CS 333
Introduction to Operating Systems

Class 4 – Synchronization Primitives
Semaphores

Jonathan Walpole
Computer Science
Portland State University
An Example using a Mutex

Shared data:

Mutex myLock;

1 repeat
2   Lock(myLock);
3   critical section
4   Unlock(myLock);
5   remainder section
6   until FALSE
But how can we implement a mutex lock?

- Both **Lock** and **Unlock** are assumed to be *atomic***

- Does a binary “lock” variable in memory work?

- Many computers have *some limited* hardware support for setting locks
  - “Atomic” Test and Set Lock instruction
  - “Atomic” compare and swap operation

- Can be used to implement “Mutex” locks
Test-and-set-lock instruction (TSL, tset)

- A lock is a single word variable with two values
  - $0 = \text{FALSE} = \text{not locked}$
  - $1 = \text{TRUE} = \text{locked}$

- Test-and-set does the following *atomically*:
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

If the returned value was FALSE...
Then you got the lock!!!

If the returned value was TRUE...
Then someone else has the lock
(so try again later)
Test and set lock

FALSE

Lock
Test and set lock

P1

FALSE

Lock
Test and set lock

FALSE = Lock Available!!

P1

FALSE

Lock
Test and set lock

P1

FALSE

TRUE

Lock
Test and set lock

P1 → FALSE

TRUE

Lock
Test and set lock
Test and set lock
Test and set lock
Test and set lock

P1

P2

P3

P4

FALSE

Lock

TRUE

TRUE

TRUE

TRUE

TRUE

TRUE
Test and set lock
Test and set lock
Test and set lock
Critical section entry code with TSL

1 repeat
2 while (TSL(lock))
3 no-op;
4 critical section
5 Lock = FALSE;
6 remainder section
7 until FALSE

- Guarantees that only one thread at a time will enter its critical section
- Note that processes are busy while waiting
  - Spin locks
Busy waiting

- Also called polling or spinning
  - The thread consumes CPU cycles to evaluate when lock becomes free!!!
- Shortcoming on a single CPU system...
  - A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!
  - Better: Block instead of busy wait!
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!
- The code that manipulates it is a critical section

- Problem:
  ✷ How can scheduler routines be programmed correctly?
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
  - ... but what if there are multiple CPUs?

- 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
But disabling interrupts is not enough ...

- **Disabling interrupts during critical sections**
  - Ensures that interrupt handling code will not run
  - But what if there are multiple CPUs?
  - A thread on a different CPU might make a system call which invokes code that manipulates the ready queue

- **Using a mutex lock (based on TSL) for critical sections**
  - Ensures mutual exclusion for all code *that follows that convention*
Some tricky issues ...

- The interrupt handling code that saves interrupted state is a critical section
  - It could be executed concurrently if multiple almost simultaneous interrupts happen
  - Interrupts must be disabled during this (short) time period to ensure critical state is not lost

- What if this interrupt handling code attempts to lock a mutex that is held?
  - What happens if we sleep with interrupts disabled?
  - What happens if we busy wait (spin) with interrupts disabled?
Classical Synchronization Problems

Producer-Consumer

- An example of the pipelined model
  - One thread produces data items
  - Another thread consumes them
- Use a bounded buffer / queue between the threads
- The buffer is a shared resource
  - Code that manipulates it is a critical section
- Must suspend the producer thread if buffer is full
- Must suspend the consumer thread if buffer is empty
Producer/Consumer with Busy Waiting

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
Producer/Consumer with Blocking

Global variables:
- char buf[n]
- int InP = 0  // place to add
- int OutP = 0  // place to get
- int count

```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 }
```

```c
0 thread consumer {
1     while(1) {
2         while (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 }
```
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 }
```

```c
0 thread consumer {
1    while(1) {
2        while (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 lock then sleeping opens up a window during which a context switch might occur ... again risking deadlock
- How can we release the lock 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 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: \(\text{Down} (S)\) and \(\text{Up} (S)\)

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

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

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

- **S will always be >= 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

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

\begin{itemize}
  \item[1] repeat
  \item[2] \texttt{down}(mutex);
  \item[3] \texttt{critical section}
  \item[4] \texttt{up}(mutex);
  \item[5] remainder section
  \item[6] until FALSE
\end{itemize}

Thread A

\begin{itemize}
  \item[1] repeat
  \item[2] \texttt{down}(mutex);
  \item[3] \texttt{critical section}
  \item[4] \texttt{up}(mutex);
  \item[5] remainder section
  \item[6] until FALSE
\end{itemize}

Thread B
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

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

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

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

**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}(\text{mutex});
3 \text{critical section}
4 \text{up}(\text{mutex});
5 \text{remainder section}
6 until FALSE

Thread B

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

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

\[ \begin{align*}
1 & \text{repeat} \\
2 & \quad \text{down}(\text{mutex}); \\
3 & \quad \text{critical section} \\
4 & \quad \text{up}(\text{mutex}); \\
5 & \quad \text{remainder section} \\
6 & \text{until FALSE}
\end{align*} \]

This thread can now be released!

Thread A

\[ \begin{align*}
1 & \text{repeat} \\
2 & \quad \text{down}(\text{mutex}); \\
3 & \quad \text{critical section} \\
4 & \quad \text{up}(\text{mutex}); \\
5 & \quad \text{remainder section} \\
6 & \text{until FALSE}
\end{align*} \]

Thread B
Using Semaphores for Mutex

Semaphore mutex = 0 -- locked

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

Thread A

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

Thread B
Exercise: Implement producer/consumer

Global variables

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

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

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

```
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 *yielding* 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?

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)
How about two “blocking” mutex locks?

```python
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)
    else
        Lock(m2)
    EndIf
Unlock(m1)

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

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!

```plaintext
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?