**Formal Element**

**Thread Synchronisation**



Name: Tyrone Verburgt

Module: Operating Systems

Lecturer: Martin O’Hanlon

**Task 1**

Write a program that creates two threads that share a global variable. Have one thread loop (100,000 times) and add 1 to the global variable and print out the current value of the variable. The other thread does exactly the same except it adds two the value of the variable. Don’t use thread pthread\_join(thread, NULL) in the program.

#include<string>

#include<iostream>

#include<pthread.h> /\*POSIX trheads \*/

#include<unistd.h>

using namespace std;

int x,total = 0;

void \*thread\_routine1(void \*arg1)

{

for(int i = 0; i<100000; i++)

x += 1;

cout << x << endl;

}

void \*thread\_routine2(void \*arg1)

{

for(int i = 0; i<100000; i++)

x += 2;

}

int main()

{

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL,thread\_routine1,NULL);

pthread\_create(&thread2, NULL,thread\_routine2,NULL);

cout <<endl;

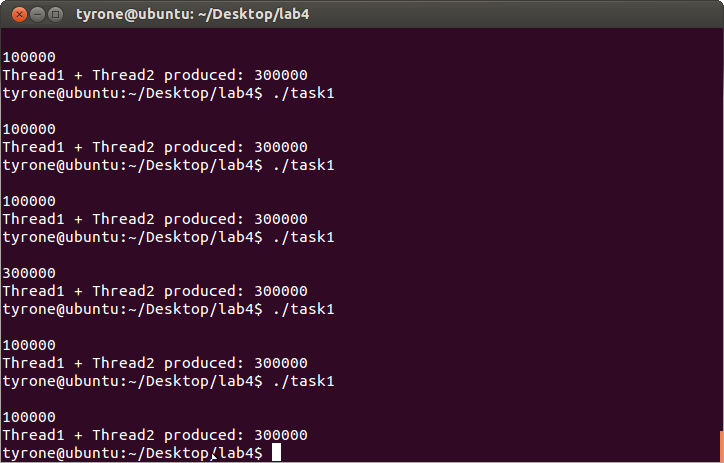
for(int k = 0; k<10000000; k++)

total = total +1;

cout << "Thread1 + Thread2 produced: " << x << endl;

pthread\_exit(NULL);

}



Comment on Results:

Observing the results, we can see that the program does not produce the consistent results we want. This inconsistency is due to a “race condition” between the threads. A race condition is when two threads try to simultaneously access and process the same memory. In a race condition a thread will sometimes block another thread in process. It will then do its processing and then allow the other thread to finish. This can have bad consequences like the example above.

In the program above the thread\_rountine1 is allowed to run first and then thread\_routine2. Thread\_routine1 prints out the value of the x global variable. One would think 100000 is always printed out, but that is not the case. We sometimes get 300000. The reason we get 3000000 is that the thread\_routine2 blocks thread\_routine1, does its processing then allows thread\_routine1 to finish.

**Task 2**

Use the mutex calls call on the the sheet to ensure correct synchronisation and consistent output for the code in Task1.

#include<string>

#include<iostream>

#include<pthread.h> /\*POSIX trheads \*/

#include<unistd.h>

using namespace std;

int x,total = 0;

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

int rc;

void \*thread\_routine1(void \*arg1)

{

//Locks the Thread.

pthread\_mutex\_lock(&amutex);

for(int i = 0; i<100000; i++)

x += 1;

cout << x << endl;

//Unlock thread so other thread can use the gobal variable x

pthread\_mutex\_unlock(&amutex);

}

void \*thread\_routine2(void \*arg1)

{

pthread\_mutex\_lock(&amutex);

for(int i = 0; i<100000; i++)

x += 2;

pthread\_mutex\_unlock(&amutex);

}

int main()

{

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL,thread\_routine1,NULL);

pthread\_create(&thread2, NULL,thread\_routine2,NULL);

cout <<endl;

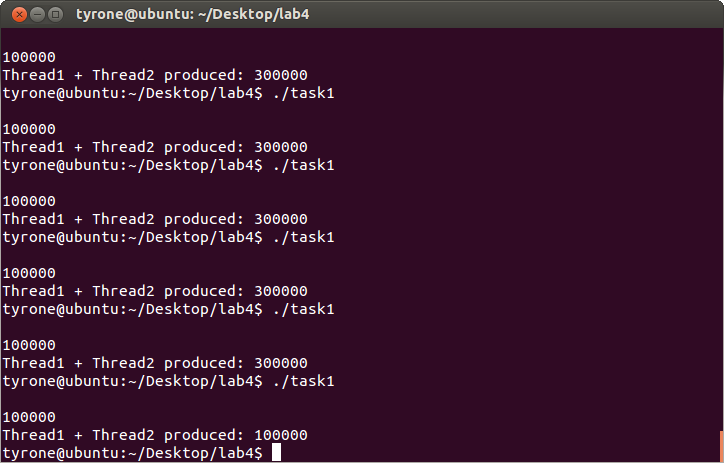
for(int k = 0; k<10000000; k++)

total = total +1;

cout << "Thread1 + Thread2 produced: " << x << endl;

pthread\_exit(NULL);

}



**Task 3**

Below is shown the last task from previous lab. It is not really possible to get this to work correctly without using Mutex variables. Add the required Mutex variables.

#include<string>

#include<iostream>

#include<pthread.h> /\*POSIX trheads \*/

#include<unistd.h>

using namespace std;

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

string message;

int x = 0;

void \*input\_routine(void \*arg1)

{

while(1){

pthread\_mutex\_lock(&mymutex);

cout <<"\nEnter a message: ";

getline(cin,message);

pthread\_mutex\_unlock(&mymutex);

}

}

void \*print\_routine(void \*arg1)

{

while(1){

pthread\_mutex\_lock(&mymutex);

cout <<"Message: "<<message<<endl;

pthread\_mutex\_unlock(&mymutex);

}

}

int main()

{

pthread\_t thread1;

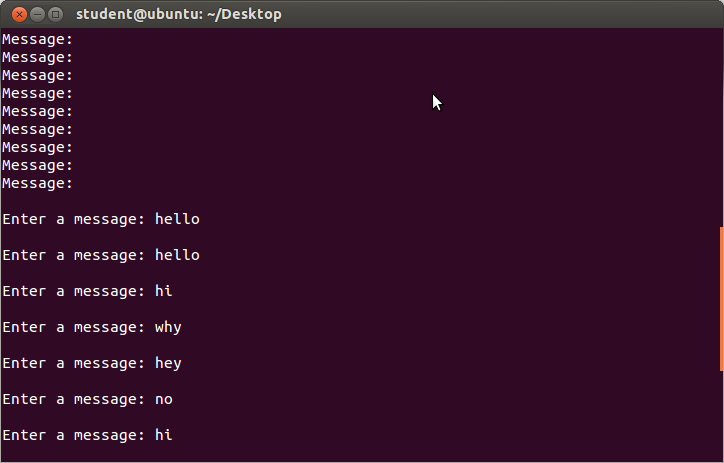
pthread\_t thread2;

pthread\_create(&thread1, NULL,input\_routine,NULL);

pthread\_create(&thread2, NULL,print\_routine,NULL);

pthread\_exit(NULL);

}



**Task 3B**

Demonstrate that your solution to Task 3 above is not efficient.

The reason the code above is not efficient is due to the fact that the Mutex variables cause other threads wait while one process it processing. The threads are not simultaneously working on one piece of memory. They are taking turns. In the while loops in both routines we have the Mutex variables that lock and unlock other threads from writing and reading the same memory. So while one routine is running, it is not possible for another thread to have a turn at memory because the loop keeps on locking the thread every time it goes around.

**Task 4**

Add condition variables to the code for Task 5 solution to ensure correct synchronisation and maximum efficiency. Demonstrate that your code is more efficient that the previous solution.

#include<string>

#include<iostream>

#include<pthread.h> /\*POSIX trheads \*/

#include<unistd.h>

using namespace std;

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

string message;

int x = 0;

void \*input\_routine(void \*arg1)

{

while(1){

pthread\_mutex\_lock(&mymutex);

if (x==0)

{

cout <<"\nEnter a message: ";

getline(cin,message);

x = 1;

}

pthread\_mutex\_unlock(&mymutex);

}

//pthread\_exit(NULL);

}

void \*print\_routine(void \*arg1)

{

while(1){

pthread\_mutex\_lock(&mymutex);

if(x==1){

cout <<"Message: "<<message<<endl;

x = 0;

}

pthread\_mutex\_unlock(&mymutex);

}

//pthread\_exit(NULL);

}

int main()

{

pthread\_t thread1;

pthread\_t thread2;

pthread\_create(&thread1, NULL,input\_routine,NULL);

pthread\_create(&thread2, NULL,print\_routine,NULL);

pthread\_exit(NULL);

}

