【OS】Day33

Class	Operating System: Three Easy Pieces
Date	@February 7, 2022

【Ch28】Locks(3)

28.7 Building Working Spin Locks with Test-and-Set

The simplest bit of hardware support to understand is known as a test-and-set instruction.

We define what the test-and-set instruction does via the following C code snippet:

```
int TestAndSet(int *old_ptr, int new) {
  int old = *old_ptr; //fetch old value at old_ptr
  *old_ptr = new; //store 'new' into old_ptr
  return old; //return the old value
}
```

What the test-and-set instructions does is as follows. It returns the old value pointed to by the `old_ptr`, and simultaneously updates said value to new.

The key, of course, is that this sequence of operations is performed atomically.

The reason it is called “test and set” is that it enables you to “test” the old value(which is what is returned) while simultaneously “setting” the memory location to a new value.

This slightly more powerful instruction is enough to build a simple spin lock.

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

Imagine first the case where a thread calls `lock()` and no other thread currently holds the lock; thus, flag should be 0. When the thread calls `TestAndSet(flag, 1)`, the routine will return the old value of flag, which is 0; thus, the calling thread, which is testing the value of flag, will not get caught spinning in the while loop and will acquire the lock.

The thread will also atomically set the value to 1, thus indicating that the lock is now held. When the thread is finished with its critical section, it calls `unlock()` to set the flag back to zero.

Now imagine we have a thread arises when another one already holds the lock. In this case, this thread will call `lock()` and then call `TestAndSet(flag, 1)` as well.

This time, `TestAndSet()` will return the old value at flag, which is 1. As long as the lock is held by another thread, `TestAndSet()` will repeatedly return 1, and thus this thread will spin and spin until the lock is finally released.

By making both the *test*(of the old lock value) and *set*(of the new value) a single atomic operation, we ensure that only one thread acquires the lock.

This type of lock is usually referred to as a spin lock. It is the simplest type of lock to build, and simply spins, using CPU cycles, until the lock becomes available.

28.8 Evaluating Spin Locks

- The most important aspect of a lock is correctness: *does it provide mutual exclusion?*

  The answer here is yes: the spin lock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.

- The next axis is fairness: *how fair is a spin lock to a waiting thread?*

  The answer here is bad news: spin locks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever, under contention. Simple spin locks are not fair and may lead to starvation.

- The final axis is performance: *what are the costs of using a spin lock?*

  For spin locks, in the single CPU case, performance overheads can be quite painful: When one thread holds the lock, then the scheduler might then run every other thread, each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.

  However, spin locks work fairly well on multiple CPUs. Imagine Thread A on CPU1 and Thread B on CPU2, both contending for a lock. If Thread A grabs the lock, and then Thread B tries to, B will spin(on CPU2).

  However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.

28.9 Compare-And-Swap

Another hardware primitive that some systems provide is known as the compare-and-swap instruction.

The C pseudocode for this single instruction is found in the figure below:

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int original = *ptr;
3      if (original == expected)
4          *ptr = new;
5      return original;
6  }
```

Figure 28.4: **Compare-and-swap**

The basic idea is for compare-and-swap to test whether the value at the address specified by `ptr` is equal to `expected`; if so, update the memory location pointed to by `ptr` with the new value. If not, do nothing.

In either case, return the original value at that memory location, thus allowing the code calling compare-and-swap to know whether it succeeded or not.

With the compare-and-swap instruction, we can build a lock in a manner quite similar to that with test-and-set. For example, we could just replace the `lock()` routine above with the following:

```
void lock(lock_t *lock) {
  while(CompareAndSwap(&lock->flag, 0, 1) == 1)
    ; //spin
}
```

Compare-and-swap is a more powerful instruction than test-and-set. It will be used in the future topic lock-free synchronization.