Avoiding Locks

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Locks are slow

- Using locks, for mutual exclusion, comes with significant overhead
  - Especially when using blocking locks!
  - But even when using spin locks on short critical sections
- Essentially, the more your program calls “lock()”, the slower it is
- But locks are fundamental for ensuring mutual exclusion, so don’t we just need them?
- Many smart people have tried to use lock() less, or not at all!

Cater to the Common Case

- Locks are used to ensure mutual exclusion, and critical sections are often short
- Unless you have a lot of threads that contend with the critical section, a thread will typically get the lock right away
- The common case: everything goes well
- The rare case: there is contention for the critical section and threads can't enter the critical section
- Then it seems very wasteful to pay the overhead of calling lock() / unlock() every time when in fact things are going to be fine with high probability
  - Especially if locks are blocking instead of spin!
- Let's look at a trivial example...

A Concurrent Array

- Your abstract data type is an array of size N
- You provide one method: increment(i)
  - Increments the i-th element
- Here is one implementation:

```
public class ConcurrentArray {
    public int values[N];

    public synchronized void increment(int i) {
        values[i]++;
    }
}
```

- Why is this not good?

A Concurrent Array

- Your abstract data type is an array of size N
- You provide one method: increment(i)
  - Increments the i-th element
- Here is one implementation:

```
public class ConcurrentArray {
    public int values[N];

    public synchronized void increment(int i) {
        values[i]++;
    }
}
```

- Why is this not good? Not enough concurrency!

Another Implementation

```
public class ConcurrentArray {
    public int values[N];
    private Lock locks[N];

    public  void increment(int i) {
        locks[i].lock();
        values[i]++;
        locks[i].unlock();
    }
}
```

- Why is this not good?

Another Implementation

```
public class ConcurrentArray {
    public int values[N];
    private Lock locks[N];

    public  void increment(int i) {
        locks[i].lock();
        values[i]++;
        locks[i].unlock();
    }
}
```

- Why is this not good? Too much memory usage!

Forget Locks!

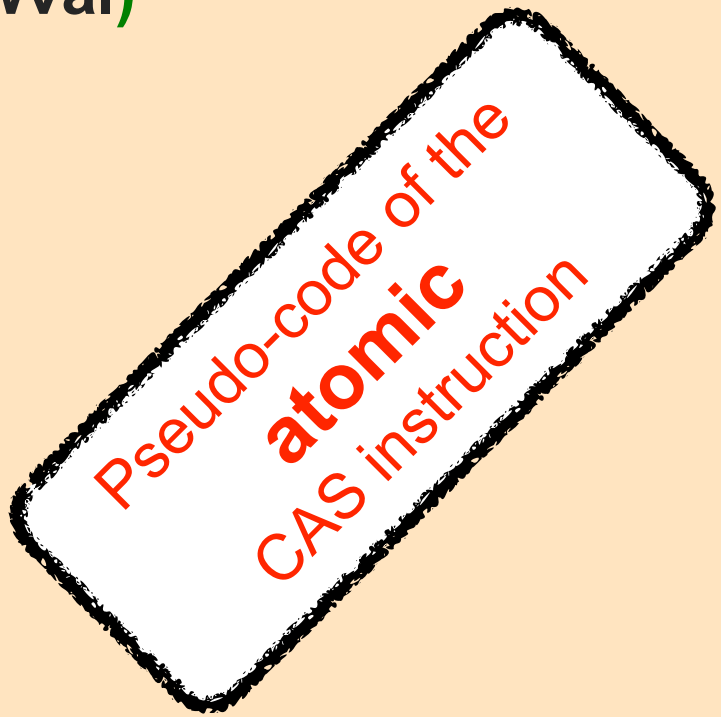
- Each implementation before corresponds to an extreme:
  - One lock, no concurrency, low memory footprint
  - N locks, full concurrency, high memory footprint
- We could pick any option in between
  - e.g., locks[i/5].lock();
- We've talked about this conundrum before
- But regardless, if you have fewer threads than N, and if threads access the array all over, the probability that you needed locks in the first place is very, very low
- Let's try to not use locks at all...

The Basic Idea

- Compute array[i] + 1
- Atomically:
  - If in the meantime nobody has changed the value, then write the incremented value to memory
  - Otherwise, re-attempt: “Oops, somebody else updated the array in the meantime, let’s forget everything I did and I am doing it again from scratch”
- The idea is very similar to that of transactional memory (see next set of lecture notes)
  - But we implement it by hand without requiring our hardware to do anything new
- The only thing the hardware must provide: the same atomic instruction that’s used to implement locks
  - And all CPUs have that

Compare-And-Swap (CAS)

```
boolean compare_and_swap(type *var,
                         type oldval,
                         type newval)
{
    if (*var == oldval) {
        *var = newval;
        return true;
    } else {
        return false;
    }
}
```



CAS in Languages

- Java: provided, e.g., for Integers, as part of the AtomicInteger class

```
AtomicInteger foo = new AtomicInteger(42);
boolean success =
        foo.CompareAndSet(expected, new);
```

- C/C++: Provided as part of many libraries (e.g., Boost)
  - For instance, the `__sync_val_compare_and_swap()` built-in function in gcc

Lock-Free Concurrent Array

```
public class ConcurrentArray {
    public int values[N];

    public  void increment(int i) {
        int old_value, new_value;
        do {
            old_value = values[i];
            new_value = old_value + 1;
        } while (!CAS(&(values[i]), old_value, new_value);
    }
```

- Record the value I see now
- Record the value I want to write
- Atomically: if the old value is still there, then write my new value and be done, **otherwise re-attempt**

Lock-Free Concurrent Array

```
public class ConcurrentArray {
    public int values[N];

    public  void increment(int i) {
        int old_value, new_value;
        do {
            old_value = values[i];
            new_value = old_value + 1;
        } while (!CAS(&(values[i]), old_value, new_value);
    }
```

- Chances are CAS is going to succeed most of the time, unless there are tons of competing threads
- But if we have tons of competing threads, perhaps we should design our application differently anyway!

Lockfree Data Structures

- We can use the same approach for all kinds of data structures
- We then call them “Lockfree data structures”
  - Used to be an active research area, and many research papers have given use efficiency lock free data structures
- Let’s look at one of the simplest: a lockfree, thread-safe Stack….

A Lock-free Stack in C

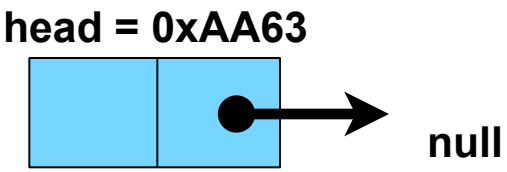
```
void push(int t) {
   Node* node = new_node(t);
   do {
     node->next = head;
  } while (!CAS(&head, node->next, node));
}
```

A Lock-free Stack in C

```
bool pop(int *t) {
    Node* current = head;
    while (current) {
        Node *next = current->next;
        if(CAS(&head, current, next)) {
            *t = current->data;
             free(current);
            return true;
         }
        current = head;
      }
      return false;
}
```

A Lock-free Stack in C

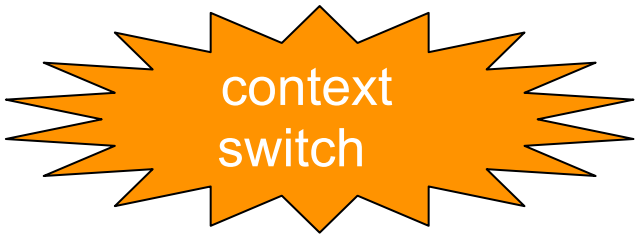
```
void push(int t) {
  Node* node = new_node(t);
  do {
       node->next = head;
  } while (!CAS(&head, node->next, node));
}
```



Thread #1: push()

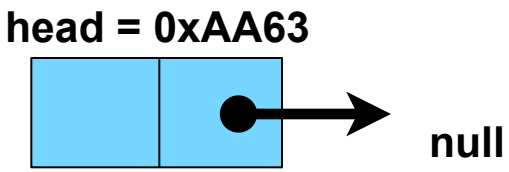
node = 0xAAAA

node->next = 0xAA63



A Lock-free Stack in C

```
void push(int t) {
   Node* node = new_node(t);
   do {
        node->next = head;
   } while (!CAS(&head, node->next, node));
}
```



Thread #1: push()

node = 0xAAAA

node->next = 0xAA63

Thread #2: push()

node = 0xFFFF

node->next = 0xAA63

// CAS

head == 0xAA63: true

head = 0xFFFF

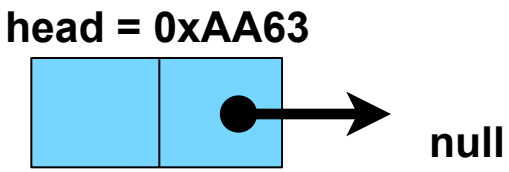
return true

exit while loop

return

A Lock-free Stack in C

```
void push(int t) {
   Node* node = new_node(t);
   do {
        node->next = head;
   } while (!CAS(&head, node->next, node));
}
```



Thread #1: push()

node = 0xAAAA

node->next = 0xAA63

// CAS

head != 0xAA63

return false

node->next = 0xFFFF

// CAS

head == 0xFFFF

return true

exit while loop

return

Thread #2: push()

node = 0xFFFF

node->next = 0xAA63

// CAS

head == 0xAA63: true

head = 0xFFFF

return true

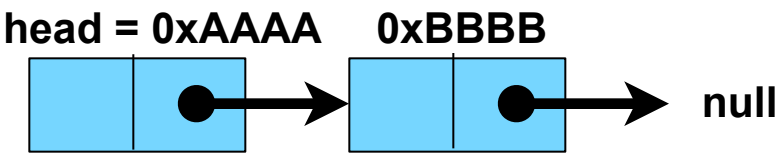
exit while loop

return

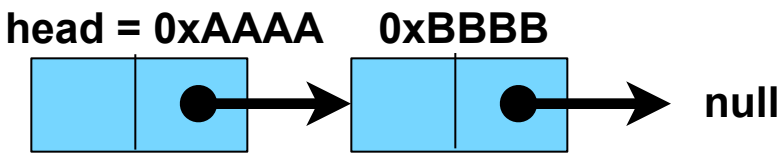
The ABA Problem

- There is a subtle problem with the code before, due to an (unlikely but possible) execution
- The behavior is due to the memory manager, i.e., the thing that does malloc/new and free/ garbage collect
- Let's see this on an example
  - Yet another example of why concurrency is hard and why you need to know low-level stuff (in this case, OS stuff)
  - Remember the "What if the constructor is inlined?" DCL problem (in that case, compiler stuff)

ABA Problem Example



ABA Problem Example



Thread #1:

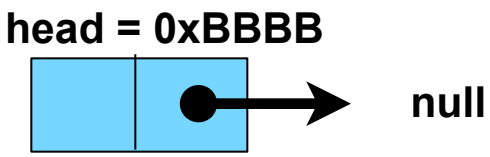
pop():

sees that head = 0xAAAA

set next to head->next = 0xBBBB

interrupted before the CAS

ABA Problem Example



Thread #1:

pop():

sees that head = 0xAAAA

set next to head->next = 0xBBBB

interrupted before the CAS

Thread #2:

pop(): removes 0xAAAA

free(0xAAAA)

ABA Problem Example

null

Thread #1:

pop():

sees that head = 0xAAAA

set next to head->next = 0xBBBB

interrupted before the CAS

Thread #2:

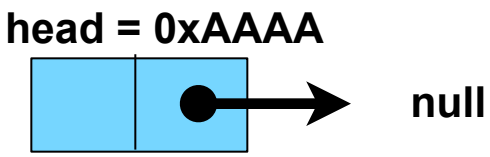
pop(): removes 0xAAAA

free(0xAAAA)

pop(): removes 0xBBBB

free(0xBBBB)

ABA Problem Example



Thread #1:

pop():

sees that head = 0xAAAA

set next to head->next = 0xBBBB

interrupted before the CAS

Thread #2:

pop(): removes 0xAAAA

free(0xAAAA)

pop(): removes 0xBBBB

free(0xBBBB)

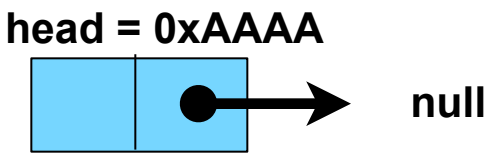
push():

new_node() returns 0xAAAA!!!

head = 0xAAAA

Address recycling by the memory allocator

ABA Problem Example



Thread #1:

pop():

sees that head = 0xAAAA

set next to head->next = 0xBBBB

interrupted before the CAS

CAS sees that head = 0xAAAA, so it hasn't changed, and therefore **sets head to next=0xBBBB**

But 0xBBBB has been freed!

Thread #2:

pop(): removes 0xAAAA

free(0xAAAA)

pop(): removes 0xBBBB

free(0xBBBB)

push():

new_node() returns 0xAAAA!!!

head = 0xAAAA

Does Address Recycling Happen?

- Easy to check
- Let's just run:

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  char *x = (char *)malloc(100);
  free(x);
  char *y = (char *)malloc(100);
  printf("Address recycling: %s\n",
         (x == y ? "yes" : "no"));
}
```

Solving the ABA Problem

- The ABA Problem is a well-known problem when implementing lock free data structures
- One solution: Doubleword CAS
  - Don’t simply use a pointer
  - Use a 128-bit data structure that has a pointer and a counter
  - Each time we use the pointer, we increment the counter
  - Each time we allocated memory for a new “object” we set the counter to 0 in the data structure
  - We always CAS the whole data structure
    - Modern architectures provide a 128-bit CAS, so we can CAS the whole 128-bit data structure
  - This way, CAS will fail if the counter value has changed

Take-away

- There is a lot of complexity there, but the rewards can be huge
  - Lockfree data structures is partly why java.util.concurrent implementations are radically better than what you could do using just synchronized
  - java.util.concurrent, Boost, etc. are full of CAS instructions
- There are many references on this topic
- A good introduction article is:
  - **Concurrent programming without locks** , K. Fraser and T. Harris, ACM Transactions on Computer Systems, Vol. 25 (2), May 2007 (yes, it's old)
  - http://www.cl.cam.ac.uk/research/srg/netos/papers/2007-cpwl.pdf
- Let's now look at the "The Silently Shifting Semicolon" reading in this module