CS 241 Honors
Concurrent Data Structures

Bhuvan Venkatesh

University of Illinois Urbana–Champaign

March 27, 2018

What to go over

- Terminology
- What is lock free
- Example of Lock Free
- Transactions and Linearizability
- The ABA problem
- Drawbacks
- Use Cases

Word of warning

- Measure, Measure, Measure!

Word of warning

- Measure, Measure, Measure!
- You need to make sure that the bottleneck of your problem is in fact your data structure. Meaning that you spend most of your time in mutex lock.

Word of warning

- Measure, Measure, Measure!
- You need to make sure that the bottleneck of your problem is in fact your data structure. Meaning that you spend most of your time in mutex lock.
- Then, you make sure that there is no other algorithm that can just reduce the number of mutex locks outright.

Word of warning

- Measure, Measure, Measure!
- You need to make sure that the bottleneck of your problem is in fact your data structure. Meaning that you spend most of your time in mutex lock.
- Then, you make sure that there is no other algorithm that can just reduce the number of mutex locks outright.
- Then, you have to choose the right lock free data structure. Most of them work best under circumstances with high contention and where the structure is full of elements.

Word of warning

- Measure, Measure, Measure!
- You need to make sure that the bottleneck of your problem is in fact your data structure. Meaning that you spend most of your time in mutex lock.
- Then, you make sure that there is no other algorithm that can just reduce the number of mutex locks outright.
- Then, you have to choose the right lock free data structure. Most of them work best under circumstances with high contention and where the structure is full of elements.
- Then, you have to measure. If you don't measure, then you don't know if you improved.

Terminology

- Atomic instructions - Instructions that happen in one step to the CPU or not at all. Some examples are atomic_add, atomic_compare_and_swap.

Terminology

- Atomic instructions - Instructions that happen in one step to the CPU or not at all. Some examples are atomic_add, atomic_compare_and_swap.
- Transaction - Like atomic operations, either the entire transaction goes through or doesn't. This could be a series of operations (like push or pop for a stack)

Terminology

- Atomic instructions - Instructions that happen in one step to the CPU or not at all. Some examples are atomic_add, atomic_compare_and_swap.
- Transaction - Like atomic operations, either the entire transaction goes through or doesn't. This could be a series of operations (like push or pop for a stack)
- Linearizability - Once a write completes, all reads after point to the new data.

Terminology

- Atomic instructions - Instructions that happen in one step to the CPU or not at all. Some examples are atomic_add, atomic_compare_and_swap.
- Transaction - Like atomic operations, either the entire transaction goes through or doesn't. This could be a series of operations (like push or pop for a stack)
- Linearizability - Once a write completes, all reads after point to the new data.
- Atomic Compare and Swap

Atomic Compare and Swap?

Atomic Compare and Swap?

```c
int atomic_cas(int *addr, int *expected,
     int value){
        if(*addr == *expected){
            *addr = value;
            return 1; //swap success
        }else{
            *expected = *addr;
            return 0; //swap failed
        }
    }
```

Atomic Compare and Swap?

```c
int atomic_cas(int *addr, int *expected,
     int value){
        if(*addr == *expected){
            *addr = value;
            return 1; //swap success
        }else{
            *expected = *addr;
            return 0; //swap failed
        }
    }
```

But this all happens atomically!

Types of Data Structures

- Blocking Data Structures
- Lock-Free Data Structures
- Bounded-Wait Data Structures
- Wait-Free Data Structures

Blocking Structures

Blocking Structures

Advantages:

- Simpler to program
- Well defined critical sections
- Built-in Linearizability (To Be Explained)

Blocking Structures

Advantages:

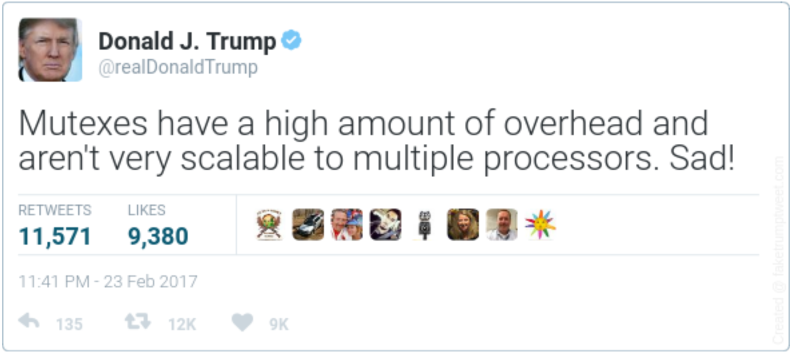
- Simpler to program
- Well defined critical sections
- Built-in Linearizability (To Be Explained)

Disadvantages

- Slower under high contention; Mutexes are not scalable across cores
- Lower priority processes often get locks
- Deadlock! Convoy Effect!
- Preemption/Signal Handler Safety

Blocking Structures

Blocking Structures



Lock-Free Structures

Lock-Free Structures

Advantages:

- Better under high contention
- Atomics are faster when the number of CPUs is small.

Lock-Free Structures

Advantages:

- Better under high contention
- Atomics are faster when the number of CPUs is small.

Disadvantages

- Critical Section a little harder to define
- Harder to debug
- Can get really complicated

Bounded Wait + Wait Free Structures

Bounded Wait + Wait Free Structures

Advantages:

- Faster than lock free. The CPU is always doing something.
- Guaranteed work after some point
- More stable, less probability

Bounded Wait + Wait Free Structures

Advantages:

- Faster than lock free. The CPU is always doing something.
- Guaranteed work after some point
- More stable, less probability

Disadvantages

- Not always possible
- Hard to guarantee
- Can get *really* complicated.

Building a lock free Data Structure

Alright, let's make a queue. What do we have to think about? Well there are two types of threads. Those pushing things on to the queue, and those popping off the queue.

Starting out - No Code but Ownership

A fundamental way to solve race conditions is to ask yourself the question "who owns this piece of memory". If you make sure that one thread has access to the piece of memory at once, you will never get deadlock.

Starting out - No Code but Ownership

A fundamental way to solve race conditions is to ask yourself the question "who owns this piece of memory". If you make sure that one thread has access to the piece of memory at once, you will never get deadlock. For the sake of not sitting here for days about lock free structures, we will assume that we have a fast, lock free malloc.

Ownership?

1. We want to introduce the idea that a thread owns a piece of memory. This is to avoid race conditions.
2. There is going to be a shared part of memory and a part only visible to the thread. We are going to do all of our initialization in our memory part and then with one atomic instructions that are carefully placed
3. After that the data structure will be initialized

Lock Free initialization

```
typedef struct node;
typedef struct queue;
new_queue()
destory_queue()
```

Lock Free Enqueue

```c
void enqueue(queue *fifo, void *val){
    node *pkg = malloc(sizeof(*pkg)), *ptr;
    *pkg = {val, NULL};
    int succeeded = 0;
    while(!succeeded){
        node *none = NULL;
        ptr = fifo->tail;
        succeeded = cas(&ptr->next, none, pkg)
    }
    // This is actually a critical section
    cas(&fifo->tail, &ptr, package);
}
```

Lock Free Dequeue

```
node* dequeue(queue *fifo){
    if (!fifo->head) { return NULL; }
    node *start = &fifo->head;
    while(!atomic_cas(&fifo->head, start,
             &fifo->head->next)){
        start = atomic_load(&fifo->head);
        if(start == NULL){
            return NULL;
        }
        // May do sleeping here
    }
    //You now have exclusive access
    return start;
}
```

Transactions

1. The idea of transactions is that a group of operations go in together and become apparent to the data structure all at once.

Transactions

1. The idea of transactions is that a group of operations go in together and become apparent to the data structure all at once.
2. This is important because this provides high concurrency. Each thread has its own place to do its work, and undergoes contention all at once.

Transactions

1. The idea of transactions is that a group of operations go in together and become apparent to the data structure all at once.
2. This is important because this provides high concurrency. Each thread has its own place to do its work, and undergoes contention all at once.
3. You can see this in the previous example.

Lock Free Enqueue

```
node *package = malloc(sizeof(*package));
package->data = val;
package->next = NULL;
node *ptr;
...
```

Lock Free Enqueue

```
ctx *tex = begin_transaction(queue);
queue_pop(tex);
queue_push(tex, 1);
queue_push(tex, 2);
queue_push(tex, 3);
end_transaction(tex);
// Results pushed (only 2 atomics computed)
```

Alterations

```
pop (tex) {
    // Push the pop on the context
}
push (tex) {
    /* Allocate memory and push the
         value on the stack. if there
         is another pop instruction,
         squish the two nodes */
}
```

Alterations

```
end_transaction (tex) {
    void *none = NULL;
    cas(&tex->queue->head,
        &tex->queue->head, none);
    // My queue
    for (transaction in tex) {
        if (transaction->type == push)
            queue_push(tex->queue,
                transaction->data)
        //...
    }
}
```

Transactions

1. Of course, you have to resolve the idea of what a pop means in an empty queue in a transaction. More often than not in high performance parallel programming we just throw away bad pops in the case of a queue.

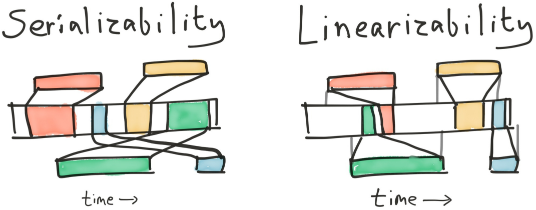
Transactions

1. Of course, you have to resolve the idea of what a pop means in an empty queue in a transaction. More often than not in high performance parallel programming we just throw away bad pops in the case of a queue.
2. You also may have to deal with a transaction failing because the queue may have run out of space.

Transactions

1. Of course, you have to resolve the idea of what a pop means in an empty queue in a transaction. More often than not in high performance parallel programming we just throw away bad pops in the case of a queue.
2. You also may have to deal with a transaction failing because the queue may have run out of space.
3. This is mainly a tool for the people using the lock free data structures keep track of their operations.

Linearizability & Serializability



Linearizability

1. Once a write completes, all reads after point to the new data.

Linearizability

1. Once a write completes, all reads after point to the new data.
2. It is kind of obvious to see the benefit here, we want a data structure to be consistent and once the writes happen, all reads after make sense (they don't get stale data).

Linearizability

1. Once a write completes, all reads after point to the new data.
2. It is kind of obvious to see the benefit here, we want a data structure to be consistent and once the writes happen, all reads after make sense (they don't get stale data).
3. Also good to determine a semi-ordering (Think of a reader writer consistency model).

Linearizability

1. Once a write completes, all reads after point to the new data.
2. It is kind of obvious to see the benefit here, we want a data structure to be consistent and once the writes happen, all reads after make sense (they don't get stale data).
3. Also good to determine a semi-ordering (Think of a reader writer consistency model).
4. But sometimes we need stronger consistency models.

Serializability

1. A series of transactions on objects has some consistent ordering.

Serializability

1. A series of transactions on objects has some consistent ordering.
2. This is a strong model of consistency (not the strongest) but usually the highest one that lock free data structures succumb to. There is a performance hit but we want our data structure to make sense.

Serializability

1. A series of transactions on objects has some consistent ordering.
2. This is a strong model of consistency (not the strongest) but usually the highest one that lock free data structures succumb to. There is a performance hit but we want our data structure to make sense.
3. Think of a stock market application that needs to tell which monetary transaction happened first. We need the application to be fast but we need to know who bought and sold first

Strong Serializability

1. When a transaction is both Serializable and Linearizable.

Strong Serializability

1. When a transaction is both Serializable and Linearizable.
2. Meaning that the order of transactions make sense if computed by one thread.

Strong Serializability

1. When a transaction is both Serializable and Linearizable.
2. Meaning that the order of transactions make sense if computed by one thread.
3. Hardest to obtain.

ABA Problem

1. The ABA problem, no it's not a problem about Dancing Queen.

ABA Problem

1. The ABA problem, no it's not a problem about Dancing Queen.
2. It is the idea that a thread A can do something half way, then thread B does a lot of things, and A tries to complete its transaction but incorrectly succeeds.

ABA Problem

1. The ABA problem, no it's not a problem about Dancing Queen.
2. It is the idea that a thread A can do something half way, then thread B does a lot of things, and A tries to complete its transaction but incorrectly succeeds.
3. This is a problem because over the long term the entire data structure could either break or leak memory.

ABA Problem

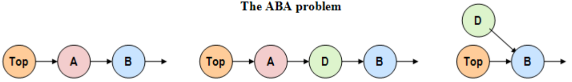
1. The ABA problem, no it's not a problem about Dancing Queen.
2. It is the idea that a thread A can do something half way, then thread B does a lot of things, and A tries to complete its transaction but incorrectly succeeds.
3. This is a problem because over the long term the entire data structure could either break or leak memory.
4. Some cases there are no ways of preventing this problem, especially in a language like C without automagic garbage collection.

ABA Problem

```
void enqueue(queue *fifo, void *val) {
    node *pkg = malloc(sizeof(*pkg)), *ptr;
    *pkg = {val, NULL};
    int succeeded = 0;
    while(!succeeded) {
        node *none = NULL;
        ptr = fifo->tail;
        succeeded = cas(&ptr->next, none, pkg)
        if(!succeeded){
            cas(&fifo->next, &ptr, ptr->next);
        }
    }
    cas(&fifo->tail, &ptr, package);
    void *data = pkg->val;
    free(pkg);
}
```

ABA Problem

The ABA problem



Thread A calls method POP that reads Top (pointer to A) and *Top (pointer to B). Thread A gets preemted before doing CAS(Top,*Top).

Thread B also does POP by doing CAS(Top,*Top), then deletes returned pointer to A. After that Thread B calls PUSH with D by doing CAS(Top, D) and PUSH with new A by doing CAS(Top,A).

Thread A resumes and completes it's CAS(Top,B) leaving D hanging.

Did you notice?

1. Our data structure does not need to worry about that!

Did you notice?

1. Our data structure does not need to worry about that!
2. Our dequeue instead of returning the item returns the entire node.

Did you notice?

1. Our data structure does not need to worry about that!
2. Our dequeue instead of returning the item returns the entire node.
3. That means the ABA problem has a near zero chance of actually occuring if the user doesn't free the node until after they are done using it.

Drawbacks

1. These data structures can get complicated fast

Drawbacks

1. These data structures can get complicated fast
2. No one knows if they always work

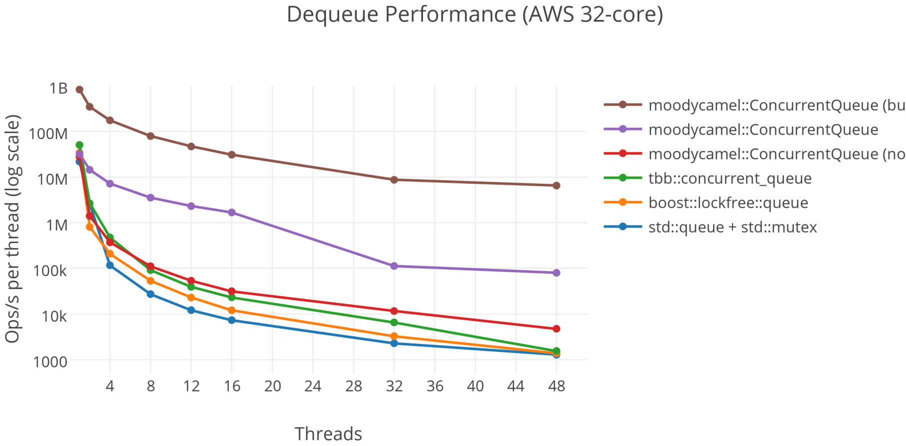
Drawbacks

1. These data structures can get complicated fast
2. No one knows if they always work
3. Sometimes they can't always work

Drawbacks

1. These data structures can get complicated fast
2. No one knows if they always work
3. Sometimes they can't always work
4. Sometimes they are slower

Drawbacks



Dequeue Performance (AWS 32-core)
Ops/s per thread (log scale)
1B
100M
10M
1M
100k
10k
1000
4
8
12
16
20
24
28
32
36
40
44
48
Threads
moodycamel::ConcurrentQueue (bu
moodycamel::ConcurrentQueue
moodycamel::ConcurrentQueue (no
tbb::concurrent_queue
boost::lockfree::queue
std::queue + std::mutex

Memory Orders

1. One last thing that we need to cover is memory orders

Memory Orders

1. One last thing that we need to cover is memory orders
2. Memory orders or barriers are a way for you to tell the compiler not to out of order execute your instructions.

Memory Orders

1. One last thing that we need to cover is memory orders
2. Memory orders or barriers are a way for you to tell the compiler not to out of order execute your instructions.
3. There are different levels of orders – most of the time you want the most restrictive in order to ensure your behavior works.

Memory Orders

1. One last thing that we need to cover is memory orders
2. Memory orders or barriers are a way for you to tell the compiler not to out of order execute your instructions.
3. There are different levels of orders – most of the time you want the most restrictive in order to ensure your behavior works.
4. Only then move down to other orders

Sequential Consistency

```
-Thread 1-                    -Thread 2-
y = 1                     if (x.load() == 2)
store(x, 2);                  y != 1 && *NULL = 1;
```

Will never segfault

Relaxed Consistency

```
-Thread 1-          -Thread 2-
store(x, 1)         first = load(x)
store(x, 2);        second = load(x)
                    first <= second
```

A new value will never become old

Use Cases

1. RabbitMQ/Apache Kafka is a distributed message queue that uses a queue similar to the one we described to distribute messages to a group of nodes.
2. Apache Spark and Hadoop use this for consensus, finger tables, and communicating and joining results together.
3. Every distributed (and a lot of non-distributed) databases use lock-free data structures to service SQL queries or read/write form disks
4. HPC uses them to manage concurrency (Possible MPI)

Bonus! Why Spurious Wakeups Happen

1. Ever wonder *why* spurious wakeups happen?

Bonus! Why Spurious Wakeups Happen

1. Ever wonder *why* spurious wakeups happen?
2. Well y'all are masters at lock free data structures now so you can guess.

Bonus! Why Spurious Wakeups Happen

1. Ever wonder *why* spurious wakeups happen?
2. Well y'all are masters at lock free data structures now so you can guess.
3. Let's take the Windows NT way of implementing a condition variable.

Bonus! Why Spurious Wakeups Happen

1. Ever wonder *why* spurious wakeups happen?
2. Well y'all are masters at lock free data structures now so you can guess.
3. Let's take the Windows NT way of implementing a condition variable.
4. The real problem with CVs are broadcast.

Windows NT

```
struct CV {
    linked_list waiters;
}

void wait(cv, mtx) { // mtx must be locked
    enqueue(cv.waiters, self());
    m.Release();
    self().sema.wait();
    m.Acquire();
}
```

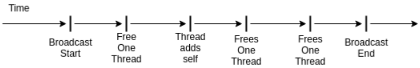
Not Interesting

```
void signal(cv) {
    if (waiters != null) {
        waiters.sema.post();
        cas(waiters, waiters, waiters.next);
    }
}
```

Interesting

```
void broadcast(cv) {
    while (waiters != null) {
        waiters.sema.post();
        cas(waiters, waiters, waiters.next);
    }
}
```

Interesting



Questions?

Thanks for sticking along!