28. Locks

Operating System: Three Easy Pieces

Locks: The Basic Idea

- Ensure that any critical section executes as if it were a single atomic instruction.
 - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

Add some code around the critical section

```
1 lock_t mutex; // some globally-allocated lock `mutex'
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

- Lock variable holds <u>the state of the lock</u>.
 - available (or unlocked or free)
 - No thread holds the lock.
 - acquired (or locked or held)
 - Exactly one thread holds the lock and presumably is in a critical section.

The semantics of the lock()

- lock()
 - Try to acquire the lock.
 - If <u>no other thread holds</u> the lock, the thread will **acquire** the lock.
 - **Enter** the *critical section*.
 - This thread is said to be the owner of the lock.
 - Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

- **D** The name that the POSIX library uses for a <u>lock</u>.
 - Used to provide mutual exclusion between threads.

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 3 Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4 balance = balance + 1;
5 Pthread_mutex_unlock(&lock);
```

 We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

Building A Lock

- **Efficient locks** provided mutual exclusion at low cost.
- **Building a lock need some help from the hardware and the OS**.

Mutual exclusion

• Does the lock work, preventing multiple threads from entering *a critical section*?

Fairness

• Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)

Performance

The time overheads added by using the lock

Controlling Interrupts

- **Disable Interrupts** for critical sections
 - One of the earliest solutions used to provide mutual exclusion
 - Invented for <u>single-processor</u> systems.

```
1 void lock() {
2 DisableInterrupts();
3 }
4 void unlock() {
5 EnableInterrupts();
6 }
```

Problem:

- Require too much *trust* in applications
 - Greedy (or malicious) program could monopolize the processor.
- Do not work on multiprocessors
- Code that masks or unmasks interrupts be executed *slowly* by modern CPUs

Why hardware support needed?

- **First attempt**: Using a *flag* denoting whether the lock is held or not.
 - The code below has problems.

```
typedef struct lock t { int flag; } lock t;
1
2
3
    void init(lock t *mutex) {
         // 0 \rightarrow lock is available, 1 \rightarrow held
4
5
         mutex - flag = 0;
6
    }
7
   void lock(lock t *mutex) {
8
9
         while (mutex->flag == 1) // TEST the flag
10
                  ; // spin-wait (do nothing)
11
         mutex->flag = 1; // now SET it !
12
   }
13
14 void unlock(lock t *mutex) {
15
         mutex - flag = 0;
16
    }
```

Why hardware support needed? (Cont.)

Problem 1: No Mutual Exclusion (assume flag=0 to begin)

Thread1	Thread2
---------	---------

```
call lock()
while (flag == 1)
interrupt: switch to Thread 2
```

```
call lock()
while (flag == 1)
flag = 1;
interrupt: switch to Thread 1
```

flag = 1; // set flag to 1 (too!)

• **Problem 2**: <u>Spin-waiting</u> wastes time waiting for another thread.

- **D** So, we need an atomic instruction supported by Hardware!
 - test-and-set instruction, also known as atomic exchange

Test And Set (Atomic Exchange)

a An instruction to support the creation of simple locks

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

- **return**(testing) old value pointed to by the ptr.
- *Simultaneously* **update**(setting) said value to new.
- This sequence of operations is performed atomically.

A Simple Spin Lock using test-and-set

```
typedef struct lock t {
1
2
        int flag;
3
    } lock t;
4
5
    void init(lock t *lock) {
6
        // 0 indicates that lock is available,
7
        // 1 that it is held
8
        lock -> flag = 0;
9
    }
10
11
   void lock(lock t *lock) {
12
        while (TestAndSet(&lock->flag, 1) == 1)
13
                    // spin-wait
                  ;
14
    }
15
16
   void unlock(lock t *lock) {
17
        lock -> flag = 0;
18
    }
```

Note: To work correctly on a single processor, it requires a preemptive scheduler.

Evaluating Spin Locks

Correctness: yes

• The spin lock only allows a single thread to entry the critical section.

Fairness: no

- Spin locks don't provide any fairness guarantees.
- Indeed, a thread spinning may spin *forever*.

Performance:

- In the single CPU, performance overheads can be quire *painful*.
- If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

- **Test whether the value at the address(**ptr) is equal to expected.
 - If so, update the memory location pointed to by ptr with the new value.
 - In either case, return the actual value at that memory location.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
                       *ptr = new;
                      return actual;
    }
```

Compare-and-Swap hardware atomic instruction (C-style)



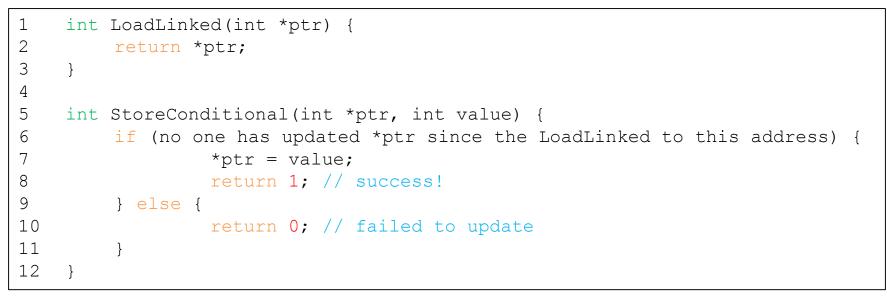
Spin lock with compare-and-swap

Compare-And-Swap (Cont.)

c-callable x86-version of compare-and-swap

```
char CompareAndSwap(int *ptr, int old, int new) {
1
2
         unsigned char ret;
3
4
         // Note that sete sets a 'byte' not the word
5
          _asm____volatile (
6
                  " lock\n"
                  " cmpxchql %2,%1\n"
7
                  " sete %0\n"
8
9
                  : "=q" (ret), "=m" (*ptr)
                  : "r" (new), "m" (*ptr), "a" (old)
10
11
                  : "memory");
12
       return ret;
13
    }
```

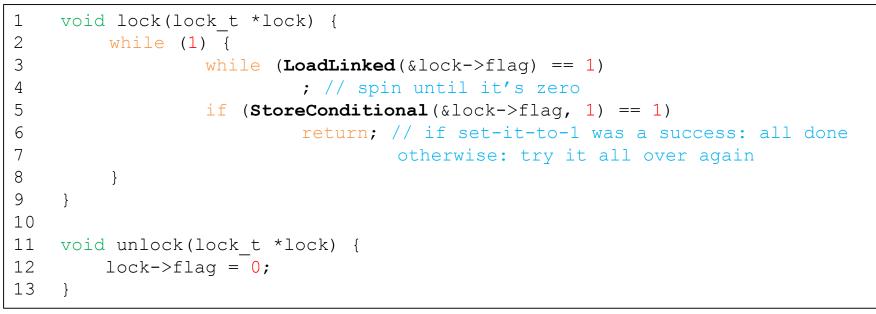
Load-Linked and Store-Conditional



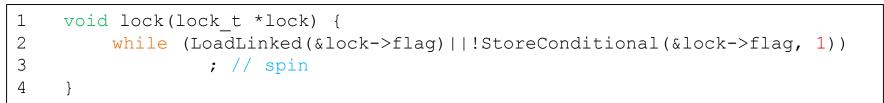
Load-linked And Store-conditional

- The store-conditional *only succeeds* if no intermittent store to the address has taken place.
 - success: return 1 and <u>update</u> the value at ptr to value.
 - fail: the value at ptr is not updates and 0 is returned.

Load-Linked and Store-Conditional (Cont.)



Using LL/SC To Build A Lock



A more concise form of the lock() using LL/SC

Fetch-And-Add

 Atomically increment a value while returning the old value at a particular address.

```
1 int FetchAndAdd(int *ptr) {
2     int old = *ptr;
3     *ptr = old + 1;
4     return old;
5 }
```

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- **Ticket lock** can be built with <u>fetch-and add</u>.
 - Ensure progress for all threads. \rightarrow fairness

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

D Hardware-based spin locks are simple and they work.

- **n** In some cases, these solutions can be quite inefficient.
 - Any time a thread gets caught *spinning*, it wastes an entire time slice doing nothing but checking a value.

How To Avoid *Spinning*? We'll need OS Support too!

A Simple Approach: Just Yield

- **•** When you are going to spin, give up the CPU to another thread.
 - OS system call moves the caller from the *running state* to the *ready state*.
 - The cost of a context switch can be substantial and the starvation problem still exists.

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

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

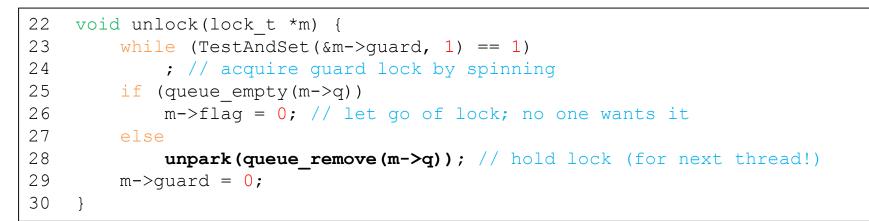
- **Queue** to keep track of which threads are <u>waiting</u> to enter the lock.
- park()
 - Put a calling thread to sleep
- unpark(threadID)
 - Wake a particular thread as designated by threadID.

Using Queues: Sleeping Instead of Spinning

```
typedef struct lock t { int flag; int guard; queue t *q; } lock t;
1
2
3
    void lock init(lock t *m) {
4
        m - > flag = 0;
5
        m - > quard = 0;
6
        queue init (m->q);
7
    }
8
9
    void lock(lock t *m) {
        while (TestAndSet(&m->guard, 1) == 1)
10
11
            ; // acquire guard lock by spinning
12
        if (m->flag == 0) {
13
            m->flag = 1; // lock is acquired
            m->quard = 0;
14
15
   } else {
16
            queue add(m->q, gettid());
            m->quard = 0;
17
18
            park();
19
        }
20 }
21
    ...
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning



Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

In case of releasing the lock (*thread A*) just before the call to park()
 (*thread B*) → Thread B would sleep forever (potentially).

- **Solaris** solves this problem by adding a third system call: setpark().
 - By calling this routine, a thread can indicate it *is about to* park.
 - If it happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping.

```
1 queue_add(m->q, gettid());
2 setpark(); // new code
3 m->guard = 0;
4 park();
```

Code modification inside of lock()

Futex

- **D** Linux provides a futex (is similar to Solaris's park and unpark).
 - futex_wait(address, expected)
 - Put the calling thread to sleep
 - If the value at address is not equal to expected, the call returns immediately.
 - futex_wake(address)
 - Wake one thread that is waiting on the queue.

Futex (Cont.)

- Snippet from lowlevellock.h in the nptl library
 - The high bit of the integer v: track whether the lock is held or not
 - All the other bits : the number of waiters

```
void mutex lock(int *mutex) {
1
2
         int v;
         /* Bit 31 was clear, we got the mutex (this is the fastpath) */
3
         if (atomic bit test set(mutex, 31) == 0)
4
5
                  return;
         atomic increment(mutex);
6
7
         while (1) {
8
                  if (atomic bit test set(mutex, 31) == 0) {
                           atomic decrement (mutex);
9
10
                           return;
11
                  /* We have to wait now. First make sure the futex value
12
13
                     we are monitoring is truly negative (i.e. locked). */
14
                  v = *mutex;
15
                  ...
```

Linux-based Futex Locks

```
16
                 if (v \ge 0)
17
                          continue;
18
                 futex wait(mutex, v);
19
20
   }
21
2.2
    void mutex unlock(int *mutex) {
23
        /* Adding 0x80000000 to the counter results in 0 if and only if
24
           there are not other interested threads */
25
        if (atomic add zero(mutex, 0x8000000))
26
                 return:
27
        /* There are other threads waiting for this mutex,
28
           wake one of them up */
29
        futex wake(mutex);
30
   }
```

Linux-based Futex Locks (Cont.)

Two-Phase Locks

 A two-phase lock realizes that spinning can be useful if the lock is about to be released.

• First phase

- The lock spins for a while, *hoping that* it can acquire the lock.
- If the lock is not acquired during the first spin phase, <u>a second phase</u> is entered,

Second phase

- The caller is put to sleep.
- The caller is only woken up when the lock becomes free later.

 Disclaimer: This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.