Linked Lists: Locking, Lock-Free, and Beyond ...

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

• Concurrent and distributed programming limitations
  - Amdahl’s vs. Gustafson’s law
  - Memory architecture: UMA, NUMA and distributed (latency vs. bandwidth)
  - Cache coherence and false sharing
• Problem partitioning/decomposition
  - SIMD vs. MIMD
  - Granularity and load balancing
  - Task-management/synchronization/communication overhead vs. strugglers.
• Concurrent and distributed libraries
  - OpenMP: shared memory
  - MPI: distributed memory (tutorials)
• Virtual time:
  - Order of distributed events and gathering distributed information
  - Data Race Detection
This Lecture

• Introduce “patterns”
  - Bag of tricks …
  - Methods that work more than once …

• For highly-concurrent objects

• Goal:
  - Concurrent access
  - More threads, more throughput
Linked List

• Illustrate these patterns ...
• Using a list-based Set
  – Common application
  – Building block for other apps
Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - \texttt{add(x)} put \( x \) in set
  - \texttt{remove(x)} take \( x \) out of set
  - \texttt{contains(x)} tests if \( x \) in set
List Node

```java
public class Node {
    public T item;
    public int key;
    public Node next;
}
```
The List-Based Set

Sorted with Sentinel nodes (min & max possible keys)
Reasoning about Concurrent Objects

• Invariant
  - Property that always holds

• Established by
  - True when object is created
  - Truth preserved by each method
    • Each step of each method
Interference

• Invariants make sense only if
  - methods considered are the only modifiers

• Language encapsulation helps
  - List nodes not visible outside class
Specifically …

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

• Which concrete values meaningful?
  - Sorted?
  - Duplicates?

• Rep invariant
  - Characterizes legal concrete reps
  - Preserved by methods
  - Relied on by methods
Blame Game

• Rep invariant is a contract
• Suppose
  – `add()` leaves behind 2 copies of x
  – `remove()` removes only 1
• Which one is incorrect?
Blame Game

• Suppose
  – add() leaves behind 2 copies of x
  – remove() removes only 1

• Which one is incorrect?
  – If rep invariant says no duplicates
    • add() is incorrect
  – Otherwise
    • remove() is incorrect
Rep Invariant (partly)

• Sentinel nodes
  - tail reachable from head

• Sorted

• No duplicates
Abstraction Map

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

Add(b)

Remove(b)
Removing a Node

remove(c)

remove(b)
Removing a Node

\[ \text{remove}(b) \]

\[ \text{remove}(c) \]
Removing a Node

![Diagram showing the removal of nodes from a linked list]

- remove(b)
- remove(c)
Uh, Oh

Bad news

remove(b)

remove(c)
Coarse Grained Locking
Coarse Grained Locking

Simple...
Coarse Grained Locking

Simple but hotspot + bottleneck
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
  - But bottleneck still an issue
Fine-grained Locking

• 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
  - Methods that work on disjoint pieces need not exclude each other
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Removing a Node

remove(c)

remove(b)
Removing a Node

remove(b)

remove(c)
Removing a Node

remove(b)

remove(c)
Removing a Node

- remove(b)
- remove(c)
Removing a Node

- remove(b)
- remove(c)
Removing a Node

remove(b)

remove(c)
Removing a Node

- remove(b)
- remove(c)
Removing a Node

```
remove(b)
```

```
remove(c)
```
Uh, Oh

Bad news

remove(b)

remove(c)
Problem

• To delete node b
  - Swing node a’s next field to c

• Problem is,
  - Someone could delete c concurrently
Insight

• If a node is locked
  - No one can delete node’s successor

• If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works
Hand-Over-Hand Again

remove(b)
Hand-Over-Hand Again

remove(b)
Hand-Over-Hand Again

remove(b)
Hand-Over-Hand Again

remove(b)

Found it!
Hand-Over-Hand Again

remove(b)

Found it!
Hand-Over-Hand Again

remove(b)
Removing a Node

\[ \text{remove}(b) \]

\[ \text{remove}(c) \]
Removing a Node

\[ \text{remove}(b) \]

\[ \text{remove}(c) \]
Removing a Node

remove(b)

remove(c)
Removing a Node

```
remove(b)
```

```
remove(c)
```
Removing a Node

remove(b)

remove(c)
Removing a Node

remove(b)

remove(c)
Removing a Node

remove(b)

remove(c)
Removing a Node

```
remove(b)
remove(c)
```
Removing a Node

remove(b)

remove(c)
Removing a Node

remove(b)

remove(c)
Removing a Node

remove(b)
Removing a Node

```
remove(b)
```
Removing a Node

```
remove(b)
```
Removing a Node

remove(b)

remove(c)
Removing a Node
Adding Nodes

• To add node e
  - Must lock predecessor
  - Must lock successor

• Neither can be deleted
  - (Is successor lock actually required?)
Same Abstraction Map

\[ S(\text{head}) = \neg \{ x \mid \text{there exists a such that} \]
\[ \quad \cdot \text{a reachable from head and} \]
\[ \quad \cdot \text{a.item} = x \]
\[ \neg \} \]
Rep Invariant

- Easy to check that
  - tail always reachable from head
  - Nodes sorted, no duplicates
Drawbacks

• Better than coarse-grained lock
  - Threads can traverse in parallel

• Still not ideal
  - Long chain of acquire/release
  - Inefficient
Optimistic Synchronization

- Find nodes without locking
- Lock nodes
- Check that everything is OK
Optimistic: Traverse without Locking

add(c)

Aha!
Optimistic: Lock and Load

add(c)
What Can Possibly Go Wrong?

add(c)
What Can Possibly Go Wrong?

add(c)

remove(b)
What Can Possibly Go Wrong?

```
add(c)
```
Validate (1)

Yes, b still reachable from head

add(c)
What Else Can Go Wrong?

```
\text{add}(c)
```
What Else Can Go Wrong?

```
add(c)
```

```
add(b')
```
What Else Can Go Wrong?

```
add(c)
```
Optimistic: Validate(2)

Yes, b still points to d
Optimistic: Linearization Point

add(c)
Same Abstraction Map

• $S(\text{head}) =$
  
  - $\{ x \mid \text{there exists } a \text{ such that}$
    
    • $a \text{ reachable from } \text{head}$ and
    
    • $a.\text{item} = x$

  - $\}$
Invariants

• Careful: we may traverse deleted nodes
• But we establish properties by
  – Validation
  – After we lock target nodes
Removing an Absent Node

remove(c)

Aha!
Validate (1)

Yes, b still reachable from head

remove(c)
Validate (2)

Yes, b still points to d

remove(c)
OK Computer

remove(c)

return true
Optimistic List

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

• Much less lock acquisition/release
  - Performance
  - Concurrency

• Problems
  - Need to traverse list twice
  - contains() method acquires locks
    • Most common method call
Evaluation

• Optimistic is effective if
  - cost of scanning twice without locks
    • 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 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
Lazy Removal

Present in list
Lazy Removal

Logically deleted
Lazy Removal

Physically deleted
Lazy Removal

Physically deleted
Business as Usual
Business as Usual
Business as Usual
Business as Usual

remove(b)
Business as Usual

a not marked
Business as Usual

a still points to b
Business as Usual

Logical delete
Business as Usual

physical delete
Business as Usual
New Abstraction Map

- \( S(\text{head}) = \{ x \mid \text{there exists node } a \text{ such that} \)
  
  - \( a \text{ reachable from } \text{head} \text{ and} \)
  
  - \( a\.item = x \text{ and} \)
  
  - \( a \text{ is unmarked} \)

- \}
Evaluation

• **Good:**
  - `contains()` doesn’t lock
  - In fact, it’s wait-free!
  - Good because typically high % contains()
  - Uncontended calls don’t re-traverse

• **Bad**
  - Contended 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 ...
    - Software error, ...
  - Everyone else using that lock is stuck!
Lock-Free Data Structures

• No matter what ...
  - Some thread will complete method call
  - Even if others halt at malicious times
  - Weaker than wait-free, yet

• Implies that
  - You can’t use locks (why?)
  - Um, that’s why they call it lock-free
Lock-free Lists

- Next logical step
- Eliminate locking entirely
- `contains()` wait-free and `add()` and `remove()` lock-free
- Use only `compareAndSet()`
  - Also called CAS
- What could go wrong?
CAS Reminder

```c
int atomicInc(int* i) {
    int oldVal, newVal;
    do {
        oldVal = *i;
        newVal = oldVal + 1;
    } while (!CAS(i, oldVal, newVal));
    return newVal;
}
```
Adding a Node
Adding a Node

```
  a -- b -- c -- d
```

BROWN
Adding a Node

CAS

b

c

d
Adding a Node
Removing a Node

remove b

remove c
Look Familiar?

Bad news

remove b

remove c
Problem

- Method updates node’s next field
- After node has been removed
Marking a Node

- AtomicMarkableReference class
  - Java.util.concurrent.atomic package
Solution

• **Use** `AtomicMarkableReference (CAS+)`

• **CAS+: Atomically**
  - Check mark bit in next field is unset
  - Check reference unchanged
  - If both true: Swing reference

• **Remove in two steps**
  - Set mark bit in next field
  - Redirect predecessor’s pointer using **CAS+**
Removing a Node

\[
\text{remove } c
\]
Removing a Node

- remove b
- remove c
- CAS+ failed
Removing a Node

remove b

remove c
Removing a Node

But failed CAS+ does not actually need to retry!
Traversing the List

- **Q**: what do you do when you find a “logically” deleted node in your path?
- **A**: finish the job.
  - CAS+ the predecessor’s next field
  - Proceed (repeat as needed)
Lock-Free Traversal

Uh-oh
Lock-free Removal

Logical Removal = Set Mark Bit

Use CAS+ to verify pointer is correct

Is it enough?
Lock-free Removal

Logical Removal = Set Mark Bit

Physical Removal CAS+

Node added Before Physical Removal CAS+

Problem:
d not added to list...
Must Prevent manipulation of removed node’s pointer
Solution: use CAS+ in ADD()

Logical Removal = Set Mark Bit

Physical Removal

FAIL CAS+: Node not added after logical Removal
Summary: A Lock-free Algorithm

1. `add()` and `remove()` physically remove marked nodes
2. `Wait-free contains()` traverses both marked and removed nodes
“To Lock or Not to Lock”

- Locking vs. Non-blocking: Extremist views on both sides
- The answer: nobler to compromise, combine locking and non-blocking
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Blocking/non-blocking is a property of a method
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - [http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/).
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.