Concurrent Algorithms and Data Structures

Yuanhao Wei
MIT 6.506 Algorithm Engineering
Concurrent Data Structure

Shared Data

Operation 1

Operation 2

Operation 3

Being applied simultaneously

Users
Examples

Database System

Users

Transaction 1

Transaction 2

Transaction 3

Other Applications: Operating Systems, Parallel Schedulers
Work Stealing – Concurrent Deque

Source: https://actor-framework.readthedocs.io/en/0.17.5/Scheduler.html
Example – Concurrent BST

Challenge:
• Ensuring data structure remains in a consistent state
• Return values are correct
• All processes make progress
Outline

• Asynchronous Shared Memory Model
• Correctness Conditions – Linearizability, Serializability
• Lock-based techniques
  • Hand-over-hand locking
  • Lock-free searches
  • Optimistic locking
• Lock-free techniques
  • Helping
  • Harris linked list
Asynchronous Shared Memory

• Processes communicate through shared variables

• Adversarial scheduler interleaves steps by the processes

• Processes can be arbitrarily slow or crash (never scheduled again)
Shared Variables

**Read-Write Variable:** \( \text{Read}(X), \text{Write}(X, \text{value}) \)

**Lock:** \( \text{Lock}(L), \text{Unlock}(L) \)

**Compare-and-Swap (CAS) Variable:**
\[ \text{Read}(X), \text{CAS}(X, \text{oldValue, newValue}) \]

\[
\text{if Read}(X) == \text{oldValue} \\
\text{Write}(X, \text{newValue}) \\
\text{return true} \\
\text{else return false}
\]
Example: Concurrent Counter

Increment():
   y = read(C)  // C is a shared variable, initially 0
   write(C, y+1)

Thread 1
For i = 1 to 10:
   Increment()

Thread 2
For i = 1 to 10:
   Increment()

When both threads complete, what are the possible values of C?
Correctness: Linearizability

A concurrent data structure is **linearizable** if we can assign linearization points to each operation such that:

1. The linearization point of each operation lies between the invocation and response of that operation
2. The operations appear to be applied sequentially, ordered by their linearization points
A concurrent data structure is **sequentially consistent** if we can order the operations such that:

1. The operations appear to be applied sequentially, according to this order
2. This order is consistent with the program order of each process
Linearizability vs Sequential Consistency

• Aka “strict serializability” vs “serializability” in database community

Sequential Consistency is not composable but Linearizability is!
Outline

- Asynchronous Shared Memory Model
- Correctness Conditions – Linearizability, Sequential Consistency
- Lock-based Linked Lists
  - Hand-over-hand locking
  - Lock-free searches
  - Optimistic locking
- Lock-free Linked Lists
  - Helping
  - Harris linked list
Set Abstract Data Type

We will cover several ways of implementing this using a concurrent linked list.

These techniques generalize to other pointer-based data structures like binary trees (balanced and unbalanced), b-trees, radix trees, etc.

```java
public interface Set<T> {
    boolean add(T x);
    boolean remove(T x);
    boolean contains(T x);
}
```
Concurrent Linked List: Challenges

Consider a Linked List where:
• Process 1 wants to remove C
• Process 2 wants to add D

Asynchrony: no assumption about relative speed of processes

1. Process 1 pauses right before writing the pointer to E
2. Process 2 adds node D
3. Process 1 unpauses and accidentally removes D as well as C. Incorrect!
Lock-based Solutions

• Corse-grained Locking:

• Fine-grained Locking:
Hand-over-hand Locking

```java
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();
            }
        }
    } finally {
        curr.unlock();
    }
    if (curr.key == key) {
        return false;
    }
    Node newNode = new Node(item);
    newNode.next = curr;
    pred.next = newNode;
    return true;
}
```
Hand-over-hand Locking

- **Invariants**: our two locked nodes are always adjacent and guaranteed to be reachable from the root

- At first glance, this looks fairly scalable, many operations can proceed in parallel
- What’s the problem?
Principles for Efficient Locking

1. Don’t hold too many locks
2. Don’t hold a lock for too long
3. Only lock the locations you plan to write to

Deadlock prevention: acquire locks in a consistent order
Cache Coherency

Source: https://www.sciencedirect.com/topics/engineering/cache-coherence
Path-copy runs into a similar issue
Lock-free contains()

- Even lock-based data structures at least want their read-only operations to be lock-free
- A sequential contains() algorithm basically works in the concurrent setting

```java
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key)
        curr = curr.next;
    return curr.key == key
}
```
Lock-free contains(): Linearizability

• There are times during the contains() where curr is not reachable from the root
• Therefore, we cannot linearize the contains() when it returns
• What’s the correct linearization point?

```java
def public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key
}
```
Speeding-up Updates: Optimistic Locking

1. Traverse optimistically without locks until you reach a node you wish to update
2. Lock neighborhood of node
3. Validate neighborhood
4. Perform updates
5. Release Locks

Validation is necessary to make sure the nodes you locked are still reachable.
A “removed” bit is added to check reachability.

Appeared as early as the 1980s, re-invented many times since then.
Go-to technique for almost all pointer-based data structures.
Speeding-up Updates: Optimistic Locking

```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();
        }
    }
}

private boolean validate(Node pred, Node curr) {
    return !pred.marked && !curr.marked && pred.next == curr;
}
```
Updated Lock-free contains()

```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;
}
```
Outline

• Asynchronous Shared Memory Model
• Correctness Conditions – Linearizability, Sequential Consistency
• Lock-based Linked Lists
  • Hand-over-hand locking
  • Lock-free contains
  • Optimistic locking
• Lock-free Linked Lists
  • Helping
  • Harris linked list
Progress Guarantee: Lock-freedom

Definition: some operation eventually completes regardless of how processes are scheduled

- Must hold for an adversarial scheduler
- Disallows a process from waiting for another process to take a step
History of Lock-freedom

- Dijkstra introduces mutual exclusion
- Scalable lock-based binary search trees developed
  Lots of work on lock-freedom
- First scalable lock-free binary search tree

Lock-free programming is hard!

1965
1980
...
...
2010
Shared Variables

Read-Write Variable: Read(X), Write(X, value)

Lock: Lock(L), Unlock(L)

Compare-and-Swap (CAS) Variable: Read(X), CAS(X, oldValue, newValue)

if Read(X) == oldValue
    Write(X, newValue)
    return true
else
    return false
Lock-freedom: Key Ideas

• Most shared writes should be done with **CAS**

• Update operations should become **visible with a single CAS**
  • E.g. instead of updating the fields of a node one by one, create a new copy of the node and install it atomically

• **Helping**: updates operations might temporarily leave data structure in an inconsistent state, if you see this, help complete their operation.
Lock-free Linked List

• Lock-free contains() the same as before

• If we only need to support lock-free add(), then a sequential implementation with some writes replaced with CAS would be sufficient

• The tricky part is supporting delete()s
Lock-free Linked List

• Key idea: deletes “freeze” the node being deleted before physically removing it

• This “freeze” prevents any other process from making modifications to it

• If other processes come across a frozen node, they have to help remove it to prevent it from blocking their progress

Source: https://concurrencyfreaks.blogspot.com/2014/03/harriss-linked-list.html
Outline

• Asynchronous Shared Memory Model
• Correctness Conditions – Linearizability, Sequential Consistency
• Lock-based Linked Lists
  • Hand-over-hand locking
  • Lock-free contains
  • Optimistic locking
• Lock-free Linked Lists
  • Helping
  • Harris linked list

These techniques can be applied to a wide range of data structures
Topic I didn’t get to cover

- Concurrent Memory Management
- Weak Memory Models
- Proving Correctness
- Consensus Hierarchy
- More complex concurrent data structures

- Lots of open problems in this area