Semaphores and Classical Synchronization Problems
The Producer-Consumer Problem

An example of the **pipeline parallelism model**
   - Producer thread puts items into a **bounded buffer**
   - Consumer thread removes them

The buffer is a shared resource
   - Code that manipulates it is a **critical section**

Producer thread waits if the buffer is full
Consumer thread waits if the buffer is empty
Is This Solution Correct?

```
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
This Code Has a Race Condition!

The “count” variable can be corrupted:
- Increments or decrements may be lost!
- Possible Consequences:
  - Both threads may spin forever
  - Buffer contents may be over-written

Code that manipulates count must be treated as a *critical section* and protected using a *lock*
Where Should Lock and Unlock Go?

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 condition
- because the decision to sleep is based on an assumption about the value of count, which may not be valid!

How can we safely keep count and decide whether to sleep or not?

How can we decide whether to sleep, and go to sleep, atomically?
A New Primitive: The Semaphore

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 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?
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!
Semaphore.down in Blitz

method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait' operation")
    EndIf
    count = count - 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
Semaphore.down in Blitz

method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait' operation")
    EndIf
    count = count - 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
Semaphore.down in Blitz

method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait' operation")
    EndIf
    count = count - 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
Semaphore.down in Blitz

method Down()
    var oldIntStat: int
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x80000000
        FatalError ("Semaphore count underflowed during 'Wait' operation")
    EndIf
    count = count - 1
    if count < 0 waitingThreads.AddToEnd (currentThread)
        currentThread.Sleep ()
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
What is `currentThread.Sleep()`?

If `sleep` stops a thread from executing, how, where, and when does it return?
- which thread enables interrupts following `sleep`?
- the thread that called `sleep` shouldn’t return until another thread has called `up`!
... but how does that other thread get to run?
... where exactly does the `thread switch` occur?

Trace down through the Blitz code until you find a call to `switch()`
- Switch is called in one thread but returns in another!
- See where registers are saved and restored
Study The Blitz Code

Thread.c
  Thread.Sleep()
  Run (nextThread)

Switch.s
  Switch (prevThread, nextThread)
method Up ()
    var oldIntStat: int
    t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during 'Signal' operation")
    endIf
    count = count + 1
    if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
Blitz Code For Semaphore.up

Method Up ()
    var oldIntStat: int
    t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during 'Signal' operation")
    endIf
    count = count + 1
    if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
Blitz Code For Semaphore.up

Method Up ()
    var oldIntStat: int
    t: ptr to Thread
    oldIntStat = SetInterruptsTo (DISABLED)
    if count == 0x7fffffff
        FatalError ("Semaphore count overflowed during 'Signal' operation")
    endIf
    count = count + 1
    if count <= 0
        t = waitingThreads.Remove ()
        t.status = READY
        readyList.AddToEnd (t)
    endIf
    oldIntStat = SetInterruptsTo (oldIntStat)
endMethod
Blitz Code For Semaphore.up

Method Up()
    var oldIntStat: int
    t: ptr to Thread
    oldIntStat = SetInterruptsTo(DISABLED)
    if count == 0x7fffffff
        FatalError("Semaphore count overflowed during 'Signal' operation")
    endIf
    count = count + 1
    if count <= 0
        t = waitingThreads.Remove()
        t.status = READY
        readyList.AddToEnd(t)
    endIf
    oldIntStat = SetInterruptsTo(oldIntStat)
endMethod
More Thoughts on Assignment 2

Disable and enable interrupts appropriately to ensure mutual exclusion in lock/unlock
Make use of existing Blitz sleep call and list operations
To wake a thread add it to the ready list
  - the scheduler will run it when its turn comes
Study the existing Blitz semaphore code to see how to do all this!
Using Atomic Instructions

Implementing semaphores with interrupt disabling only works on uni-processors
- What should we do on a multiprocessor?

Special (hardware) atomic instructions for synchronization
- test and set lock (TSL)
- compare and swap (CAS)

Semaphore can be built indirectly using atomic instructions
1. build spin locks from atomic instructions
2. protect semaphore code using spin locks
3. make sure the lock is always held on entry to sleep and always released on exit
Classical Synchronization Problems

Producer Consumer (bounded buffer)

Dining philosophers

Sleeping barber

Readers and writers
Producer Consumer Problem

Also known as the bounded buffer problem

Producer and consumer are separate threads
Is This a Valid Solution?

thread producer {
    while(1){
        // Produce char c
        while (count==n) {
            no_op
        }
        buf[InP] = c
        InP = InP + 1 mod n
        count++
    }
}

thread consumer {
    while(1){
        while (count==0) {
            no_op
        }
        c = buf[OutP]
        OutP = OutP + 1 mod n
        count--
        // Consume char
    }
}

Global variables:
    char buf[n]
    int InP = 0   // place to add
    int OutP = 0  // place to get
    int count
Does This Solution Work?

Global variables

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

```c
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 }
```

```c
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 }
```
Producer Consumer Problem

What is the shared state in the last solution?
Does it apply mutual exclusion? If so, how?

Producer and consumer are separate threads
Problems With Solution

What if we have multiple producers and multiple consumers?
- Producer-specific and consumer-specific data becomes shared
- We need to define and protect critical sections

*You’ll do this in the second part of assignment 2, using the mutex locks you built!*
Dining Philosophers Problem

Five philosophers sit at a table
One chopstick between each philosopher
(need two to eat)

while(TRUE) {
    Think();
    Grab first chopstick;
    Grab second chopstick;
    Eat();
    Put down first chopstick;
    Put down second chopstick;
}

Why do they need to synchronize? How should they do it?
Is This a Valid Solution?

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_chopstick(i);
        take_chopstick((i+1)% N);
        Eat();
        put_chopstick(i);
        put_chopstick((i+1)% N);
    }
}
```
Problems

Potential for deadlock!
Working Towards a Solution

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_chopstick(i);
        take_chopstick((i+1)% N);
        Eat();
        put_chopstick(i);
        put_chopstick((i+1)% N);
    }
}
```

`take_chopsticks(i)`

`put_chopsticks(i)`
Working Towards a Solution

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_chopsticks(i);
        Eat();
        put_chopsticks(i);
    }
}
```
Taking Chopsticks

int state[N]
semaphore mutex = 1
semaphore sem[i]

take_chopsticks(int i) {
  down(mutex);
  state[i] = HUNGRY;
  test(i);
  up(mutex);
  down(sem[i]);
}

// only called with mutex set!
test(int i) {
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    up(sem[i]);
  }
}
Putting Down Chopsticks

```cpp
test(int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
        state[i] = EATING;
        up(sem[i]);
    }
}
```

```cpp
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```cpp
put_chopsticks(int i) {
    down(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(mutex);
}
```

```
// only called with mutex set!
test(int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
        state[i] = EATING;
        up(sem[i]);
    }
}
```
Dining Philosophers

Is the previous solution correct?
What does it mean for it to be correct?
Is there an easier way?

Important: In assignment 2 you must solve this problem using monitors and condition variables (to be discussed later), not semaphores!
The Sleeping Barber Problem
The Sleeping Barber Problem

Barber:

While there are people waiting for a hair cut, put one in the barber chair, and cut their hair
When done, move to the next customer
Else go to sleep, until someone comes in

Customer:

If barber is asleep wake him up for a haircut
If someone is getting a haircut wait for the barber to become free by sitting in a chair
If all chairs are all full, leave the barbershop
Designing a Solution

How will we model the barber and customers?
What state variables do we need?
  .. and which ones are shared?
  .... and how will we protect them?
How will the barber sleep?
How will the barber wake up?
How will customers wait?
What problems do we need to look out for?
Is This a Good Solution?

```plaintext
const CHAIRS = 5
var customers: Semaphore
barbers: Semaphore
lock: Mutex
numWaiting: int = 0

Barber Thread:
while true
    Down(customers)
    Lock(lock)
    numWaiting = numWaiting-1
    Up(barbers)
    Unlock(lock)
endWhile

Customer Thread:
Lock(lock)
if numWaiting < CHAIRS
    numWaiting = numWaiting+1
    Up(customers)
    Unlock(lock)
    Down(barbers)
    GetHaircut()
else    -- give up & go home
    Unlock(lock)
endIf
```
Readers and Writers Problem

Multiple readers and writers want to access a database (each one is a thread)
Multiple readers can proceed concurrently
Writers must synchronize with readers and other writers
  - only one writer at a time!
  - when someone is writing, there must be no readers!

Goals:
  Maximize concurrency
  Prevent starvation
Designing a Solution

How will we model the readers and writers?
What state variables do we need?
  .. and which ones are shared?
  .... and how will we protect them?
How will the writers wait?
How will the writers wake up?
How will readers wait?
How will the readers wake up?
What problems do we need to look out for?
Is This a Valid Solution?

**var** mut: Mutex = unlocked
  **db**: Semaphore = 1
  rc: int = 0

**Reader Thread:**
while true
  Lock(mut)
  rc = rc + 1
  if rc == 1
    Down(db)
  endIf
  Unlock(mut)

... Read shared data...

Lock(mut)
rc = rc - 1
if rc == 0
  Up(db)
endIf
Unlock(mut)
... Remainder Section...
endWhile

**Writer Thread:**
while true
  ...Remainder Section...
  Down(db)
  ...Write shared data...
  Up(db)
endWhile
Readers and writers solution

Does the previous solution have any problems?
- Is it “fair”?
- Can any threads be starved? If so, how could this be fixed?
- How much confidence would you have in your solution?
Quiz

What is a race condition?
How can we protect against race conditions?
Can locks be implemented simply by reading and writing to a binary variable in memory?
How can a kernel make synchronization-related system calls atomic on a uniprocessor?
   Why wouldn’t this work on a multiprocessor?
Why is it better to block rather than spin on a uniprocessor?
Why is it sometimes better to spin rather than block on a multiprocessor?
Quiz

When faced with a concurrent programming problem, what strategy would you follow in designing a solution?

What does all of this have to do with Operating Systems?