**Name: Muhammad Saad Tariq**

**CMS: 224921**

**Class: BSCS-7A**

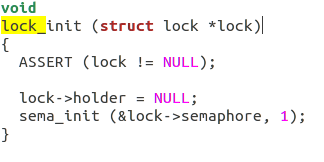
**Course: Operating Systems (CS-330)**

**Lab 5**

**Task 1**

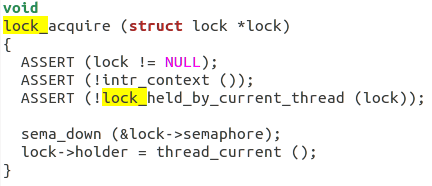
A lock is sort of a semaphore (a variable or abstract data type used to control access to a common resource by multiple processes) whose initial value is 1. A lock is affiliated with two major functions, a 'release' and an 'acquire'. Locks differ from semaphores in the way that only the thread that acquires a lock, called the lock's owner is allowed to release it. This property also determines whether a lock or semaphore should be used. Some of the lock functions used in PintOS include:

- **lock\_init**: This initializes a new lock, however the lock is not owned by any thread.



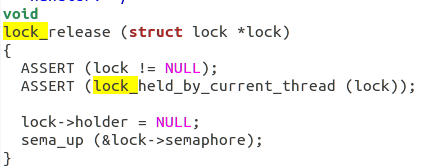
Uses the sema\_init() function which initializes sema as a new semaphore with the given initial value. This way creating a new lock and a semaphore to inform other threads.

- **lock\_acquire**: It acquires a lock for the current thread making it the sole 'owner' of the lock. This lock is not previously owned by any thread.



It uses the sema\_down() function which executes the "down" or "P" operation on sema, waiting for its value to become positive and then decrementing it by one. Thus, downs the semaphore and passes the acquired indication.

- **lock\_release**: This function releases the lock, which the current thread must own.



It uses the sema\_up() function which executes the "up" or "V" operation on sema, incrementing its value. If any threads are waiting on sema, wakes one of them up. This can signal other threads that the lock has been released.

Conclusion – These locks protect signals or condition variables that are supposed to be provided by other created threads.

**Task 2**

**Sample 1: “Thread Creation and Termination”**

Description

The following code demonstrates how to create and terminate threads using the pthread library, to allow multi-threading. The functions used are pthread\_create(), pthread\_join() and pthread\_exit() functions.

pthread\_create() - uses 4 parameters to create a new thread. These include the thread identity, its attributes, the function to call the thread to, and the argument to pass to the function.

pthread\_join() - waits for the current thread to terminate.

pthread\_exit() - terminates the current thread.

Running

- Save in a .c file extension e.g. <filename.c>

- Access the respective directory usig linux terminal

- Use command 'gcc <filename.c> -o <filename> -lpthread'

- Run executable file using command './<filename>'

Code

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

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

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 )

{

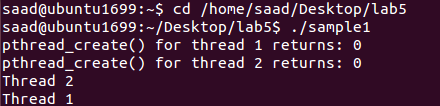
char \*message;

message = (char \*) ptr;

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

}

Output



**Sample 2: “Mutexes”**

Description

The following code implements the pthread library's mutex functions. Mutexes are basically used to block access to variables being used by a thread to other concurrent threads, until mutex is unlocked. The functions used include are pthread\_mutex\_lock(), pthread\_mutex\_unlock() and pthread\_mutex\_trylock().

pthread\_mutex\_lock() - acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

pthread\_mutex\_unlock()- remove lock from a mutex variable. An error is returned if mutex is already unlocked or owned by another thread.

pthread\_mutex\_trylock()- attempt to lock a mutex or will return error code if busy. Useful for preventing deadlock conditions.

Running

- Save in a .c file extension e.g. <filename.c>

- Access the respective directory usig linux terminal

- Use command 'gcc <filename.c> -o <filename> -lpthread'

- Run executable file using command './<filename>'

Code

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \*functionC();

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

int counter = 0;

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);

}

void \*functionC()

{

pthread\_mutex\_lock( &mutex1 );

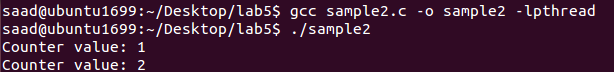
counter++;

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

pthread\_mutex\_unlock( &mutex1 );

}

Output



**Sample 3: “Joins”**

Description

Thread join function is used to wait for threads to finish. In the following code, after several threads were created, pthread\_join function is used to wait for completion of all threads before executing the next code line.i.e printing the final counter value.

Running

- Save in a .c file extension e.g. <filename.c>

- Access the respective directory usig linux terminal

- Use command 'gcc <filename.c> -o <filename> -lpthread'

- Run executable file using command './<filename>'

Code

#include <stdio.h>

#include <pthread.h>

#define NTHREADS 10

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

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

int counter = 0;

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);

}

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

{

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

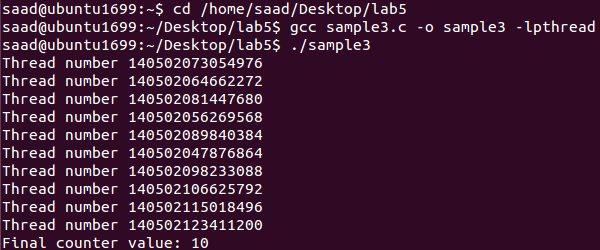
pthread\_mutex\_lock( &mutex1 );

counter++;

pthread\_mutex\_unlock( &mutex1 );

}

Output



**Sample 4: “Condition Variable”**

Description

A condition variable program allows a thread to suspend termination until a certain condition is fulfilled. The pthread\_cond\_t library functions are used along with mutex functions in order to avoid a race condition between threads where one thread waits before the previous thread signalling the wait condition.

pthread\_cond\_wait() - unlocks the mutex and waits for the condition variable cond to be signaled, taking 2 parameters – the signalling condition pointer and the mutex pointer.

Running

- Save in a .c file extension e.g. <filename.c>

- Access the respective directory usig linux terminal

- Use command 'gcc <filename.c> -o <filename> -lpthread'

- Run executable file using command './<filename>'

Code

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

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

pthread\_cond\_t condition\_var = PTHREAD\_COND\_INITIALIZER;

void \*functionCount1();

void \*functionCount2();

int count = 0;

#define COUNT\_DONE 10

#define COUNT\_HALT1 3

#define COUNT\_HALT2 6

main()

{

pthread\_t thread1, thread2;

pthread\_create( &thread1, NULL, &functionCount1, NULL);

pthread\_create( &thread2, NULL, &functionCount2, NULL);

pthread\_join( thread1, NULL);

pthread\_join( thread2, NULL);

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

exit(EXIT\_SUCCESS);

}

// Write numbers 1-3 and 8-10 as permitted by functionCount2()

void \*functionCount1()

{

for(;;)

{

// Lock mutex and then wait for signal to relase mutex

pthread\_mutex\_lock( &count\_mutex );

// Wait while functionCount2() operates on count

// mutex unlocked if condition varialbe in functionCount2() signaled.

pthread\_cond\_wait( &condition\_var, &count\_mutex );

count++;

printf("Counter value functionCount1: %d\n",count);

pthread\_mutex\_unlock( &count\_mutex );

if(count >= COUNT\_DONE) return(NULL);

}

}

// Write numbers 4-7

void \*functionCount2()

{

for(;;)

{

pthread\_mutex\_lock( &count\_mutex );

if( count < COUNT\_HALT1 || count > COUNT\_HALT2 )

{

// Condition of if statement has been met.

// Signal to free waiting thread by freeing the mutex.

// Note: functionCount1() is now permitted to modify "count".

pthread\_cond\_signal( &condition\_var );

}

else

{

count++;

printf("Counter value functionCount2: %d\n",count);

}

pthread\_mutex\_unlock( &count\_mutex );

if(count >= COUNT\_DONE) return(NULL);

}

}

Output

