Lecture 13: Locks

Mythili Vutukuru
IIT Bombay

Locks: Basic idea

- Consider update of shared variable

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

- We can use a special lock variable to protect it

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

- All threads accessing a critical section share a lock
- One threads succeeds in locking – owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the owner
- Pthreads library in Linux provides such locks

Building a lock

- Goals of a lock implementation
  - Mutual exclusion (obviously!)
  - Fairness: all threads should eventually get the lock, and no thread should starve
  - Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources
- Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code
- Implementing locks needs support from hardware and OS

Is disabling interrupts enough?

- Is this enough?
- No, not always!
- Many issues here:
  - Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)
  - Will not work on multiprocessor systems, since another thread on another core can enter critical section
- This technique is used to implement locks on single processor systems inside the OS
  - Need better solution for other situations

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

A failed lock implementation (1)

- Lock: spin on a flag variable until it is unset, then set it to acquire lock
- Unlock: unset flag variable

```
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;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

A failed lock implementation (2)

- Thread 1 spins, lock is released, ends spin
- Thread 1 interrupted just before setting flag
- Race condition has moved to the lock acquisition code!

flag = 0

Thread 1	Thread 2
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!)	

Figure 28.2: **Trace: No Mutual Exclusion**

Solution: Hardware atomic instructions

- Very hard to ensure atomicity only in software
- Modern architectures provide hardware atomic instructions
- Example of an atomic instruction: test-and-set
  - Update a variable and return old value, all in one hardware instruction

```
int TestAndSet(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
}
```

Simple lock using test-and-set

- If TestAndSet(flag,1) returns 1, it means the lock is held by someone else, so wait busily
- This lock is called a spinlock – spins until lock is acquired

```
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 (do nothing)
}

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

flag = 0

1

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

Spinlock using compare-and-swap

- Another atomic instruction: compare-and-swap

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

Figure 28.4: **Compare-and-swap**

- Spinlock using compare-and-swap

1 ⇒ wait

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

0 ⇒ lock

Alternative to spinning

- Alternative to spinlock: a (sleeping) mutex
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
  - `yield()` moves thread from running to ready state

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

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

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

Figure 28.8: **Lock With Test-and-set And Yield**

Spinlock vs. sleeping mutex

- Most userspace lock implementations are of the sleeping mutex kind
  - CPU wasted by spinning contending threads
  - More so if a thread holds spinlock and blocks for long
- Locks inside the OS are always spinlocks
  - Why? Who will the OS yield to?
- When OS acquires a spinlock:
  - It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
  - It must not perform any blocking operation – never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

How should locks be used?

- A lock should be acquired before accessing any variable or data structure that is shared between multiple threads of a process
  - "Thread-safe" data structures
- All shared kernel data structures must also be accessed only after locking
- Coarse-grained vs. fine-grained locking: one big lock for all shared data vs. separate locks
  - Fine-grained allows more parallelism
  - Multiple fine-grained locks may be harder to manage
- OS only provides locks, correct locking discipline is left to the user