Synchronization Module 6
## Implementing Synchronization

Concurrent Applications

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>Locks</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Disable</td>
<td>Atomic Read/Modify/Write Instructions</td>
<td></td>
</tr>
<tr>
<td>Multiple Processors</td>
<td>Hardware Interrupts</td>
<td></td>
</tr>
</tbody>
</table>
Synchronization Variable Interfaces

- (spin) lock
  - acquire() / release() [lock()/unlock()]

- (blocking) lock [mutex]
  - acquire() / release() [lock()/unlock()]

- Semaphore(int n)
  - P – if value <= 0 then wait; decrement value
  - V – increment value; if there is a waiter, wake one up

- Condition variable(lock)
  - wait() - suspend this thread and release lock
  - signal() - wake up one waiting thread, if there is one, and regain its lock
  - broadcast() - wake up all waiting threads, if any, and let them battle for lock
Question: Can this panic?

Thread 1

\[ p = \text{someComputation}(); \]
\[ \text{pInitialized} = \text{true}; \]

Thread 2

\[ \text{while (!pInitialized)} \]
\[ \quad ; \]
\[ q = \text{someFunction}(p); \]
\[ \text{if (}q \neq \text{someFunction}(p)\text{)} \]
\[ \quad \text{panic} \]
Will this code work?

```c
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```c
newP() {
    p = malloc(sizeof(p));
    p->field1 = ...
    p->field2 = ...
    return p;
}
```
Why Reordering?

• Why do compilers reorder instructions?
  – Efficient code generation requires analyzing control/data dependency
  – If variables can spontaneously change, most compiler optimizations become impossible

• Why do CPUs reorder instructions?
  – Write buffering: allow next instruction to execute while write is being completed

Fix: memory barrier
  – Instruction to compiler/CPU
  – All ops before barrier complete before barrier returns
  – No op after barrier starts until barrier returns
Implementing Synchronization

Take 1: using memory load/store
  – See too much milk solution/Peterson’s algorithm

Take 2:
  Lock::acquire()
    { disable interrupts }
  Lock::release()
    { enable interrupts }
void acquire(struct spinlock *lk) {
    pushcli(); // disable interrupts to avoid deadlock.
    if (holding(lk))
        panic("acquire");

    // The xchg is atomic.
    while (xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();

    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
Spinlock Implementation in xk

```c
void release(struct spinlock *lk) {
    if (!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;

    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m"(lk->locked) :);  

    popcli();
}
```
Multiprocessor

• Read-modify-write instructions
  – Atomically read a value from memory, operate on it, and then write it back to memory
  – Intervening instructions prevented in hardware

• Examples
  – Test and set
  – Intel: xchgb, lock prefix
  – Compare and swap

• Any of these can be used for implementing locks and condition variables!
Spinlocks

A **spinlock** is a lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect the CPU scheduler and to implement locks

Spinlock::acquire() {
    while (testAndSet(&lockValue) == BUSY) ;
}

Spinlock::release() {
    lockValue = FREE;
    memorybarrier();
}
How many spinlocks?

- Various data structures
  - Queue of waiting threads on lock X
  - Queue of waiting threads on lock Y
  - List of threads ready to run
- One spinlock per kernel?
  - Bottleneck!
- Instead:
  - One spinlock per blocking lock
  - One spinlock for the scheduler ready list
    - Per-core ready list: one spinlock per core
Mutex Implementation, Uniprocessor

Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}

Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
Lock Implementation, Multiprocessor

Lock::acquire() {
    disableInterrupts();
    spinLock.acquire();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value = BUSY;
    }
    spinLock.release();
    enableInterrupts();
}

Lock::release() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
What thread is currently running?

• Thread scheduler needs to find the TCB of the currently running thread
  – To suspend and switch to a new thread
  – To check if the current thread holds a lock before acquiring or releasing it

• On a uniprocessor, easy: just use a global

• On a multiprocessor, various methods:
  – Compiler dedicates a register (e.g., r31 points to TCB running on the this CPU; each CPU has its own r31)
  – If hardware has a special per-processor register, use it
  – Fixed-size stacks: put a pointer to the TCB at the bottom of its stack
    • Find it by masking the current stack pointer
Lock Implementation, Linux

• Most locks are free most of the time
  – Why?
  – Linux implementation takes advantage of this fact
• Fast path
  – If lock is FREE, and no one is waiting, two instructions to acquire the lock
  – If no one is waiting, two instructions to release the lock
• Slow path
  – If lock is BUSY or someone is waiting, use multiproc impl.

• User-level locks
  – Fast path: acquire lock using test&set
  – Slow path: system call to kernel, use kernel lock
Lock Implementation, Linux

```c
struct mutex {
    /* 1: unlocked; 0: locked;
        negative: locked,
        possible waiters */
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
};
```

```c
// atomic decrement
// %eax is pointer to count
lock decl (%eax)
jns 1 // jump if not signed
    // (if value is now 0)
call slowpath_acquire
1:
```
Semaphores

• Semaphore has a non-negative integer value
  – P() atomically waits for value to become > 0, then decrements
  – V() atomically increments value (waking up waiter if needed)

• Semaphores are like integers except:
  – Only operations are P and V
  – Operations are atomic
    • If value is 1, two P’s will result in value 0 and one waiter
Semaphore Bounded Buffer

get() {
    fullSlots.P();
    mutex.P();
    item = buf[front % MAX];
    front++;
    mutex.V();
    emptySlots.V();
    return item;
}

put(item) {
    emptySlots.P();
    mutex.P();
    buf[last % MAX] = item;
    last++;
    mutex.V();
    fullSlots.V();
}

Initially: front = last = 0; MAX is buffer capacity
mutex = 1; emptySlots = MAX; fullSlots = 0;
Communicating Sequential Processes (CSP/Google Go)

• A thread per shared object
  – Only thread allowed to touch object’s data
  – To call a method on the object, send thread a message with method name, arguments
  – Thread waits in a loop, get msg, do operation

• No memory races (in user code)!
Bounded Buffer (CSP)

while (cmd = getNext()) {
  if (cmd == GET) {
    if (front < tail) {
      // do get
      // send reply
      // if pending put, do it
      // and send reply
    } else
      // queue get operation
  } else {
    // cmd == PUT
    if ((tail - front) < MAX) {
      // do put
      // send reply
      // if pending get, do it
      // and send reply
    } else
      // queue put operation
  }
}
Locks/CVs vs. CSP

• Create a lock on shared data
  = create a single thread to operate on data
• Call a method on a shared object
  = send a message/wait for reply
• Wait for a condition
  = queue an operation that can’t be completed just yet
• Signal a condition
  = perform a queued operation, now enabled
“Rules” for Using Synchronization

• Use consistent structure
• Always use locks and condition variables
• Always acquire lock at beginning of procedure, release at end
• Always hold lock when using a condition variable
• Always wait in while loop
• Never spin in sleep()