CS510 Concurrent Systems

Jonathan Walpole
Introduction to Concurrency
Why is Concurrency Important?

Scaling up vs scaling out
Concurrent hardware vs concurrent software
Sequential Programming

Sequential programming with processes

- Private memory
  - a program
  - data
  - stack
  - heap

- CPU context
  - program counter
  - stack pointer
  - registers
Sequential Programming Example

```c
int i = 0
i = i + 1
print i
```

What output do you expect? Why?
Concurrent Programming

Concurrent programming with threads

- Shared memory
  - a program
  - data
  - heap
- Private stack for each thread
- Private CPU context for each thread
  - program counter
  - stack pointer
  - registers
Concurrent Threads Example

\[ \text{int } i = 0 \]

\[ \text{Thread 1: } i = i + 1 \]
\[ \text{print } i \]

What output do you expect with 1 thread?
Why?
Concurrent Threads Example

int i = 0

Thread 1:  i = i + 1  
            print i

Thread 2:  i = i + 1  
            print i

What output do you expect with 2 threads? Why?
Race Conditions

How is $i = i + 1$ implemented?

- load $i$ to register
- increment register
- store register value to $i$

Registers are part of each thread’s *private* CPU context
Race Conditions

Thread 1

load i to regn
inc regn
store regn to i

Thread 2

load i to regn
inc regn
store regn to i
Critical Sections

What is the danger in the previous example?
How can we make it safe?
What price do we pay?
Mutual Exclusion

Process A

- A enters critical region
- A leaves critical region

Process B

- T1
- T2
- B attempts to enter critical region
- B blocked
- T3
- B enters critical region
- T4
- B leaves critical region

Time
Mutual Exclusion

How can we implement it?
Locks

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 it's held?
How do we acquire it?
How do we release it?
How do we block/wait if it is already held when we test?
Does this work?

```plaintext
bool lock = false

while lock = true;  /* wait */
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 is set
Spin Locks

while TSL (lock); /* spin while return value is true */
critical section
lock = false
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?
How can we implement sleep and wakeup?
- context switch when acquire finds the lock held
- check and potential wakeup on lock release
- system calls to acquire and release lock

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 {
    while(1){
        // Produce char c...
        down(empty_buffs)
        buf[InP] = c
        InP = InP + 1 mod n
        up(full_buffs)
    }
}

0 thread consumer {
    while(1){
        down(full_buffs)
        c = buf[OutP]
        OutP = OutP + 1 mod n
        up(empty_buffs)
        // Consume char...
    }
}
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 invariants may be broken
- what points are those?