Advanced Locking Techniques

- Coarse-grained Synchronization
- Fine-grained Synchronization
- Optimistic Synchronization
- Lazy Synchronization

List-based Set

```java
class node {
    T item; int key; Node next;
}
```
Coarse-grained Synchronization

• Take a sequential implementation, add a lock field, and ensure that each method call acquires and releases this lock
• Simple, and works well when levels of concurrency is low
• When too many threads try to access the object at the same time, the object becomes a sequential bottleneck

Coarse-grained Locking

```java
class CoarseList<T> {
    private Node head;
    private Lock lock = new ReentrantLock();
    public CoarseList() {
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }
    public boolean add(T item) {
        lock.lock();
        try {
            Node pred, curr;
            int key = item.hashCode();
            lock.lock();
            try {
                pred = head;
                curr = pred.next;
                while (curr.key < key) {
                    pred = curr;
                    curr = curr.next;
                }
                if (key == curr.key) {
                    return false;
                } else {
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            } finally {
                lock.unlock();
            }
        } finally {
            lock.unlock();
        }
    }
    public boolean remove(T item) {
        Node pred, curr;
        int key = item.hashCode();
        lock.lock();
        try {
            pred = head;
            curr = pred.next;
            while (curr.key < key) {
                pred = curr;
                curr = curr.next;
            }
            if (key == curr.key) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        } finally {
            lock.unlock();
        }
    }
}
```
Fine-grained Synchronization

- Improve concurrency by locking individual nodes, instead of locking the entire list
- Operations interfere only when trying to access the same node at the same time
- Higher concurrency, but requires a lock to be added for each node

Fine-grained Locking
Why Two Locks?

Hand-Over-Hand Locking

Except for the initial head node, acquire the lock for curr only while holding the lock for pred.
Optimistic Synchronization

- Search a component without acquiring any locks
- When a node is found, it locks the component, and then checks whether the component has been changed
- Optimistic about the possibility of conflicting actions

Optimistic Locking (1)

```java
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();
        }
    }
}
```

```java
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();
        }
    }
}
```
**Optimistic Locking (2)**

```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;
        }
        try {
            pred.lock (); curr.lock ();
            if (validate(pred, curr)) {
                return (curr.key = key);
            }
        }
        finally {
            pred.unlock (); curr.unlock ();
        }
    }
}
```

```java
public 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;
}
```

**Why validation?**

![Diagram showing the validation process in an optimistic locking scheme.](diagram.png)
Lazy Synchronization

- Postpone significant changes, e.g., physically removing a node from a linked list
- Allow a list to be traversed only once without locking, and contains() is wait-free.
- Require a new field marked to be added into each node

Lazy Locking (1)

```java
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();  
        try {  
            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 {  
                curr.unlock();  
            }  
        } finally {  
            pred.unlock();  
        }  
    }
}
```

```java
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();  
        try {  
            curr.lock();  
            try {  
                if (validate(pred, curr)) {  
                    if (curr.key != key) {  
                        return false;  
                    } else {  
                        curr.marked = true;  
                        pred.next = curr.next;  
                        return false;  
                    }  
                }  
            } finally {  
                curr.unlock();  
            }  
        } finally {  
            pred.unlock();  
        }  
    }
}
```
## Lazy Locking (2)

```java
public boolean contains (T item) {
    int key = item.hashCode ();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
```

```java
private boolean validate (Node pred, Node curr) {
    return !pred.marked && !curr.marked &&
           pred.next = curr;
}
```

---

### Can we do better?

- Non-blocking synchronization: eliminate locks entirely, relying on built-in atomic operations such as compareAndSet() for synchronization