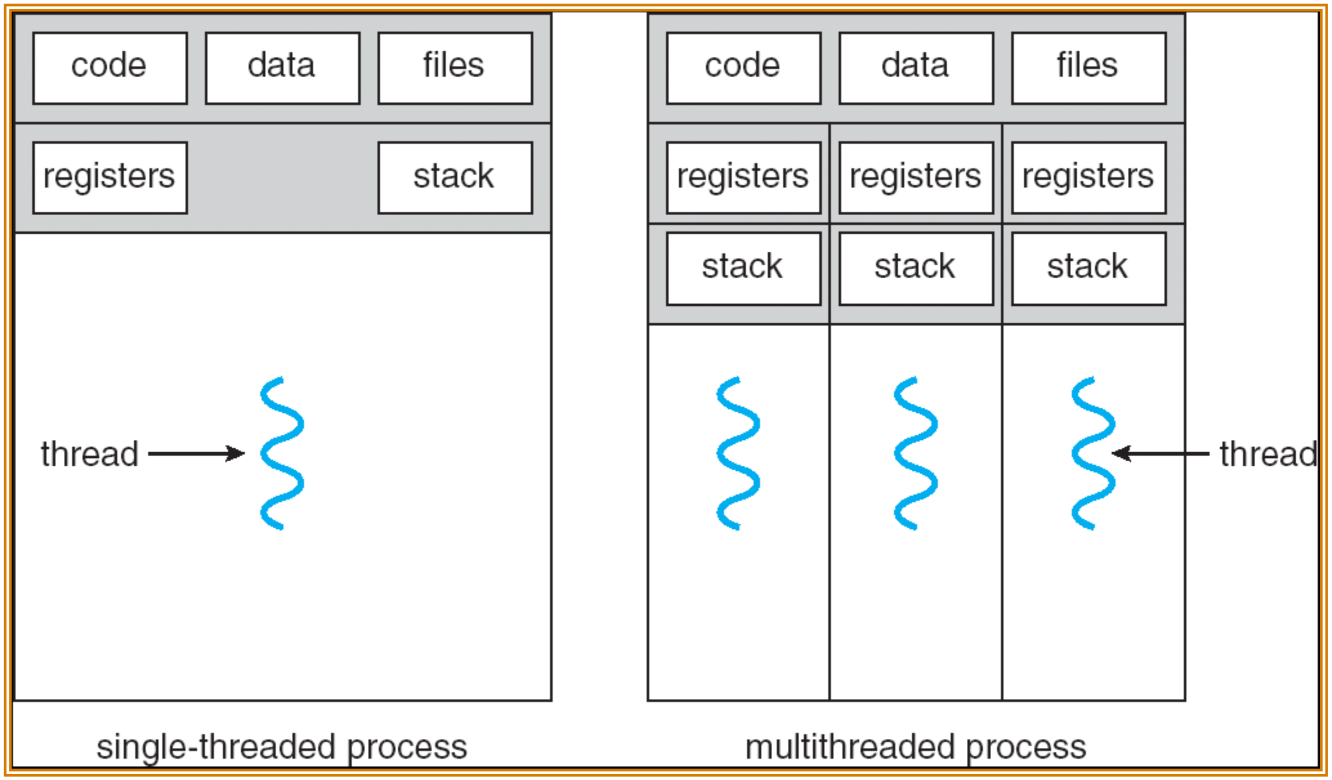
# Synchronization I

COMS W4118

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

# Single and multithreaded processes




More details on implementing threads later

# 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) 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

# 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

# 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
  - Cant 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

Reference: G. L. Peterson: "Myths About the Mutual Exclusion Problem", *Information Processing Letters* 12(3) 1981, 115–116

# 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)

# Unsafe software lock, 1st attempt

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

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

flag=0;

Thread 0:

call lock()
1: while (flag ==1) // it is 0, so continue

Thread 1:

call lock()
1: while(flag == 1) // it is 0, so continue

Thread 0:

2: flag = 1;

Thread 1:

2: flag = 1; // ! Thread 0 is already in critical section

In general, adversarial scheduler model useful to think about concurrency problems

# Software-based locks: 2nd attempt

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

lock()                                    unlock()
{                                         {
        flag[self] = 1; // I need lock            // not any more
        while (flag[1- self] == 1)                flag[self] = 0;
            ; // spin wait                }
}
```

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

| Thread 0 | Thread1 |
|---|---|
| call lock() | |
| flag[0] = 1; | |
| | flag[1] = 1; |
| | while (flag[0] == 1) ; |
| | //spins forever! |
| | ... |
| while (flag[1] == 1) ; | |
| // spins forever too! | |

# 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
  - 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?

# Software-based lock

- Problem
  - It's hard!
  - N>2 threads? (Lamport's Bakery algorithm)
  - Modern out of order processors?

# 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!

Reference: http://www.linuxjournal.com/article/8211

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

# Instruction Reordering affects Locking

Thread 0

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

Thread 1

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

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

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

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

Reorder

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

```
lock() {
        flag[self] = 1; // I need lock
        turn = 1 – self;
        sfence; // Store barrier
        while (flag[1-self] == 1 && turn == 1 – self);
}
```

# Lamport's Bakery Algorithm

- Support more than 2 processes
  - Integer tokens (increasing numbers)
  - Each customer gets next largest token
  - Same token? Smaller thread_id gets priority
  - Smallest token enters critical region

```
bool flag[1..NUM_THREADS] = {0};    // Want to enter
int token[1..NUM_THREADS] = {0};    // My token
lock(i) {    // Lock by thread i
   flag [i] = 1;
   token[i] = 1 + max(token[0..NUM_THREADS-1]);
   flag[i] = 0;
   for (j = 1; j <= NUM_THREADS; j++) {
      while (flag[j]); // Is j getting token?
      while ((token[j] && ((token[j], j) < (token[i], i))); // j has smaller token?
}
```

```
unlock(integer i) {
   token[i] = 0;
}
```

Reference: A New Solution of Dijkstra's Concurrent Programming Problem. L. Lamport. Communications of the ACM, 1974. http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

# Version 3: Hardware Instructions

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;
lock()                                  unlock()
{                                       {
    while(test_and_set(&flag))              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
- 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

# Spin-wait or block?

- Problem of spin-wait: waste CPU cycles
  - Worst case: 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

# Version 4: Sleep Locks

```
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
- Problem II: wrong thread gets lock

# 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
  …
}
```

| Thread 0: | Thread 1 |
|---|---|
| call lock() | |
| while (test_and_set(&flag)) { | |
| | call unlock() |
| | flag = 0 |
| | if (any thread in wait queue) // No! |
| | wake_up_one_wait_thread |
| add myself to wait queue | |
| yield | |
| } // wait forever (or until next unlock)! | |

# 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: | Thread 1 | Thread 2 |
|---|---|---|
| call lock() | | |
| while (test_set(&flag)) | | |
| add myself to wait queue | | |
| yield | call unlock() | |
| | flag = 0 | |
| | if (thread in wait queue) | |
| | wake_up_thread | 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;
}
```

# Adaptive Mutexes

- Cons of Spinlocks
  - Inefficient if lock is held for long duration
- Cons of Sleeplocks
  - Higher overhead, state maintenance
- Solaris, OS X, FreeBSD
  - Idea: use spinlock if holder is currently running, sleeplock otherwise
  - Best of both worlds

# 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

# Readers-writer lock

```
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

```
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!

# Driving out readers in a RW-Lock

```
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);
     unlock(&l->writer);
}

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

```
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?