**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS**

**Information and Computer Science Department**

**ICS 431 Operating Systems**

**Lab # 11**

**Threads Synchronization ( Mutex Locks & Semaphores )**

**Objectives:**

* To learn how to use Mutual Exclusion (Mutex) Locks
* To learn how to use semaphores

**Mutex Variables:**

Mutex is a shortened form of the words "mutual exclusion".

Mutex variables are one of the primary means of implementing thread synchronization.

A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.

Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".

A typical sequence in the use of a mutex is as follows:

* Create and initialize a mutex variable
* Several threads attempt to lock the mutex
* Only one succeeds and that thread owns the mutex
* The owner thread performs some set of actions
* The owner unlocks the mutex
* Another thread acquires the mutex and repeats the process
* Finally the mutex is destroyed

When several threads compete for a mutex, the losers block at that call an unblocking call is available with "trylock" instead of the "lock" call.

**Mutex Functions:**

pthread\_mutex\_init ( pthread\_mutex\_t mutex, pthread\_mutexattr\_t attr)

pthread\_mutex\_destroy ( pthread\_mutex\_t mutex )

pthread\_mutexattr\_init ( pthread\_mutexattr\_t attr )

pthread\_mutexattr\_destroy ( pthread\_mutexattr\_t attr )

pthread\_mutex\_lock ( pthread\_mutex\_t mutex )

pthread\_mutex\_trylock ( pthread\_mutex\_t mutex )

pthread\_mutex\_unlock ( pthread\_mutex\_t mutex )

pthread\_mutex\_init( ) creates and initializes a new mutex object, and sets its attributes according to the mutex attributes object, attr. The mutex is initially unlocked.

Mutex variables must be of type pthread\_mutex\_t.

The attr object is used to establish properties for the mutex object, and must be of type pthread\_mutexattr\_t if used (may be specified as NULL to accept defaults).

If implemented, the pthread\_mutexattr\_init() and pthread\_mutexattr\_destroy() routines are used to create and destroy mutex attribute objects respectively.

pthread\_mutex\_destroy( ) should be used to free a mutex object which is no longer needed. The pthread\_mutex\_lock( ) routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, the call will block the calling thread until the mutex is unlocked.

pthread\_mutex\_trylock( ) will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

Mutex contention: when more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released? Unless thread priority scheduling (not covered) is used, the assignment will be left to the native system scheduler and may appear to be more or less random.

pthread\_mutex\_unlock() will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.

An error will be returned if the mutex was already unlocked, or if the mutex is owned by another thread

**POSIX Semaphore:**

The following header summarizes how we can use POSIX.1b unnamed semaphore:

int sem\_init(sem\_t \*sem, int pshared, unsigned value);

int sem\_destroy(sem\_t \*sem);

int sem\_wait(sem\_t \*sem);

int sem\_post(sem\_t \*sem);

int sem\_trywait(sem\_t \*sem);

All of the POSIX.1b semaphore functions return **-1** to indicate an error.

**sem\_init** function initializes the semaphore to have the value value. The value parameter cannot be negative. If the value of pshared is not 0, the semaphore can be used between processes (i.e. the process that initializes it and by children of that process). Otherwise it can be used only by threads within the process that initializes it.

**sem\_wait** is a standard semaphore wait operation. If the semaphore value is 0, the sem\_wait blocks unit it can successfully decrement the semaphore value.

**sem\_trywait** is similar to sem\_wait except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns -1.

**sem\_post** is a standard semaphore signal operation. The POSIX.1b standard requires that sem\_post be reentrant with respect to signals, that is, it is asynchronous-signal safe and may be invoked from a signal-handler.

**Example#1:**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

void\* threadroutine(void\*);

pthread\_mutex\_t m;

int x = 0;

int main(void)

{

    pthread\_t tid;

    pthread\_mutex\_init(&m, NULL);

    pthread\_create(&tid, NULL, threadroutine, NULL);

    pthread\_mutex\_lock(&m);

    for(int i = 0; i < 100000; i++)

        x += 10;

    pthread\_mutex\_unlock(&m);

    pthread\_join(tid, NULL);

    pthread\_mutex\_destroy(&m);

    printf("x = %d\n", x);

    return 0;

}

void\* threadroutine(void\* arg)

{

    pthread\_mutex\_lock(&m);

    for(int i = 0; i < 100000; i++)

        x += 5;

    pthread\_mutex\_unlock(&m);

}

Run the same program by commenting pthread\_mutex\_lock(&m) and pthread\_mutex\_unlock(&m) and observe the differences.

**Example#2:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

int x = 0;

sem\_t s;

void \*threadroutine(void \*arg)

{

    sem\_wait(&s);

    printf("x = %d\n", x);

}

int main()

{

    pthread\_t tid;

    sem\_init(&s, 0, 0);

    pthread\_create(&tid, NULL, threadroutine, NULL);

    x = 55;

    sem\_post(&s);

    pthread\_join(tid, NULL);

    sem\_destroy(&s);

    return 0;

}

**Example#3:**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

int x = 0;

sem\_t m;

void \*threadroutine(void \*arg)

{

    sem\_wait(&m);

    x++;

    sem\_post(&m);

}

int main ()

{

    pthread\_t tid[10000];

    int i;

    sem\_init(&m, 0, 1);

    for (i = 0; i < 10000; i++)

        pthread\_create(&tid[i], NULL, threadroutine, NULL);

    for (i = 0; i < 10000; i++)

        pthread\_join(tid[i], NULL);

    sem\_destroy(&m);

    printf("Final value of x is %d\n", x);

    return 0;

}

Exercise:

Write program that has two threads such that the first thread generates 5 random numbers and the second thread prints them but the first thread generates a new number only after the previously generated number has been printed by the second thread.

Make two versions of your program:

(a) using mutex locks, and

(b) using semaphores