CS 333
Introduction to Operating Systems

Class 4 – Synchronization Primitives
Semaphores

Jonathan Walpole
Computer Science
Portland State University
Synchronization primitives

- **Sleep**
  - Put a thread to sleep
  - Thread becomes BLOCKed

- **Wakeup**
  - Move a BLOCKed thread back onto “Ready List”
  - Thread becomes READY (or RUNNING)

- **Yield**
  - Move to another thread
  - Does not BLOCK thread
  - Just gives up the current time-slice
But how can these be implemented?

- **In User Programs:**
  - System calls to the kernel

- **In Kernel:**
  - Calls to the thread **scheduler** routines
Concurrency control in the kernel

- Different threads call Yield, Sleep, ...
- Scheduler routines manipulate the “ready list”
- The ready list is shared data!

- Problem:
  - How can scheduler routines be programmed correctly?

- Solution:
  - Scheduler can disable interrupts, or
  - Scheduler can use the TSL instruction
Concurrency in the kernel

- The kernel can avoid performing context switches while manipulating the ready list
  - prevents concurrent execution of system call code
  - ... but what about interrupts?
  - ... what if interrupt handlers touch the ready list?

- Disabling interrupts during critical sections
  - Ensures that interrupt handling code will not run

- Using TSL for critical sections
  - Ensures mutual exclusion for all code that follows that convention
Disabling interrupts

- Disabling interrupts in the OS vs disabling interrupts in user processes
  - why not allow user processes to disable interrupts?
  - is it ok to disable interrupts in the OS?
  - what precautions should you take?
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

Scenario 2:

Interrupts are already disabled and a second thread wants to access the critical section
...using the above sequence...
Disabling interrupts in the kernel

Scenario 2: 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
Classical Synchronization Problems

Producer–Consumer

- One thread produces data items
- Another thread consumes them
- Use a bounded buffer / queue between the threads
- The buffer is a shared resource
  - Must control access to it!!!
- Must suspend the producer thread if buffer is full
- Must suspend the consumer thread if buffer is empty
Producer/Consumer with Busy Waiting

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
    }
}
```
Problems with this code

- Count variable can be corrupted if context switch occurs at the wrong time
  - A race condition exists!
  - Race bugs very difficult to track down

- What if buffer is full?
  - Produce will busy-wait
  - Consumer will not be able to empty the buffer

- What if buffer is empty?
  - Consumer will busy-wait
  - Producer will not be able to fill the buffer
Producer/Consumer with Blocking

Global variables:

- `char buf[n]`
- `int InP = 0` // place to add
- `int OutP = 0` // place to get
- `int count`
This code is still 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?**
This code is still 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 still 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**!
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 invariant holds before proceeding
  - *signal* when invariant 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: Down (S) and Up (S)

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

- **Down ($S$) ... called “Wait” in SPANK**
  - decrement $S$ by 1
  - if $S$ would go negative, wait/sleep until signaled

- **Up ($S$) ... called “Signal” in SPANK**
  - increment $S$ by 1
  - signal/wakeup a waiting thread

- **$S$ will always be $\geq 0$.**
  - Both *Up ()* and *Down ()* are assumed to be *atomic***
  - A kernel implementation must ensure atomicity
Variation: Binary Semaphores

- **Counting Semaphores**
  - same as just “semaphore”

- **Binary Semaphores**
  - a specialized use of semaphores
  - the semaphore is used to implement a *Mutex Lock*
Variation: Binary Semaphores

- **Counting Semaphores**
  - same as just “semaphore”

- **Binary Semaphores**
  - a specialized use of semaphores
  - the semaphore is used to implement a *Mutex Lock*
  - the count will always be either
    - 0 = locked
    - 1 = unlocked
Using Semaphores for Mutex

\[
\text{semaphore mutex} = 1 \quad \text{-- unlocked}
\]

**Thread A**

1 repeat
2 \text{down}(mutex);
3 \text{critical section}
4 \text{up}(mutex);
5 remainder section
6 until FALSE

**Thread B**

1 repeat
2 \text{down}(mutex);
3 \text{critical section}
4 \text{up}(mutex);
5 remainder section
6 until FALSE
Using Semaphores for Mutex

\[
\text{semaphore mutex} = 0 \quad \text{-- locked}
\]

Thread A

1 repeat
2 \(\text{down}(\text{mutex});\)
3 \text{critical section}
4 \(\text{up}(\text{mutex});\)
5 remainder section
6 until FALSE

Thread B

1 repeat
2 \(\text{down}(\text{mutex});\)
3 \text{critical section}
4 \(\text{up}(\text{mutex});\)
5 remainder section
6 until FALSE
Using Semaphores for Mutex

semaphore mutex = 0 --locked

Thread A

1 repeat
2 down(mutex); ↓
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread B

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE
Using Semaphores for Mutex

$\text{semaphore mutex} = 0 \quad -- \text{locked}$

Thread A

1 repeat
2 $\text{down(mutex)}$;
3 critical section
4 $\text{up(mutex)}$;
5 remainder section
6 until FALSE

Thread B

1 repeat
2 $\text{down(mutex)}$;
3 critical section
4 $\text{up(mutex)}$;
5 remainder section
6 until FALSE
Using Semaphores for Mutex

semaphore mutex = 0 -- locked

Thread A
1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread B
1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE
Using Semaphores for Mutex

semaphore mutex = 1 -- unlocked

This thread can now be released!

Thread A

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread B

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE
Using Semaphores for Mutex

semaphore mutex = 0  -- locked

Thread A

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE

Thread B

1 repeat
2 down(mutex);
3 critical section
4 up(mutex);
5 remainder section
6 until FALSE
Exercise: Implement producer/consumer

Global variables
semaphore full_buffs = ?;
semaphore empty_buffs = ?;
char buff[n];
int InP, OutP;

0 thread producer {
1     while(1){
2         // Produce char c...
3         buf[InP] = c
4         InP = InP + 1 mod n
5     }
6 }

0 thread consumer {
1     while(1){
2         c = buf[OutP]
3         OutP = OutP + 1 mod n
4         // Consume char...
5     }
6 }
Counting semaphores in producer/consumer

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 }
Implementing semaphores

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

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

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

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

*How can we ensure that they are atomic?*

- Implement Up() and Down() as system calls?
  - how can the kernel ensure Up() and Down() are completed atomically?
  - avoid scheduling another thread when they are in progress?
  - ... but how exactly would you do that?
  - ... and what about semaphores for use in the kernel?
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
        block(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
        block(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
}
```
But what are block() and wakeup()?

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

- Scheduler routines such as block() contain calls to switch() which is called in one thread but returns in a different one!
Semaphores using atomic instructions

- 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 *blocking* 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
CALL thread_yield    | mutex is busy, so schedule another thread
JMP mutex_lock       | try again later
Ok: RET              | return to caller; enter critical section
```

**Mutex_unlock:**

```
MOVE MUTEX,#0        | store a 0 in mutex
RET                  | return to caller
```
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
- CALL thread_yield | mutex is busy, so schedule another thread
- JMP mutex_lock | try again later

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

**Mutex_unlock:**

- MOVE MUTEX,#0 | store a 0 in mutex
- RET | return to caller
To block or not to block?

- **Spin locks** do *busy waiting*
  - wastes CPU cycles on uni-processors
  - Why?

- **Blocking locks** put the thread to *sleep*
  - may waste CPU cycles on multi-processors
  - Why?
Building semaphores using mutex locks

**Problem:** Implement a counting semaphore

- Up ()
- Down ()

...using just Mutex locks
How about two “blocking” mutex locks?

```plaintext
default cnt: int = 0         -- Signal count
default 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(m1)
        Unlock(m2)
    endIf
    Unlock(m1)
```
How about two “blocking” mutex locks?

\[
\begin{align*}
\text{var cnt: int} &= 0 \quad \text{-- Signal count} \\
\text{var m1: Mutex = unlocked} \quad \text{-- Protects access to “cnt”} \\
\text{m2: Mutex = locked} \quad \text{-- Locked when waiting}
\end{align*}
\]

Down():

\[
\begin{align*}
\text{Lock(m1)} \\
\text{cnt} &= \text{cnt - 1} \\
\text{if cnt}<0 \\
\text{Unlock(m1)} \\
\text{Lock(m2)} \\
\text{else} \\
\text{Unlock(m1)} \\
\text{endIf}
\end{align*}
\]

Up():

\[
\begin{align*}
\text{Lock(m1)} \\
\text{cnt} &= \text{cnt + 1} \\
\text{if cnt}</0 \\
\text{Unlock(m2)} \\
\text{endIf} \\
\text{Unlock(m1)}
\end{align*}
\]

Contains a Race Condition!
Oops! How about this then?

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
   Unlock(m1)
Oops! How about this 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
    Lock(m2)
    Unlock(m1)
  else
    Unlock(m1)
  endIf

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

Contains a Deadlock!
Ok! Let's have another try!

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

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

Up ():
    Lock(m1)
    cnt = cnt + 1
    if cnt=1
        Unlock(m2)
    endIf
    Unlock(m1)

... is this solution valid?
```
What about this solution?

Mutex m1, m2; // binary semaphores
int C = N;     // N is # locks
int W = 0;     // W is # wakeups

Down():
    Lock(m1);
    C = C – 1;
    if (C<0)
        Unlock(m1);
    Lock(m2);
    Lock(m1);
    W = W – 1;
    if (W>0)
        Unlock(m2);
    endif;
else
    Unlock(m1);
endif;

Up():
    Lock(m1);
    C = C + 1;
    if (C<=0)
        W = W + 1;
    Unlock(m2);
    endif;
    Unlock(m1);
Implementation possibilities

- Implement Mutex Locks
  ... using Semaphores

- Implement Counting Semaphores
  ... using Binary Semaphores
  ... using Mutex Locks

- Implement Binary Semaphores
- ... etc

Can also implement using Test-And-Set Calls to Sleep, Wake-Up
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?
Spare slides
Semaphores in UNIX

- User-accessible semaphores in UNIX are somewhat complex
- Each up and down operation is done atomically on an “array” of semaphores.

********** WARNING **********

- Semaphores are allocated by (and in) the operating system
  - Number based on configuration parameters
- Semaphores in UNIX are a shared resource, potentially used by almost everyone
- Must REMOVE your semaphores after you are done with them
Typical usage

```c
main()
{
    int sem_id;
    sem_id = NewSemaphore(1);
    ...
    Down(sem_id);

    [CRITICAL SECTION]

    Up (sem_id);

    ...
    FreeSemaphore(sem_id);
}
```
Managing your UNIX semaphores

Listing currently allocated ipc resources

ipcs

Removing semaphores

ipcrm -s <sem number>