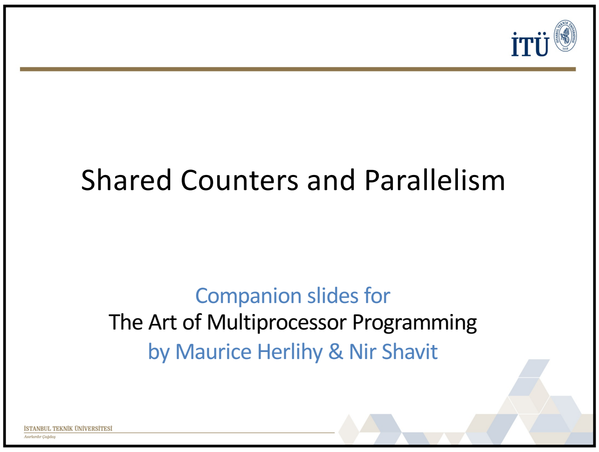
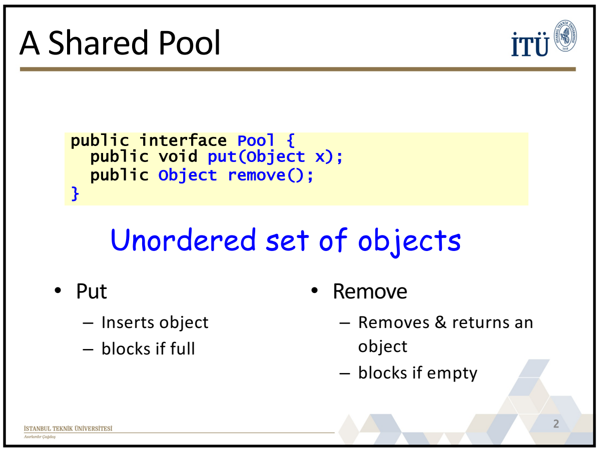


Shared Counters and Parallelism

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

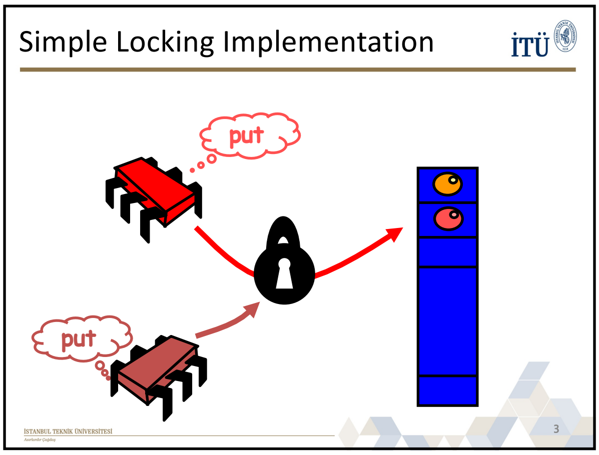


A Shared Pool

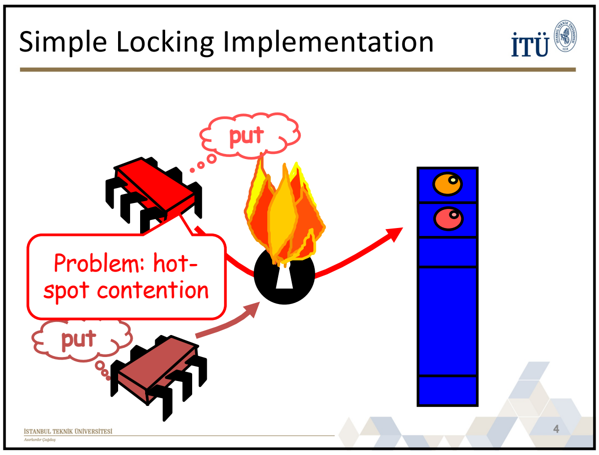
```
public interface Pool {
  public void put(Object x);
  public Object remove();
}
```

Unordered set of objects

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

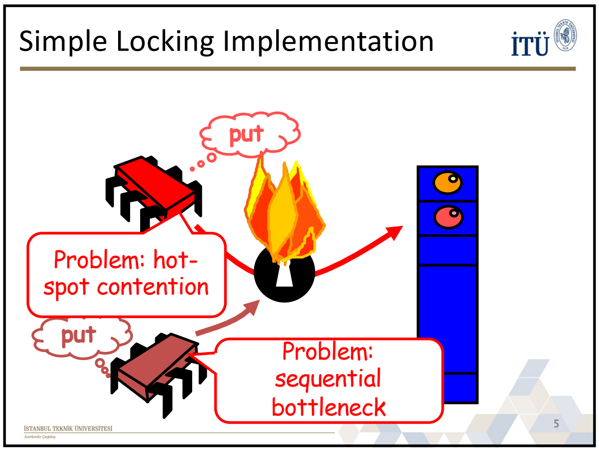


Simple Locking Implementation



Simple Locking Implementation

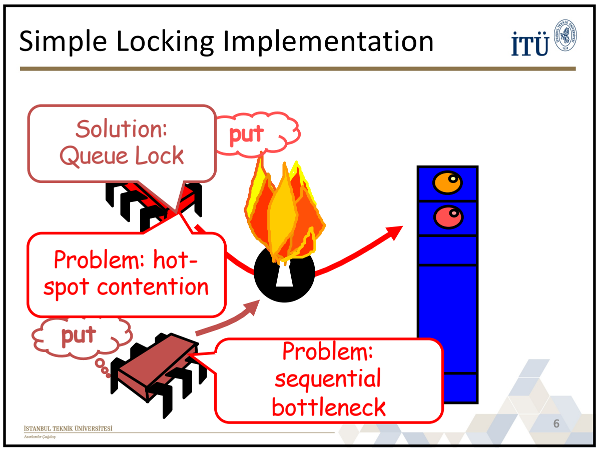
Problem: hot-spot contention



Simple Locking Implementation

Problem: hot-spot contention

Problem: sequential bottleneck

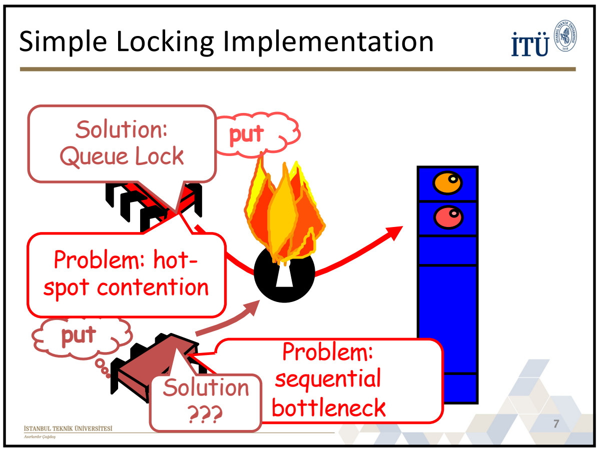


Simple Locking Implementation

Solution: Queue Lock

Problem: hot-spot contention

Problem: sequential bottleneck



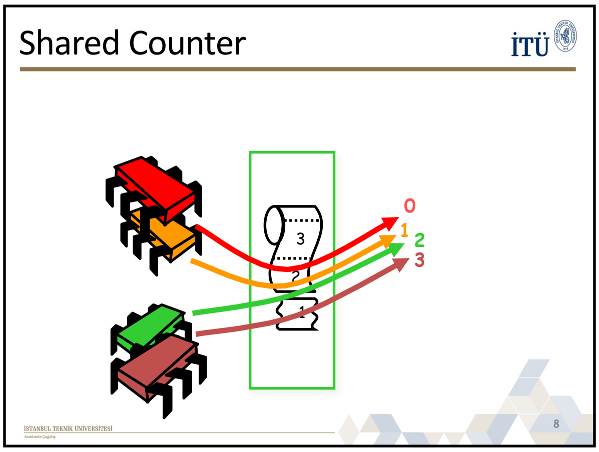
Simple Locking Implementation

Solution: Queue Lock

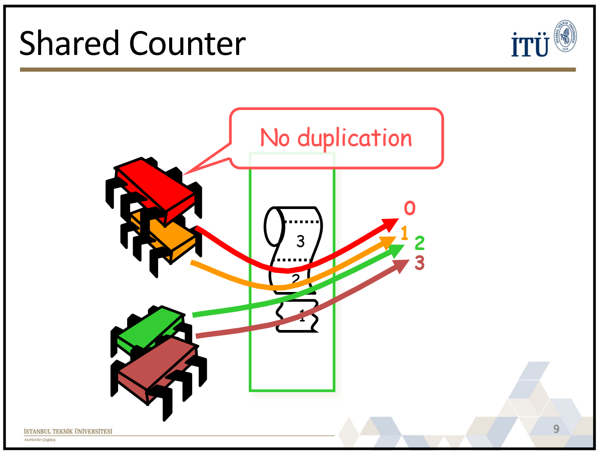
Problem: hot-spot contention

Problem: sequential bottleneck

Solution ???

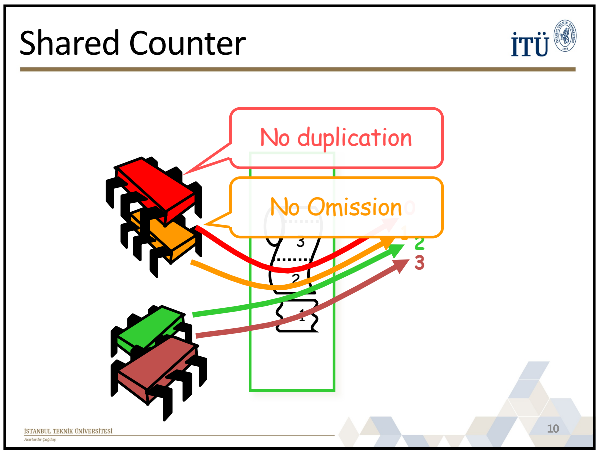


Shared Counter



Shared Counter

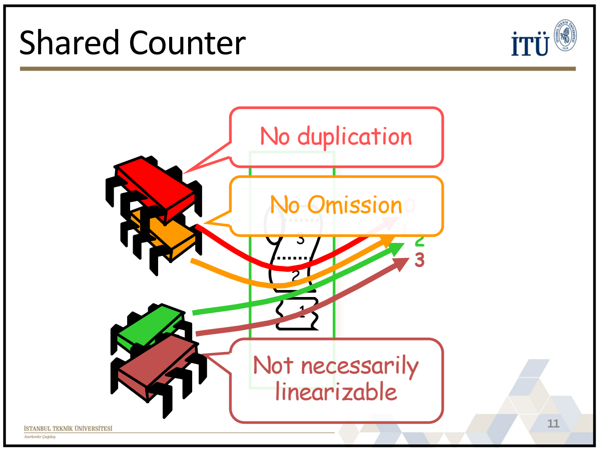
No duplication



Shared Counter

No duplication

No Omission

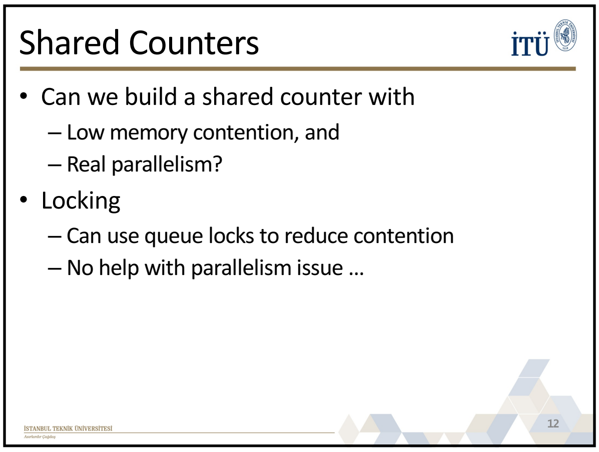


Shared Counter

No duplication

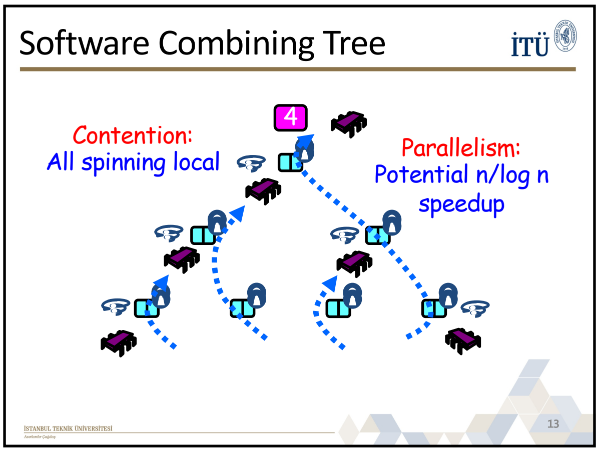
No Omission

Not necessarily linearizable



Shared Counters

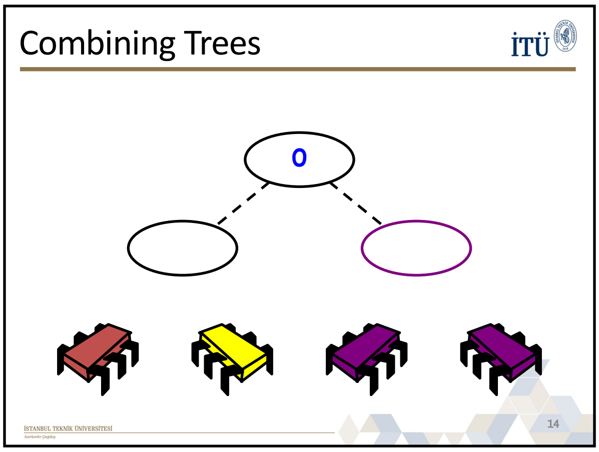
- Can we build a shared counter with
  - Low memory contention, and
  - Real parallelism?
- Locking
  - Can use queue locks to reduce contention
  - No help with parallelism issue …



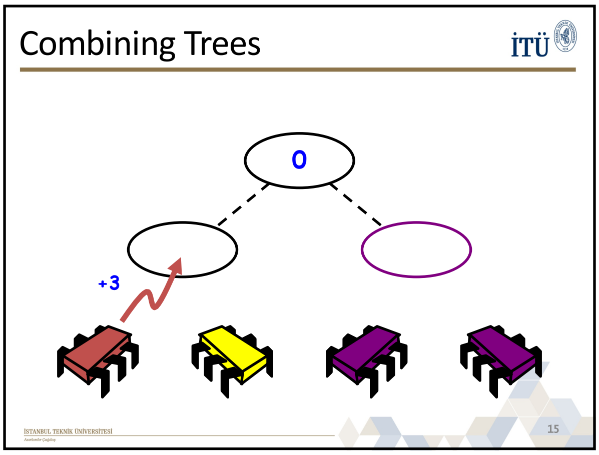
Software Combining Tree

Contention: All spinning local

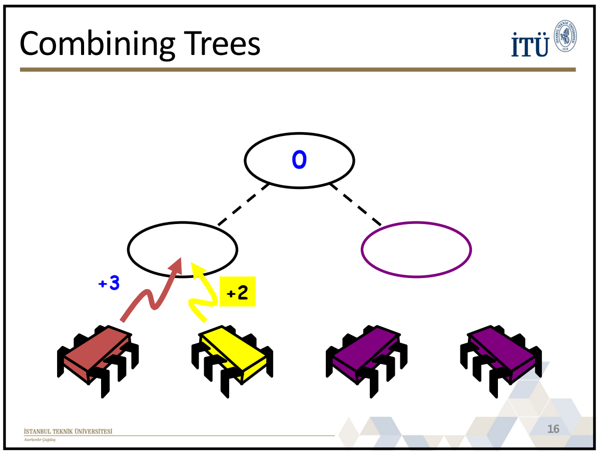
Parallelism: Potential n/log n speedup



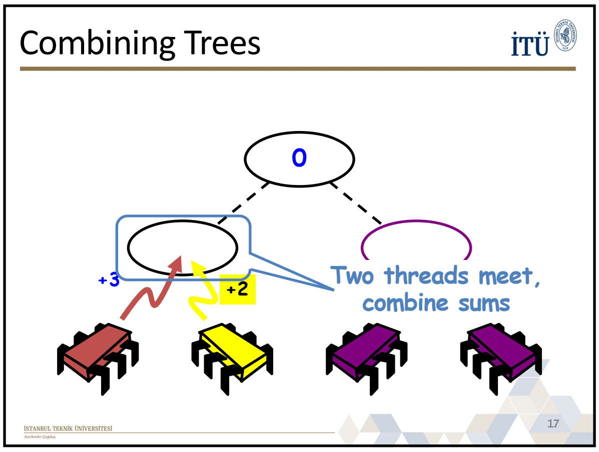
Combining Trees



Combining Trees

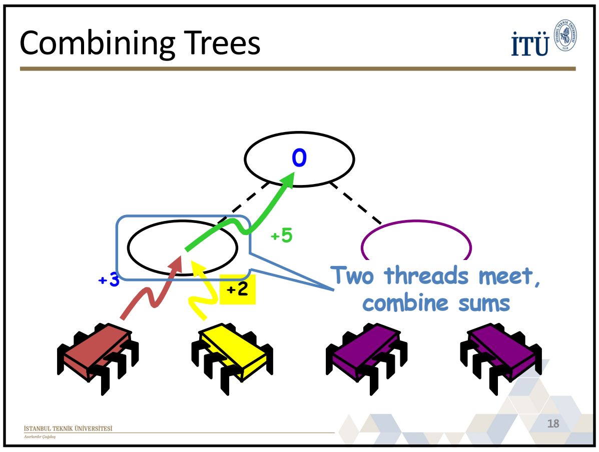


Combining Trees



Combining Trees

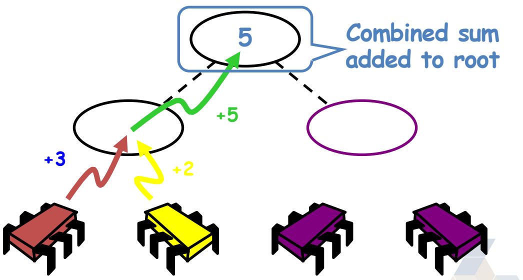
Two threads meet, combine sums



Combining Trees

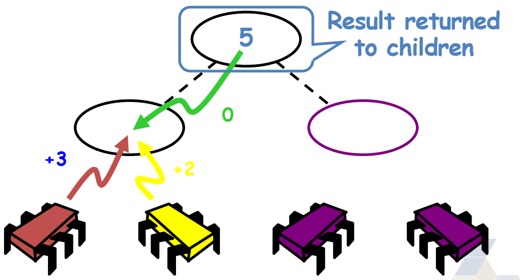
Two threads meet, combine sums

Combining Trees



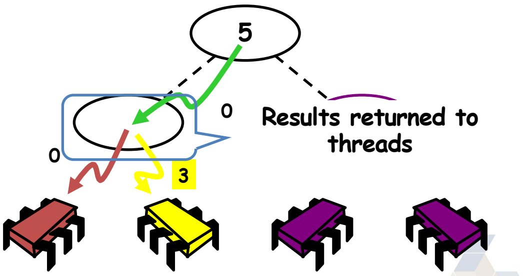
Combined sum added to root

Combining Trees



Result returned to children

Combining Trees



Results returned to threads

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 …

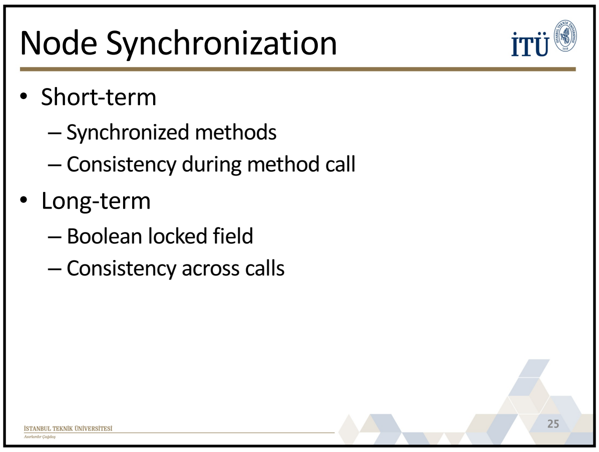
Combining Status

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

getAndIncrement

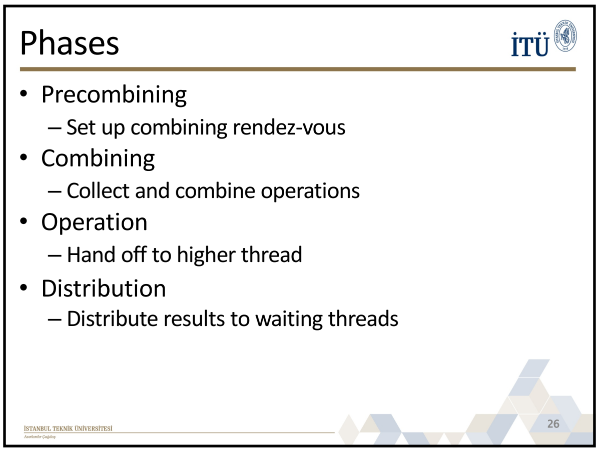
```
public int getAndIncrement() {
  Stack<Node> stack = new Stack<Node>();
  Node myLeaf = leaf[ThreadID.get()/2];

  // precombining phase
  // combining phase
  // operation phase
  // distribution phase
}
```



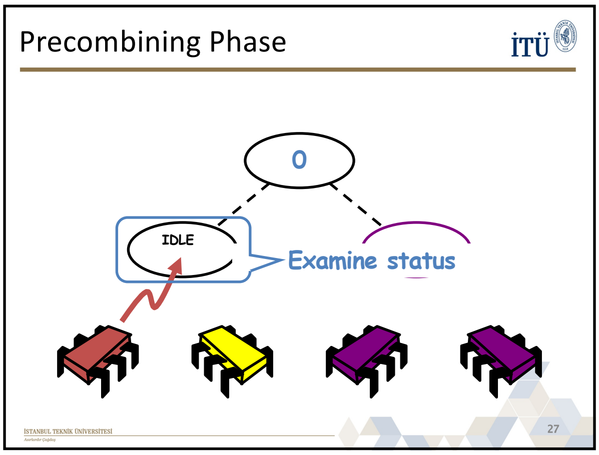
Node Synchronization

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



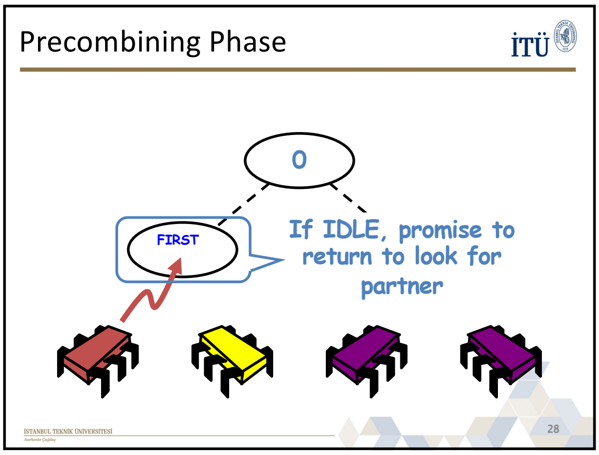
Phases

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



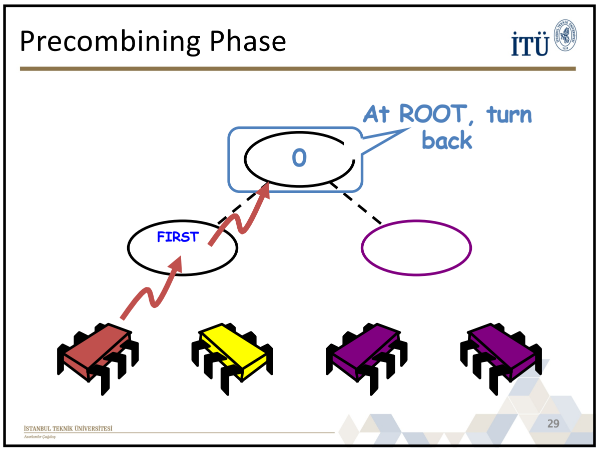
Precombining Phase

Examine status



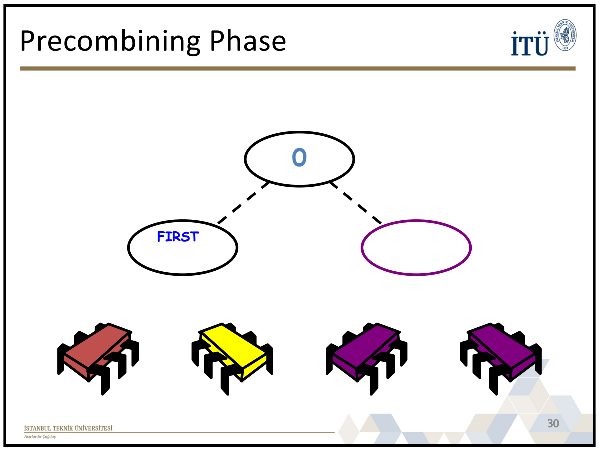
Precombining Phase

If IDLE, promise to return to look for partner

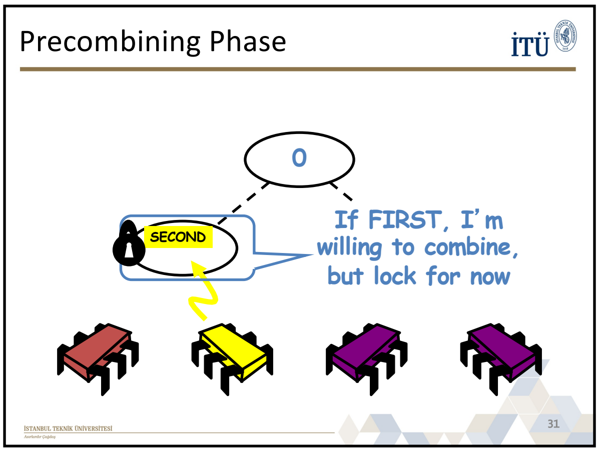


Precombining Phase

At ROOT, turn back



Precombining Phase

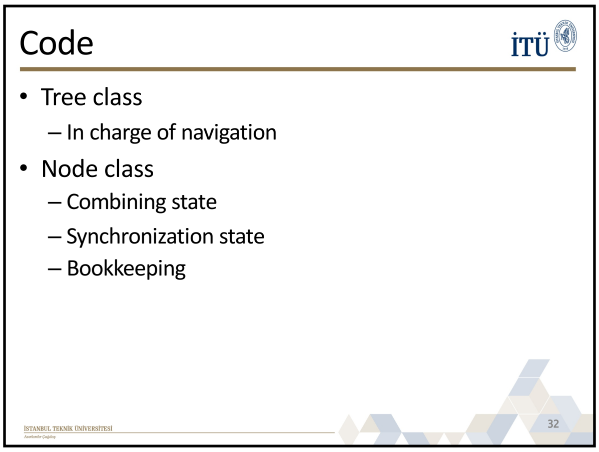


Precombining Phase

0

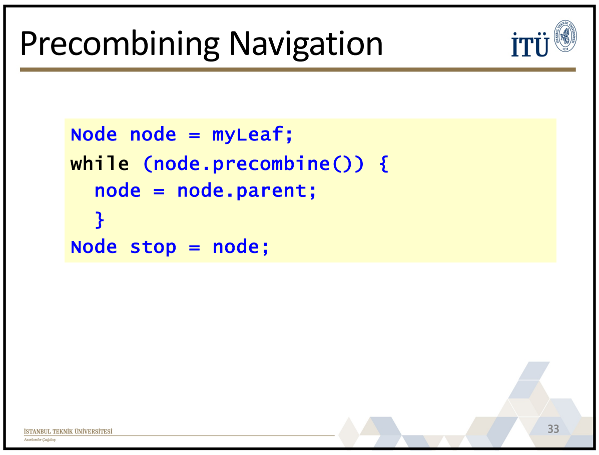
SECOND

If FIRST, I'm willing to combine, but lock for now



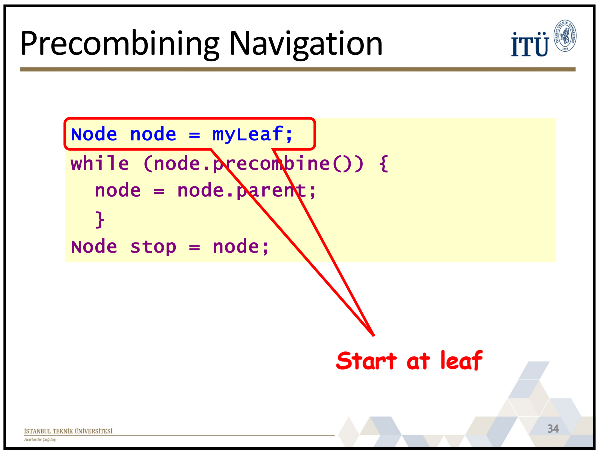
Code

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



Precombining Navigation

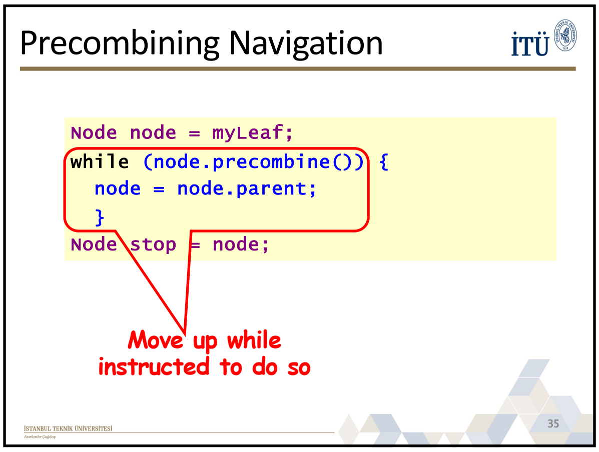
```
Node node = myLeaf;
while (node.precombine()) {
  node = node.parent;
  }
Node stop = node;
```



Precombining Navigation

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

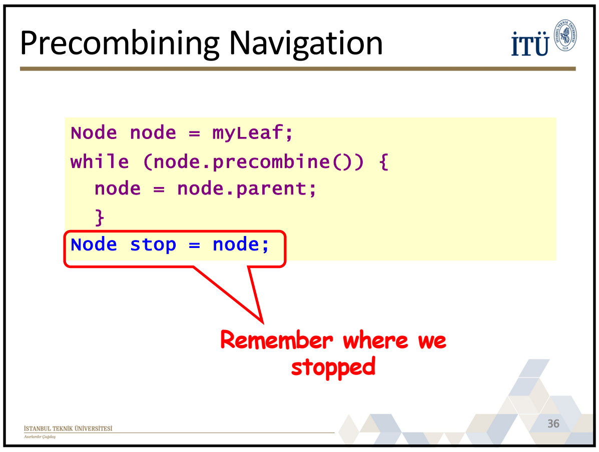
Start at leaf



Precombining Navigation

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

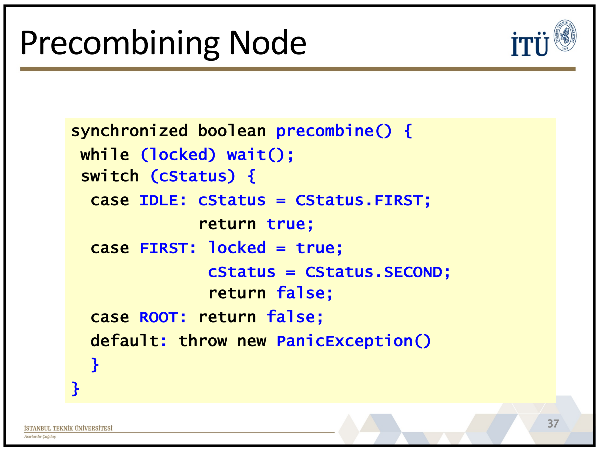
Move up while instructed to do so



Precombining Navigation

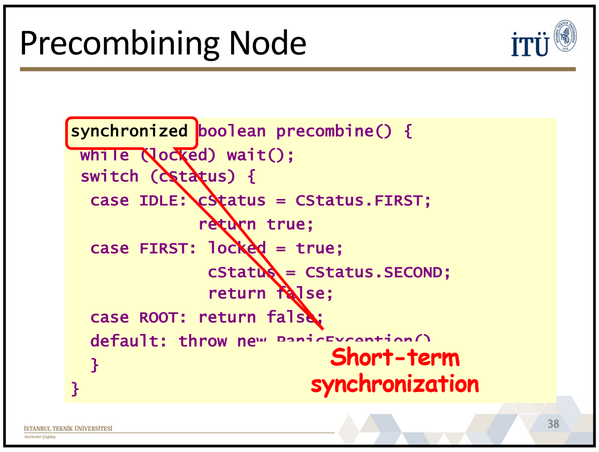
```
Node node = myLeaf;
while (node.precombine()) {
  node = node.parent;
  }
Node stop = node;
```

Remember where we stopped



Precombining Node

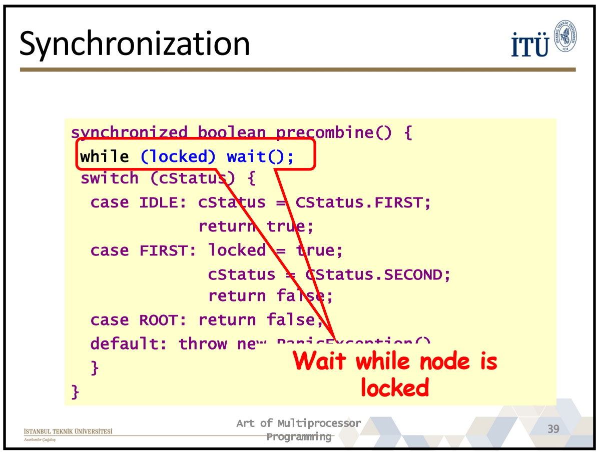
```
synchronized boolean precombine() {
 while (locked) wait();
 switch (cStatus) {
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()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



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()
  }
}
```

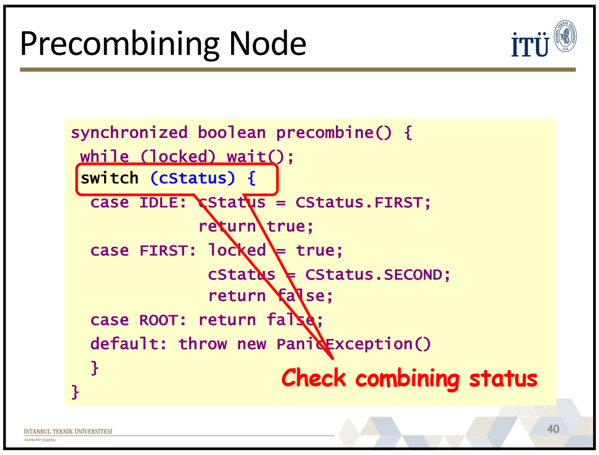
Short-term synchronization



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()
  }
}
```

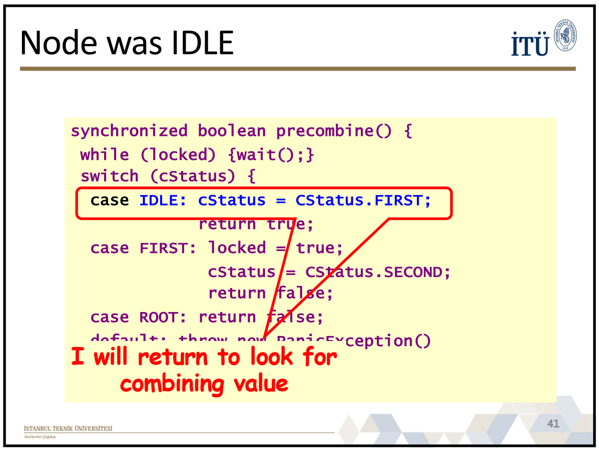
Wait while node is locked



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()
  }
}
```

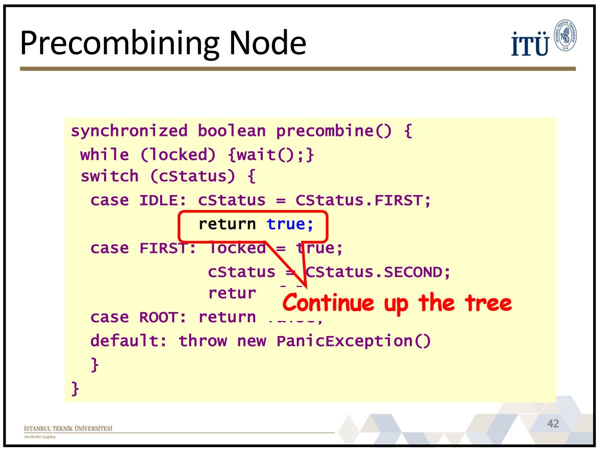
Check combining status



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()
```

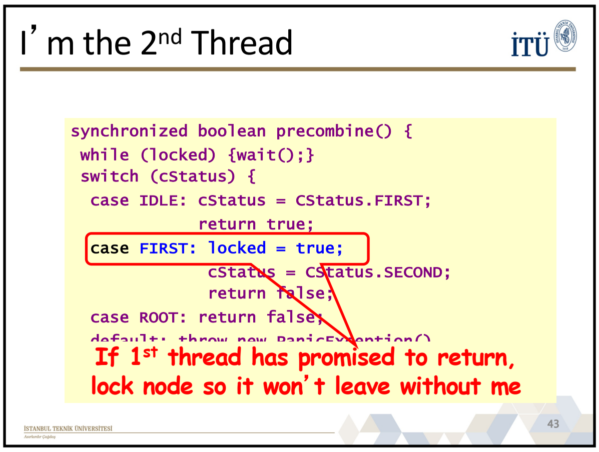
I will return to look for combining value



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 false;
  default: throw new PanicException()
  }
}
```

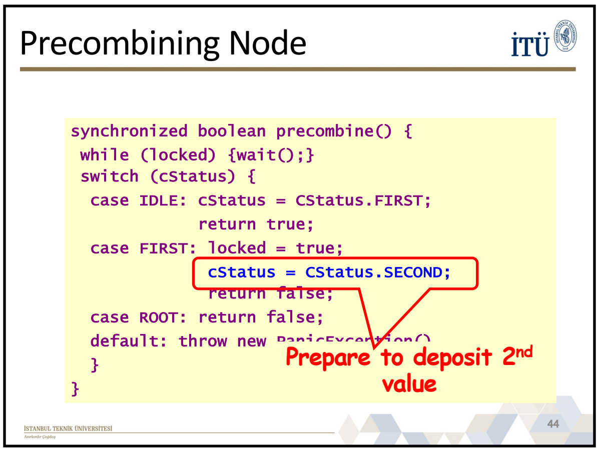
Continue up the tree



I’m the 2nd 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()
```

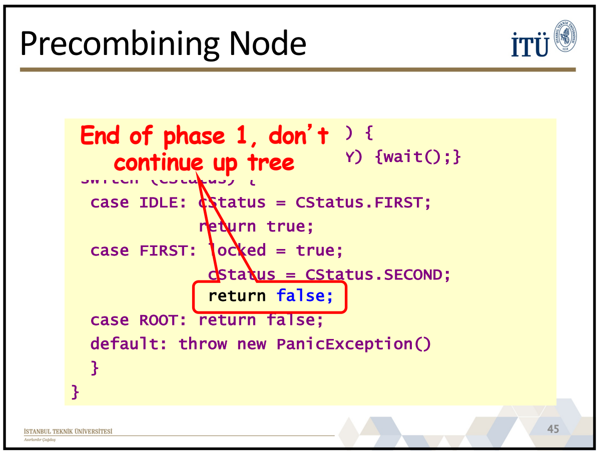
If 1st thread has promised to return, lock node so it won’t leave without me



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()
  }
}
```

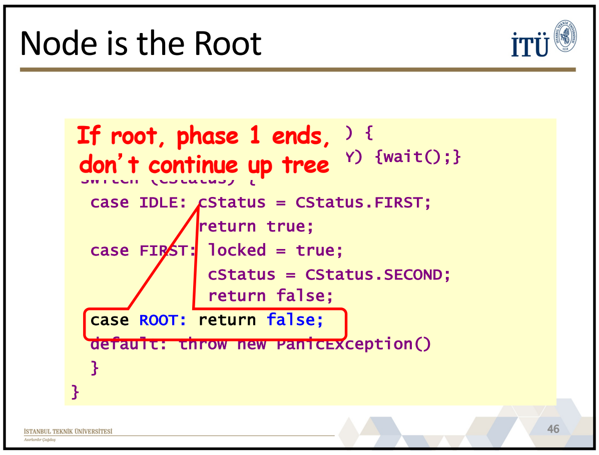
Prepare to deposit 2nd value



Precombining Node

End of phase 1, don’t continue up tree

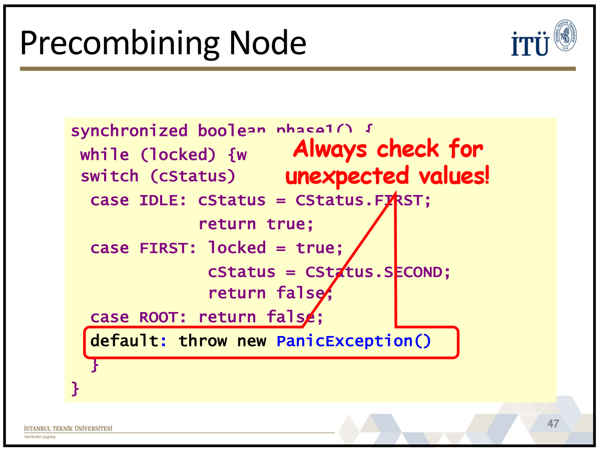
```
                        ) {
                        Y) {wait();}
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}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



Node is the Root

If root, phase 1 ends, don’t continue up tree

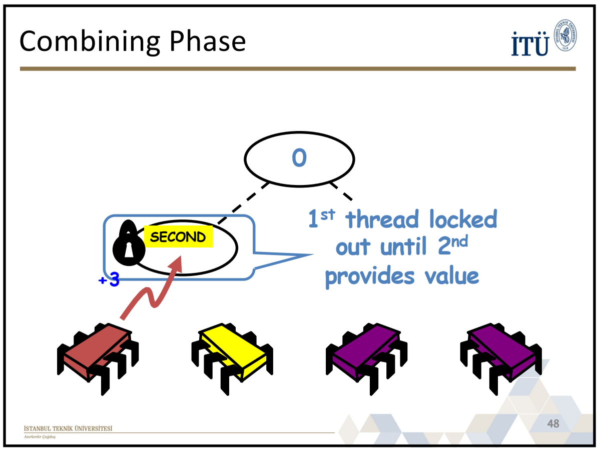
```
                        ) {
                        Y) {wait();}
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
  }
}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



Precombining Node

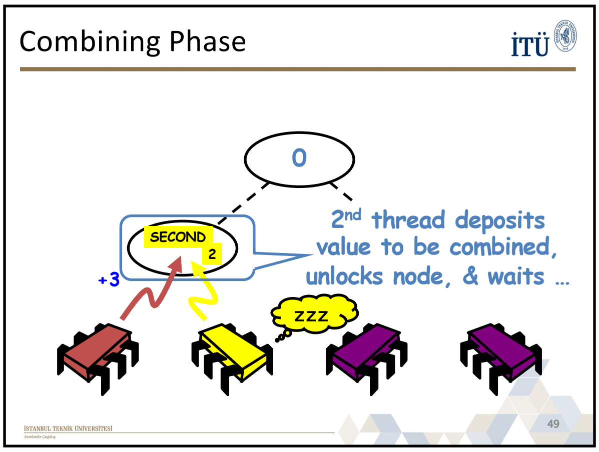
```
synchronized boolean phase1() {
 while (locked) {w
 switch (cStatus)
  case IDLE: cStatus = CStatus.FIRST;
             return true;
  case FIRST: locked = true;
              cStatus = CStatus.SECOND;
              return false;
  case ROOT: return false;
  default: throw new PanicException()
 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

Always check for unexpected values!



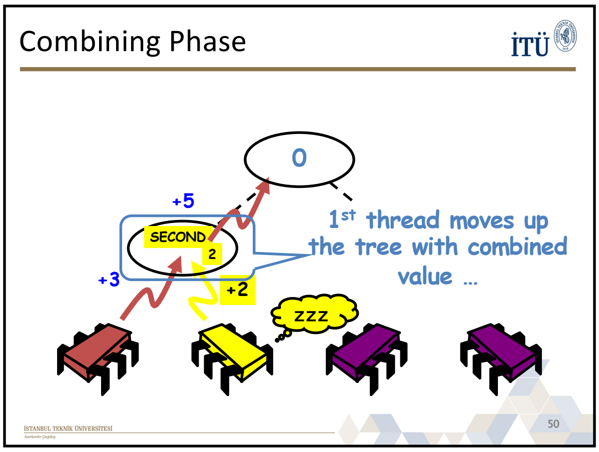
Combining Phase

1st thread locked out until 2nd provides value



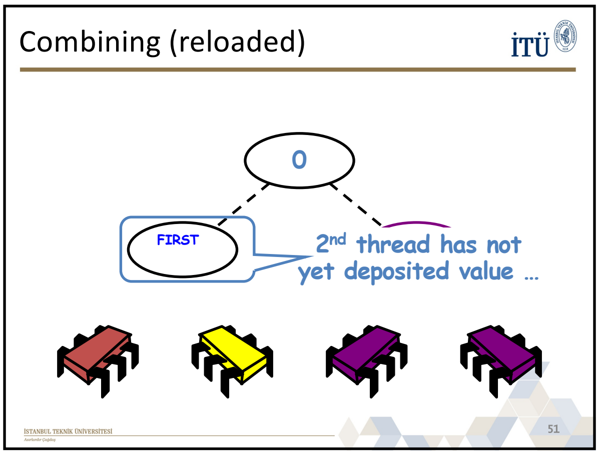
Combining Phase

2nd thread deposits value to be combined, unlocks node, & waits …



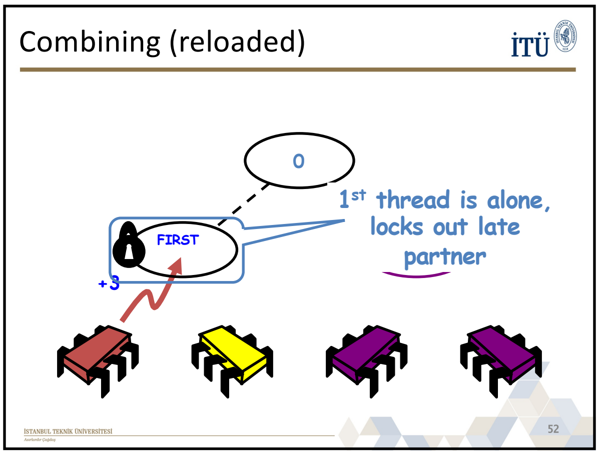
Combining Phase

1st thread moves up the tree with combined value …



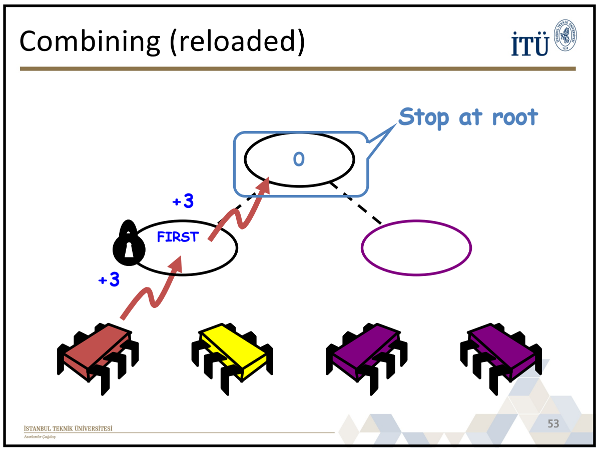
Combining (reloaded)

2nd thread has not yet deposited value …



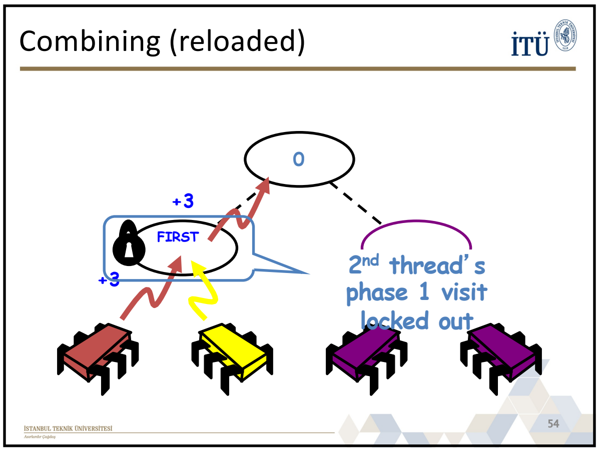
Combining (reloaded)

1st thread is alone, locks out late partner



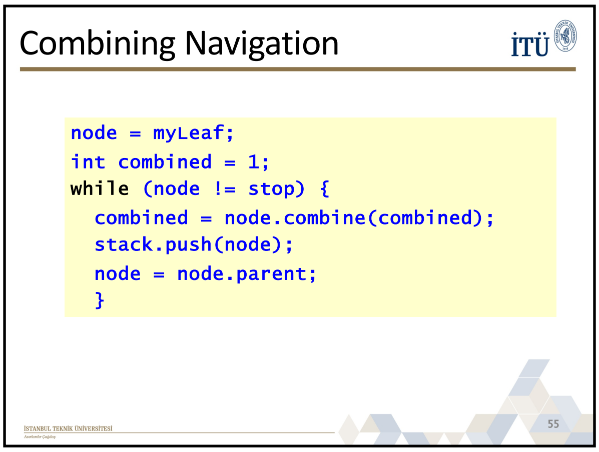
Combining (reloaded)

Stop at root



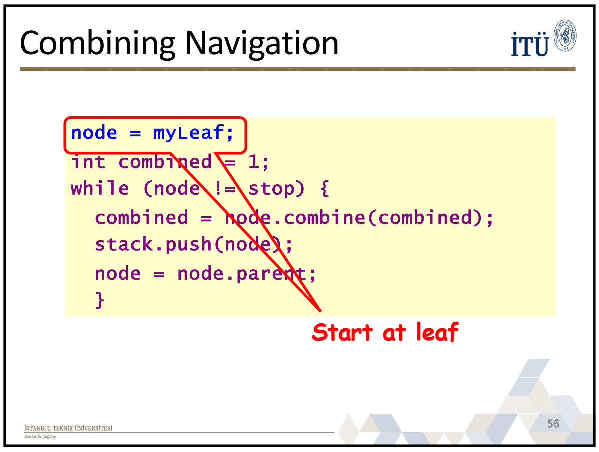
Combining (reloaded)

2nd thread's phase 1 visit locked out



Combining Navigation

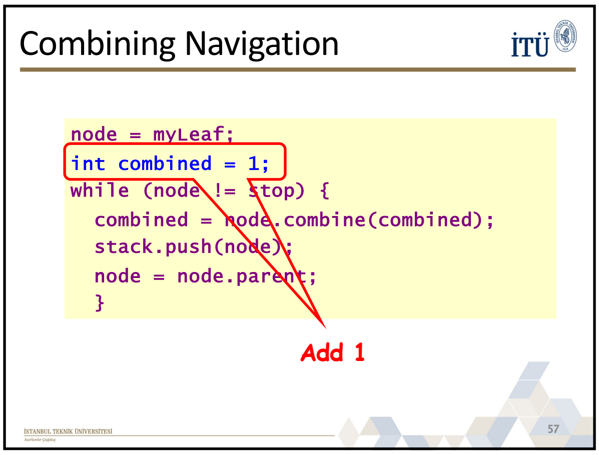
```
node = myLeaf;
int combined = 1;
while (node != stop) {
  combined = node.combine(combined);
  stack.push(node);
  node = node.parent;
  }
```



Combining Navigation

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

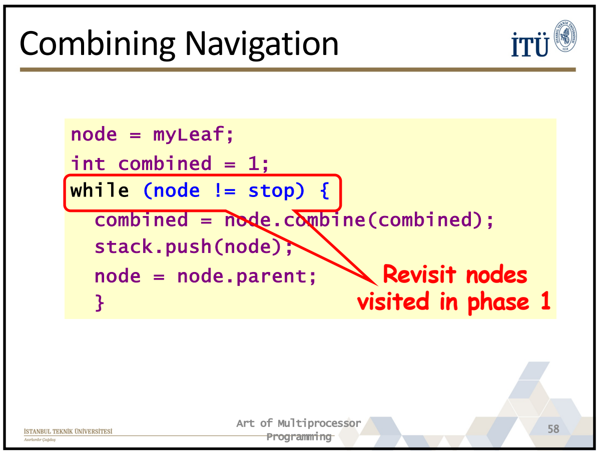
Start at leaf



Combining Navigation

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

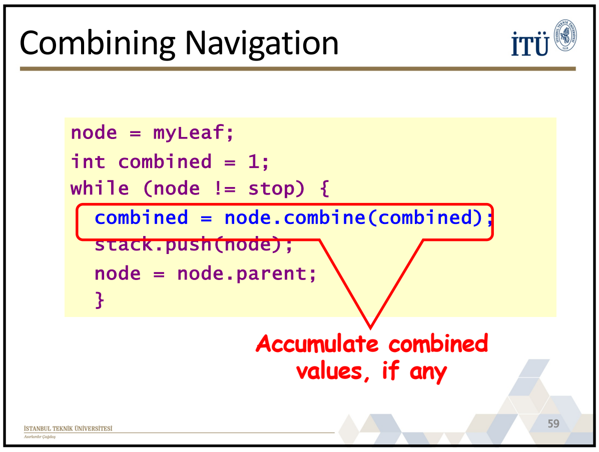
Add 1



Combining Navigation

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

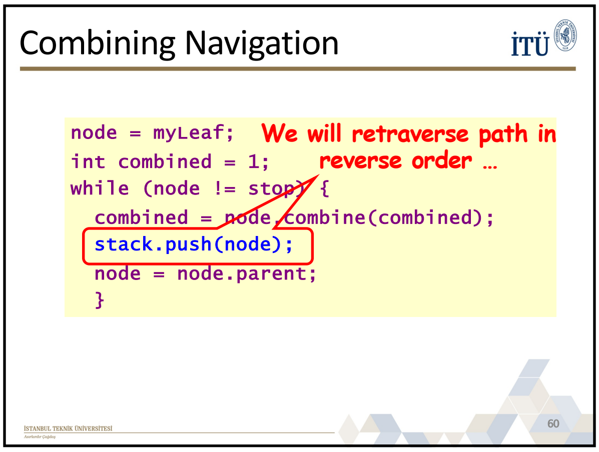
Revisit nodes visited in phase 1



Combining Navigation

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

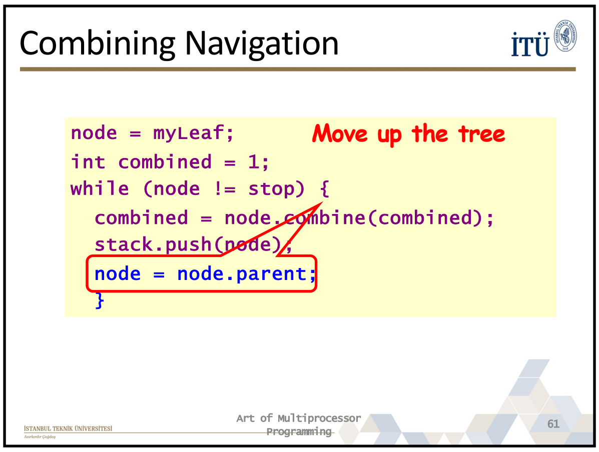
Accumulate combined values, if any



Combining Navigation

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

We will retraverse path in reverse order …



Combining Navigation

İTÜ

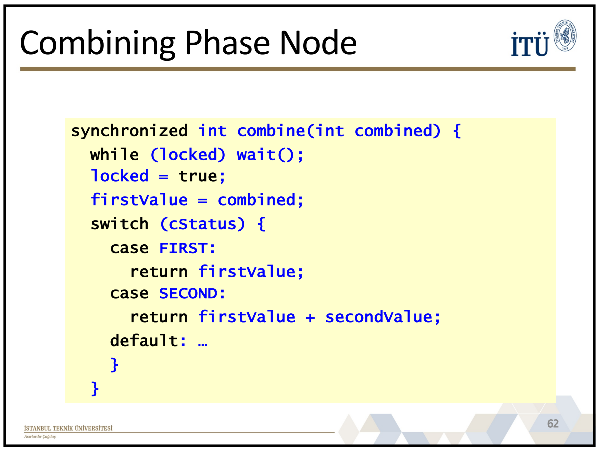
```
node = myLeaf;
int combined = 1;
while (node != stop) {
  combined = node.combine(combined);
  stack.push(node);
  node = node.parent;
 }
```

Move up the tree

İSTANBUL TEKNİK ÜNİVERSİTESİ

Art of Multiprocessor Programming

61



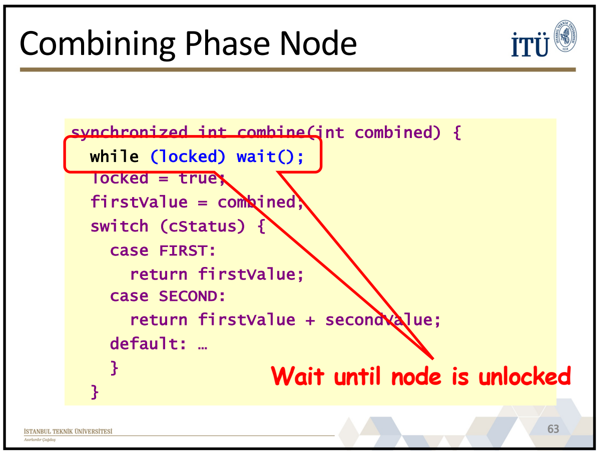
Combining Phase Node

İTÜ

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

İSTANBUL TEKNİK ÜNİVERSİTESİ

62



Combining Phase Node

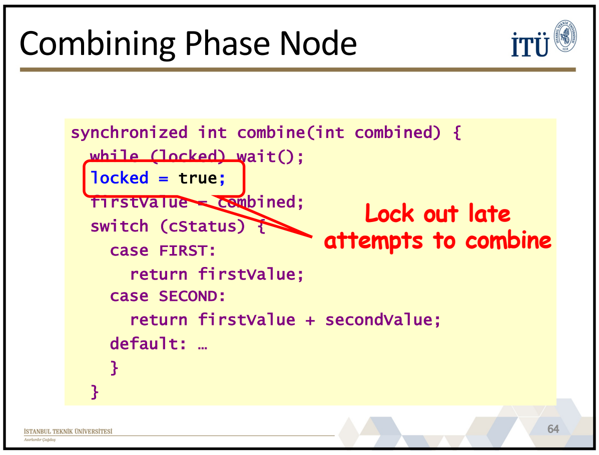
İTÜ

```
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

İSTANBUL TEKNİK ÜNİVERSİTESİ

63



Combining Phase Node

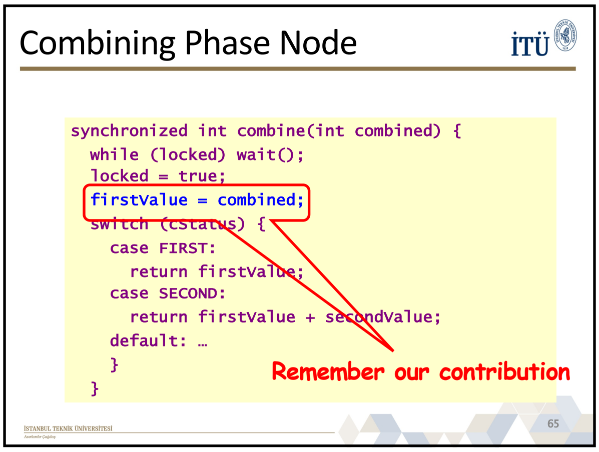
İTÜ

```
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

İSTANBUL TEKNİK ÜNİVERSİTESİ

64



Combining Phase Node

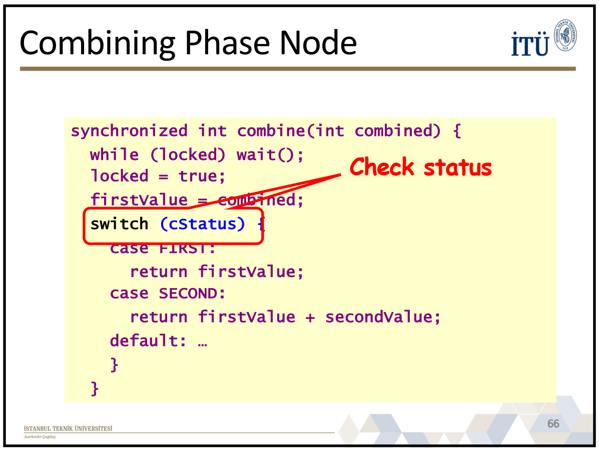
İTÜ

```
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

İSTANBUL TEKNİK ÜNİVERSİTESİ

65



Combining Phase Node

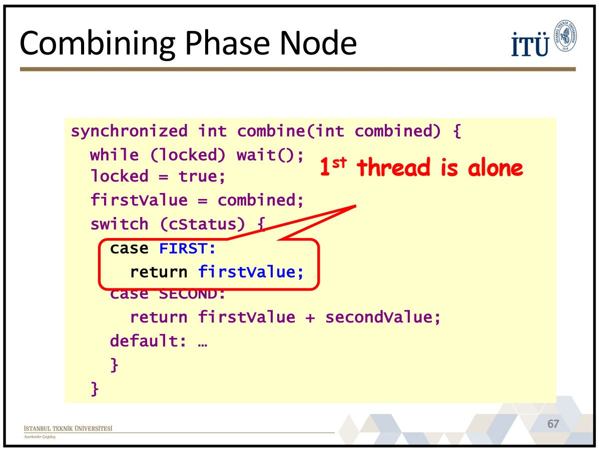
İTÜ

```
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

İSTANBUL TEKNİK ÜNİVERSİTESİ

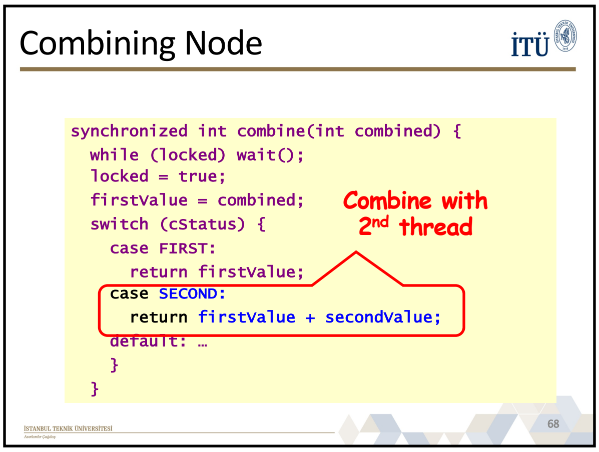
66



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: …
    }
  }
```

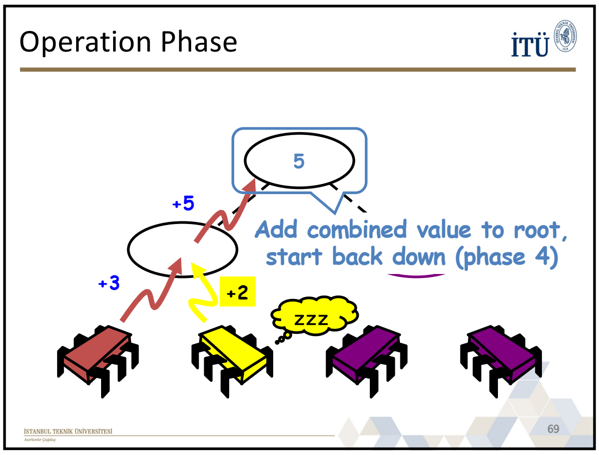
1st thread is alone



Combining Node

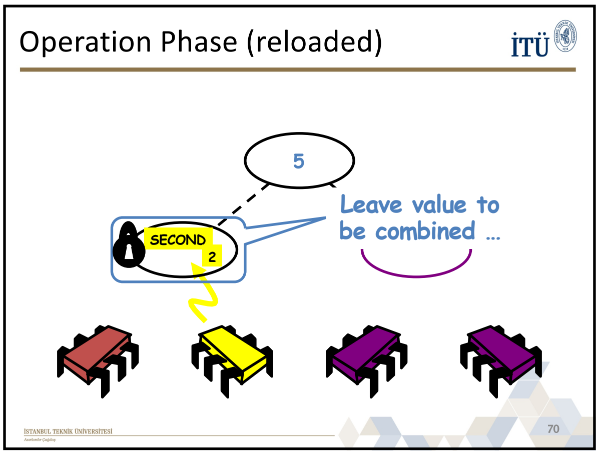
```
synchronized int combine(int combined) {
  while (locked) wait();
  locked = true;
  firstValue = combined;
  switch (cStatus) {
    case FIRST:
      return firstValue;
    case SECOND:
      return firstValue + secondValue;
    default: …
    }
  }
```

Combine with 2nd thread



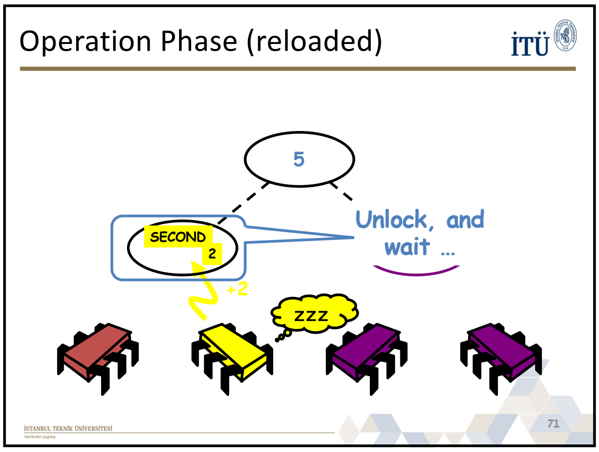
Operation Phase

Add combined value to root, start back down (phase 4)



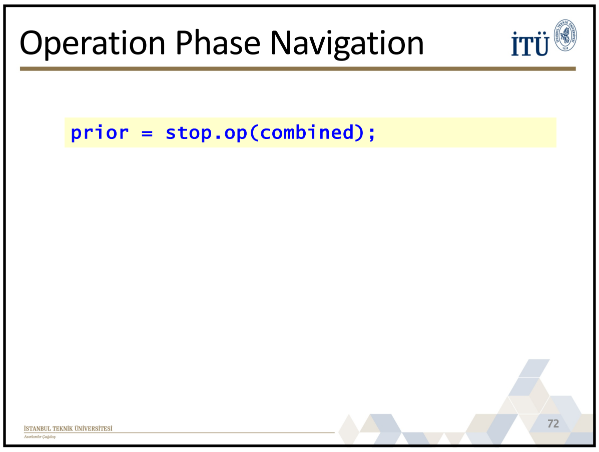
Operation Phase (reloaded)

Leave value to be combined …



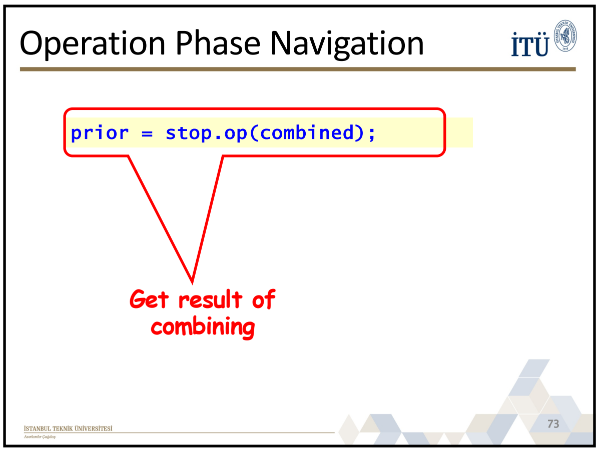
Operation Phase (reloaded)

Unlock, and wait …



Operation Phase Navigation

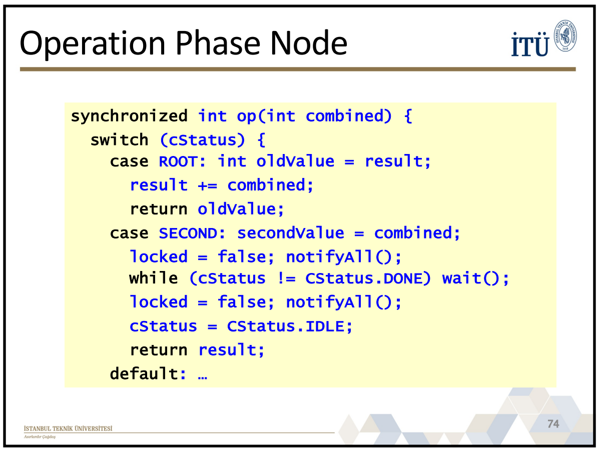
```
prior = stop.op(combined);
```



Operation Phase Navigation

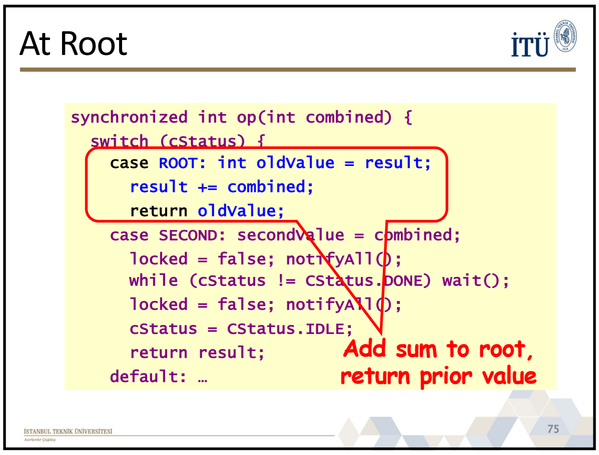
```
prior = stop.op(combined);
```

Get result of combining



Operation Phase Node

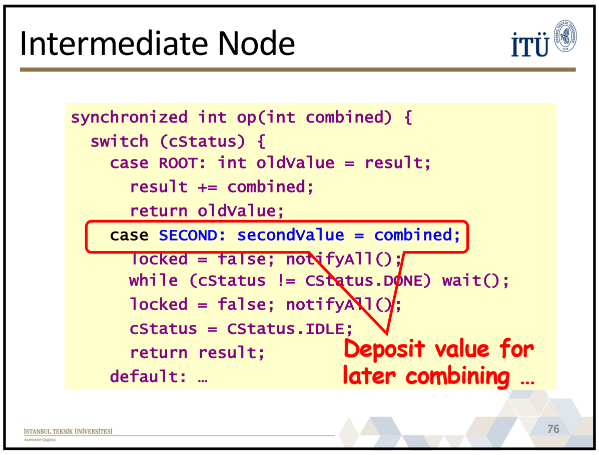
```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int oldValue = result;
      result += combined;
      return oldValue;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.DONE) wait();
      locked = false; notifyAll();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

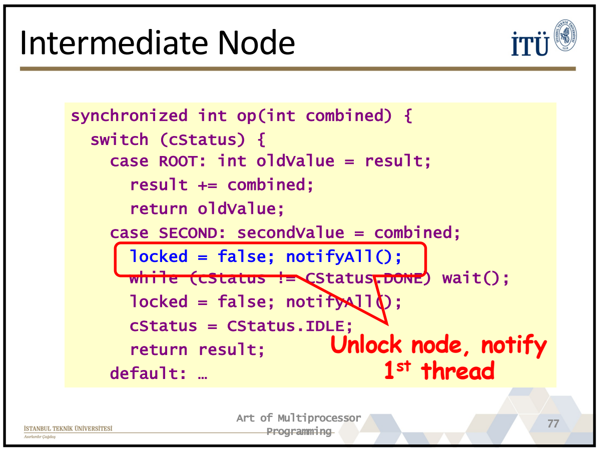
Add sum to root, return prior value



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: …
```

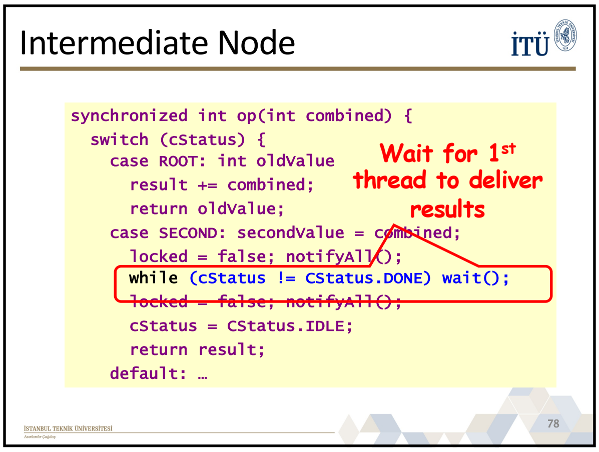
Deposit value for later combining …



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: …
```

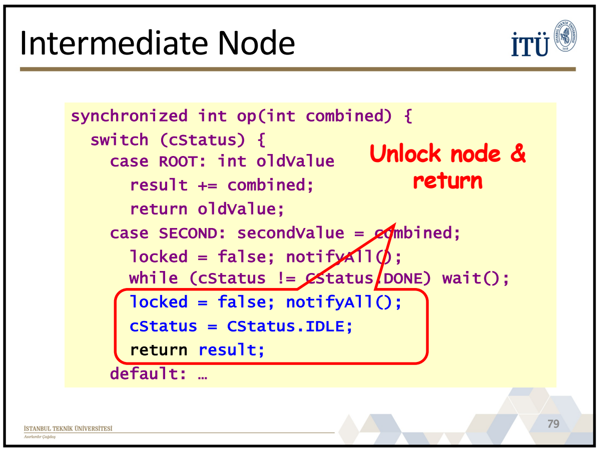
Unlock node, notify 1st thread



Intermediate Node

```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int oldValue
      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: …
```

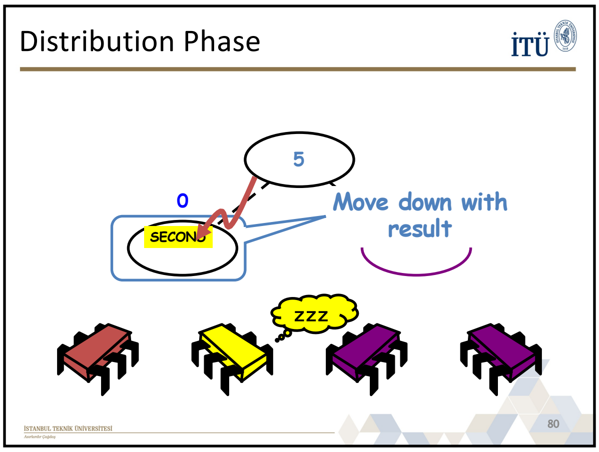
Wait for 1st thread to deliver results



Intermediate Node

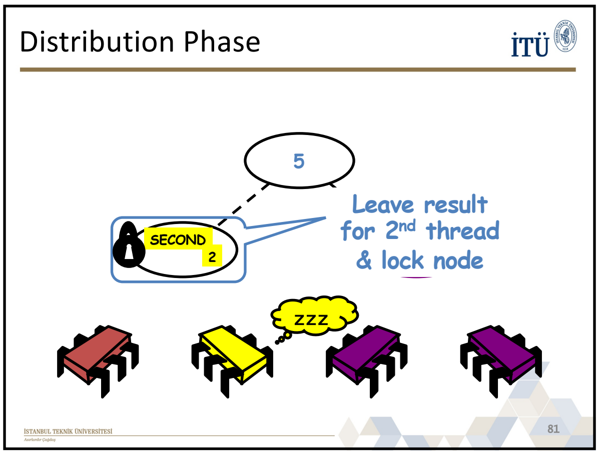
```
synchronized int op(int combined) {
  switch (cStatus) {
    case ROOT: int oldValue
      result += combined;
      return oldValue;
    case SECOND: secondValue = combined;
      locked = false; notifyAll();
      while (cStatus != CStatus.DONE) wait();
      locked = false; notifyAll();
     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

Unlock node & return



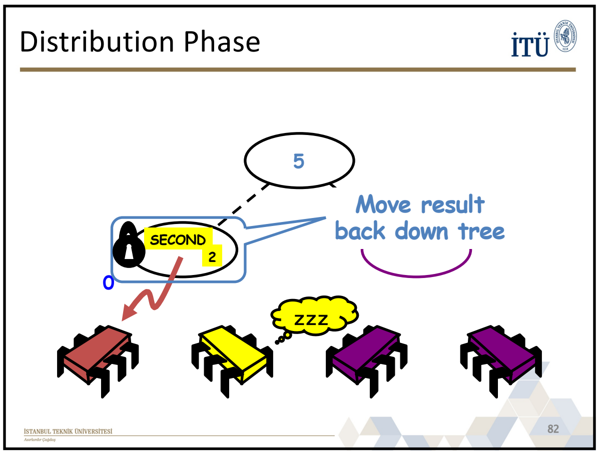
Distribution Phase

Move down with result



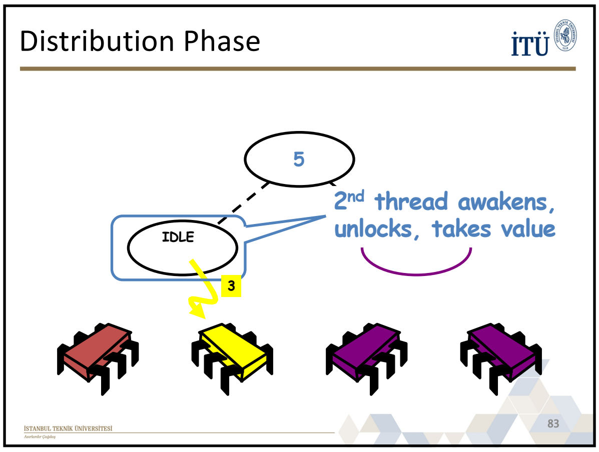
Distribution Phase

Leave result for 2nd thread & lock node



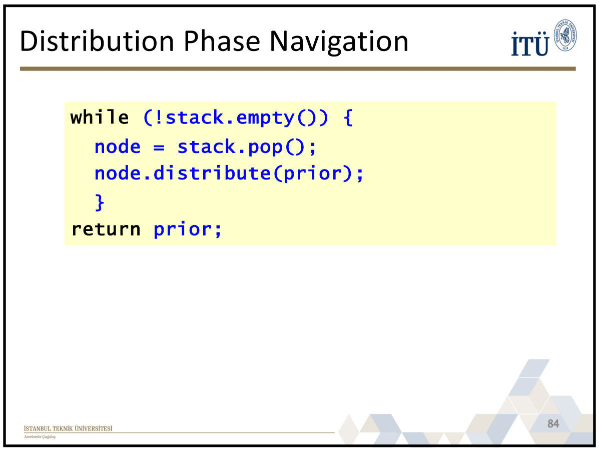
Distribution Phase

Move result back down tree



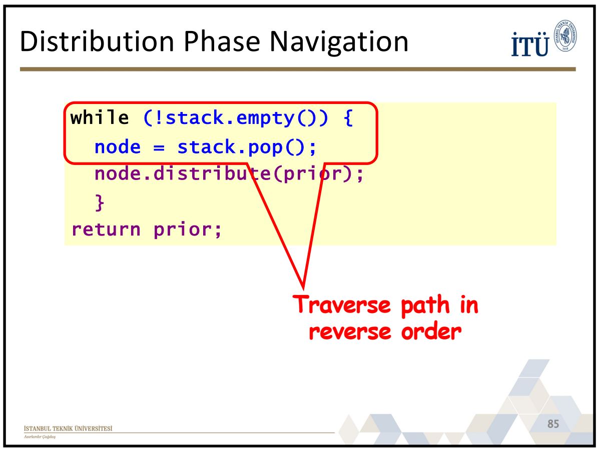
Distribution Phase

2nd thread awakens, unlocks, takes value



Distribution Phase Navigation

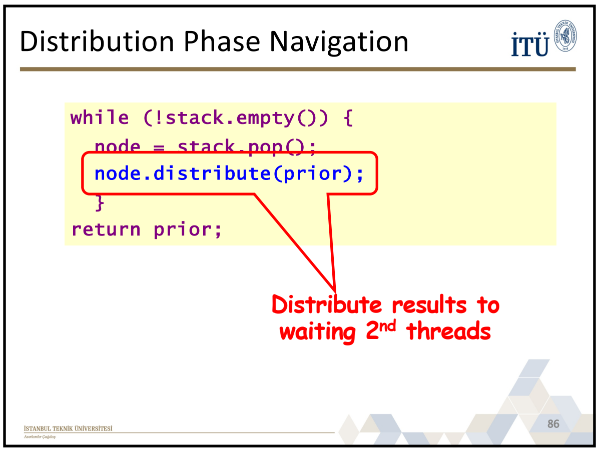
```
while (!stack.empty()) {
  node = stack.pop();
  node.distribute(prior);
  }
return prior;
```



Distribution Phase Navigation

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

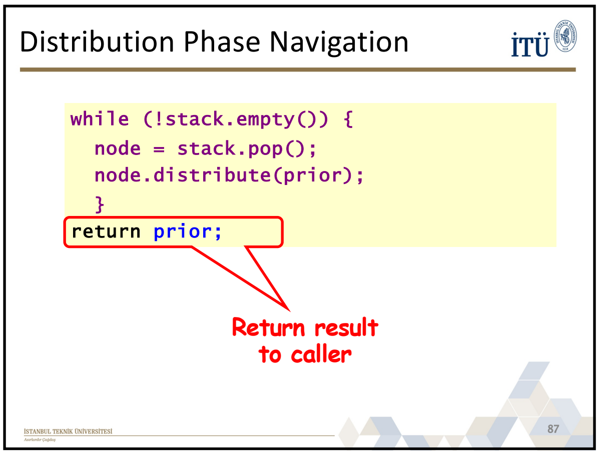
Traverse path in reverse order



Distribution Phase Navigation

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

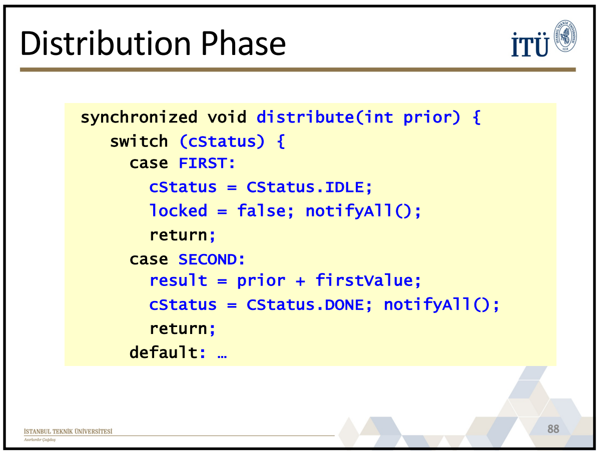
Distribute results to waiting 2nd threads



Distribution Phase Navigation

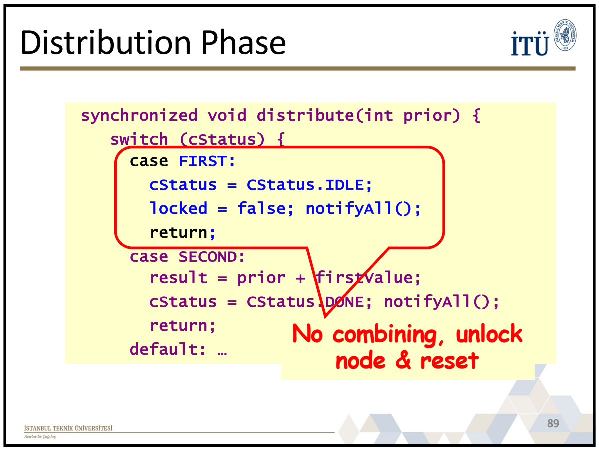
```
while (!stack.empty()) {
  node = stack.pop();
  node.distribute(prior);
  }
return prior;
```

Return result to caller



Distribution Phase

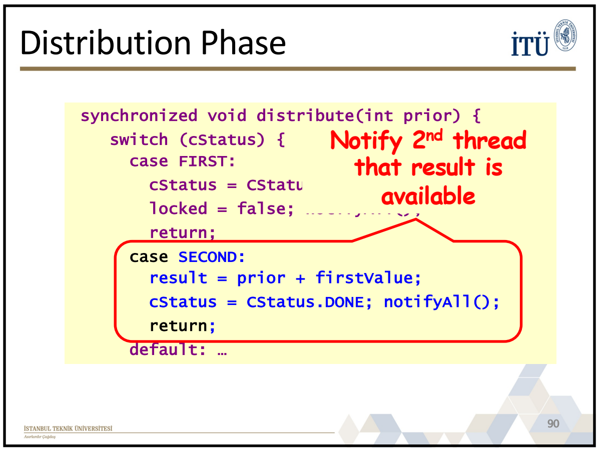
```
synchronized void distribute(int prior) {
   switch (cStatus) {
     case FIRST:
       cStatus = CStatus.IDLE;
       locked = false; notifyAll();
       return;
     case SECOND:
       result = prior + firstValue;
       cStatus = CStatus.DONE; notifyAll();
       return;
     default: …
```



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: …
```

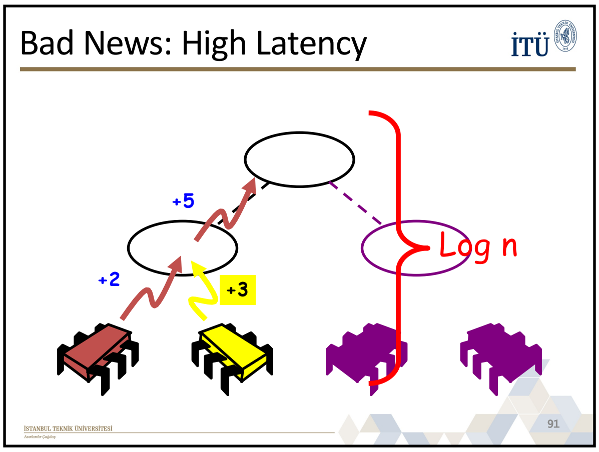
No combining, unlock node & reset



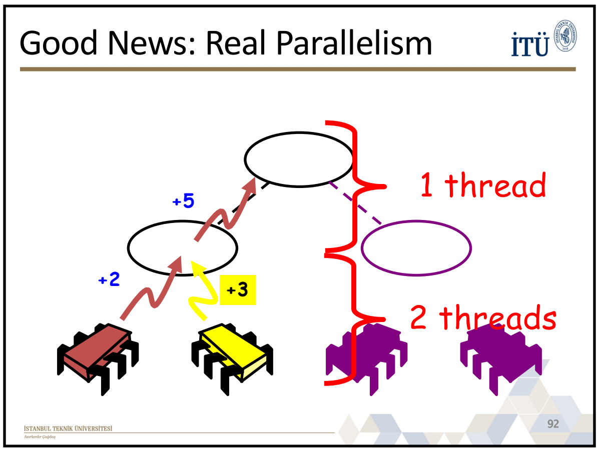
Distribution Phase

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

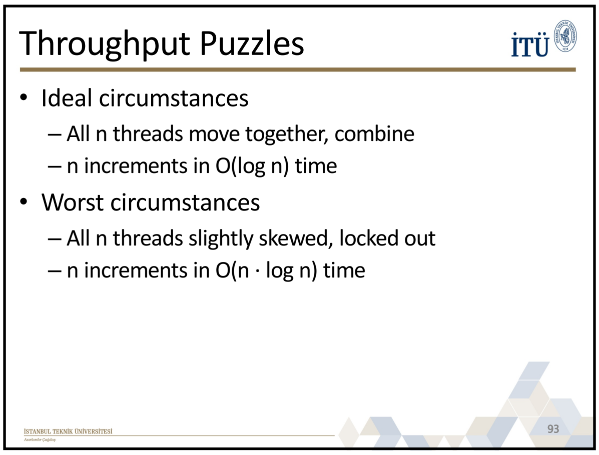
Notify 2nd thread that result is available



Bad News: High Latency

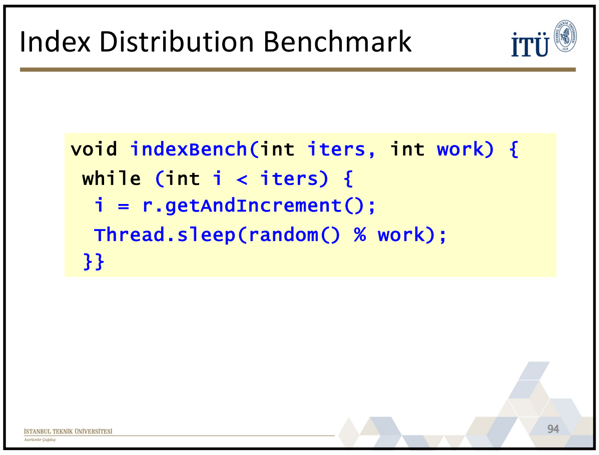


Good News: Real Parallelism



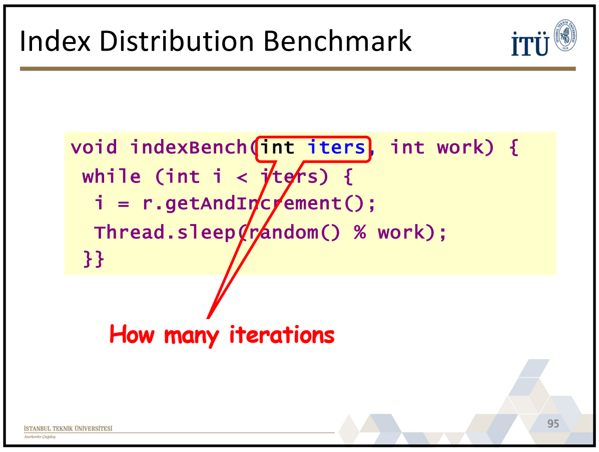
Throughput Puzzles

- 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 · log n) time



Index Distribution Benchmark

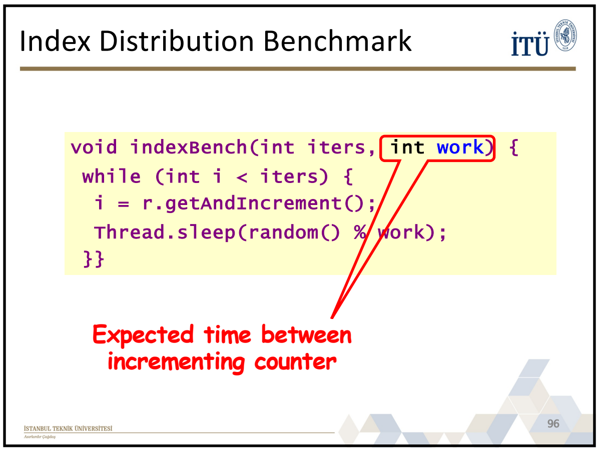
```
void indexBench(int iters, int work) {
 while (int i < iters) {
  i = r.getAndIncrement();
  Thread.sleep(random() % work);
 }}
```



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

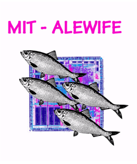
Index Distribution Benchmark

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

Pretend to work
(more work, less concurrency)

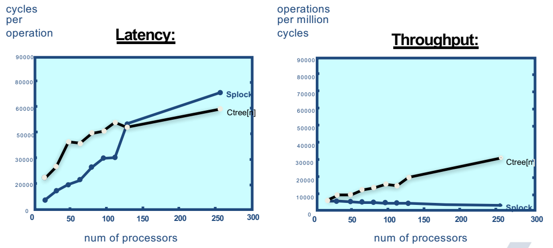
Performance Benchmarks

- Alewife
  - NUMA architecture
  - Simulated



- **Throughput:**
  - average number of **inc** operations in 1 million cycle period.
- **Latency:**
  - average number of simulator cycles per **inc** operation.

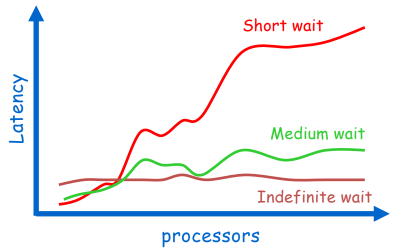
Performance

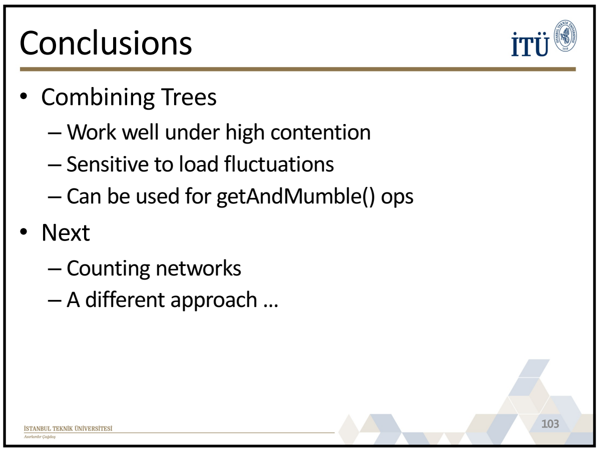


The Combining Paradigm

- 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!

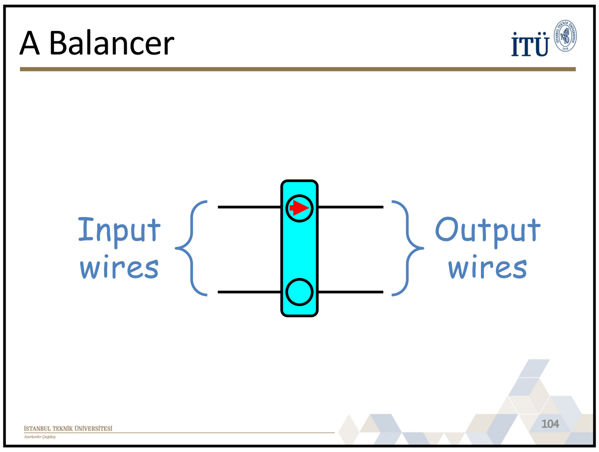
Better to Wait Longer





Conclusions

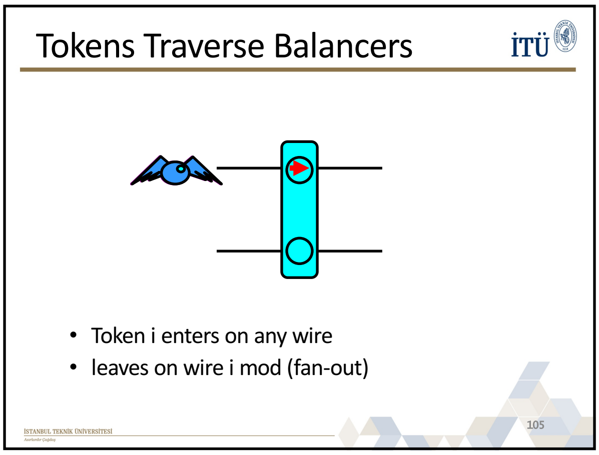
- Combining Trees
  - Work well under high contention
  - Sensitive to load fluctuations
  - Can be used for getAndMumble() ops
- Next
  - Counting networks
  - A different approach …



A Balancer

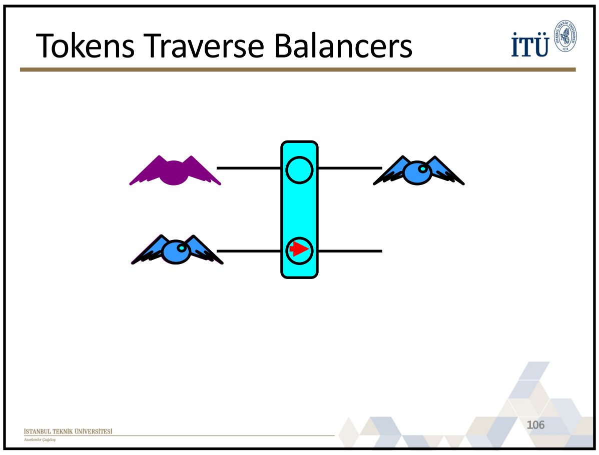
Input wires

Output wires

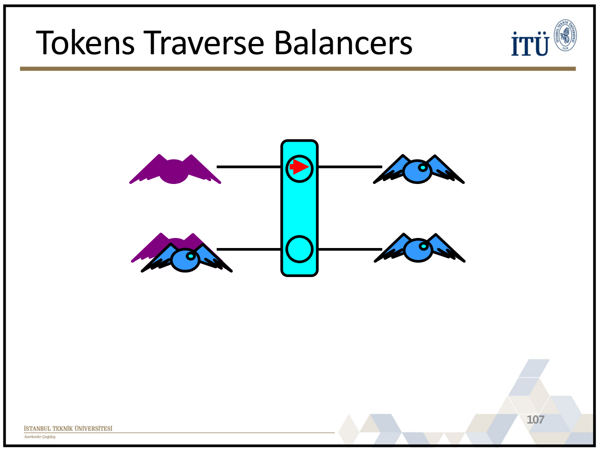


Tokens Traverse Balancers

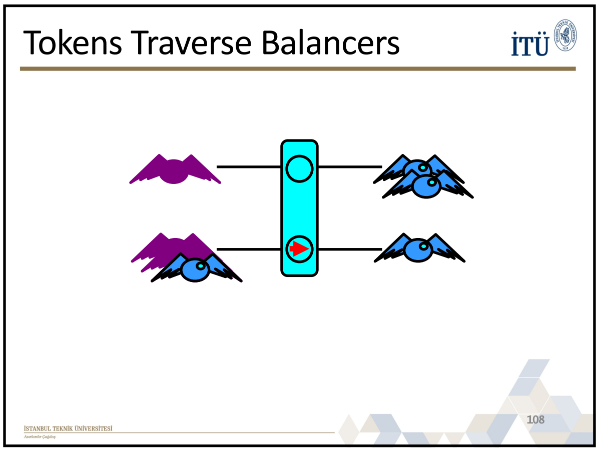
- Token i enters on any wire
- leaves on wire i mod (fan-out)



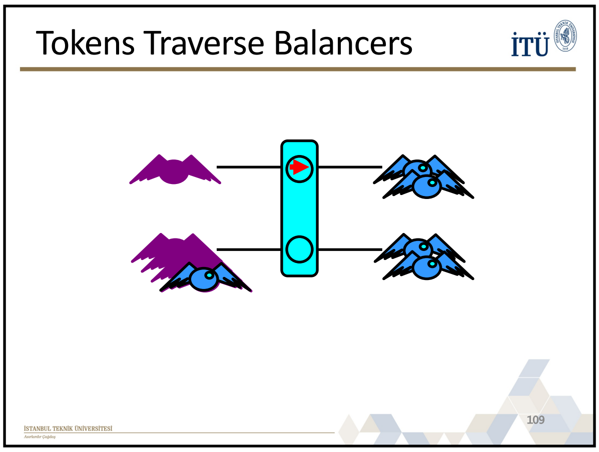
Tokens Traverse Balancers



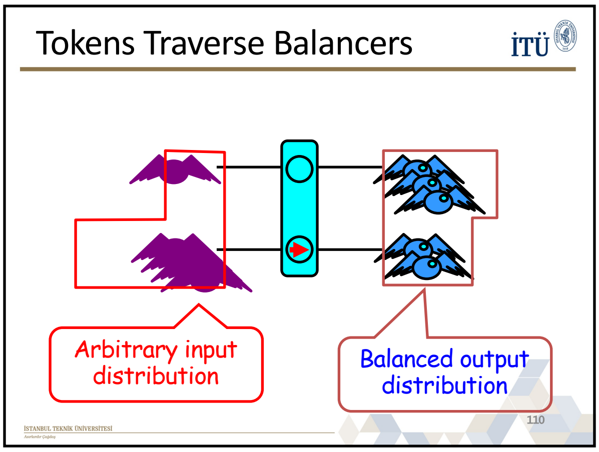
Tokens Traverse Balancers



Tokens Traverse Balancers



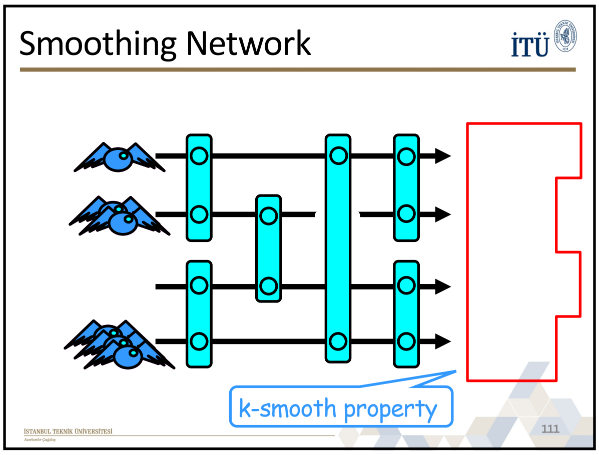
Tokens Traverse Balancers



Tokens Traverse Balancers

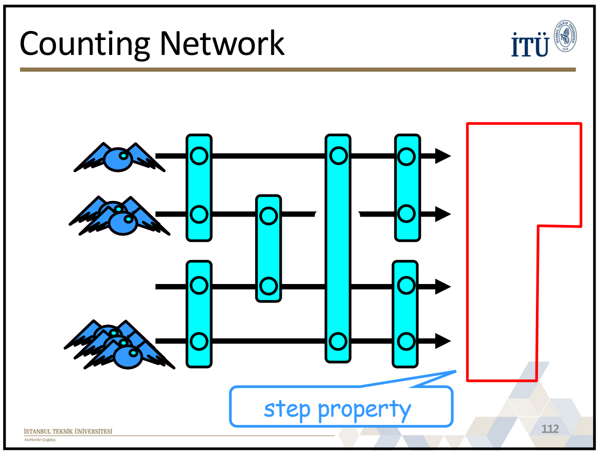
Arbitrary input distribution

Balanced output distribution



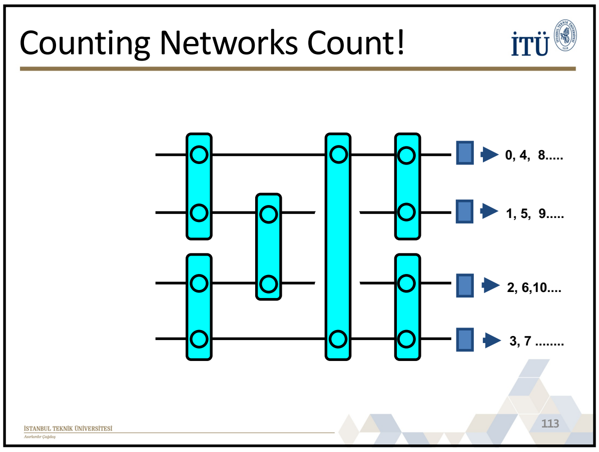
Smoothing Network

k-smooth property

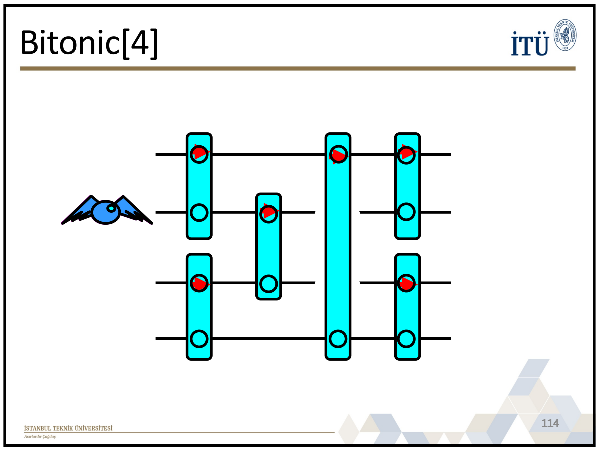


Counting Network

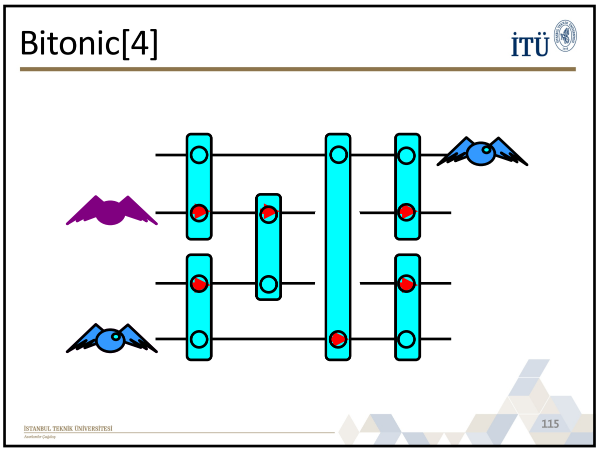
step property



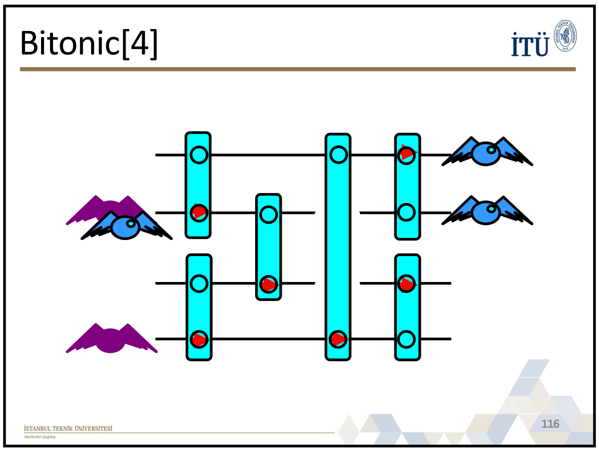
Counting Networks Count!



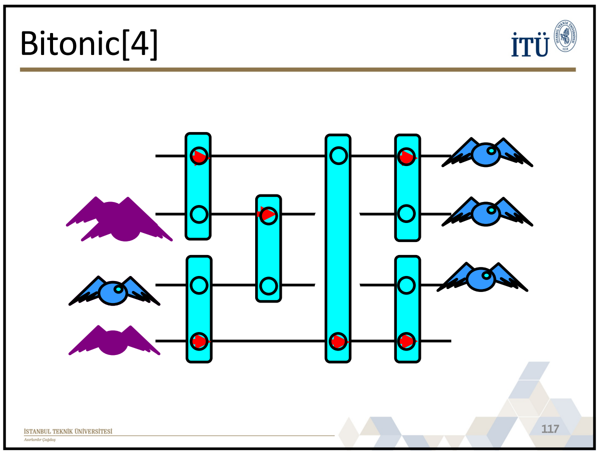
Bitonic[4]



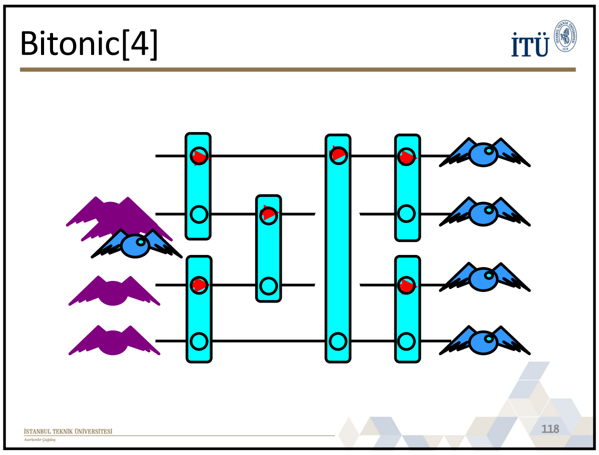
Bitonic[4]



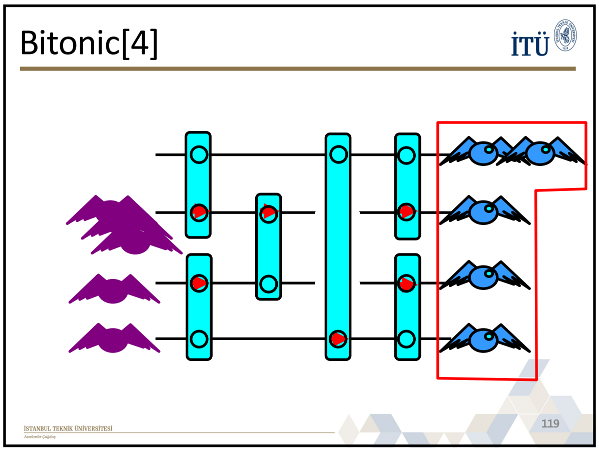
Bitonic[4]



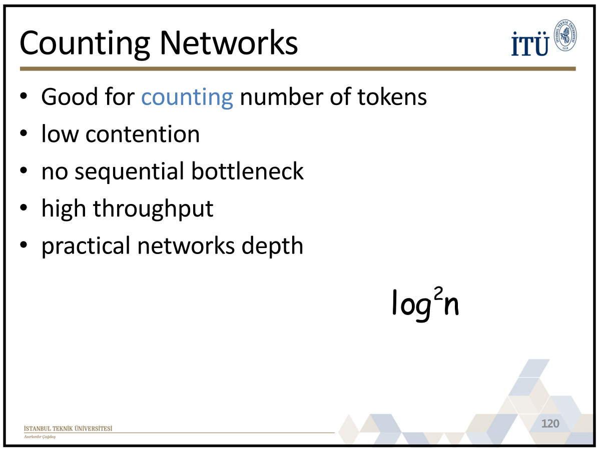
Bitonic[4]



Bitonic[4]



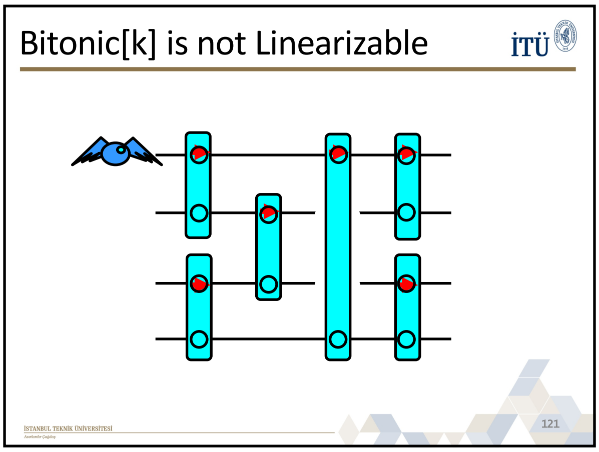
Bitonic[4]



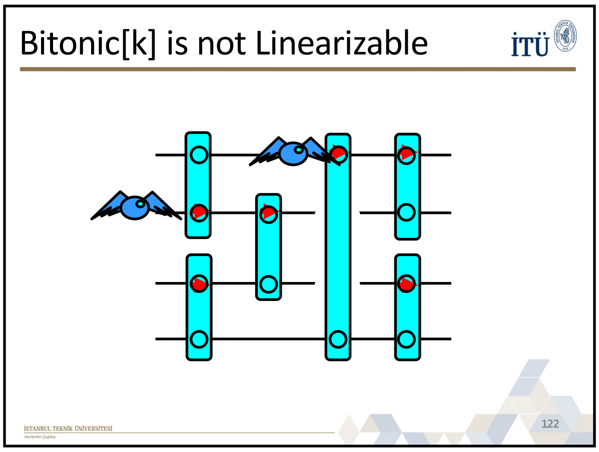
Counting Networks

- Good for counting number of tokens
- low contention
- no sequential bottleneck
- high throughput
- practical networks depth

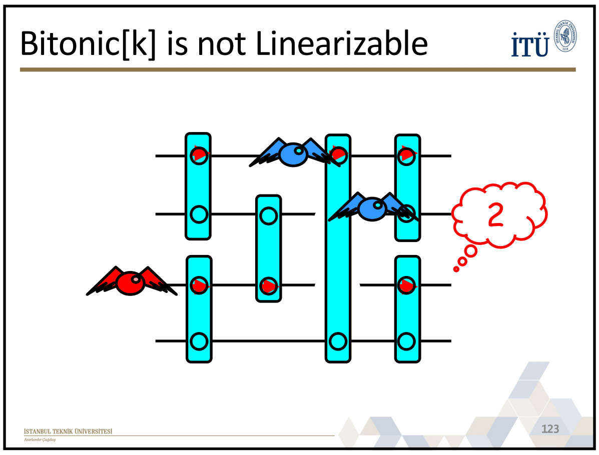
$\log^2 n$



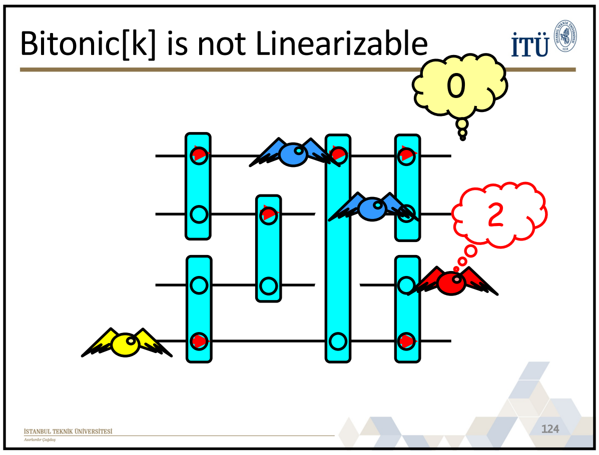
Bitonic[k] is not Linearizable



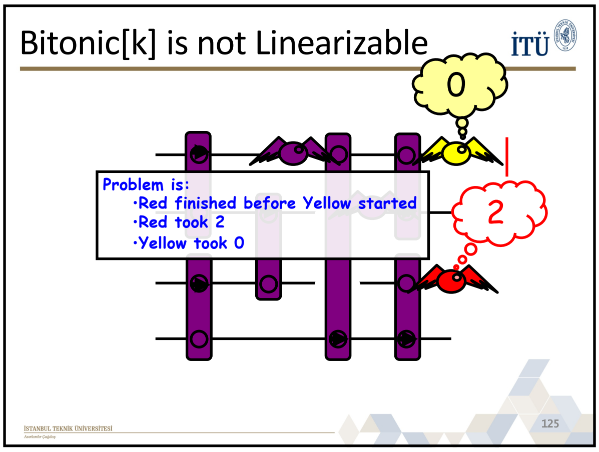
Bitonic[k] is not Linearizable



Bitonic[k] is not Linearizable

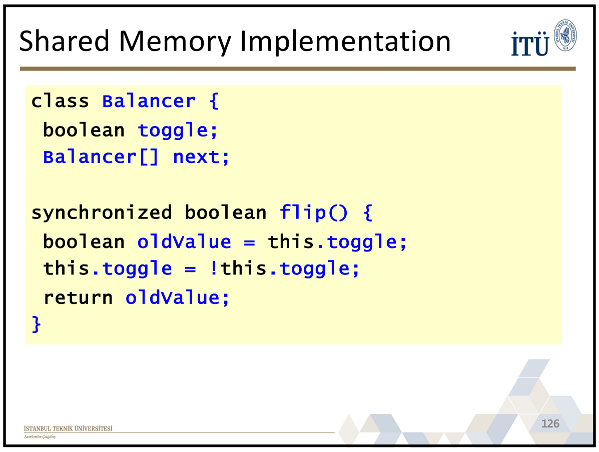


Bitonic[k] is not Linearizable



Bitonic[k] is not Linearizable

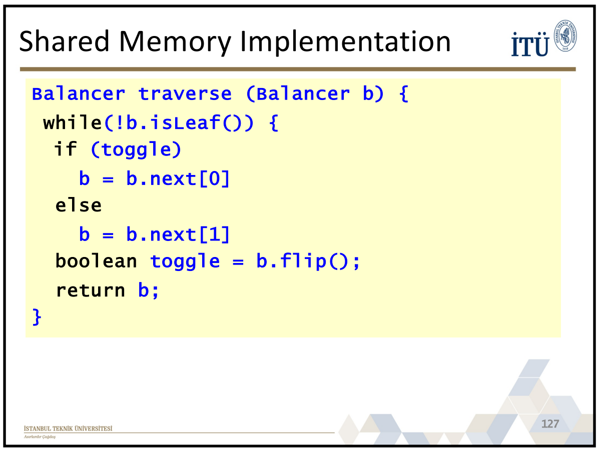
Problem is:
- Red finished before Yellow started
- Red took 2
- Yellow took 0



Shared Memory Implementation

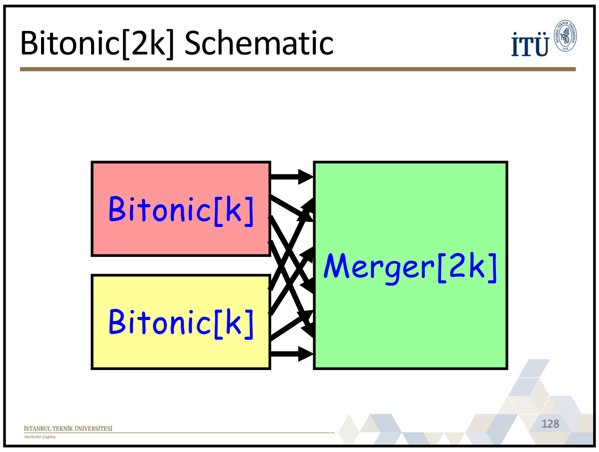
```
class Balancer {
 boolean toggle;
 Balancer[] next;

synchronized boolean flip() {
 boolean oldValue = this.toggle;
 this.toggle = !this.toggle;
 return oldValue;
}
```



Shared Memory Implementation

```
Balancer traverse (Balancer b) {
 while(!b.isLeaf()) {
  if (toggle)
    b = b.next[0]
  else
    b = b.next[1]
  boolean toggle = b.flip();
  return b;
}
```

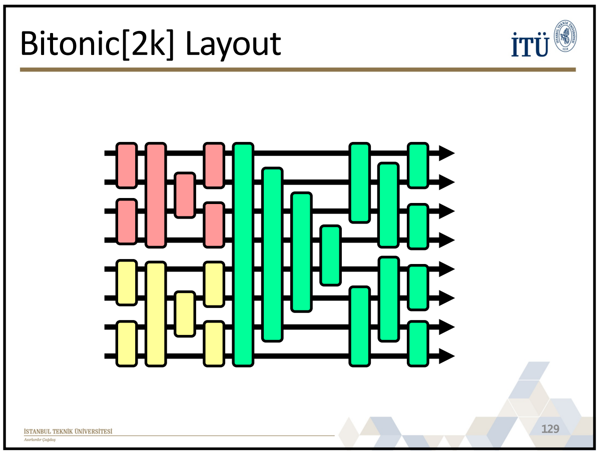


Bitonic[2k] Schematic

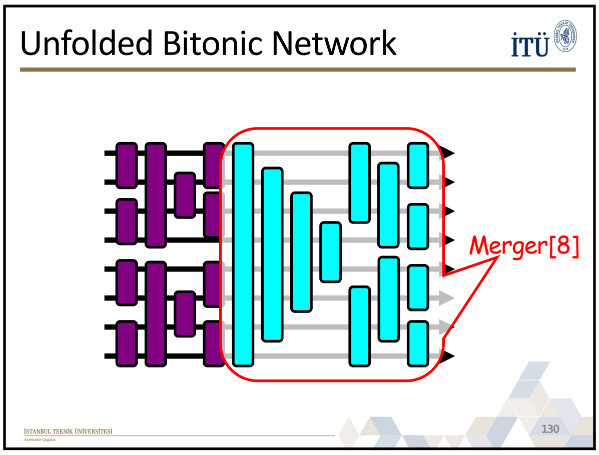
Bitonic[k]

Bitonic[k]

Merger[2k]

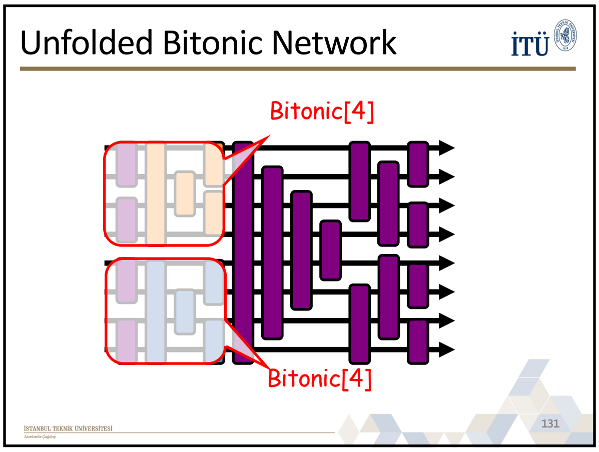


Bitonic[2k] Layout



Unfolded Bitonic Network

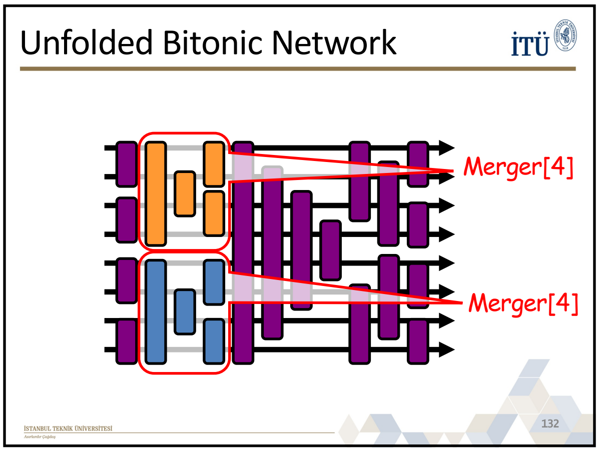
Merger[8]



Unfolded Bitonic Network

Bitonic[4]

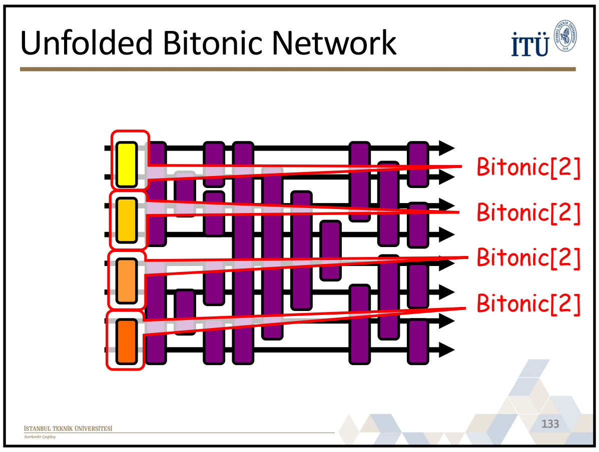
Bitonic[4]



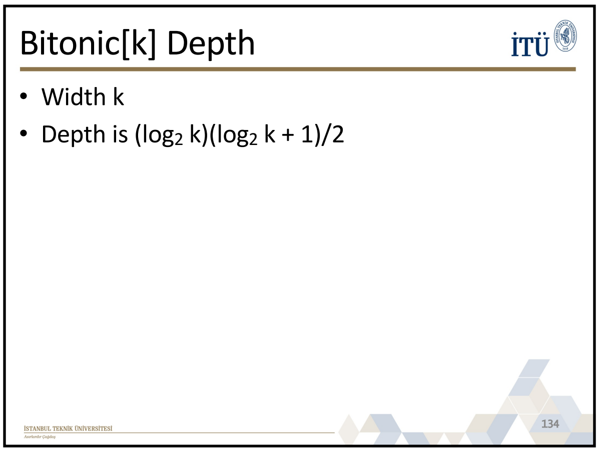
Unfolded Bitonic Network

Merger[4]

Merger[4]

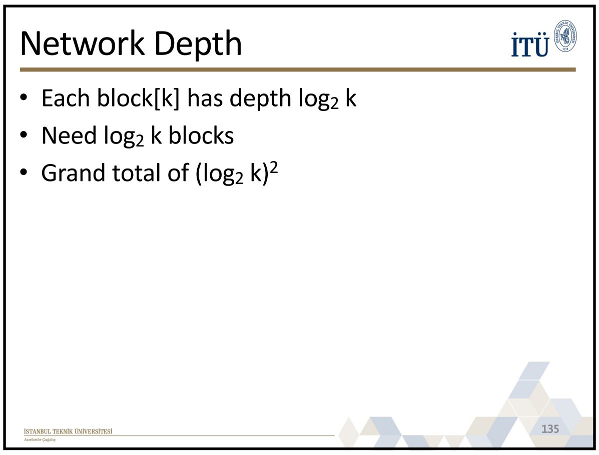


Unfolded Bitonic Network



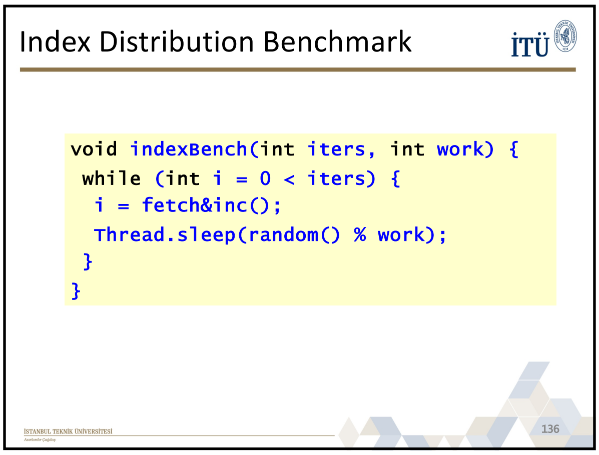
Bitonic[k] Depth

- Width k
- Depth is $(\log_2 k)(\log_2 k + 1)/2$



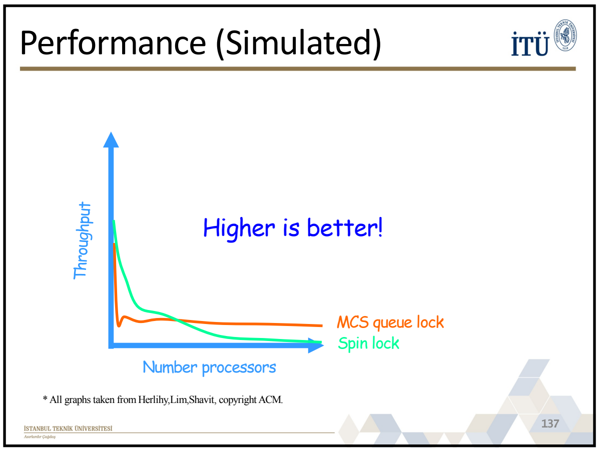
Network Depth

- Each block[k] has depth $\log_2 k$
- Need $\log_2 k$ blocks
- Grand total of $(\log_2 k)^2$



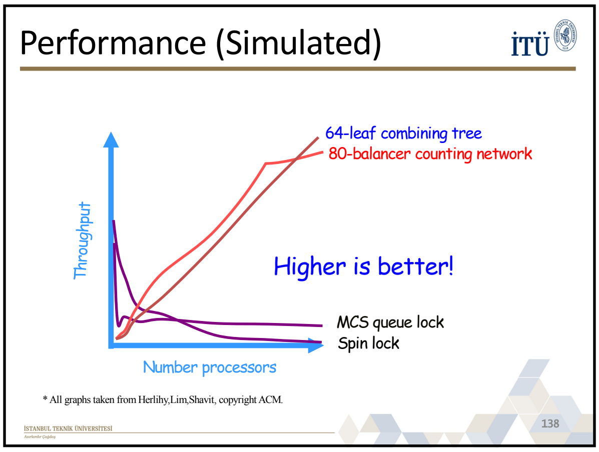
Index Distribution Benchmark

```
void indexBench(int iters, int work) {
 while (int i = 0 < iters) {
  i = fetch&inc();
  Thread.sleep(random() % work);
 }
}
```



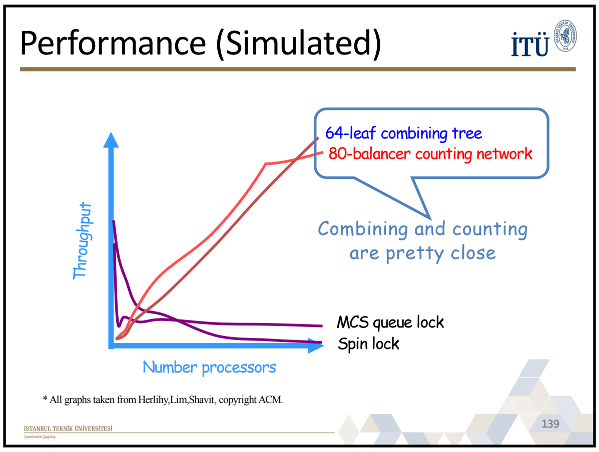
Performance (Simulated)

* All graphs taken from Herlihy,Lim,Shavit, copyright ACM.



Performance (Simulated)

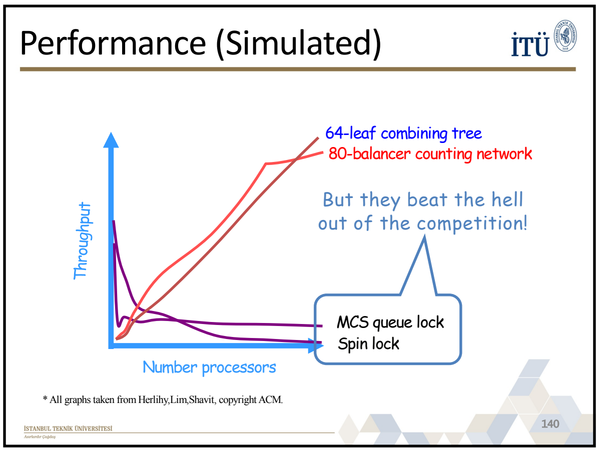
* All graphs taken from Herlihy,Lim,Shavit, copyright ACM.



Performance (Simulated)

Combining and counting are pretty close

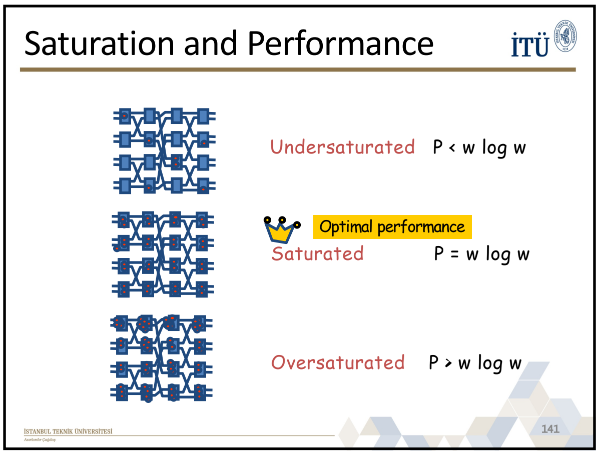
* All graphs taken from Herlihy,Lim,Shavit, copyright ACM.



Performance (Simulated)

But they beat the hell out of the competition!

* All graphs taken from Herlihy,Lim,Shavit, copyright ACM.



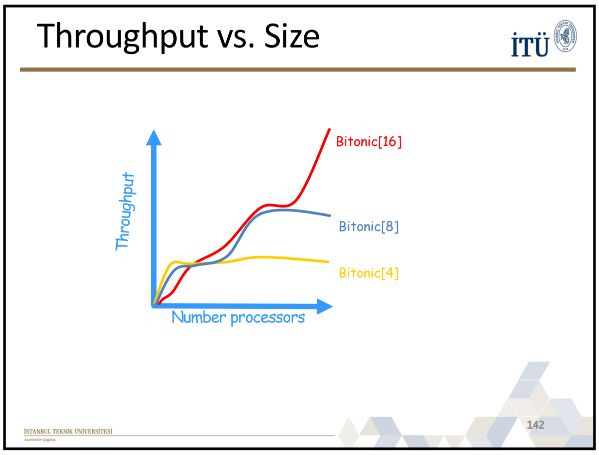
Saturation and Performance

Undersaturated P < w log w

Optimal performance

Saturated P = w log w

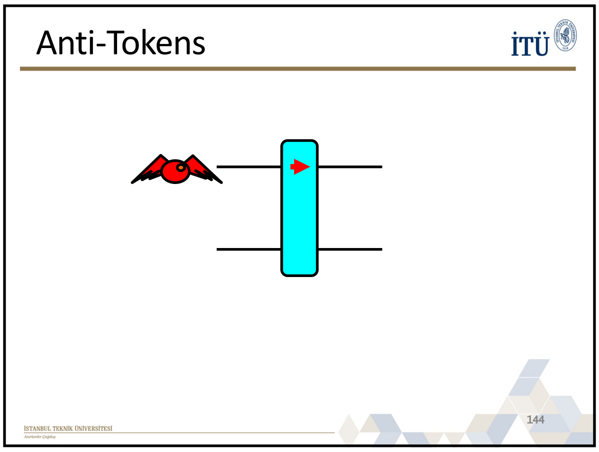
Oversaturated P > w log w



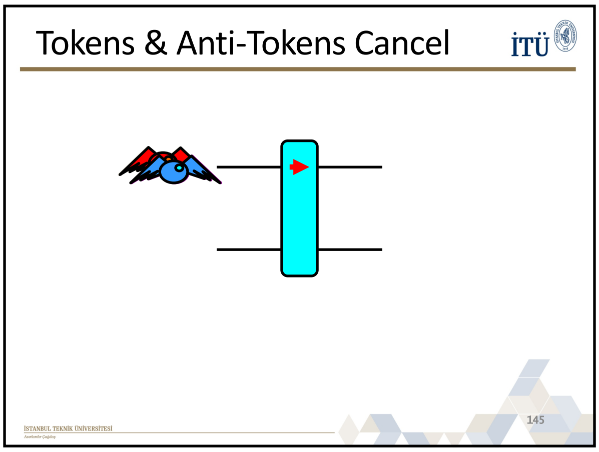
Throughput vs. Size

What About

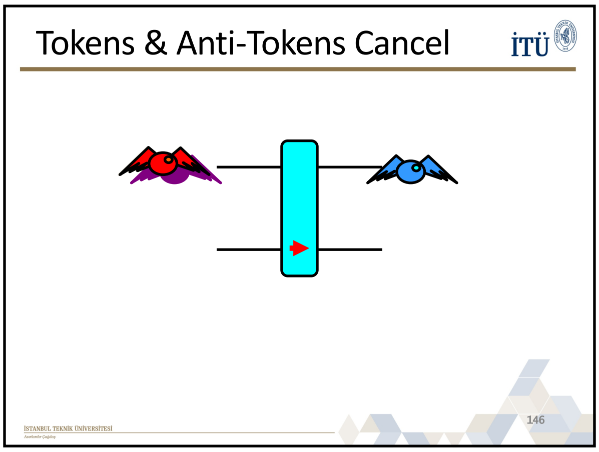
- Decrements
- Adding arbitrary values



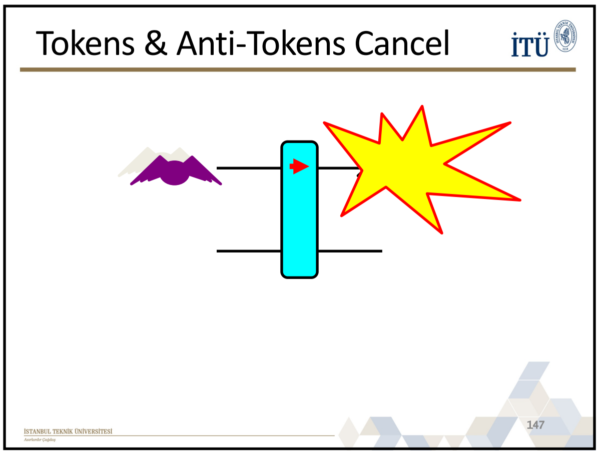
Anti-Tokens



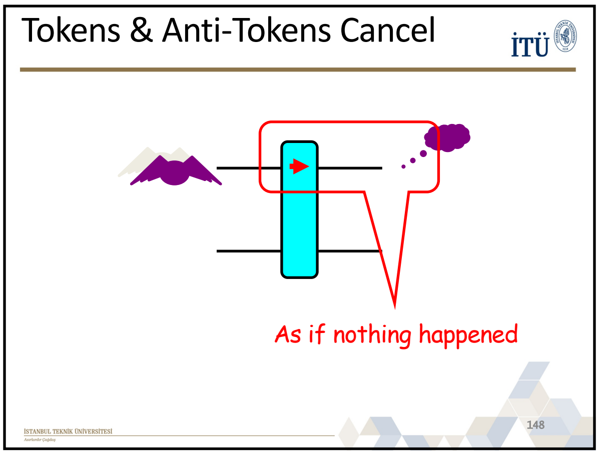
Tokens & Anti-Tokens Cancel



Tokens & Anti-Tokens Cancel

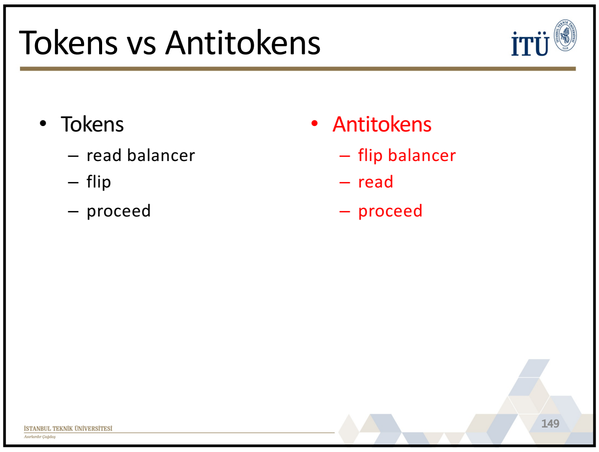


Tokens & Anti-Tokens Cancel



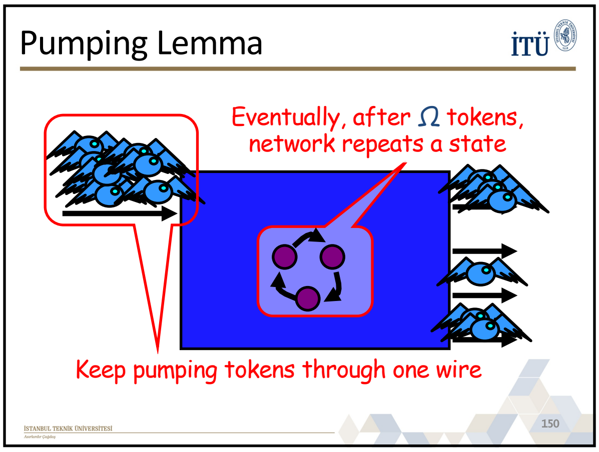
Tokens & Anti-Tokens Cancel

As if nothing happened



Tokens vs Antitokens

- Tokens
  - read balancer
  - flip
  - proceed
- Antitokens
  - flip balancer
  - read
  - proceed

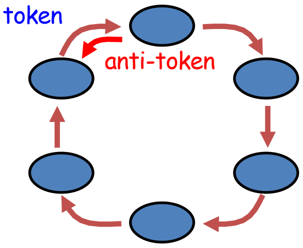


Pumping Lemma

Eventually, after Ω tokens, network repeats a state

Keep pumping tokens through one wire

Anti-Token Effect


token
anti-token