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

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
This Lecture

• Introduce four “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**
  - `add(x)` put x in set
  - `remove(x)` take x out of set
  - `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
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
Rep Invariant

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

• Rep invariant
  - Characterizes legal concrete reps
  - Preserved by methods
  - Relyed 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 \text{ such that}
  \begin{itemize}
    \item \text{a reachable from head and}
    \item \text{a.item = x}
  \end{itemize}
\}
Sequential List Based Set

Add()

Remove()
Sequential List Based Set

Add()

Remove()
Coarse Grained Locking
Coarse Grained Locking
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

\[ \text{Diagram: Hand-overtaking sequence} \]
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Hand-over-Hand locking
Removing a Node

remove(b)
Removing a Node

remove(b)
Removing a Node

remove(b)
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

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

\[\text{remove(c)}\]
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)
Uh, Oh

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

- 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

- $\text{remove}(b)$
- $\text{remove}(c)$
Removing a Node

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

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

remove(b)

remove(c)
Removing a Node

![Diagram of removing a node](image)
Removing a Node

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

\texttt{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

- $S(\text{head}) = \{ x \mid \text{there exists an } a \text{ such that}
  \begin{align*}
  &\cdot a \text{ reachable from } \text{head} \text{ and} \\
  &\cdot a.\text{item} = x \\
  \end{align*}
\}$
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?

![Diagram showing a sequence of operations with a thought bubble saying `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?

```
add(c)
```
What Else Can Go Wrong?

- add(c)
- add(b')
What Else Can Go Wrong?

\[\text{add}(c)\]
Optimistic: Validate(2)

add(c)

Yes, b still points to d
Optimistic: Linearization Point

add(c)
Same Abstraction Map

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

![Diagram showing a lazy removal process with elements a, b, c, and d.]
Lazy Removal

Present in list
Lazy Removal

Logically deleted
Lazy Removal

Physically deleted
Lazy Removal

Physically deleted
Lazy List

• **All Methods**
  - Scan through locked and marked nodes
  - Removing a node doesn’t slow down other method calls ...

• **Must still lock** \textit{pred} and \textit{curr} nodes.
Validation

• No need to rescan list!
• Check that \texttt{pred} is not marked
• Check that \texttt{curr} is not marked
• Check that \texttt{pred} points to \texttt{curr}
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
Business as Usual
New Abstraction Map

\[ S(\text{head}) = \{ \ x \mid \text{there exists node } a \text{ such that} \]
\[ \quad \cdot a \text{ reachable from head and} \]
\[ \quad \cdot a.\text{item} = x \text{ and} \]
\[ \quad \cdot a \text{ is unmarked} \]
\[ \} \]
Wait-free Contains - No need to lock in "Contains"

1. Not marked $\rightarrow$ in the set
2. Marked or missing $\rightarrow$ not in the set

Use Mark bit + Fact that List is ordered
Lazy List

Lazy add() and remove() + Wait-free contains()
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()
- What could go wrong?
Adding a Node
Adding a Node

Diagram showing a linked list with nodes labeled a, b, and c. There is an additional node labeled b added to the list.
Adding a Node

![Diagram showing a CAS operation to add a node]

- CAS
- b
- c
- b

Art of Multiprocessor Programming © Herlihy-Shavit 2007
Adding a Node
Adding a Node
Removing a Node

\[ \text{remov } e \text{ } b \quad \rightarrow \quad \text{CAS } \quad \text{CAS } \quad \rightarrow \text{remov } e \text{ } c \]
Look Familiar?

Bad news

remov e b

remov e c
Problem

• Method updates node’s `next` field
• After node has been removed
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

Removing node c using CAS+ operation.
Removing a Node

- **CAS+**
- **failed**

- **remove b**
- **remove c**
Removing a Node

removing e b

removing 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

CAS+
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

Problem:
d not added to list...
Must Prevent manipulation of removed node’s pointer

Node added Before Physical Removal CAS+

Physical Removal CAS+
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
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/.
- 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.