Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes

OSPP Chapter 5 – Part A

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

Lost Update / Record Out Problem

Thread A with shared “x”

```
x = x + 1;
```

// at machine code level

```
load r1, x
add r1, r1, #1
```

// switch threads ✉

```
store r1, x
```

Thread B with shared “x”

```
x = x + 1;
```

// at machine code level

```
load r1, x
add r1, r1, #1
store r1, x
```

// switch threads

Question: Are All Pushes Successful?

Consider shared linked list accessed as a stack: (adapted from Michael Scott)

```
struct node{
    ...
    struct node *next;
}

struct node *head;

void push( struct node *new ){
    new->next = head;
            // what if a second push starts at this point?
    head = new;
}
```

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior can change when you re-run program
- Also:
  - Compiler/hardware instruction reordering
  - Multi-word operations are not atomic

Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while ( !pInitialized )
  /* empty loop body */ ;
q = someFunction(p);
if ( q != someFunction(p) )
  panic();
```

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

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if ( !milk )
    if ( !note ) {
        leave note
        buy milk
        remove note
    }
```

Too Much Milk, Try #2

<u>Thread A</u>

```
leave note A
if ( !note B ) {
    if ( !milk )
        buy milk
}
remove note A
```

<u>Thread B</u>

```
leave note B
if ( !noteA ) {
    if ( !milk )
        buy milk
}
remove note B
```

Too Much Milk, Try #3

Thread A

```
leave note A
while ( note B ) // X
    do nothing;
if ( !milk )
    buy milk;
remove note A
```

Thread B

```
leave note B
if ( !noteA ) {   // Y
    if ( !milk )
        buy milk
}
remove note B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii)Other will buy, ok to quit

Lessons

- Solution is complicated
  - “obvious” code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson’s algorithm

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

Locks

- Lock::acquire()
  - wait until lock is free, then take it
- Lock::release()
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Question: Why only Acquire/Release

- Suppose we add a method to ask if a lock is free. Suppose it returns true. Is the lock:
  - Free?
  - Busy?
  - Don't know?

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
    buy milk
lock.release();
```

Lock Example: Malloc/Free

```
char *malloc (n) {
  heaplock.acquire();
    p = allocate memory
  heaplock.release();
    return p;
}
```

```
void free(char *p) {
  heaplock.acquire();
    put p back on free list
  heaplock.release();
}
```

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Example: Bounded Buffer

```
tryget() {
    item = NULL;
    lock.acquire();
    if ( front < tail ) {
        item = buf[front %
   MAX];
        front++; // ignoring ovf
    }
    lock.release();
    return item;
}
```

```
tryput(item) {
    lock.acquire();
    if ( (tail – front) <
   size ) {
        buf[tail % MAX] =
   item;
        tail++;    // ignoring
   ovf
    }
    lock.release();
}
```

ally: front = tail = 0; lock = FREE; MAX is buffer capa

Example: Bounded Buffer

- For simplicity, assume no wraparound on the integers front and tail; I'll assume you can fix that if you want
  - front = total number of items that have ever been removed
  - tail = total number of items ever inserted

- Lock at beginning of procedure; unlock at end; no access outside of locks

- Note that we don't know whether the buffer is still empty once we release the lock– we only know the state of the buffer while holding the lock!

Question

- If tryget() returns NULL, do we know the buffer is empty?

- If we poll tryget() in a loop, what happens to a thread calling tryput()?

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Condition Variable Design Pattern

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state

    while ( ! testSharedState() ) {
        cv.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state

    // If testSharedState is now true
    cv.signal(&lock);

    // Read/write shared state
    lock.release();
}
```

Example: Bounded Buffer

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

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

tially: front = tail = 0; MAX is buffer capacity
npty/full are condition variables

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - front + MAX >= tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

Pre/Post Conditions

```
methodThatWaits() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

    while ( !testSharedState() ) {
        cv.wait(&lock);
     }
    // WARNING: shared state may
    // have changed!  But
   // testSharedState is TRUE
   // and pre-condition is true

   // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent

    // Read/write shared state

    // If testSharedState is now true
    cv.signal(&lock);

    // NO WARNING: signal keeps lock

    // Read/write shared state
    lock.release();
}
```

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast puts thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while ( needToWait() ) {
    condition.Wait(lock);
}
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) { condition.Wait(&lock); }
  - Do not assume when you wake up that signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

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)

Pthread Syntax

Thread A

pthread_mutex_lock(&my_lock);

while(/* test of shared state fails */){

pthread_cond_wait(&my_cond_var,
        &my_lock);
}

/* read/write shared state */

pthread_mutex_unlock(&my_lock);

Thread B

pthread_mutex_lock(&my_lock);

/* read/write shared state */

/* if state has changed in a way that
     allows other threads to make
     progress, then signal or broadcast */

pthread_cond_signal(&my_cond_var);

pthread_mutex_unlock(&my_lock);

Blocking Bounded Buffer in Java

```
class BoundedBuffer {
    Lock lock = new ReentrantLock();
    Condition notFull  = lock.newCondition();
    Condition notEmpty = lock.newCondition();
    Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x)throws IE {
      lock.lock(); try {
        while ( count == items.length )
            notFull.await();
        items[putptr] = x;
        if ( ++putptr == items.length )
            putptr = 0;
        ++count;
        notEmpty.signal();
      } finally { lock.unlock(); }
    }
```

```
public Object take() throws IE {
      lock.lock(); try {
        while ( count == 0 )
            notEmpty.await();
        Object x = items[takeptr];
        if ( ++takeptr == items.length )
            takeptr = 0;
        --count;
        notFull.signal();
        return x;
      } finally { lock.unlock(); }
    }
  }
```

(code example from gee.cs.oswego.edu/dl/concurrency-interest/jsr166-slides.pdf)

Pthreads and Java 5 (and later)

When waiting upon a Condition Variable, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition Variable should always be waited upon in a loop, testing the state predicate that is being waited for.

Mesa vs. Hoare semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

FIFO Bounded Buffer (Hoare semantics)

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

```
put(item) {
    lock.acquire();
    if ( (tail – front) == MAX ) {
        full.wait(&lock);
    }
    buf[last % MAX] = item;
    last++;   // ignoring ovf
    empty.signal(&lock);
  // CAREFUL: someone else ran
    lock.release();
}
```

tially: front = tail = 0; MAX is buffer capacity
npty/full are condition variables

FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, ...

FIFO Bounded Buffer (Mesa semantics, put() is similar)

```
get() {
    lock.acquire();
    myPosition = numGets+
   +;
   // ignoring ovf
    self = new Condition;
    nextGet.append(self);
    while ( front < myPosition
   || front == tail ) {
       self.wait(&lock);
     }
    delete self;
    item = buf[front % MAX];
    front++; // ignoring ovf
    if (next =
   nextPut.remove()) {
       next->signal(&lock);
    }
    lock.release();
    return item;
}
```

ally: front = tail = numGets = 0; MAX is buffer capa

tGet, nextPut are queues of Condition Variables