Lecture 21 — Atomic Types

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi
jzarnett@uwaterloo.ca, smzahedi@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

Atomic Types

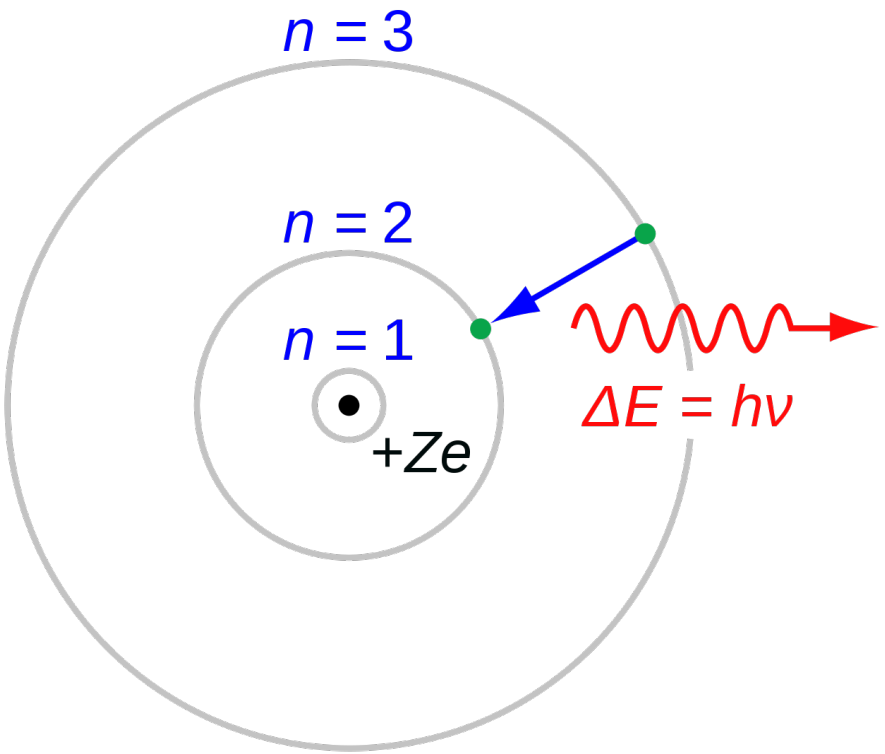

Image Credit: Wikipedia user JabberWok

A testament to how good humans are at blowing things up...

Atomic Types

Frequently we have a code pattern that looks something like this:

```
pthread_mutex_lock( lock );
shared_var++;
pthread_mutex_unlock( lock );
```

It seems like a lot of work to lock and unlock the mutex, no?

Thinking back to the “test and set” type of instruction from earlier, wouldn’t it be nice if we could do that sort of thing for something like incrementing a variable?

Hey glibc, Could You Please...

The GNU (Linux) standard C library (`glibc`) provides operations that are guaranteed to execute atomically, to avoid simple race conditions

Where possible, the compiler will try to turn these into uninterruptible hardware instructions.

Otherwise a function that has locking will be used to implement the atomic nature.

You Can't Take It With You

These are, however, glib specific, and not necessarily available or portable.

In the C11 (2011) standard, atomic types were finally introduced as part of the language specification.

In the specification, we see `type` as the type.

In its place you would use an `int` for an integer.

A valid type is one that 1, 2, 4, or 8 bytes in length (integral or pointer).

Atomic Operations

To set a value:

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

The following functions are used to swap two values, only if the old value matches the expected (i.e., what was provided as the second argument):

```
bool __sync_bool_compare_and_swap( type *ptr, type oldval, type newval );
type __sync_val_compare_and_swap( type *ptr, type oldval, type newval );
```

Change and Return

The following functions perform the operation and return the *old* value:

```
type __sync_fetch_and_add( type *ptr, type value );
type __sync_fetch_and_sub( type *ptr, type value );
type __sync_fetch_and_or( type *ptr, type value );
type __sync_fetch_and_and( type *ptr, type value );
type __sync_fetch_and_xor( type *ptr, type value );
type __sync_fetch_and_nand( type *ptr, type value );
```

The following functions perform the operation and return the *new* value:

```
type __sync_add_and_fetch( type *ptr, type value );
type __sync_sub_and_fetch( type *ptr, type value );
type __sync_or_and_fetch( type *ptr, type value );
type __sync_and_and_fetch( type *ptr, type value );
type __sync_xor_and_fetch( type *ptr, type value );
type __sync_nand_and_fetch( type *ptr, type value );
```

Reading is... Fun?

Interestingly, for x86 there is no atomic read operation.

The (normal) read itself is atomic for 32-bit-aligned data.

This behaviour is specific to x86 and we try to avoid that.

If we do rely on this, however, we could get an out-of-date value.

If you want to really be sure you did get the latest, you can use one of the above functions and add or subtract 0.

Use Tool Appropriately

```
struct point {
  volatile int x;
  volatile int y;
};
__sync_lock_test_and_set( p1->x, 0 );
__sync_lock_test_and_set( p1->y, 0 );

/* Somewhere else in the program */
__sync_lock_test_and_set( p1->x, 25 );
__sync_lock_test_and_set( p1->y, 30 );
```

Does this work?

Use Tool Appropriately

Although the set of each of x and y is atomic, the operation as a whole is not.

We could see invalid states, like (25, 0) or (0, 30).

You Spin Me Right Round...

Another common technique for protecting a critical section in Linux is the *spinlock*.

This is a handy way to implement constant checking to acquire a lock.

Unlike semaphores where the process is blocked if it fails to acquire the lock, a thread will constantly try to acquire the lock.

When would we want this behaviour?

When It's Worth It!

It would be better to let another thread execute.

Except when the amount of time waiting on the lock might be small.

Specifically, less than it would take to block the process, switch to another, and unblock it when the value changes.

Spinlock

```
spin_lock( &lock )
    /* Critical Section */
spin_unlock( &lock )
```

In addition to the regular spinlock, there are *reader-writer-spinlocks*.

Like the readers-writers problem discussed earlier, the goal is to allow multiple readers but give exclusive access to a writer.

Counter	**Flag**	**Interpretation**
0	1	The spinlock is released and available.
0	0	The spinlock has been acquired for writing.
n ($n > 0$)	0	The spin lock has been acquired for reading by n threads.
n ($n > 0$)	1	Invalid state.