

# CS241 System Programming

## Process Synchronization (3)

Klara Nahrstedt

Lecture 8

2/6/2006

---

# Content

- Mutex
- Semaphores
- Monitors
- Summary

# Administrative

- Read: T: Chapter 2.2
- Read: R&R 13.1-13.4 (Mutex Locks and Conditional Variables)
- Read: R&R 14.1- 14.5 (Semaphores)
- MP1 is running
- Quiz 2: Friday, February 10, 2006 (topic – processes/threads and synchronization)

# Semaphores

- A semaphore count represents count number of abstract resources.
- New variable having 2 operations
  - The Down/P operation is used to acquire a resource and decrements count.
  - The Up/V operation is used to release a resource and increments count.
- Any semaphore operation is indivisible (**atomic**)
- Semaphores solve the problem of the wakeup-bit

# What's Up? What's Down?

- Definitions of Down/P and Up/V (P and V is a different notation for operations Down and Up):

Down(S):  $S.value := S.value - 1$

if  $S.value < 0$  then {add process to queue S.L ;  
block;}

Up(S):  $S.value = S.value + 1$

if  $S.value \leq 0$  then {remove process from queue S.L;  
wakeup;}

- Counting semaphores:  $0..N$
- Binary semaphores: 0,1

# Mutex: Binary Semaphore

- Variable with only 2 states
  - Lock
  - Unlock
- Simplified version of semaphore
- Mutex is used for mutual exclusion

# Mutex Implementation using TSL

- Using Test\_and\_Set (TSL) instruction to implement
  - Mutex\_lock
  - Mutex\_unlock

# Mutex Implementation 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 was unlocked, so return
CALL thread_yield      | mutex is busy; schedule another thread
JMP mutex_lock         | try again later
```

ok: RET | return to caller; critical region entered

mutex\_unlock:

```
MOVE MUTEX,#0          | store a 0 in mutex
RET | return to caller
```

Implementation of *mutex\_lock* and *mutex\_unlock*

# Producer-Consumer Problem using Semaphores

```
semaphore mutex = 1; /* controls access to critical region */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0; /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* decrement empty count \*/*  
*/\* enter critical region \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*

# Using Mutex to Implement Semaphores

- Using `mutex_lock` and `mutex_unlock` to implement a **counting semaphore**
  - Up
  - Down

# Counting Semaphore

## Implementation using Binary Semaphores

(for Mutual Exclusion)

```
class Semaphore {
    Mutex m1; // Mutual exclusion to protect count
    Mutex m2; // Mutual exclusion to protect critical region
    int count; // Resource count.
public:
    Semaphore( int count );
    void Up();
    void Down();
};

static inline Semaphore::Semaphore( int count )
{
    count = count;
}
```

# Counting Semaphore Implementation (2)

```
void  
Semaphore::Down()  
{  
    mutex_lock(m1);  
    count--;  
    if ( count < 0 )  
    {  
        mutex_unlock(m1);  
        mutex_lock(m2);  
    }  
    mutex_unlock(m1);  
}
```

```
void  
Semaphore::Up()  
{  
    mutex_lock(m1);  
    count++;  
    if (count <=0)  
    then  
        mutex_unlock(m2);  
    else  
        mutex_unlock(m1);  
}
```

# Implementations of Semaphore using Sleep & Wakeup

```
type Semaphore = record
    value:integer;
    L: list of processes;
Semaphore S;
```

## Down(S):

```
S.value:= S.value - 1;
if S. value < 0 then
{
    add this process to the S.L;
    sleep;
};
```

## Up(S):

```
S.value:= S.value + 1;
if S.value > 0 then
{
    remove process P from S.L;
    wakeup(P);
}
```

Does it work?

# Implementations of Semaphore using Sleep & Wakeup

```
type Semaphore = record
    value:integer;
    L: list of processes;
Semaphore S;
```

## Down(S):

```
S.value:= S.value - 1;
if S. value < 0 then
{
    add this process to the S.L;
    sleep;
};
```

## Up(S):

```
S.value:= S.value + 1;
if S.value <= 0 then
{
    remove process P from S.L;
    wakeup(P);
}
```

Does it work?

# Tradeoffs

- Busy waiting (spinlock)
  - Waste CPU cycles
- Sleep&Wakeup (blocked lock)
  - Context switch overhead
- Hybrid competitive solution (spin-block)
  - Apply spinlocks if the waiting time is shorter than the context switch time
  - Use sleep & wakeup if the waiting time is longer than the context switch time

# Possible Deadlocks with Semaphores

Example:

P0

P1

share two semaphores S and Q

S:= 1; Q:=1;

Down(S); // S=0 -----> Down(Q); //Q=0

Down(Q); // Q= -1 <-----

-----> Down(S); // S=-1

// P0 blocked

// P1 blocked

DEADLOCK

Up(S);

Up(Q);

Up(Q);

Up(S);

# Be Careful When Using Semaphores

// Violation of Mutual Exclusion

```
Up(mutex);           mutexUnlock();
critical section     criticalSection();
Down(mutex);        mutexLock();
```

// Deadlock Situation

```
Down(mutex);        mutexLock();
critical section     criticalSection();
Down(mutex);        mutexLock(P);
```

// Violation of Mutual Exclusion (omit wait(mutex)/mutexLock())

```
critical section     critical Section();
Up(mutex);           mutexUnlock();
```

// Deadlock Situation (omit signal(mutex)/mutexUnlock())

```
Down(mutex);        mutexLock();
critical section     criticalSection();
```

# Implementation of Simple Synchronization Primitives in POSIX

- Mutex Lock (use `<pthread.h>`) – discussed this week in **discussion sections**
  - Mutex lock (`pthread_mutex_lock`)
  - Mutex Unlock (`pthread_mutex_unlock`)
- Semaphore (use `<semaphore.h>`) - discussed this week in **discussion sections**
  - Initialize semaphore (`sem_init`)
  - Destroy semaphore (`sem_destroy`)
  - Signal (Up) semaphore (`sem_post`)
  - Wait (Down) semaphore (`sem_wait`)
- Condition Variables
  - Waiting on condition (`pthread_cond_wait`)

# POSIX Mutex Locks Functions

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);  
int pthread_mutex_init(pthread_mutex_t *restrict  
    mutex,);  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

# Mutex Locks (1)

- A mutex, or mutex lock, is a special variable that can be either in the locked state or the unlocked state.
- If the mutex is locked, it has a distinguished thread that *holds* or *owns* the mutex.
- If the mutex is unlocked, we say that the mutex is *free* or *available*.
- The mutex also has a queue of threads that are waiting to hold the mutex.
- POSIX does not require that this queue be accessed FIFO.

# Mutex Locks (2)

- It is usually used to protect access to a shared variable.  
idea:  
lock the mutex  
    critical section  
unlock the mutex
- Only the owner of the mutex should unlock the mutex.
- Do not lock a mutex that is already locked.
- Do not unlock a mutex that is not already locked.

# Use

- A mutex has type `pthread_mutex_t`
- Since a mutex is meant to be used by multiple threads, it is usually declared to have static storage class.
- It can be defined inside a function using the static qualifier if it will only be used by that function or it can be defined at the top level.
- A mutex must be initialized before it is used.
- This can be done when the mutex is defined, as in:  

```
pthread_mutex_t mymutex =  
PTHREAD_MUTEX_INITIALIZER;
```

# Counter example

```
#include <pthread.h>
static int count = 0;
static pthread_mutex_t countlock =
    PTHREAD_MUTEX_INITIALIZER;

int increment(void) {                /* increment the counter */
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    count++;
    return pthread_mutex_unlock(&countlock);
}
```

# Counter Example (2)

```
int decrement(void) {                /* decrement the counter */
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    count--;
    return pthread_mutex_unlock(&countlock);
}
```

```
int getcount(int *countp) {          /* retrieve the counter */
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    *countp = count;
    return pthread_mutex_unlock(&countlock);
}
```

# Monitor – Sync Primitive

- A simpler way to synchronize
- A set of programmer defined operators

```
monitor monitor-name
{
/* variable declaration */

procedure P1(..);
{... };

.....

procedure Pn(..);
{...};

begin
  initialization code
end
```

- **Monitor for producer-consumer**

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

procedure insert(item:integer);
  { if (count == N) then wait(full);
    insert_item(item);
    count++;
    if (count == 1 ) then signal(empty)}

function remove: integer;
  { if (count==0) then wait(empty);
    remove = remove_item;
    count--;
    if (count == N-1) then signal(full)}

Count:: = 0 /*init code*/

end monitor
```

# Monitor Properties

- The internal implementation of a monitor type cannot be accessed directly by the various threads.
- The encapsulation provided by the monitor type limits access to the local variables only by the local procedures.
- Monitor construct **does not allow concurrent** access to all procedures defined within the monitor.
- **Only one thread/process can be active within the monitor at a time.**
- Synchronization is built in.
- Monitor also needs **condition variable** with two operations
  - **Wait** – suspends process until condition is signaled  
Pthread\_cond\_wait
  - **Signal** – resumes one process awaiting condition
    - Pthread\_cond\_signal

# Summary

- **Mutex Locks** are simpler synchronization primitives than semaphores, but also more sync mistakes can happen (e.g., with mutex one must carefully count how many lock/unlock operations happened since there is no control built in – counting semaphores help here)
- **Counting semaphores** are simpler synchronization primitives than monitors, but still more sync mistakes can happen (e.g., if a process forgets to set the semaphore ‘up’ – monitors help here)
- **Monitors** are the most elaborate sync primitives, but not each language has then
- Most programming languages will have mutex locks and semaphores