Concurrent Data Structures

Zhaoguo Wang
Acknowledgement

This slides are adapted from Maurice Herlihy & Nir Shavit

The Art of Multiprocessor Programming

Maurice Herlihy
Professor at Brown

Nir Shavit
Professor at MIT

https://en.wikipedia.org/wiki/Maurice_Herlihy
https://en.wikipedia.org/wiki/Nir_Shavit
Linked List

Assumption

– Sorted, no duplicates, an empty head node

Node
typedef struct node {
    struct node* next;
    int key;
} node;

Interfaces

– int add(node *head, int k)
– int remove(node *head, int k)
– int contain(node *head, int k)
int add(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
}

add(head, 3);
void add(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    while(succ != null && succ->key < key) {
        pred = succ;
        succ = pred->next;
    }
    add(head, 3);
}

1

5

add(head, 3);

head

pred

succ
int add(node *head, int key)

int add(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    while(succ != null
        && succ->key < key) {
        pred = succ;
        succ = pred->next;
    }
    if(succ == null
        || succ->key == key)
        return 1;
    node *n = malloc(sizeof(node));
    n->key = key;
    n->next = pred->next;
    pred->next = n;
}
int remove(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
}

remove(head, 3);
int remove(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    while(succ != null
        && succ->key < key) {
        pred = succ;
        succ = pred->next;
    }
}
int remove(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    while(succ != null && succ->key < key) {
        pred = succ;
        succ = pred->next;
    }
    if(succ == null || succ->key == key) {
        pred->next = succ->next;
        return 1;
    }
    return 0;
}

remove(head, 3);
Add & Remove

```c
int add(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    while(succ != null && succ->key < key) {
        pred = succ;
        succ = pred->next;
    }
    if(succ == null || succ->key == key)
        return 1;
    node *n = malloc(sizeof(node));
    n->key = key;
    n->next = pred->next;
    pred->next = n;
    return 0;
}
```

```c
int remove(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    while(succ != null && succ->key < key) {
        pred = succ;
        succ = pred->next;
    }
    if(succ == null || succ->key != key)
        return 1;
    pred->next = succ->next;
    return 0;
}
```
Question

Use fine-grained lock to implement a concurrent linked list

- 5 minutes
- You can discuss with your neighbors
- Do not check following slides
Intuitive solution – Add

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

add(head, 3);

head

pred

succ
Intuitive solution – Add

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

`add(head, 3);`
Intuitive solution – Add

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

add(head, 3);

head

![Diagram showing the addition of a node with a lock to protect access.](image)
Intuitive solution – Add

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

add(head, 3);
Intuitive solution – Add

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

```c
add(head, 3);
```

![Diagram showing the addition of node 3 to the list](image)

---

*Note: The diagram illustrates the addition of node 3 to the list, linking it between nodes 1 and 5.*
Intuitive solution – Remove

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

remove(head, 3);
Intuitive solution – Remove

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

remove(head, 3);

head

1

pred

3

succ

5
**Intuitive solution – Remove**

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

```c
remove(head, 3);
```

![Diagram showing the removal of node 3]
Intuitive solution – Remove

Use lock to protect each node’s access

typedef struct node {
    node* next;
    int key;
    pthread_mutex_t mutex;
} node;

remove(head, 3);

Intuitive solution has correctness issues!!!
Issue – Concurrent Removes

Remove(2)

Remove(3)
Issue – Concurrent Removes

- Remove(2)
- Remove(3)
Issue – Concurrent Removes

Remove(2)

Remove(3)
Issue – Concurrent Removes

Remove(2)

Remove(3)
Issue – Concurrent Removes

Remove(2)

Remove(3)
Issue – Concurrent Removes

Remove(2)

Remove(3)
Issue – Concurrent Removes

Remove(2)

Remove(3)
Issue – Concurrent Removes

Remove(2)

Remove(3)
Uh, Oh

Remove(2)

Remove(3)
Bad news, 3 not removed
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
Concurrent linked list with fine-grained locking

Hand-over-hand

– First grab succ’s lock, then release pred’s lock
– When unlinking a node, need to hold both succ’s lock and pred’s lock
– if concurrent remove() calls try to remove adjacent nodes, then they acquire conflicting locks
Hand-over-Hand locking
Hand-over-Hand locking

Diagram showing a sequence with three steps: 1, 2, and 3, connected by arrows.
Hand-over-Hand locking
Hand-over-Hand locking
Hand-Over-Hand locking

Remove(2)
Hand-Over-Hand locking

Remove(2)
Hand-Over-Hand locking

Remove(2)
Hand-Over-Hand locking

Remove(2)

Found it!
Hand-Over-Hand locking

1

2

3

4

Remove(2)

Found it!
Hand-Over-Hand Locking

1 → 3 → 4

Remove(2)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Remove(2)  Remove(3)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Remove(2)

Remove(3)
Removing a Node

Must acquire Lock for 2

Remove(3)
Removing a Node

Waiting to acquire lock for 2

Remove(3)
Removing a Node

Wait!

Remove(3)
Removing a Node

Proceed to remove(2)
Removing a Node

Remove(2)
Removing a Node

Remove(2)
Removing a Node

Remove(2)
Removing a Node
Removing a Node

To remove node e
   Must lock e
   Must lock e’s predecessor

Therefore, if you lock a node
   It can’t be removed
   And neither can its successor
Removing a Node

To remove node e
  Must lock e
  Must lock e’s predecessor

Therefore, if you lock a node
  It can’t be removed
  And neither can its successor

```c
int remove(node *head, int key) {
    node *pred = head;
    node *succ = pred->next;
    pthread_mutex_lock(&pred->mutex);
    if(succ == null)
        return 1;
    pthread_mutex_lock(&succ->mutex);
    while(succ != null
        && succ->key <= key) {
        if(succ->key == key) {
            pred->next = succ->next;
            pthread_mutex_unlock(&pred->mutex);
            pthread_mutex_unlock(&succ->mutex);
        }
        pthread_mutex_unlock(&pred->mutex);
    }
    pthread_mutex_unlock(&pred->mutex);
    pred = succ;
    succ = pred->next;
    pthread_mutex_lock(&succ->mutex);
}

pthread_mutex_unlock(&pred->mutex);
if(succ != null)
    pthread_mutex_unlock(&pred->mutex);
return 1;
```
Question: do we need a hand-over-hand scheme?

– Must lock predecessor
– Must lock successor
Adding Nodes

Question: do we need a hand-over-hand scheme?
  – Must lock predecessor
  – Must lock successor

Answer: No
  – Only need to lock predecessor
  – Enough to guarantee neither of predecessor nor successor can be deleted
Drawbacks

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

Still not ideal
  – Long chain of acquire/release
  – Accessing disjoint parts of the list may still block one another

ReadWrite lock?
Drawbacks

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

Still not ideal
  – Long chain of acquire/release
  – Accessing disjoint parts of the list may still block one another

ReadWrite lock?
  – Upgrade read to write
  – Deadlock
  – Each lock operation takes a cost
Optimistic Synchronization

Step 1. Traverse the list and find the position without locking

Step 2. Lock pred and succ

Step 3. Check that pred and succ are still valid (reachable from the head)
Optimistic: Traverse without Locking

Add(3)

Aha!
Optimistic: Lock and Load

Add(3)
Optimistic: Validate nodes 2 and 4

Add(3)

Validate 2 and 4
Optimistic: Lock and Load

Add(3)
What could go wrong without validation?

Add(3)
What could go wrong without validation?

Add(3)
What could go wrong without validation?

Remove(2)
What could go wrong without validation?
What could go wrong without validation?

Add(3)
What could go wrong without validation?
What could go wrong without validation?

Uh-oh

Add(3)
Validate I –
Researchable from the head

Add(3)

Yes, 2 still reachable from head
What Else Could Go Wrong?

Add(3) → 1 → 2 → 5 → 6

Aha!
What Else Could Go Wrong?

Add(3)

Add(4)
What Else Could Go Wrong?

Add(3)

Add(4)
What Else Could Go Wrong?

Add(3)
What Else Could Go Wrong?

Add(3)
Validate II –
Node 2 still points to node 5

Add(3)

Yes, 2 still points to 5
Validate II –
Node 2 still points to node 5
Validate II –
Node 2 still points to node 5

The validation should be protected by the locks of pred and succ.
Basic Algorithm

Traverse the list to find the position without locking

Lock pred and succ

Validation?

Yes

Get/Add/Remove

Release locks and return

No

Release locks and retry
int add(node *head, int key) {
    while(true) {
        node *pred = head;
        node *succ = pred->next;
        if(succ == null)
            return 1;

        while(succ != null
                && succ->key <= key) {
            pred = succ;
            succ = pred->next;
        }

        if(succ == null || succ->key == key)
            return 1;

        pthread_mutex_lock(&pred->mutex);
        pthread_mutex_lock(&succ->mutex);
        if(validate(head, pred, succ) != 0) {
            node *n = malloc(sizeof(node));
            n->key = key;
            n->next = pred->next;
            pred->next = n;
            return 0;
        }
    }
}
int add(node *head, int key) {
    while(true) {
        node *pred = head;
        node *succ = pred->next;
        if(succ == null)
            return 1;

        while(succ != null
                && succ->key <= key) {
            pred = succ;
            succ = pred->next;
        }

        if(succ == null || succ->key == key)
            return 1;

        pthread_mutex_lock(&pred->mutex);
        pthread_mutex_lock(&succ->mutex);
        if(validate(head, pred, succ) != 0) {
            node *n = malloc(sizeof(node));
            n->key = key;
            n->next = pred->next;
            pred->next = n;
            return 0;
        }
    }
}

int validate(node *head, node* pred, node* succ) {
    node *n = head;
    while(n->key <= pred->key) {
        if(n == pred)
            return n->next == succ;
        n = n->next;
    }
    return 0;
}
So Far, So Good

Much less lock acquisition/release
  – Performance
  – Concurrency

Problems
  – Need to traverse list twice
Lazy List

Like optimistic, except
  – Scan once

Key insight
  – Removing nodes causes trouble
  – Do it “lazily”
    • Use a bit to indicate if it has been removed or not
Lazy List

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

Free the node’s memory
- When it is safe / no one holds its reference
Lazy Removal
Lazy Removal
Lazy Removal
Lazy Removal
Lazy Removal

Present in list
Lazy Removal

Logically deleted
Lazy Removal

Physically unlinked
Lazy Removal

Physically unlinked
Lazy List

All Methods
  – Scan through locked and marked nodes
  – Removing a node doesn’t slow down other method calls …

Must still lock pred and curr nodes.
Validation

Protected by locks of pred and succ
1. No need to rescan list!
2. Check that pred and succ are not marked
3. Check that pred still points to succ
Business as Usual
Business as Usual
Business as Usual

Remove(2)
Business as Usual

1 and 2 not marked
Business as Usual
Business as Usual

Logical delete
Business as Usual

physical
delete
Business as Usual
typedef struct node {
    node* next;
    int key;
    int marked;
    pthread_mutex_t mutex;
} node;

int remove(node *head, int key) {
    while(true) {
        node *pred = head;
        node *succ = pred->next;
        if(succ == null)
            return 1;
        while(succ != null && succ->key < key) {
            pred = succ;
            succ = pred->next;
        }
        if(succ == null || succ->key != key)
            return 1;
        pthread_mutex_lock(&pred->mutex);
        pthread_mutex_lock(&succ->mutex);
        if(validate(pred, succ) != 0
            && succ->key == key)
            { succ->marked = 1;
            pred->next = succ->next;
            pthread_mutex_unlock(&pred->mutex);
            pthread_mutex_unlock(&succ->mutex);
            return 0;
        }
    }
}

int validate(node* pred, node* succ) {
    return !pred->marked
    && !succ->marked
    && pred->next == succ;
}
When can we free the memory?

Can we free it immediately after being unlinked?
When can we free the memory?

Can we free it immediately after being unlinked?

No! There can be some threads hold still its reference.
When can we free the memory?

Can we free it immediately after being unlinked?

No! There can be some threads hold still its reference.

Counter ?
When can we free the memory?

Can we free it immediately after being unlinked?

No! There can be some threads hold still its reference.

Counter?

→ For each read operation, it needs to update the counter protected by lock.
When can we free the memory?

Insight
  – After being unlinked, it can not be reachable from the head

Basic idea
  – Do not free the node until all the other threads finish the traversing
When can we free the memory?

Insight
– After being unlinked, it can not be reachable from the head

Basic idea
– Do not free the node until all the other threads finish the traversing

```c
void free(node* n) {
    for each thread
        if it is in the function of add() / remove() / contain()
            wait for it to finish
    free(n)
}
```