Operating Systems

Lecture 5

Locks (Mutex)

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);
```

Locks: The Basic Idea

Lock variable holds the state of the lock.

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 no other thread holds 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.

Pthread Locks - mutex

The name that the POSIX library uses for a lock.

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**.

Evaluating locks – Basic criteria

Correctness

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

Disable Interrupts for critical sections

One of the earliest solutions used to provide mutual exclusion

Invented for single-processor 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.

```
1    typedef struct __lock_t { int flag; } lock_t;
2
3    void init(lock_t *mutex) {
4        // 0 → lock is available, 1 → held
5        mutex->flag = 0;
6    }
7
8    void lock(lock_t *mutex) {
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 : 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!)	

So, we need an atomic instruction supported by Hardware!

test-and-set instruction, also known as *atomic exchange*

A Simple Spin Lock using test-and-set

```
1    typedef struct __lock_t {
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   }
```

Test And Set (Atomic Exchange)

An instruction to support the creation of simple locks

```
1    int TestAndSet(int *ptr, int new) {
2        int old = *ptr;   // fetch old value at ptr
3        *ptr = new;       // store 'new' into ptr
4        return old;       // return the old value
5    }
```

Test-And-Set Hardware atomic instruction (C-style)

return(testing) old value pointed to by the `ptr`.

Simultaneously **update**(setting) said value to `new`.

This sequence of operations is performed atomically.

Test-and-Set-Lock (TSL) Instruction

- A lock is a single word variable with two values
  0 = FALSE = not locked ; 1 = TRUE = locked
- Test-and-set-lock does the following *atomically*:
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

If the returned value was FALSE...
Then you got the lock!!!

If the returned value was TRUE...
Then someone else has the lock
(so try again later)

Compare-And-Swap

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.

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

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

```
1    void lock(lock_t *lock) {
2        while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3                ; // spin
4    }
```

Spin lock with compare-and-swap

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.

No guarantee that threads enter CS in order.

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*.

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 fetch-and-add.

Ensure progress for all threads. → fairness

```
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   void unlock(lock_t *lock) {
17       FetchAndAdd(&lock->turn);
18   }
```

So Much Spinning

Hardware-based Spin locks are simple and they work.

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!

Using Queues: Sleeping Instead of Spinning

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*.

Queue to keep track of which threads are waiting 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;       // lock is acquired or not
   int guard;      // to protect the queue
   queue_t *q;
} lock_t;
```

Using Queues: Sleeping Instead of Spinning

KOREA UNIVERSITY

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

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

Using Queues: Sleeping Instead of Spinning

```
22  void unlock(lock_t *m) {
23      while (TestAndSet(&m->guard, 1) == 1)
24          ; // acquire guard lock by spinning
25      if (queue_empty(m->q))
26          m->flag = 0; // let go of lock; no one wants it
27      else
28          unpark(queue_remove(m->q)); // hold lock (for next thread!)
29      m->guard = 0;
30  }
```

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

Summary

evaluation creteria of mutex

- correctness, fairness, performance

mutex implementations

- controlling interrupts
- spin lock
  - needs hardware support (TSL, etc)
- using queue
  - needs OS support (park(), unpark(), etc)

The END