【OS】Day33(2)

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

【Ch28】Locks(4)

28.10 Load-Linked and Store-Conditional

Some platforms provide a pair of instructions that work in concert to help build critical sections.

On the MIPS architecture, the load-linked and store-conditional instructions can be used in tandem to build locks and other concurrent structures.

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no update to *ptr since LoadLinked to this address) {
7          *ptr = value;
8          return 1; // success!
9      } else {
10         return 0; // failed to update
11     }
12 }
```

Figure 28.5: **Load-linked And Store-conditional**

This is how we can build a lock using the load-linked and store-conditional instructions:

```
1  void lock(lock_t *lock) {
2      while (1) {
3          while (LoadLinked(&lock->flag) == 1)
4              ; // spin until it's zero
5          if (StoreConditional(&lock->flag, 1) == 1)
6              return; // if set-it-to-1 was a success: all done
7                      // otherwise: try it all over again
8      }
9  }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

Figure 28.6: **Using LL/SC To Build A Lock**

First, a thread spins waiting for the flag to be set to 0 (and thus indicate the lock is not held). Once so, the thread tries to acquire the lock via the store-conditional; if it succeeds, the thread had atomically changed the flag's value to 1 and thus can proceed into the critical section.

Note how failure of the store-conditional might arise. One thread calls `lock()` and executes the load-linked, returning 0 as the lock is not held. Before it can attempt the store-conditional, it is interrupted and another thread enters the lock code, also executing the load-linked instruction., and also getting a 0 and continuing.

At this point, two threads have each executed the load-linked and each are about to attempt the store-conditional. The key feature of these instructions is that only one of these threads will succeed in updating the flag to 1 and thus acquire the lock; the second thread to attempt the store-conditional will fail(because the other thread updated the value of flag between its load-linked and store-conditional) and thus have to try to acquire the lock again.

An equivalent approach by David Capel:

```
void lock(lock_t *lock) {
  // If the lock is currently in hold(lock->flag == 1)
  // Or there was a write since the last updateto *ptr
  // Spin Wait
  while(LoadLinked(&lock->flag) || !SotreConditional(&lock->flag, 1))
```

```
    ; //spin
}
```