**Ahmed Hassan Ismail – BESE-9B – 237897**

**OS Lab 7**

**Locks and pThreads**

**Tasks:**

**1 - Thread Creation and Termination:**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \*print\_message\_function( void \*ptr );

int main()

{

pthread\_t thread1, thread2;

const char \*message1 = "Thread 1";

const char \*message2 = "Thread 2";

int iret1, iret2;

/\* Create independent threads each of which will execute function \*/

iret1 = pthread\_create( &thread1, NULL, print\_message\_function, (void\*) message1);

if(iret1)

{

fprintf(stderr,"Error - pthread\_create() return code: %d\n",iret1);

exit(EXIT\_FAILURE);

}

iret2 = pthread\_create( &thread2, NULL, print\_message\_function, (void\*) message2);

if(iret2)

{

fprintf(stderr,"Error - pthread\_create() return code: %d\n",iret2);

exit(EXIT\_FAILURE);

}

printf("pthread\_create() for thread 1 returns: %d\n",iret1);

printf("pthread\_create() for thread 2 returns: %d\n",iret2);

/\* Wait till threads are complete before main continues. Unless we \*/

/\* wait we run the risk of executing an exit which will terminate \*/

/\* the process and all threads before the threads have completed. \*/

pthread\_join( thread1, NULL);

pthread\_join( thread2, NULL);

exit(EXIT\_SUCCESS);

}

void \*print\_message\_function( void \*ptr )

{

char \*message;

message = (char \*) ptr;

printf("%s \n", message);

}

**Output:**



**Explanation:**

In this code example, we are simply creating 2 separate threads. The function we have used to create threads is pthread\_create(). The pthread\_create() function is used to create a new thread, with attributes specified by attr, within a process. Then we use the pthread\_join() function to which is used to suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated. On return from a successful pthread\_join() call with a non-NULL value\_ptr argument, the value passed to pthread\_exit() by the terminating thread shall be made available in the location referenced by value\_ptr. The pthread\_exit() function terminates the calling thread and makes the value value\_ptr available to any successful join with the terminating thread.

Finally, we are using the printMessage function to print out the respective values to the console. The output of this code compiled using cc is shown above.

**2 - Mutexes:**

**Source Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \*functionC();

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

int counter = 0;

int main()

{

int rc1, rc2;

pthread\_t thread1, thread2;

/\* Create independent threads each of which will execute functionC \*/

if( (rc1=pthread\_create( &thread1, NULL, &functionC, NULL)) )

{

printf("Thread creation failed: %d\n", rc1);

}

if( (rc2=pthread\_create( &thread2, NULL, &functionC, NULL)) )

{

printf("Thread creation failed: %d\n", rc2);

}

/\* Wait till threads are complete before main continues. Unless we \*/

/\* wait we run the risk of executing an exit which will terminate \*/

/\* the process and all threads before the threads have completed. \*/

pthread\_join( thread1, NULL);

pthread\_join( thread2, NULL);

exit(EXIT\_SUCCESS);

return 0;

}

void \*functionC()

{

pthread\_mutex\_lock( &mutex1 );

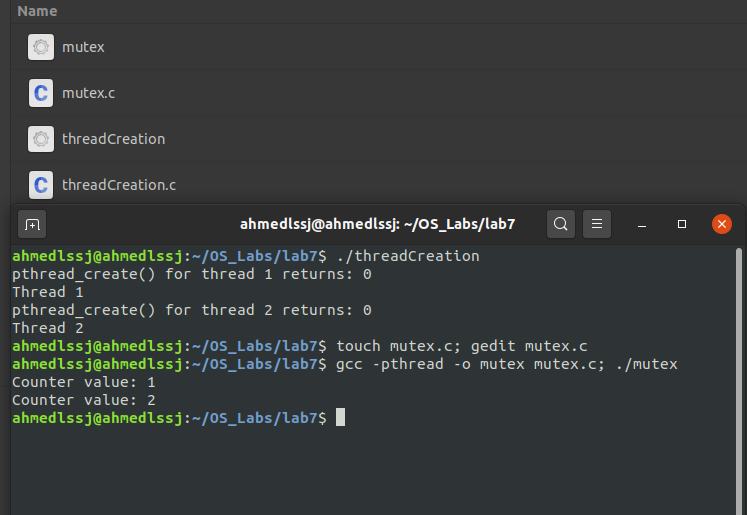
counter++;

printf("Counter value: %d\n",counter);

pthread\_mutex\_unlock( &mutex1 );

}

**Output:**



**Explanation:**

Mutexes are used to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to prevent race conditions where an order of operation upon the memory is expected. A race condition occurs when different threads try to access the same shared memory values leading to inconsistent and incorrect values of the shared variables.

Mutexes are used to prevent race conditions. Mutexes are used for serializing shared resources such as memory. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. In this way, mutexes protect the critical section of a thread from race conditions and inconsistent data values.

In this code example, we are creating 2 separate threads using the pthread\_create() function. The pthread\_join() function shall suspend execution of the calling thread until the target thread terminates, unless the target thread has already terminated. Finally, in the function we are using mutex\_lock() to block access to shared variables when a thread is accessing them. After accessing and incrementing the value of counter, we then use mutex\_unlock to remove this lock.

**3 - Joins:**

**Source Code:**

#include <stdio.h>

#include <pthread.h>

#define NTHREADS 10

void \*thread\_function(void \*);

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

int counter = 0;

int main()

{

pthread\_t thread\_id[NTHREADS];

int i, j;

for(i=0; i < NTHREADS; i++)

{

pthread\_create( &thread\_id[i], NULL, thread\_function, NULL );

}

for(j=0; j < NTHREADS; j++)

{

pthread\_join( thread\_id[j], NULL);

}

/\* Now that all threads are complete I can print the final result. \*/

/\* Without the join I could be printing a value before all the threads \*/

/\* have been completed. \*/

printf("Final counter value: %d\n", counter);

return 0;

}

void \*thread\_function(void \*dummyPtr)

{

printf("Thread number %ld\n", pthread\_self());

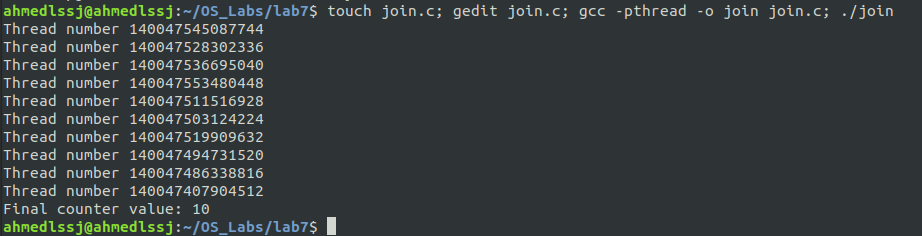
pthread\_mutex\_lock( &mutex1 );

counter++;

pthread\_mutex\_unlock( &mutex1 );

}

**Output:**



**Explanation:**

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One waits for the completion of the threads with a join.

In this code sample, we are creating multiple threads using pthread\_create() inside of a for loop. And we wait for the completion of the thread using pthread\_join() function. The value of each thread if printied to the console using the thread\_function() which also mutex\_locks() the critical section and increments the counter value before unlocking again. At the end, we print the value of the final count i.e. 10 to the console, since we have created 10 threads and have incremented the counter by 1 each time. The output is shown of this code sample is shown above.