W4118: locks

Instructor: Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

Outline

- Critical section requirements
- Implementing locks
- Readers-writer lock

Avoid race conditions

- Critical section: a segment of code that accesses a shared variable (or resource)
- No more than one thread in critical section at a time.

```
// ++ balance
mov    0x8049780,%eax
add    $0x1,%eax
mov    %eax,0x8049780
…

// -- balance
mov    0x8049780,%eax
sub    $0x1,%eax
mov    %eax,0x8049780
…
```

Critical section requirements

- Safety (aka mutual exclusion): no more than one thread in critical section at a time.
- Liveness (aka progress):
  - If multiple threads simultaneously request to enter critical section, must allow one to proceed
  - Must not depend on threads outside critical section
- Bounded waiting (aka starvation-free)
  - Must eventually allow waiting thread to proceed
- Makes no assumptions about the speed and number of CPU
  - However, assumes each thread makes progress

Critical section desirable properties

- Efficient: don't consume too much resource while waiting
  - Don't busy wait (spin wait). Better to relinquish CPU and let other thread run
- Fair: don't make one thread wait longer than others. Hard to do efficiently
- Simple: should be easy to use

Implementing critical section using locks

- lock(l): acquire lock exclusively; wait if not available
- unlock(l): release exclusive access to lock

```
pthread_mutex_t   l = PTHREAD_MUTEX_INITIALIZER
```

```
void* deposit(void *arg)
{
        int i;
        for(i=0; i<1e7; ++i) {
                pthread_mutex_lock(&l);
                ++ balance;
                pthread_mutex_unlock(&l);
        }
}
```

```
void* withdraw(void *arg)
{
        int i;
        for(i=0; i<1e7; ++i) {
                pthread_mutex_lock(&l);
                -- balance;
                pthread_mutex_unlock(&l);
        }
}
```

Outline

- Critical section requirements
- Implementing locks
- Readers-writer lock

Implementing locks: version 1

- Can cheat on uniprocessor: implement locks by disabling and enabling interrupts

```
lock()                          unlock()
{                               {
    disable_interrupt();            enable_interrupt();
}                               }
```

- Good: simple!
- Bad:
  - Both operations are privileged, can't let user program use
  - Doesn't work on multiprocessors

Implementing locks: version 2

- Peterson's algorithm: software-based lock implementation
- Good: doesn't require much from hardware
- Only assumptions:
  - Loads and stores are atomic
  - They execute in order
  - Does not require special hardware instructions

Software-based lock: 1st attempt

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;

lock()
{
        while (flag == 1)
            ; // spin wait
        flag = 1;
}

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

- Idea: use one flag, test then set; if unavailable, spin-wait
- Problem?
  - Not safe: both threads can be in critical section
  - Not efficient: busy wait, particularly bad on uniprocessor (will solve this later)

Bug in software lock, 1st attempt

```
lock()
{
        1: while (flag == 1)
             ; // spin wait
        2: flag = 1;
}
```

```
unlock()
{
        3: flag = 0;
}
```

Software-based lock

- 2nd attempt: use per thread flags, set then test, to achieve mutual exclusion
  - Not live: can deadlock
- 3rd attempt: strict alternation to achieve mutual exclusion
  - Not live: depends on threads outside critical section
- Final attempt: combine above ideas
- Problem
  - It's hard!
  - N>2 threads? (Lamport's Bakery algorithm)
  - Modern out of order processors?

Implementing locks: version 3

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;
```

```
lock()
{
        while(test_and_set(&flag))
            ;
}
```

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

- Problem with the test-then-set approach: test and set are not atomic
- Fix: special atomic operation
  - ```
    int test_and_set (int *lock) {
        int old = *lock;
        *lock = 1;
        return old;
    }
    ```
  - Atomically returns *lock and sets *lock to 1
    ```

Implementing test_and_set on x86

```
long test_and_set(volatile long* lock)
{
        int old;
        asm("xchgl %0, %1"
            : "=r"(old), "+m"(*lock)  // output
            : "0"(1)                  // input
            : "memory"                // can clobber anything in memory
            );
        return old;
}
```

- xchg reg, addr: atomically swaps *addr and reg
- Most spin locks on x86 are implemented using this instruction
  - xv6 spinlock.h, spinlock.c, x86.h

Spin-wait or block?

- Problem: waste CPU cycles
  - Worst case: prev thread holding a busy-wait lock gets preempted, other threads try to acquire the same lock
- On uniprocessor: should not use spin-lock
  - Yield CPU when lock not available (need OS support)
- On multi-processor
  - Thread holding lock gets preempted ➔ ???
  - Correct action depends on how long before lock release
    - Lock released "quickly" ➔ ?
    - Lock released "slowly" ➔ ?

Problem with simple yield

```
lock()
{
        while(test_and_set(&flag))
            yield();
}
```

- Problem:
  - Still a lot of context switches: thundering herd
  - Starvation possible
- Why? No control over who gets the lock next
- Need explicit control over who gets the lock

Implementing locks: version 4

```
lock() {
  while (test_and_set(&flag))
     add myself to wait queue
     yield
  …
}
```

```
unlock() {
    flag = 0
    if(any thread in wait queue)
       wake up one wait thread
  …
}
```

Lock from another thread?

- The idea: add thread to queue when lock unavailable; in unlock(), wake up one thread in queue
- Problem I: lost wakeup
  - Fix: use a spin_lock or lock w/ simple yield!
  - Doesn't avoid spin-wait, but make wait time short
- Problem II: wrong thread gets lock
  - Fix: unlock() directly transfers lock to waiting thread

Lost wakeup

```
lock() {
  1: while (test_and_set(&flag)))
      2: add myself to wait queue
      3: yield
  …
}
```

```
unlock() {
    4: flag = 0
    5: if(any thread in wait queue)
       6: wake up one wait thread
  …
}
```

Wrong thread gets lock

```
lock() {
  1: while (test_and_set(&flag)))
      2: add myself to wait queue
      3: yield
  ...
}
```

```
unlock() {
    4: flag = 0
    5: if(any thread in wait queue)
       6: wake up one wait thread
  ...
}
```

Implementing locks: version 4, the code

```
typedef struct __mutex_t {
    int flag;        // 0: mutex is available,  1: mutex is not available
    int guard;       // guard lock to avoid losing wakeups
    queue_t *q;  // queue of waiting threads
} mutex_t;
```

```
void lock(mutex_t *m) {
    while (test_and_set(m->guard))
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // acquire mutex
        m->guard = 0;
    } else {
        enqueue(m->q, self);
        m->guard = 0;
        yield();
    }
}
```

```
void unlock(mutex_t *m) {
    while (test_and_set(m->guard))
        ;
    if (queue_empty(m->q))
        // release mutex; no one wants mutex
        m->flag = 0;
    else
        // direct transfer mutex to next thread
        wakeup(dequeue(m->q));
    m->guard = 0;
}
```

Outline

- Critical section requirements
- Implementing locks
- Readers-writer lock

Readers-Writers problem

- A reader is a thread that needs to look at the shared data but won't change it
- A writer is a thread that modifies the shared data
- Example: making an airline reservation
- Courtois et al 1971

Solving Readers-Writers w/ regular lock

```
lock_t lock;
```

Writer

```
lock (&lock);
. . .
// write shared data
. . .
unlock (&lock);
```

Reader

```
lock (&lock);
. . .
// read shared data
. . .
unlock (&lock);
```

- Problem: unnecessary synchronization
  - Only one writer can be active at a time
  - However, any number of readers can be active simultaneously!
- Solution: acquire lock for read mode and write mode

Readers-writer lock

```
rwlock_t lock;

Writer                        Reader

  write_lock (&lock);           read_lock (&lock);
  . . .                         . . .
  // write shared data          // read shared data
  . . .                         . . .
  write_unlock (&lock);         read_unlock (&lock);
```

- read_lock: acquires lock in read (shared) mode
  - If lock is not acquired or in read mode ➔ success
  - Otherwise, lock is in write mode ➔ wait
- write_lock: acquires lock in write (exclusive) mode
  - If lock is not acquire ➔ success
  - Otherwise ➔ wait

Implementing readers-writer lock

```
struct rwlock_t {
    int nreader;      // init to 0
    lock_t guard;     //  init to unlocked
    lock_t lock;      // init to unlocked
};

write_lock(rwlock_t *l)
{
    lock(&l->lock);
}

write_unlock(rwlock_t *l)
{
    unlock(&l->lock);
}
```

```
read_lock(rwlock_t *l)
{
    lock(&l->guard);
    ++ nreader;
    if(nreader == 1) // first reader
        lock(&l->lock);
    unlock(&l->guard);
}

read_unlock(rwlock_t *l)
{
    lock(&l->guard);
    -- nreader;
    if(nreader == 0) // last reader
       unlock(&l->lock);
    unlock(&l->guard);
}
```

Problem: may starve writer!

Backup slides

Software-based locks: 2nd attempt

```
// 1: a thread wants to enter critical section, 0: it doesn't
int flag[2] = {0, 0};

lock()
{
        flag[self] = 1; // I need lock
        while (flag[1- self] == 1)
            ; // spin wait
}

unlock()
{
        // not any more
        flag[self] = 0;
}
```

- Idea: use per thread flags, set then test, to achieve mutual exclusion
- Why doesn't work?
    - Not live: can deadlock

Software-based locks: 3rd attempt

```
// whose turn is it?
int turn = 0;

lock()
{
        // wait for my turn
        while (turn == 1 – self)
            ; // spin wait
}

unlock()
{
        // I'm done. your turn
        turn = 1 – self;
}
```

- Idea: strict alternation to achieve mutual exclusion
- Why doesn't work?
  - Not live: depends on threads outside critical section

Software-based locks: final attempt (Peterson's algorithm)

```
// whose turn is it?
int turn = 0;
// 1: a thread wants to enter critical section, 0: it doesn't
int flag[2] = {0, 0};
```

```
lock()
{
        flag[self] = 1; // I need lock
        turn = 1 – self;
        // wait for my turn
        while (flag[1-self] == 1
            && turn == 1 – self)
            ;  // spin wait while the
               // other thread has intent
               // AND it is the other
               // thread's turn
}
```

```
unlock()
{
        // not any more
        flag[self] = 0;
}
```

- Why works?
  - Safe?
  - Live?
  - Bounded wait?