Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes

OSPP Chapter 5 – Part B

(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

Roadmap

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

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 }

Lock 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();
}
```

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
  - Exchange (Intel: xchgb, w/ lock prefix to make atomic)
  - 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 lock
  - One spinlock for the scheduler ready list
    - Perhaps per-core ready lists: one spinlock per core

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();
}
```

Lock Implementation, Multiprocessor

```
Sched::suspend(SpinLock ∗lock) {
    TCB ∗next;

    disableInterrupts();
    schedSpinLock.acquire();
    lock−>release();
    myTCB−>state = WAITING;
    next = readyList.remove();
    thread_switch(myTCB, next);
    myTCB−>state = RUNNING;
    schedSpinLock.release();
    enableInterrupts();
}
```

```
Sched::makeReady(TCB
    ∗thread) {

    disableInterrupts ();
    schedSpinLock.acquire();
    readyList.add(thread);
    thread−>state = READY;
    schedSpinLock.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 multiprocessor impl.

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

Lock Implementation, Linux

struct mutex {

/∗ 1: unlocked ; 0: locked; negative : locked, possible waiters ∗/

atomic_t count;

spinlock_t wait_lock;

struct list_head wait_list;

};

// atomic decrement

// %eax is pointer to count

lock decl (%eax)

jns 1f // 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
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

Bounded Buffer with Locks/CVs

```
get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(&lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(&lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    while ((tail – front) ==
   MAX) {
        full.wait(&lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(&lock);
    lock.release();
}
```

tially: front = tail = 0; MAX is buffer capacity
CVs: empty and full

Bounded Buffer with Semaphores

```
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();
}
```

ally: front = last = 0; MAX is buffer capacity
maphores: mutex = 1; emptySlots = MAX; fullSlots

Compare Semaphore P()/V() Implementation with Locks/CVs

```
Semaphore::P() {
    disableInterrupts();
    spinLock.acquire();
    if (value == 0) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value--;
    }
    spinLock.release();
    enableInterrupts();
}
```

```
Semaphore::V() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler-
   >makeReady(next);
    } else {
      value++;
    }
    spinLock.release();
    enableInterrupts();
}
```

Remember the rules

- 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()

(if time permits)

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!

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

```
[ bounded_buffer || producer || consumer ]

    producer :: *[ <produce item> ;
    bounded_buffer ! item
    ]

    consumer ::   *[ bounded_buffer ? item;
    <consume item>
    ]

    bounded_buffer :: buffer: (0..9) item;
    count, in, out: integer;

    count := 0; in := 0; out := 0;

    *[ count < 10 & producer ? buffer(in)  ->
    in := (in + 1) mod 10;
    count := count + 1
    ||
       count > 0 & consumer ! buffer(out)  ->
    out := (out + 1) mod 10;
    count := count - 1
    ]
```

|| is concurrent execution

*[ ] is repetition

! is send

? is receive

-> marks a guarded statement

Implementing Condition Variables using Semaphores (Take 1)

```
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    semaphore.V();
}
```

Implementing Condition Variables using Semaphores (Take 2)

```
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    if (semaphore is not empty)
        semaphore.V();
}
```

Implementing Condition Variables using Semaphores (Take 3)

```
wait(lock) {
    semaphore = new Semaphore;
    queue.Append(semaphore);   // queue of waiting threads
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    if (!queue.Empty()) {
        semaphore = queue.Remove();
        semaphore.V();  // wake up waiter
    }
}
```