CS510 Concurrent Systems

Jonathan Walpole

Portland State
UNIVERSITY

Introduction to Concurrency

Why Study Concurrency?

We are well into the era of concurrent hardware

- Moore's law still holds (more or less)
- processor cycles per sec is not increasing
- cores per processor is increasing
- hardware trending from multicore to manycore

What does this mean for software?

Software Implications

Software must be concurrent!

Concurrency has been taught for at least 40 years

- Isn’t it a solved problem?
- Which problems have been solved?
- Do these solutions solve our current problem?

What is the Current Problem?

Challenge 1: how to write software whose performance improves as core counts increase

Challenge 2: how to reason about the correctness of such software

Challenge 3: how to ensure that such software is portable across different hardware platforms

- in terms of its correctness and its performance scalability characteristics!

Program Correctness

How do we reason about program correctness?

- for sequential programs

Why are concurrent programs any different?

Sequential Program

Process 1

print “1”
print “2”

What output do you expect?
Why?

Concurrent Program

Thread 1	Thread 2
print “1”	print “2”

What output do you expect?
Why?

Non-Determinism

The output depends on external factors

- relative execution speed
- cache hit rates
- interrupts
- preemptions, scheduling order, etc

All are outside the control of the programmer

Concurrent Writes

Thread 1	Thread 2
x = 1 print x	x = 2

What output do you expect?

What will be the final value of x?

Why?

Non-Determinism

But this time it affects memory values

... which influence the behavior of programs that read and use them

Concurrent Updating

Thread 1	Thread 2
x = x + 1	x = x + 1
print x	

What output do you expect (x initialized to 0)?

What will be the final value of x?

Why?

Concurrent Updating

Thread 1	Thread 2
t1_temp = x	t2_temp = x
x = t1_temp + 1	x = t2_temp + 1
print x	

What output do you expect (x initialized to 0)?
What will be the final value of x?
Why?

An Alternative Implementation

Maybe x = x + 1 is implemented as:

load x to register
increment register
store register value to x

x is a global variable, ie. a shared memory location

Registers are part of each thread's *private* CPU context

An Alternative Implementation

Thread 1

load x to t1register
increment t1register
store t1register to x

Thread 2

load x to t2register
increment t2register
store t2register to x

Memory Accesses

Thread 1

read x
write x

Thread 2

read x
write x

In terms of memory accesses to the shared variable, both implementations are the same!

Memory Invariance Property

A process executing sequential code can assume that memory values only change as a result of its writes!

A thread executing concurrent code can not assume this unless it is enforced somehow!

Increment Instruction?

Would it help if x = x + 1 is implemented as an increment instruction that operates directly on x?

- an increment instruction on x must involve a memory read of x followed by memory write to x
- the reads in thread 1 and thread 2 may occur before either thread writes

How can we prevent this?
How can we make the increment atomic?

Race Conditions

The basic problem is called a *race condition* or a *data race*

Race conditions occur with

- concurrent accesses to the same memory location
- at least one of the accesses is a write

How can we prevent race conditions?

Synchronization

Two types of synchronization:

- Serialization
  - A must happen before B
- Mutual Exclusion
  - A and B must not happen at the same time

We could use mutual exclusion to prevent data races, if A and B are the critical sections of code that must not execute concurrently

Mutual Exclusion

How can we implement it?

Locks – the basic idea

Each shared data has a unique lock associated with it

Threads acquire the lock before accessing the data

Threads release the lock after they are finished with the data

The lock can only be held by one thread at a time

Locks - Implementation

How can we implement a lock?

How do we test to see if its held?

How do we lock it?

How do we unlock it?

What do we do if it is already held when we test?

Does this work?

```
bool lock = false

while lock = true;              /* repeatedly poll */
lock = true;                    /* lock */
        critical section
lock = false;                   /* unlock */
```

Reads, Writes, Memory Invariance

```
bool lock = false

while lock = true;              /* repeatedly poll */
lock = true;                    /* lock */
        critical section
lock = false;                   /* unlock */
```

Atomicity

Lock and unlock operations must be atomic

Modern hardware provides a few simple atomic instructions that can be used to build atomic lock and unlock primitives.

Atomic Instructions

Atomic "test and set" (TSL)

Compare and swap (CAS)

Load-linked, store conditional (ll/sc)

Atomic Test and Set

TSL performs the following in a single atomic step:

- set lock and return its previous value

Using TSL in a lock operation

- if the return value is false then you got the lock
- if the return value is true then you did not
- either way, the lock value is set!

TSL is a read and a write!

Spin Locks

```
while (TSL (lock)= true); /* poll while waiting */
        critical section  /* lock value is now true */
lock = false              /* release the lock */
```

Spin Locks

What price do we pay for mutual exclusion?

How well will this work on uniprocessor?

Blocking Locks

How can we avoid wasting CPU cycles?

Can we sleep instead of polling?

How can we implement sleep and wakeup?

- join waiting list and context switch when lock is held
- wakeup next thread on lock release
- need explicit calls to acquire and release lock, can't just set lock value in memory

But how can we make these system calls atomic?

Blocking Locks

Is this better than a spinlock on a uniprocessor?

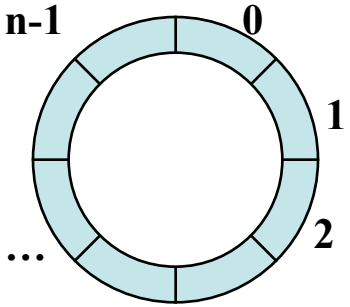
Is this better than a spinlock on a multiprocessor?

When would you use a spinlock vs a blocking lock on a multiprocessor?

Tricky Issues With Locks

```
0   thread producer {
1     while(1) {
2       // Produce char c
3       if (count==n) {
4         sleep(full)
5       }
6       buf[InP] = c;
7       InP = InP + 1 mod n
8       count++
9       if (count == 1)
10        wakeup(empty)
11      }
12    }
```

```
0   thread consumer {
1     while(1) {
2       if(count==0) {
3         sleep(empty)
4       }
5       c = buf[OutP]
6       OutP = OutP + 1 mod n
7       count--;
8       if (count == n-1)
9         wakeup(full)
10      // Consume char
11    }
12  }
```




```
Global variables:
     char buf[n]
     int InP = 0    // place to add
     int OutP = 0   // place to get
     int count
```

Conditional Waiting

Sleeping while holding the lock leads to deadlock

Releasing the lock then sleeping opens up a window for a race

Need to atomically release the lock and sleep

Semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

```
Down (S)
    S.val = S.val - 1
    If S.val < 0
        add calling thread to S.list;
        sleep;

Up (S)
    S.val = S.val + 1
    If S.val <= 0
        remove a thread T from S.list;
        wakeup (T);
```

Semaphores

Down and up are assumed to be atomic

How can we implement them?

- on a uniprocessor?
- on a multiprocessor?

Semaphores in Producer-Consumer

```
Global variables
   semaphore full_buffs = 0;
   semaphore empty_buffs = n;
   char buff[n];
   int InP, OutP;
```

```
0 thread producer {
1   while(1){
2     // Produce char c...
3     down(empty_buffs)
4     buf[InP] = c
5     InP = InP + 1 mod n
6     up(full_buffs)
7   }
8 }
```

```
0 thread consumer {
1   while(1){
2     down(full_buffs)
3     c = buf[OutP]
4     OutP = OutP + 1 mod n
5     up(empty_buffs)
6     // Consume char...
7   }
8 }
```

Monitors and Condition Variables

Correct synchronization is tricky

What synchronization rules can we automatically enforce?

- encapsulation and mutual exclusion
- conditional waiting

Condition Variables

Condition variables (cv) for use within monitors

cv.wait(mon-mutex)

- thread blocked (queued) until condition holds
- Must not block while holding mutex!
- Monitor's mutex must be released!
- Monitor mutex need not be specified by programmer if compiler is enforcing mutual exclusion

cv.signal()

- signals the condition and unblocks (dequeues) a thread

Condition Variables –Semantics

What can I assume about the state of the shared data?

- when I wake up from a wait?
- when I issue a signal?

Hoare Semantics

Signaling thread hands monitor mutex directly to signaled thread

Signaled thread can assume condition tested by signaling thread holds

Mesa Semantics

Signaled thread eventually wakes up, but signaling thread and other threads may have run in the meantime

Signaled thread can not assume condition tested by signaling thread holds

- signals are a hint

Broadcast signal makes sense with MESA semantics, but not Hoare semantics

Memory Invariance

A thread executing a sequential program can assume that memory only changes as a result of the program statements

- can reason about correctness based on pre and post conditions and program logic

A thread executing a concurrent program must take into account the points at which memory invariance may be lost

- what points are those?

Reasoning About Locks

Memory invariance holds for a variable if the thread holds the lock that protects it

It is lost when the lock is released!

Subsequent use of the variable requires both acquiring the lock and re-reading the variable!

Reasoning About Monitors

Points at which memory invariance is lost:

- unlock monitor lock
- wait on condition variable
- signal condition variable (if it has Hoare semantics)

Subsequent use of monitor data after these points requires data be re-read!!!!

Homework!

Read class website and follow instructions

Start programming assignment 1