Concurrent Data Structure

Erik Saule
esaule@uncc.edu

Parallel Computing

Learning Outcomes

At the end of this lecture, you will be able to

- Make data structure thread safe
- Show a tradeoff between fine-grain locking and coarse-grain locking
- Give a cause of why instructions might not be executed in the order they are written
- Name one atomic operation
- Implement one example of a lock-free data structure

Outline

Motivation

A simple issue

A typical data race on a data structure.

```
Set* s;
s->add(something);
```

Motivation

A simple issue

A typical data race on a data structure.

```
Set* s;
s->add(something);
```

A simple solution

Mutual exclusion works every time.

```
Set* s;
pthread_mutex_lock(&mut);
s->add(something);
pthread_mutex_unlock(&mut);
```

Motivation

A simple issue

A typical data race on a data structure.

```
Set* s;
s->add(something);
```

A simple solution

Mutual exclusion works every time.

```
Set* s;
pthread_mutex_lock(&mut);
s->add(something);
pthread_mutex_unlock(&mut);
```

Why would we want to do something else ?

Locking granularity

What if the set was implemented as an hash table with open hashing (separate chaining)?

Locking granularity

What if the set was implemented as an hash table with open hashing (separate chaining)?

- `add` actually does `bucket[hash(something)]->add(something);`
- `hash` is certainly re-entrant
- One could associate a lock with each `bucket[i]`
- Very unlikely two threads will collide

Locking granularity

What if the set was implemented as an hash table with open hashing (separate chaining)?

- `add` actually does `bucket[hash(something)]->add(something);`
- `hash` is certainly re-entrant
- One could associate a lock with each `bucket[i]`
- Very unlikely two threads will collide

- That may be too many locks
- Initializing the locks could become expensive
- Maybe 100 locks for the entire hash table ?
  - More likely to have collisions than one per bucket
  - But way less locks to manage

Locking granularity

What if the set was implemented as a Binary Search Tree?

Locking granularity

What if the set was implemented as a Binary Search Tree?

- One could lock per intermediate node
- The $\theta(\log(n))$ locks would be held for $O(1)$ time instead of holding 1 lock for $\theta(\log(n))$ time
- Could fluidify the execution

Locking granularity

What if the set was implemented as a Binary Search Tree?

- One could lock per intermediate node
- The $\theta(\log(n))$ locks would be held for $O(1)$ time instead of holding 1 lock for $\theta(\log(n))$ time
- Could fluidify the execution

- Maybe a different kind of tree, such as a k-ary tree, could reduce locking overhead.

Alternative strategies

What if each thread was working on its own set?

- Assume only add and no look-up during adds
- No locking overhead
- But you'll have a merging overhead once all adds are done

Outline

Atomicity of reads and writes

How bad can things get when two threads execute `*p = 1;` or `v = *p;` ?

Atomicity of reads and writes

How bad can things get when two threads execute `*p = 1;` or `v = *p;` ?
Of course there is a data race.
But depending on the type of `*p` worse can happen.

- If `*p` is too small (less than 32-bit on x86), then 32-bit are read, `*p` is written at the correct location in a register and then the 32-bit are written.
- If `*p` is too large (depending on compiler and arch), then reads and writes happen per chunks.

Atomic operations

Hardware support atomic instruction. The precise set depends on architecture.

Compare and Swap

CAS (A, val, newval) does atomically:

```
bool cnd = (*A == val);
if (cnd)
  *A = newval;
return cnd;
```

fetch-and-add

FAD(A, val) does atomically:

```
oldval = *A;
*A += val;
return oldval;
```

The volatile keyword

`volatile int c;` tells the compiler that the value of `c` might change without local code doing anything.

```
int c;

void waitOnC(){
   while (c != 0) {
      sleep(1);
   }
}
```

might not do what you think it does.
`c` should be volatile.

Out-of-order execution and fences

Compiler reordering

- Compilers optimize code for coherent sequential execution.
- It may change the order of the instruction.

```
data = x;
dataavailable = true;
```

What if these are swapped?
Unintended consequence for multiple thread.

Out-of-order execution and fences

Compiler reordering

- Compilers optimize code for coherent sequential execution.
- It may change the order of the instruction.

```
data = x;
dataavailable = true;
```

What if these are swapped? Unintended consequence for multiple thread.
`asm volatile("" ::: "memory");` prevents GCC to reorder the instruction.

Out-of-order execution and fences

Compiler reordering

- Compilers optimize code for coherent sequential execution.
- It may change the order of the instruction.

```
data = x;
dataavailable = true;
```

What if these are swapped?
Unintended consequence for multiple thread.
`asm volatile("" ::: "memory");` prevents GCC to reorder the instruction.

Processor reordering

Some processors (e.g., all x86) can execute instructions out-of-order.
Assembly instructions can be executed in an order different than the order they appear in the compiled code.
Of course, the processor guarantees the sequential correctness of these operations. But one needs to be careful of memory operation reordering as well.
Different architectures have different memory models, so be careful for portability.

Outline

non-blocking, lock-free and wait-free operations

Non-blocking

The suspension of one thread can not indefinitely block others.
So if there is a mutex, the operation is not non-blocking.

non-blocking, lock-free and wait-free operations

Non-blocking

The suspension of one thread can not indefinitely block others.
So if there is a mutex, the operation is not non-blocking.

Lock-freedom

Non-blocking and for any ordering execution and suspension, some global progress eventually happens.

non-blocking, lock-free and wait-free operations

Non-blocking

The suspension of one thread can not indefinitely block others.
So if there is a mutex, the operation is not non-blocking.

Lock-freedom

Non-blocking and for any ordering execution and suspension, some global progress eventually happens.

Wait-freedom

Lock-free and guarantees that each operation completes in a bounded number of steps.

Appending to a buffer

```
char* buffer;
char* buffer_end;

void append (char* data) {
   size_t len = strlen(data);
   char* pos = fetch_and_add (buffer_end, len);
   memcpy (pos, data, len);
}
```

Pay attention to having enough memory.

Appending to a linked list

```
struct node {
  int val;
  node* next;
};

void append (int val, node* n) {
  node* newnode = new node;
  newnode->val = val;
  newnode->next = NULL;
  bool done = false;
  while (!done) {
    while (n->next != NULL)
      n = n->next;
    done = compare_and_swap(&(n->next), NULL, newnode);
  }
}
```

Appending to a linked list

```
struct node {
  int val;
  node* next;
};

void append (int val, node* n) {
  node* newnode = new node;
  newnode->val = val;
  newnode->next = NULL;
  bool done = false;
  while (!done) {
    while (n->next != NULL)
      n = n->next;
    done = compare_and_swap(&(n->next), NULL, newnode);
  }
}
```

Works great for concurrent add and concurrent traversals.

Be careful with deletion.

Adding in a set with presence flag

With integer flags

```
int* flags;

void add(int obj) {
  flags[obj] = 1;
}
```

Be careful that shorter types than `int` might not have atomic writes.

Adding in a set with presence flag

With integer flags

```
int* flags;

void add(int obj) {
  flags[obj] = 1;
}
```

Be careful that shorter types than `int` might not have atomic writes.

With bit flags

```
int* flags;

void add(int obj) {
  int off = obj/(sizeof(int)*4);
  int bit = obj%(sizeof(int)*4);

  bool done = false;
  while (!done) {
    int oldv = flags[obj];
    int newv  = oldv | (1<<bit);
    if (compare_and_swap (&(flags[obj]), oldv, newv))
      done = true;
  }
}
```

Outline

External

Lock-free:

- Andrei Alexandrescu. Lock-Free Data Structures. 2007.
- Keir Fraser, Tim Harris. Concurrent Programming Without Locks.ACM Transactions on Computer Systems, Vol. 25 (2), May 2007
- Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. 19th International Symposium on Distributed Computing (DISC), September 2005

Atomics:

- `https://en.wikipedia.org/wiki/Compare-and-swap`
- `https://en.wikipedia.org/wiki/Fetch-and-add`
- `https://en.wikipedia.org/wiki/Test-and-set`

concurrent data structure:

- in TBB `https://software.intel.com/en-us/node/506169`
- lib lfds `http://liblfds.org/`

Compiler:

- volatile for embedded programming: `http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword`

ordering:

- On compiler reordering: `http://preshing.com/20120625/memory-ordering-at-compile-time/`
- On cpu reordering: `http://preshing.com/20120515/memory-reordering-caught-in-the-act/`
- C++ on memory ordering: `http://en.cppreference.com/w/cpp/atomic/memory_order`
- Memory ordering of different architectures: `https://en.wikipedia.org/wiki/Memory_ordering`