CS 241 Honors
Concurrent Data Structures

Bhuvan Venkatesh

University of Illinois Urbana–Champaign

October 25, 2017

What to go over

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

Terminology

- 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

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

Atomic Compare and Swap?

Atomic Compare and Swap?

```
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?

```
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

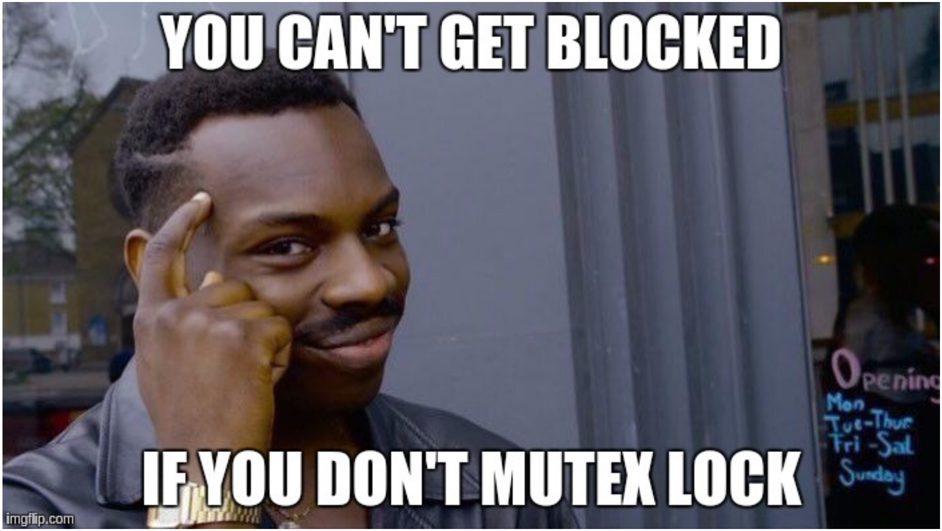
Advantages:

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

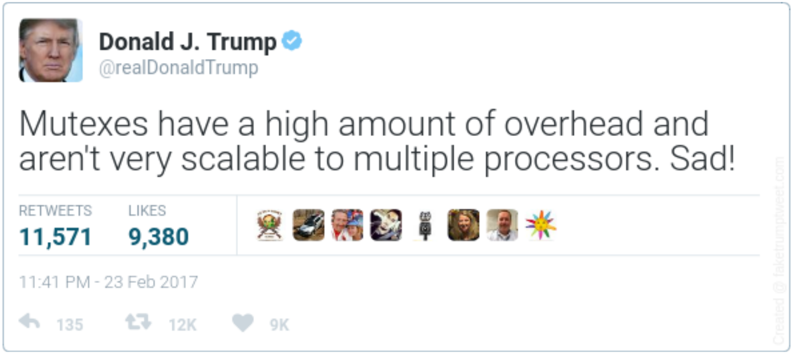
Disadvantages

- 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



Donald J. Trump
@realDonaldTrump

Mutexes have a high amount of overhead and aren't very scalable to multiple processors. Sad!

RETWEETS 11,571 LIKES 9,380

11:41 PM - 23 Feb 2017

135 12K 9K

Lock-Free Structures

Lock-Free Structures

Advantages:

- Better under high contention
- Atomics are a little faster
- Bit more stable, no locks!

Lock-Free Structures

Advantages:

- Better under high contention
- Atomics are a little faster
- Bit more stable, no locks!

Disadvantages

- Critical Section a little harder
- 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 thread, and taking off the thread.

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.
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?

We want to introduce the idea that a thread owns a piece of memory. This is to avoid race conditions. So, 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 or two atomic instructions that are carefully placed, we are going to complete the swap and finish the function.

Lock Free initialization

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

Lock Free Enqueue

```
void enqueue(queue *fifo, void *val){
    node *pkg = malloc(sizeof(*pkg));
    pkg->data = val;
    pkg->next = NULL;
    node *ptr;
    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);
}
```

Lock Free Dequeue

```
    node* dequeue(queue *fifo){
        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

```
begin_transaction(queue);
queue_pop(queue);
queue_push(queue, 1);
queue_push(queue, 2);
queue_push(queue, 3);
end_transaction(queue); // Results pushed
```

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.

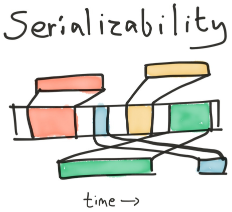
Transactions

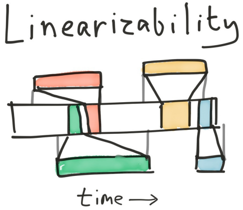
1. Of course, you have to resolve the idea of what a pop means in an empty queue in a transaction. More often than not in high performance parallel programming we just throw away bad pops in the case of a queue.
2. You also may have to deal with a transaction failing because the queue may have run out 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. Once a write completes, all reads after point to the new data, like linearizability.

Serializability

1. Once a write completes, all reads after point to the new data, like linearizability.
2. A series of transactions looks like it could have been executed by one thread even though it was executed by multiple threads.

Serializability

1. Once a write completes, all reads after point to the new data, like linearizability.
2. A series of transactions looks like it could have been executed by one thread even though it was executed by multiple threads.
3. 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. Once a write completes, all reads after point to the new data, like linearizability.
2. A series of transactions looks like it could have been executed by one thread even though it was executed by multiple threads.
3. 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.
4. Think of a stock market application that needs to tell which monetary transaction happend first. We need the application to be fast but we need to know who bought and sold first

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

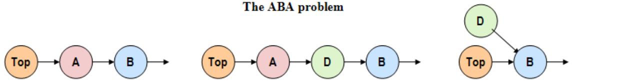
1. The ABA problem, no it's not a problem about Dancing Queen.
2. It is the idea that a thread A can do something half way, then thread B does a lot of things, and A tries to complete its transaction but incorrectly succeeds.
3. This is a problem because over the long term the entire data structure could 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));
    pkg->data = val;
    pkg->next = NULL;
    node *ptr;
    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);
}
```

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 with high probability 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

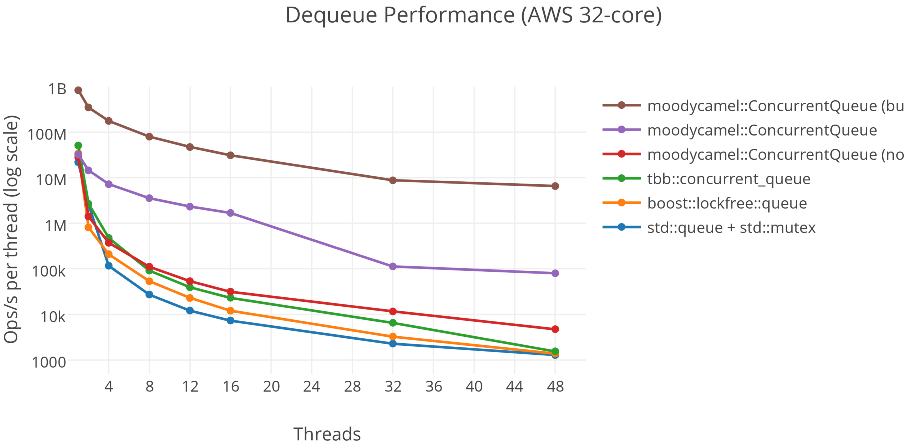
Drawbacks

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

Benefits

1. Atomics are fast in of themselves under moderate workloads
2. Theoriticians like this; We can declare the random variable X_i as the number of times that the atomic instruction occurs for a particular thread i.

Benefits

1. Atomics are fast in of themselves under moderate workloads
2. Theoriticians like this; We can declare the random variable X_i as the number of times that the atomic instruction occurs for a particular thread i.
3. We know that the variables are **not** pairwise independent (even though the variance may be known) but that means that we can apply Markov's Inequality.

Benefits

1. Atomics are fast in of themselves under moderate workloads
2. Theoriticians like this; We can declare the random variable X_i as the number of times that the atomic instruction occurs for a particular thread i.
3. We know that the variables are **not** pairwise independent (even though the variance may be known) but that means that we can apply Markov's Inequality.
4. ($Pr[X \geq a] \leq \frac{E[X]}{a}$) Meaning that if we work through the algebra that it would mean that with high probability that the number of atomic instructions is bounded where $a = n * E[X]$.

Benefits

1. Atomics are fast in of themselves under moderate workloads
2. Theoriticians like this; We can declare the random variable X_i as the number of times that the atomic instruction occurs for a particular thread i.
3. We know that the variables are **not** pairwise independent (even though the variance may be known) but that means that we can apply Markov's Inequality.
4. $(Pr[X \geq a] \leq \frac{E[X]}{a})$ Meaning that if we work through the algebra that it would mean that with high probability that the number of atomic instructions is bounded where $a = n * E[X]$.
5. This means (given we execute on average 10 atomics on average) that we expect 98% of the time that we execute 500 or less instructions. This may seem like a lot, but computers are exeucting billions every second, they're good.

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();
}
```

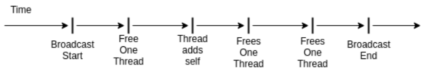
Not Interesting

```
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!