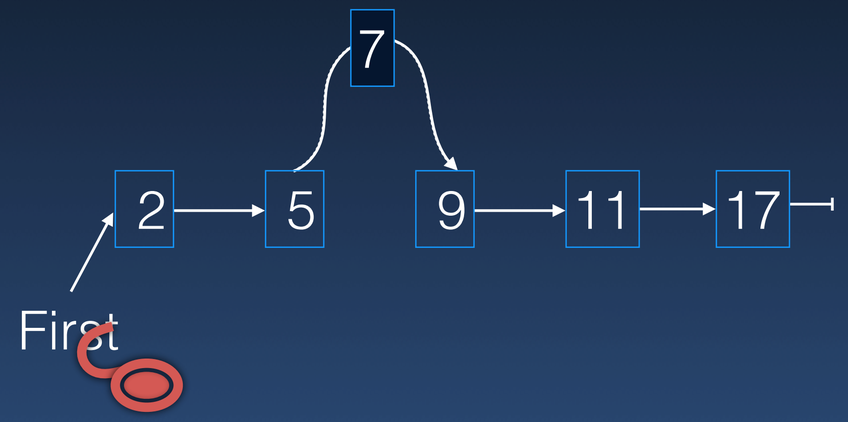
# Lock

- Lock “resources”
- Process
- Unlock “resources”



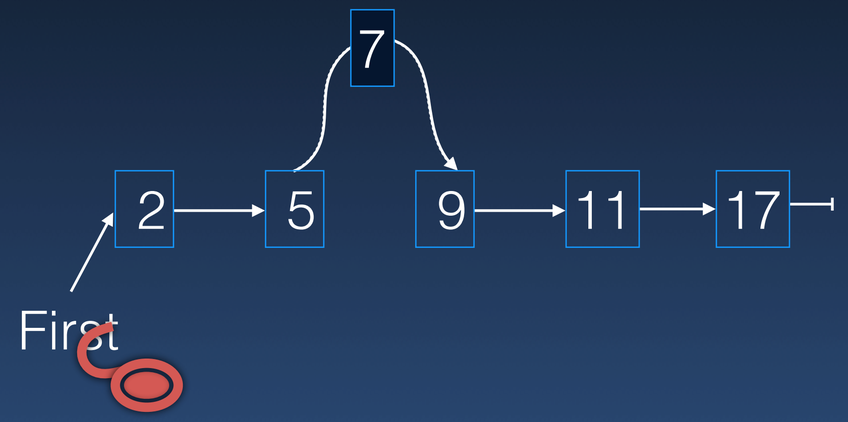
# Lock

- Lock “resources”
- Process
- Unlock “resources”



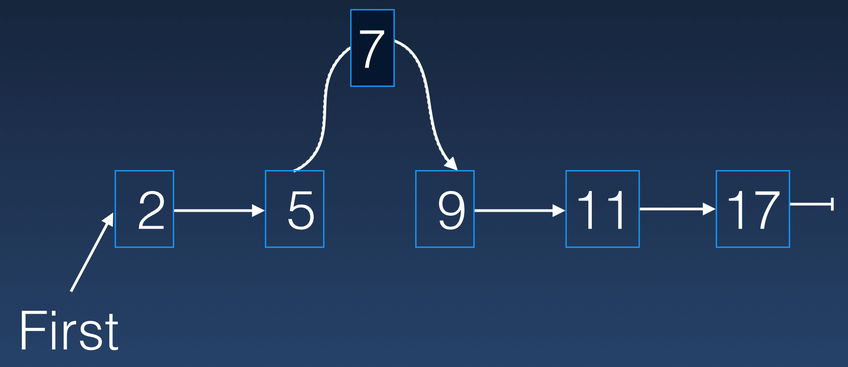
# Lock

- Lock “resources” lock($lockA$)
- Process
- Unlock “resources”



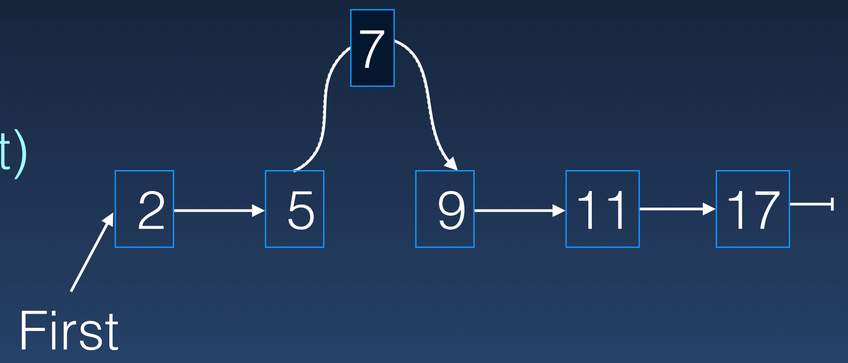
# Lock

- Lock “resources” lock($lockA$)
- Process
- Unlock “resources”



# Lock

- Lock “resources” — lock($lockA$)
- Process — pred = Find(key)<br>pred.nxt = Node(key, pred.nxt)
- Unlock “resources” — unlock($lockA$)



# Lock

- Lock “resources”
- Process
- Unlock “resources”

<Request> [?block] <Acquired>

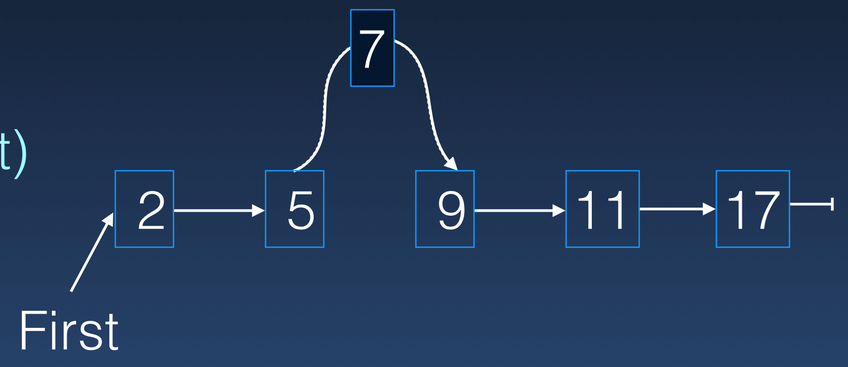
lock($lockA$)

pred = Find(key)

pred.nxt = Node(key, pred.nxt)

unlock($lockA$)

<Release> [schedule]



# Lock

- Lock “resources”
- Process
- Unlock “resources”

<Request> [?block] <Acquired>

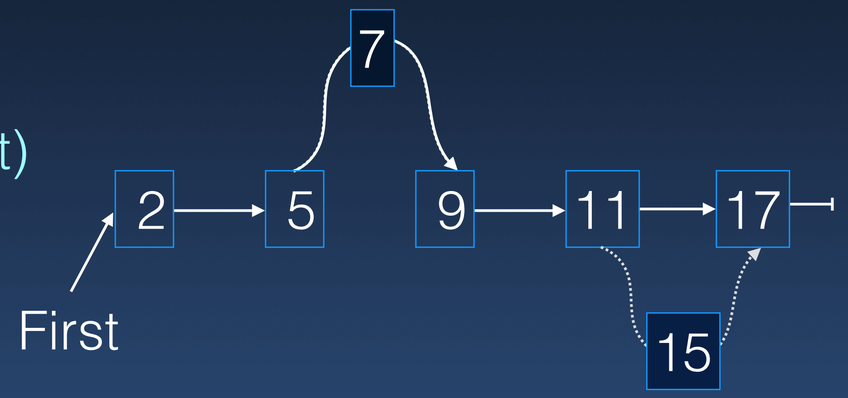
lock($lockA$)

pred = Find(key)

pred.nxt = Node(key, pred.nxt)

unlock($lockA$)

<Release> [schedule]

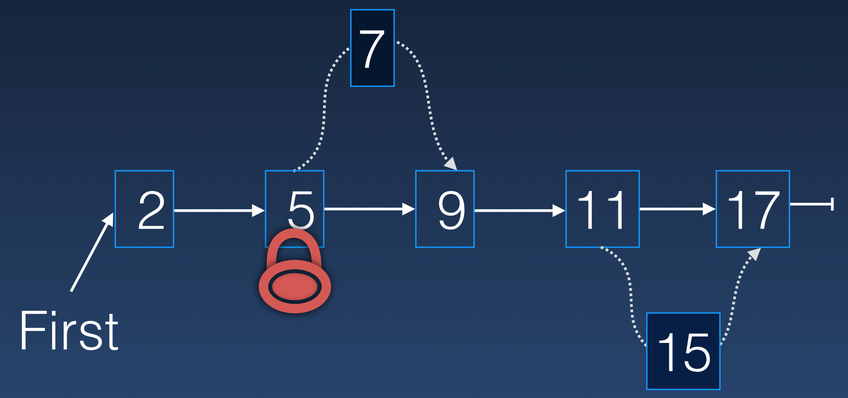



```
C++:
    std::mutex m;
    std::lock(m);
    doCriticalwork();
    std::unlock(m);
```

Lock the entire list?

# Lock

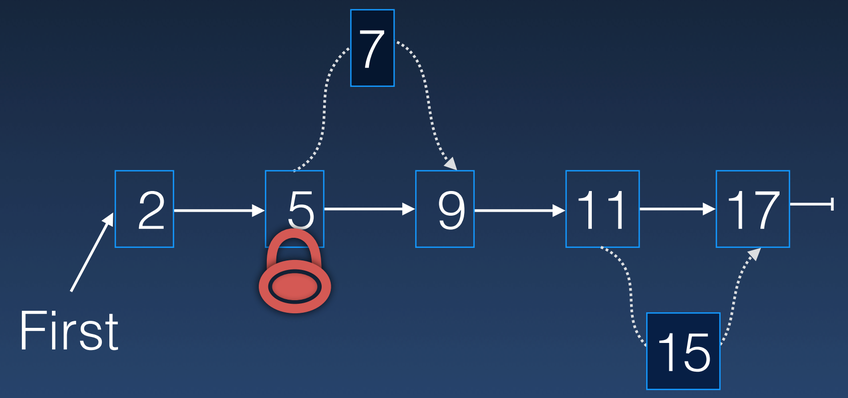
- Lock “resources”
- Process
- Unlock “resources”




```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”



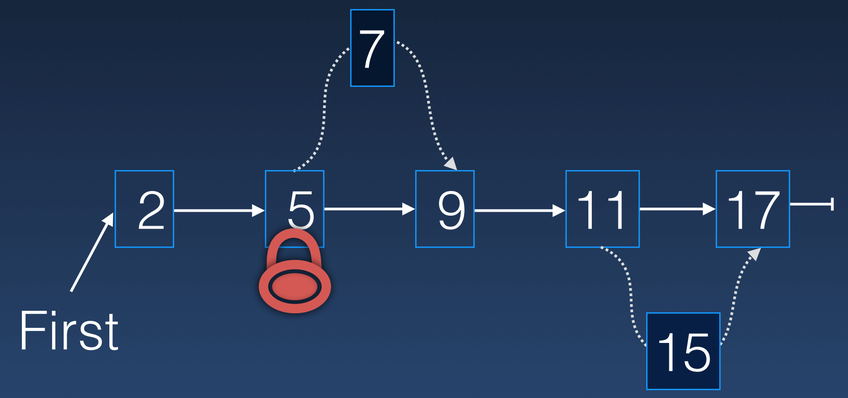

Insertion Loop

```
lock(pred.lock)
if(key in [pred.key:pred.nxt.key)) {
    pred.nxt = Node(key, pred.nxt, new(Lock))
}
unlock(pred.lock)
pred = pred.nxt
```

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”




Insertion Loop

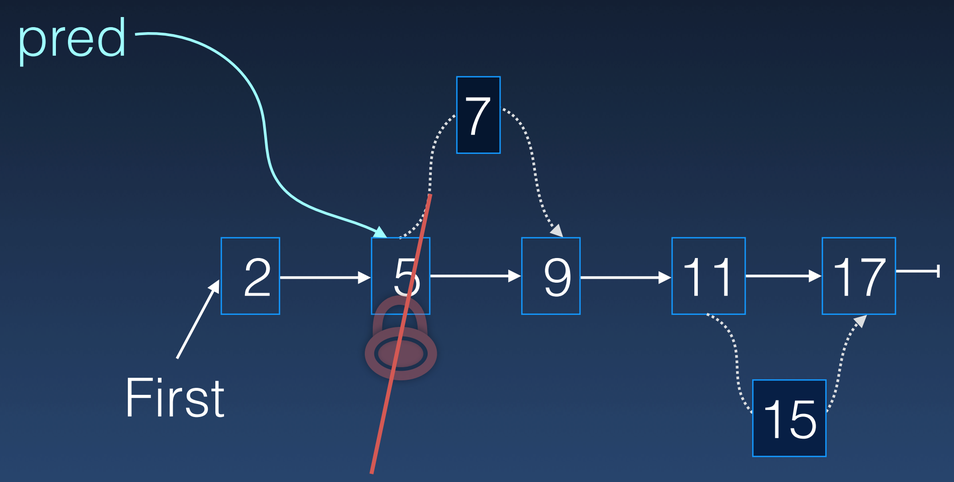
```
lock(pred.lock)
if(key in [pred.key:pred.nxt.key)) {
      pred.nxt = Node(key, pred.nxt, new(Lock))
}
unlock(pred.lock)
pred = pred.nxt
```

Is pred still referring to the right node?

```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”




Insertion Loop

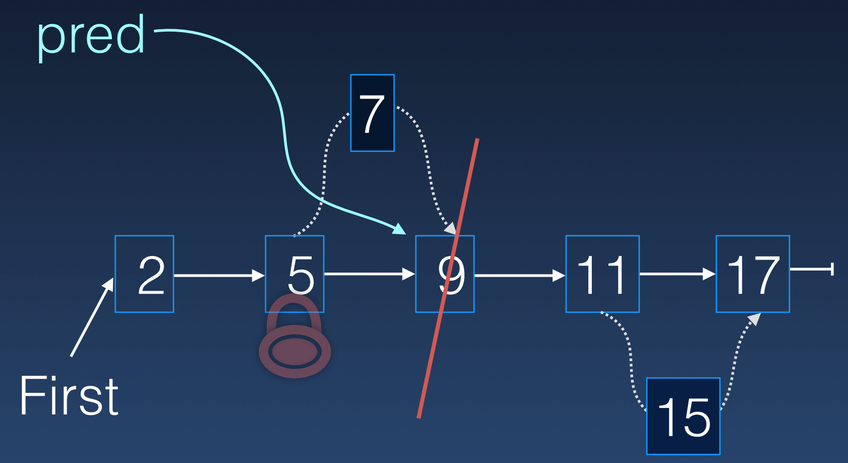
```
lock(pred.lock)
if(key in [pred.key:pred.nxt.key)) {
      pred.nxt = Node(key, pred.nxt, new(Lock))
}
unlock(pred.lock)
pred = pred.nxt
```

Is pred still referring to the right node?

```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”




Insertion Loop

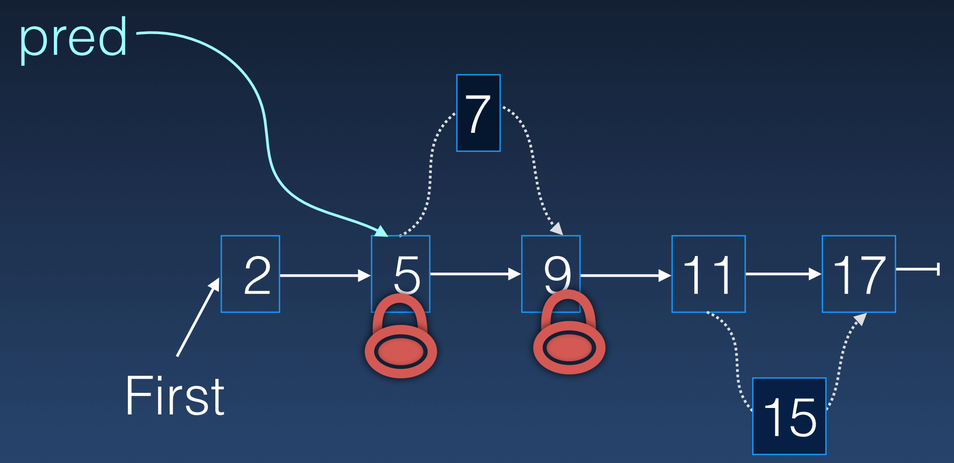
```
lock(pred.lock)
if(key in [pred.key:pred.nxt.key)) {
      pred.nxt = Node(key, pred.nxt, new(Lock))
}
unlock(pred.lock)
pred = pred.nxt
```

Before unlocking pred, save its ‘nxt’?

```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”



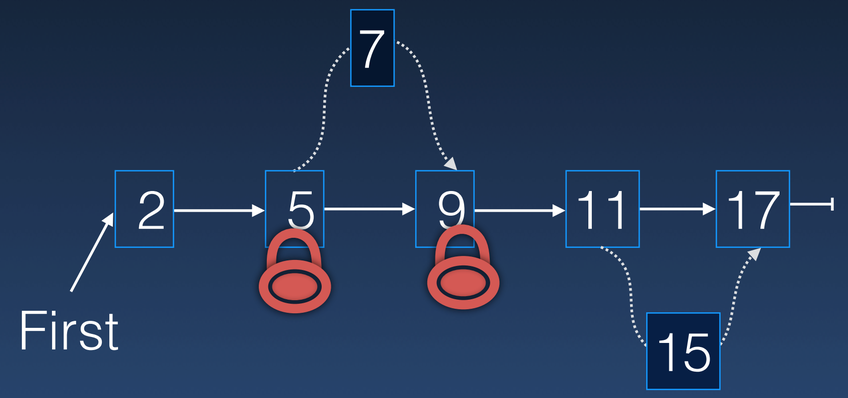

Insertion Loop

```
lock(pred.lock) And lock(pred.nxt.lock)
if(key in [pred.key:pred.nxt.key)) {
    pred.nxt = Node(key, pred.nxt, new(Lock))
}
unlock(pred.lock)
pred = pred.nxt
```

```
Node {
    Key   key
    Node nxt
    Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”



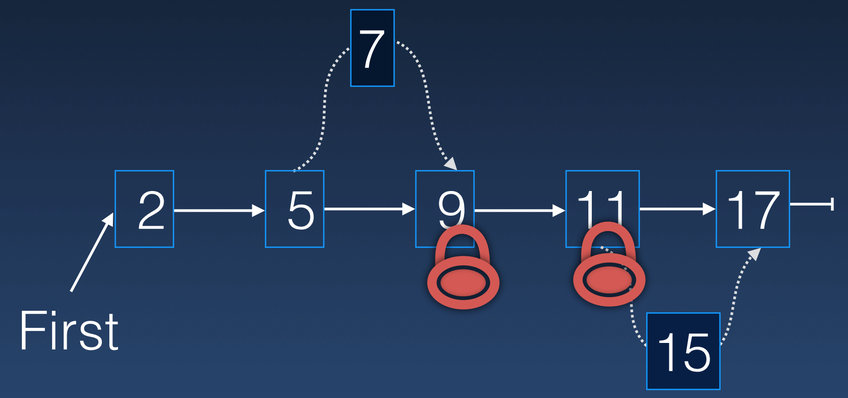

Insert

```
pred = first; lock(pred.lock); lock(pred.nxt.lock)
if(key in [pred.key:pred.nxt.key)) {
        pred.nxt = Node(key, pred.nxt, new(Lock))
        Unlock both
}
lock(pred.nxt.nxt.lock)
unlock(pred.lock)
pred = pred.nxt
```

```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”



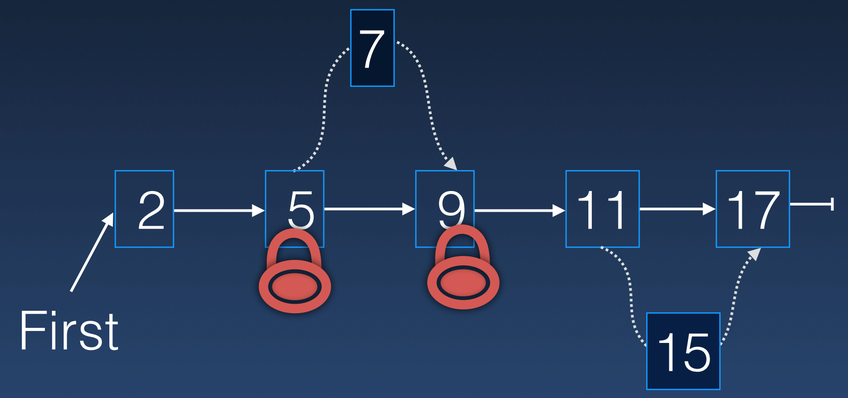

Insert

```
pred = first; lock(pred.lock); lock(pred.nxt.lock)
if(key in [pred.key:pred.nxt.key)) {
        pred.nxt = Node(key, pred.nxt, new(Lock))
        Unlock both
}
lock(pred.nxt.nxt.lock)
unlock(pred.lock)
pred = pred.nxt
```

```
Node {
   Key   key
   Node nxt
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”




Insert

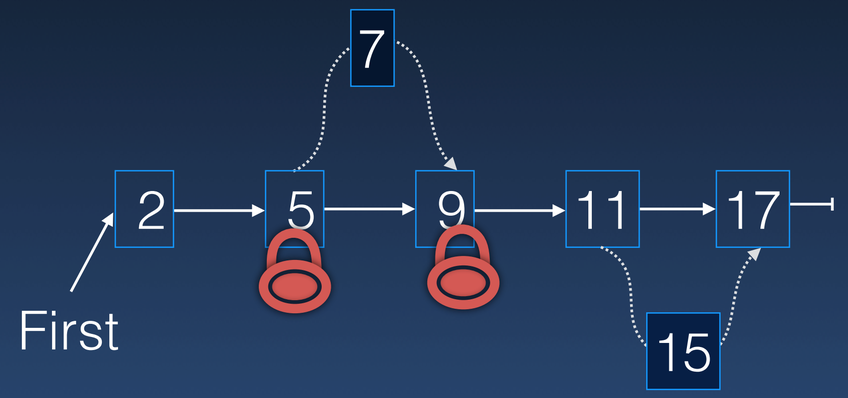
```
pred = first; lock(pred.lock); lock(pred.nxt.lock)
if(key in [pred.key:pred.nxt.key)) {
        pred.nxt = Node(key, pred.nxt, new(Lock))
        Unlock both
}
lock(pred.nxt.nxt.lock)
unlock(pred.lock)
pred = pred.nxt
```

Not atomic

```
Node {
    Key   key
    Node nxt   volatile
    Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”



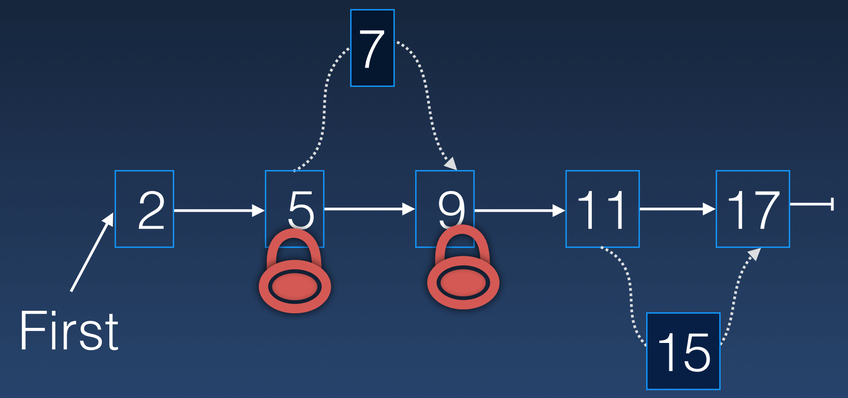

Insert

```
pred = first; lock(pred.lock); lock(pred.nxt.lock)
if(key in [pred.key:pred.nxt.key)) {
        pred.nxt = Node(key, pred.nxt, new(Lock))
        Unlock both
}
lock(pred.nxt.nxt.lock)
unlock(pred.lock)
pred = pred.nxt  Not atomic  (Update before unlocking)
```

```
Node {
   Key   key
   Node nxt   volatile
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”




Remove

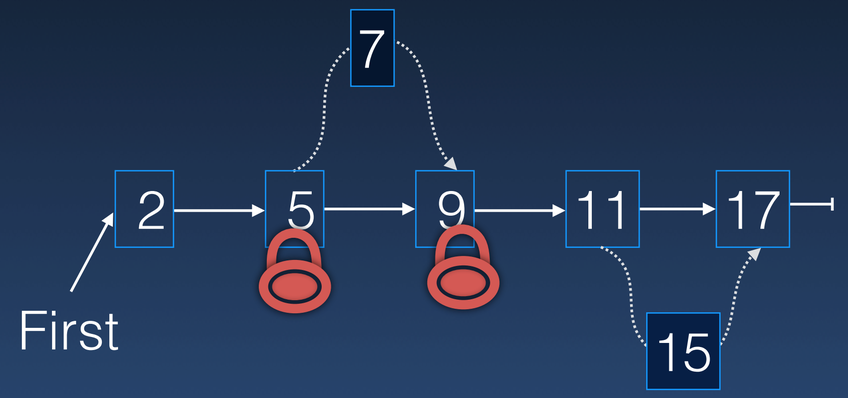
```
pred = first; lock(pred.lock); lock(pred.nxt.lock)
if(key in [pred.key:pred.nxt.key)) {
        pred.nxt = pred.nxt.nxt  Not atomic
        Unlock both
}
lock(pred.nxt.nxt.lock)
unlock(pred.lock)
pred = pred.nxt  Not atomic  (Update before unlocking)
```

```
Node {
   Key   key
   Node nxt   volatile
   Lock  lock
}
```

# Lock

- Lock “resources”
- Process
- Unlock “resources”

➡ Depend on others following protocol




```
Node {
    Key   key
    Node nxt   volatile
    Lock   lock
}
```

# Review

- Properties of Synchronization methods
- Compare and Swap based synchronization
- Example of locking protocol (using sorted list)