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 – 1\textsuperscript{st} attempt

Global variables:
\begin{itemize}
  \item char buf[n]
  \item int InP = 0 \hspace{1em} // place to add
  \item int OutP = 0 \hspace{1em} // place to get
  \item int count
\end{itemize}

0 thread producer {
1 \hspace{1em} while(1) {
2 \hspace{2em} // Produce char c
3 \hspace{2em} if (count==n) {
4 \hspace{3em} sleep(full)
5 \hspace{2em} }
6 \hspace{1em} buf[InP] = c;
7 \hspace{1em} InP = InP + 1 mod n
8 \hspace{1em} count++
9 \hspace{1em} if (count == 1)
10 \hspace{2em} wakeup(empty)
11 \hspace{1em} }
12 \}

0 thread consumer {
1 \hspace{1em} while(1) {
2 \hspace{2em} if(count==0) {
3 \hspace{3em} sleep(empty)
4 \hspace{2em} }
5 \hspace{1em} c = buf[OutP]
6 \hspace{1em} OutP = OutP + 1 mod n
7 \hspace{1em} count--;
8 \hspace{1em} if (count == n-1)
9 \hspace{2em} wakeup(full)
10 \hspace{1em} // Consume char
11 \hspace{1em} }
12 \}

n-1
0
1
2
...
Use a mutex to fix the race condition in this code

```
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: **Wait** \((S)\) and **Signal** \((S)\)

- **Alternative names for the two operations**
  - **Wait**\((S)\) = **Down**\((S)\) = **P**\((S)\)
  - **Signal**\((S)\) = **Up**\((S)\) = **V**\((S)\)

- **Blitz names its semaphore operations Down and Up**
Classical Definition of Down and Up

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

Up(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.

**Down (S)**

S.val = S.val - 1
If S.val < 0 /* negative value of S.val */
{ 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;
  wakeup (T);
}
Implementing semaphores

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

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

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

_How can we ensure that they are atomic?_

- 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 interrupt 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 interrupt 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
 }```

Blitz code for Semaphore.down

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
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
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
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
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 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 == 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 == 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:**

- TSL REGISTER,MUTEX
  - copy mutex to register and set mutex to 1
- CMP REGISTER,#0
  - was mutex zero?
- JZE ok
  - if it was zero, mutex is unlocked, so return
- JMP mutex_lock
  - try again
- Ok: RET
  - return to caller; enter critical section

**Mutex_unlock:**

- MOVE MUTEX,#0
  - store a 0 in mutex
- RET
  - return to caller
Using Mutex 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?

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 }
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)

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

Each philosopher is modeled with a thread

- 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

```c
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

```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);
}

// 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?

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
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?

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?
  - ... 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?