Concurrent Singly-Linked Lists

Amgad Rady

DisCoVeri Group
Department of Electrical Engineering and Computer Science
York University

November 5, 2015

Outline

Introduction to Singly-Linked Lists

- Why implement concurrent singly-linked lists?
- SLL's are used to implement many abstract data types (LIFO and FIFO queues, disjoint sets.)
- SLL's are themselves part of larger data structures (hash tables, skip lists.)
- SLL's are simple.

Introduction to Singly-Linked Lists

- Why implement concurrent singly-linked lists?
- SLL's are used to implement many abstract data types (LIFO and FIFO queues, disjoint sets.)
- SLL's are themselves part of larger data structures (hash tables, skip lists.)
- SLL's are simple.

Introduction to Singly-Linked Lists

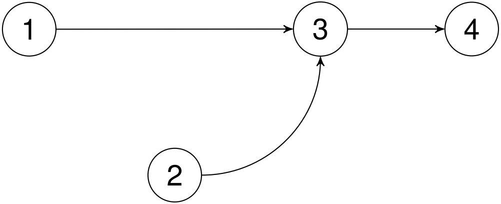
- Why implement concurrent singly-linked lists?
- SLL's are used to implement many abstract data types (LIFO and FIFO queues, disjoint sets.)
- SLL's are themselves part of larger data structures (hash tables, skip lists.)
- SLL's are simple.

Introduction to Singly-Linked Lists

- Why implement concurrent singly-linked lists?
- SLL's are used to implement many abstract data types (LIFO and FIFO queues, disjoint sets.)
- SLL's are themselves part of larger data structures (hash tables, skip lists.)
- SLL's are simple.

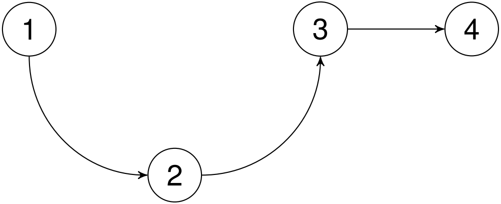
SLL Operations: INSERT

Inserting the node containing 2 into the list $\{1, 3, 4\}$. First, find the appropriate successor for 2 by searching the list from the head.



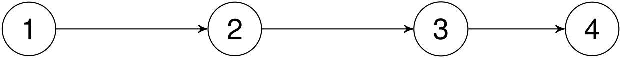
SLL Operations: INSERT

Inserting the node containing 2 into the list $\{1, 3, 4\}$. Next, swing the pointer from the predecessor (1) to the node (2).



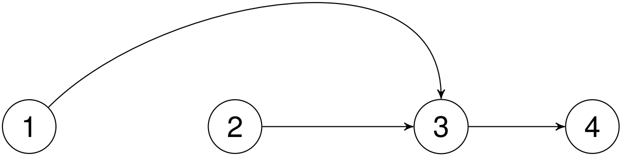
SLL Operations: DELETE

Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$. First, find the node's predecessor by searching the list from the head.



SLL Operations: DELETE

Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$.
Next, swing (2)'s predecessor's pointer to (2)'s successor.



Concurrent SLL’s

- Sequential singly-linked lists are a very simple data structure.
- We would like to be able to “lift” SLL’s into a concurrent setting without using expensive abstractions like locks, semaphores, monitors, etc.
- In addition to being costly, these abstractions do not have the property of *lock-freedom*.

Concurrent SLL's

- Sequential singly-linked lists are a very simple data structure.
- We would like to be able to “lift” SLL's into a concurrent setting without using expensive abstractions like locks, semaphores, monitors, etc.
- In addition to being costly, these abstractions do not have the property of *lock-freedom*.

Concurrent SLL's

- Sequential singly-linked lists are a very simple data structure.
- We would like to be able to "lift" SLL's into a concurrent setting without using expensive abstractions like locks, semaphores, monitors, etc.
- In addition to being costly, these abstractions do not have the property of *lock-freedom*.

Lock-freedom

Definition (Lock-freedom)

An algorithm is lock-free if at any configuration in an execution of the algorithm, if there is at least one processor that has not crashed then some processor will finish its operation in a finite number of steps.

Concurrent SLL's cont.

- Can we construct a concurrent implementation of SLL's using only COMPARE & SWAP?
- Yes! But it's very difficult.
- Let's consider a naïve implementation replacing READ's and WRITES's with COMPARE & SWAP.

Concurrent SLL's cont.

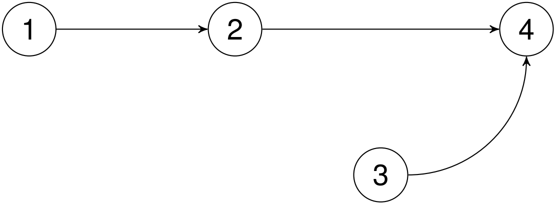
- Can we construct a concurrent implementation of SLL's using only COMPARE & SWAP?
- Yes! But it's very difficult.
- Let's consider a naïve implementation replacing READ's and WRITES's with COMPARE & SWAP.

Concurrent SLL's cont.

- Can we construct a concurrent implementation of SLL's using only COMPARE & SWAP?
- Yes! But it's very difficult.
- Let's consider a naïve implementation replacing READ's and WRITES's with COMPARE & SWAP.

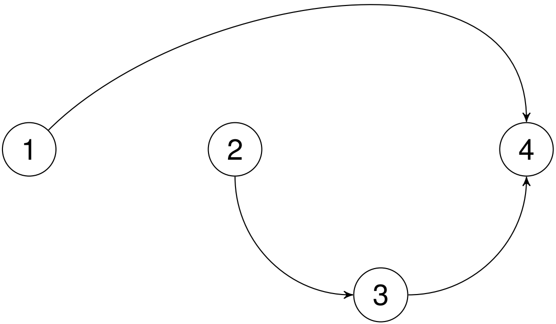
Concurrent INSERT and DELETE

We delete the node (2) and insert the node (3) concurrently into the list $\{1, 2, 4\}$.



Concurrent INSERT and DELETE

The resulting list is $\{1, 4\}$, rather than the correct $\{1, 3, 4\}$.



What Went Wrong?

- The issue in this example is that the INSERT procedure has no indication that the node (2) is about to be deleted.
- We can fix this by augmenting each node with a *mark* bit to indicate that the node is *logically* deleted before it is *physically* deleted.
- Once a node has been marked, its pointer cannot be changed.
- The next section presents a solution due to Timothy Harris.

What Went Wrong?

- The issue in this example is that the INSERT procedure has no indication that the node (2) is about to be deleted.
- We can fix this by augmenting each node with a *mark* bit to indicate that the node is *logically* deleted before it is *physically* deleted.
- Once a node has been marked, its pointer cannot be changed.
- The next section presents a solution due to Timothy Harris.

What Went Wrong?

- The issue in this example is that the INSERT procedure has no indication that the node (2) is about to be deleted.
- We can fix this by augmenting each node with a *mark* bit to indicate that the node is *logically* deleted before it is *physically* deleted.
- Once a node has been marked, its pointer cannot be changed.
- The next section presents a solution due to Timothy Harris.

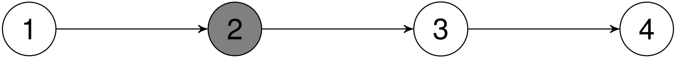
What Went Wrong?

- The issue in this example is that the INSERT procedure has no indication that the node (2) is about to be deleted.
- We can fix this by augmenting each node with a *mark* bit to indicate that the node is *logically* deleted before it is *physically* deleted.
- Once a node has been marked, its pointer cannot be changed.
- The next section presents a solution due to Timothy Harris.

DELETE Procedure

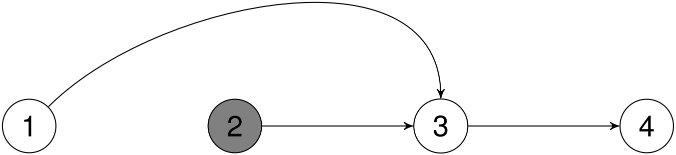
Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$.

Mark the node.

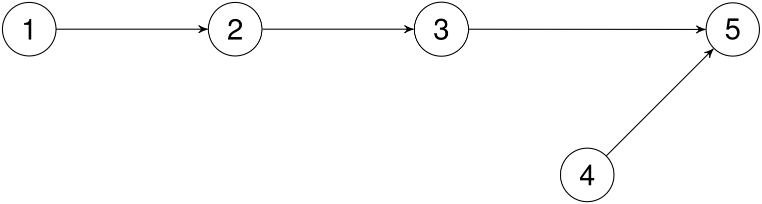


DELETE Procedure

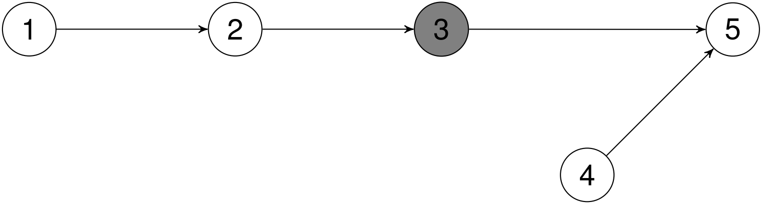
Deleting the node containing 2 from the list $\{1, 2, 3, 4\}$. Mark the node.



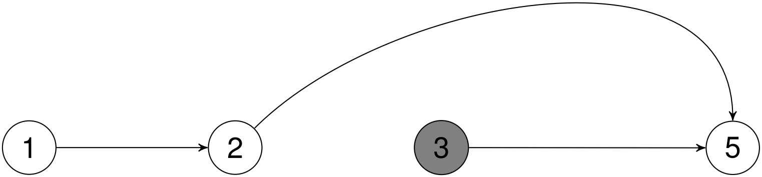
A Problematic Execution



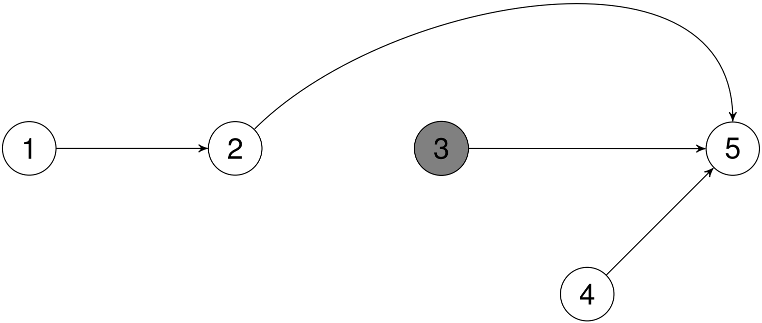
A Problematic Execution



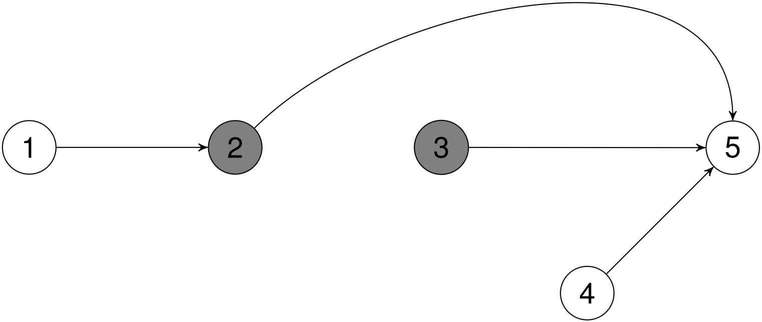
A Problematic Execution



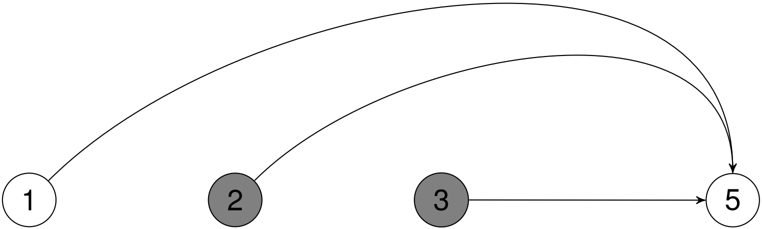
A Problematic Execution



A Problematic Execution



A Problematic Execution



A Problematic Execution

- In the worst case, INSERT can have $\Omega(n^2)$ running time.
- The problem here is that INSERT begins searching from the head of the list each time it finds a marked node.
- This can be solved by having marked nodes also point to their predecessors with a *backlink* pointer.
- But this is not *quite* enough as these backlinks can grow and affect asymptotic performance.
- We solve this by introducing a *flag* bit to each node to indicate that the successor is being deleted. A flagged node cannot be marked for the duration of the flag, which prevents the backlinks from growing.
- This solution is due to Fomitchev and Ruppert.

A Problematic Execution

- In the worst case, INSERT can have $\Omega(n^2)$ running time.
- The problem here is that INSERT begins searching from the head of the list each time it finds a marked node.
- This can be solved by having marked nodes also point to their predecessors with a *backlink* pointer.
- But this is not *quite* enough as these backlinks can grow and affect asymptotic performance.
- We solve this by introducing a *flag* bit to each node to indicate that the successor is being deleted. A flagged node cannot be marked for the duration of the flag, which prevents the backlinks from growing.
- This solution is due to Fomitchev and Ruppert.

A Problematic Execution

- In the worst case, INSERT can have $\Omega(n^2)$ running time.
- The problem here is that INSERT begins searching from the head of the list each time it finds a marked node.
- This can be solved by having marked nodes also point to their predecessors with a *backlink* pointer.
- But this is not *quite* enough as these backlinks can grow and affect asymptotic performance.
- We solve this by introducing a *flag* bit to each node to indicate that the successor is being deleted. A flagged node cannot be marked for the duration of the flag, which prevents the backlinks from growing.
- This solution is due to Fomitchev and Ruppert.

A Problematic Execution

- In the worst case, INSERT can have $\Omega(n^2)$ running time.
- The problem here is that INSERT begins searching from the head of the list each time it finds a marked node.
- This can be solved by having marked nodes also point to their predecessors with a *backlink* pointer.
- But this is not *quite* enough as these backlinks can grow and affect asymptotic performance.
- We solve this by introducing a *flag* bit to each node to indicate that the successor is being deleted. A flagged node cannot be marked for the duration of the flag, which prevents the backlinks from growing.
- This solution is due to Fomitchev and Ruppert.

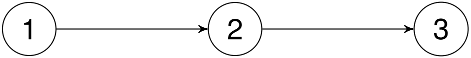
A Problematic Execution

- In the worst case, INSERT can have $\Omega(n^2)$ running time.
- The problem here is that INSERT begins searching from the head of the list each time it finds a marked node.
- This can be solved by having marked nodes also point to their predecessors with a *backlink* pointer.
- But this is not *quite* enough as these backlinks can grow and affect asymptotic performance.
- We solve this by introducing a *flag* bit to each node to indicate that the successor is being deleted. A flagged node cannot be marked for the duration of the flag, which prevents the backlinks from growing.
- This solution is due to Fomitchev and Ruppert.

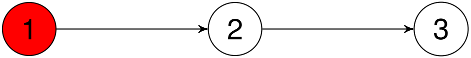
A Problematic Execution

- In the worst case, INSERT can have $\Omega(n^2)$ running time.
- The problem here is that INSERT begins searching from the head of the list each time it finds a marked node.
- This can be solved by having marked nodes also point to their predecessors with a *backlink* pointer.
- But this is not *quite* enough as these backlinks can grow and affect asymptotic performance.
- We solve this by introducing a *flag* bit to each node to indicate that the successor is being deleted. A flagged node cannot be marked for the duration of the flag, which prevents the backlinks from growing.
- This solution is due to Fomitchev and Ruppert.

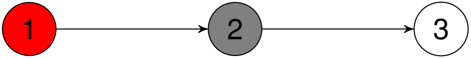
DELETE Procedure



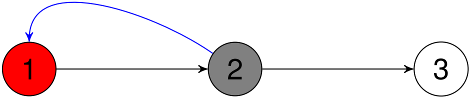
DELETE Procedure



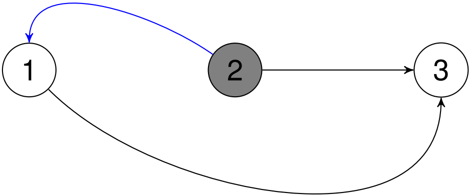
DELETE Procedure



DELETE Procedure



DELETE Procedure



F&R Performance

- What does this increase in complexity give us?
- In Harris's algorithm, the average cost of an operation is $\Omega(\bar{n} \cdot \bar{c})$ where $\bar{n}$ is the average length of the list during an execution and $\bar{c}$ is the average contention.
- In Fomitchev and Ruppert's algorithm, the average cost of an operation is $O(\bar{n} + \bar{c})$.
- Is this increase in performance worth the increase in complexity?

F&R Performance

- What does this increase in complexity give us?
- In Harris's algorithm, the average cost of an operation is $\Omega(\bar{n} \cdot \bar{c})$ where $\bar{n}$ is the average length of the list during an execution and $\bar{c}$ is the average contention.
- In Fomitchev and Ruppert's algorithm, the average cost of an operation is $O(\bar{n} + \bar{c})$.
- Is this increase in performance worth the increase in complexity?

F&R Performance

- What does this increase in complexity give us?
- In Harris's algorithm, the average cost of an operation is $\Omega(\bar{n} \cdot \bar{c})$ where $\bar{n}$ is the average length of the list during an execution and $\bar{c}$ is the average contention.
- In Fomitchev and Ruppert's algorithm, the average cost of an operation is $O(\bar{n} + \bar{c})$.
- Is this increase in performance worth the increase in complexity?

F&R Performance

- What does this increase in complexity give us?
- In Harris's algorithm, the average cost of an operation is $\Omega(\bar{n} \cdot \bar{c})$ where $\bar{n}$ is the average length of the list during an execution and $\bar{c}$ is the average contention.
- In Fomitchev and Ruppert's algorithm, the average cost of an operation is $O(\bar{n} + \bar{c})$.
- Is this increase in performance worth the increase in complexity?

Plan & Challenges

- Plan
  - Implement Fomitchev and Ruppert's algorithm and assess its performance on a massive number of threads, causing Manycore Testing Lab much grief.
- Challenges
  - Legal liability.
  - The usual challenges when implementing any non-trivial algorithm, except...
  - Java doesn't have Compare & Swap. It has the weaker primitive Compare & Set. Adapting the algorithm without introducing errors or degrading performance will be challenging.

Plan & Challenges

- Plan
  - Implement Fomitchev and Ruppert's algorithm and assess its performance on a massive number of threads, causing Manycore Testing Lab much grief.
- Challenges
  - Legal liability.
  - The usual challenges when implementing any non-trivial algorithm, except...
  - Java doesn't have COMPARE & SWAP. It has the weaker primitive COMPARE & SET. Adapting the algorithm without introducing errors or degrading performance will be challenging.

Plan & Challenges

- Plan
  - Implement Fomitchev and Ruppert's algorithm and assess its performance on a massive number of threads, causing Manycore Testing Lab much grief.
- Challenges
  - Legal liability.
  - The usual challenges when implementing any non-trivial algorithm, except...
  - Java doesn't have COMPARE & SWAP. It has the weaker primitive COMPARE & SET. Adapting the algorithm without introducing errors or degrading performance will be challenging.

Plan & Challenges

- Plan
  - Implement Fomitchev and Ruppert's algorithm and assess its performance on a massive number of threads, causing Manycore Testing Lab much grief.
- Challenges
  - Legal liability.
  - The usual challenges when implementing any non-trivial algorithm, except...
  - Java doesn't have COMPARE & SWAP. It has the weaker primitive COMPARE & SET. Adapting the algorithm without introducing errors or degrading performance will be challenging.

Plan & Challenges

- Plan
  - Implement Fomitchev and Ruppert's algorithm and assess its performance on a massive number of threads, causing Manycore Testing Lab much grief.
- Challenges
  - Legal liability.
  - The usual challenges when implementing any non-trivial algorithm, except...
  - Java doesn't have COMPARE & SWAP. It has the weaker primitive COMPARE & SET. Adapting the algorithm without introducing errors or degrading performance will be challenging.

Plan & Challenges

- Plan
  - Implement Fomitchev and Ruppert's algorithm and assess its performance on a massive number of threads, causing Manycore Testing Lab much grief.
- Challenges
  - Legal liability.
  - The usual challenges when implementing any non-trivial algorithm, except...
  - Java doesn't have COMPARE & SWAP. It has the weaker primitive COMPARE & SET. Adapting the algorithm without introducing errors or degrading performance will be challenging.

The End

Questions?