



www.kingofthecomics.com

© 2004 Barron, Inc. Distributed by North Atlantic
Syndicates Inc. All Rights Reserved.

BRITTON



Synchronization Primitives – Semaphore and Mutex

Lecture 12

Klara Nahrstedt

CS241 Administrative

- Regular Quiz 4 this week based on Self-Assessment4 and Self-Assessment5
- Read Chapter 14.1-14.4 in R&R and Chapter 13.1-13.2
- Don't forget Self-Assessment5 Quiz on Compass

Outline

- Using Semaphores – Examples
- Thread Synchronization
 - Mutex Synchronization Primitive
 - Operations on Mutex
 - Examples

Review: Implementation of Semaphores in POSIX

POSIX:SEM semaphore is
variable of type sem_t

Atomic Operations:

```
int sem_init(sem_t *sem, int pshared, unsigned value);  
int sem_destroy(sem_t *sem);  
int sem_post(sem_t *sem);  
int sem_trywait(sem_t *sem);  
int sem_wait(sem_t *sem);
```

Use <semaphore.h>

Example 1 on Semaphore (RR: 497)

We want a shared variable **'shared'** (*critical section*) to be protected by semaphore to allow for two functions

getshared – is a function that returns the current value of the shared variable **'shared'**

incshared – is a function that that atomically increments the **'shared'** variable.

Example, creating shared variable

```
#include <errno.h>
```

```
#include <semaphore.h>
```

```
static int shared = 0;
```

```
static sem_t sharedsem;
```

```
int initshared(int val) {
```

```
    if (sem_init(&sharedsem, 0, 1) == -1)
```

```
        return -1;
```

```
    shared = val;
```

```
    return 0;
```

```
}
```

Example – shared variable

```
int getshared(int *sval) {
    while (sem_wait(&sharedsem) == -1)
        if (errno != EINTR)
            return -1;
    *sval = shared;
    return sem_post(&sharedsem);
}

int incshared() {
    while (sem_wait(&sharedsem) == -1)
        if (errno != EINTR)
            return -1;
    shared++;
    return sem_post(&sharedsem);
}
```

Example 2 on Semaphore (RR:500)

A program to generate a set of threads and each thread writes to *standard error*

Standard error (stderr) is a shared resource, hence if a thread outputs an informative message to standard error one character at the time, it becomes a critical region and we must protect it.

Thread with Critical Section (Example RR:500) -

```
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define TEN_MILLION 10000000L
#define BUFSIZE 1024
```

```
void *threadout(void *args) {
    char buffer[BUFSIZE];
    char *c;
    sem_t *semlockp;
    struct timespec sleeptime;
```

Thread with Critical Section

```
/* entry section */
while (sem_wait(semlockp) == -1) /* Entry section */
    if(errno != EINTR) {
        fprintf(stderr, "Thread failed to lock semaphore\n");
        return NULL;
    }

/* start of critical section */
while (*c != '\0') {
    fputc(*c, stderr);
    c++;
    nanosleep(&sleeptime, NULL);
}

/* exit section */
if (sem_post(semlockp) == -1) /* Exit section */
    fprintf(stderr, "Thread failed to unlock semaphore\n");

/* remainder section */
return NULL; }
```

Main program (Example RR:501)

```
#include <pthread.h>
```

```
#include <semaphore.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
void *threadout(void *args);
```

```
int main(int argc, char *argv[]) {
```

```
    int error;
```

```
    int i;
```

```
    int n;
```

```
sem_t semlock;
```

```
pthread_t *tids;
```

Main program (Example RR:501)

```
if (argc != 2){ /* check for valid number of command-line
arguments */
    fprintf (stderr, "Usage: %s numthreads\n", argv[0]);
    return 1;
}
n = atoi(argv[1]);
tids = (pthread_t *)calloc(n, sizeof(pthread_t));
if (tids == NULL) {
    perror("Failed to allocate memory for thread IDs");
    return 1;
}
if (sem_init(&semlock, 0, 1) == -1) {
    perror("Failed to initialize semaphore");
    return 1;
}
```

*What happens if we replace **sem_init** with **sem_init(&semlock,0,0)** ?*

Main program

```
for (i = 0; i < n; i++)
    if (error = pthread_create(tids + i, NULL,
                              threadout,
                              &semlock)) {
        fprintf(stderr, "Failed to create thread: %s\n",
                strerror(error));
        return 1;
    }
for (i = 0; i < n; i++)
    if (error = pthread_join(tids[i], NULL)) {
        fprintf(stderr, "Failed to join thread: %s\n",
                strerror(error));
        return 1;
    }
return 0; }
```

Mutex (Locks, Latches)

Useful for short-term locking

Simplest and most efficient thread synchronization mechanism



A special variable that can be either in

locked state: a distinguished thread that *holds* or *owns* the *mutex*; or

unlocked state: no thread holds the *mutex*

When several threads compete for a *mutex*, the losers block at that call

The *mutex* also has a queue of threads that are waiting to hold the *mutex*.

POSIX does not require that this queue be accessed FIFO¹⁵.

POSIX Mutex-related Functions

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,  
                        const pthread_mutexattr_t *restrict attr);
```

```
int pthread_mutex_destroy(pthread_mutex_t *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 and Shared Variables

Mutex locks are usually used to protect access to a shared variable.

The idea:

- lock the mutex

- critical section

- unlock the mutex

Unlike a semaphore, a mutex does not have a value, it has **states** (locked and unlocked).

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.

A Typical Way to Use Mutex

1. Create and initialize a mutex variable
2. Several threads attempt to lock the mutex
3. Only one succeeds and that thread owns the mutex
4. The owner thread performs some set of actions
5. The owner unlocks the mutex
6. Another thread acquires the mutex and repeats the process
7. Finally the mutex is destroyed

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;
```

Mutex vs. Semaphore

Differences/similarity between the two?

Mutex is also called binary semaphore in contrast to general counting semaphore

Counter is initialized to ???

`pthread_mutex_lock` = ???

`pthread_mutex_unlock` = ???

Binary Semaphore Primitives (from previous lecture)

```
struct binary_semaphore {  
    enum {0,1} value;  
    queueType queue; };  
  
void semWaitB(binary_semaphore s)  
{  
    if (s.value == 1)  
        s.value = 0;  
    else  
        { place this process in s.queue;}  
        block this process;  
    }  
}  
  
void semSignalB(binary_semaphore s)  
{  
    if (s.queue is empty())  
        s.value = 1;  
    else  
        {  
            remove a process P from s.queue;  
            place process P on ready list;  
        }  
}
```

These primitives could be used to implement pthread_thread_lock and unlock

So Which One is More Powerful?

Is semaphore more powerful?

Is there anything that semaphores can implement but mutex cannot?

The answer: they are equivalent!

But sometimes one may be more convenient than the other

Due to this reason, some systems support only *mutex*, not general counting semaphores.

Pair Discussion

Can you discuss with another student to find out

How to use a mutex to implement a general counting semaphore?

Definition of Semaphore Primitives (Counting Semaphore) — PseudoCode from Previous Lecture

```
struct semaphore{  
    int count;  
    queueType queue; };
```

```
void semWait(semaphore s)  
{  
    s.count--;  
    if (s.count < 0)  
    { place this process in s.queue;  
      block this process;  
    }  
}
```

```
void semSignal(semaphore s)  
{  
    s.count++;  
    if (s.count ≤ 0)  
    {  
        remove a process P from  
        s.queue;  
        place process P on ready list;  
    }  
}
```

Busy Wait Counting Semaphore Implementation (1)

```
typedef struct {  
    mutex m; // Mutual exclusion to do ???  
    int count; // Resource count.  
}semaphore;
```

```
int sem_init (semaphore *sem, int count);  
{  
    mutex *m = &(sem->m);  
    mutex_lock(m);  
    sem->count = count;  
    mutex_unlock(m);  
    return 0;  
}
```

Busy Wait Counting Semaphore Implementation(2)

```
int
sem_wait( semaphore *sem)
{
    mutex *m = &(sem->m);

    mutex_lock(m);
    sem->count--;
    while ( sem->count < 0 )
    {
        mutex_unlock(m);
        /* do nothing */
        mutex_lock(m);
    }
    mutex_unlock(m);
}

int sem_post(semaphore *sem)
{
    mutex *m = &(sem->m);

    mutex_lock(m);
    sem->count++;
    mutex_unlock(m);
}
```

what is the limitation?

Answer: busy polling → wasting CPU time 26

Summary

Mutex and Semaphore are very powerful synchronization primitives

Both allow set of instructions to be **executed atomically**

Mutex is a very **simple primitive** used only for short time usage of data structure updates

Counting semaphores are powerful if one wants to
- allow for example only '**count \geq 1**' number of processes/threads into the critical region