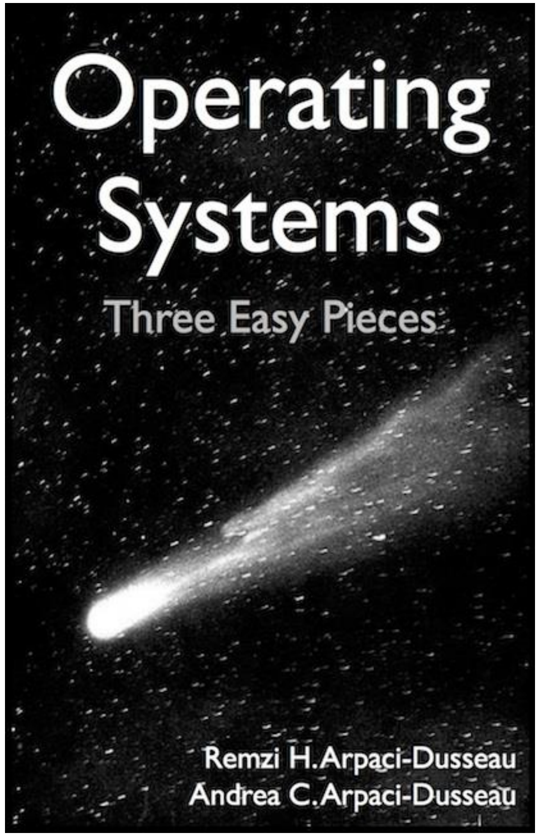
CMSC 125: Operating Systems

- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org

Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- 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.

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

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks: The Basic Idea

- Lock variable holds <u>the state of</u> 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 <u>no other thread holds</u> the lock, the thread will **acquire** the lock
  - **Enter** the *critical section*
    - This thread is said to be <u>the owner of</u> 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

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
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 needs some help from the **hardware** and the **OS**

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)
- **Performance**
  - The time overheads added by using the lock

Approach 1: Controlling Interrupts

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems – examples: `STI` and `CLI` instructions in x86

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

  - 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 is 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;

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 is 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**: 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

```
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 {
    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 <u>a preemptive scheduler</u>

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section

- **Fairness**: no
  - Spin locks <u>don't provide any fairness</u> guarantees
  - Indeed, a thread spinning may spin *forever*

- **Performance**:
  - In the 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

```
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)

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

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) {
    unsigned char ret;

    // Note that sete sets a 'byte' not the word
    __asm__ __volatile__ (
            " lock\n"
            " cmpxchgl %2,%1\n"
            " sete %0\n"
            : "=q" (ret), "=m" (*ptr)
            : "r" (new), "m" (*ptr), "a" (old)
            : "memory");
    return ret;
}
```

Load-Linked and Store-Conditional

```
int LoadLinked(int *ptr) {
    return *ptr;
}

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

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 <u>not updates</u> and 0 is returned

Load-Linked and Store-Conditional (Cont.)

```
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;
}
```

Using LL/SC To Build A Lock

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
        ; // spin
}
```

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

```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 → fairness

```
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 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

```
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();
    }
}
…
```

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

Using Queues: Sleeping Instead of Spinning

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

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

Wakeup/waiting race

- 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

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

Code modification inside of `lock()`

Futex

- 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) {
    int v;
    /* Bit 31 was clear, we got the mutex (this is the fastpath) */
    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 have to wait now. First make sure the futex value
           we are monitoring is truly negative (i.e. locked). */
        v = *mutex;
        …
```

Linux-based Futex Locks

Futex (Cont.)

```
            if (v >= 0)
                    continue;
            futex_wait(mutex, v);
    }
}

void mutex_unlock(int *mutex) {
    /* Adding 0x80000000 to the counter results in 0 if and only if
       there are not other interested threads */
    if (atomic_add_zero(mutex, 0x80000000))
            return;
    /* There are other threads waiting for this mutex,
       wake one of them up */
    futex_wake(mutex);
}
```

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, 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