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.

Exercise 1. To check the race condition while running two threads without synchronization

Create a directory called `posixsem` in your class Unix directory. Download in this directory the code `badcnt.c` and compile it using

```
gcc badcnt.c -o badcnt -lpthread
```

The program `badcnt.c` creates two new threads, both of which increment a global variable called `cnt` exactly `NITER`, with `NITER = 1,000,000`.

Run the executable `badcnt` and observe the ouput.
Quite unexpected! Since **`cnt`** starts at 0, and both threads increment it `NITER` times, we should see `cnt` equal to `2*NITER` at the end of the program.

But we see unexpected results. Find out the reason.

Reason :

Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space, like the variable `cnt` in our earlier example.

To understand what might happen, let us analyze this simple piece of code:

```
THREAD 1                    THREAD 2
a = data;                   b = data;
a++;                        b--;
data = a;                   data = b;
```

Now if this code is executed serially (for instance, `THREAD 1` first and then `THREAD 2`), there are no problems. However threads execute in an arbitrary order, so consider the following situation:

Thread 1	**Thread 2**	**data**
`a = data;`	`---`	0
`a = a+1;`	`---`	0
`---`	`b = data;  // 0`	0
`---`	`b = b + 1;`	0
`data = a;  // 1`	`---`	1
`---`	`data = b;  // 1`	1

So data could end up +1, 0, -1, and there is **NO WAY** to know which value! It is completely non-deterministic!

The solution to this is to provide functions that will block a thread if another thread is accessing data that it is using.

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 destroyedWhen 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)

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

```
1. Declare the ptread_mutex global (outside of any function):

            pthread_mutex_t mutex;

2. Initialize the mutex in the main function:

            pthread_mutex_init(&mutex,NULL);;
```

Thread 1	Thread 2	data
`pthread_mutex_lock (&mutex);`	---	0
---	`Pthread_mutex_lock (&mutex);`	0
`a = data;`	`/* blocked */`	0
`a = a+1;`	`/* blocked */`	0
`data = a;`	`/* blocked */`	1
`pthread_mutex_unlock (&mutex);`	`/* blocked */`	1
`/* blocked */`	`b = data;`	1
`/* blocked */`	`b = b + 1;`	1
`/* blocked */`	`data = b;`	2
`/* blocked */`	`Pthread_mutex_unlock (&mutex);`	2
[data is fine. The data race is gone.]		

Exercise 2.

Use the example above as a guide to fix the program `badcnt.c`, so that the program always produces the expected output (the value `2*NITER`).

Make a copy of `badcnt.c` into `goodcnt.c` before you modify the code.

To compile a program that uses pthreads *and* posix semaphores, use

```
gcc -o goodcount goodcount.c -lpthread
```

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 mutexAnother 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_cons_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_cons_wait( )*.

<u>Using Condition Variables:</u>

Exercise 3: /* 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 mutex(defaultattributes)*/

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;
}
```