departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Alternative Synchronization Strategies

lecture 17 (2020-04-29)

Master in Computer Science and Engineering

— Concurrency and Parallelism / 2019-20 —

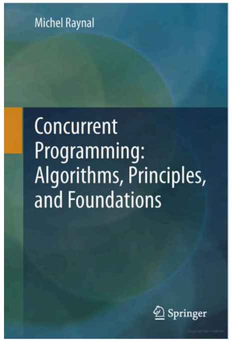
João Lourenço <joao.lourenco@fct.unl.pt>

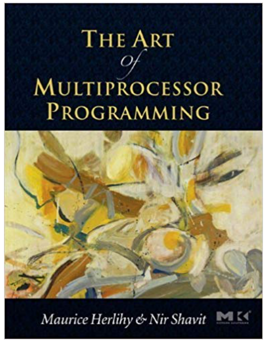
Alternative Synchronization Strategies

- Contents:
  - Liveness: Types of Progress
  - Coarse-Grained Synchronization
  - Fine-Grained Synchronization
  - Optimistic Synchronization
  - Lazy Synchronization
  - Lock-Free Synchronization

Last lecture

Today

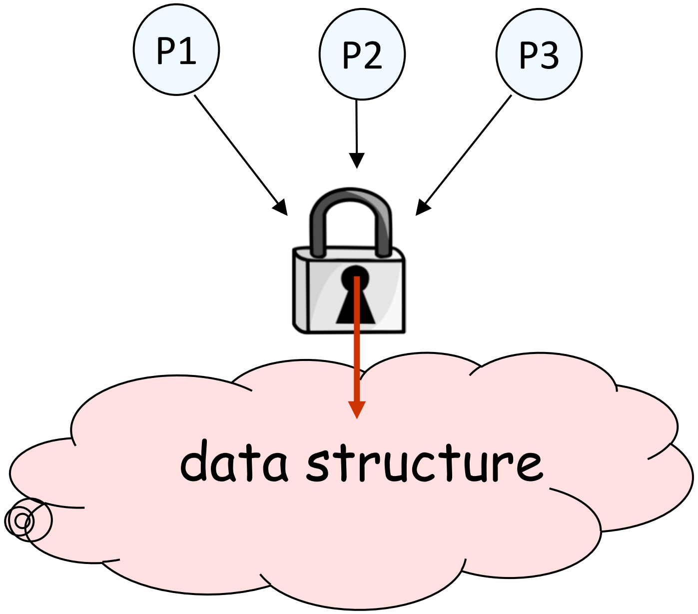




- Reading list:
  - chapter 5 of the Textbook
  - Chapter 9 of "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit *(available at clip)*

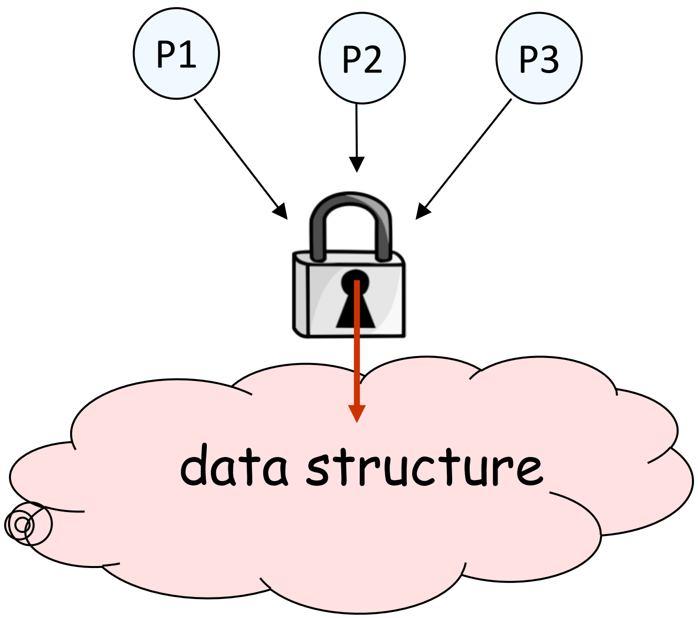
Concurrent Data Structures

Using locks



Concurrent Data Structures

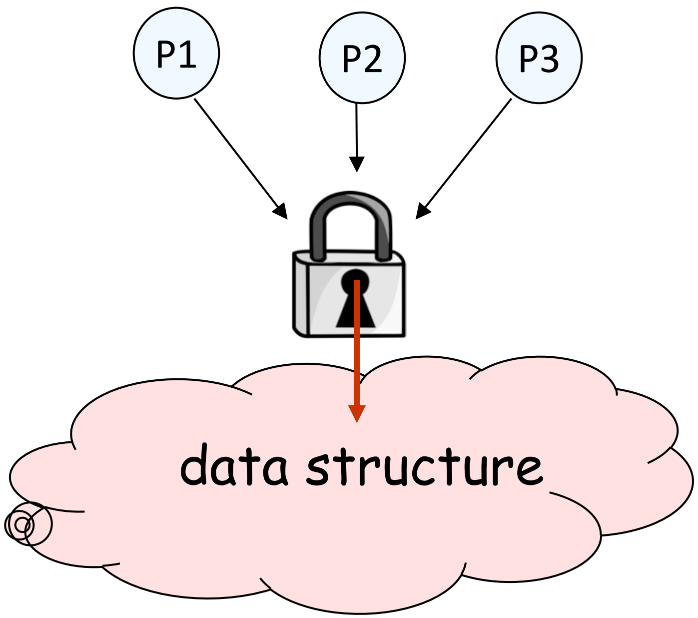
Using locks



- Simple programming model
- False conflicts
- Fault-free solutions only
- Sequential bottleneck

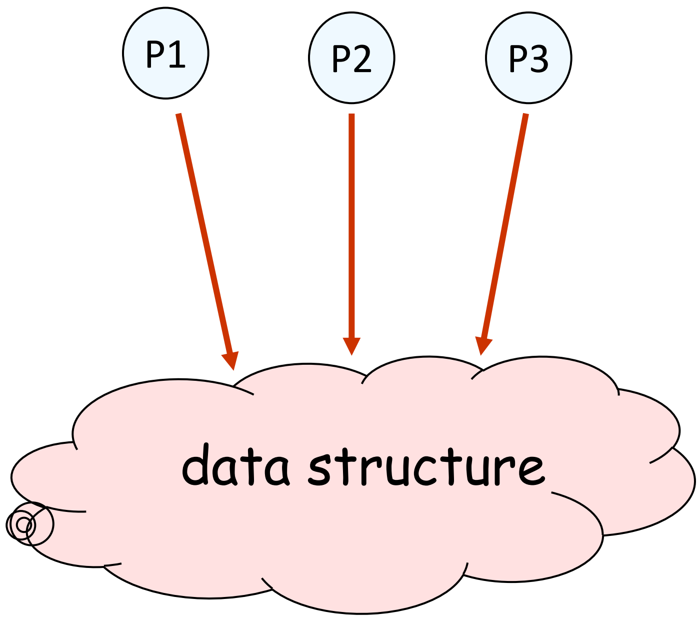
Concurrent Data Structures

Using locks



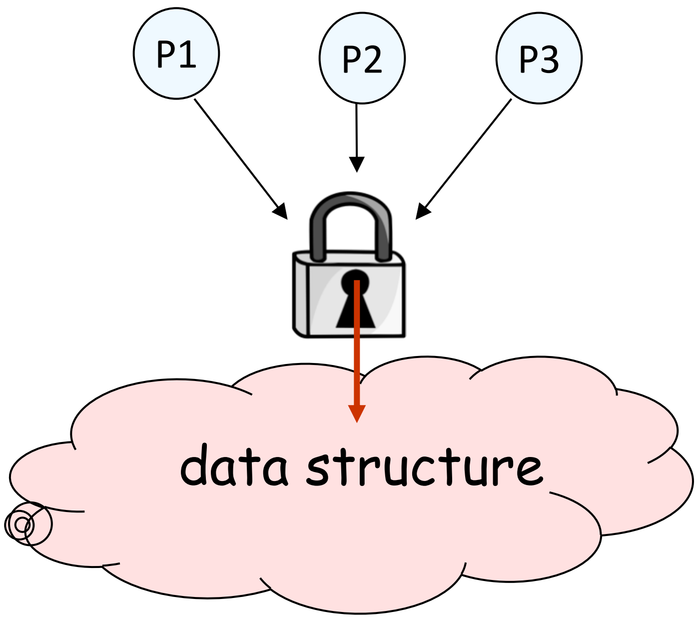
- Simple programming model
- False conflicts
- Fault-free solutions only
- Sequential bottleneck

Without locks



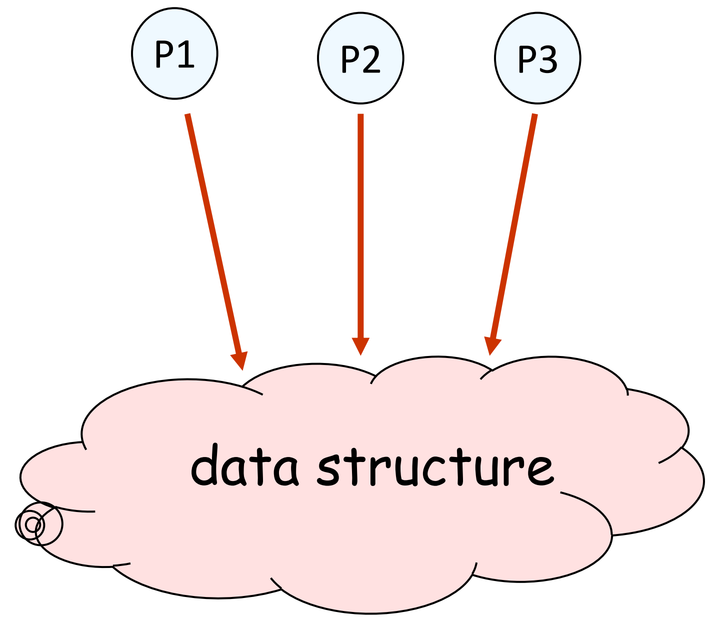
Concurrent Data Structures

Using locks



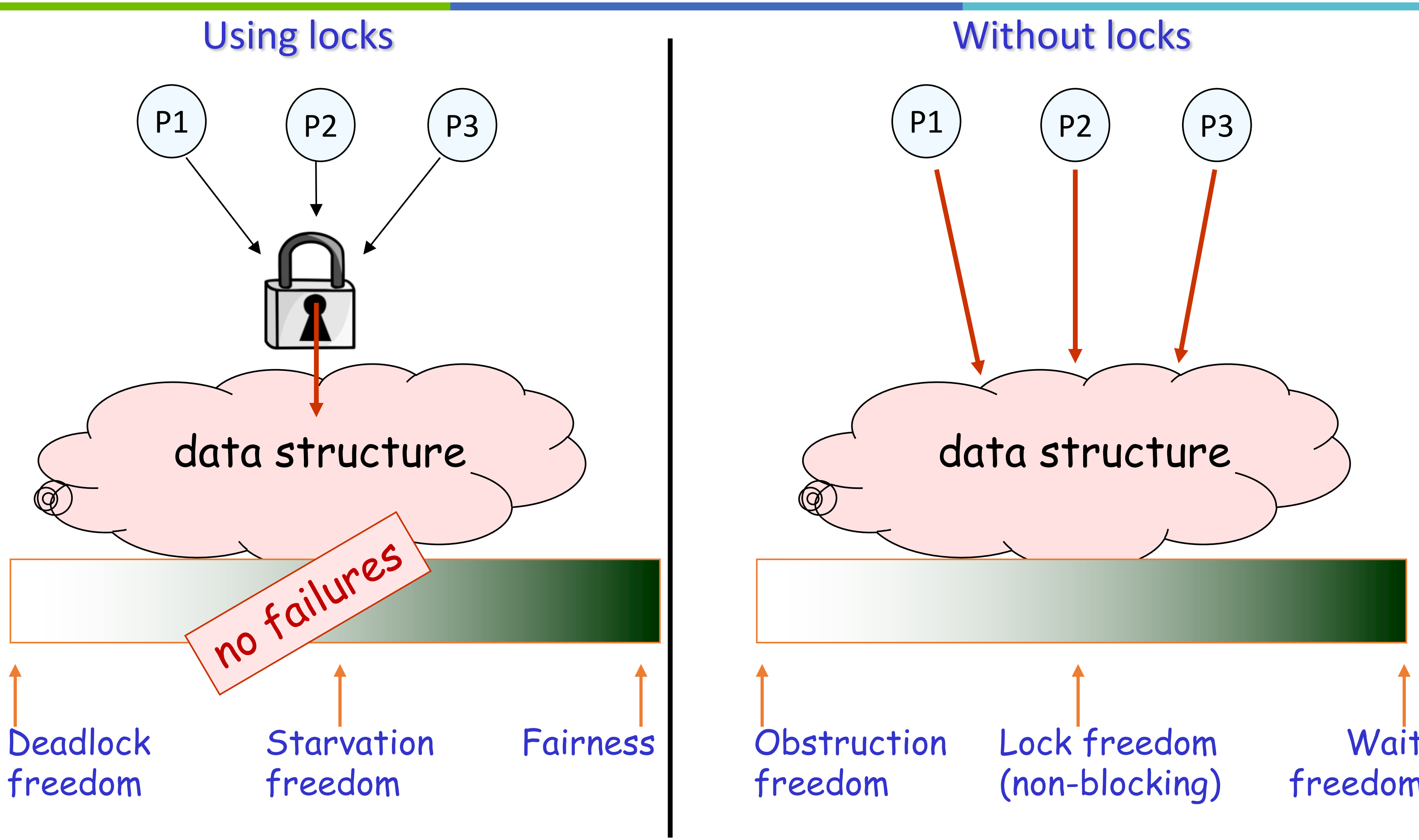
- Simple programming model
- False conflicts
- Fault-free solutions only
- Sequential bottleneck

Without locks



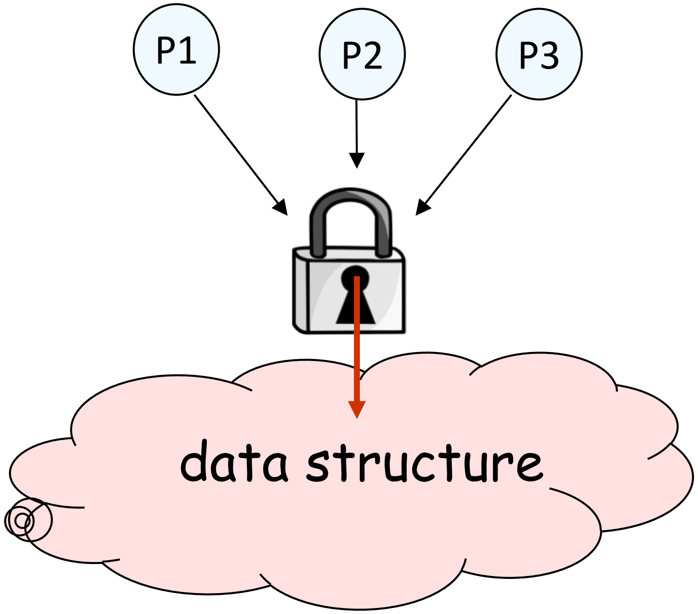
- Resilient to failures, etc.
- Often (really very) complex
- Memory consuming
- Sometimes — weak progress cond.

Progress in Concurrent Data Structures

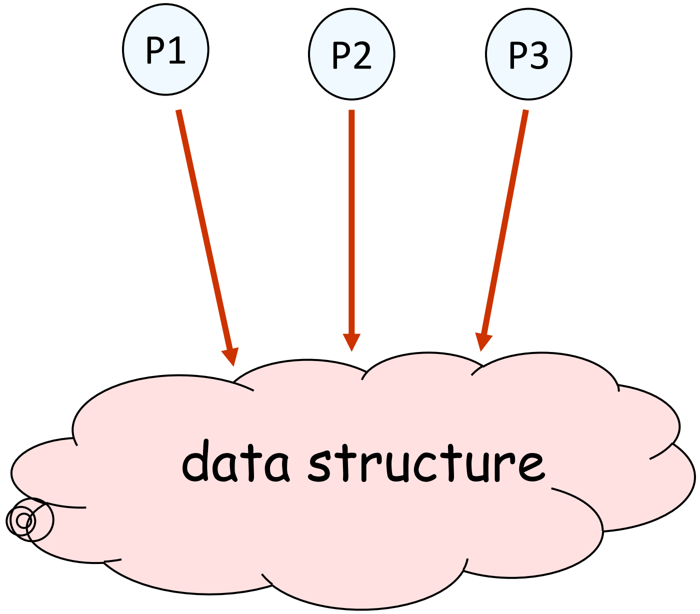


Progress Conditions

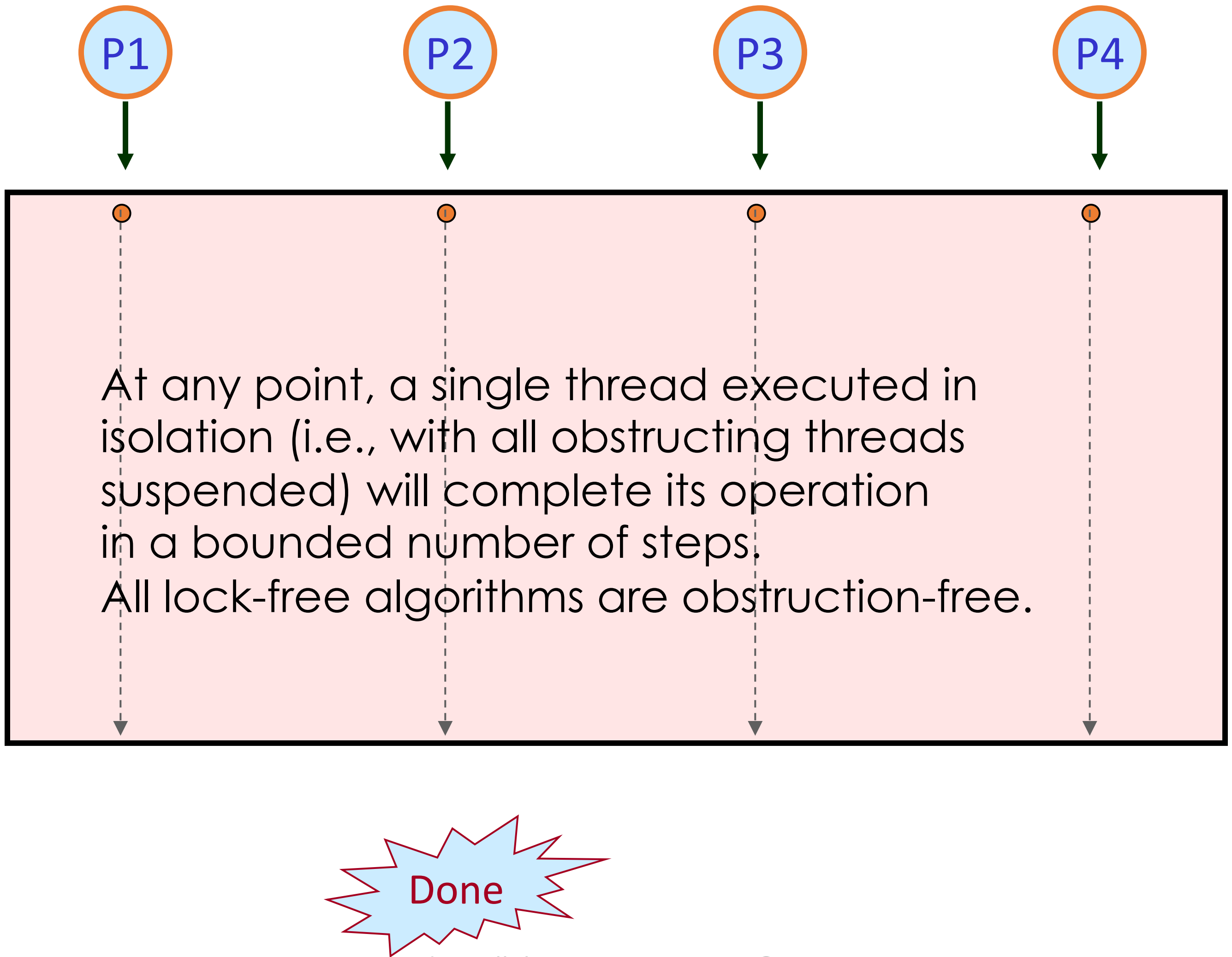
Using locks



Without locks

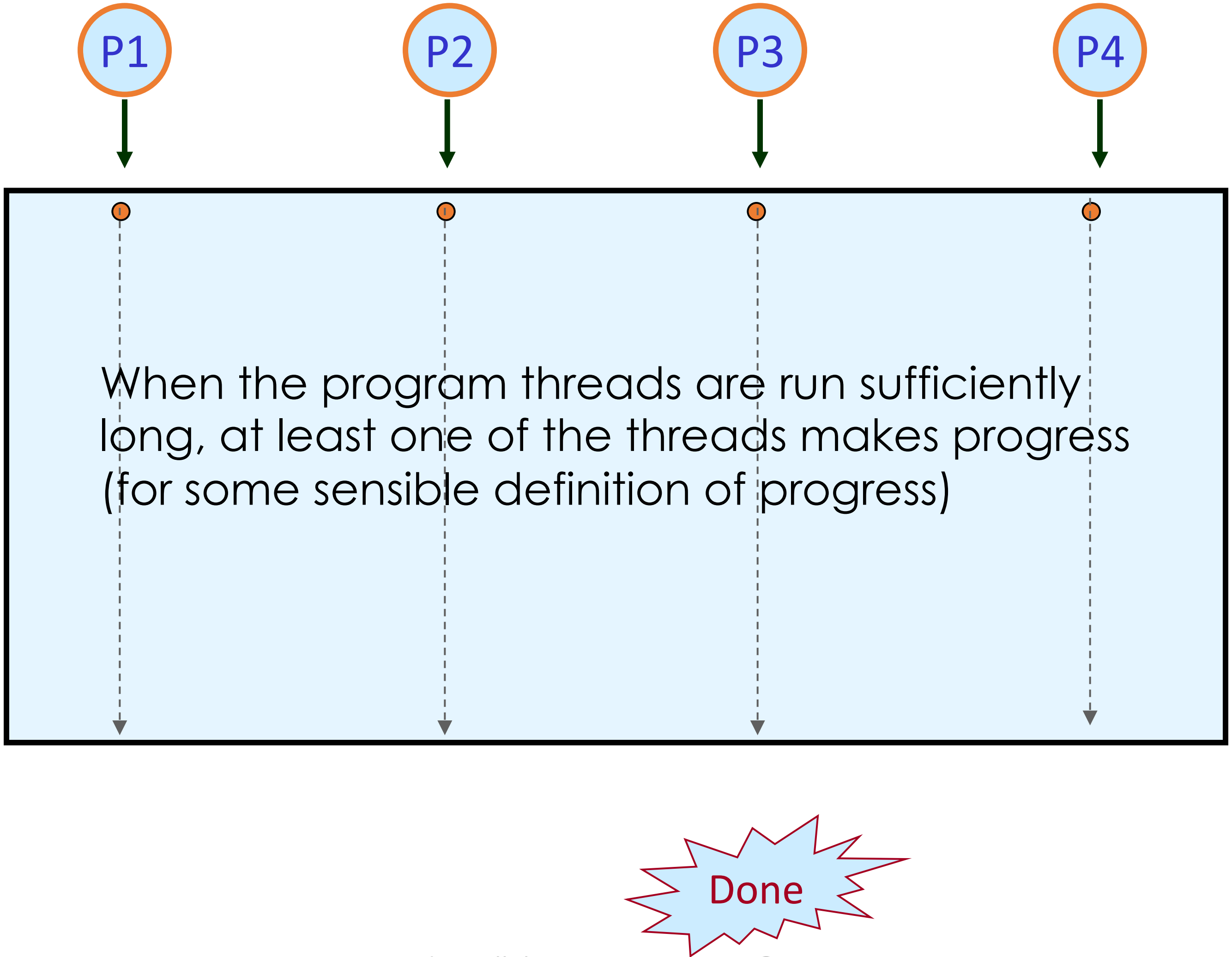


Obstruction-freedom

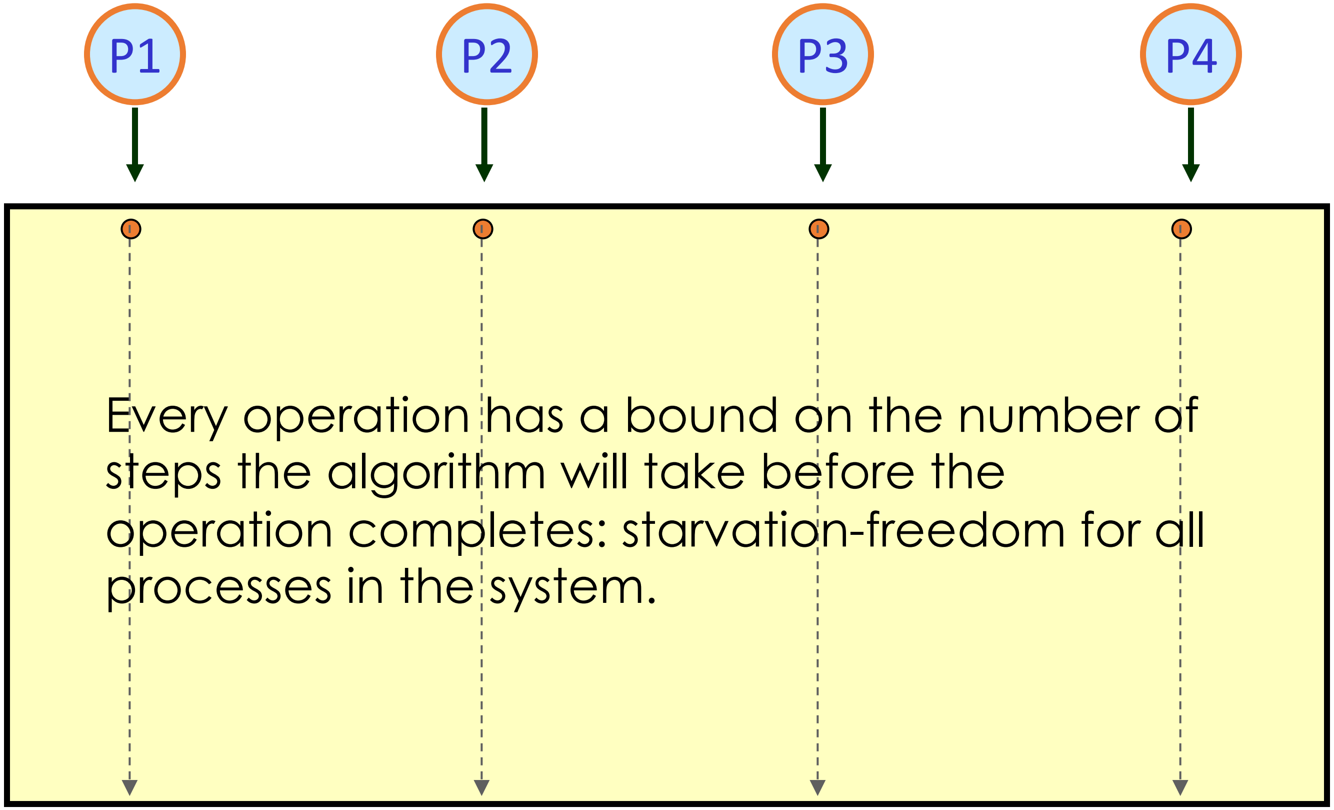


At any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) will complete its operation in a bounded number of steps.
All lock-free algorithms are obstruction-free.

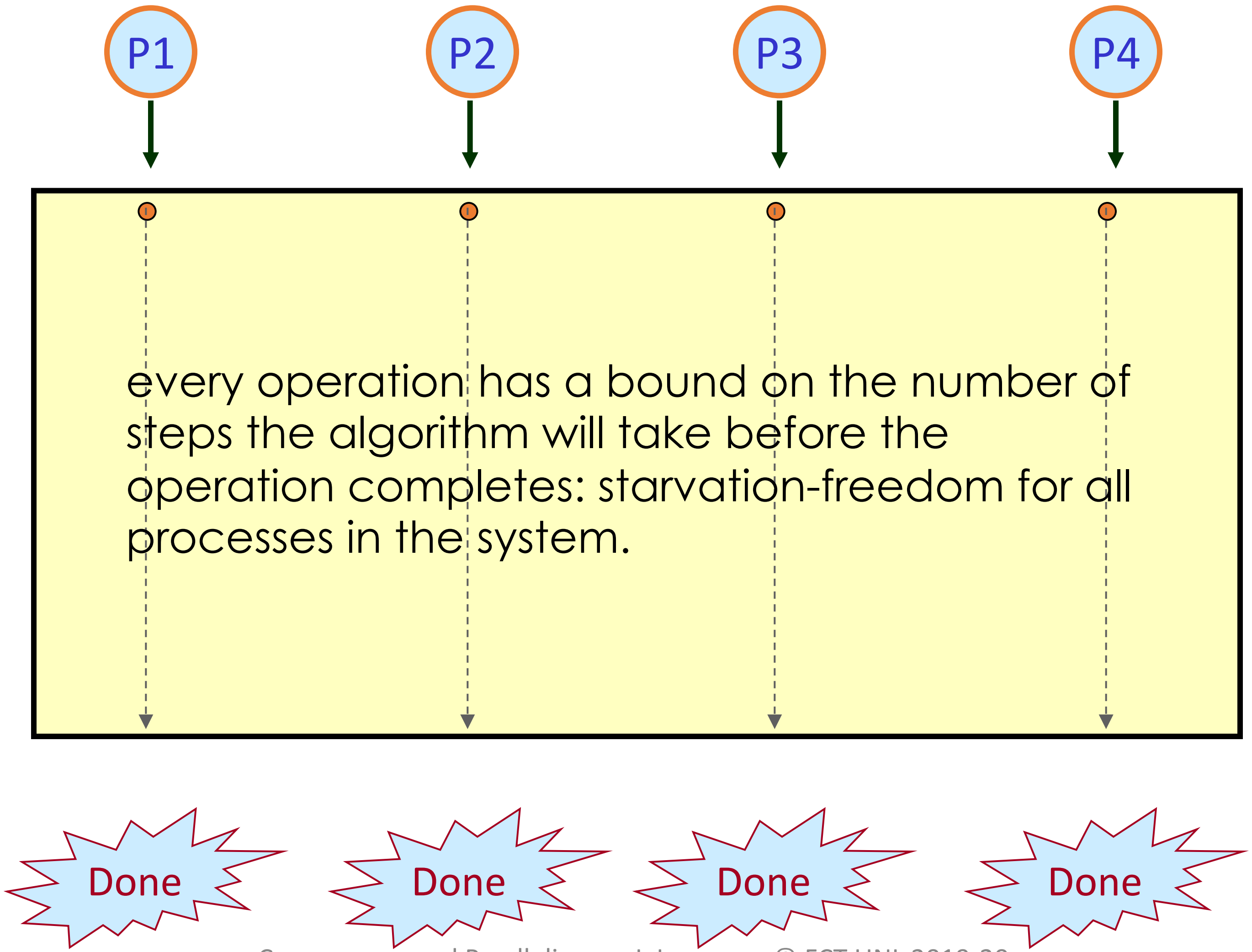
Lock-freedom



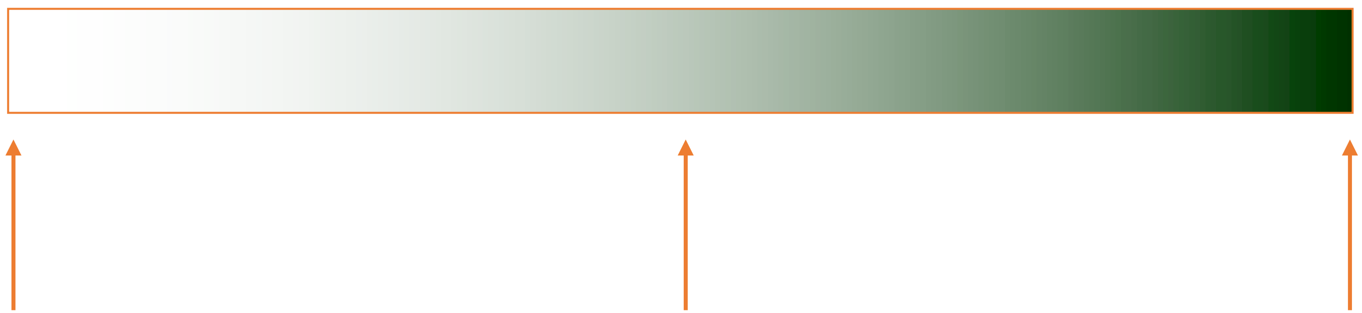
Wait-freedom



Wait-freedom



Lock-free Data Structures



Obstruction-freedom

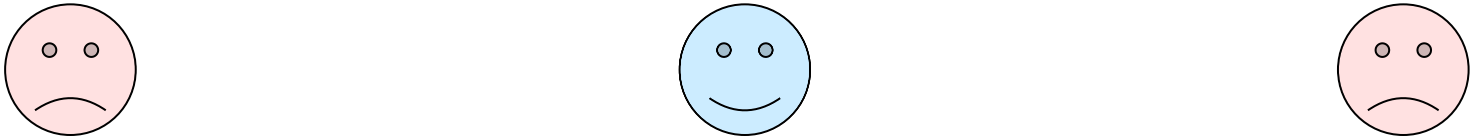
- too weak progress condition
- not complex

Lock-freedom

- strong enough
- not so complex
- in limited contention behaves as wait-free

Wait-freedom

- strong/desirable
- complex/less efficient



Synchronization strategies

- Coarse-Grained Synchronization
- Fine-Grained Synchronization
- Optimistic Synchronization
- Lazy Synchronization
- Lock-Free Synchronization

Coarse-Grained Synchronization

- Use a single lock…

✓ Methods are always executed in mutual exclusion

  ✓ Methods never conflict

✗ Eliminates all the concurrency within the object

Fine-Grained Synchronization

- Instead of using a single lock…
- Split object into multiple independently-synchronized components

✓ Methods only conflict when they access
  - The same component…
  - (And) at the same time!

✗ Lots and lots of lock acquire/release

Alternative Synchronization Strategies

Optimistic Synchronization

- Check if the operation can be done
  - E.g., to remove a value from the set, search if present without locking…
- If the op can be done, lock and check again…
  - E.g., if element was found, lock predecessor and current nodes and check again
- Act upon status (of last check)
  - Failure: start over again (optionally with another locking strategy)
  - Success: execute the operation (locks were already acquired)
- Evaluation/considerations on this strategy
  - ✓ Has to recheck (e.g., repeat the search) after locking
  - ✓ Usually cheaper than hand-over-hand locking
  - ✗ Mistakes are expensive (safety easily compromised)
  - ✗ Is not starvation free (liveness compromised)

Lazy Synchronization

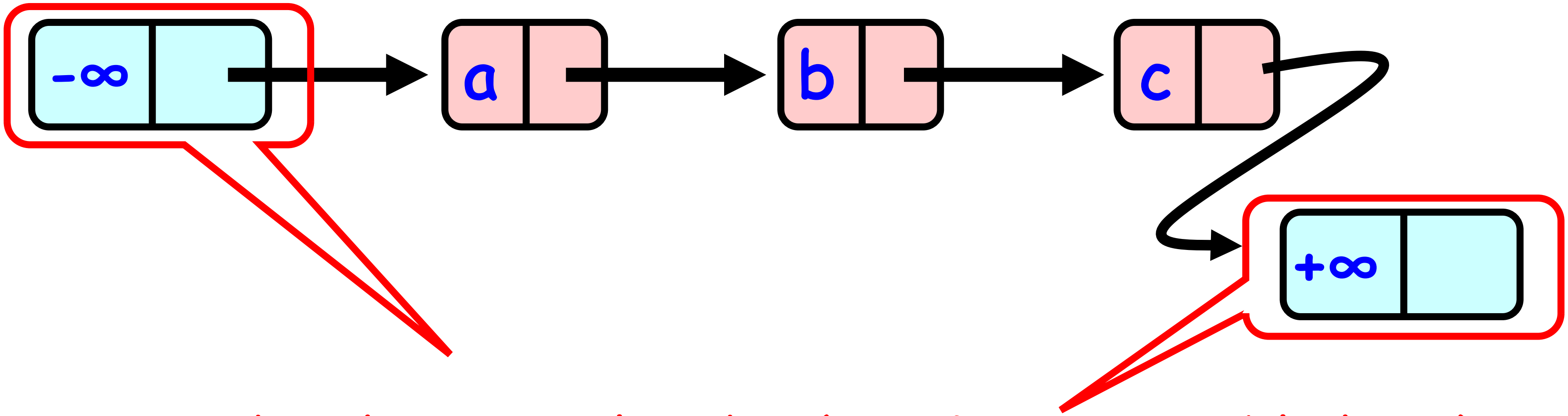
- Procrastinate! Procrastinate! Procrastinate! ☺
- Make common operations fast
- Postpone hard work
  - E.g., removing components is tricky… use two phases:
    - Logical removal
      - Mark component to be deleted
    - Physical removal
      - Do what needs to be done to remove the component
- Evaluation
  - ✓ Recheck after locking is simpler (just check nodes are unmarked)
  - ✓ Also usually cheaper than hand-over-hand locking
  - ✗ Mistakes are expensive (safety easily compromised)
  - ✗ Is not starvation free on *add* and *remove* (liveness compromised)
  - ✓ (List is starvation free on *contains*)

Lock-Free Synchronization

- Don't use locks at all… never!
  - Use compareAndSet() & relatives …
- Advantages
  - ✓ No scheduler assumptions/support
- Disadvantages
  - ✗ Very complex
  - ✗ Sometimes high overhead
  - ✗ Mistakes are very expensive (safety and liveness)

Linked List

- Illustrate these patterns …
- Using a list-based Set
  - Common application
  - Building block for other apps



Sorted with Sentinel nodes (min & max possible keys)

Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - *add(x)* put x in set *true if x was not in the set*
  - *remove(x)* take x out of set *true if x was in the set*
  - *contains(x)* tests if x in set *true if x is in the set*

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Add item to set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(Tt x);
}
```

Remove item from set

List-Based Sets

```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Is item in set?

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

List Node

```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

item of interest

List Node

```
public class Node {
  public T item;
  public int key;
  public Node next;
}
```

Usually hash code

List Node

```
public class Node {
  public T item;
  public int key;
  public Node next;
}
```

Reference to next node

Optimistic Concurrency List

- Traverse the list without locking until location is found
- Lock node(s)
- Validate
  - Traverse again to confirm that the locked nodes are still in the list
- Do the operation

Optimistic Add

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    return false;
                } else {
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Calculate hash

Try until
success or failure

Optimistic Add

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    return false;
                } else {
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Initialize pointers to traverse the list

Traverse the list looking for 'item'

Lock the nodes

Try the operation and either succeed or fail

Always unlock (with both success and failure)

Optimistic Add

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    return false;
                } else {
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

If the locked nodes are still accessible, that means they are still in the list

If item already in list, fail

If item not present, create new node insert into the list, and succeed

Remember: always unlocking (with both success and failure)

Optimistic Validate

```
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    return false;
                } else {
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

```
private boolean validate (Node pred,
                          Node curr) {
    Node node = head;
    while (node.key <= pred.key) {
        if (node == pred) {
            return pred.next == curr;
        }
        node = node.next;
    }
    return false;
}
```

Traverse the list looking for both 'pred' and 'curr'

Fail if 'pred' is not found

Optimistic Remove

```
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    pred.next = curr.next;
                    return true;
                } else {
                    return false;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Calculate hash

Try until
success or failure

Optimistic Remove

```
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    pred.next = curr.next;
                    return true;
                } else {
                    return false;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Initialize pointers to traverse the list

Traverse the list looking for 'item'

Lock the nodes

Try the operation and either succeed or fail

Always unlock (with both success and failure)

Optimistic Remove

```
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    pred.next = curr.next;
                    return true;
                } else {
                    return false;
                }
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

If the locked nodes are still accessible, that means they are still in the list

If item already in list, remove node and succeed

If item not present, fail

Remember: always unlocking (with both success and failure)

Optimistic Contains

```java
public boolean contains(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                return (curr.key == key);
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Calculate hash

Try until
success or failure

Optimistic Contains

```
public boolean contains(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                return (curr.key == key);
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Initialize pointers to traverse the list

Traverse the list looking for 'item'

Try the operation and either succeed or fail

Always unlock (with both success and failure)

Optimistic Contains

```
public boolean contains(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        curr.lock();
        try {
            if (validate(pred, curr)) {
                return (curr.key == key);
            }
        } finally {
            pred.unlock();
            curr.unlock();
        }
    }
}
```

Lock the nodes

If the locked nodes are still accessible, that means they are still in the list

Return success if item found, and failure otherwise

Remember: always unlocking (with both success and failure)

The END