
Operating System Design and Implementation

Lecture 16: Locking

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm

Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
MIT 6.828 Operating system engineering class, 2018
MIT 6.004 Operating system, 2018
Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University

Outline

- Locks
- Hardware synchronization operators
 - test-and-set
 - compare-and-swap
 - fetch-and-add
 - Load-linked / stored-conditional
- Reducing spin-locking overhead
 - yield ()
 - Futex in Linux

Locks and unlocks

- **Lock: synchronization mechanism that enforces atomicity**
- lock(L) : acquire lock L exclusively
 - Only the process with L can access the critical section
- Unlock(L): release exclusive access to lock L
 - Permitting other processes to access the critical section

Program 0

```
{  
  lock(L)  
  counter ++  
  unlock(L)  
}
```

Shared variable

```
int counter=5  
lock_t L
```

Program 1

```
{  
  lock(L)  
  counter --  
  unlock(L)  
}
```

Software locking -- Interrupt

- In a single-processor system

- **How does it work ?**

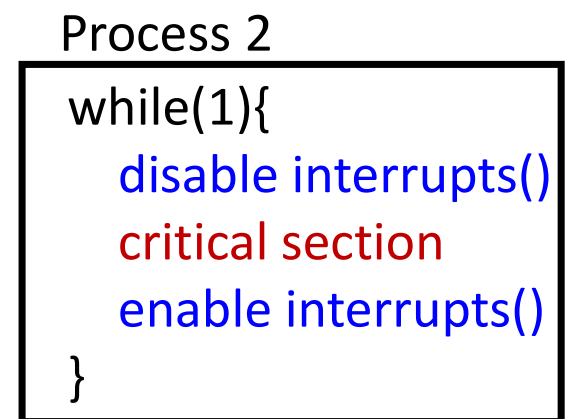
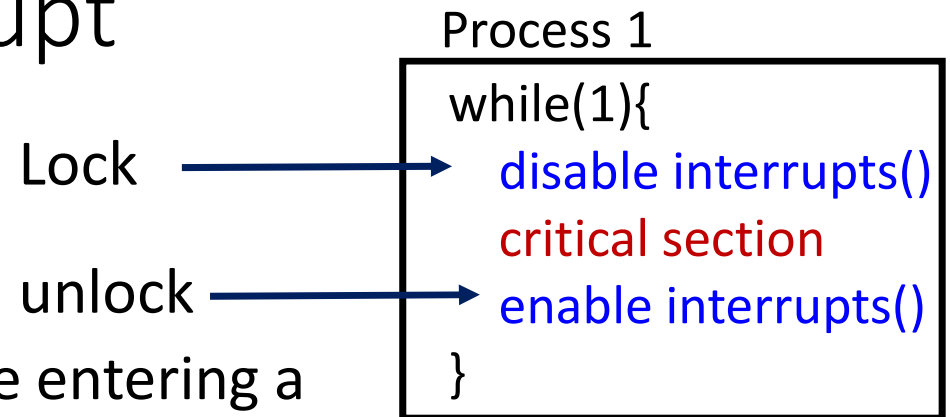
- **Lock** -- turning off interrupts before entering a critical section
- Ensure the code inside the critical section won't be interrupted -> execute as if it were atomic

- **Requires privileges**

- User processes generally cannot disable interrupts (how to trust every requests ?)

- **Not suited for multicore systems**

- Threads can run on different processors and enter the critical section



Problems with disabling interrupts

- Disabling interrupts for long is always bad
 - Can result in **lost interrupts** and **dropped data**
- But what about multiprocessors ?
 - Disabling interrupts on just the local processor is not very helpful
 - Unless all processes are running on the local processor
 - Disabling interrupts on **all** processors is **expensive**

Hardware synchronization Operators

- **test-and-set (loc, t)**
 - Atomically read original value and replace it with “t”
- **compare-and-swap (loc, a, b)**
 - Atomically: if (loc == a) {loc = b;}
- **fetch-and-add (loc, n)**
 - Atomically read the value at loc and replace it with its value incremented by n
- **Load-linked / stored-conditional**
 - Load-linked: loads values from specified address
 - Store-conditional: if no other thread has touched value -> store, else return error

How about hardware locking ?

- Does this scheme provide mutual exclusion ?

Process 1

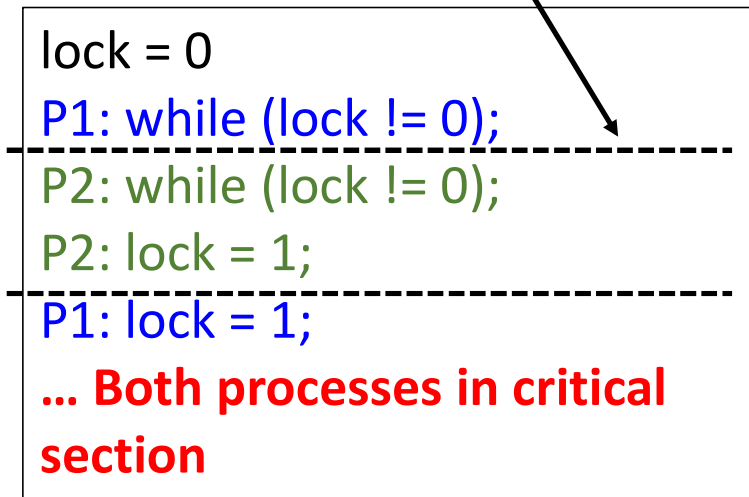
```
while(1){
  while(lock != 0);
  lock = 1; // lock
  critical section
  lock = 0; //unlock
}
```

Process 2

```
while(1){
  while(lock != 0);
  lock = 1; // lock
  critical section
  lock = 0; //unlock
}
```

lock = 0

Context switch



How to make mutual execution ?

- Make the following operations be atomic

Process 1

```
while(1){  
  while(lock != 0);  
  lock = 1; // lock  
  critical section  
  lock = 0; //unlock  
}
```

Make atomic, how?

Test & Set instruction

- Test & set instruction
 - Return the old value pointed to by the old_ptr
 - 'test' the old value
 - 'setting' the memory location to a new value

```
int test_and_set (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
}
```

How to use test_and_set ?

```
int TestAndSet(int *old_ptr, int new)
{
    int old = *old_ptr;
    *old_ptr = new;
    return old;
}
```

- The first invocation of test_and_set will read a 0 and set lock to 1 then return
- The second test_and_set invocation will see lock as 1, and loop continuously until lock becomes 0

```
typedef struct __lock_t {
    int flag;
};
void init (lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}
void lock (lock_t *lock) {
    while (TestAndSet (&lock->flag, 1) == 1);
}
void unlock (lock_t *lock) {
    lock->flag = 0;
}
```

Intel hardware atomic exchange (xchg)

- Why does xchg work ?
 - If two CPUs execute xchg at the same time
 - The hardware ensures that one xchg completes
 - Then the second xchg starts

```
int xchg (int *L)
{
    int prev = *L;
    *L = 1;
    return prev;
}
```

Typical usage
xchg reg, mem

```
int xchg (addr, value) {
    %eax = value
    xchg %eax, (addr)
}
void acquire (int *locked) {
    while (1) {
        if(xchg (locked, 1) == 0)
            break;
    }
}
void release (int *locked) {
    locked = 0;
}
```

Compare-And-Swap

- **Compare-And-Swap**

- 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
- Return the original value at that memory location

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

Compare-And-Swap

- Compare-And-Swap with lock
 - Check if the flag is 0
 - If so, atomically swaps in a 1 thus acquiring the lock
 - Spinning while the lock is held

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

Load-linked and store-conditional (llsc)

- **The load-linked**

- Fetches a value from memory and places it in a register

- **The store-conditional**

- Only succeeds if no intervening store to the address has taken place
- If success, return 1 and update the value at ptr to value
- If fail, 0 is returned

```
int LoadLinked (int *ptr) {  
    return *ptr;  
}  
int StoreConditional (int *ptr, int value) {  
    if (no update to *ptr since LoadLinked  
to this address) {  
        *ptr = value;  
        return 1; // success !  
    } else {  
        return 0; // failed to update  
    }  
}
```

Lock implementation with llsc

- **lock ()**

- A thread spins waiting for the flag to be set to 0
- The thread tries to acquire the lock via the store-conditional
- If succeeds, the thread has atomically changed the flag's value to 1

```
void lock (lock_t *lock) {
    while (1) {
        while (LoadLinked (&lock->flag) == 1);
        // spin until it's zero
        if (StoreConditional (&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock (lock_t *lock) {
    lock->flag = 0;
}
```


llsc Case study

1. The first thread calls lock() and executes ll, return 0 as the lock is not held
2. The first thread is interrupted and another thread enters the lock code
3. The second thread get a 0 in ll
4. Both of them attempt the ss
5. The second thread that attempt ss will fail (why ?)

The key feature of llsc instruction is only one of these thread will succeed in updating the flag 1 and acquire the lock

```
void lock (lock_t *lock) {
    while (1) {
        while (LoadLinked (&lock->flag) == 1);
        // spin until it's zero
        if (StoreConditional (&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock (lock_t *lock) {
    lock->flag = 0;
}
```

Fetch-and-add

- **Fetch-and-add**

- Atomically increments a value while returning the old value at a particular address

- **Ticket lock**

- A thread first does an atomic fetch-and-add on the ticket value (myturn as turn value)
- Globally shared lock->turn is used to decide which thread's turn it is
- Enter the critical section when (myturn == turn)

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

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init (lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock (lock_t *lock) {
    int myturn = FetchAndAdd (&lock->ticket);
    while (lock->turn != myturn);
}

void unlock (lock_t *lock) {lock->turn += 1;}
```

Evaluating spin locks

- **Correctness**

- Does it provide mutual exclusion ?
- Yes, spin lock only allows a single thread to enter the critical section at a time

- **Fairness**

- Does it guarantee a waiting thread will enter the critical section ?
- No, spin locks don't provide any fairness guarantees
- A thread spinning may spin forever under contention

- **Performance**

- The performance overhead is high in the single CPU
- On multiple CPUs, spin locks work reasonably well (why ?)

Case study: yield ()

- What to do ?
 - When a context switch occurs in a critical section
 - Will threads need to spin endlessly and wait for the interrupted (lock-holding) thread to be run again ?
- **yield ()** system call
 - Moves the caller from running state to the ready state
 - Promote another thread to running
 - The yielding thread essentially **deschedules** itself
 - A thread can call when it wants to give up the CPU and let another thread run

yield ()

- Two threads on one CPU
 - A thread happens to call lock() and find a lock held
 - It will simply yield the CPU without spinning
 - The other thread will run and finish its critical section
 - Thus, yield () relieves the spinning lock problem

```
void init () {  
    flag = 0;  
}  
void lock () {  
    while (TestAndSet (&flag, 1) == 1)  
        yield (); // give up the CPU  
}  
void unlock () {  
    flag = 0;  
}
```

The yield () problem

- There are many threads contending for a lock repeatedly
 - **One thread acquires the lock** and is preempted before releasing it
 - The **other 99 threads will each call lock ()**, then find lock held
 - Finally, yield the CPU
 - **Each of the 99 thread will execute the run-and-yield pattern** before the thread holding the lock gets to run again
 - **plenty of waste**
 - **The starvation problem**
 - A thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section

Using queues: Sleeping instead of spinning

```
int lock (lock_t *m) {
    // acquire guard lock by spinning
    while (TestAndSet (&m->guard, 1) == 1);
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park ();
    }
}
```

```
int unlock(lock_t *m) {
    // acquire guard lock by spinning
    while (TestAndSet (&m->guard, 1) == 1);
    if (queue_empty (m->q)) {
        m->flag = 0;
    } else {
        // hold lock for next thread !
        unpark (queue_remove (m->q));
    }
    m->guard = 0;
}
```

park (): put a calling thread to sleep. **unpark(tid)**: wake a particular thread

Wakeup / waiting race

- **Where is the race condition ?**

- A thread will be about to park (it should sleep until the lock is no longer held.)
- A switch at that time to another thread holding the lock and the lock is released
- The subsequent park by the first thread would then sleep forever
- **Wakup / waiting race:**
 - The thread that unpark doesn't know threads are going to park
 - Threads that park don't know the thread is going to unpark

```
int lock (lock_t *m) {
    // acquire guard lock by spinning
    while (TestAndSet (&m->guard, 1) == 1);
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park ();
    }
}
```


setpark()

- **Adding setpark()**
 - If another thread calls unpark before park is actually called
 - **The subsequent park returns immediately instead of sleeping**

```
int lock (lock_t *m) {  
    // acquire guard lock by spinning  
    while (TestAndSet (&m->guard, 1) == 1);  
    if (m->flag == 0) {  
        m->flag = 1; // lock is acquired  
        m->guard = 0;  
    } else {  
        queue_add(m->q, gettid());  
        setpark();  
        m->guard = 0;  
        park ();  
    }  
}
```

futex in Linux

- Callers can use **futex** calls to sleep and wake as need be
 - Each futex has associated with it a specific physical memory location
 - **futex_wait (address, expected)**
 - Puts the calling thread to sleep
 - **futex_wake (address)**
 - Wakes one thread that is waiting on the queue

Locks by using futex

- **Lock using futex**

- A single integer to track
 - Whether the lock is held or not (The high bit of the integer)
 - The number of waiters on the lock (all the other bits)
- If the lock is negative, it is held
 - Because the high bit is set and the bit determines the sign of the integer

```
int mutex_lock (int *mutex) {
    int v;
    /*Bit 31 was clear, we got the mutex*/
    if (atomic_bit_test_set (mutex, 31) == 0)
        return;
    atomic_increment (mutex);
    while (1) {
        if (atomic_bit_test_set (mutex, 31) == 0) {
            atomic_decrement (mutex);
            return;
        }
        // we are monitoring it truly negative (locked)
        v = *mutex;
        if (v >= 0)
            continue;
        futex_wait (mutex, v);
    }
}
```

Summary

- **Lock**
 - Enforce atomicity through the synchronization
- **Interrupt-based lock**
 - Expensive on multiprocessor
- **Hardware synchronization operators**
 - test-and-set ...
- **Spin lock is expensive and error-prone**
 - yield ()