

Formal Element Thread Synchronisation



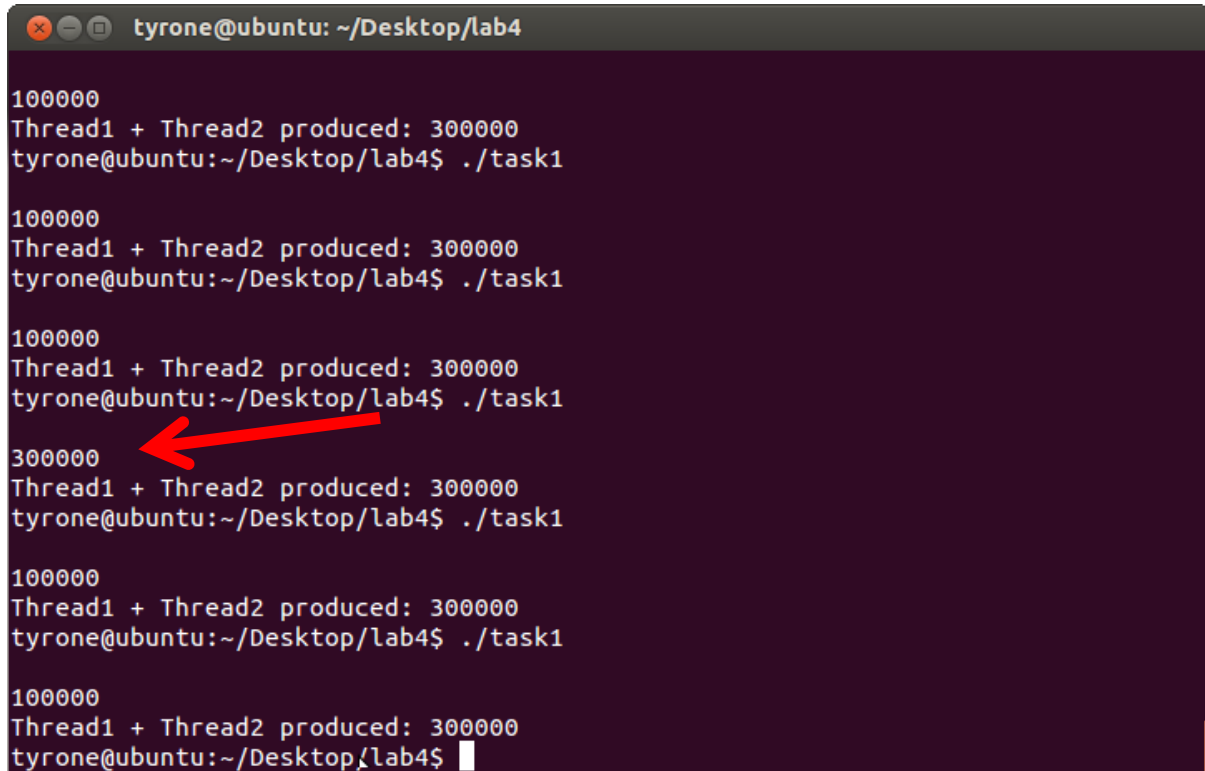
Name: Tyrone Verburt
Module: Operating Systems
Lecturer: Martin O'Hanlon



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);
}
```



```
tyrone@ubuntu: ~/Desktop/lab4
100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

300000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$
```

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.

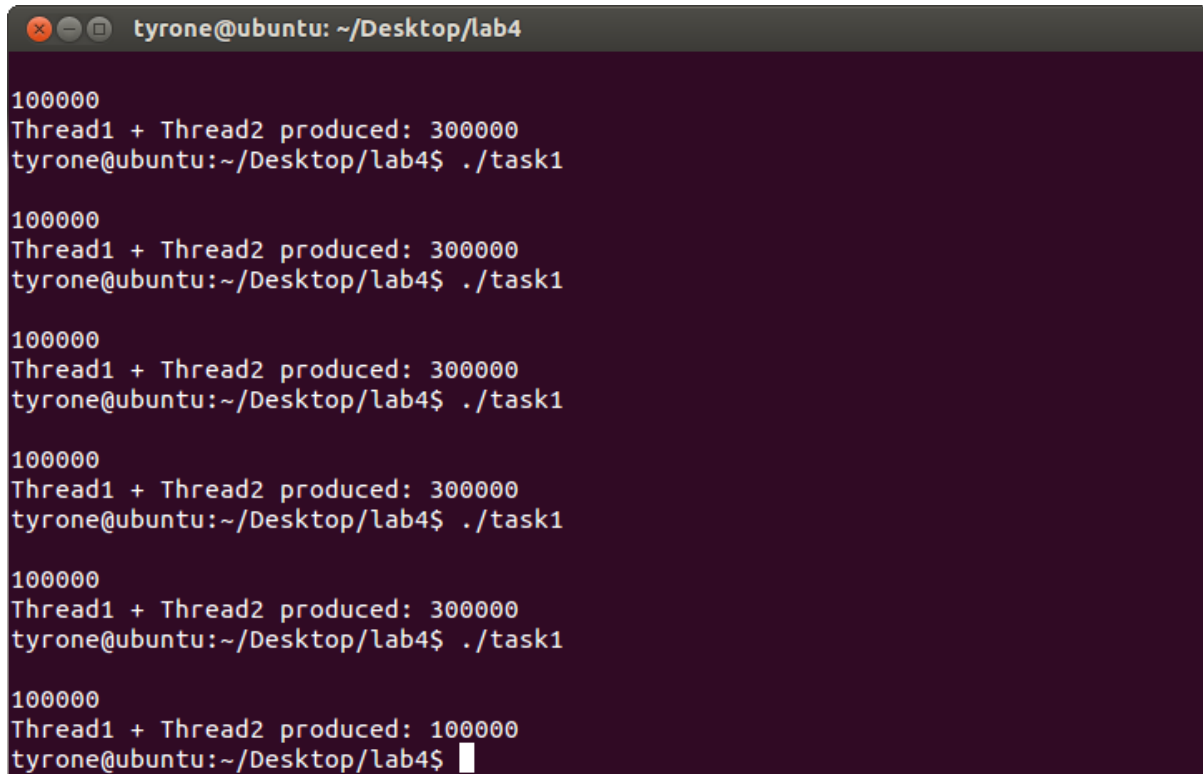
In the program above the thread_routine1 is allowed to run first and then thread_routine2. Thread_routine1 prints out the value of the x global variable. One would think 100000 is always printed out, but that is not the case. We sometimes get 300000. The reason we get 300000 is that the thread_routine2 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 threads */
#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);
}
```



```
tyrone@ubuntu: ~/Desktop/lab4

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 300000
tyrone@ubuntu:~/Desktop/lab4$ ./task1

100000
Thread1 + Thread2 produced: 100000
tyrone@ubuntu:~/Desktop/lab4$
```



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 threads */
#include<unistd.h>
using namespace std;

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
string message;
int x = 0;

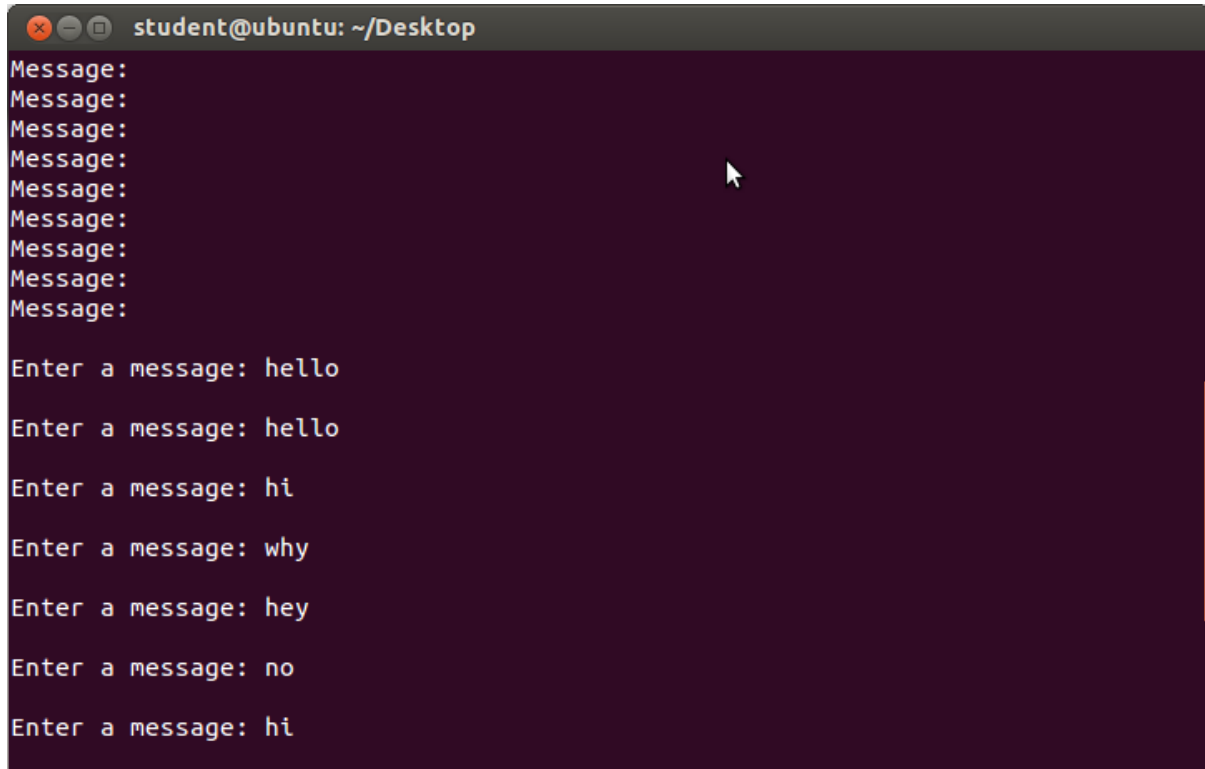
void *input_routine(void *arg1)
{
    while(1){
        pthread_mutex_lock(&mymutex);
        cout <<"\nEnter a message: ";
        getline(cin,message);
        pthread_mutex_unlock(&mymutex);
    }
}

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);
}
```

A terminal window titled 'student@ubuntu: ~/Desktop' with a dark purple background. It displays a sequence of 'Message:' prompts followed by user input. The inputs are: 'hello', 'hello', 'hi', 'why', 'hey', 'no', and 'hi'.

```
student@ubuntu: ~/Desktop
Message:
Message:
Message:
Message:
Message:
Message:
Message:
Message:
Message:
Message:

Enter a message: hello
Enter a message: hello
Enter a message: hi
Enter a message: why
Enter a message: hey
Enter a message: no
Enter a message: hi
```

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 than the previous solution.

```
#include<string>
#include<iostream>
#include<pthread.h> /*POSIX threads */
#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);
    }

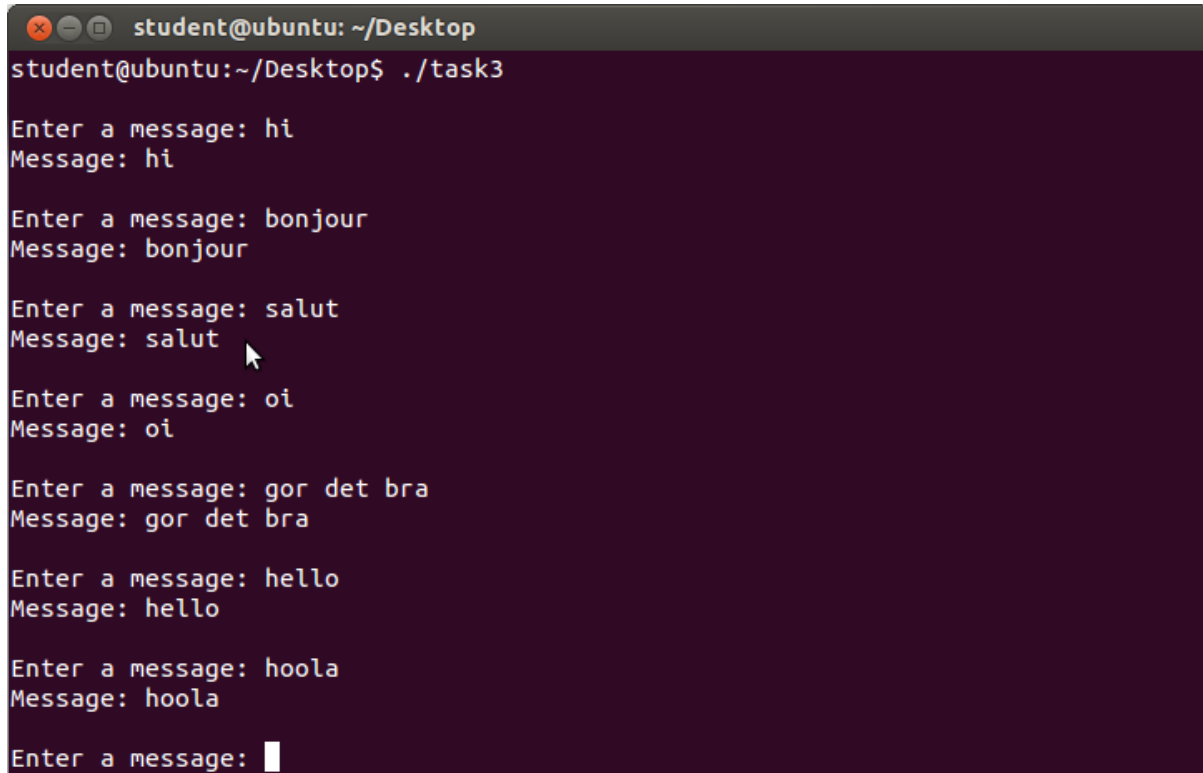
    //pthread_exit(NULL);
}

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);  
}
```



```
student@ubuntu: ~/Desktop  
student@ubuntu:~/Desktop$ ./task3  
  
Enter a message: hi  
Message: hi  
  
Enter a message: bonjour  
Message: bonjour  
  
Enter a message: salut  
Message: salut  
  
Enter a message: oi  
Message: oi  
  
Enter a message: gor det bra  
Message: gor det bra  
  
Enter a message: hello  
Message: hello  
  
Enter a message: hoola  
Message: hoola  
  
Enter a message: 
```