Concurrent Programming
&
Synchronization Primitives
Concurrent Threads Example

```java
int i = 0

Thread 1: i = i + 1    Thread 2: i = i + 1
    print i              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
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?
Critical Sections

Why is the previous example difficult to reason about?

How should we reason about this kind of code?

What property did we have in the sequential version, that was lost in the concurrent version?

Why was that property important?
Memory Invariance

Sequential programs have the property that memory values do not change unless the control flow changes them. Hence, we can easily reason about the effects of a control flow. How can we regain this memory invariance property in concurrent programs?
Mutual Exclusion

Thread A

- A enters critical region
- A leaves critical region

Thread B

- B attempts to enter critical region
- B blocked
- B enters critical region
- B leaves critical region

Time

T_1, T_2, T_3, T_4
Terminology

Race condition:
- Multiple accesses to the same variable, from different threads, with at least one write
- Result depends on the execution order of the threads
- Also called a data race

Critical section:
- A section of code that contains a race condition
Preventing Race Conditions

Enforce mutual exclusion on critical section code:
- make sure only one thread can execute it at a time
- how can threads coordinate with each other?

Would this give us memory invariance?
- for the critical section?
- for the whole program?
- we still have to reason differently!

How can implement mutual exclusion?
Locks

The basic idea:
- Every global variable has a unique lock associated with it
-Threads acquire/lock the lock before access
-If the lock is already held, the thread waits
-Threads release/unlock the lock after access

A lock is an abstract data type with two operations:
- lock() ... if locked then wait, else lock it and continue
- unlock() ... if a thread is waiting, give it the lock, else free the lock
Implementing a Lock

If the lock was just a binary variable, how would we implement the lock and unlock calls?

How can two thread’s concurrently read the lock’s value and then set it without having a race condition on the lock itself?

Can we protect the lock with another lock?
Implementing a Lock

Lock and Unlock operations must be made *atomic*!

Implementation options on a uniprocessor:
- OS kernel code can disable interrupts
- User code cannot disable interrupts, nor run with interrupts disabled!
- User code must access lock and unlock via system calls
Implementing a Lock

Implementation options on a multiprocessor:
- disabling interrupts does not prevent concurrent accesses from other CPUs!
- Hardware support for atomic test and set instructions is required
Atomic Test and Set

TSL *instruction* 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

An atomic TSL instruction can be used to implement a spin lock:

while (TSL (lock) == set); /* spin until return value is not set */

critical section

lock = free
Spin Locks

while (TSL (lock) == set); /* spin until return value is not set */
critical section
lock = free

Lock()

Unlock()
Spin Locks

What price do we pay for mutual exclusion?

How well will this work on uniprocessor?
Blocking Locks

Can we sleep instead of busy waiting?

Is this better to block rather than spin-wait on a uniprocessor?

Is this better to block rather than spin-wait on a multiprocessor?

When would you use a spinlock vs a blocking lock on a multiprocessor?
Implementing Sleep/Wakeup

If sleep and wakeup are not atomic:
- a waiting thread might decide to sleep
- a context switch may occur
- the lock holding thread might release the lock and wakeup the waiting thread before its sleep
- the waiting thread might miss its wakeup and sleep forever!
Implementing Sleep/Wakeup

If we make sleep and wakeup atomic by implementing them as system calls and by having interrupts disabled:
- When do we re-enable interrupts?
- Before sleeping would introduce a race condition
- After sleeping ...
  - what would that mean?
- who would enable interrupts?
Blitz Code Walk-through

Sleep() function defined on threads
- blocks *this* thread
- runs the *next* thread from the ready list

Run(next) function that takes a thread parameter
- calls switch() with *this* thread and next as parameters

Switch(this, next)
- saves the CPU context of this thread
- restores the CPU context of next thread
How Are Interrupts Enabled?

Interrupts are enabled by the thread that we switch to!

We must guarantee that all possible entry points to switch disable interrupts on the way in and enable them on the return path.

Can you verify this in Blitz?
- what about the code you are adding?
Thoughts on Assignment 2

Disable and enable interrupts appropriately
Make use of existing Blitz sleep call
To wake a thread add it to the ready list
- the scheduler will run it when its turn comes
Study the existing semaphore code to see how to do all this in Blitz
Adding Lock and Unlock Calls

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
Locks Are Not Always Enough

Sleeping while holding the lock leads to deadlock
Releasing the lock then sleeping opens up a window for a race
How can we safely keep count and decide whether to sleep or not?
How can we decide and sleep atomically?
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;
    add it to the ready list;     /* wake it up */
Semaphores

Down and up are assumed to be atomic
How can we implement them?
- on a uniprocessor?
- on a multiprocessor?
Making Down and Up Atomic

Uniprocessor:
- make them system calls
- disable interrupts when they are called and re-enable interrupts before they return

Multiprocessor:
- use TSL to build a spinlock
- use the spinlock to enforce mutual exclusion
- acquire the spinlock on entry to each call and release it prior to return from each call
- make sure all paths in and out have appropriately paired lock acquisitions and releases!
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...
    }
}
Is This Solution Safe?

Does it have a race condition?
If so, how should we fix it?

In assignment 2 you have multiple producer threads and multiple consumer threads!
  - use your mutex locks to fix this!
  - how many locks do you need?
  - what about print statements?
The Blitz Semaphore Code
Spare Slides
Disabling Interrupts in the Kernel

Scenario 1: A thread is running; wants to access shared data

- Disable interrupts
- Access shared data ("critical section")
- Enable interrupts
Disabling Interrupts in the Kernel

Problem:

Interrupts are already disabled and a thread wants to access the critical section
...using the above sequence...

ie. One critical section gets nested inside another
Disabling Interrupts in the Kernel

Problem: Interrupts are already disabled. Thread wants to access critical section using the previous sequence...

Save previous interrupt status (enabled/disabled)
Disable interrupts
Access shared data (“critical section”)
Restore interrupt status to what it was before
Doesn’t Work on Multiprocessors

Disabling interrupts during critical sections
- Ensures that interrupt handling code will not run
- But what if there are multiple CPUs?
- A thread on a different CPU might make a system call which invokes code that manipulates the ready queue
- Disabling interrupts on one CPU didn’t prevent this!

Solution: use a mutex lock (based on TSL)
- Ensures mutual exclusion for all code _that uses it_
Some Other Tricky Issues

The interrupt handling code that saves interrupted state is a critical section

- It could be executed concurrently if multiple almost simultaneous interrupts happen
- Interrupts must be disabled during this (short) time period to ensure critical state is not lost

What if this interrupt handling code attempts to lock a mutex that is held?

- What happens if we sleep with interrupts disabled?
- What happens if we busy wait (spin) with interrupts disabled?