Engineering & Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Concurrency – Locks

Endadul Hoque

Acknowledgement

- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

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 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 <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);
```

  - Instead of using one big lock (a **coarse-grained** approach), we may be using *different locks* to protect *different variables* (a more **fine-grained** approach) → Increase concurrency.

Building A Lock

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

Evaluating locks – Basic criteria

- **Correctness** (aka 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? (Addressing starvation)

- **Performance**
  - The time overheads added by using the lock
    - **Cases**: (a) No contention, (b) contention on single CPU, (c) contention on multiple CPUs

Naïve solution: Controlling Interrupts

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

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

Naïve solution: Controlling Interrupts

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

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

- Problems:
  - Require too much *trust* in applications
    - Greedy (or malicious) program could monopolize the processor.
  - Do not work on multiprocessors
  - Disabling interrupts for a longer time can lose important interrupts
  - Code that masks or unmasks interrupts be executed *slowly* by modern CPUs

Why is hardware support needed?

- **First attempt**: Using a *flag* denoting whether the lock is held or not.
  - The code 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 is hardware support needed?

- **Problem 1**: No Mutual Exclusion

	Thread1	Thread2
Assume `flag=0`	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!
  - **E.g.,** *test-and-set* instruction, also known as atomic exchange

Test And Set (Atomic Exchange)

- An H/W instruction to support the creation of simple locks
- This instruction operates as follows (C pseudocode):

```
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** old value pointed to by the `ptr` (testing)
  - *Simultaneously* **update** said value to `new` (setting)
  - 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 a preemptive scheduler.

Evaluating Spin Locks (using Test-and-Set)

- **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 quite *substantial*.
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

Some other H/W instructions for Spin Locks

- Compare-and-Swap
- Load-Linked and Store-Conditional
- Fetch-And-Add

If interested to learn more on this, read sections 28.9, 28.10, 28.11 from the book chapter

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 <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;

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;
}
```

```
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

Wakeup/waiting race

- Scenario:
  1. Thread A holds the lock
  2. Thread B attempts to acquire lock, hence ends up adding itself to the queue; but before it could call `park()`, context switch happens
  3. Thread A releases the lock and calls `unpark`(), but no thread to wake up
  4. Thread B regains the CPU and calls `park`() now; thus Thread B goes to sleep (and potentially sleeps forever).

Wakeup/waiting race

- **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`), part of **nptl** library
  - `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.

Reading Material

- **Chapter 28** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin) http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf

Questions?