【OS】Day33(3)

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

【Ch28】Locks(5)

28.11 Fetch-And-Add

One final hardware primitive is the fetch-and-add instruction, which atomically increments a value while returning the old value at a particular address.

The C pseudocode for the fetch-and-add instruction look like this:

```
int FetchAndAdd(int *ptr) {
  int oldVal = *ptr;
  *ptr = oldVal + 1;
  return oldVal;
}
```

In this example, we'll use fetch-and-add to build a more interesting ticket lock.

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

Figure 28.7: **Ticket Locks**

Instead of a single value, this solution uses a `ticket` and `turn` variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn"(myturn).

The globally shared `lock->turn` is then used to determine which thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread(if there is one) can now enter the critical section.

Note one important difference with this solution versus our previous attempts: it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future(once those in front of it have passed through the critical section and released the lock).

Crux: How to avoid spinning?

28.13 A Simple Approach: Just Yield

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

Figure 28.8: **Lock With Test-and-set And Yield**

In this approach, we assume an operating system primitive `yield()` which a thread can call when it wans to give up the CPU and let another thread run.

A thread can be in one of three states(running, ready, or blocked); `yield` is simply a system call that moves the caller from running state to the ready state, and thus promotes another thread to running. Thus, the yielding thread essentially deschedules itself.

Think about the example with two threads on one CPU; in this case, our yield-based approach works quite well. If a thread happens to call `lock()` and find a lock held, it will simply yield the CPU, and thus the other thread will run and finish its critical section.

Let us know consider the case where there are many threads(say 100) contending for a lock repeatedly. In this case, if one thread acquires the lock and is preempted before releasing it, the other 99 will each call `lock()`, find the lock held, and yield the CPU.

Assuming some kind of round-robin scheduler, each of the 99 will execute this run-and-yield pattern before the thread holding the lock gets to run again. While better than our spinning approach(which would waste 99time slice spinning), this approach is still costly; the cost of a context switch can be substantial, and there is thus plenty of waste.

Worse, we have not tackled the starvation problem at all!

28.14 Using Queues: Sleeping Instead Of Spinning

The real problem with our previous approaches is that they leave too much to chance.

The scheduler determines which thread runs next; if the scheduler makes a bad choice, a thread runs that must either spin waiting for the lock(our first approach), or `yield` the CPU immediately(our second approach). Either way, there is potential for waste and no prevention of starvation.

Thus, we must explicitly exert some control over which thread next gets to acquire the lock after the current holder releases it. To do this; we will need a little more OS support, as well as a queue to keep track of which treads are waiting to acquire the lock.

We will use the support provided by Solaris, in terms of two calls: `park()` to put a calling thread to sleep, and `unpark(threadIP)` to wake a particular thread as designated by `threadID`. These two routines can be used to build a lock that puts a caller to sleep if it tries to acquire a held lock and wakes it when the lock is free.

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

We do a couple of things in this example. First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock.

Second, we use a queue to help control who gets the lock next and thus avoid starvation.

You might also observe that in `lock()`, when a thread can not acquire the lock(it is already held), we are careful to add ourselves to a queue(by calling the `gettid()` function to get the thread ID of the current thread), set guard to 0, and yield the CPU.