Experiment 5
Process / Threads Synchronization[1]

Objectives

- ✓ 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 To learn and practice how processes communicate among themselves
  - Semaphores

Prelab Activities

- ✓ Read the manual and try to do the experiment yourself before the lab.
- ✓ Write (copy and paste) and compile the codes given. Also, perform the exercises.

General Information

Why we need Synchronization and how to achieve it?
Suppose the multiple threads share the common address space (thru a common variable), then there is a problem.

```
THREAD A                      THREAD B
x = common_variable ;         y = common_variable ;
x++ ;                         y-- ;
common_variable = x ;         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.

1 **Ref-1:** M. Akbar Badhusha, King Fahd University of Petroleum and Minerals, Saudi Arabia.
Ref-2: College Of Engineering, Sasthamcotta

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 complexity of a condition variable predicate is defined by the programmer. A condition can be signaled by any thread to either one or all waiting threads.

Mutexes
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.

<u>Creating / Destroying Mutexes :</u>

```
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 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.

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

int x=1;

void* compute_thread(void * argument){
      printf("X value in thread before sleep = %d\n",x);
      printf("X value in thread is increment by 1 before sleep\n");
      x++;
      sleep(2);
      printf("X value in thread after sleep = %d\n",x);
      return;
}
void main(){
      pthread_t tid;
      pthread_attr_t attr;
      pthread_attr_init(&attr);
      pthread_create(&tid, &attr, compute_thread,(void *)NULL);
      sleep(1);
      x++;
      printf("Main thread incs X, after that X value is %d\n",x);
      pthread_join(tid,NULL);
      exit(0);
}
```

Program 1.

Compile and run Program 1. You will not get the expected output. If you run again, still the output may be different. To avoid this we use Mutex variables as shown in Program 2.

Example2:

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

int x=1;
/* This is the lock for thread synchronization */
pthread_mutex_t my_sync;

void* compute_thread(void * argument) {
      printf("X value in thread before sleep = %d\n",x);
      printf("X value in thread is increment by 1 before sleep\n");
      pthread_mutex_lock(&my_sync);
      x++;
      sleep(2);
      printf("X value in thread after sleep = %d\n",x);
      pthread_mutex_unlock(&my_sync);
      return;
}

void main(){
      pthread_t tid;
      pthread_attr_t attr;
      pthread_attr_init(&attr);

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

      pthread_create(&tid, &attr, compute_thread,(void *)NULL);
      sleep(1);
      pthread_mutex_lock(&my_sync);
      x++;
      printf("Main thread increments 1 to X, after that X value is
%d\n",x);
      pthread_mutex_unlock(&my_sync);
      pthread_join(tid,NULL);
      exit(0);
}
```

Program 2.

Condition Variables:

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 "signaled" when this occurs by:

```
Locking the mutex
Changing the predicate
Signaling 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( ).

Example3: Using Condition variables:

```
#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;
int x=1;

void main(){
     /* This is data describing the thread created */
     pthread_t tid;
     pthread_attr_t attr;
     /* 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 */
     x++;
     printf("Main thread incs X,after that X value is %d\n",x);
     pthread_mutex_unlock(&my_sync);
     exit(0);
}
/* The thread to be run by create_thread */
void* compute_thread(void* dummy){
     printf("X value in thread before sleep = %d\n",x);
     printf("X value in thread is increment by 1 before sleep\n");
     /* Lock the mutex - the cond_wait has unlocked it */
     pthread_mutex_lock(&my_sync);
     x++;
     sleep(2);
     printf("X value in thread after sleep = %d\n",x);
     /* set the predicate and signal the other thread */
     thread_done = TRUE;
     pthread_cond_signal(&rx);
     pthread_mutex_unlock(&my_sync);
     return;
}
```

Program 3.

Semaphore:

Semaphores are a programming construct designed by E. W. Dijkstra in the late1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter. In the computer version, a semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it.

Semaphore is a variable that can take only the values 0 and 1, binary semaphore. This is the most common form. Semaphores that can take many positive values are called general semaphores.

The definition of P and V are surprisingly simple. Suppose we have semaphore variable sv. The two operations are defined as follows:
P(sv) - If sv is greater than zero, decrement sv, if sv is zero, suspend execution of this process.
V(sv) - If some other process has been suspend waiting for sv, make it resume execution. If no process is suspended waiting for sv, Increment sv.

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

POSIX semaphore functions are:

`sem_open()` -- Connects to, and optionally creates, a named semaphore

`sem_init()` -- Initializes a semaphore structure (internal to the calling program, so not anamed semaphore).

`sem_close()` -- Ends the connection to an open semaphore.

`sem_unlink()` -- Ends the connection to an open semaphore and causes the semaphore to beremoved when the last process closes it.

`sem_destroy()` -- Initializes a semaphore structure (internal to the calling program, so not anamed semaphore).

`sem_getvalue()` -- Copies the value of the semaphore into the specified integer.

`sem_wait(), sem_trywait()` -- Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

`sem_post()` -- Increments the count of the semaphore

All ***POSIX*** semaphore functions and types are prototyped or defined in semaphore.h. To define a semaphore object, use `sem_t sem_name;`

To initialize a semaphore, use sem_init():

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem points to a semaphore object to initialize
- pshared is a flag indicating whether or not the semaphore should be shared with fork()ed processes. Linux Threads does not currently support shared semaphores
- value is an initial value to set the semaphore to

Example of use: `sem_init(&sem_name, 0, 10);`

To wait on a semaphore, use `sem_wait`:

```
int sem_wait(sem_t *sem);
```

`Example of use: sem_wait(&sem_name);`

- If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls sem_post.

To increment the value of a semaphore, use sem_post:

```
int sem_post(sem_t *sem);
```

`Example of use: sem_post(&sem_name);`

- It increments the value of the semaphore and wakes up a blocked process waiting onthe semaphore, if any.

To find out the value of a semaphore, use

```
int sem_getvalue(sem_t *sem, int *valp);
```

- gets the current value of sem and places it in the location pointed to by valp

Example of use:

```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

To destroy a semaphore, use

```
int sem_destroy(sem_t *sem);
```

- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use: `sem_destroy(&sem_name);`

Using semaphores - a short example

Consider the problem we had before and now let us use semaphores:

Declare the semaphore global (outside of any funcion):

```
sem_t mutex;
```

Initialize the semaphore in the main function:

```
sem_init(&mutex, 0, 1);
```

Thread 1	Thread 2	data
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b + 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2
[data is fine. The data race is gone.]		

The basic operation of these functions is essence the same as described above, except note there are more specialized functions, here.

```
/*Program to demonstrate the usage of semaphore variable in
controlling the access to specific resources.
This program contains two functions which display their own
messages.
The display is suitably controlled by the use of semaphore variable.
It is a program in which a semaphore variable is shared between the
main function and a thread function.
This module can be compiled using 'cc program4.c –o program4 –
lpthread' and executed './program4'*/

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

/*function prototype for the thread function.*/
void * display_function();

/*global semaphore variable to be shared.*/
sem_t sem_var;
int main(){
	pthread_t tid;
	pthread_attr_t attr;
	int i;
	/*initialising the semaphore variable with an initial value 0
(third arguement.)*/
	sem_init(&sem_var,0,0);

	pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
	pthread_create(&tid,&attr,display_function,NULL);

	for(i=0;i<5;i++){
		printf("(main)   Displaying:\t%d\n",i+1);
		/*waiting for the resources.*/
		sem_wait(&sem_var);
	}
	/*destroying the semaphore variable.*/
	sem_destroy(&sem_var);
}
void * display_function(){
	int i;
	for(i=0;i<5;i++){
		printf("(Thread) Displaying:\t%d\n",i+1);

		/*releasing resources.*/
		sem_post(&sem_var);
		sleep(1);
	}
}
```

Program 4.

Using semaphores - a short example - Serialisability Problem

* This program uses a variable 'data' whose value is used by two thread function.
* But the condition is that when a function uses it no other function should use it.
* This problem is tackled by defining two thread function and suitably making them accessed by the use of semaphore variable.
* This module can be compiled using 'cc program5.c –o program5 –lpthread' and executed './program5'

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
int data=0,end=0;
sem_t sem_var;
void *thread_function1(){
      int a;
      printf("\nEntered into thread function:1\n");
      printf("\nThread function1 waiting to gain access\n");
      sleep(1);
      sem_wait(&sem_var);
      printf("\nAccess gained by thread function:1\n");
      printf("\nThread function1 using the value of 'DATA' \n");
      a=data;
      a=a+1;
      data=a;
      sem_post(&sem_var);
      printf("\nResources released by thread function:1\n");
      end++;
}
void *thread_function2(){
      int b;
      printf("\nEntered into thread function:2\n");
      printf("\nThread function2 waiting to gain access\n");
      sleep(1);
      sem_wait(&sem_var);
      printf("\nAccess gained by thread function:2\n");
      printf("\nThread function2 using the value of 'DATA' \n");
      b=data;
      b=b+1;
      data=b;
      sem_post(&sem_var);
      printf("\nResources released by thread function:2\n");
      end++;
}
int main(){
      pthread_t t1,t2;
      pthread_attr_t attr;
      pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
      sem_init(&sem_var,0,1);
      pthread_create(&t1,&attr,thread_function1,NULL);
pthread_create(&t2,&attr,thread_function2,NULL);
      while(end!=2){}
      printf("\n\t****** The value of DATA is:\t%d ******\n",data);
}
```

Program 5.