

Atomics and Memory Consistency

Aug 29th, CS 6530

James McMahon, Hunter McCoy

Putting the A in ACID



Nope, the other kind of ACID

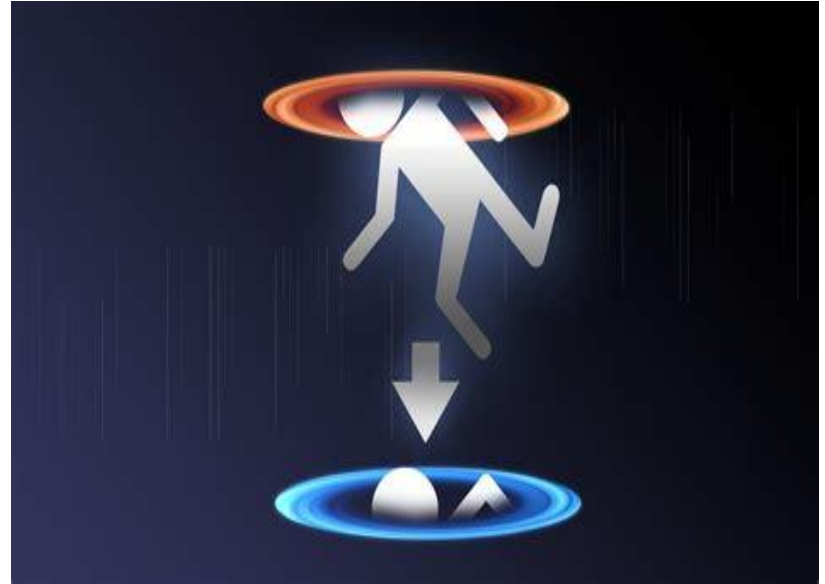
Specifically, **Atomicity**

- Atomicity comes from the Greek atomos, meaning “indivisible”
- An atomic operation is a set of smaller operations that all occur **simultaneously**

Why is this important?

What we mean by “Consistency”

- It is important for memory to be in a “consistent” state, meaning that there are no partially modified portions of memory.
- For example, if you had a portal, you would want to ensure that you either made it entirely through the portal, or not at all. You wouldn’t want your torso to make it through but not your legs!



Applying this idea to Memory

- There are cases when we don't want operations to be broken up by other operations. And if these operations were to be broken up our memory would be "inconsistent".
- If we had two threads writing to a string and then reading from it we would want that string to have one of two options.
 - Foo = "hello world"
 - Foo = "this is a test"
- We wouldn't want to see something like:
 - "this isworld"
 - "hello a test".

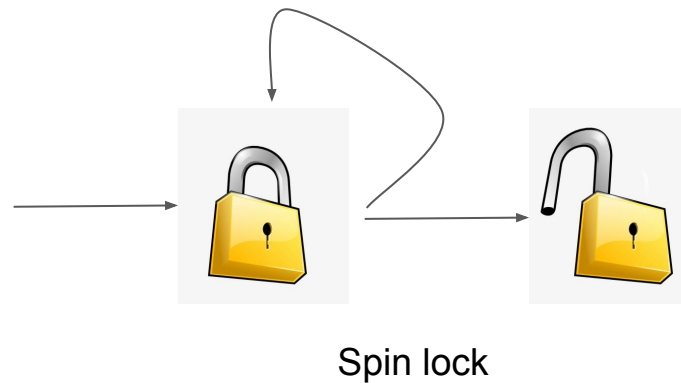
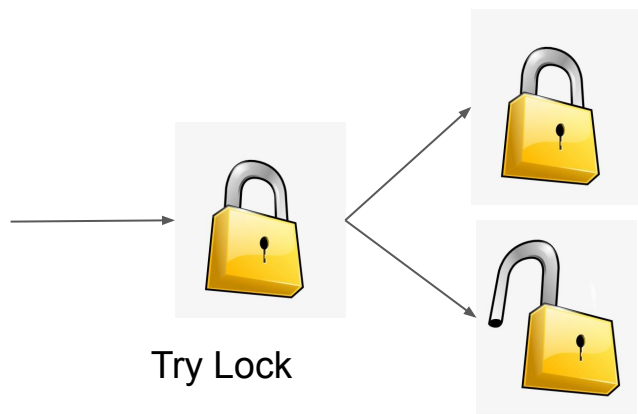
Enforcing memory consistency

- When multiple threads are working on the same object, we can enforce consistency by only allowing one thread to operate at a time.
 - This is a property known as **mutual exclusion**, and a section of code that enforces it is known as a **critical region**.
- To build a critical region, we can use locks (mutexes) or atomics



Locks

- A **lock** is a barrier that only allows one thread at a time to pass through
 - Comes in two flavors
 - A **try-lock** makes one attempt to acquire the lock, and returns the result
 - A **wait-lock** stalls the thread until the lock is acquired.
- Library locks (`pthread_mutex_t`) typically controlled by the OS
 - Slower for most use cases



Atomics

- Atomic instructions create a critical region for **one** operation
- Typically implemented at the hardware level, much faster than locking
- Necessary for building lock-free and wait-free data structures.

Common atomics

type atomic_load(type * x): atomically read memory from x

void atomic_store(type * x, type val): store val in x

type atomic_exchange(type * x, type val): replace the value in x with val, and return the old value.

type compare_exchange(type * x, type expected, type new): if x == expected, atomically replace with new. Else do nothing. Returns the old value in x.

Arithmetic: Add, Sub, And, or, Xor: **op(type * x, type y).**

More info in the project README

Relevant GNU compiler atomics

bool: `__sync_bool_compare_and_swap` (type *ptr, type oldval, type newval)

If *ptr == oldval then set *ptr = newval (returns true if operation succeeded)

type: `__sync_val_compare_and_swap` (type *ptr, type oldval, type newval)

If *ptr == oldval, set *ptr = newval (returns oldvalue either way)

type: `__sync_lock_test_and_set` (type *ptr, type value)

Returns the old value, sets it equal to the new value

void: `__sync_lock_release` (type *ptr)

Sets the value of the pointer to zero

//to access these compiler atomics

```
#if !defined(_GNU_SOURCE)
#define _GNU_SOURCE
#endif
```

Examples and descriptions taken from:

https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html

https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

Basic idea behind implementing a spin lock

A way of thinking through a spin lock is that you have acquired it if the old value was “unlocked” and you set it to “locked”. If that didn’t happen (the lock was already taken) then you “spin” and retry until you can set it from “unlocked” to “locked”. Using atomics, only one thread at a time will be able to see the case of changing “unlocked” to “locked”.

Example: Building a lock with atomics

```
bool lock = 0;

void stall_lock(){

    bool currently_locked = 1;

    while (currently_locked){

        //attempt to acquire the lock
        //expect old value to be 0 (unlocked)
        //if it is 0 we acquire the lock
        currently_locked = __sync_val_compare_and_swap(&lock, 0, 1);

    }

}

void unlock(){

    __sync_val_compare_and_swap(&lock, 1, 0);

}
```

Volatile

- **Volatile** is a keyword that prevents the compiler from making optimizations on a variable
 - Prevents memory reordering during compilation
 - Prevents caching
 - Prevents compiler optimizations

For example, without volatile something like: `while(foo > bar){//empty}` could become `if(foo > bar){while();}`.





*At least for new material

Project 1

Your task: implement a reader-writer lock

- Write lock/unlock: prevents readers and writers from entering. Waits until all readers in the lock have exited before entering.
- Read lock/unlock: Waits until no writer is in the lock. **Multiple readers can enter the lock**

Debugging parallel programs

Always turn on debug flags for debugging (**make D=1**)

- 1) GDB: set breakpoints and step through code
 - a) <https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>
- 2) Perf: record data on a run and identify hotspots
 - a) <https://phoenixnap.com/kb/linux-perf>

Fixing correctness

One of the most effective ways to debug parallel code is to play an adversary:

- 1) Identify the issue that is breaking (you're trying to make it happen)
- 2) You control thread scheduling - try and think of a schedule that



You got this :D

How to enforce Memory Consistency

We enforce memory consistency by using locks. Locks allow us to stop a second thread from operating on a region that another thread is already operating on.

But how do we implement locks?

Trying to make a lock

```
while(lock); //wait for the lock to be unlocked
```

```
lock = true //lock the region
```

```
//important code here
```

```
lock = false //unlock the region
```

Trying to make a lock pt 2

```
while(1){  
    while(lock);  
    lock = my_id //lock the region  
    if(lock != my_id)  
        continue;  
    //important code here  
    lock = 0 //unlock the region  
}
```

So, how do we make a lock?

How do we “lock” the operations that occur when using the lock?

Motivating case: serial vs parallel addition

```
int x = 0

for (int i = 0; i < 1000; i++)
    x += 1
}
```

Parallel case

```
int x = 0

#pragma omp parallel for
for (int i = 0; i < 1000; i++){
    x += 1
}
```

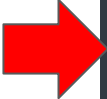

Why are they different?

$x = x+1$ isn't really one instruction


```
ld x, r1
add r1, r1, 1
str x, r1
```

Why are they different?

$x = x+1$ isn't really one instruction




```
ld x, r1
add r1, r1, 1
str x, r1
```




```
ld x, r1
add r1, r1, 1
str x, r1
```

Why are they different?

$x = x+1$ isn't really one instruction



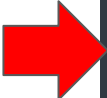
```
ld x, r1
add r1, r1, 1
str x, r1
```




```
ld x, r1
add r1, r1, 1
str x, r1
```

Why are they different?

$x = x+1$ isn't really one instruction




```
ld x, r1
add r1, r1, 1
str x, r1
```




```
ld x, r1
add r1, r1, 1
str x, r1
```

Why are they different?

$x = x+1$ isn't really one instruction



```
ld x, r1
add r1, r1, 1
str x, r1
```



```
ld x, r1
add r1, r1, 1
str x, r1
```



Two adds but x is only incremented once

Using atomics

Atomic operations allow us to perform a combination of instructions “instantaneously”. This allows us to ensure that specific sets of CPU instructions do not become interleaved with another thread’s instructions and execute to completion before being descheduled.

Types of atomics

Locks and critical regions

Lock contention

