CS510 Concurrent Systems

Introduction to Concurrency

DRAFT IN PROGRESS
Sequential Programming with Processes

- What is a Process?
  - Private memory
    - a program
    - data
    - stack
    - heap
  - CPU context
    - program counter
    - stack pointer
    - registers
Sequential programming example:

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

What output do you expect?
Why?
Concurrent Programming with Threads

- What is a Thread?
  - Shared memory
    - a program
    - data
    - heap
  - Private stack
  - Private CPU context
    - program counter
    - stack pointer
    - registers
Concurrent programming example:

```plaintext
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 regx
  - inc regx
  - store regx i

- Thread 2
  - load i regx
  - inc regx
  - store regx i
Critical Sections

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

Time

A enters critical region

A leaves critical region

B attempts to enter critical region

B enters critical region

B leaves critical region

B blocked

Process A

Process B

T₁ T₂ T₃ T₄
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 its 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?

```cpp
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); /* while return value is true, spin */
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

Global variables:

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

thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1) // Consume char
      wakeup(empty)
  }
}

thread consumer {
  while(1) {
    // Consume char
    if (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1) // Consume full
      wakeup(full)
  }
}
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?
Producer-Consumer with Semaphores

Global variables

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

```cpp
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 }
```

```cpp
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 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?
Condition Variables - Hoare Semantics

- Signaling thread hands monitor mutex directly to signaled thread
  - signaled thread can assume condition tested by signaling thread holds
Condition Variables - 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

- A thread executing a concurrent program must take into account the points at which memory invariants may be broken
  - what points are those?