Lecture 7
Synchronized Structures Part 1

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
1. Use a single pthread mutex to make this function thread-safe. Add global variables and content to the init() function as necessary.

2. Modify the original code from to make it thread-safe, but use two mutexes this time, one for sum_stat_a and one for sum_stat_b.
Invariant: The returned value from aggregateStats() must be equal to exactly the sum of statistic A and statistic B at exactly the moment in time when the new additions were aggregated into each sum.
Invariant: The returned value from aggregateStats() must be equal to exactly the sums of statistic A and statistic B, each taken at the moment in time when the new addition was aggregated into each sum. The aggregate sum may therefore represent a value covering two close (but different) time periods.

\[
\begin{align*}
  \text{sum_stat}_a & : \text{stat}_a_{\text{mutex}} \\
  \text{sum_stat}_b & : \text{stat}_b_{\text{mutex}}
\end{align*}
\]
Homework 2 Review
Aggregation (2 Locks)

```c
#include <pthread.h>

pthread_mutex_t stat_a_mutex;
pthread_mutex_t stat_b_mutex;

1 static int sum_stat_a = 0;
2 static int sum_stat_b = 0;
3 int aggregateStats(int stat_a, int stat_b) {
    int rval = 0;
    pthread_mutex_lock(&stat_a_mutex.lock);
7 sum_stat_a += stat_a;
8 rval += sum_stat_a;
9 pthread_mutex_unlock(&stat_a_mutex);
10 pthread_mutex_lock(&stat_b_mutex);
14 sum_stat_b += stat_b;
15 rval += sum_stat_b;
16 pthread_mutex_unlock(&stat_b_mutex);
17 return rval;
}
8 void init(void) {
    pthread_mutex_init(&stat_a_mutex, NULL);
    pthread_mutex_init(&stat_b_mutex, NULL);
}
```
We’re going to look at how to create more higher level synchronized data structures

- This week: Concurrent queues
- Next week: Building up to a lock-free hash table
Outline

• Lock-Based Concurrent Queue
• Unlocking: A Lock-Free Concurrent Queue
• Understanding The ABA Problem
• Solving ABA: Memory Management and Reuse
Lock-Based Concurrent Queue

• Consider a naïve concurrent queue
  • How many locks?
  • What problems with 1? (Correctness, performance, ...?)
  • What problems with 2? (Correctness, performance, ...?)

• New Concept: Scoped locks
  • Automatically unlocked when destroyed
  • Would have helped in Homework 2!
  • C++ scoped lock:
    `std::lock_guard<...> scoped_lock(my_mutex);`
Concurrent Queue: Protecting Shared State

elem* dequeue() {
    lock_guard(mutex);
    elem* node = nullptr;
    if (head != nullptr) {
        node = head;
        head = head->next;
    }
    if (head == nullptr) {
        tail = nullptr;
    }
    return node;
    // mutex unlocked
}

tenqueue(const& cnode) {
    lock_guard(mutex);
    elem* node =
        new elem(cnode)
    if (tail != nullptr) {
        tail->next = node;
    } else {
        head = node;
    }
    tail = node;
    // mutex unlocked
}
Concurrent Queue: Considering Correctness

• enqueue() and deque() must be protected under the same single lock
  • Why?

• Correctness: Linearizability
  • We dealt with this in Lab 1
  • Definition: There is some total serial order among all operations
    • Corollary: Each operation appears instantaneous
    • Corollary: No operation can see the intermediate state of another
Concurrent Queue: Considering Correctness

• Can’t split the lock as-is: no more linearizability
  • Another operation could see tail update before head update, or vice versa

• Linearization point: right after lock released
  • If there’s no obvious linearization point, algorithm may be wrong, hard to reason about, or both.

• We can still split the lock and be linearizable!
  • How?
Towards a Lock-Free Concurrent Queue

- Key concept: Sentinel
  - Dummy node, always first in the list (i.e., what head points to)
  - Make enqueue() and dequeue() touch distinct portions of the state
  - dequeue() checks if tail and head point to same element
  - Assume pointer reads and writes are atomic operations

- Still a blocking, locked structure
Dequeueing with Sentinel

- dequeue(): only touches head portion

Linearization point: head pointer update
Enqueuing with Sentinel

- enqueue(): only touches tail portion

Linearization point: tail pointer moved to new node
Simultaneous Enqueue and Dequeue on Empty Queue

- Depends on ordering of tail pointer write in dequeue() and read in enqueue()
- Write->write reordering would break this! (Why?)
A Lock-Free Queue
Lock-Free Queue?

• It’s possible!
• Current locks: 2
  • Single concurrent enqueue()
  • Single concurrent dequeue()
  • No protection between enqueue() and dequeue()
• Necessary atomic primitive: CAS
  • Compare and Swap
CAS(address, expected, desired)
Atomically:
1. Check if (*address == expected)
2. If so, set *address = desired, return true
3. Otherwise, return false

• C++11:
  std::atomic<T> type -> variable + locking!
  template<class T>
  bool atomic_compare_exchange_strong(volatile std::atomic<T>* obj,
  T* expected, T desired);
enqueue(elem* node) {
  ...
  tail->next = node;
  CAS1{
    if (tail->next == node)
      tail = tail->next;
  }
}

Why can’t we use this? 😞
enqueue(elem* node) {
    elem* cur_tail = tail;
    CAS1{
        if (tail->next == cur_tail->next)
            tail->next = node;
    }
}
Two-Step Lock-Free Enqueue

```c
enqueue(elem* node) {
    elem* cur_tail = tail;
    CAS1{
        if (tail->next == cur_tail->next)
            tail->next = node;
    }
    CAS2{
        if (tail == cur_tail)
            tail = node;
    }
}
```
enqueue(const value& val) {
    elem* node = new elem();
    node->val = val;
    node->next = nullptr;
    while(true) {
        elem* cur_tail = tail;
        if (CAS(tail->next, cur_tail->next, node)) {
            if (CAS(tail, cur_tail, node)) {
                break;
            }
        }
    }
}
Two-Step Lock-Free Enqueue Gotcha

• What happens during the two CAS operations?
• Tail might not point to last node
  • Why? On CAS2, node may not be real tail; another thread may have set tail->next to own node in CAS1.
• New invariant: tail may only point to last node or second-to-last node
Two-Step Lock-Free Enqueue Gotcha

Thread 1 wants to enqueue node B
Thread 2 wants to enqueue node C

1. Thread 1 caches tail in cur_tail, performs CAS 1.

2. Thread 2 caches tail in cur_tail, performs CAS 1
Two-Step Lock-Free Enqueue Gotcha

Thread 1 wants to enqueue node B
Thread 2 wants to enqueue node C

3. Thread 1 finishes enqueue with CAS 2. Tail hasn’t moved, so its CAS 2 succeeds
Two-Step Lock-Free Enqueue Gotcha

Thread 1 wants to enqueue node B
Thread 2 wants to enqueue node C

4. Thread 2 tries to finish enqueue with CAS 2. It fails, because tail has changed. It starts over and performs CAS 1 and CAS 2.

5. If enqueue checked that tail->next was nullptr before updating tail->next, this problem would be avoided, but the tail pointer could then become stale.
Two-Step Lock-Free Enqueue Fix

enqueue(const value& val) {
    elem* node = new elem();
    node->val = val;
    node->next = nullptr;
    while(true) {
        elem* cur_tail = tail;
        elem* next = cur_tail->next;
        if (cur_tail == tail) {
            if (next == nullptr) {
                if (CAS(tail->next, next, node)) {
                    break;
                }
            } else {
                CAS(tail, cur_tail, next); // Correct stale tail
            }
        } else {
            CAS(tail, cur_tail, next); // Correct stale tail
        }
    }
    // If the following fails, someone else will handle it.
    CAS(tail, cur_tail, node);
}
Two-Step Lock-Free Dequeue

• What if dequeue() misses a node because of invariant (i.e., tail points to second-to-last node)?
  • We’ll get this on the next slide.

• General lock-free approach:

```c
value dequeue() {
    if (head == tail) {
        return null_val;
    }
    while(true) {
        value val = head->next;
        elem* cur_head = head;
        if (CAS(head, node, node->next)) {
            free(cur_head);  // Old sentinel
            return val;
        }
    }
}
```
Two-Step Lock-Free Dequeue

```c
val dequeue() {
    value val = null_val;
    elem* free_node = nullptr;
    while(true) {
        elem* cur_head = head;
        elem* cur_tail = tail;
        elem* next = cur_head->next; // Catch half-done enqueue()
        if (cur_head == cur_tail) {
            if (next == nullptr) {
                return nullptr; // Empty queue
            }
            CAS(tail, cur_tail, next); // Fix: tail->next != nullptr
        } else {
            free_node = head;
            val = next->val;
            if (CAS(head, cur_head, next)) {
                break;
            }
        }
    }
    free(free_node);
    return val;
}
```

Approach: fix the tail, then dequeue
Lock-Free Queue
Linearizability

• Still linearizable
• dequeue() “happens” with atomic head = head->next
• enqueue() “happens” with atomic tail->next = node
• No transient state is misleading, even the misplaced tail!
  • dequeue() and enqueue() both handle it
The ABA Problem
The ABA Problem

1. dequeue() wants to dequeue a node

```
  elem* node = next;
  if (CAS(head, cur_head, next) {
    return node;
  }
```
The ABA Problem

2. Another dequeue() interrupts and dequeues A

![Diagram of a linked list with nodes C, B, and A, and arrows indicating the head and tail connections. Freed Nodes indicated on the right.]
The ABA Problem

3. Another dequeue() interrupts and dequeues B
4. An enqueue() re-uses A’s memory for a new node
5. [Optional: other enqueue()s/dequeue()s occur]
The ABA Problem

6. Finally, the original dequeue() completes

```c
elem* node = next;
if (CAS(head, cur_head, next) {
    return node;
}
```

Still A, but next was B, and B was freed (and could contain anything!)
The ABA Problem

• Pointers are spatially unique, **not** also temporally unique
  • Re-using memory is necessary, but can cause unexpected problems

• If we re-use memory, we want the CAS to fail, even if the pointer points to the same memory
  1. Additional pointer information
  2. Free-list tracking
The ABA Solution

• Version our pointers
  • 128-bit pointer
    • 64 bits of target address
    • 64-bit counter
  • Requires 128-bit CAS (64-bit x86 supports this)

• Track “freed” pointers in free list with associated counter
  • Re-use freed pointers but increment counter
Lab 2 Check-In

- Lab 2: Performance testing
  - Must run at most one httpperf client per machine
  - Try running on a different machine than your server
  - Experiment with parameters
    - We will provide a script to generate workloads
Conclusions

• Turning a single lock into multiple (smaller-scope) locks can be done
  • Careful invariant consideration
  • Linearizability checking

• Removing locks entirely is possible
  • Atomicity still needed: CAS, TAS
  • Substantial engineering effort

• Next time:
  • Lock-free ordered lists
  • Lock-free hash tables