CS399 New Beginnings

Jonathan Walpole
Semaphores and Classical Synchronization Problems
The Producer-Consumer Problem

An example of the **pipelined model**
- One thread produces data items
- Another thread consumes them

Use a bounded buffer between the threads
The buffer is a shared resource
- Code that manipulates it is a **critical section**

Must suspend the producer thread if the buffer is full
Must suspend the consumer thread if the buffer is empty
Is This Solution Correct?

```java
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
```
This Code is Incorrect!

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

*What is this problem called?*
This Code is Incorrect!

What is this problem called?

Race Condition

Code that manipulates count must be made into a ??? and protected using ???
This Code is Incorrect!

What is this problem called?

Race Condition

Code that manipulates count must be made into a critical section and protected using mutual exclusion
More Problems With This Code

What if buffer is full?
- Producer will busy-wait
- On a single CPU system the consumer will not be able to empty the buffer

What if buffer is empty?
- Consumer will busy-wait
- On a single CPU system the producer will not be able to fill the buffer

We need a solution based on blocking!
A Solution Based On Blocking

```
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
Use a Mutex to Fix The Problem

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
Problems

1. Sleeping while holding the mutex causes deadlock!
2. Releasing the mutex then sleeping opens up a window during which a context switch might occur ... again risking deadlock
3. How can we release the mutex and sleep in a single atomic operation?

We need a more powerful synchronization primitive
Semaphores

An abstract data type that can be used for condition synchronization and mutual exclusion

*What is the difference between mutual exclusion and condition synchronization?*
Semaphores

Condition synchronization
  \textbf{wait} until condition holds before proceeding
  \textbf{signal} when condition holds so others may proceed
Mutual exclusion
  only one at a time in a critical section
Semaphores

An abstract data type
  containing an integer variable (S)
  Two operations: Wait (S) and Signal (S)

Alternative names for the two operations
  \( \text{Wait}(S) = \text{Down}(S) = P(S) \)
  \( \text{Signal}(S) = \text{Up}(S) = V(S) \)

Blitz names its semaphore operations Down and Up
Classical Definition

Down(S)
{
    while S <= 0 do noop; /* busy wait! */
    S = S - 1;
    /* S >= 0 */
}

Up(S)
{
    S = S + 1;
}
Problems With The Definition

Waiting threads hold the CPU
- Waste of time in single CPU systems
- Required preemption to avoid deadlock
Blocking 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; /* is # waiting threads */
        block; /* sleep */
    }

**Up(S)**

S.val = S.val + 1
If S.val <= 0
    { remove a thread T from S.list; /* negative value of S.val */
        wakeup (T);
    }
Implementing Semaphores

Down () and Up () are assumed to be atomic

How can we ensure that they are atomic?
Implementing Semaphores

Implement Down() and Up() as system calls?
- How can the kernel ensure Down() and Up() are completed atomically?
- Same solutions as before (disable interrupts, or use TSL-based mutex)
Semaphores With Disabling

```c
struct semaphore {
    int val;
    list L;
}

Down(semaphore sem)
    DISABLE_INTS
    sem.val--
    if (sem.val < 0) {
        add thread to sem.L
        sleep(thread)
    }
    ENABLE_INTS

Up(semaphore sem)
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next
        thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
```
Semaphores With Disabling

```c
struct semaphore {
    int val;
    list L;
}

Down(semaphore sem)
    DISABLE_INTS
    sem.val--
    if (sem.val < 0){
        add thread to sem.L
        sleep(thread)
    }
    ENABLE_INTS

Up(semaphore sem)
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
```
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

```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 `signal`!
... 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 == 0xffffffff
        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
Method Up ()

var oldIntStat: int
t: ptr to Thread

oldIntStat = SetInterruptsTo (DISABLED)
if count == 0xffffffff
    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
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 using atomic instructions
   1. build mutex locks from atomic instructions
   2. build semaphores from mutex locks
Building *spinning* locks with TSL

**Mutex_lock:**

```
TSL REGISTER,MUTEX
CMP REGISTER,#0
JZE ok
JMP mutex_lock
```

| copy mutex to register and set mutex to 1
| was mutex zero?
| if it was zero, mutex is unlocked, so return
| try again

```
Ok:
RET
```

| return to caller; enter critical section

**Mutex_unlock:**

```
MOVE MUTEX,#0
RET
```

| store a 0 in mutex
| return to caller
Using Locks to Build Semaphores

How would you modify the Blitz code to do this?
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?

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
        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
    }
}
```
Does This Solution Work?

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 }
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 the current Blitz project, 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?

Each philosopher is modeled with a thread.
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

```c
int state[N]
semaphore mutex = 1
semaphore 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]);
    }
}

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

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

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

```c
// 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?
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)
    CutHair()
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

Writer Thread:
while true
    ...Remainder Section...
    Down(db)
    ...Write shared data...
    Up(db)
endWhile

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
```
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?