KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Information and Computer Science Department
ICS 431 Operating Systems
Lab # 8
Threads Synchronization (Mutex & Condition Variables)

Objective:

When multiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities.

Threads need to synchronize their activities to effectively interact. This includes:

- Implicit communication through the modification of shared data
- Explicit communication by informing each other of events that have occurred.

This lab describes the synchronization types available with threads and discusses when and how to use synchronization. There are a few possible methods of synchronizing threads and **here we will discuss**:

- Mutual Exclusion (Mutex) Locks
- Condition Variables

Why we need Synchronization and how to achieve it?

Suppose the multiple threads share the common address space (thru a common variable), and then there is a problem.

THREAD A
```
x = common_variable ;
x++ ;
common_variable = x ;
```

THREAD B
```
y = common_variable ;
y-- ;
common_variable = y ;
```

If threads execute this code independently it will lead to garbage. The access to the common_variable by both of them simultaneously is prevented by having a lock, performing the thing and then releasing the lock.

Mutexes and Race Conditions:

Mutual exclusion locks (mutexes) can prevent data inconsistencies due to race conditions.
A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Consider, for example, a single counter, X, that is incremented by two threads, A and B. If X is originally 1, then by the time threads A and B increment the counter, X should be 3. Both threads are independent entities and have no synchronization between them.

Although the C statement X++ looks simple enough to be atomic, the generated assembly code may not be, as shown in the following pseudo-assembler code:

```
move X, REG
inc REG
move REG, X
```

If both threads are executed concurrently on two CPUs, or if the scheduling makes the threads alternatively execute on each instruction, the **following steps may occur:**

1. Thread A executes the first instruction and puts X, which is 1, into the thread A register. Then thread B executes and puts X, which is 1, into the thread B register. The following illustrates the resulting registers and the contents of memory X.

Thread A Register	Thread B Register	Memory X
1	1	1

2. Next, thread A executes the second instruction and increments the content of its register to 2. Then thread B increments its register to 2. Nothing is moved to memory X, so memory X stays the same. The following illustrates the resulting registers and the contents of memory X.

Thread A Register	Thread B Register	Memory X
2	2	1

3. Last, thread A moves the content of its register, which is now 2, into memory X. Then thread B moves the content of its register, which is also 2, into memory X, overwriting thread A's value. The following illustrates the resulting registers and the contents of memory X.

Thread A Register	Thread B Register	Memory X
2	2	2

Note that in most cases thread A and thread B will execute the three instructions one after the other, and the result would be 3, as expected. Race conditions are usually difficult to discover, because they occur intermittently. To avoid this race condition, each thread should lock the data before accessing the counter and updating memory X. For example, if thread A takes a lock and updates the counter, it leaves memory X with a value of 2. Once thread A releases the lock, thread B takes the lock and updates the counter, taking 2 as its initial value for X and incrementing it to 3, the expected result.

Waiting for Threads:

Condition variables allow threads to block until some event or condition has occurred.

Boolean predicates indicate whether the program has satisfied a condition variable.
The programmer defines the complexity of a condition variable predicate. A condition can be signaled by any thread to either one or all-waiting threads.

Mutex Variables:

Overview

- 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.

Creating / Destroying Mutexes :

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_init( ) creates and initializes a new mutex 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.

Locking / Unlocking Mutexes :

pthread_mutex_lock (pthread_mutex_t mutex)

pthread_mutex_trylock (pthread_mutex_t mutex)

pthread_mutex_unlock (pthread_mutex_t mutex)

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:

If the mutex was already unlocked
If the mutex is owned by another thread

Example: (Using Mutexes)

This simple example code demonstrates the use of several Pthread mutex routines. The serial version may be reviewed first to compare how the threaded version performs the same task.

Lab1.c:

/ Code for hellothread.c */*

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

/* The function run when the thread is created */
void* compute_thread (void*);
main( )
{
      /* This is data describing the thread created */
      pthread_t    tid; /* thread ID structure */
```

```
        pthread_attr_t    attr; /* thread attributes */
        char hello[ ] = {"Hello, "}; /* some typical data */
        char thread[ ] = {"thread"};

        /* Initialize the thread attribute structure */
        pthread_attr_init(&attr);
        /* Create another thread. ID is returned in &tid */
        /* The last parameter passed to the thread function */
        pthread_create(&tid, &attr, compute_thread, thread);

        /* Continue on with the base thread */
        printf(hello);
        sleep(1);
        printf("\n");
        exit(0);
}

/* The thread function to be run */
void* compute_thread(void* dummy)
{
        printf(dummy);
        return;
}
```

Lab2.c:

/* Code for mutex-hellothread.c */

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

/* Function run when the thread is created */
void* compute_thread (void*);
/* This is the lock for thread synchronization */
pthread_mutex_t    my_sync;

main( )
{
        /* This is data describing the thread created */
        pthread_t   tid;
        pthread_attr_t   attr;
        char hello[ ] = {"Hello, "};
        char thread[ ] = {"thread"};

        /* Initialize the thread attributes */
        pthread_attr_init (&attr);

        /* Initialize the mutex (default attributes) */
        pthread_mutex_init (&my_sync,NULL);

        /* Create another thread. ID is returned in &tid */
        /* The last parameter is passed to the thread function */
        /* Note reversed the order of "hello" and "thread" */

        pthread_create(&tid, &attr, compute_thread, hello);

        sleep(1); /* Let the thread get started */
```

```
        /* Lock the mutex when it's our turn to do work */
        pthread_mutex_lock(&my_sync);
        printf(thread);
        printf("\n");
        pthread_mutex_unlock(&my_sync);
        exit(0);
}

/* The thread function to be run */
void* compute_thread(void* dummy)
{
        /* Lock the mutex when its our turn */
        pthread_mutex_lock(&my_sync);
        printf(dummy);
        pthread_mutex_unlock(&my_sync);
        sleep(1);
        return;
}
```

Condition Variables:

Overview

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

- A condition variable is always used in conjunction with a mutex lock.

- The typical sequence for using condition variables:
  Create and initialize a condition variable
  Create and initialize an associated mutex

- Define a predicate variable (variable whose condition must be checked)

- A thread does work up to the point where it needs a certain condition to occur (such as the predicate must reach a specified value). It then "waits" on a condition variable by:

  Locking the mutex
  While predicate is unchanged wait on condition variable
  Unlocking the mutex

- Another thread does work which results in the waited for condition to occur (such as changing the value of the predicate). Other waiting threads are "signalled" when this occurs by:
  Locking the mutex

Changing the predicate
Signalling on the condition variable
Unlocking the mutex

Creating / Destroying Condition Variables:

pthread_cond_init (pthread_cond_t condition, pthread_condattr_t attr)

pthread_cond_destroy (pthread_cond_t condition)

pthread_condattr_init (pthread_condattr_t attr)

pthread_condattr_destroy (pthread_condattr_t attr)

pthread_cond_init( ) creates and initializes a new condition variable object. The ID of the created condition variable is returned to the calling thread through the condition parameter.

Condition variables must be of type pthread_cond_t .

The optional attr object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type pthread_condattr_t (may be specified as NULL to accept defaults).

Currently, the attributes type attr is ignored in the AIX implementation of pthreads; use NULL.

If implemented, the pthread_condattr_init( ) and pthread_condattr_destroy( ) routines are used to create and destroy condition variable attribute objects.

pthread_cond_destroy( ) should be used to free a condition variable that is no longer needed.

Waiting / Destroying Condition Variables :

pthread_cond_wait (pthread_cond_t condition, pthread_mutex_t mutex)

pthread_cond_signal (pthread_cond_t condition)

pthread_cond_broadcast (pthread_cond_t condition)

pthread_cond_wait( ) blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits.

The pthread_cond_signal( ) routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked.

The pthread_cond_broadcast( ) routine should be used instead of pthread_cond_signal( ) if more than one thread is in a blocking wait state.

It is a logical error to call pthread_cond_signal( ) before calling pthread_cond_wait( ).

Example: Using Condition Variables:

Lab3.c:

/ Code for cond-hellothread.c */*

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

/* This is the initial thread routine */
void* compute_thread (void*);

/* This is the lock for thread synchronization */
pthread_mutex_t    my_sync;

/* This is the condition variable */
pthread_cond_t    rx;

#define  TRUE   1
#define  FALSE  0
/* this is the Boolean predicate */
int   thread_done = FALSE;

main( )
{
      /* This is data describing the thread created */
      pthread_t   tid;
      pthread_attr_t   attr;
      char hello[ ] = {"Hello, "};
      char thread[ ] = {"thread"};
      /* Initialize the thread attributes */
      pthread_attr_init (&attr);
      /* Initialize the mutex (default attributes) */
      pthread_mutex_init (&my_sync, NULL);
      /* Initialize the condition variable (default attr) */
      pthread_cond_init (&rx, NULL);
      /* Create another thread. ID is returned in &tid */
      /* The last parameter is passed to the thread function */
      pthread_create(&tid, &attr, compute_thread, hello);
      /* wait until the thread does its work */
      pthread_mutex_lock(&my_sync);
      while (!thread_done)
            pthread_cond_wait(&rx, &my_sync);
      /* When we get here, the thread has been executed */
      printf(thread);
      printf("\n");
      pthread_mutex_unlock(&my_sync);
      exit(0);
}
```

```
/* The thread to be run by create_thread */
void* compute_thread(void* dummy)
{
      /* Lock the mutex - the cond_wait has unlocked it */
      pthread_mutex_lock (&my_sync);
      printf(dummy);
      /* set the predicate and signal the other thread */
      thread_done = TRUE;
      pthread_cond_signal (&rx);
      pthread_mutex_unlock (&my_sync);
      return;
}
```

Lab4.c:

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

void *increment(void*);
void *watch(void*);
pthread_mutex_t  my_mutex;
pthread_cond_t  w_cond;
int count=0;

main()
{
      pthread_t tid1, tid2, tid3;
      pthread_mutex_init (&my_mutex, NULL);
      pthread_cond_init(&w_cond, NULL);

      pthread_create(&tid1,NULL,watch, NULL);
      pthread_create(&tid2,NULL,increment, NULL);
      pthread_create(&tid3,NULL,increment, NULL);

      pthread_join(tid1,NULL);
      pthread_mutex_destroy(&my_mutex);
      pthread_cond_destroy(&w_cond);
      printf("Parent is DONE ....\n");
      exit(0);
}


void* increment(void *dummy)
{
      while(count<10)
      {
            pthread_mutex_lock(&my_mutex);
            printf("Cound =%d by thread %d\n",++count,pthread_self());
            if(count==10)
            pthread_cond_signal(&w_cond);
            pthread_mutex_unlock(&my_mutex);
            sleep(1);
      }
      return;
}
```

```
void* watch(void *dummy)
{
      pthread_mutex_lock(&my_mutex);
      pthread_cond_wait(&w_cond,&my_mutex);
      pthread_mutex_unlock(&my_mutex);
      printf("Watch is DONE...\n");
      return;
}
```

Exercises

Note:

Lab Problems will be given during the lab based on material covered in this lab manual.