ECE 252: Systems Programming and Concurrency Spring 2024

Lecture 21 — Atomic Types

Prepared by Jeff Zarnett, taught by Seyed Majid Zahedi

Atomic Types

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

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

While is is fully correct, if this happens frequently there is a lot of locking and unlocking on the same mutex, just to do the increment. So there's a fair amount of overhead on this. 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? We can!

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.

The kernel itself contains an atomic type, `atomic_t`, but this is not intended for use outside of the kernel.

The following function listings are an overview of the atomic operations, from [FSF19]. These are, however, `glibc` specific, and not necessarily available in a general system. In the C11 (2011) standard, atomic types were finally introduced as part of the language specification itself. But before this we just had implementation-specific options. In the specification, we see `type` as the type, but that's of course not a real 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, if it's an integral type or a pointer.

The following function is used to assign a new value, and returns the old value. However, the name is unfortunate, because it conflicts with the hardware instruction test-and-set which we discussed earlier. But this atomically sets the value of the variable as expected:

```
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 );
```

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 );
```

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 have mostly tried to avoid relying on anything that is implementation-specific behaviour... 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. And that would be a bit more portable as well. But for true portability you will need to use C11 or a semaphore/mutex.

Atomic operations are helpful for scenarios like a single variable being modified and read. But atomic operations are not always ideal. Consider this:

```
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 );
```

Although the set of each of `x` and `y` is atomic, the operation as a whole is not. The write of `x` could succeed and then a read of both in a different thread could take place before the write of `y`, meaning that a reader would see (25, 0) when that's probably not valid. Similarly, the state could be totally corrupted if another thread did atomic writes of (10, 15) in between the two, leading to a final state of (10, 30).

When a number of writes need to take place as a "package", then a mutex type is the appropriate choice.

Spinlocks. 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. The implementation is an integer that is checked by a thread; if the value is 0, the thread can lock it (set the value to 1) and continue; if it is nonzero, it constantly checks the value until the value becomes 0. As you know, this is very inefficient; it would be better to let another thread execute, except in the circumstances where the amount of time waiting on the lock might be less than it would take to block the process, switch to another, and unblock it when the value changes [Sta14].

```
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. This is implemented as a 24-bit reader counter and an unlock flag, with the meaning defined as follows [Sta14].

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.

There are further additional details related to use of spinlocks, which can of course be explored by reading the Linux kernel documentation.

References

[FSF19] Inc. Free Software Foundation. Built-in functions for atomic memory access, 2019. Online; accessed 3-July-2019. URL: `https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html`.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.