Locks

CMPU 334 – Operating Systems
Jason Waterman
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

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

• Add some code around the critical section

```c
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 object 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 `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 holds the lock
  - Other threads will **block** on the call to lock, until the lock is released
  - If several threads are waiting on the lock, only one will get it when it is released
Pthread Locks - mutex

• The name that the POSIX library uses for a lock
  • Used to provide mutual exclusion between threads

```c
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)
Evaluating locks – Basic criteria

• **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 free)

• **Performance**
  • The time overheads added by using the lock
  • Locks must provide mutual exclusion at low cost

• Building a lock needs help from the **hardware** and the **OS**
Controlling Interrupts

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

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

- **Problems**
  - 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 is executed *slowly* by modern CPUs
Why is hardware support needed?

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

```c
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```
Why hardware support needed? (Cont.)

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

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
</tr>
</thead>
</table>
| ```
call lock()
while (mutex->flag == 1)
interrupt: switch to Thread 2
``` | ```
call lock()
while (mutex->flag == 1)
mutex->flag = 1;
interrupt: switch to Thread 1
``` |

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

• **Problem 2:** Spin-waiting wastes time waiting for another thread

• So, we need an atomic instruction supported by **hardware**
  • *test-and-set* instruction, also known as *atomic exchange*
Test And Set (Atomic Exchange)

• An instruction to support the creation of simple locks

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

• `return` (test) old value pointed to by the `ptr`
• `Simultaneously update` (set) said value to `new`
• This sequence of operations is performed atomically
• x86_64:
  • `xchg rax, (mem)`
A Simple Spin Lock using test-and-set

```c
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1) // spin-wait
}

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

- **Note**: To work correctly on a *single processor*, it requires a preemptive scheduler
  - Why?
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**:
  • For a single CPU, performance overheads can be quite *painful*
  • If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*
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
• x86_64
  • cmpxchg

```c
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)

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

Spin lock with compare-and-swap
Fetch-And-Add

- **Atomically increment** a value while returning the old value at a particular address

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

Fetch-And-Add **Hardware atomic instruction (C-style)**
Ticket Lock

- **Ticket lock** can be built with fetch-and add
  - Ensure progress for all threads \( \rightarrow \) **fairness**

```c
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)
        ; // spin
}

void unlock(lock_t *lock) {
    fetchAndAdd(&lock->turn);
}
```
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**!
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

```
void init() {
    flag = 0;
}

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

void unlock() {
    flag = 0;
}
```

**Lock with Test-and-set and Yield**
Using Queues: Sleeping Instead of Spinning

- **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

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

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

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

- Potential race condition
  - Thread A holds lock
  - Thread B tries to get lock; fails
  - About to call park; switch B -> A
  - Thread A releases lock; switch A -> B
  - Thread B calls park; no thread will wakeup B!

Lock With Queues, Test-and-set, Yield, And Wakeup
Using Queues: Sleeping Instead of Spinning

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

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

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

- 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 the thread happens to be interrupted and the lock is freed before `park` is actually called, the subsequent `park` returns immediately instead of sleeping

Lock With Queues, Test-and-set, Yield, And Wakeup
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, a second phase is entered,
  • Second phase
    • The caller is put to sleep
    • The caller is only woken up when the lock becomes free later

• Another example of a hybrid approach