

# Shared Counters and Parallelism



*Christof Fetzer, TU Dresden*

*Based on slides by Maurice Herlihy  
and Nir Shavit*

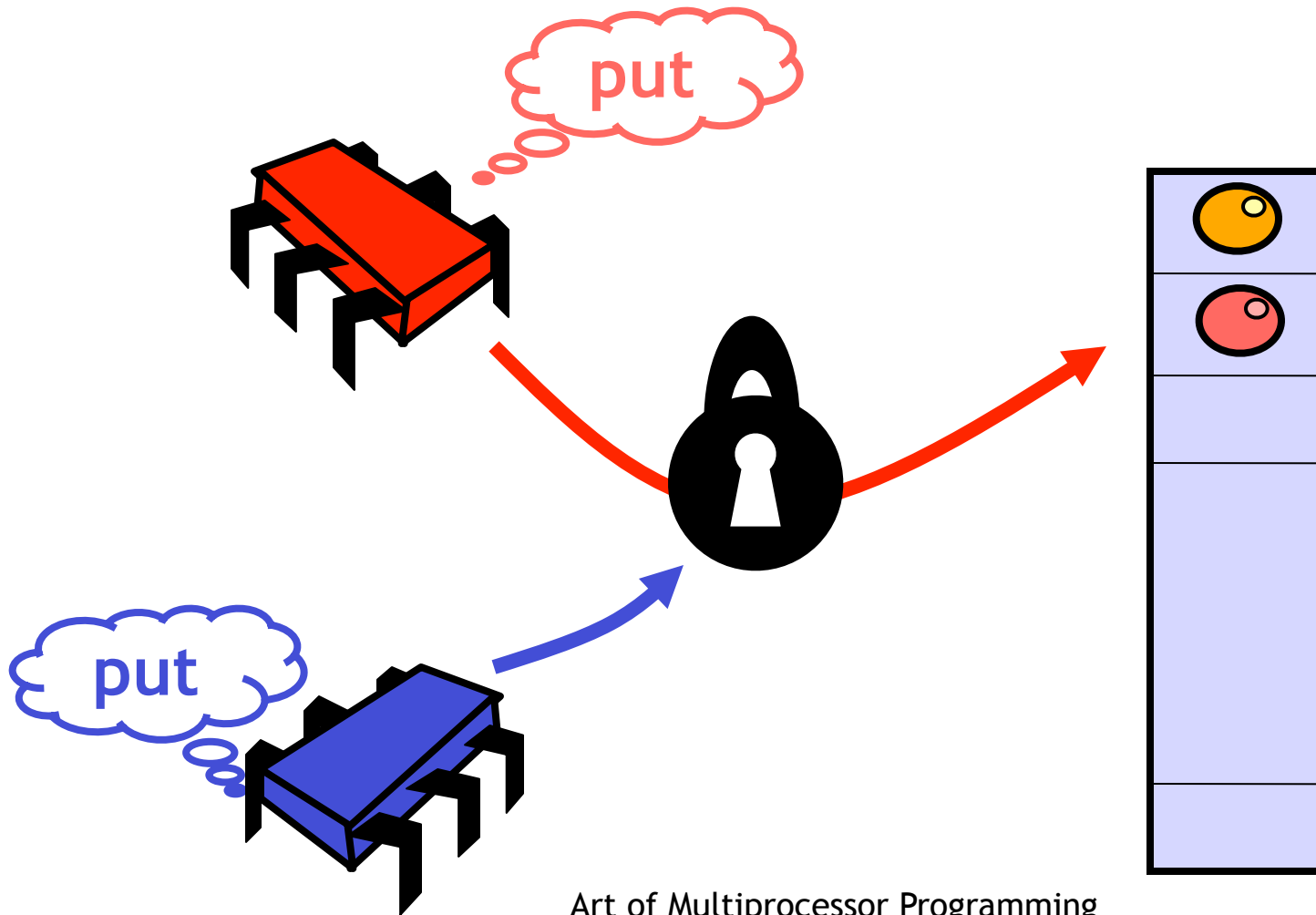
# A Shared Pool

```
public interface Pool {  
    public void put(Object x);  
    public Object remove();  
}
```

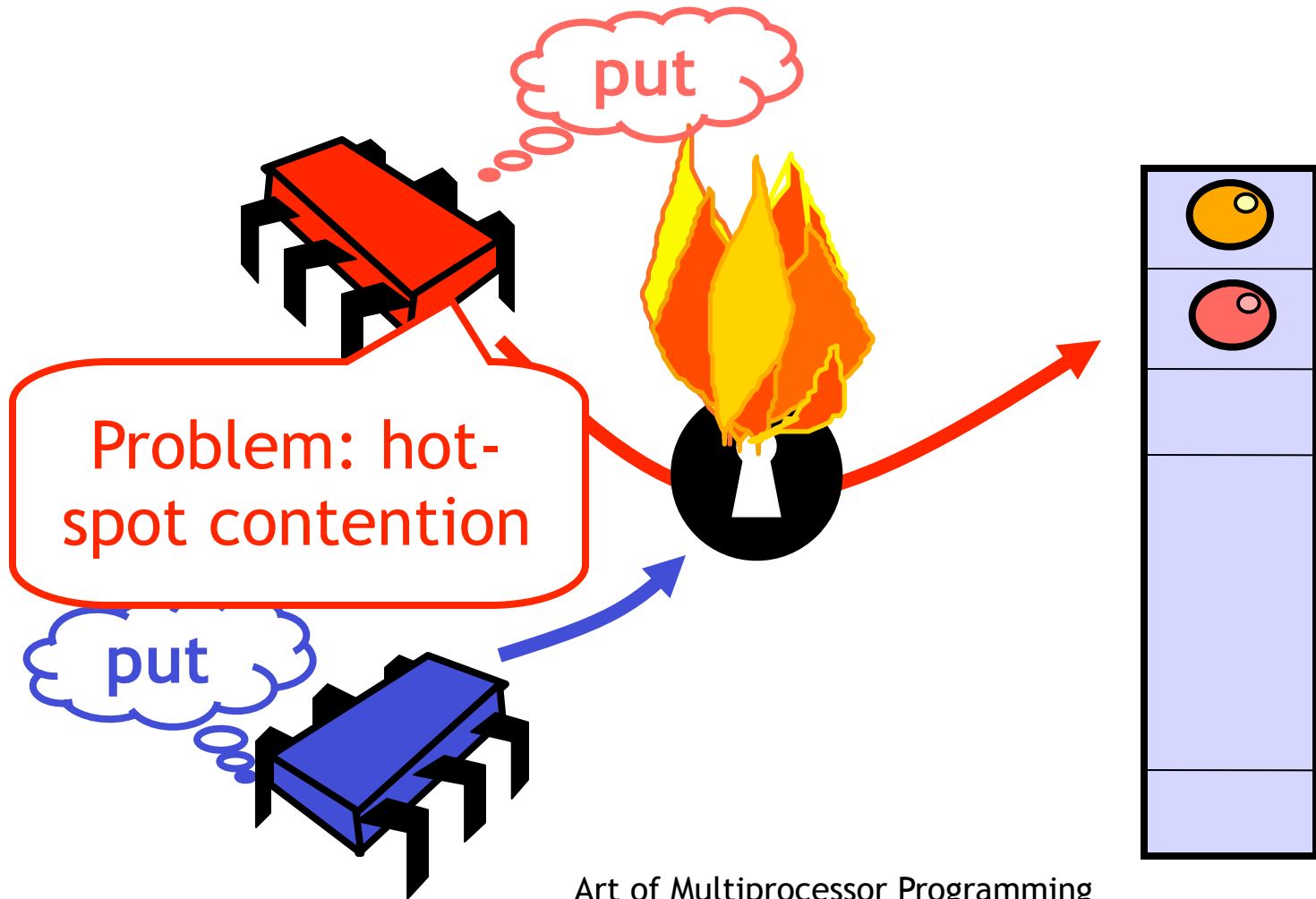
## Unordered set of objects

- Put
  - Inserts object
  - blocks if full
- Remove
  - Removes & returns an object
  - blocks if empty

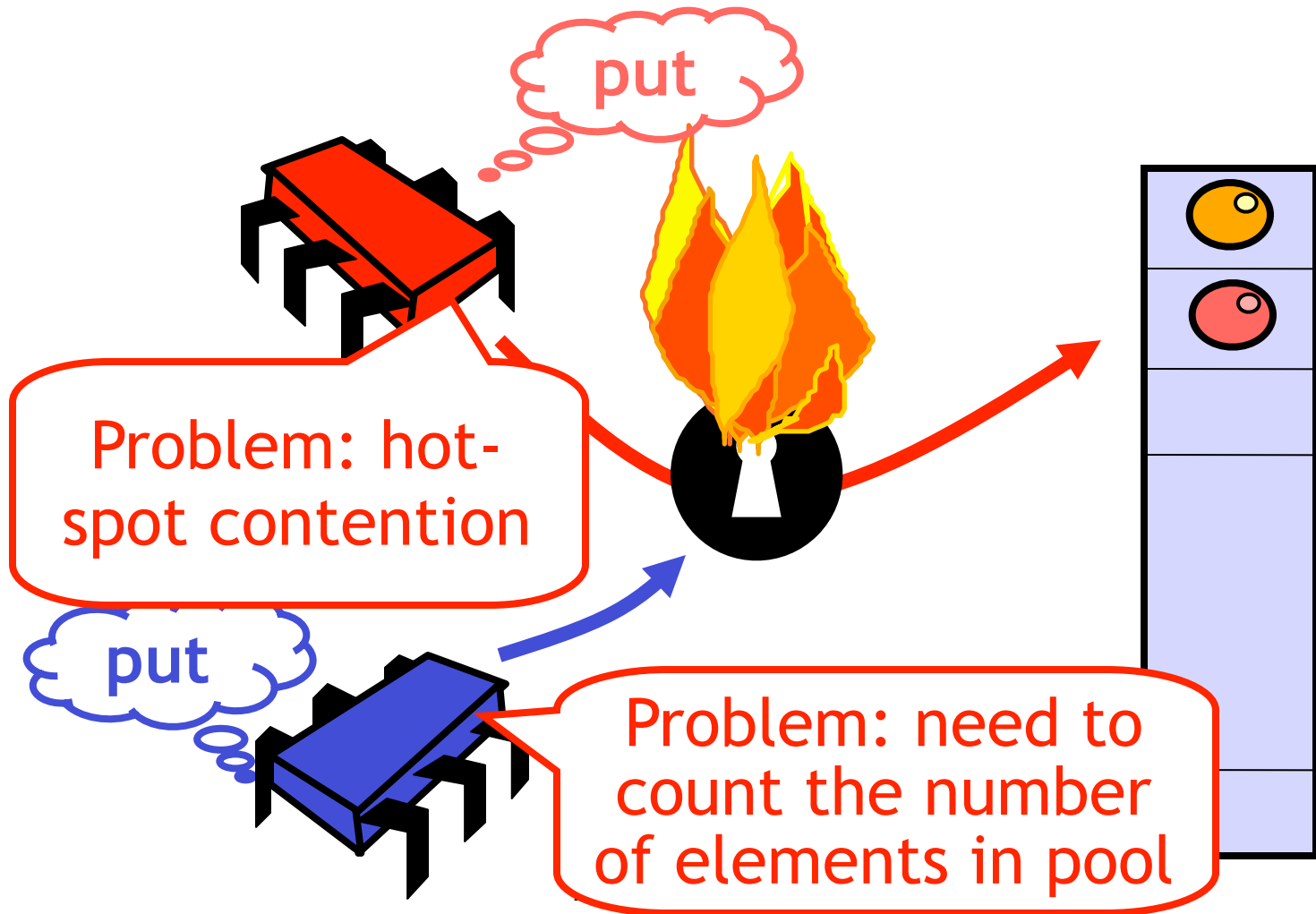
# Simple Locking Implementation



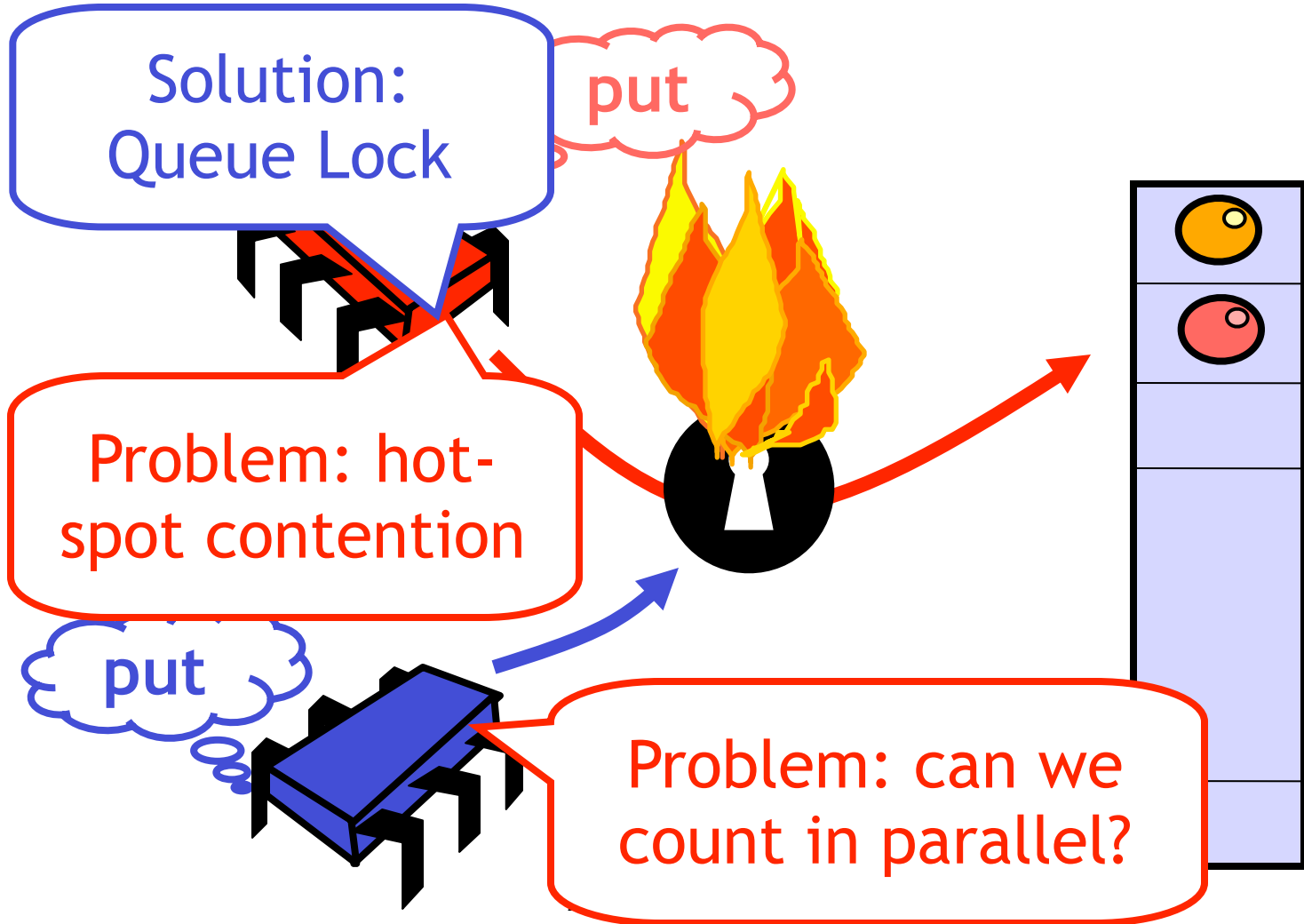
# Simple Locking Implementation



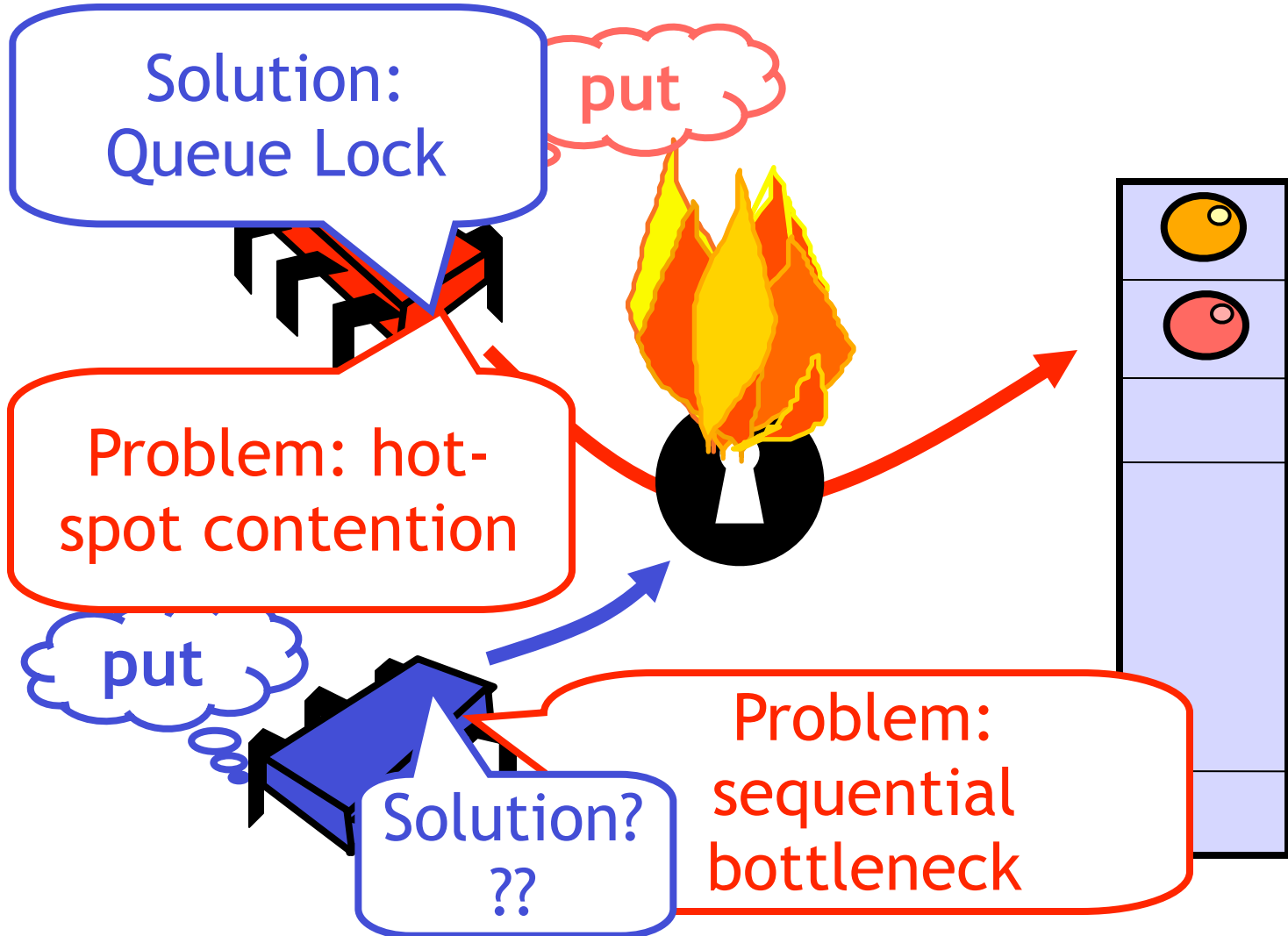
# Simple Locking Implementation



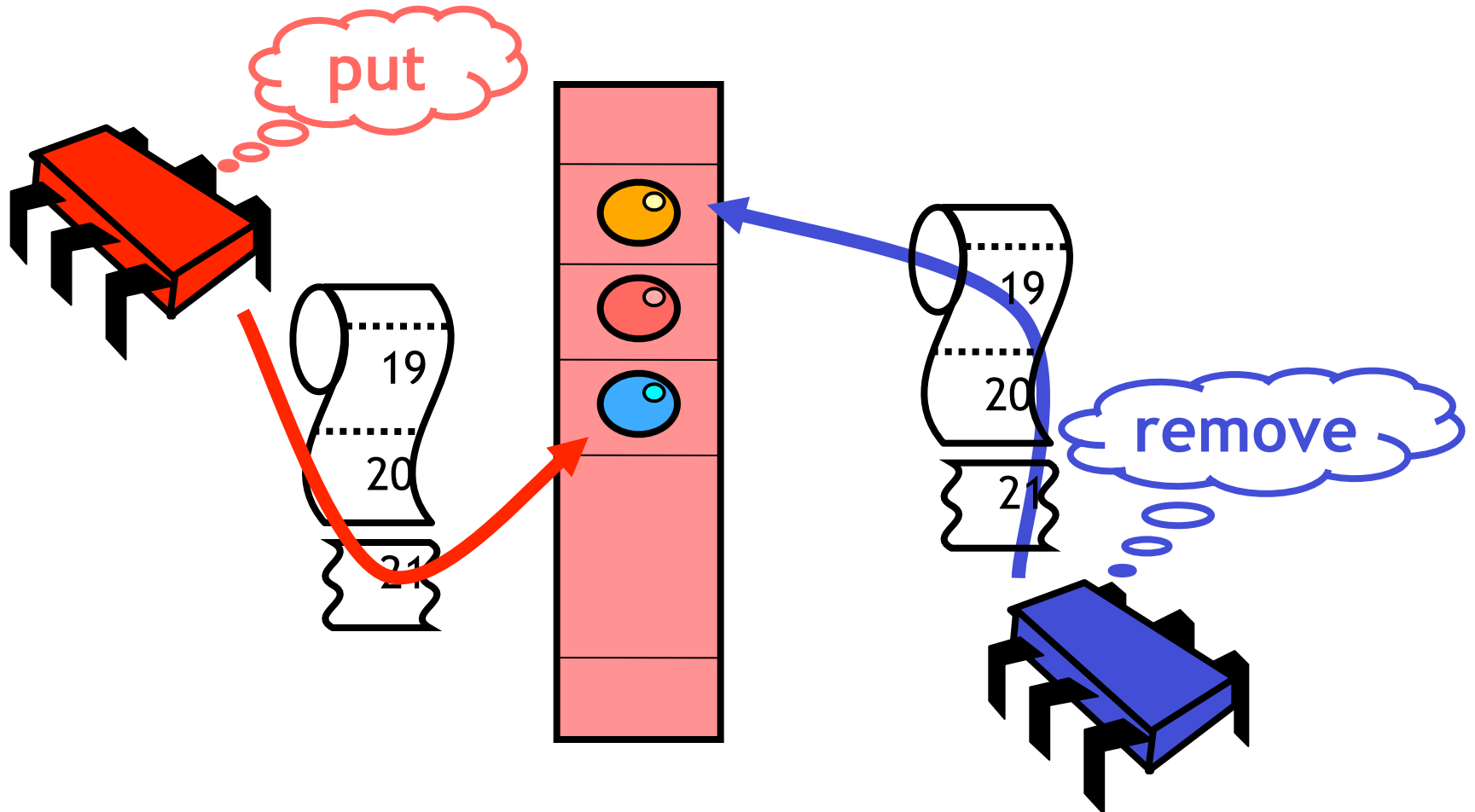
# Simple Locking Implementation



# Simple Locking Implementation

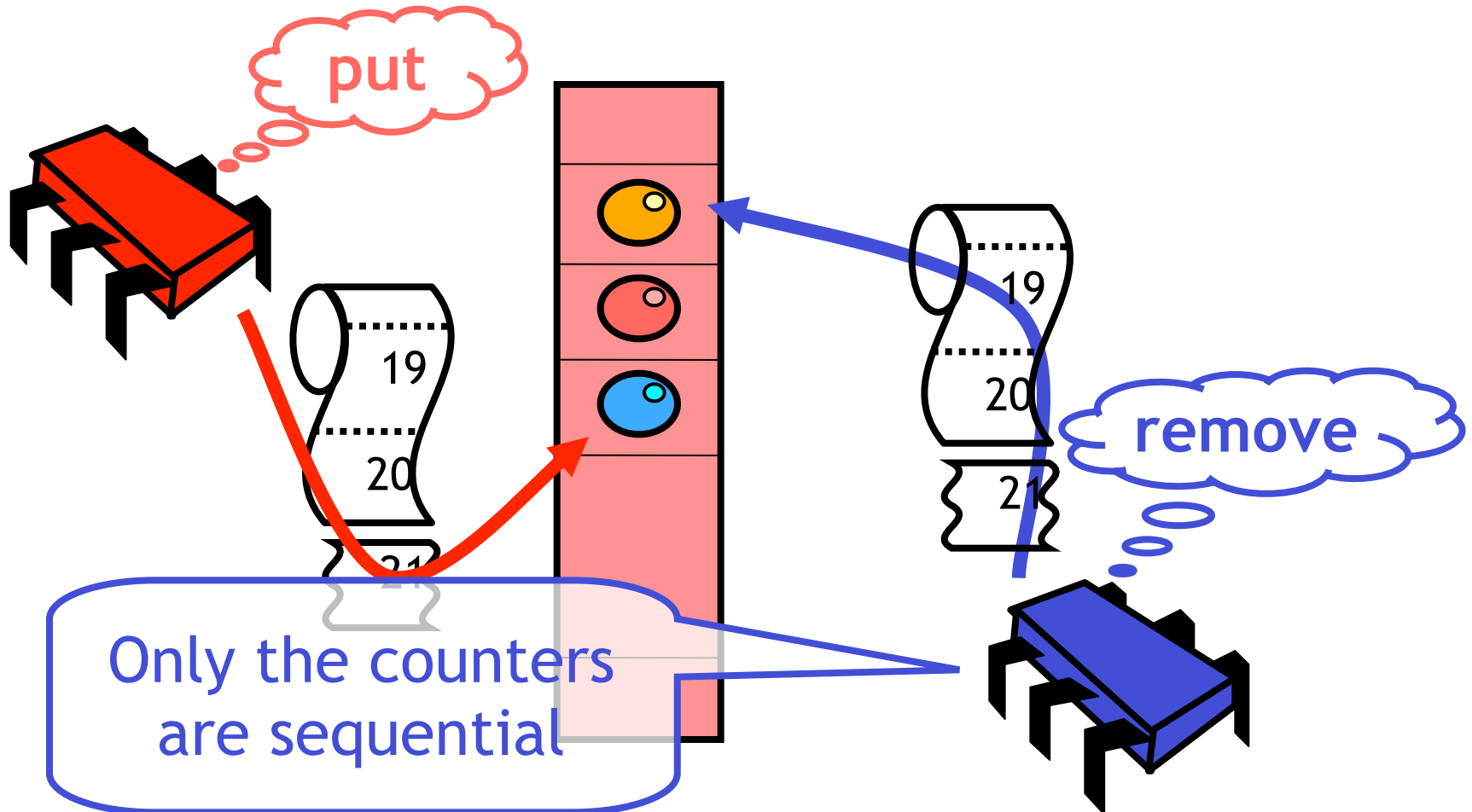


# Counting Implementation

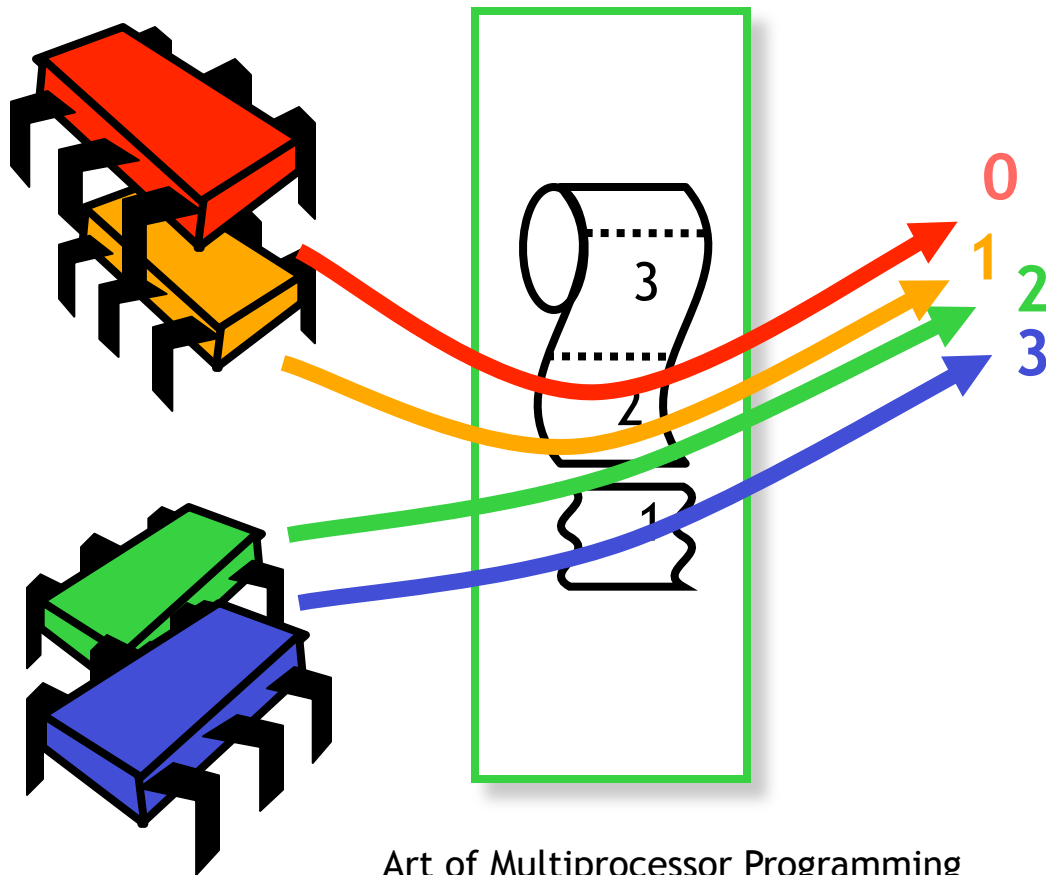




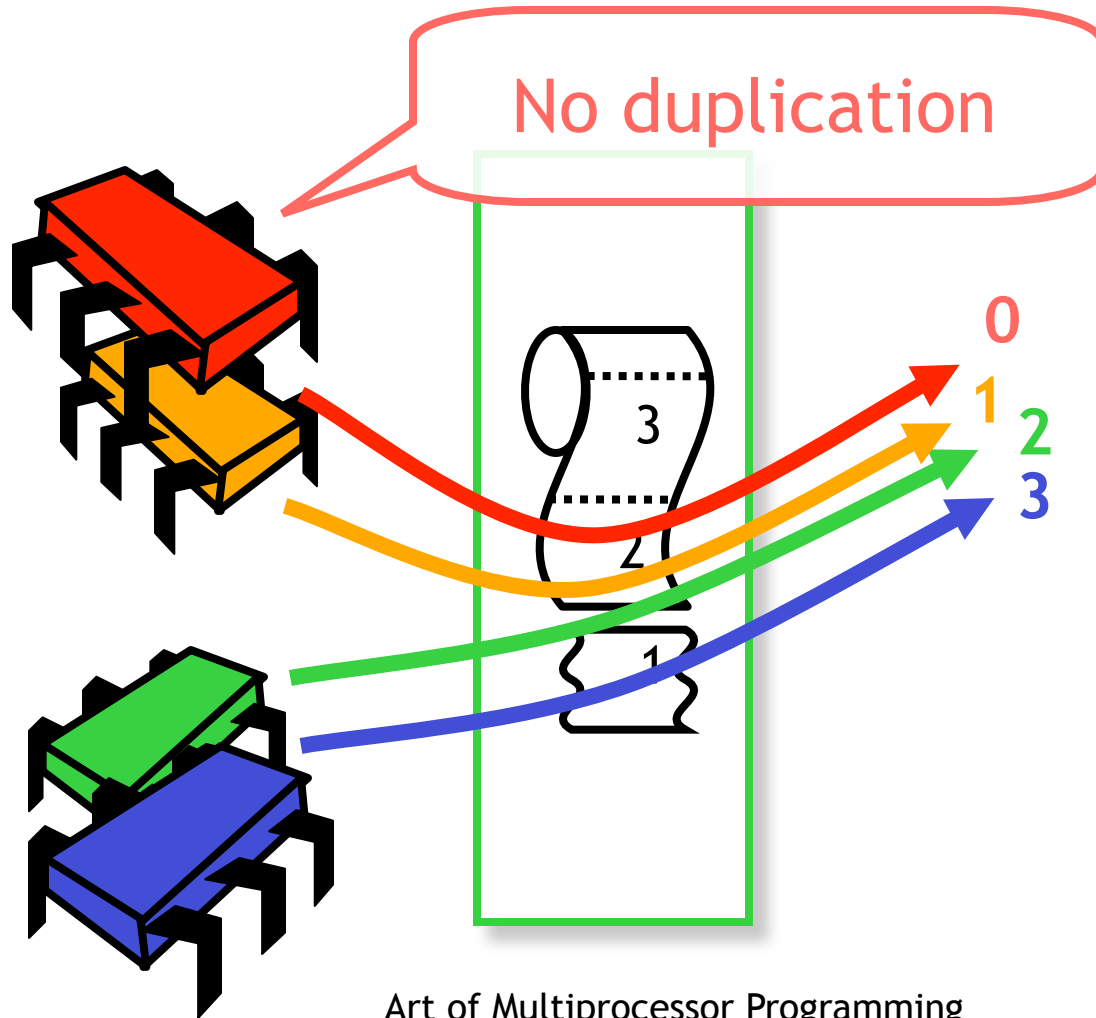
# Counting Implementation



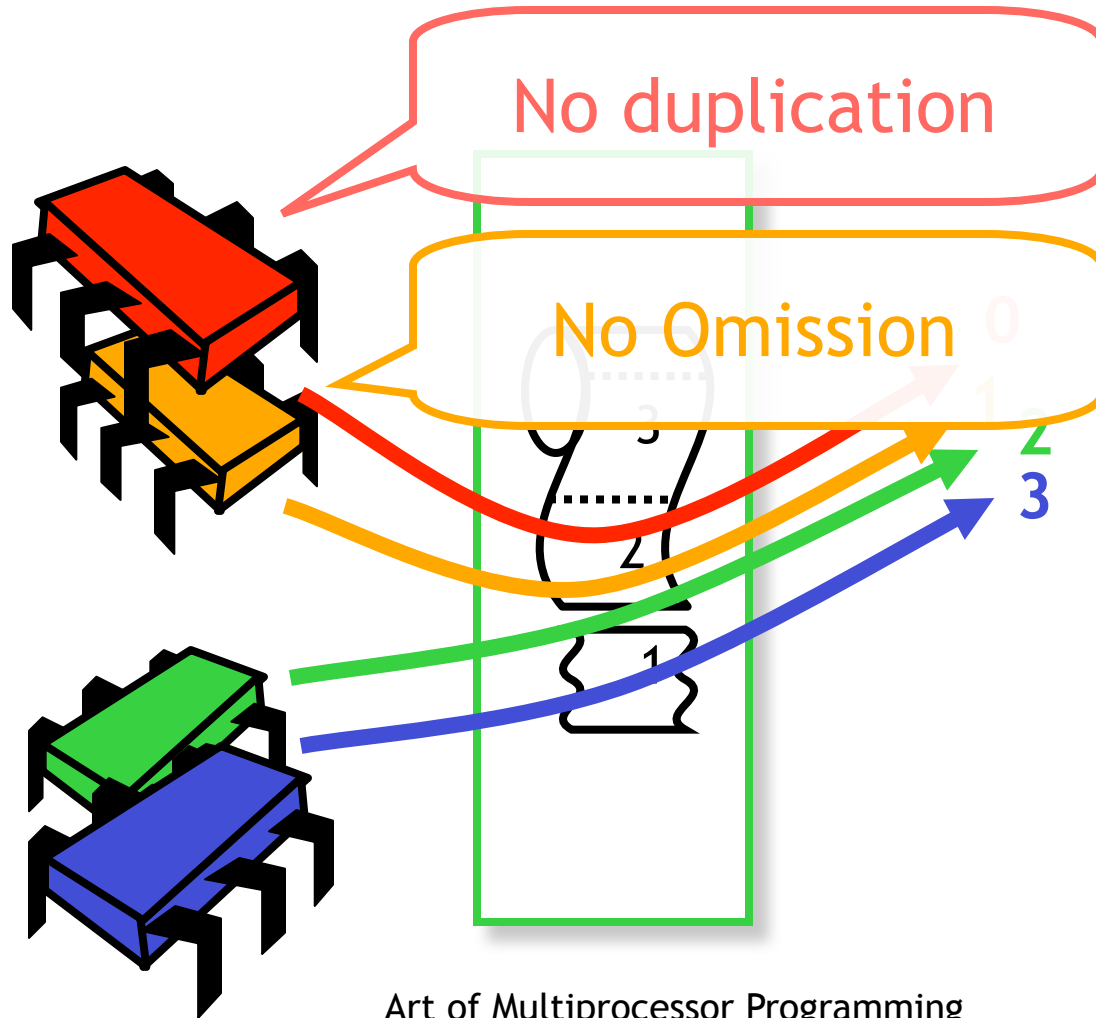
# Shared Counter



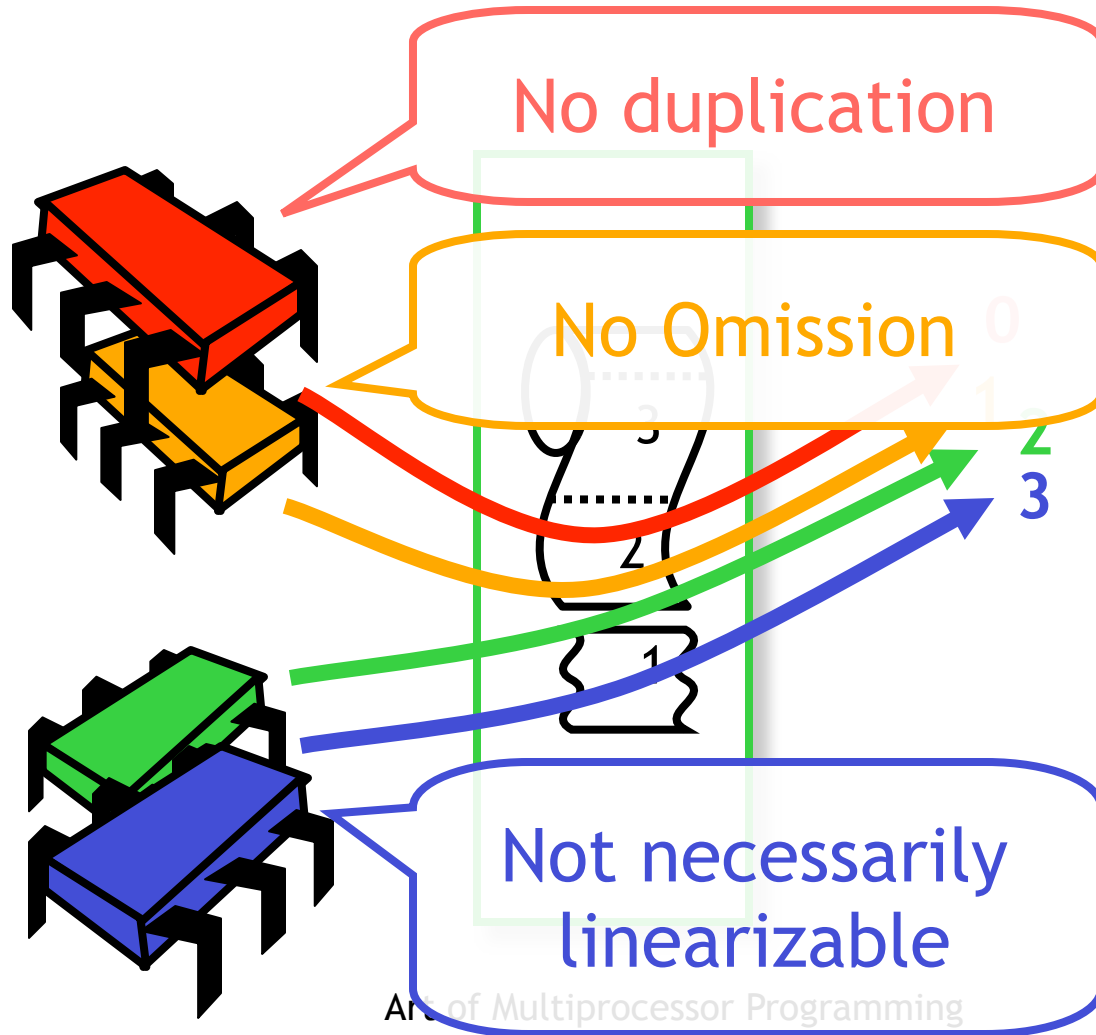
# Shared Counter



# Shared Counter



# Shared Counter



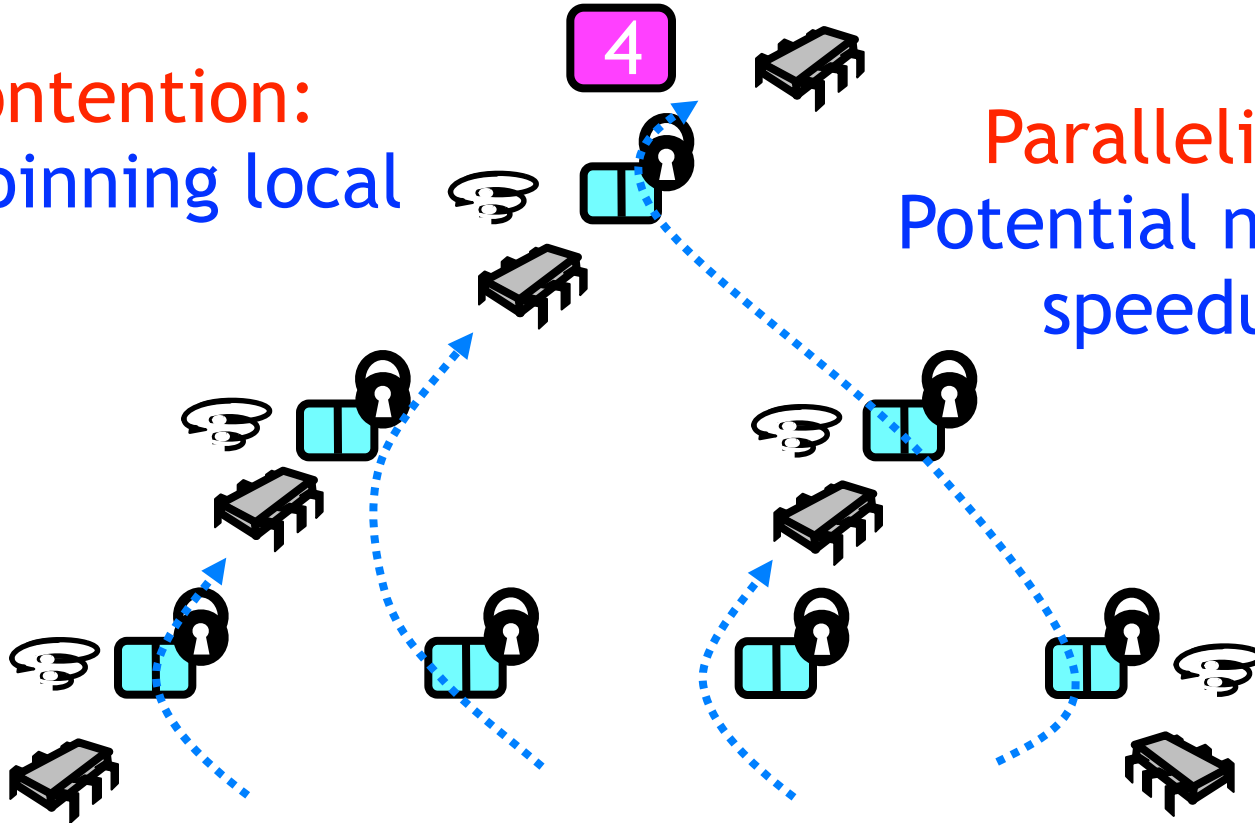
# Shared Counters

- Can we build a shared counter with
  - Low memory contention, and
  - Real parallelism?
- Locking
  - Can use queue locks to reduce contention
  - No help with parallelism issue ...

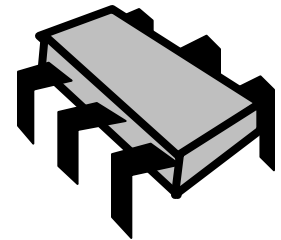
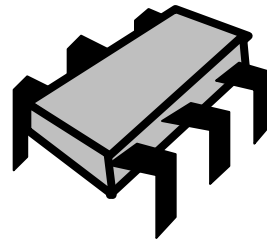
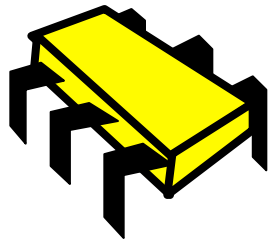
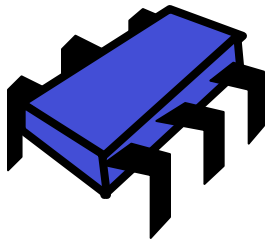
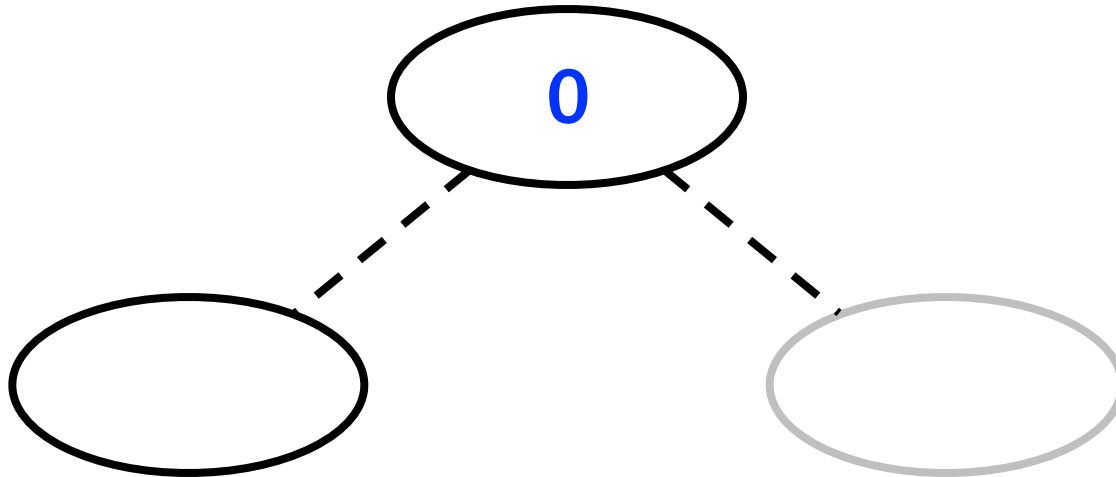
# Software Combining Tree

**Contention:**  
All spinning local

**Parallelism:**  
Potential  $n/\log n$   
speedup

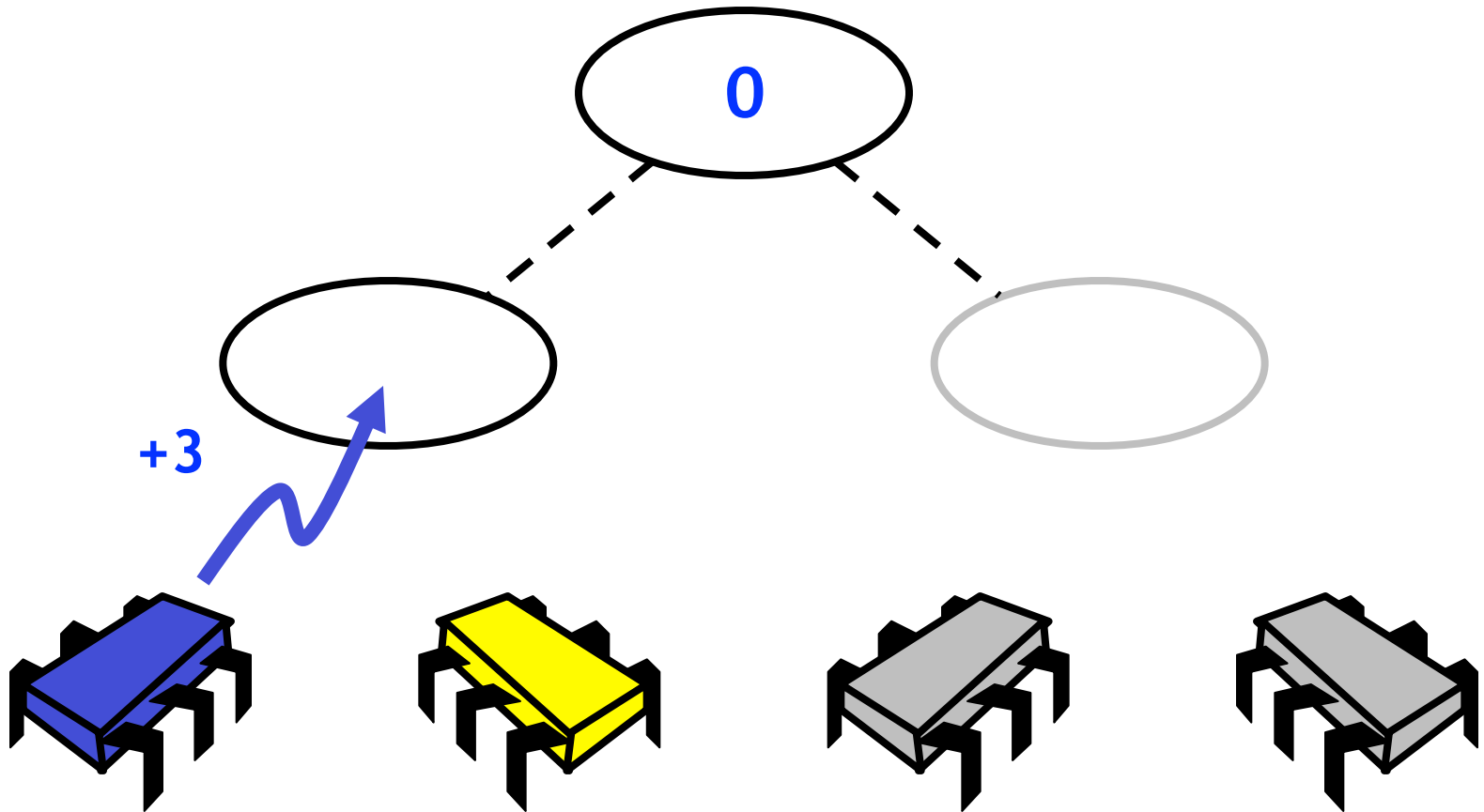


# Combining Trees

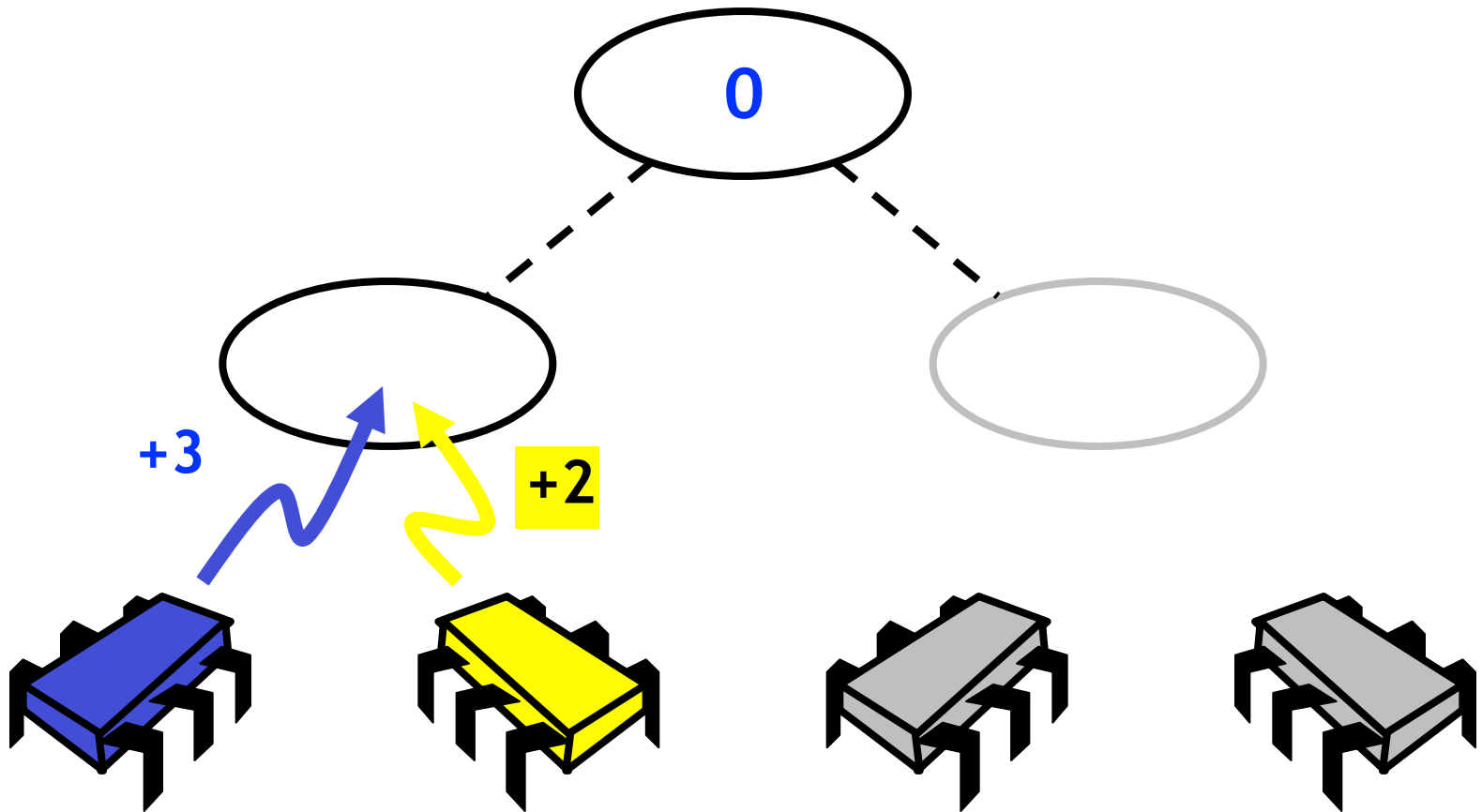




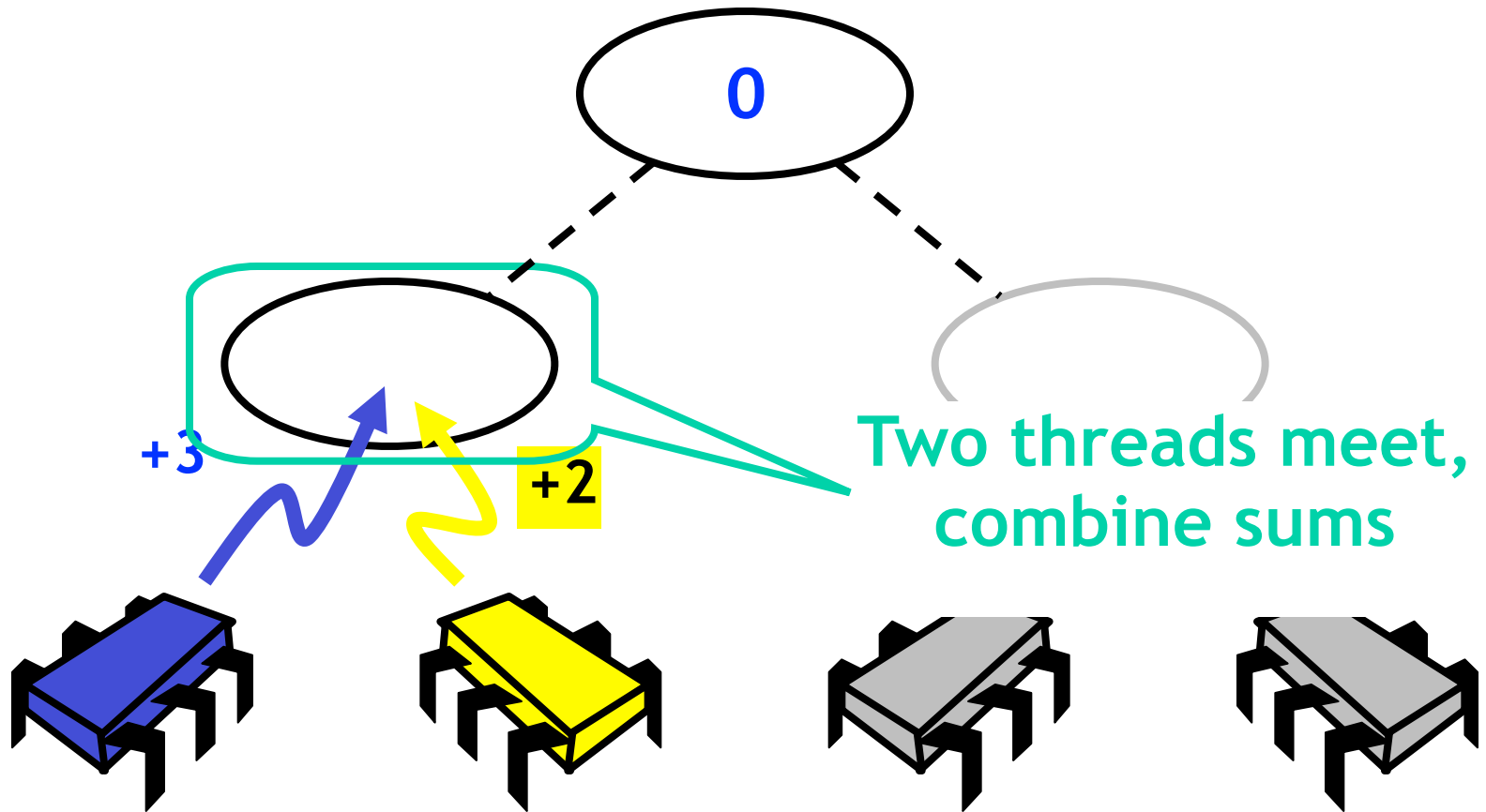
# Combining Trees



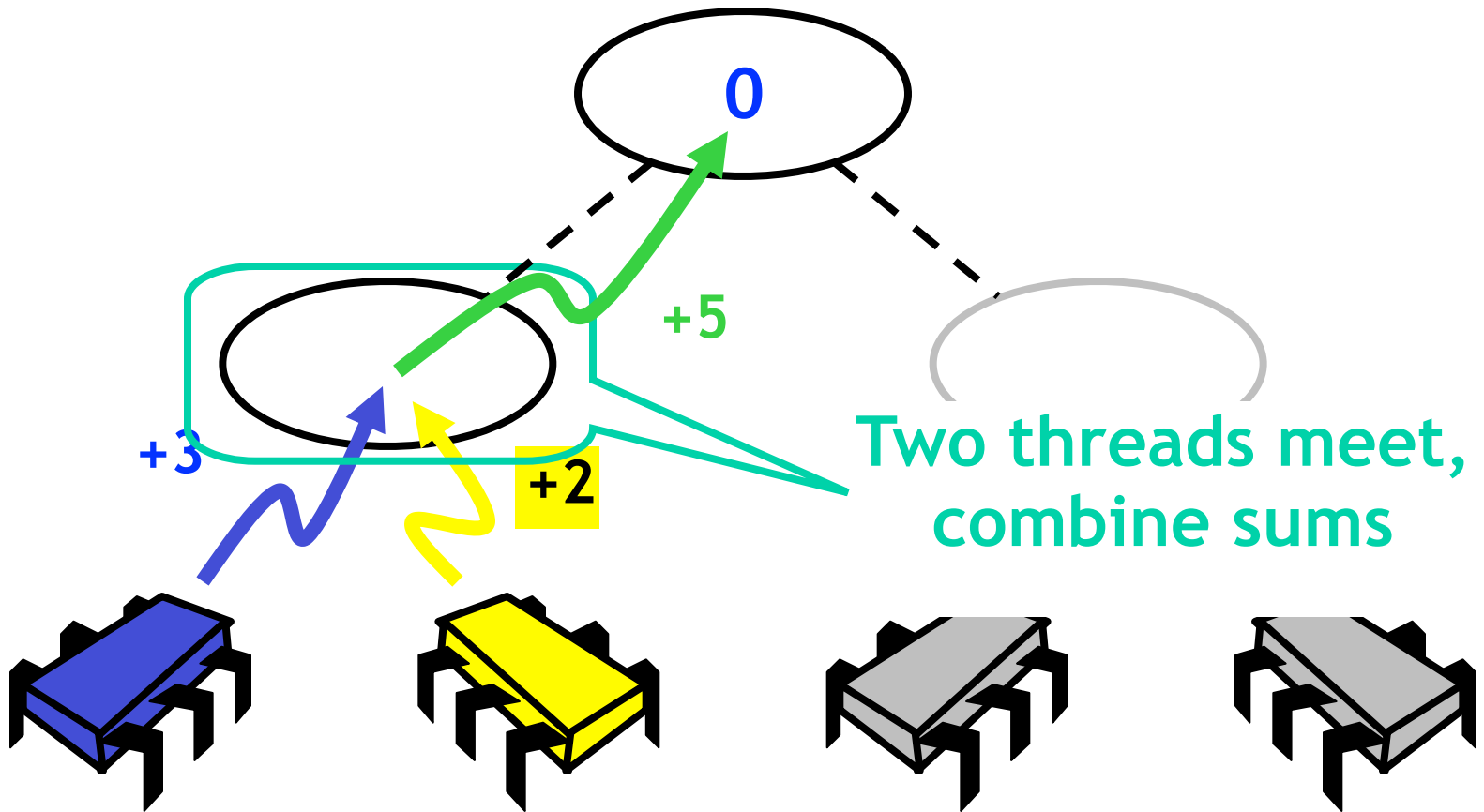
# Combining Trees



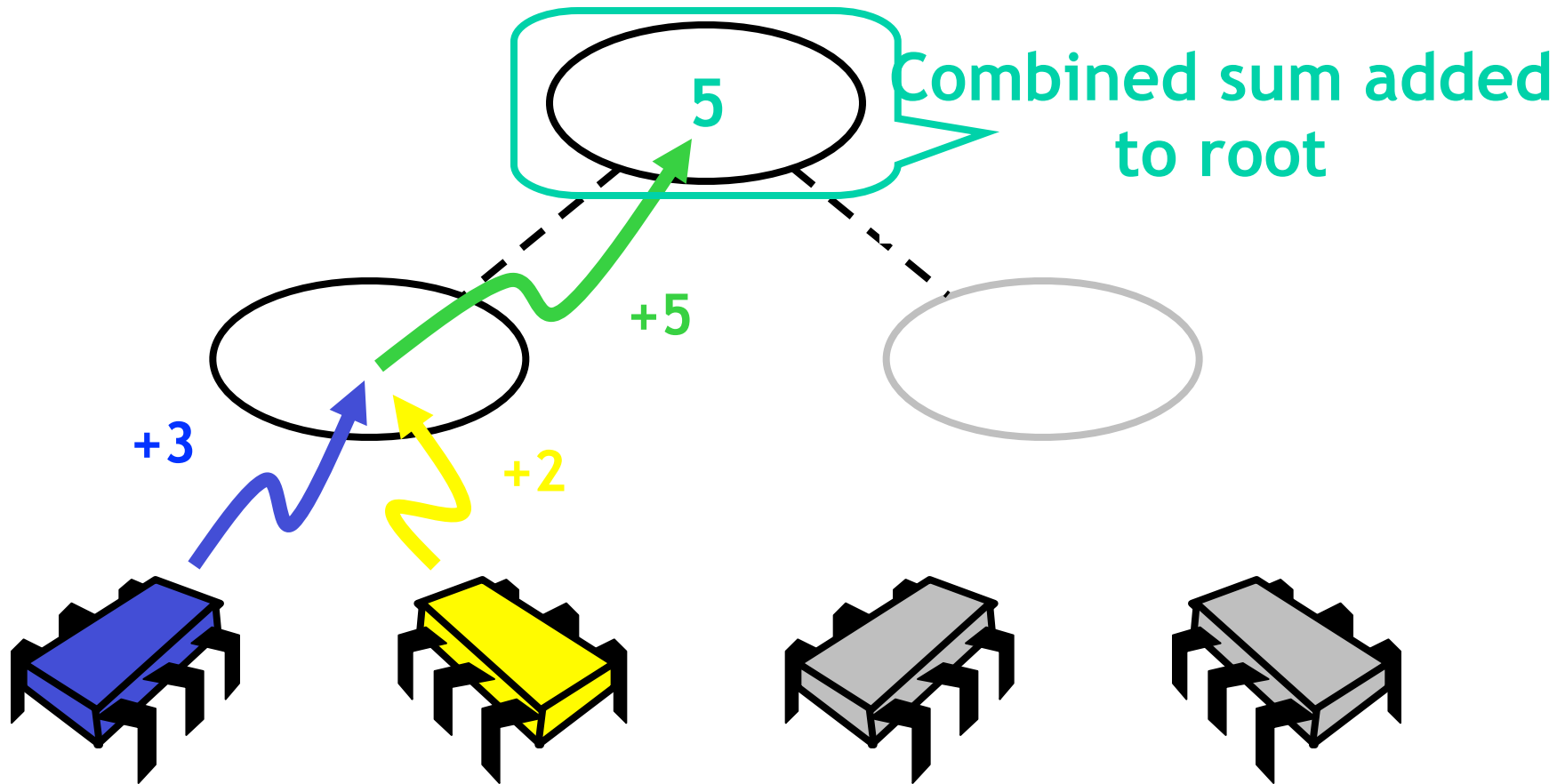
# Combining Trees



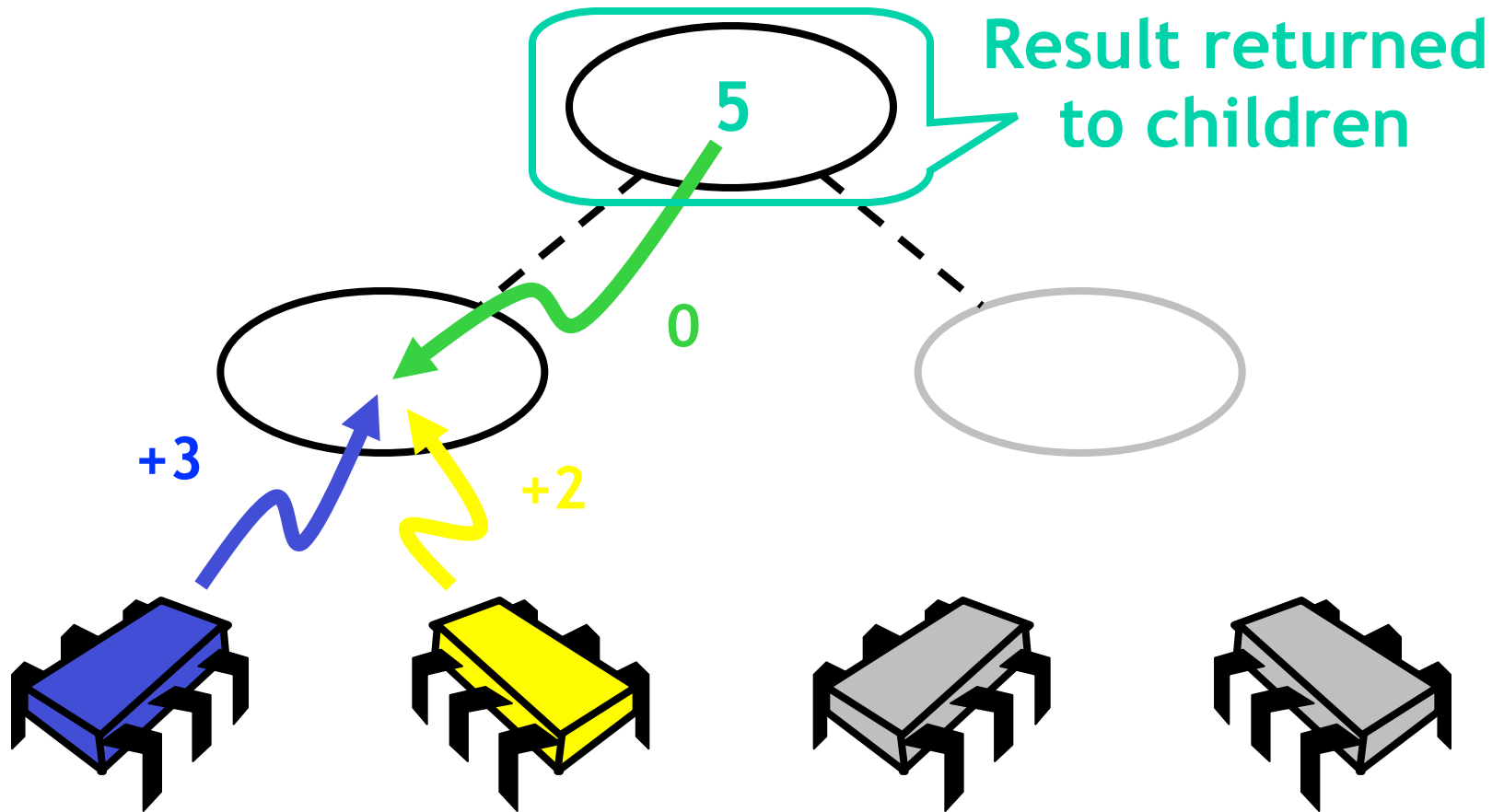
# Combining Trees



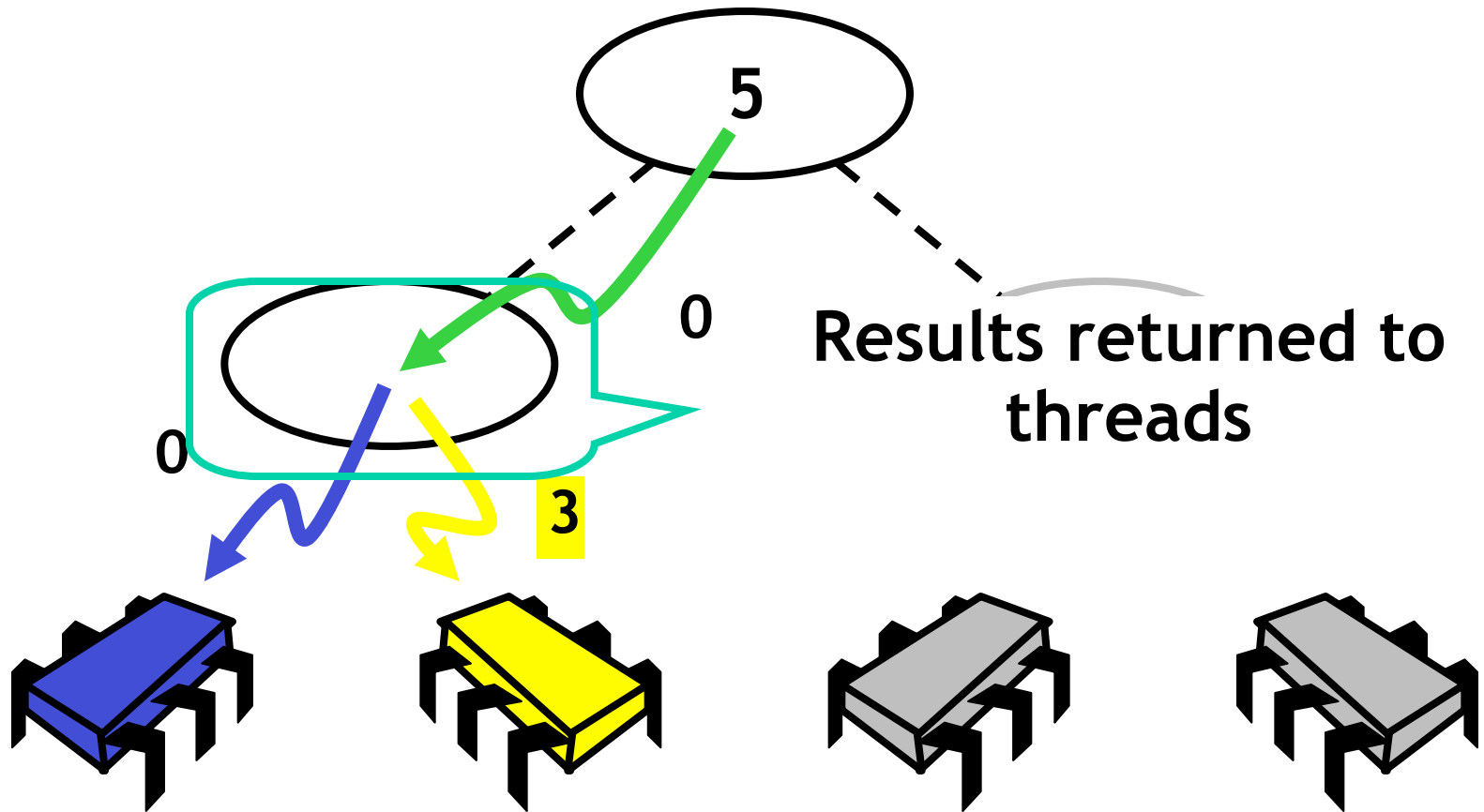
# Combining Trees



# Combining Trees



# Combining Trees



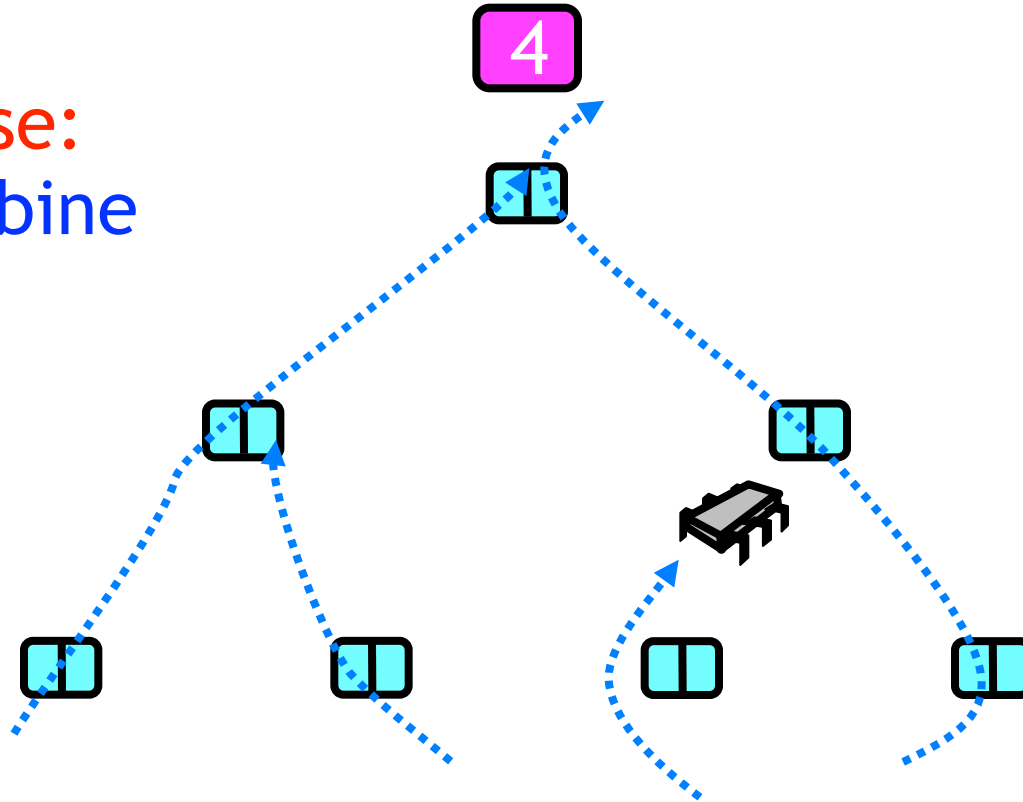
# Devil in the Details

- What if
  - threads don't arrive at the same time?
- Wait for a partner to show up?
  - How long to wait?
  - Waiting times add up ...
- Instead
  - Use multi-phase algorithm
  - Try to wait in parallel ...



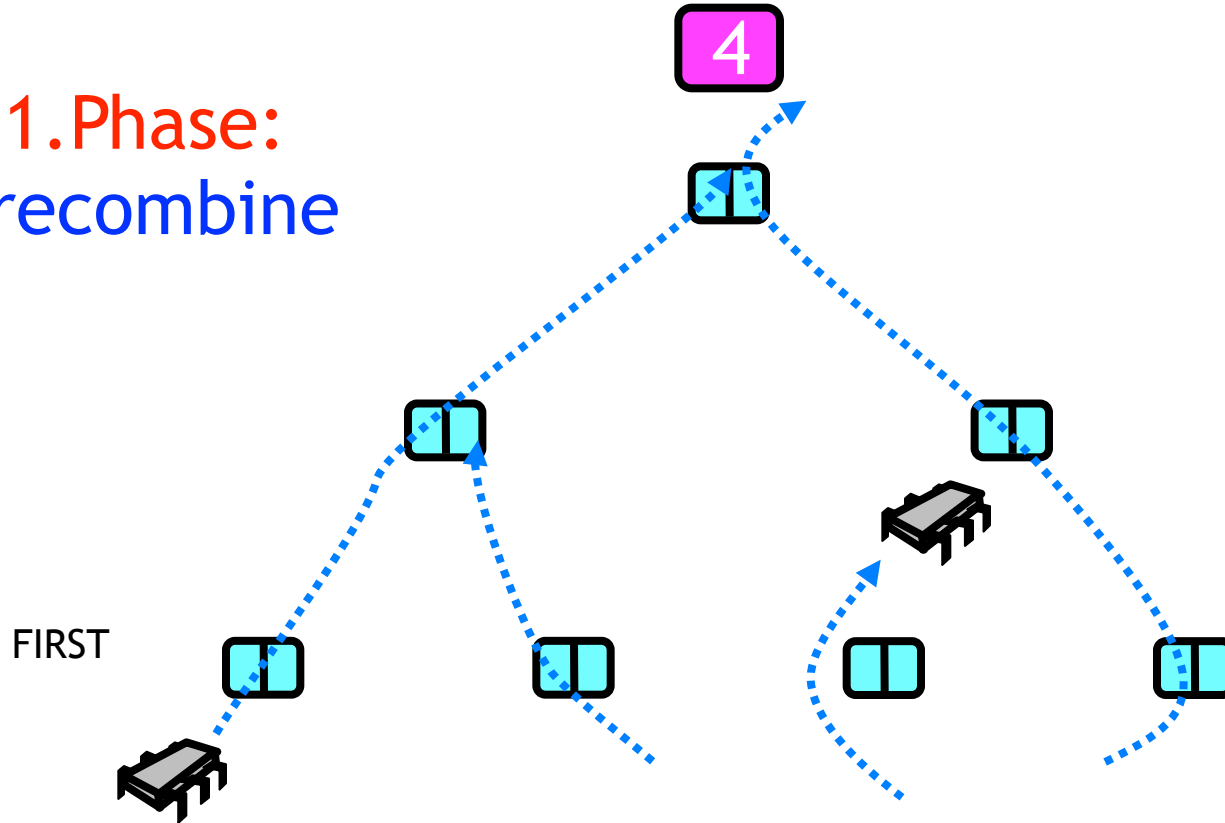
# Software Combining Tree

1.Phase:  
Precombine



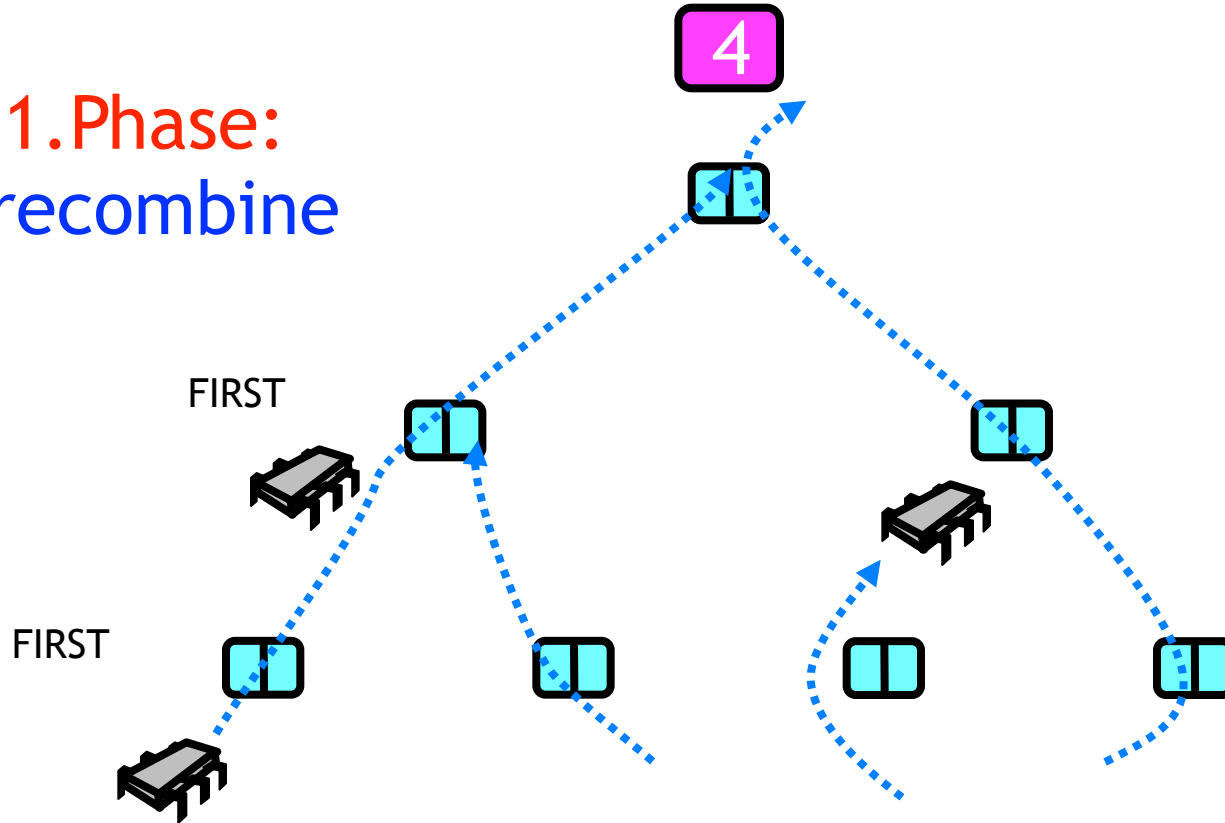
# Software Combining Tree

1.Phase:  
Precombine



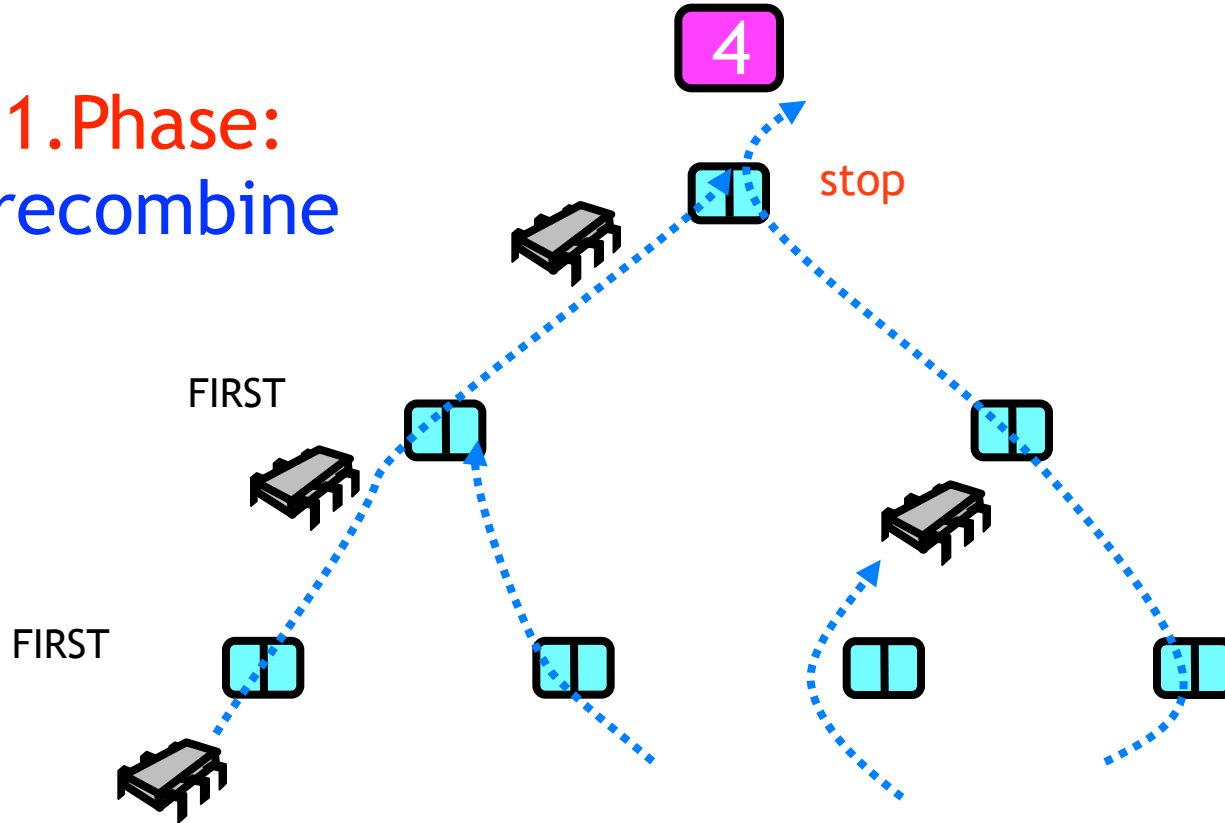
# Software Combining Tree

1.Phase:  
Precombine



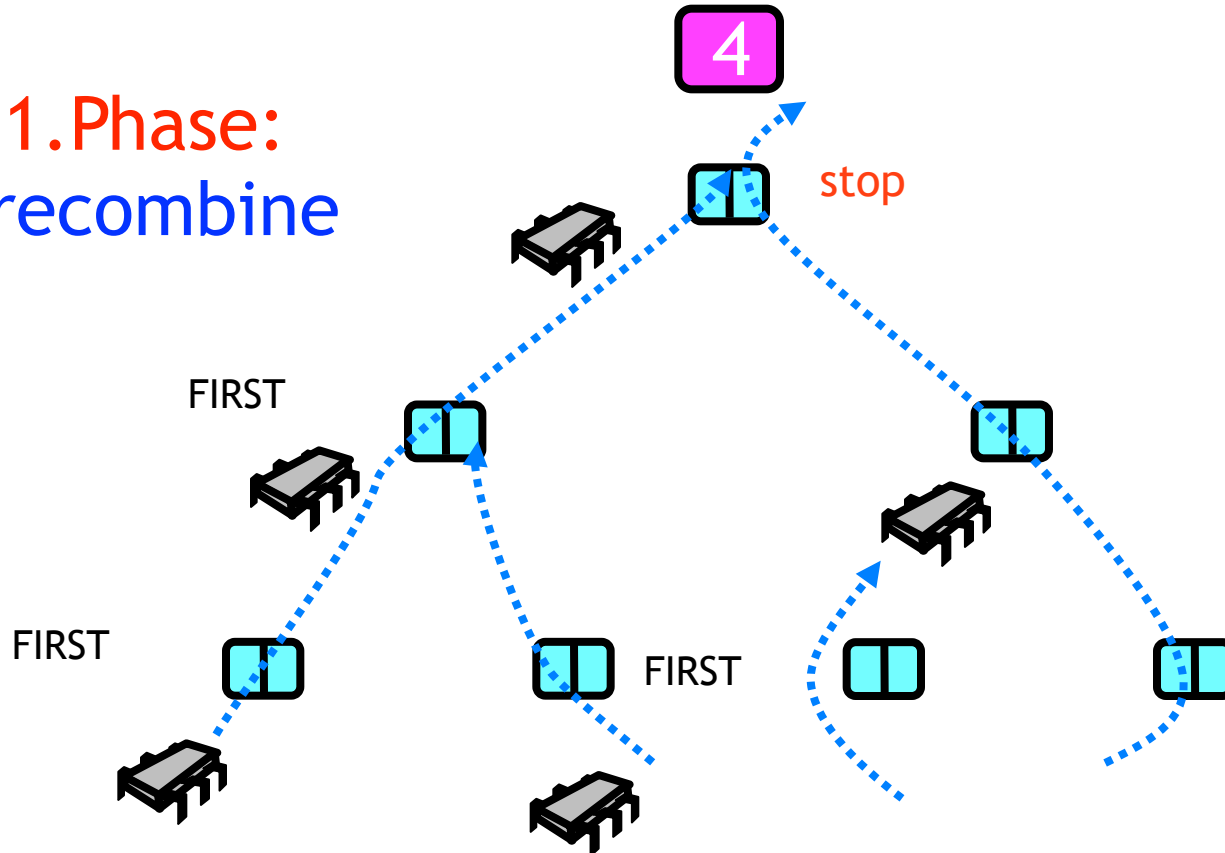
# Software Combining Tree

1.Phase:  
Precombine



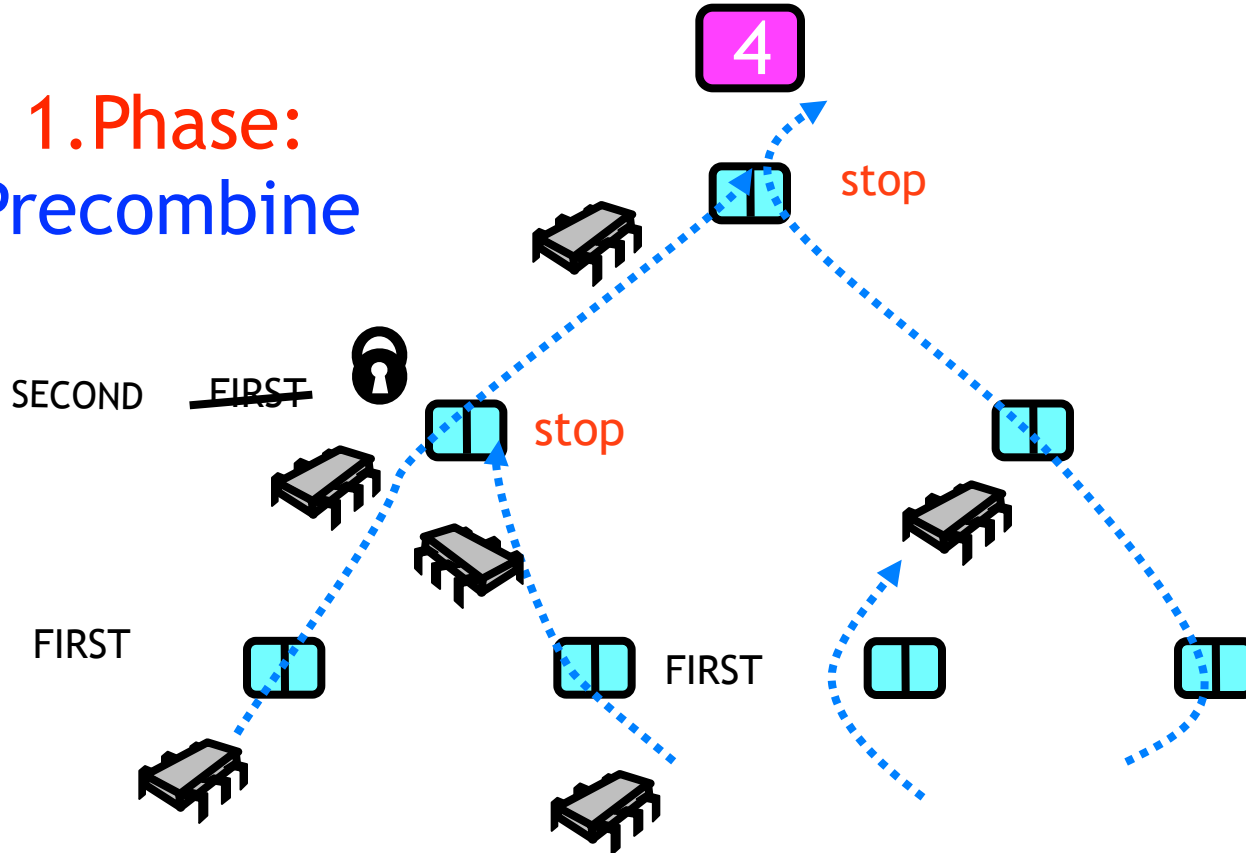
# Software Combining Tree

1.Phase:  
Precombine



# Software Combining Tree

1.Phase:  
Precombine

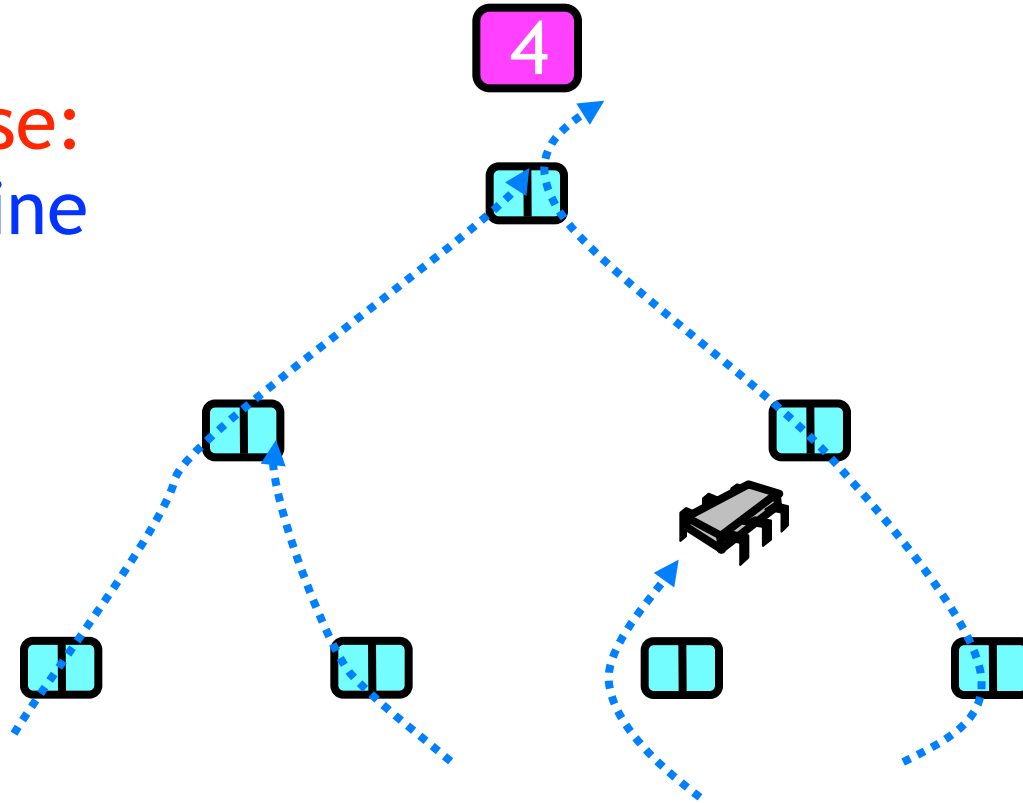


# Phases

- Precombining
  - Set up combining rendezvous
- Combining
  - Collect and combine operations
- Operation
  - Hand off to higher thread
- Distribution
  - Distribute results to waiting threads

# Software Combining Tree

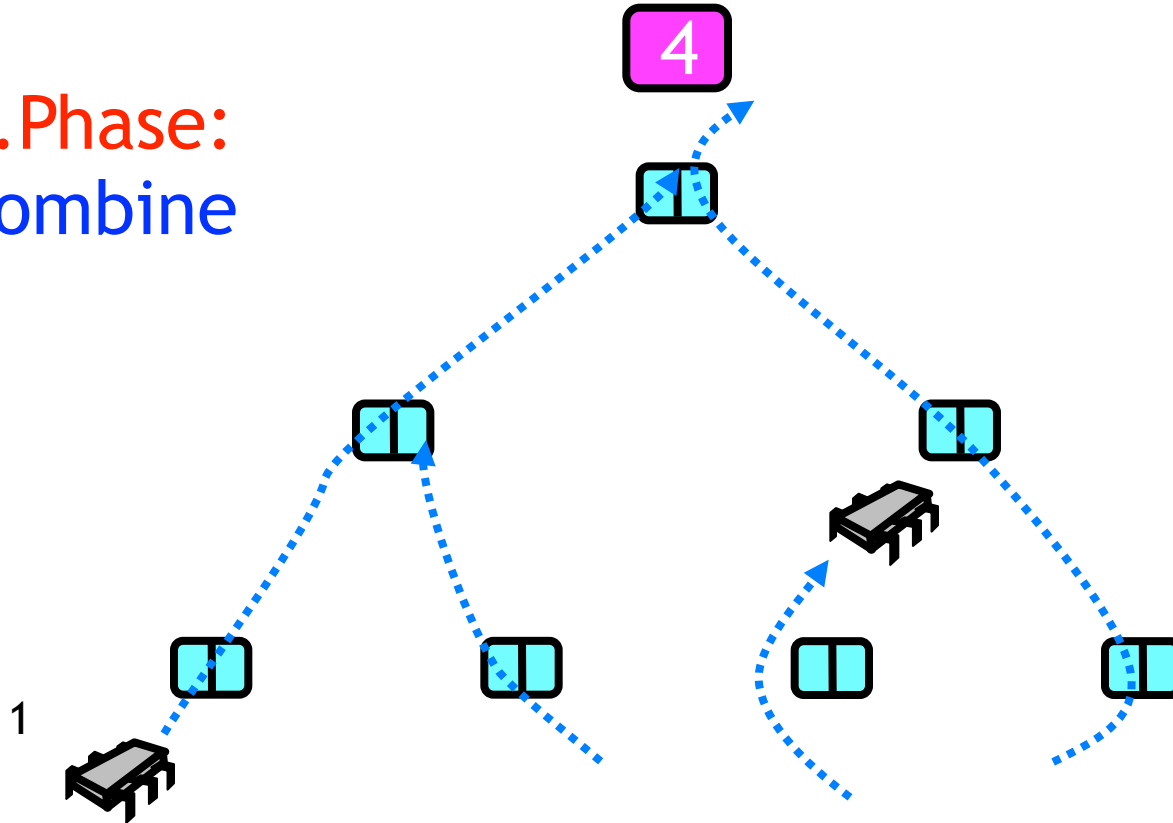
2.Phase:  
Combine





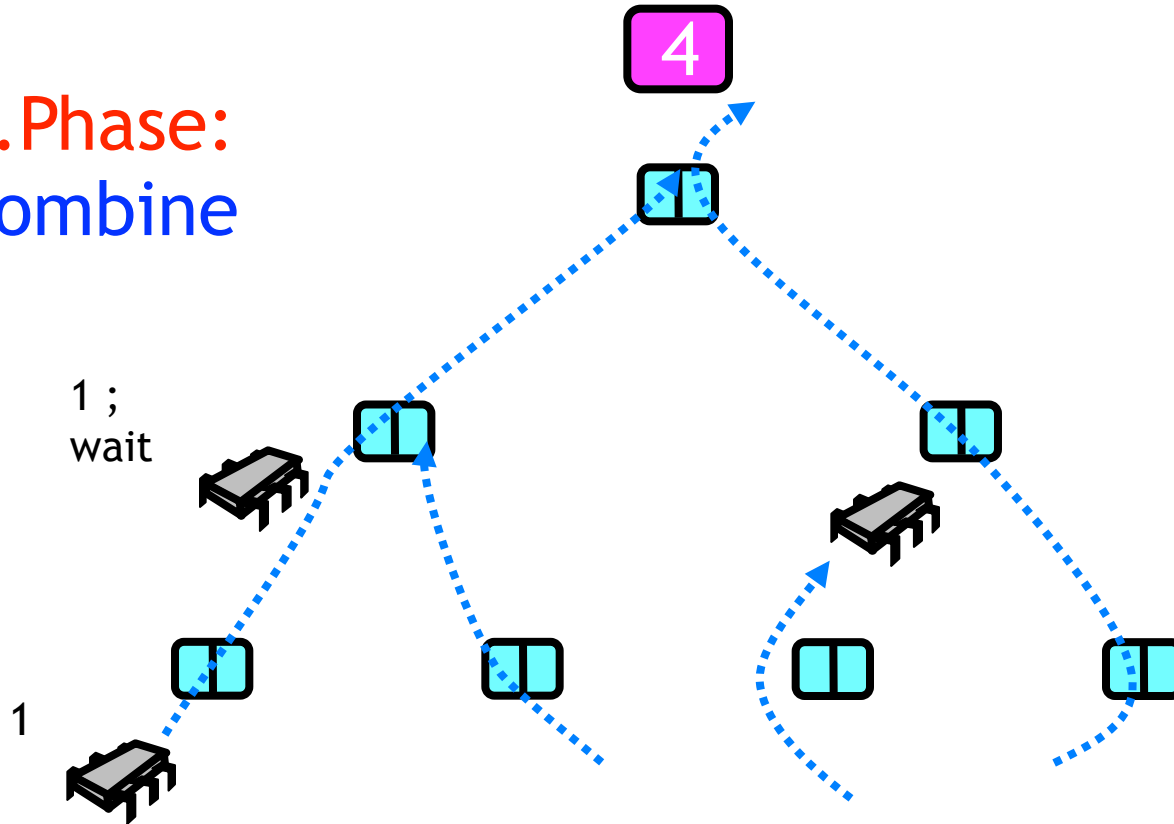
# Software Combining Tree

2.Phase:  
Combine



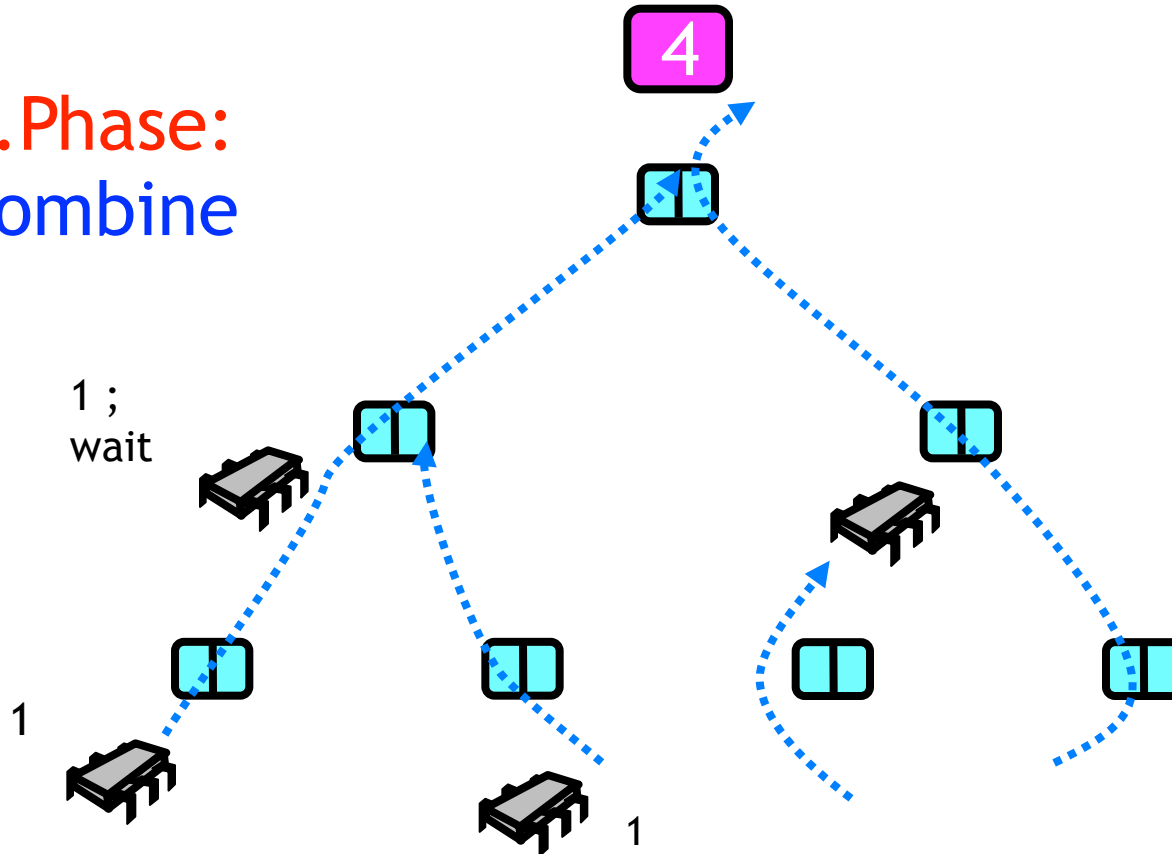
# Software Combining Tree

2.Phase:  
Combine



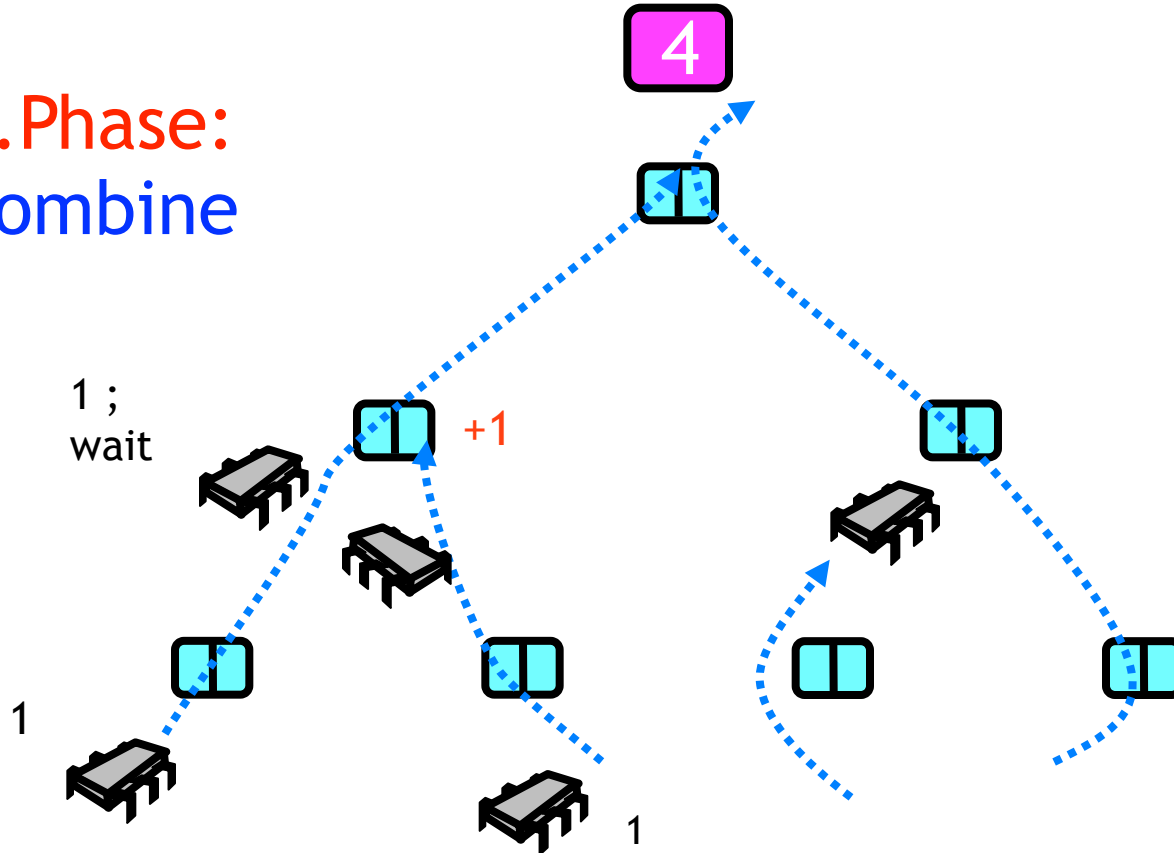
# Software Combining Tree

2.Phase:  
Combine



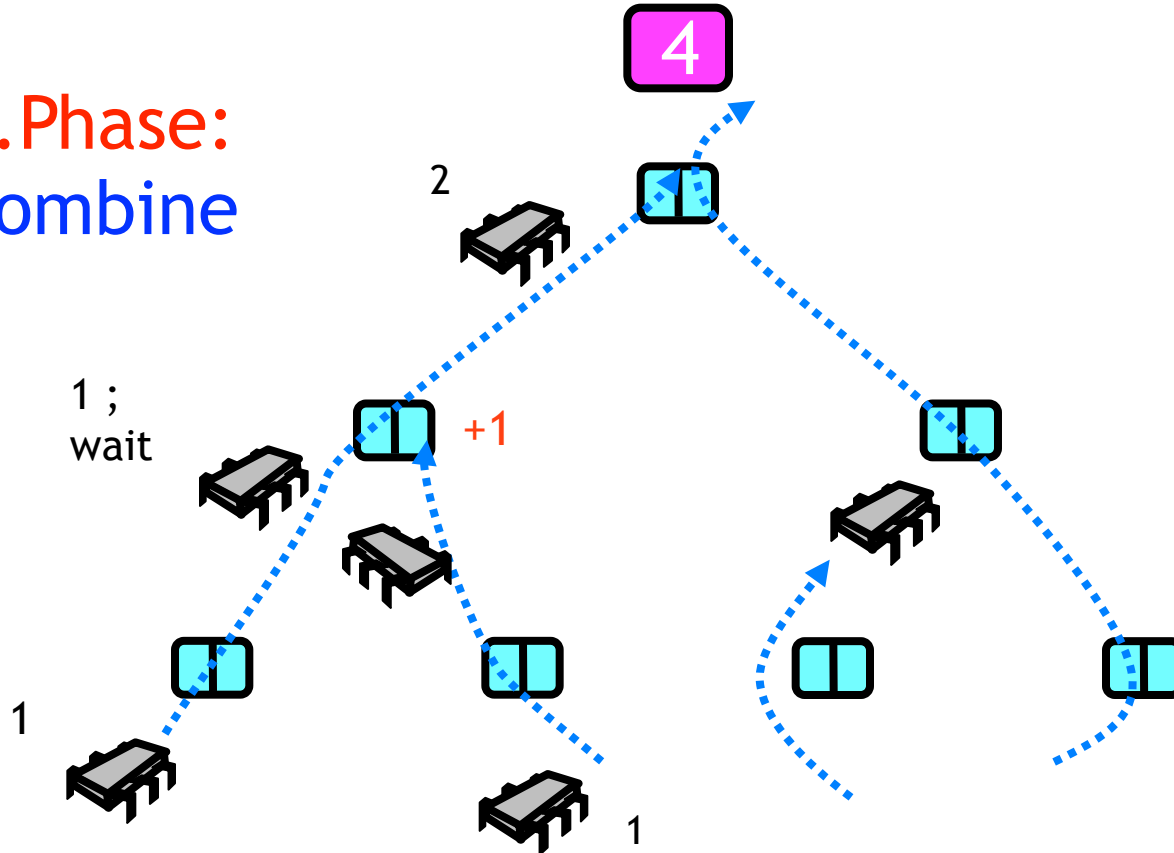
# Software Combining Tree

2.Phase:  
Combine



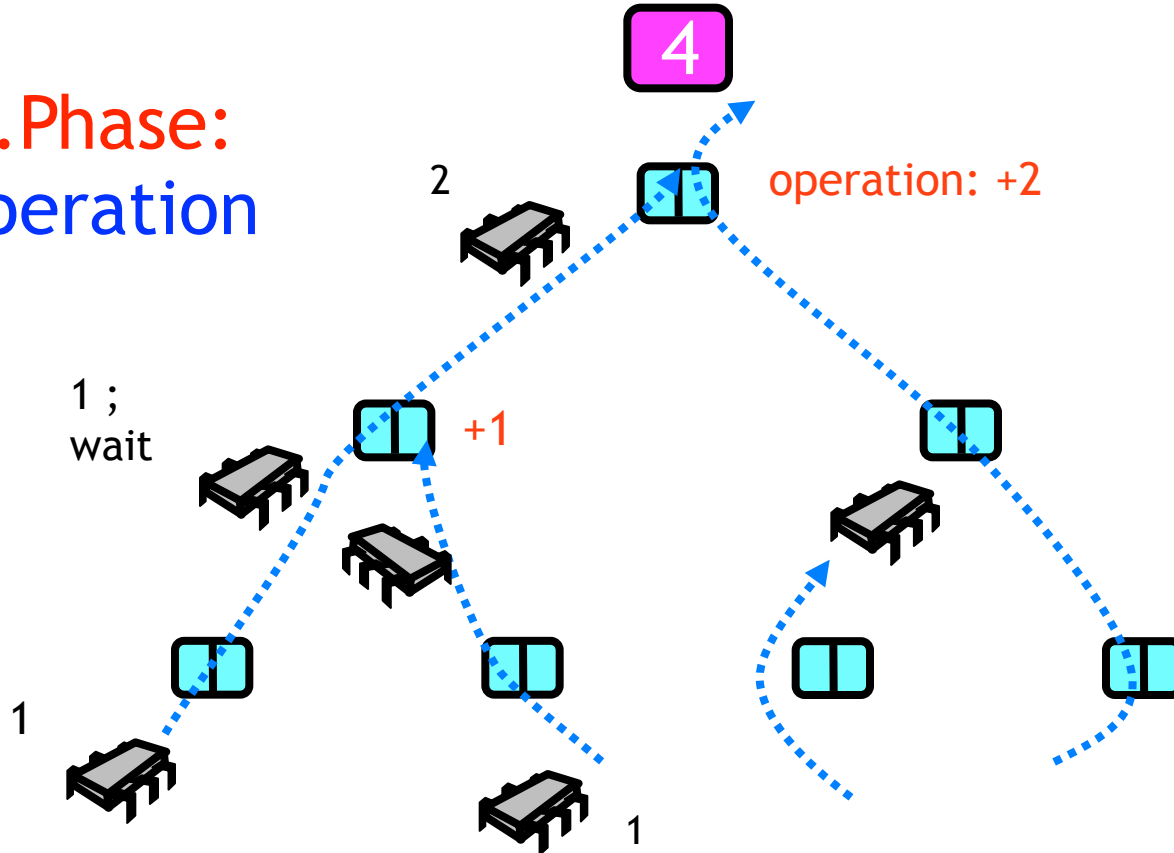
# Software Combining Tree

2.Phase:  
Combine



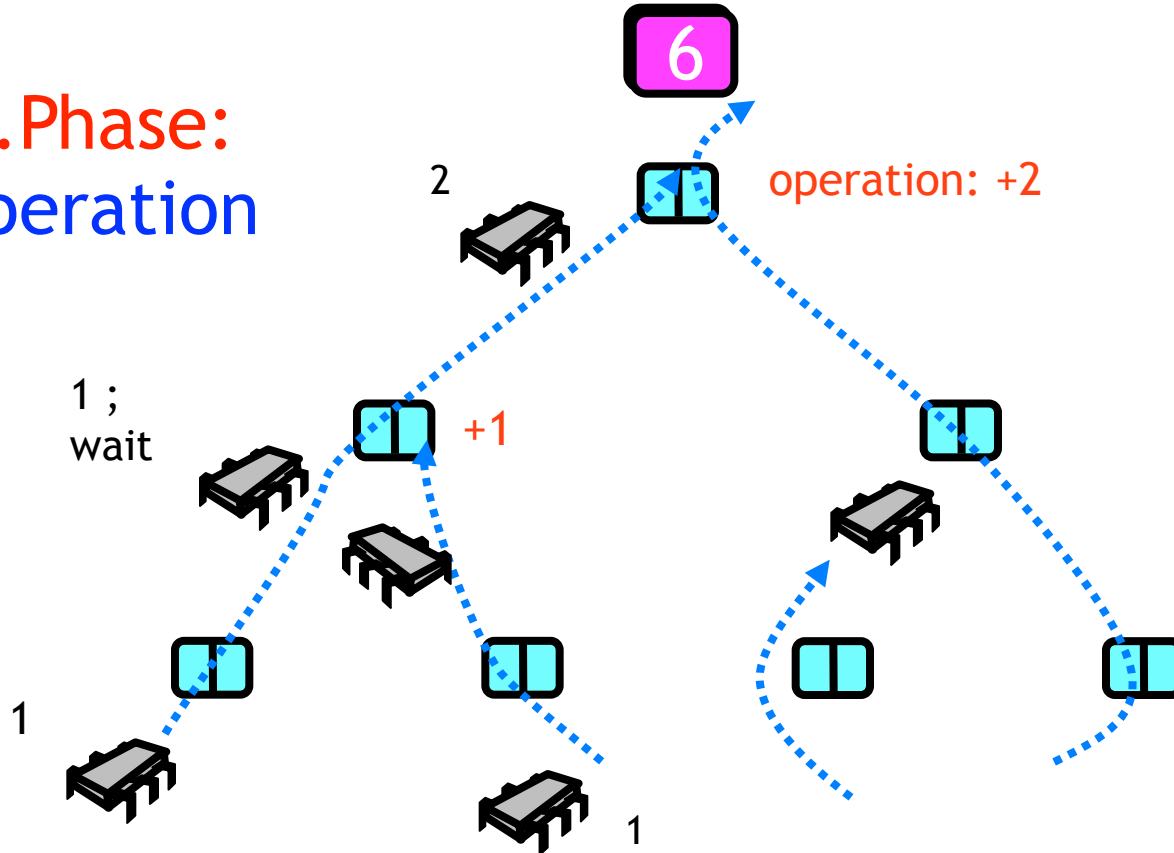
# Software Combining Tree

3.Phase:  
Operation



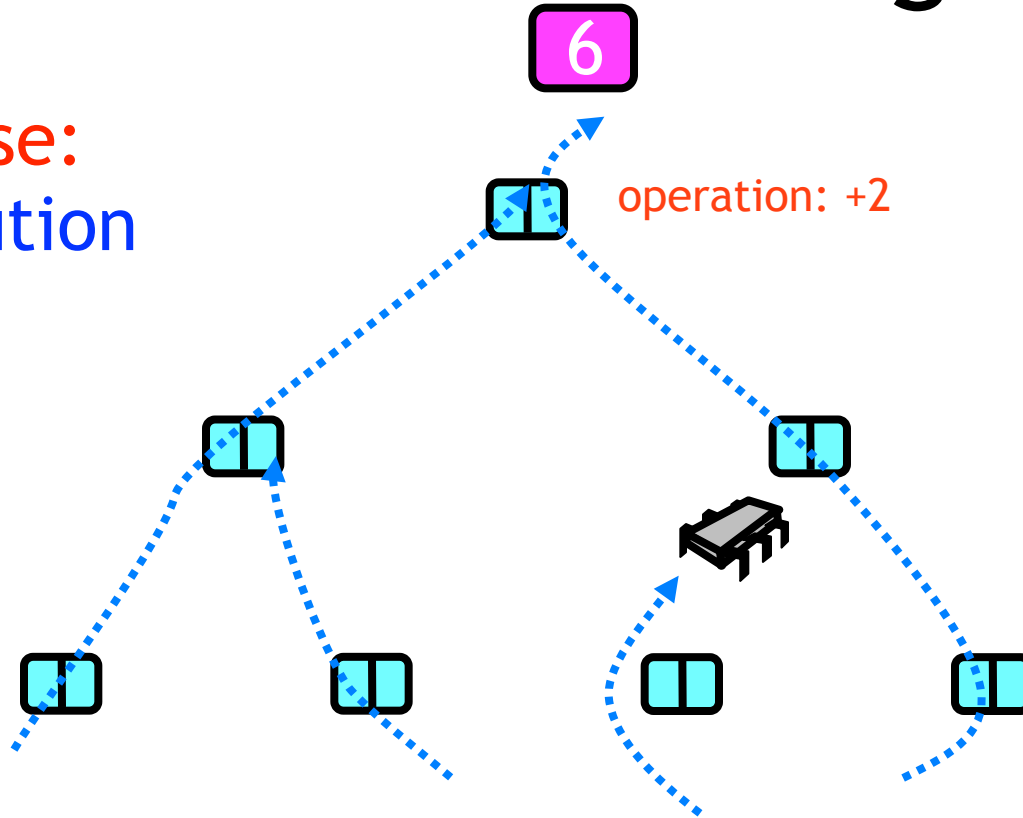
# Software Combining Tree

3.Phase:  
Operation



# Software Combining Tree

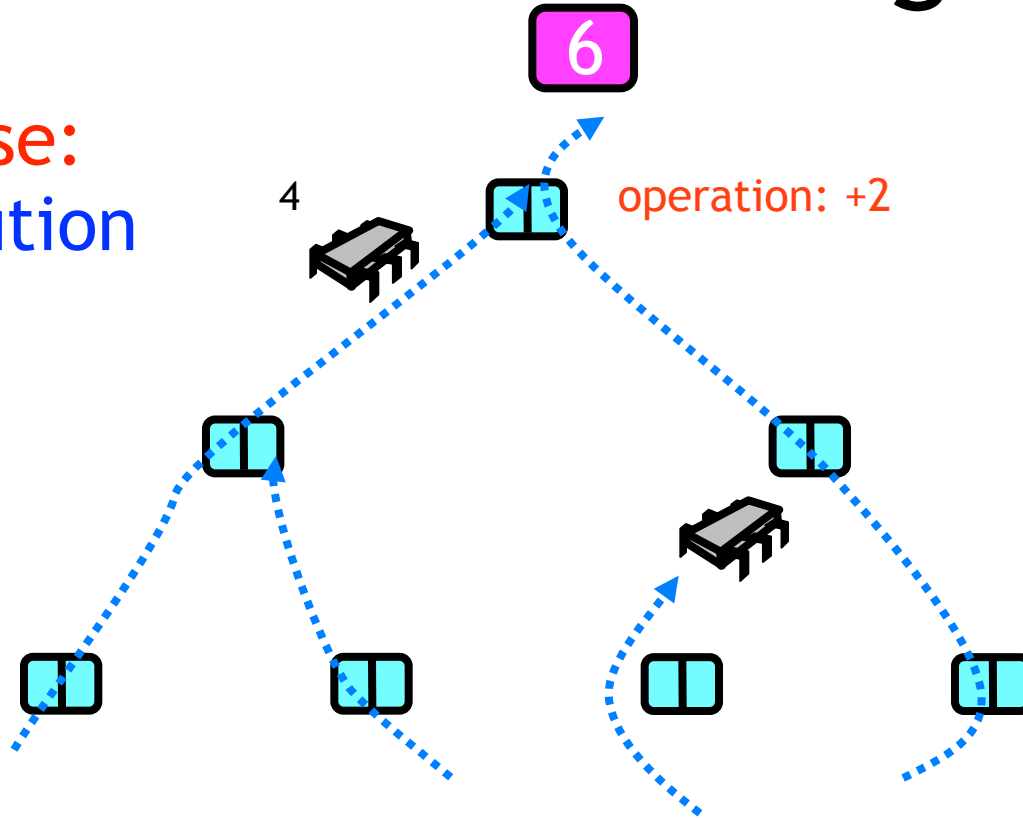
4.Phase:  
Distribution





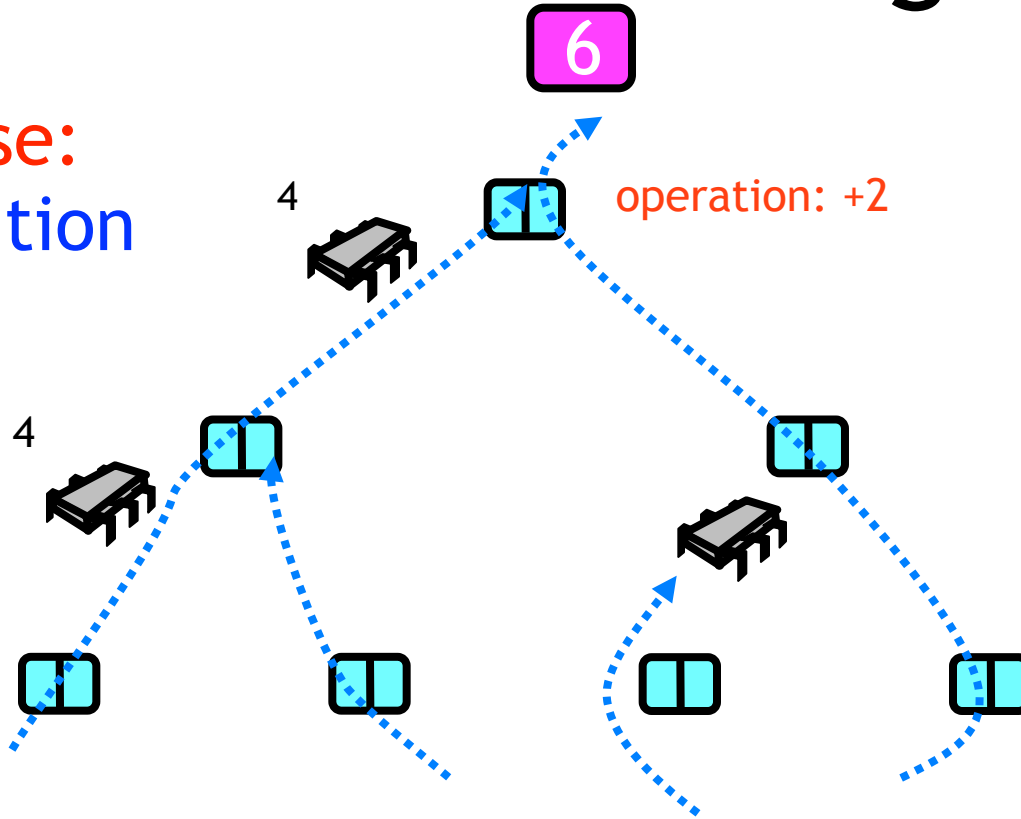
# Software Combining Tree

4.Phase:  
Distribution



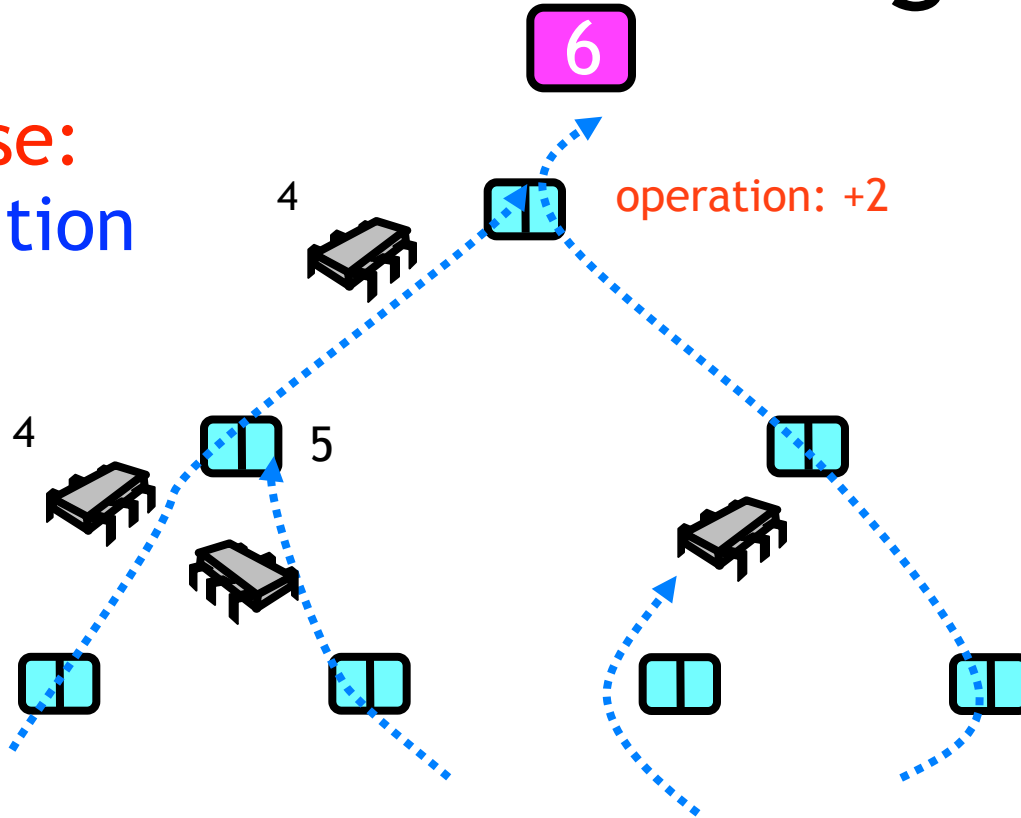
# Software Combining Tree

4.Phase:  
Distribution



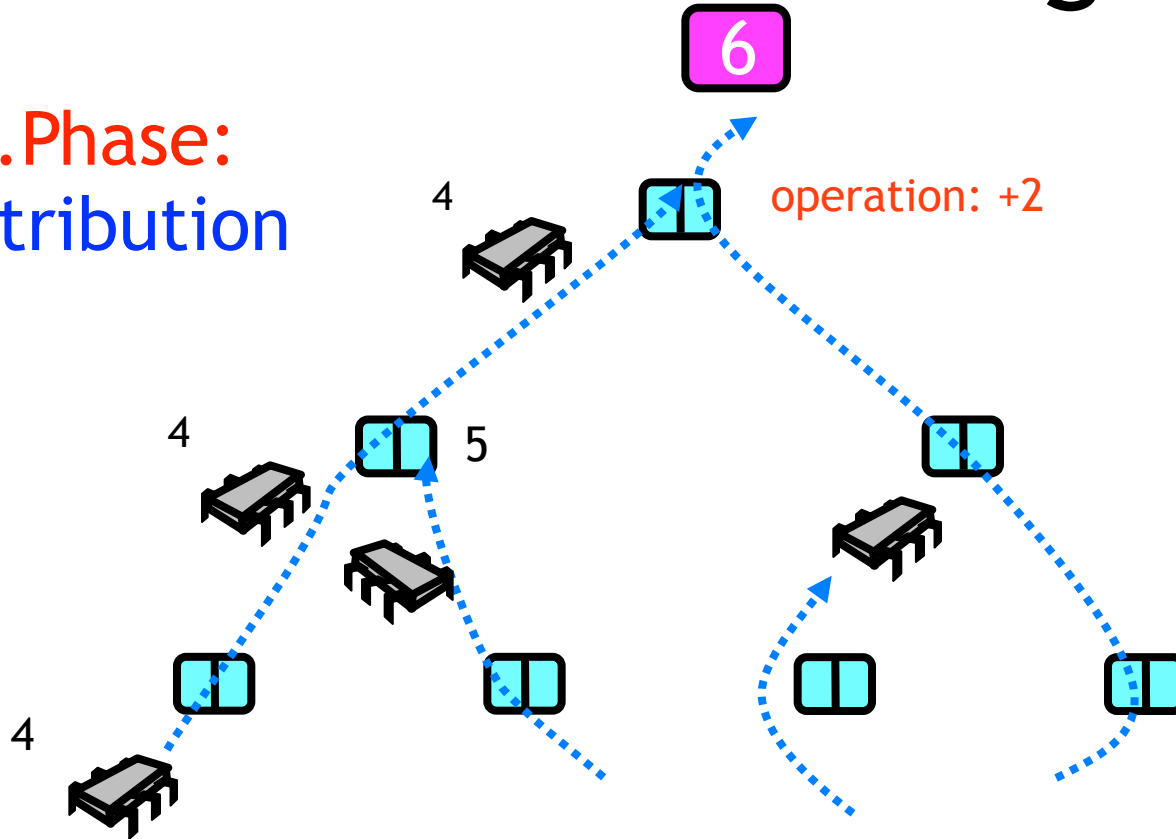
# Software Combining Tree

4.Phase:  
Distribution



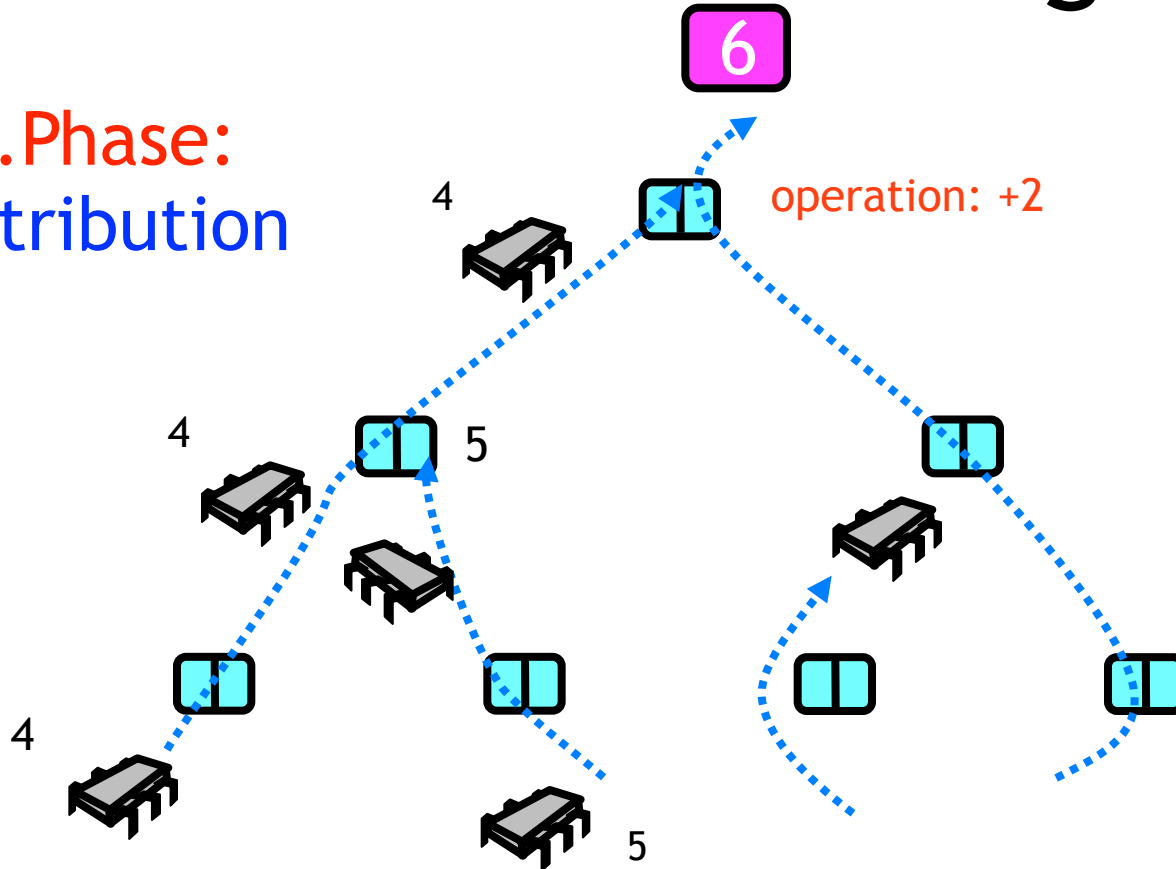
# Software Combining Tree

4.Phase:  
Distribution



# Software Combining Tree

4.Phase:  
Distribution



# Combining Status

```
enum CStatus{  
    IDLE, FIRST, SECOND, DONE, ROOT};
```

# Combining Status

```
enum CStatus{  
  IDLE, FIRST, SECOND, DONE, ROOT};
```



**Nothing going on**

# Combining Status

```
enum CStatus{  
  IDLE, FIRST, SECOND, DONE, ROOT};
```

**1<sup>st</sup> thread is partner for combining,  
will return soon to check for 2<sup>nd</sup>  
thread**



# Combining Status

```
enum CStatus{  
  IDLE, FIRST, SECOND, DONE, ROOT};
```

**2<sup>nd</sup> thread arrived with  
value for combining**

# Combining Status

```
enum CStatus{  
  IDLE, FIRST, SECOND, DONE, ROOT};
```



**1<sup>st</sup> thread has completed  
operation & deposited result for  
2<sup>nd</sup> thread**

# Combining Status

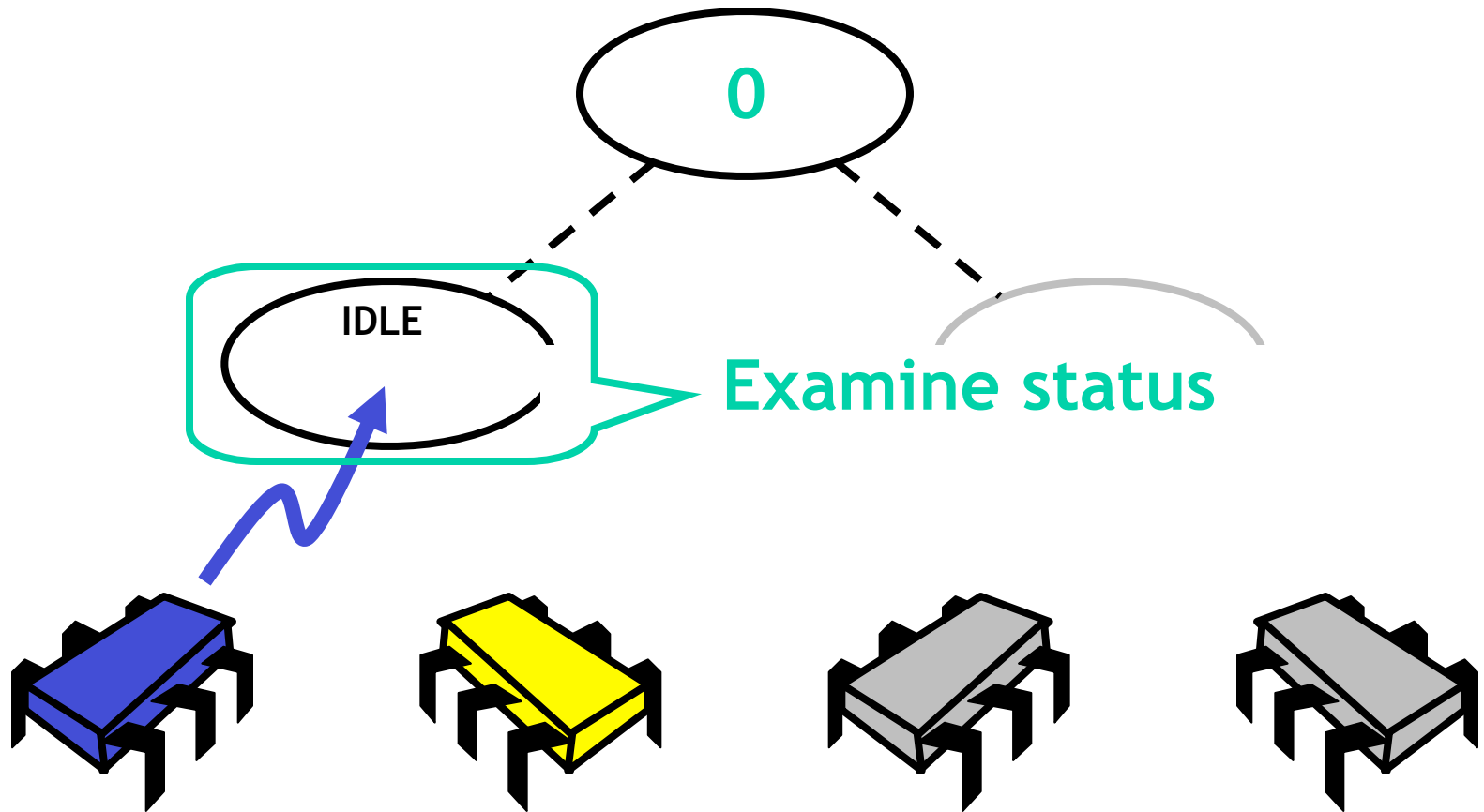
```
enum CStatus{  
  IDLE, FIRST, SECOND, DONE, ROOT};
```

**Special case: root node**

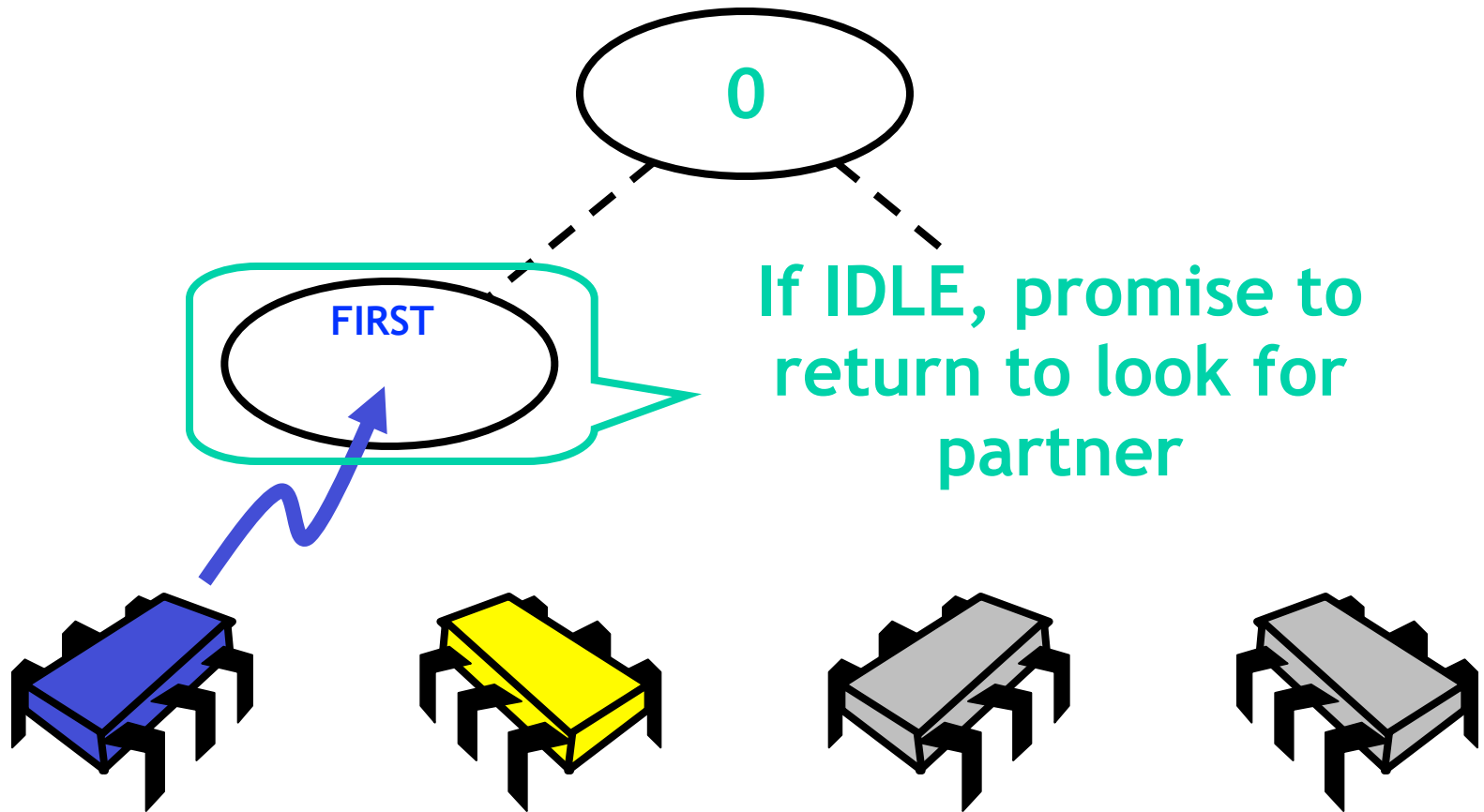
# Node Synchronization

- Short-term
  - Synchronized methods
  - Consistency during method call
- Long-term
  - Boolean locked field
  - Consistency across calls

# Precombining Phase

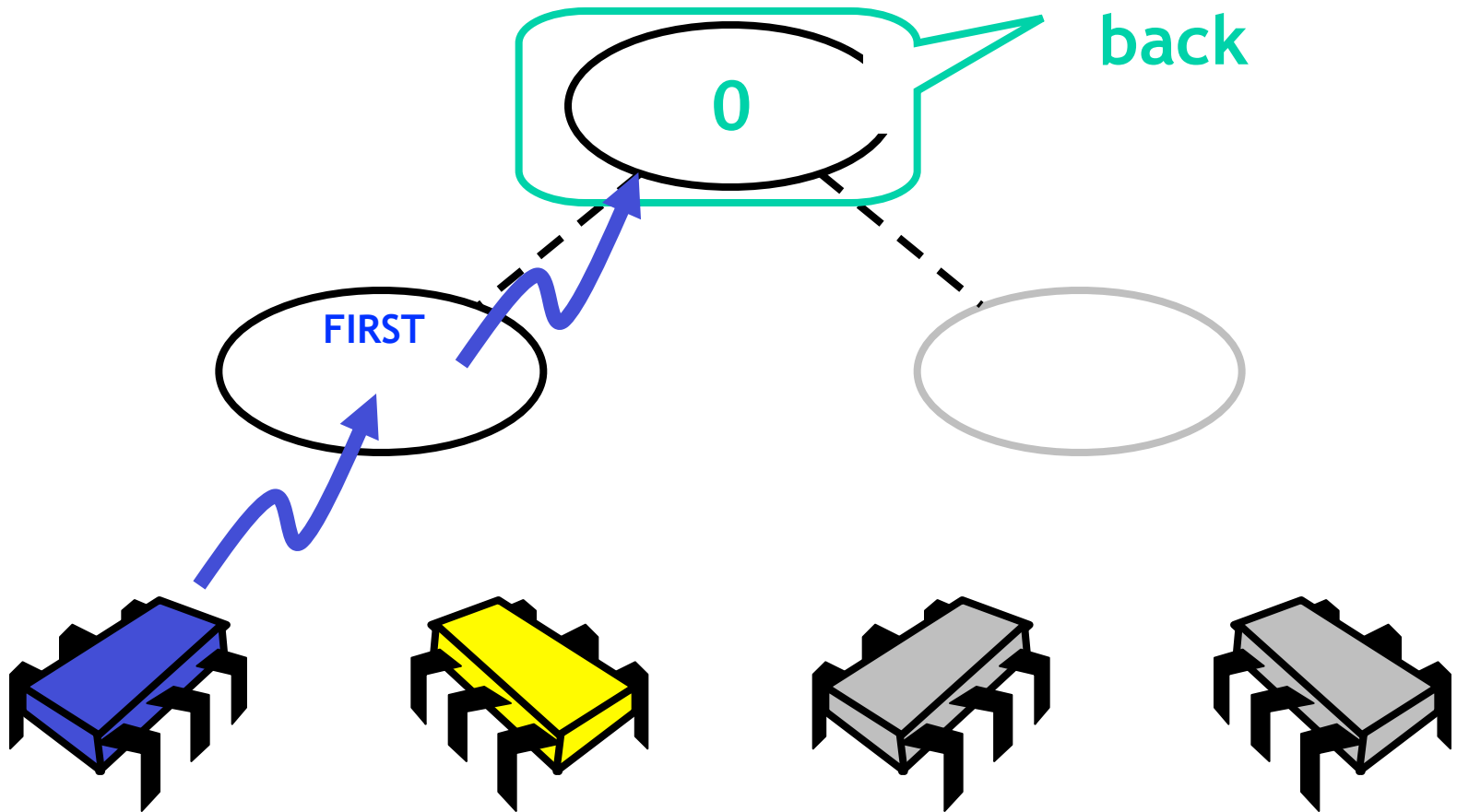


# Precombining Phase

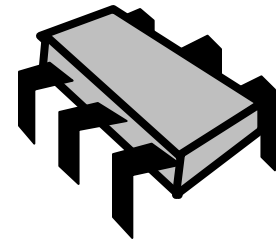
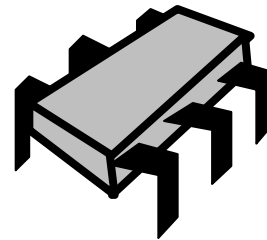
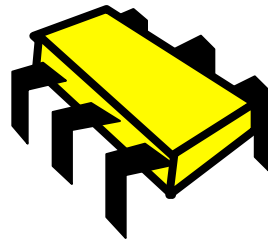
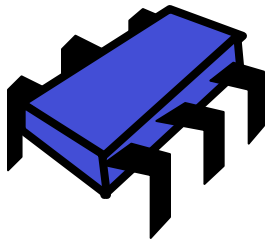
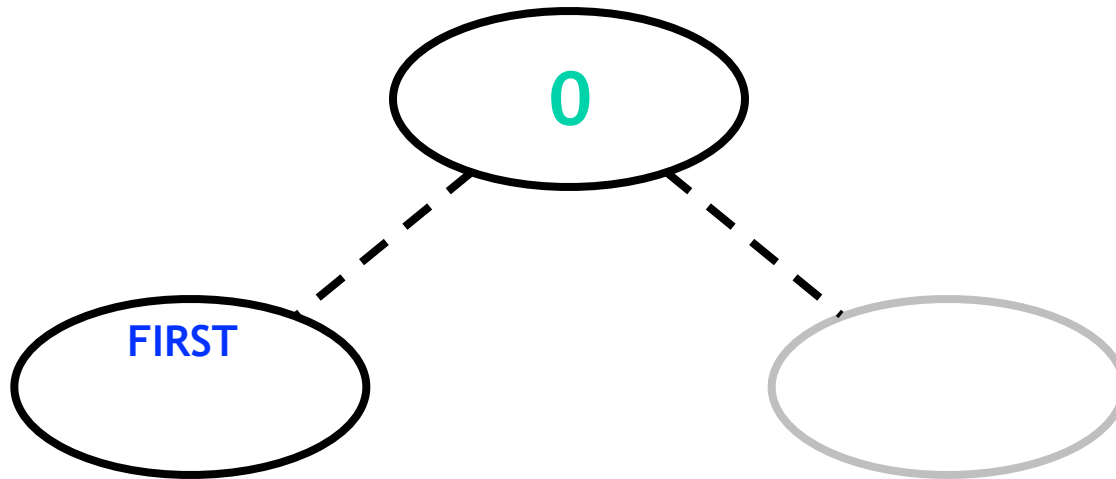


# Precombining Phase

At ROOT, turn back

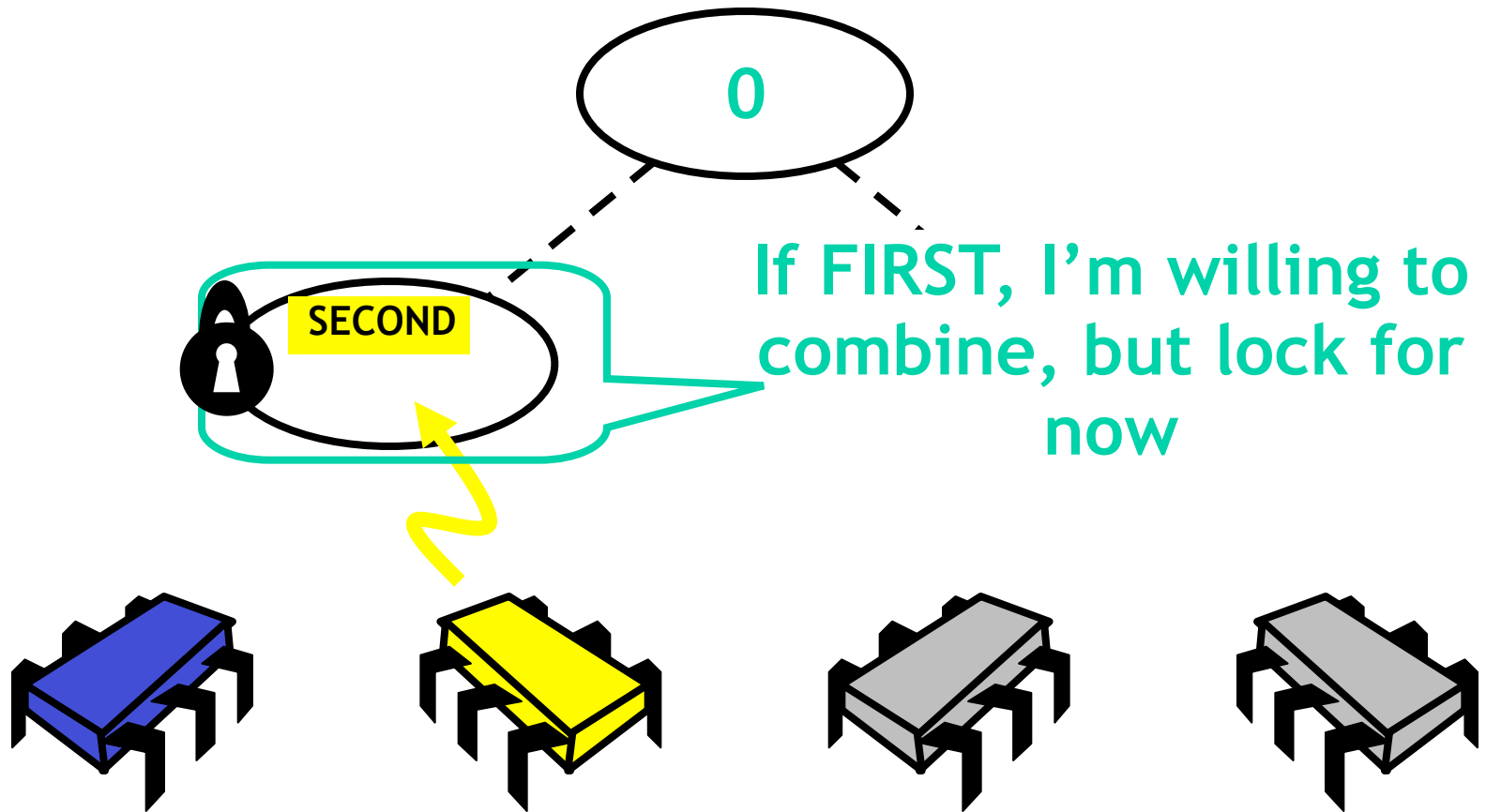


# Precombining Phase





# Precombining Phase



# Code

- Tree class
  - In charge of navigation
- Node class
  - Combining state
  - Synchronization state
  - Bookkeeping

# Precombining Navigation

```
Node node = myLeaf;  
while (node.precombine()) {  
    node = node.parent;  
}  
Node stop = node;
```

# Precombining Navigation

```
Node node = myLeaf;
```

```
while (node.precombine()) {  
    node = node.parent;  
}
```

```
Node stop = node;
```

**Start at leaf**

# Precombining Navigation

```
Node node = myLeaf;  
while (node.precombine()) {  
    node = node.parent;  
}  
Node stop = node;
```

**Move up while instructed  
to do so**

# Precombining Navigation

```
Node node = myLeaf;  
while (node.precombine()) {  
    node = node.parent;  
}
```

```
Node stop = node;
```

**Remember where we  
stopped**

# Precombining Node

```
synchronized boolean precombine() {  
    while (locked) wait();  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
            return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return false;  
        default: throw new PanicException()  
    }  
}
```

# Precombining Node

```
synchronized boolean precombine() {  
    while (locked) wait();  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
            return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return false;  
        default: throw new PanicException()  
    }  
}
```

**Short-term  
synchronization**



# Synchronization

```
synchronized boolean precombine() {  
    while (locked) wait();  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
            return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return false;  
        default: throw new PanicException()  
    }  
}
```

**Wait while node is locked**

# Precombining Node

```
synchronized boolean precombine() {  
    while (locked) wait();  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
            return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return false;  
        default: throw new PanicException()  
    }  
}
```

**Check combining status**

# Node was IDLE

```
synchronized boolean precombine() {  
    while (locked) {wait();}  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
            return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return false;  
        default: throw new PanicException()  
    }  
}
```

**I will return to look for  
combining value**

# Precombining Node

```
synchronized boolean precombine() {  
    while (locked) {wait();}  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
        return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return  
        default: throw new PanicException()  
    }  
}
```

**Continue up the tree**

# I'm the 2<sup>nd</sup> Thread

```
synchronized boolean precombine() {  
    while (locked) {wait();}  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
                 return true;  
        case FIRST: locked = true;  
                  cStatus = CStatus.SECOND;  
                  return false;  
        case ROOT: return false;  
        default: throw new PanicException();  
    }  
}
```

**If 1<sup>st</sup> thread has promised to return, lock node so it won't leave without me**

# Precombining Node

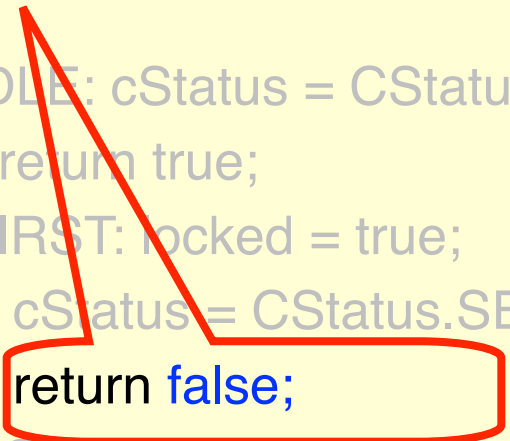
```
synchronized boolean precombine() {  
    while (locked) {wait();}  
    switch (cStatus) {  
        case IDLE: cStatus = CStatus.FIRST;  
            return true;  
        case FIRST: locked = true;  
            cStatus = CStatus.SECOND;  
            return false;  
        case ROOT: return false;  
        default: throw new PanicException()  
    }  
}
```

**Prepare to deposit 2<sup>nd</sup>  
value**

# Precombining Node

```
synchr  
while (  
switch  
case IDLE: cStatus = CStatus.FIRST;  
    return true;  
case FIRST: locked = true;  
    cStatus = CStatus.SECOND;  
    return false;  
case ROOT: return false;  
default: throw new PanicException()  
}  
}
```

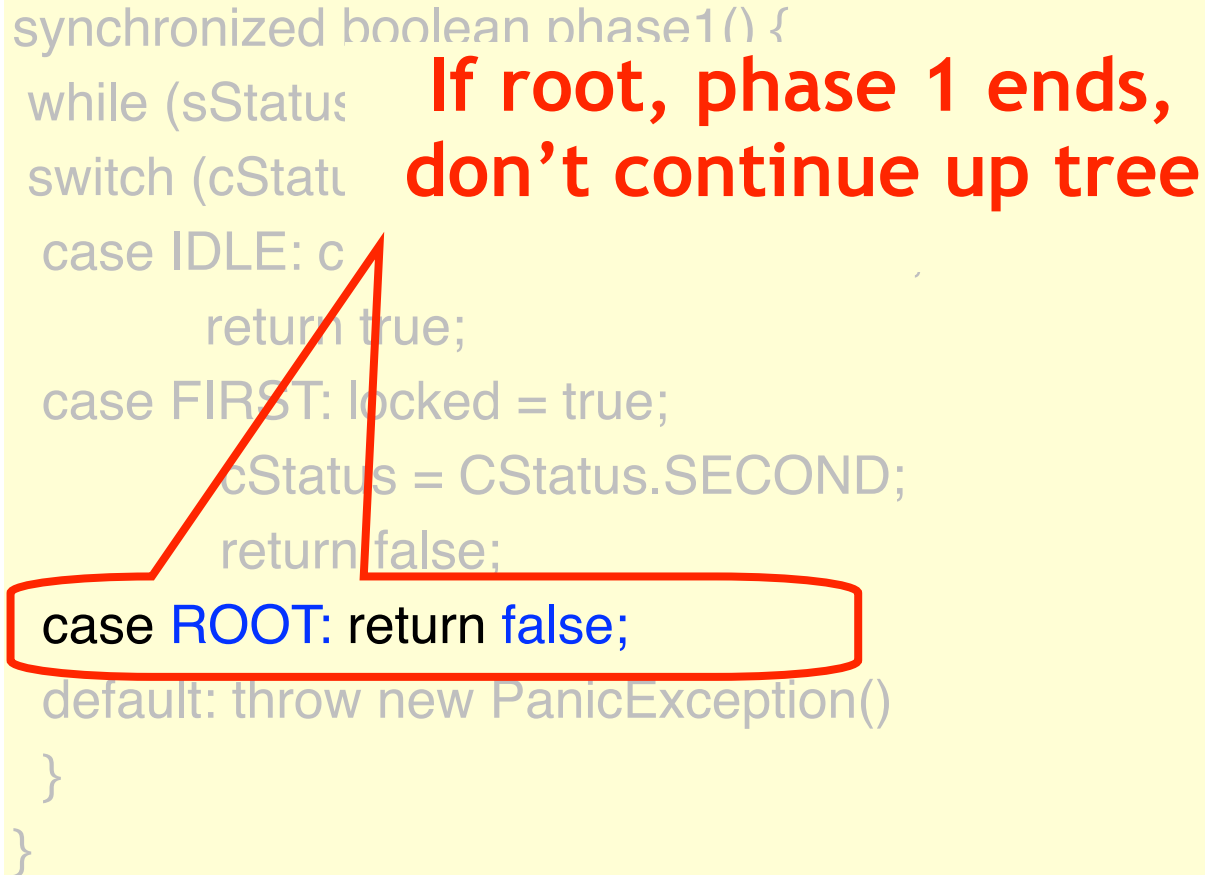
**End of phase 1, don't  
continue up tree**



# Node is the Root

```
synchronized boolean phase1() {  
    while (sStatus == CStatus.FIRST) {  
        switch (cStatus) {  
            case IDLE: cStatus = CStatus.IDLE;  
                return true;  
            case FIRST: locked = true;  
                cStatus = CStatus.SECOND;  
                return false;  
            case ROOT: return false;  
            default: throw new PanicException("Invalid state");  
        }  
    }  
}
```

**If root, phase 1 ends,  
don't continue up tree**





# Precombining Node

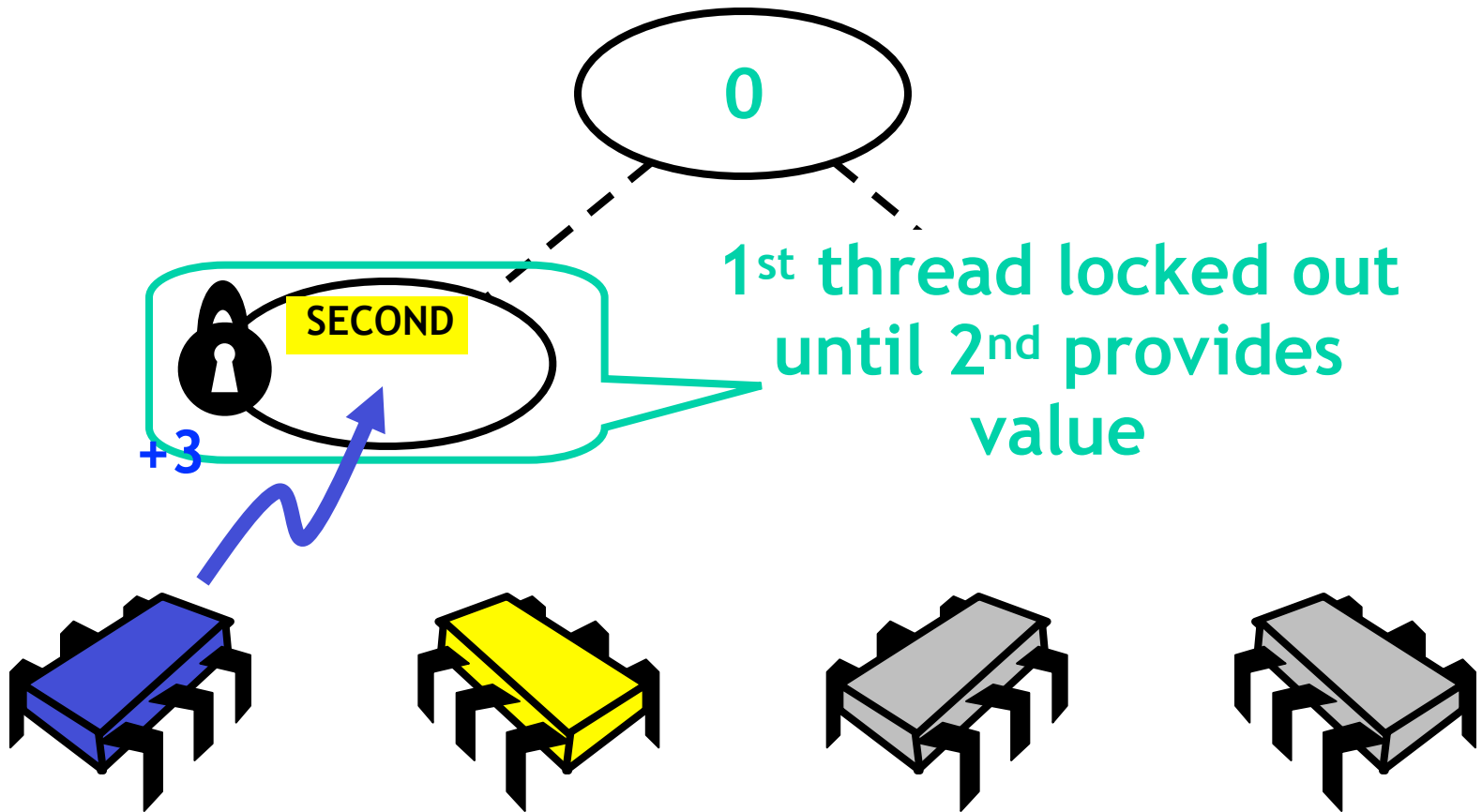
```
synchronized boolean phase1() {  
    while (locked)  
        switch (cStatu  
            case IDLE: c  
                return true;  
            case FIRST: locked = true;  
                cStatus = CStatus.SECOND;  
                return false;  
            case ROOT: return false;  
            default: throw new PanicException()  
        }  
}
```

**Always check for  
unexpected values!**

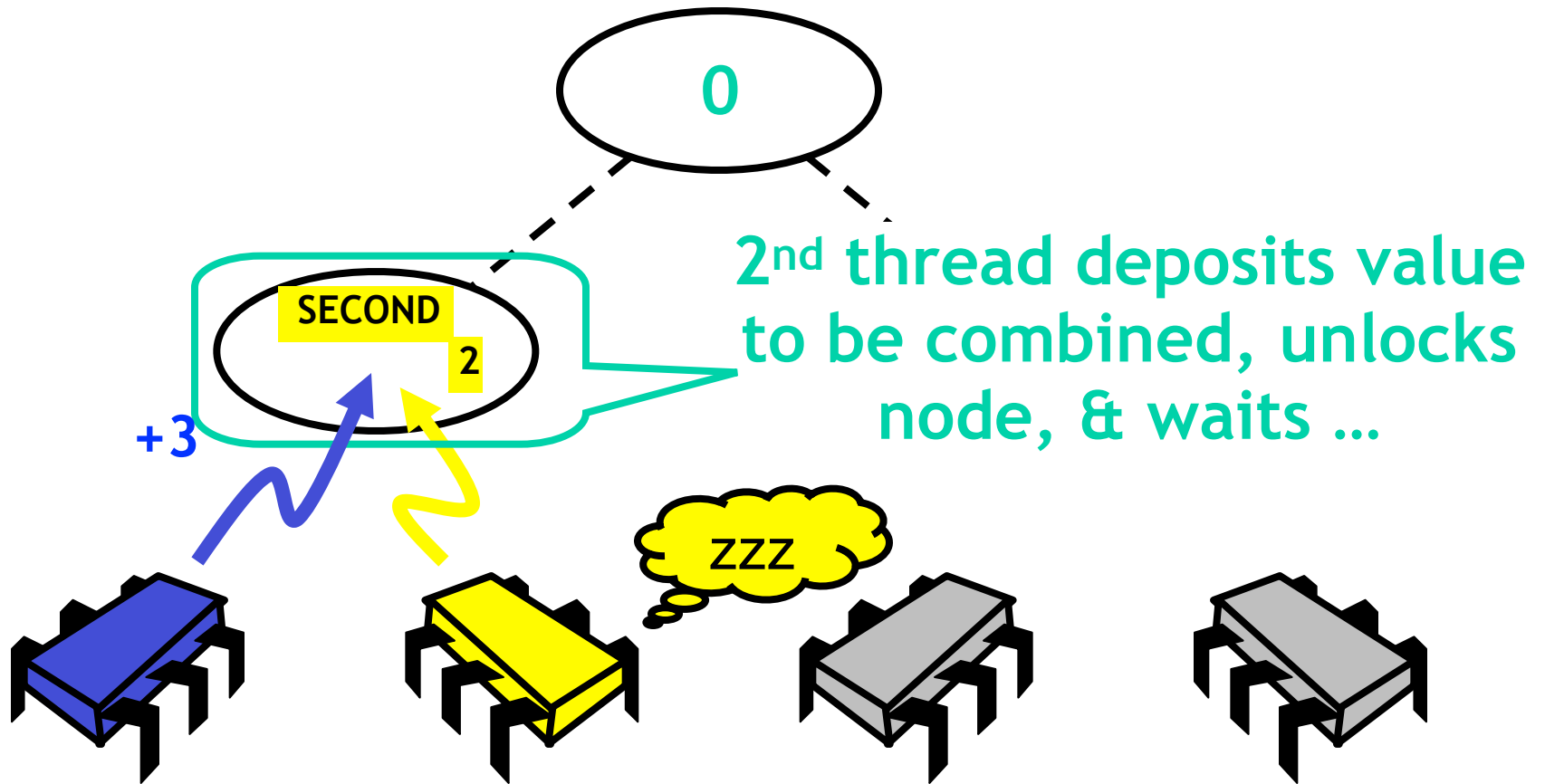


default: throw new `PanicException()`

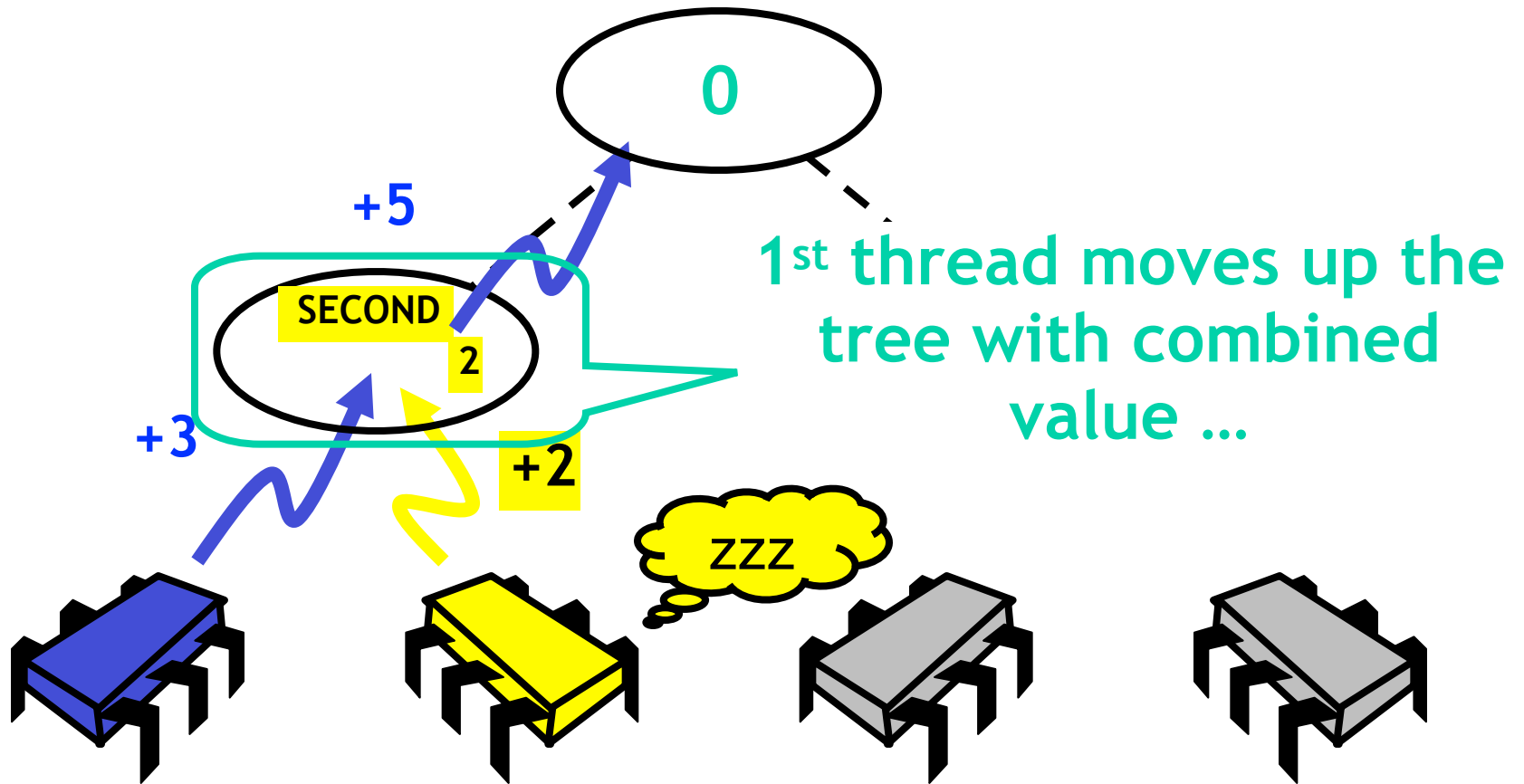
# Combining Phase



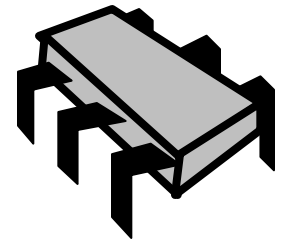
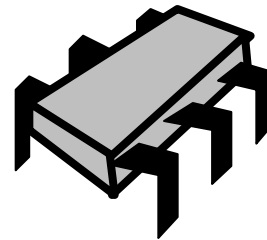
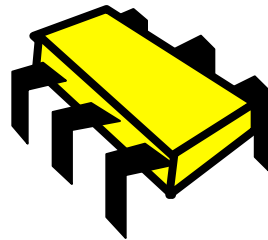
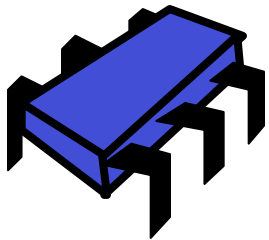
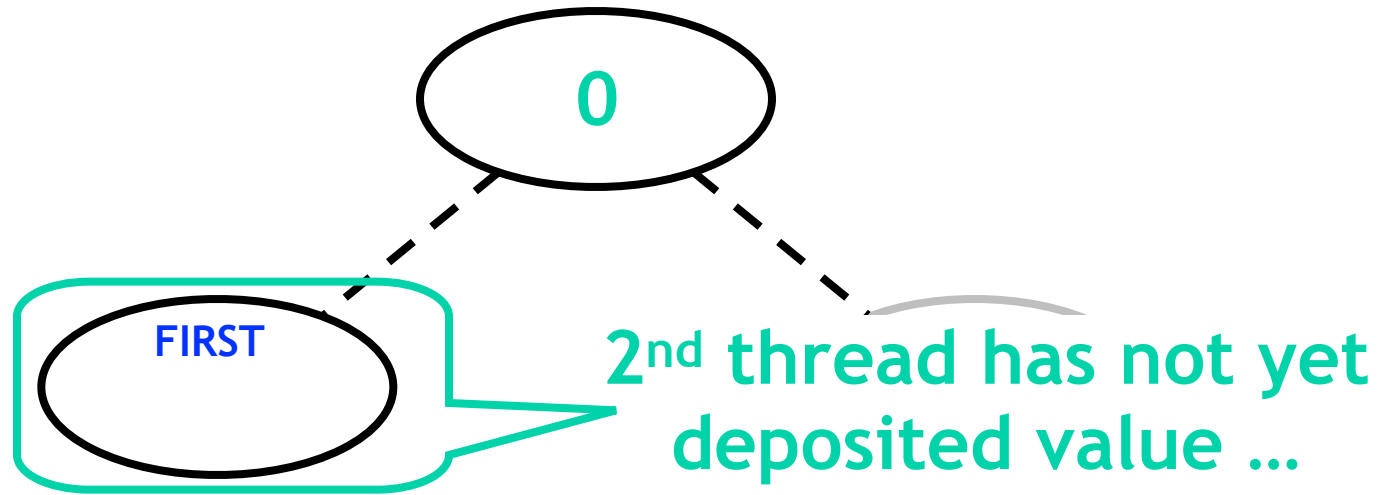
# Combining Phase



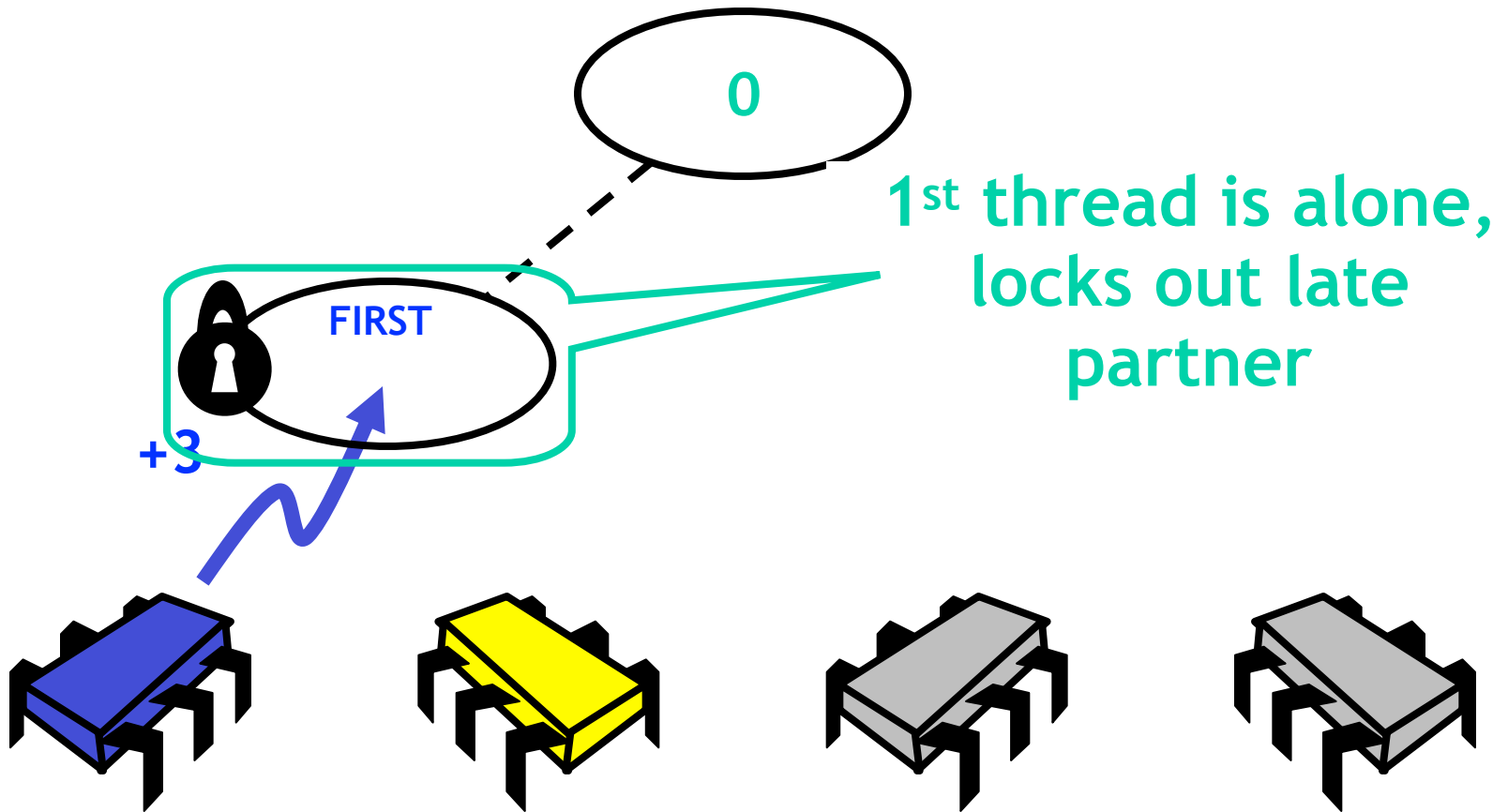
# Combining Phase



# Combining (reloaded)

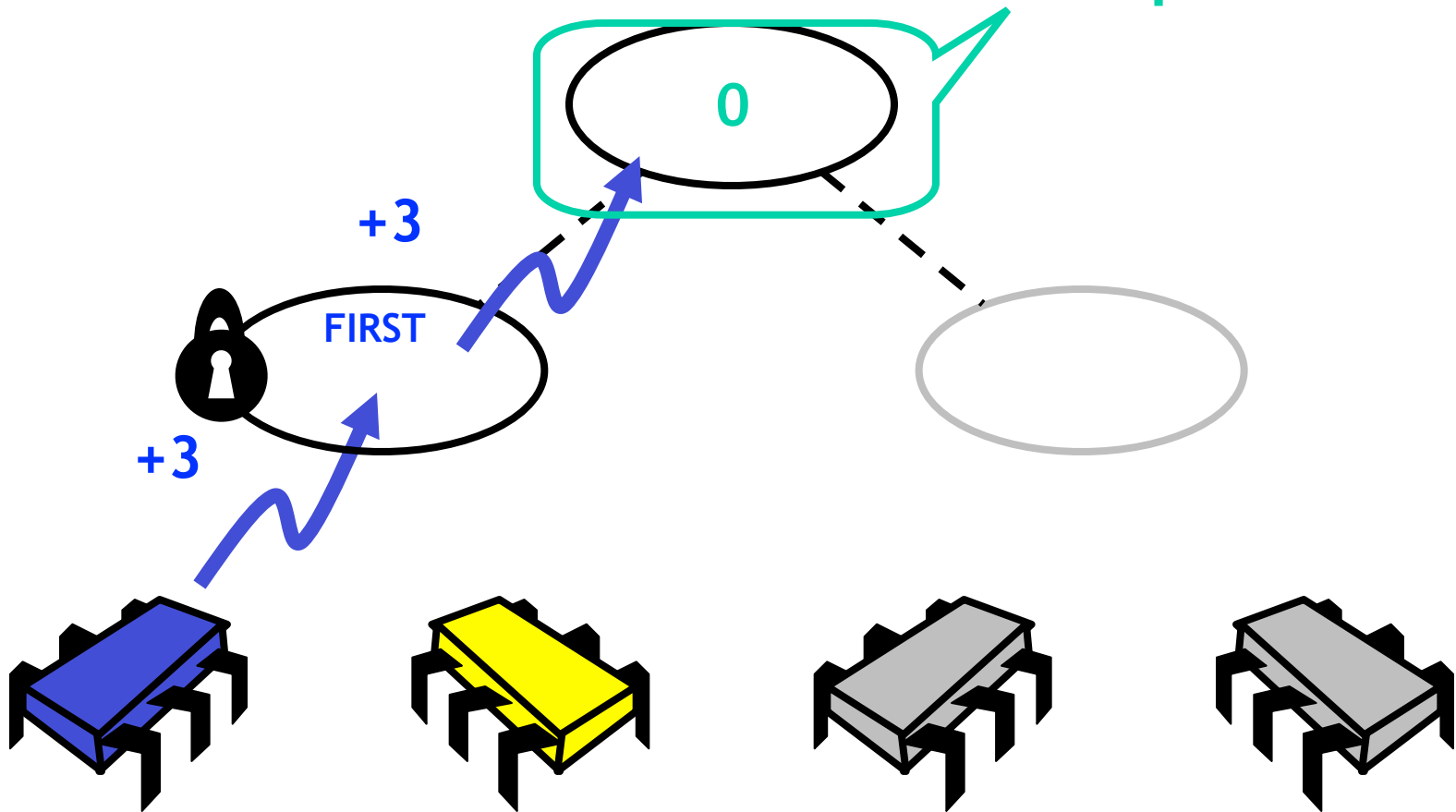


# Combining (reloaded)

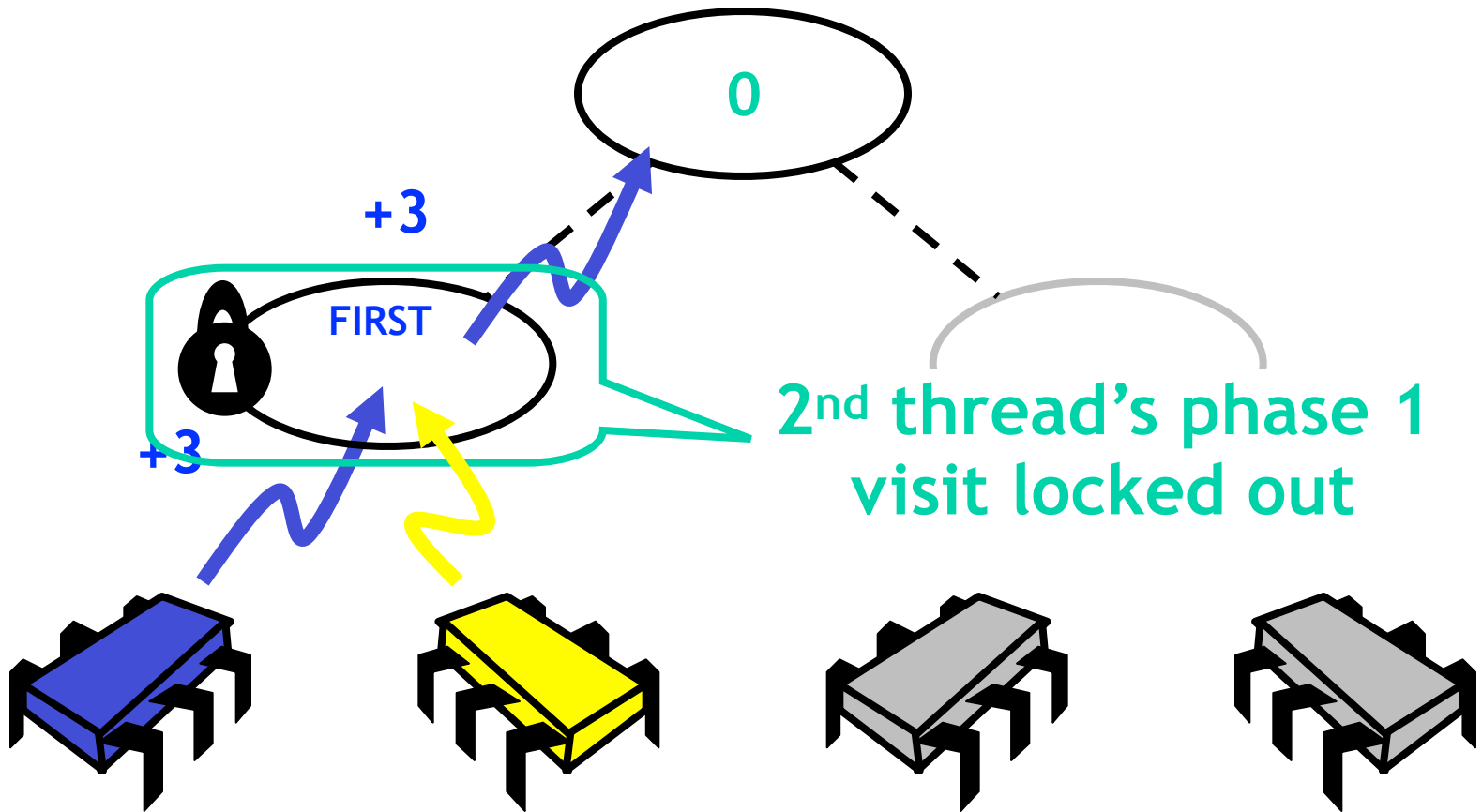


# Combining (reloaded)

Stop at root



# Combining (reloaded)





# Combining Navigation

```
node = myLeaf;  
int combined = 1;  
while (node != stop) {  
    combined = node.combine(combined);  
    stack.push(node);  
    node = node.parent;  
}
```

# Combining Navigation

```
node = myLeaf;
```

```
int combined = 1,  
while (node != stop) {  
    combined = node.combine(combined);  
    stack.push(node);  
    node = node.parent;  
}
```

**Start at leaf**

# Combining Navigation

```
node = myLeaf;  
int combined = 1;  
while (node != stop) {  
    combined = node.combine(combined);  
    stack.push(node);  
    node = node.parent;  
}
```

**Add 1**

# Combining Navigation

```
node = myLeaf;  
int combined = 1;  
while (node != stop) {  
    combined = node.combine(combined);  
    stack.push(node);  
    node = node.parent;  
}
```

**Revisit nodes visited  
in phase 1**

# Combining Navigation

```
node = myLeaf;  
int combined = 1;  
while (node != stop) {  
    combined = node.combine(combined);  
    stack.push(node);  
    node = node.parent;  
}
```

**Accumulate combined  
values, if any**

# Combining Navigation

```
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}
```

**We will retrace path in reverse order ...**

**stack.push(node);**

# Combining Navigation

```
node = myLeaf;  
int combined = 1;  
while (node != stop) {  
    combined = node.combine(combined);  
    stack.push(node);  
    node = node.parent;  
}
```

**Move up the tree**

# Combining Phase Node

```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
        case FIRST:  
            return firstValue;  
        case SECOND:  
            return firstValue + secondValue;  
        default: ...  
    }  
}
```



# Combining Phase Node

```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
        case FIRST:  
            return firstValue;  
        case SECOND:  
            return firstValue + secondValue;  
        default: ...  
    }  
}
```

**Wait until node is unlocked**

# Combining Phase Node

```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
        case FIRST:  
            return firstValue;  
        case SECOND:  
            return firstValue + secondValue;  
        default: ...  
    }  
}
```

**Lock out late attempts  
to combine**

# Combining Phase Node

```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
        case FIRST:  
            return firstValue;  
        case SECOND:  
            return firstValue + secondValue;  
        default: ...  
    }  
}
```

**Remember our contribution**

# Combining Phase Node

```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
        case FIRST:  
            return firstValue;  
        case SECOND:  
            return firstValue + secondValue;  
        default: ...  
    }  
}
```

**Check status**



# Combining Phase Node

```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
    case FIRST:  
        return firstValue;  
    case SECOND:  
        return firstValue + secondValue;  
    default: ...  
    }  
}
```

**1<sup>st</sup> thread is alone**

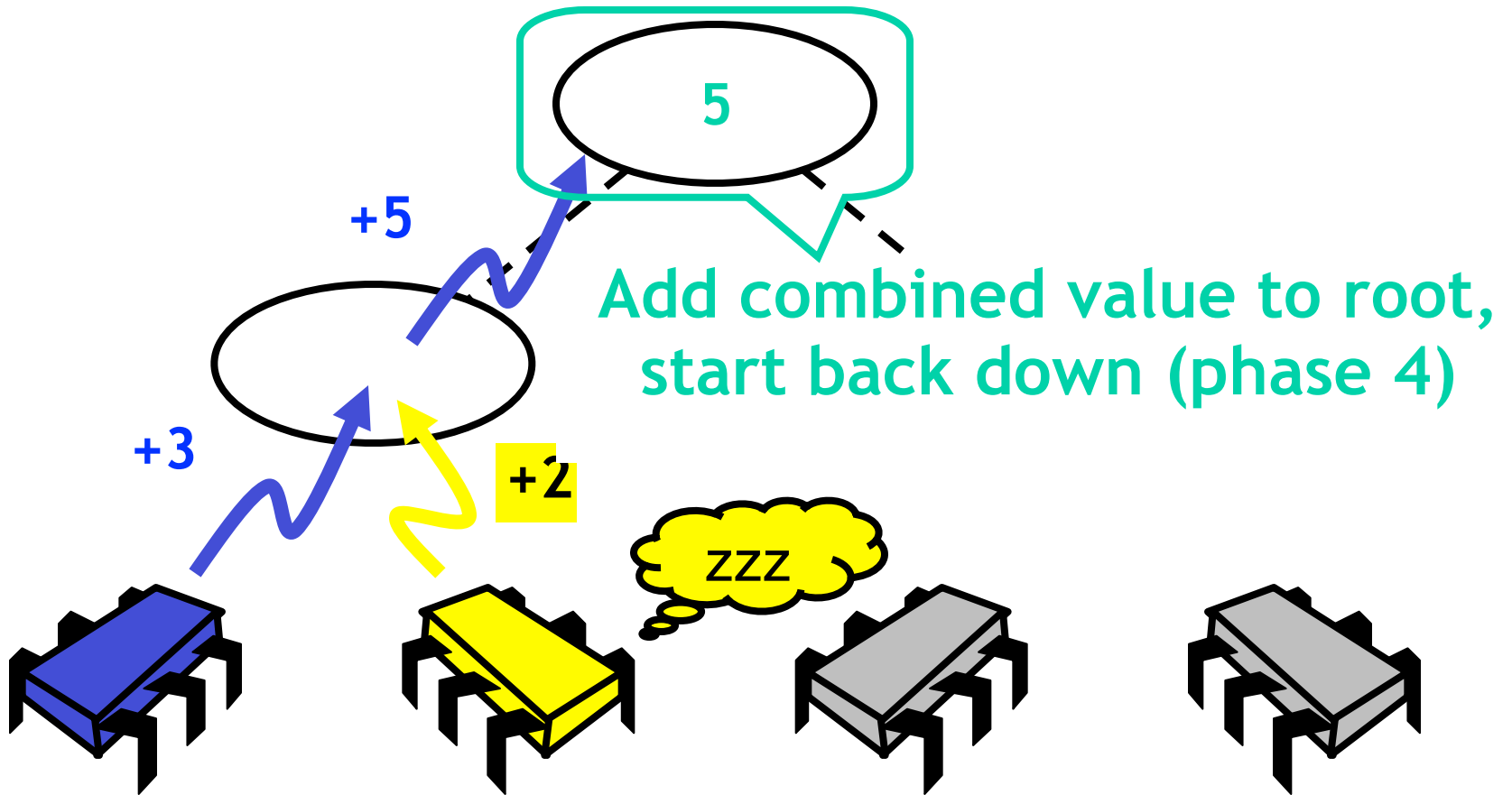


# Combining Node

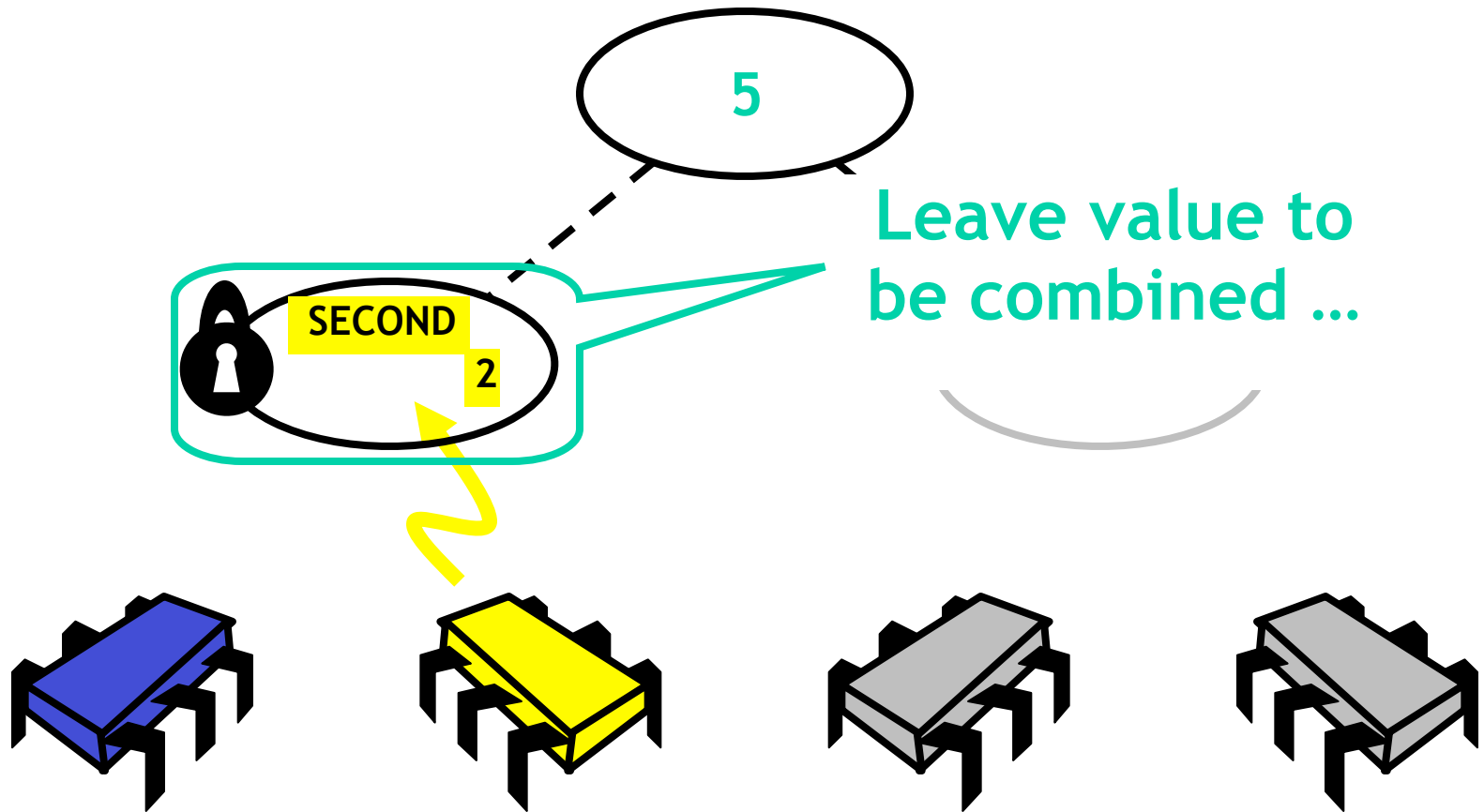
```
synchronized int combine(int combined) {  
    while (locked) wait();  
    locked = true;  
    firstValue = combined;  
    switch (cStatus) {  
        case FIRST:  
            return firstValue;  
        case SECOND:  
            return firstValue + secondValue;  
        default: ...  
    }  
}
```

**Combine with  
2<sup>nd</sup> thread**

# Operation Phase

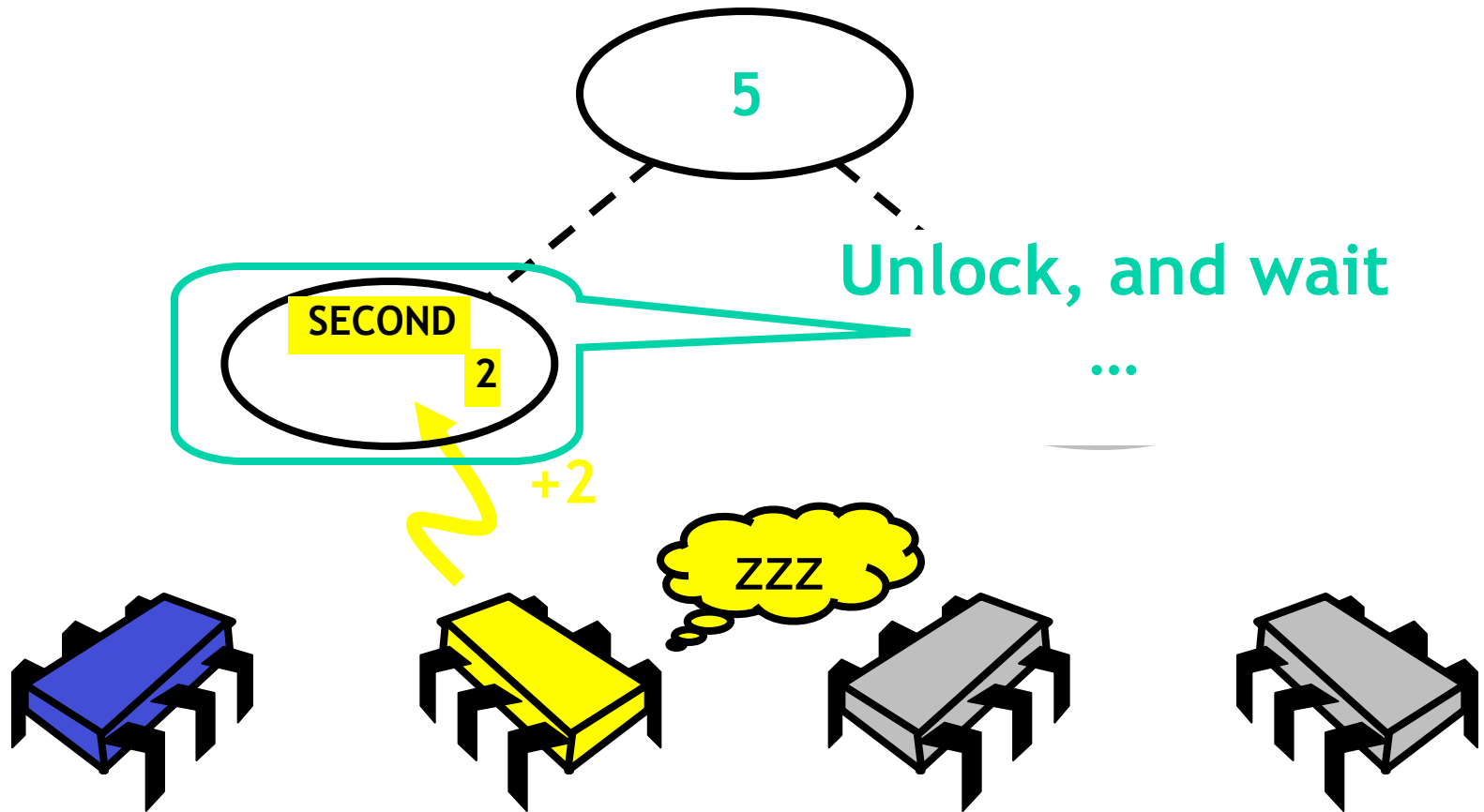


# Operation Phase (reloaded)





# Operation Phase (reloaded)



# Operation Phase Navigation

```
prior = stop.op(combined);
```

# Operation Phase Navigation

```
prior = stop.op(combined);
```

**Get result of  
combining**

# Operation Phase Node

```
synchronized int op(int combined) {  
    switch (cStatus) {  
        case ROOT: int oldValue = result;  
            result += combined;  
            return oldValue;  
        case SECOND: secondValue = combined;  
            locked = false; notifyAll();  
            while (cStatus != CStatus.DONE) wait();  
            locked = false; notifyAll();  
            cStatus = CStatus.IDLE;  
            return result;  
        default: ...  
    }  
}
```

# At Root

```
synchronized int op(int combined) {  
    switch (cStatus) {
```

```
    case ROOT: int oldValue = result;  
               result += combined;  
               return oldValue;
```

```
    case SECOND: secondValue = combined;  
                 locked = false; notifyAll();  
                 while (cStatus != CStatus.DONE) wait();  
                 locked = false; notifyAll();  
                 cStatus = CStatus.IDLE;  
                 return result;  
    default: ...
```

**Add sum to root,  
return prior value**

# Intermediate Node

```
synchronized int op(int combined) {  
    switch (cStatus) {  
        case ROOT: int oldValue = result;  
            result += combined;  
            return oldValue;  
        case SECOND: secondValue = combined;  
            locked = false; notifyAll();  
            while (cStatus != CStatus.DONE) wait();  
            locked = false; notifyAll();  
            cStatus = CStatus.IDLE;  
            return result;  
        default: ...  
    }  
}
```

**Deposit value for later  
combining ...**

# Intermediate Node

```
synchronized int op(int combined) {  
    switch (cStatus) {  
        case ROOT: int oldValue = result;  
            result += combined;  
            return oldValue;  
        case SECOND: secondValue = combined;  
            locked = false; notifyAll();  
            while (cStatus != CStatus.DONE) wait();  
            locked = false; notifyAll();  
            cStatus = CStatus.IDLE;  
            return result;  
        default: ...
```

**Unlock node, notify  
1<sup>st</sup> thread**

# Intermediate Node

```
synchronized int op(int combined) {  
    switch (cStatus) {  
        case ROOT: int oldValue = result;  
            result += combined;  
            return oldValue;  
        case SECOND: secondValue = combined;  
            locked = false; notifyAll();  
            while (cStatus != CStatus.DONE) wait();  
            locked = false; notifyAll();  
            cStatus = CStatus.IDLE;  
            return result;  
        default: ...
```

**Wait for 1<sup>st</sup> thread  
to deliver results**

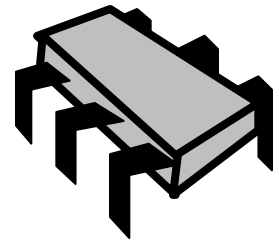
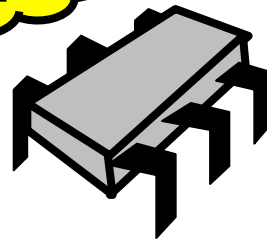
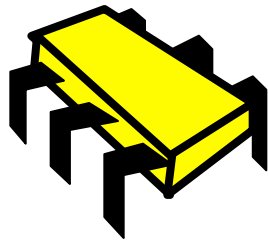
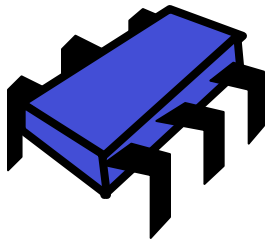
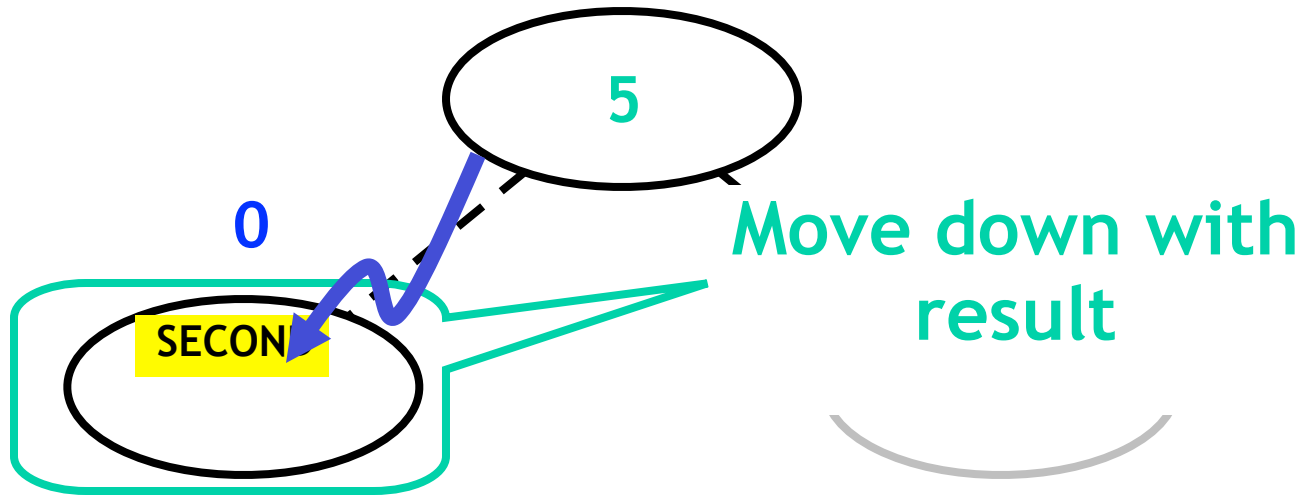


# Intermediate Node

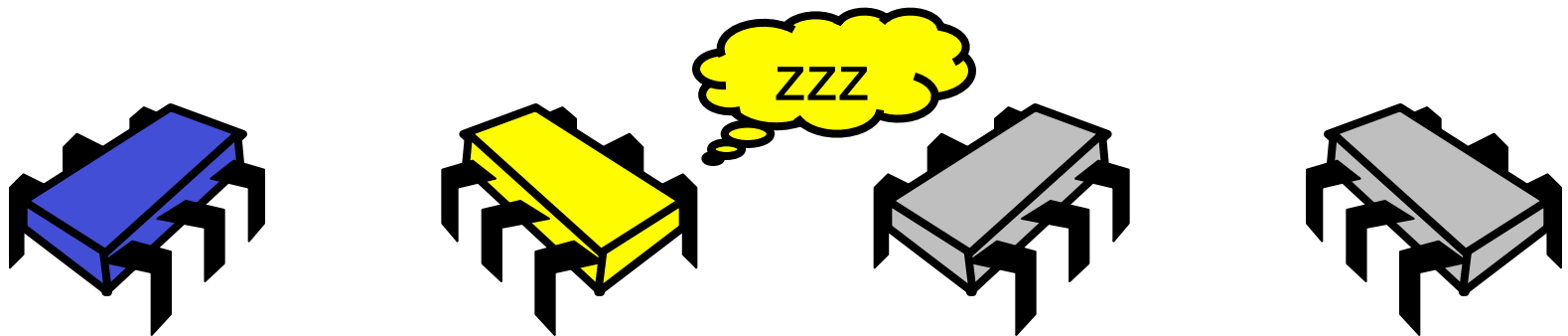
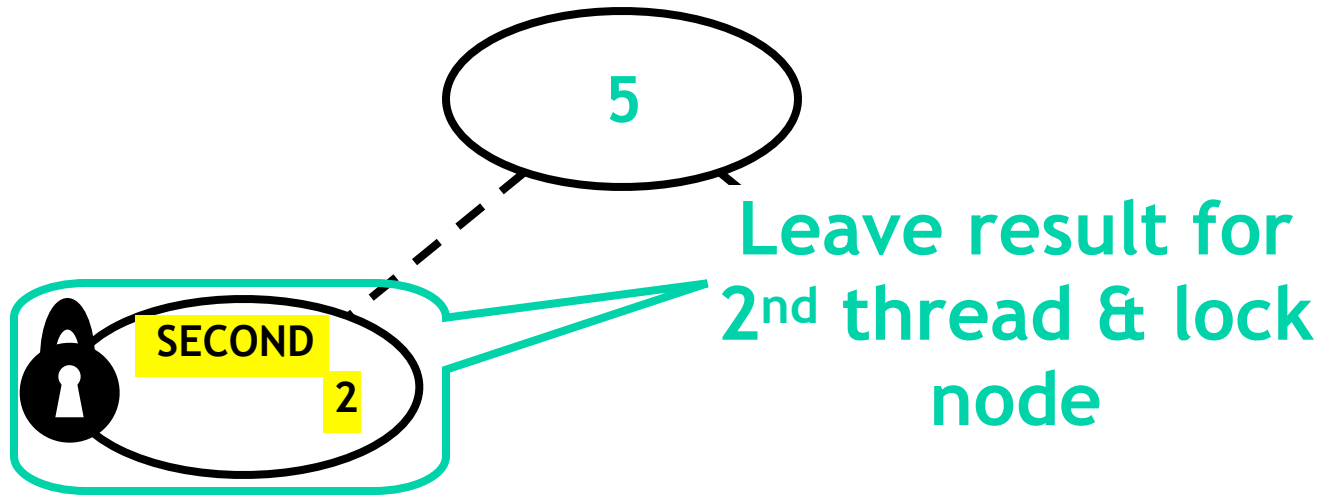
```
synchronized int op(int combined) {  
    switch (cStatus) {  
        case ROOT: int oldValue = result;  
            result += combined;  
            return oldValue;  
        case SECOND: secondValue = combined;  
            locked = false; notifyAll();  
            while (cStatus != CStatus.DONE) wait();  
            locked = false; notifyAll();  
            cStatus = CStatus.IDLE;  
            return result;  
        default: ...  
    }  
}
```

**Unlock node &  
return**

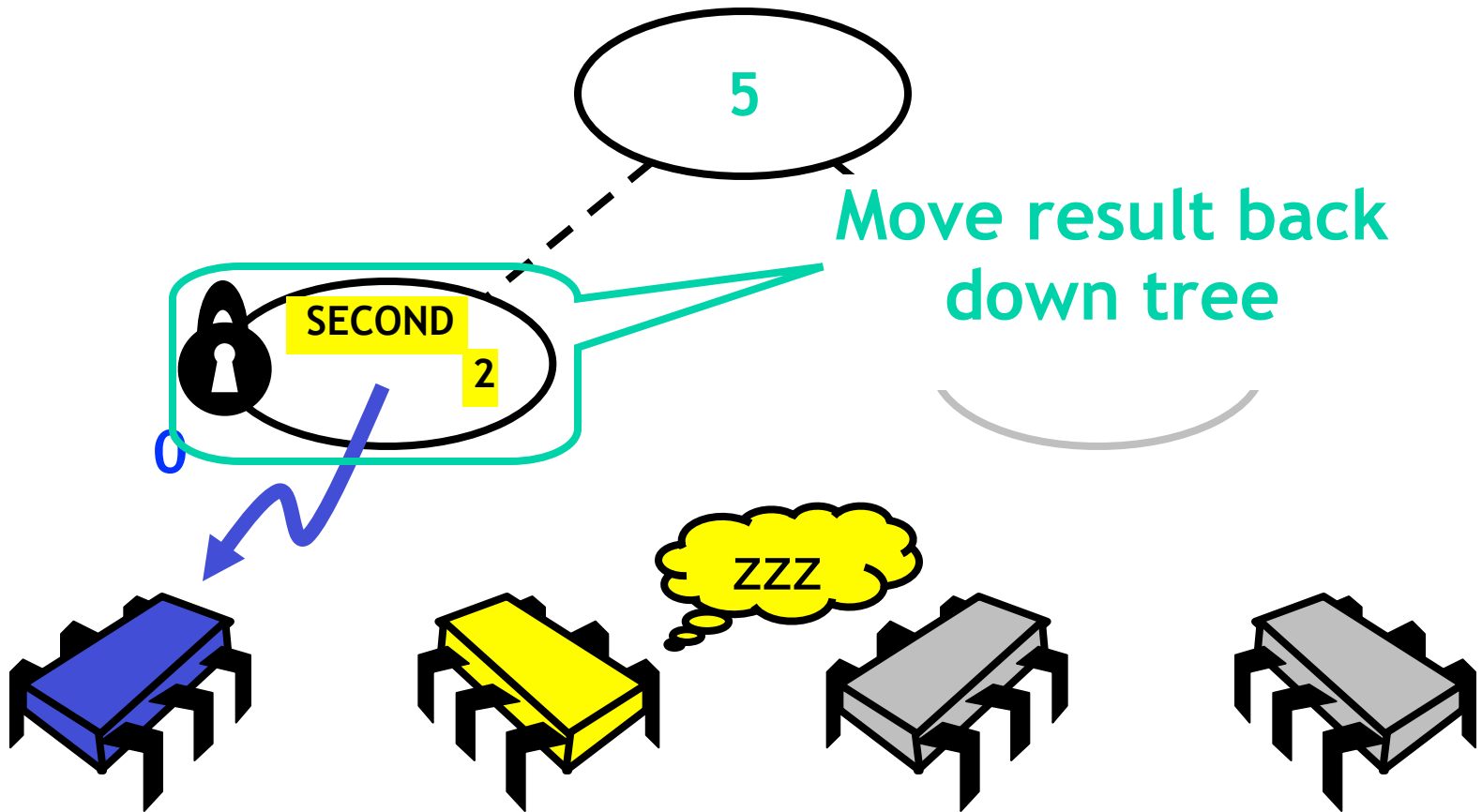
# Distribution Phase



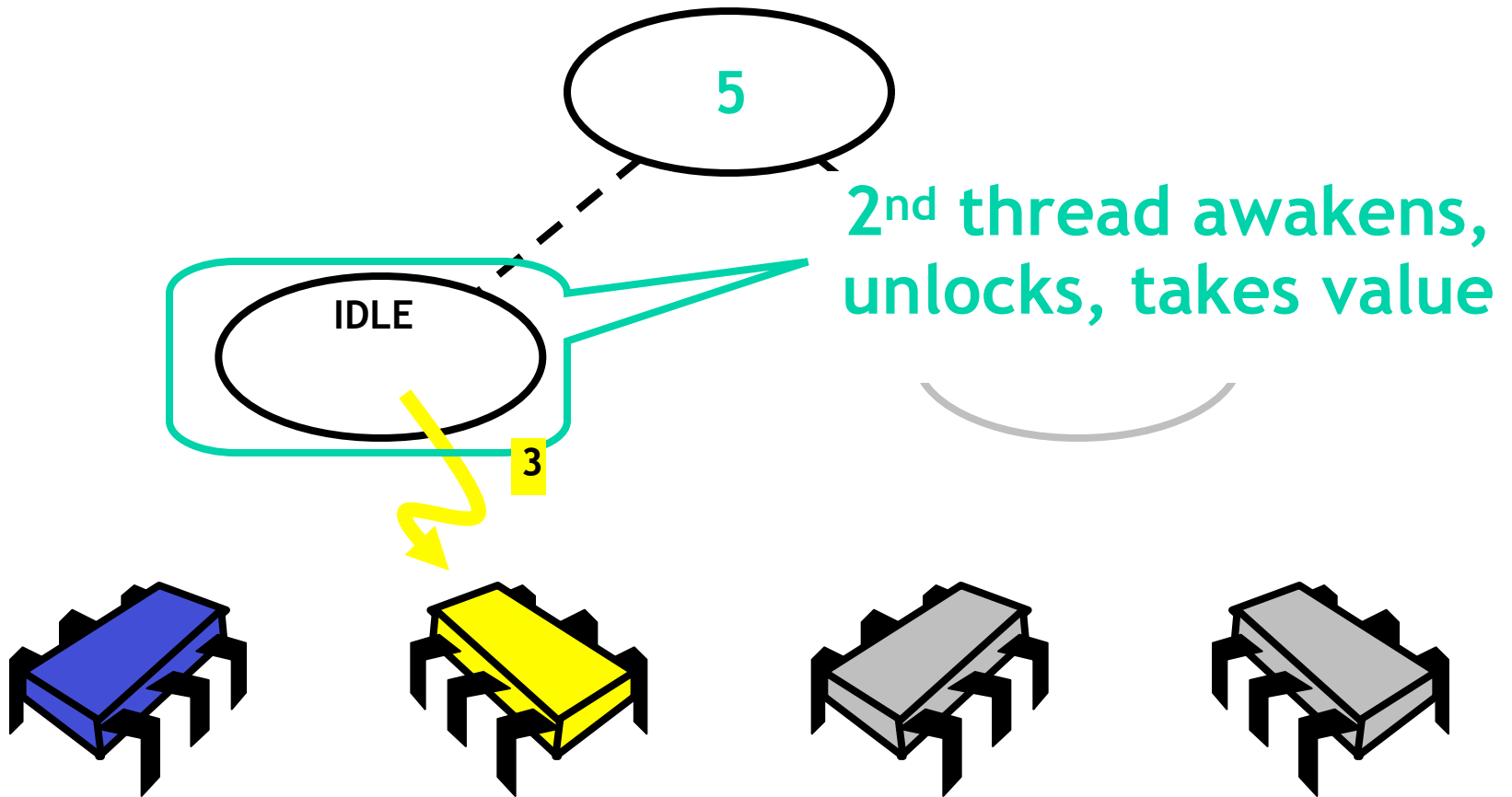
# Distribution Phase



# Distribution Phase



# Distribution Phase



# Distribution Phase Navigation

```
while (!stack.empty()) {  
    node = stack.pop();  
    node.distribute(prior);  
}  
return prior;
```

# Distribution Phase Navigation

```
while (!stack.empty()) {  
    node = stack.pop();  
    node.distribute(prior);  
}  
return prior;
```

**Traverse path in  
reverse order**

# Distribution Phase Navigation

```
while (!stack.empty()) {  
    node = stack.pop();  
    node.distribute(prior);  
}  
return prior;
```

**Distribute results to  
waiting 2<sup>nd</sup> threads**



# Distribution Phase Navigation

```
while (!stack.empty()) {  
    node = stack.pop();  
    node.distribute(prior);  
}
```

return prior;

**Return result  
to caller**

# Distribution Phase

```
synchronized void distribute(int prior) {  
    switch (cStatus) {  
        case FIRST:  
            cStatus = CStatus.IDLE;  
            locked = false; notifyAll();  
            return;  
        case SECOND:  
            result = prior + firstValue;  
            cStatus = CStatus.DONE; notifyAll();  
            return;  
        default: ...  
    }  
}
```

# Distribution Phase

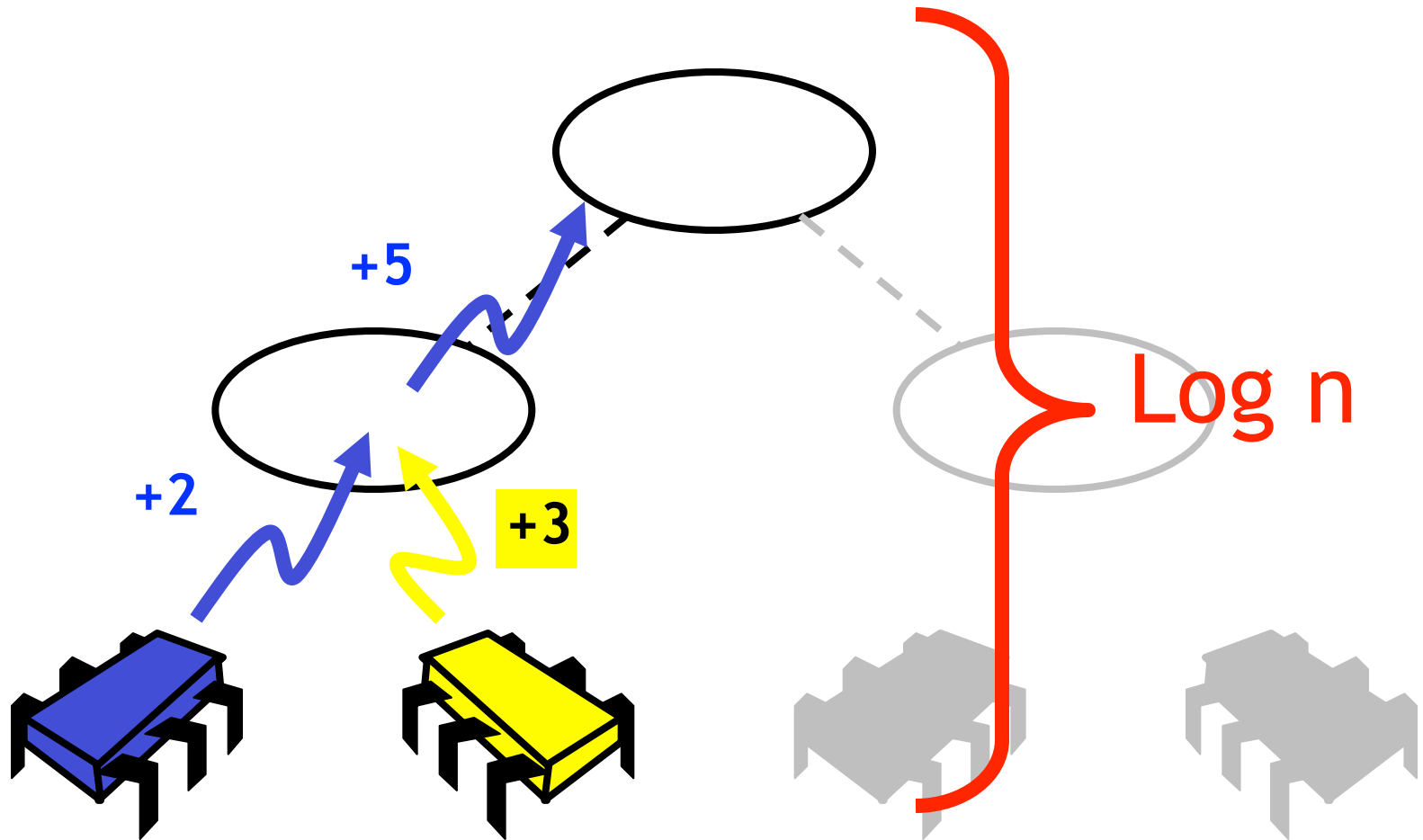
```
synchronized void distribute(int prior) {  
    switch (cStatus) {  
        case FIRST:  
            cStatus = CStatus.IDLE;  
            locked = false; notifyAll();  
            return;  
        case SECOND:  
            result = prior + firstValue;  
            cStatus = CStatus.DONE; notifyAll();  
            return;  
        default: ...  
    }  
}
```

**No combining, unlock  
node & reset**

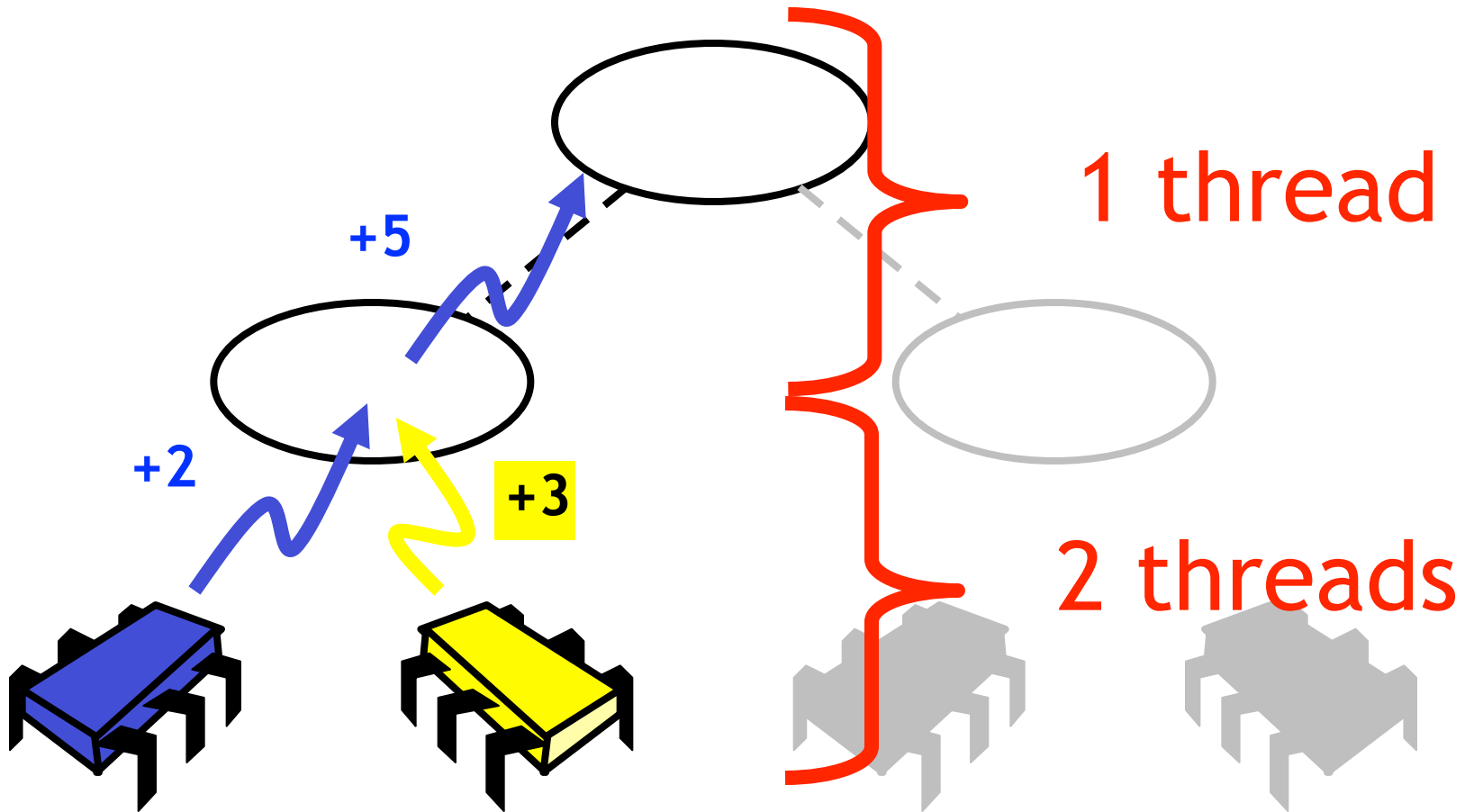
# Distribution Phase

```
synchronized void distribute(int prior) {  
    switch (cStatus) {  
        case FIRS: Notify 2nd thread that  
                   result is available  
            cStatus =  
            locked = 1  
            return;  
        case SECOND:  
            result = prior + firstValue;  
            cStatus = CStatus.DONE; notifyAll();  
            return;  
        default: ...
```

# Bad News: High Latency



# Good News: Real Parallelism



# Throughput Puzzles

- Ideal circumstances
  - All  $n$  threads move together, combine
  - $n$  increments in  $O(\log n)$  time
- Worst circumstances
  - All  $n$  threads slightly skewed, locked out
  - $n$  increments in  $O(n \cdot \log n)$  time

# Index Distribution Benchmark

```
void indexBench(int iters, int work) {  
    while (int i < iters) {  
        i = r.getAndIncrement();  
        Thread.sleep(random() % work);  
    }  
}
```



# Index Distribution Benchmark

```
void indexBench(int iters, int work) {  
    while (int i < iters) {  
        i = r.getAndIncrement();  
        Thread.sleep(random() % work);  
    }  
}
```

**How many iterations**

# Index Distribution Benchmark

```
void indexBench(int iters, int work) {  
    while (int i < iters) {  
        i = r.getAndIncrement(),  
        Thread.sleep(random() % work);  
    }  
}
```

**Expected time between  
incrementing counter**

# Index Distribution Benchmark

```
void indexBench(int iters, int work) {  
    while (int i < iters) {  
        i = r.getAndIncrement();  
        Thread.sleep(random() % work);  
    }  
}
```

**Take a number**

# Index Distribution Benchmark

```
void indexBench(int iters, int work) {  
  while (int i < iters) {  
    i = r.getAndIncrement();  
    Thread.sleep(random() % work);  
  }  
}
```

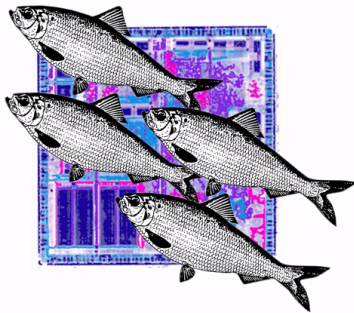
**Pretend to work  
(more work, less concurrency)**

# Performance Benchmarks

- Alewife

- NUMA architecture
- Simulated

MIT - ALEWIFE



- **Throughput:**

- average number of **inc** operations in 1 million cycle period.

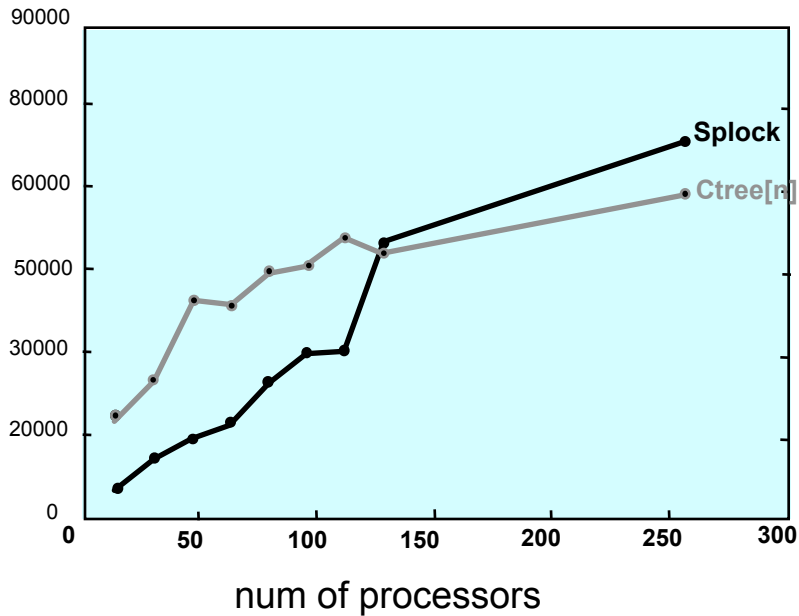
- **Latency:**

- average number of simulator cycles per **inc** operation.

# Performance

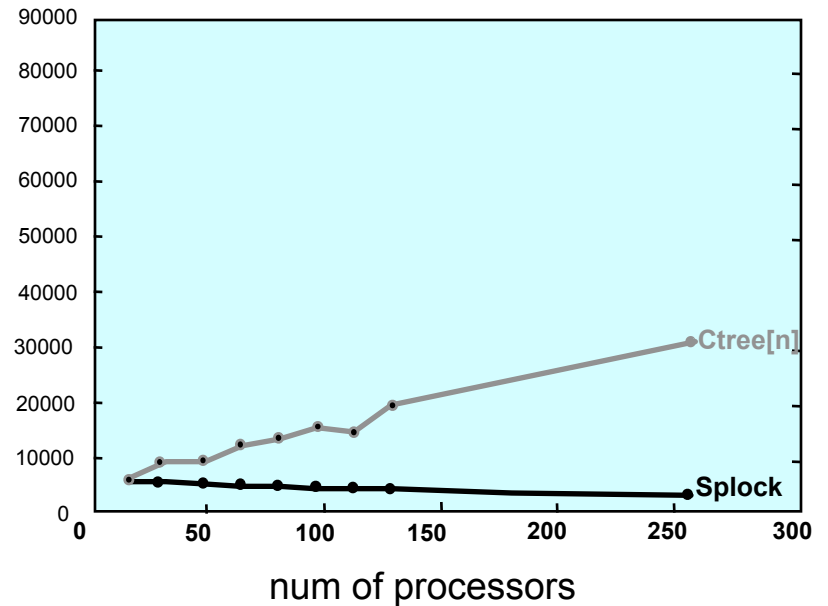
cycles  
per  
operation

Latency:



operations  
per million  
cycles

Throughput:

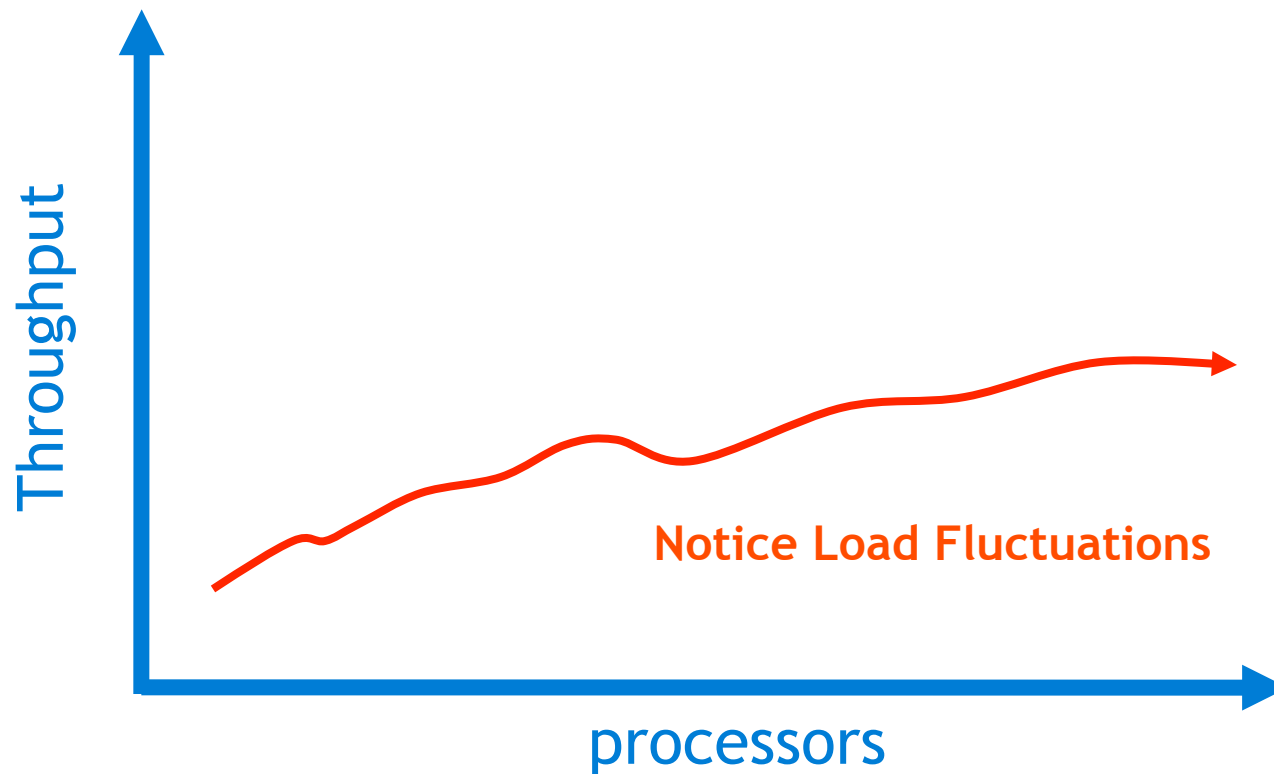


**work = 0**

# The Combining Paradigm

- Implements any RMW operation
- When tree is loaded
  - Takes  $2 \log n$  steps
  - for  $n$  requests
- Very sensitive to load fluctuations:
  - if the arrival rates drop
  - the combining rates drop
  - overall performance deteriorates!

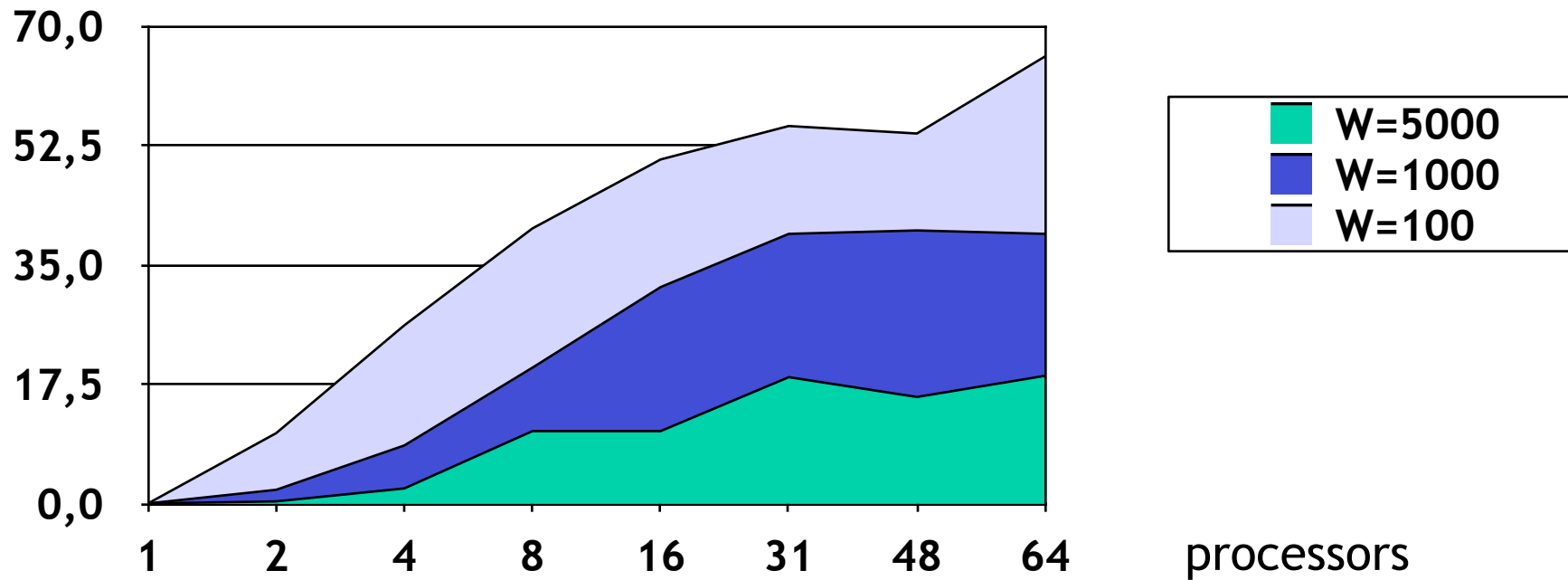
# Combining Load Sensitivity





# Combining Rate vs Work

Throughput



# Conclusions

- Combining Trees
  - Work well under high contention
  - Sensitive to load fluctuations
  - Can be used for `getAndMumble()` ops