Concurrent Programming

Dr A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline
- Synchronization primitive: TAS, TTAS, BTAS
- Example of Parallel Programming
  - Shared memory: C/C++ Pthread, C++11 thread, OpenMP, Cilk
  - Distributed Memory: MPI
- Concurrent Objects
  - Concurrent, List, Q, stack, Tree, Priority Queue, Hash, SkipList
- Use of Concurrent objects

Guideline for designing CDS
- Ensure no thread can see a state where invariant of the DS have broken by the action of other thread.
- Take care to avoid race condition inherent in the interface to the DS by proving function for complete functions for the operation rather than for operation steps. Race condition (unexpected output)
- Pay attention to how the DS behaves in the presence of exceptions to ensure that the invariants are not broken.
- Minimized the opportunities for deadlock when using the DS by restricting the scope of locked and avoid nested locks where possible.

Enabling genuine concurrent access to CDS
- Can the scope of locks be restricted to allow some parts of an operation to e performed outside the lock?
- Can different part of the DS be protected by different locks?
- Do all operation require the same level of protection?
- Can a simple change to the DS improve the improve the opportunities for concurrency without affecting the operational semantics?

CDS: Concurrent List

Last Lecture: Spin-Locks

- Resets lock upon exit
- spin lock critical section
- spin lock
**Coarse-Grained Synchronization**

- Each method locks the object
  - Avoid contention using queue locks
  - Easy to reason about
    - In simple cases
- So, are we done?

- NOPE

**Coarse-Grained Synchronization**

- Sequential bottleneck
  - Threads “stand in line”
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel ...

**First: Fine-Grained Synchronization**

- Instead of using a single lock ...
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component ...
  - At the same time

**Second: Optimistic Synchronization**

- Search without locking ...
- If you find it, lock and check ...
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking, but
  - Mistakes are expensive

**Third: Lazy Synchronization**

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done
**Fourth: Lock-Free Synchronization**

- Don’t use locks at all
  - Use compareAndSet() & relatives ...
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

**Linked List based set: Interface**

- Unordered collection of items
  - Sorted w. r. t. Key
  - Key is hash code of Value
- No duplicates
- Methods
  - add(x) put x in set
  - remove(x) take x out of set
  - contains(x) tests if x in set

**List-Based Sets and Nodes**

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

public class Node {
    public T item;    // item of interest
    public int key; // usually hash code
    public Node next; // ref to next node
}
```

**The List-Based Set**

Sorted with Sentinel nodes
(min & max possible keys)

**Reasoning about Concurrent Objects**

- Invariant
  - Property that always holds
- Established because
  - True when object is created
  - Truth preserved by each method
    - Each step of each method

**Specifically ...**

- Invariants preserved by
  - add()
  - remove()
  - contains()
- Most steps are trivial
  - Usually one step tricky
  - Often linearization point
Interference

- Invariants make sense only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List nodes not visible outside class

Interference

- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with malloc() & free()!
- Garbage collection helps here

Representative Invariant

- Which concrete values meaningful?
  - Sorted?
  - Duplicates?
- Rep invariant
  - Characterizes legal concrete reps
  - Preserved by methods
  - Relied on by methods

Blame Game

- Suppose
  - add() leaves behind 2 copies of x
  - remove() removes only 1
- Which is incorrect?
  - If rep invariant says no duplicates
  - add() is incorrect
    - add() is incorrect
  - Otherwise
    - remove() is incorrect

Rep Invariant (partly)

- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

Abstraction Map

- $S(head) = \{ x \mid \text{there exist a such that}$
  - a reachable from head and
    - a.item = x
  - $\}$
Sequential List Based Set
Add()

Remove()

Coarse-Grained Locking

One lock for whole list:
to access one need to hold the lock

Coarse Grain: add

public boolean add(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock(); //Lock the whole list
    try {
        pred = head; curr = pred.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        if (key==curr.key) {return false; } //Item key exist
        else {
            Node node = new Node(item);
            node.next = curr; pred.next = node;
            return true;
        }
    }
    finally { lock.unlock(); } //Unlock the whole list
}

Coarse Grain: remove

public boolean remove(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock(); //Lock the whole list
    try {
        pred = this.head; curr = pred.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        if (key==curr.key) { // present
            pred.next = curr.next;
            return true;
        } else { return false; } // not present
    }
    finally { lock.unlock(); } //always unlock
}
**Coarse Grain: Contains**

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

**Coarse-Grained Locking**

- Easy, same as synchronized methods
  - “One lock to rule them all ...”
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
    - Also queue with backoff
    - But bottleneck still an issue

**Fine-grained Locking**

- Instead of using a single lock ...
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component ...
  - At the same time

**Fine-Grained Synchronization**

- Requires careful thought
  - “Do not meddle in the affairs of wizards, for they are subtle and quick to anger”
- Split object into pieces
  - Each piece has own lock: Every node of list have a lock
  - Methods that work on disjoint pieces need not exclude each other
Hand-over-Hand locking

Removing Node Sequentially
Why hold 2 locks?
Concurrent Removes with one lock per node
Concurrent Removes

Art of Multiprocessor Programming

Uh, Oh
Uh, Oh
Bad news, c not removed

Problem
• To delete node c
  — Swing node b’s next field to d
• Problem is,
  — Someone deleting b concurrently could
direct a pointer to c

Remove and Add
• Both addition and remove require to hold
  — Lock of curr and lock of prev

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

public boolean contains(T item) {
  Node last = null, pred = null, curr = null;
  int key = item.hashCode();
  head.lock();
  try {
    Node curr = head; curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock(); pred = curr;
        curr = curr.next; curr.lock();
      }
      return (curr.key == key);
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
**Agrue: Finalist is Starvation free**

- Sub problems: Agrue the finalist is deadlock free
  - All the methods acquires locks in the same down-the-list order, deadlock is impossible
  - Simply: no circular wait
- As there is no deadlocks, eventually all locks held by thread ahead of A in the list will be released
  - And A will succeed in locking \( \text{pred}_A \) and \( \text{curr}_A \)

**Drawbacks**

- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Inefficient
  - Thread removing ‘z’ need to wait for removal of ‘a’
  - A thread removing the second item in the list blocks all concurrent threads searching for later nodes.

---

**Optimistic Synchronization**

- Find nodes without locking
- Lock nodes
  - \( \text{prev} \) and \( \text{curr} \)
- Validate: Check that everything is OK
  - Check that \( \text{prev} \) and \( \text{curr} \) are still in list and adjacent or whether \( \text{prev} \) and \( \text{curr} \) have changed

- Validate : Guarantee freedom from interference

---

**Optimistic Traverse without Locking**

---

```java
/* Check that prev and curr are still in list and adjacent whether predecessor and current have changed */
private boolean validate(Entry pred, Entry curr) {
    Entry entry = head;
    while (entry.key <= pred.key) {
        if (entry == pred)
            return pred.next == curr;
        entry = entry.next;
    }
    return false;
}
```

```java
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        pred = this.head; curr = pred.next;
        while (curr.key <= key) {//Traverse without locking
            pred = curr; curr = curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred, curr)) {//Validate
                if (curr.key == key) 
                    return false; // present
                else 
                    entry = new Entry(item); // not present
                    entry.next = curr; pred.next = entry;
                    return true;
            }
        } finally {
            pred.unlock(); curr.unlock(); // unlock
        }
    }
}
```
**Optimistic List: remove()**

```java
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        pred = this.head; curr = pred.next;
        while (curr.key < key) { // Traverse without locking
            pred = curr; curr = curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) { // present in list
                    pred.next = curr.next;
                    return true;
                } else { return false; } // not present in list
            }
        } finally { pred.unlock(); curr.unlock(); } // unlock
    }
}
```

**Optimistic List: Contains()**

```java
public boolean contains(T item) {
    int key = item.hashCode();
    while (true) {
        pred = this.head; curr = pred.next;
        while (curr.key < key) { // Traverse without locking
            pred = curr; curr = curr.next;
        }
        try {
            pred.lock(); curr.lock();
            if (validate(pred, curr)) {
                return (curr.key == key);
            }
        } finally { pred.unlock(); curr.unlock(); }
    }
}
```

**Optimistic List: Summary**

- In absence of contention (at Curr and Pred), methods traverse list once otherwise many times
- Optimistic List is not Starvation free
  - A thread might delayed for ever if new node are repeatedly added and removed

```java
while (1) { // This will block other threads
    Optlist.add(X); Optlist.remove(X);
}
```

- This algorithm expect to do well in practice, since starvation is rare

**Optimistic List**

- Limited hot-spots
  - Targets of add(), remove(), contains()
  - No contention on traversals
- Moreover
  - Traversals are wait-free

**Optimistic List: So Far, So Good**

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - contains() method acquires locks

**Optimistic List: Evaluation**

- Optimistic is effective if
  - Cost of scanning twice without locks is less than cost of scanning once with locks
- Drawback
  - Contains() acquires locks
  - 90% of calls in many apps
**Lazy List**

- Like optimistic, except
  - Scan once
  - `contains(x)` never locks ...
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

**Lazy Synchronization List**

- Postpone hard work
  - Remove is hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done

**Lazy List**

- `remove()`
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor’s next (as before)

**Lazy Removal**

- Present in list
- Logically deleted
Physically deleted

**Lazy Removal**

- All Methods
  - Scan through locked and marked nodes without locking
  - Removing a node doesn’t slow down other method calls ...
- Must still lock pred and curr nodes.

**Validation**

- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

**LazyList: Add()**

```java
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        pred = this.head; curr = head.next;
        while (curr.key < key) {
            // Traverse without locking
            pred = curr; curr = curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                if (validate(pred, curr)) {
                    if (curr.key == key) return false;
                    else { Node NN = new Node(item);
                        NN.next = curr; pred.next = NN;
                        return true;
                    }
                }
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
}
```

**LazyList: remove()**

```java
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        pred = this.head; curr = head.next;
        while (curr.key < key) {
            // Traverse without locking
            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; // logically remove
                        pred.next = curr.next; // physically remove
                        return true;
                    }
                }
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
}
```
LazySyncList: contains()

public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) // Traverse without locking
        curr = curr.next;
    return curr.key == key && !curr.marked;
}

Summary: Wait-free Contains

Use Mark bit + list ordering
1. Not marked → in the set
2. Marked or missing → not in the set

Lazy List

Lazy add() and remove() + Wait-free contains()

Evaluation

• Good:
  – contains() doesn’t lock
  – In fact, its wait-free!
  – Good because typically high % contains()
  – Uncontended calls don’t re-traverse
• Bad
  – Contended add() and remove() calls do re-traverse
  – Traffic jam if one thread delays

Traffic Jam

• Any concurrent data structure based on mutual exclusion has a weakness
• If one thread
  – Enters critical section
  – And “eats the big muffin”
    • Cache miss, page fault, descheduled …
    – Everyone else using that lock is stuck!
  – Need to trust the scheduler….

Reminder: Lock-Free Data Structures

• No matter what ...
  – Guarantees minimal progress in any execution
  – i.e. Some thread will always complete a method call
  – Even if others halt at malicious times
  – Implies that implementation can’t use locks
**Locks: Internal**

- Lock Type
  - Spin lock TAS, TTAS, TTAS-Backoff
- Every one use of Atomic operation
- Lets define our own lock to suite our data structure bases on Atomic operation
- Instead of using Lock, let us use Atomic Operation
- Java provides operations like
  - CompareAndSet, attemptMark, get

**Lock-free Lists**

- Next logical step
  - Wait-free contains()
  - lock-free add() and remove()
- Use only compareAndSet()
  - Simple compareAndSet may go wrong...
  - Uses Atomic Markable Reference
  - This encapsulate both object reference and a Boolean field Mark and these field can be updated atomically.

**Thanks**