CS 333
Introduction to Operating Systems

Class 5 – Semaphores and Classical Synchronization Problems

Jonathan Walpole
Computer Science
Portland State University
An Example Synchronization Problem
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 busy-waiting solution correct?

```
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!

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

- What is this problem called? Race Condition

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

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

- *What is this problem called?*  *Race Condition*

- Code that manipulates count must be made into a *critical section* and protected using *mutual exclusion*!
Some 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!**
Producer/Consumer with Blocking – 1st attempt

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 race condition in this code

```c
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

- Sleeping while holding the mutex causes deadlock!
- Releasing the mutex then sleeping opens up a window during which a context switch might occur ... again risking deadlock
- 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

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

- **Condition synchronization**
  - *wait* until condition holds before proceeding
  - *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: \(\text{Wait} (S)\) and \(\text{Signal} (S)\)

- **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 of Wait and Signal

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

Signal (S)
{
    S = S + 1;
}
Problems with classical definition

- Waiting threads hold the CPU
  - Waste of time in single CPU systems
  - Required preemption to avoid deadlock
Blocking implementation of semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

**Wait (S)**

\[
\text{S.val} = \text{S.val} - 1
\]

If \(\text{S.val} < 0\) /* negative value of \text{S.val} */

\{
    \text{add calling thread to S.list;} /* is \# waiting threads */
    \text{block;} /* sleep */
\}

**Signal (S)**

\[
\text{S.val} = \text{S.val} + 1
\]

If \(\text{S.val} \leq 0\)

\{
    \text{remove a thread T from S.list;}
    \text{wakeup (T);}
\}
Implementing semaphores

- Wait () and Signal () are assumed to be atomic

*How can we ensure that they are atomic?*
Implementing semaphores

- Wait () and Signal () are assumed to be atomic

How can we ensure that they are atomic?

- Implement Wait() and Signal() as system calls?
  - how can the kernel ensure Wait() and Signal() are completed atomically?
  - Same solutions as before
    - Disable interrupts, or
    - Use TSL-based mutex
Semaphores with interrupt disabling

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

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

Signal(semaphore sem) {
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next
        thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
}
Semaphores with interrupt disabling

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

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

Signal(semaphore sem) {
    DISABLE_INTS
    sem.val++
    if (sem.val <= 0) {
        th = remove next thread from sem.L
        wakeup(th)
    }
    ENABLE_INTS
}
```
method Wait ()
  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
method Wait ()
    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
method Wait ()
    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
Blitz code for Semaphore.wait

```blitz
method Wait ()
    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
```
But 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
Look at the following Blitz source code

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

- **Switch.s**
  - Switch(prevThread, nextThread)
method Signal ()
  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 Signal ()
    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 Signal ()
    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 Signal ()
    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
Semaphores using atomic instructions

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

- As we saw earlier, hardware provides special atomic instructions for synchronization
  - test and set lock (TSL)
  - compare and swap (CAS)
  - etc

- Semaphore can be built using atomic instructions
  1. build mutex locks from atomic instructions
  2. build semaphores from mutex locks
Building *spinning* mutex locks using TSL

**Mutex_lock:**

1. `TSL REGISTER,MUTEX` | copy mutex to register and set mutex to 1
2. `CMP REGISTER,#0` | was mutex zero?
3. `JZE ok` | if it was zero, mutex is unlocked, so return
4. `JMP mutex_lock` | try again

**Ok:** `RET` | return to caller; enter critical section

**Mutex_unlock:**

1. `MOVE MUTEX,#0` | store a 0 in mutex
2. `RET` | return to caller
Using Mutex Locks to Build Semaphores

- How would you modify the Blitz code to do this?
What if you had a blocking mutex lock?

**Problem:** Implement a counting semaphore

Up ()
Down ()

...using just Mutex locks

- **Goal:** Make use of the mutex lock’s blocking behavior rather than reimplementing it for the semaphore operations
How about this solution?

```plaintext
var cnt: int = 0           -- Signal count
var m1: Mutex = unlocked   -- Protects access to “cnt”
    m2: Mutex = locked    -- Locked when waiting

Down():
    Lock(m1)
    cnt = cnt - 1
    if cnt<0
        Lock(m2)
        Unlock(m1)
    else
        Unlock(m1)
    endIf

Up():
    Lock(m1)
    cnt = cnt + 1
    if cnt<=0
        Unlock(m2)
        Unlock(m1)
    else
        Unlock(m1)
    endIf
```
How about this solution?

```java
var cnt: int = 0         -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
m2: Mutex = locked     -- Locked when waiting

Down():
Lock(m1)
cnt = cnt – 1
if cnt<0
  Lock(m2)
  Unlock(m1)
else
  Unlock(m1)
endIf

Up():
Lock(m1)
cnt = cnt + 1
if cnt<=0
  Unlock(m2)
  endIf
else
  Unlock(m1)
endIf
```

Contains a Deadlock!
How about this solution then?

```plaintext
var cnt: int = 0         -- Signal count
var m1: Mutex = unlocked -- Protects access to "cnt"
    m2: Mutex = locked    -- Locked when waiting

Down():
    Lock(m1)
    cnt = cnt - 1
    if cnt<0
        Unlock(m1)
        Lock(m2)
    else
        Unlock(m1)
    endIf

Up():
    Lock(m1)
    cnt = cnt + 1
    if cnt<=0
        Unlock(m2)
    endIf
    Unlock(m1)
```
Classical Synchronization problems

- Producer Consumer (bounded buffer)
- Dining philosophers
- Sleeping barber
- Readers and writers
Producer consumer problem

- Also known as the bounded buffer problem
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

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 }

Global variables

semaphore full_buffs = 0;
semaphore empty_buffs = n;
char buff[n];
int InP, OutP;
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
Dining philosophers problem

- Five philosophers sit at a table
- One fork between each philosopher

Why do they need to synchronize?
How should they do it?

```c
while(TRUE) {
    Think();
    Grab first fork;
    Grab second fork;
    Eat();
    Put down first fork;
    Put down second fork;
}
```

Each philosopher is modeled with a thread
Is this a valid solution?

```c
#define N 5

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

- Potential for deadlock!
Working towards a solution ...

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_fork(i);
take_fork((i+1)% N);
        Eat();
        put_fork(i);
        put_fork((i+1)% N);
    }
}
```
Working towards a solution ...

```c
#define N 5

Philosopher() {
    while(TRUE) {
        Think();
        take_forks(i);
        Eat();
        put_forks(i);
    }
}
```
Picking up forks

```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;
signal(sem[i]);
    }
}

take_forks(int i) {
    wait(mutex);
    state [i] = HUNGRY;
test(i);
signal(mutex);
wait(sem[i]);
}
```
Putting down forks

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

put_forks(int i) {
    wait(mutex);
    state [i] = THINKING;
    test(LEFT);
    test(RIGHT);
    signal(mutex);
}

// only called with mutex set!
test(int i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
        state[i] = EATING;
        signal(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?

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

**Barber Thread:**
while true
    Wait(customers)
    Lock(lock)
    numWaiting = numWaiting-1
    Signal(barbers)
    Unlock(lock)
    CutHair()
endWhile

**Customer Thread:**
Lock(lock)
if numWaiting < CHAIRS
    numWaiting = numWaiting+1
    Signal(customers)
    Unlock(lock)
    Wait(barbers)
    GetHaircut()
else -- give up & go home
    Unlock(lock)
endIf
The 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 to readers & writers?

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

Reader Thread:
    while true
        Lock(mut)
        rc = rc + 1
        if rc == 1
            Wait(db)
        endIf
        Unlock(mut)
        ... Read shared data...
        Lock(mut)
        rc = rc - 1
        if rc == 0
            Signal(db)
        endIf
        Unlock(mut)
        ... Remainder Section...
    endWhile

Writer Thread:
    while true
        ... Remainder Section...
        Wait(db)
        ... Write shared data...
        Signal(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?
  - ... and 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?