Synchronization I

References: Operating Systems Concepts, Linux Kernel Development, previous W4118s

Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
Critical section

- **Critical section**: a segment of code that accesses a shared 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
...
Implementing critical section using locks

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

```c
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);
    }
}
```
• **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) for a long time. 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
Version 1: Disable interrupts

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

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

- **Good**: simple!

- **Bad**:  
  - Both operations are **privileged** -- can’t let user program use  
  - Doesn’t work on **multiprocessors**  
  - Can’t use for long critical sections
Version 2: Software Locks

• Peterson’s algorithm: software-based lock implementation (2 page paper with proof)

• 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
Unsafe software lock, 1\textsuperscript{st} attempt

In general, adversarial scheduler model useful to think about concurrency problems
Software-based locks: 2\textsuperscript{nd} 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
        ; // spin wait
}

unlock()
{
    flag[self] = 0;

    // not any more
}

// 1: a thread wants to enter critical section, 0: it doesn’t

• Idea: use per thread flags, set then test, to achieve mutual exclusion

• Why doesn’t work?
  – Not live: can deadlock
Deadlock: 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()
{
    flag[self] = 0;
}

// 1: a thread wants to enter critical section, 0: it doesn’t

Thread 0

call lock()
flag[0] = 1;

Thread1

flag[1] = 1;
while (flag[0] == 1) ;
//spins forever!
...

while (flag[1] == 1) ;
// spins forever too!
Software-based locks: 3\textsuperscript{rd} 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
  – Can’t handle repeated calls to lock by same thread
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?
Multiprocessor Challenges

• Modern processors are out-of-order/speculative
  – Reorder instructions to keep execution units full
  – Try very hard to avoid inconsistency
  – Guarantees valid only within single execution stream

• Memory access guarantees on x86
  – x86 is relatively conservative with reordering
  – Loads not reordered with other loads
  – Stores not reordered with other stores
  – Stores not reordered with older loads
  – All loads and stores to same location are not reordered
  – Load can reorder with older store to different addr

• Breaks Peterson’s algorithm!

Instruction Reordering affects Locking

- Possible for mutual exclusion to be violated?
  - Yes!

Thread 0

```c
Lock: flag[0] = 1; // I need lock
turn = 1;
while (flag[1]==1 && turn==1);
}
```

Thread 1

```c
Lock: flag[1] = 1; // I need lock
turn = 0;
while (flag[0]==1 && turn==0);
}
```

Reorder

```
Lock: r1 = Load(flag[1])

turn = 1;
flag[0] = 1; // I need lock
while (r1==1 && turn==1);
  // flag[1]==0
}
```
Memory Barriers

• A memory barrier or fence
  – Ensures that all memory operations up to the barrier are executed before proceeding

• x86 provides several memory fence instructions
  – Relatively expensive (100s of cycles)
  – mfence: all prior memory accesses completed
  – lfence: all prior loads completed
  – sfence: all prior stores flushed

```c
lock() {
    flag[self] = 1; // I need lock
    turn = 1 - self;
    sfence; // Store barrier
    while (flag[1-self] == 1 && turn == 1 - self);
}
```
Version 3: Hardware Instructions

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

lock()
{
    while(test_and_set(&flag))
        flag = 0;
}

unlock()
{
}

• 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
• Spin locks on x86 are implemented using this instruction
• x86 also provides a lock prefix that allows bus to be locked for inst
• In Linux:
  – Arch independent: kernel/spinlock.c
  – Arch dependent: arch/x86/include/asm/spinlock.h
Limitations of spin locks

• Spin lock is heavily used in Linux kernel
  – Kernel preemption disabled while spin lock is held

• Available in user space, but of limited use
  – `pthread_spin_init` man page says:
    Spin locks should be employed in conjunction with real-time scheduling policies (SCHED_FIFO, or possibly SCHED_RR). Use of spin locks with nondeterministic scheduling policies such as SCHED_OTHER probably indicates a design mistake. The problem is that if a thread operating under such a policy is scheduled off the CPU while it holds a spin lock, then other threads will waste time spinning on the lock until the lock holder is once more rescheduled and releases the lock.
What if we yield instead of spinning?

```c
lock()
{
    while (test_and_set(&flag)) {
        // give up CPU and let another thread run
        yield();
    }
}
```

- Many threads can cause a lot of context switches
  - 100 threads; lock holder preempted; 99 run and yield
  - Starvation still possible
- Need explicit control over who gets the lock
Version 4: Sleep Locks

- The idea: add thread to queue when lock unavailable; in unlock(), wake up one thread in queue

- Problem I: lost wakeup

- Problem II: 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

  ...
}

Thread 0:
call lock()
while (test_and_set(&flag)) {
  add myself to wait queue
  yield
} // wait forever (or until next unlock)!

Thread 1

call unlock()
flag = 0
if (any thread in wait queue) // No!
  wake_up_one_wait_thread
Wrong thread gets lock

```c
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
    ...
}
```

Thread 0:
call lock()
while (test_set(&flag))
    add myself to wait queue
    yield

Thread 1:
call unlock()
    flag = 0
    if (thread in wait queue)
        wake_up_thread

Thread 2:
call lock()
    while (test_set(&flag))

• Fix: `unlock()` directly transfers lock to waiting 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;
}
Fixing the last race condition

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);
        prepare_to_yield();
        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;
}
Reader-Writer 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
Readers-writer lock

```c
rwlock_t lock;

**Writer**

write_lock (&lock);
...
// write shared data
...
write_unlock (&lock);

**Reader**

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

- **read_lock**: acquires lock in read (shared) mode
  - Lock is not acquired or is acquired in read mode → success
  - Otherwise (lock is in write mode) → wait

- **write_lock**: acquires lock in write (exclusive) mode
  - Lock is not acquired → success
  - Otherwise → wait
Implementing readers-writer lock

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

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

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

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

Problem: may starve writer!
```
Driving out readers in a RW-Lock

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

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

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

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

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

Q: In write_lock, can we just use guard instead of writer lock?
```