CS2106 Operating Systems
Lab 4
Introduction to Mutexes

INTRODUCTION

In this lab you will work with mutexes. A mutex is a special operating system data structure that allows only one process to enter the critical section at any time, preventing race conditions.

There are three activities in this lab:

> Activity 1: An introduction to mutexes, where you will learn how to write code that mutexes as part of the POSIX thread package.
>
> Activity 2: You will explore why having multiple threads writing to the same buffer is usually a bad idea.
>
> Activity 3: You will improve on your web server from Lab 3 to use mutexes to control access to the logging thread.

TEAM FORMATION

Please work in teams of 2 or 3. Single person submissions are not allowed.

SETUP

You will need your completed web server from Lab 3 to do Activity 3.

SUBMISSION INSTRUCTIONS

Please submit the answer book to your respective lab group's folder on IVLE by 2359 on Sunday 18 march 2018. Your answer book should be named ExxxxxxxA.docx, where ExxxxxxA is the student number of the team leader.

Ensure that every team member's name is recorded in the answer book. This lab totals 25 marks.

ACTIVITY 1. INTRODUCTION TO MUTEXES

In multi-threaded applications there are sections of code where no more than one thread can execute at a time. For example, code that updates global variables should not be executed by more than one thread or the updating may go wrong. Such sections of code are called "critical sections".

The word "mutex" stands for "Mutual Exclusion", and the idea here is that when a thread wants to enter a critical section the thread must first successfully obtain a lock called a "mutex". When it has the mutex, the thread can safely enter the critical section. Once it exits the critical section, the thread frees the mutex.

Only one thread can obtain this mutex, and other threads that are trying will block until the mutex is freed.

The mutex must be shared by several threads and must therefore be "global", and of type pthread_mutex_t. So to create a mutex call "my_mutex", the statement would be:

```
pthread_mutex_t my_mutex=PTHREAD_MUTEX_INITIALIZER;
```

This creates a new mutex lock, and initializes it to a default starting value. The functions below are used to manipulate the mutex:

Call	Description
int pthread_mutex_lock(pthread_mutex_t *mutex)	Locks the mutex. Blocks and does not return if mutex is already locked. If mutex locks successfully, this function returns with a 0. This function returns a non-0 value if something goes wrong.
int pthread_mutex_unlock(pthread_mutex_t *mutex)	Unlocks the mutex. Returns 0 if successful.
int pthread_mutex_destroy(pthread_mutex_t *mutex)	Destroys the mutex. Returns 0 if successful.

Type out the program below, calling it assg2p4.c.

```
#include <stdio.h>
#include <pthread.h>

int glob;

void *child(void *t)
{
      // Increment glob by 1, wait for 1 second, then increment by 1 again.
      printf("Child %d entering. Glob is currently %d\n", t, glob);
      glob++;
      sleep(1);
      glob++;
      printf("Child %d exiting. Glob is currently %d\n", t, glob);
}

int main()
{
      int i;
      glob=0;

      for(i=0; i<10; i++)
            child((void *) i);

      printf("Final value of glob is %d\n", glob);
      return 0;
}
```

Question 1 (1 mark)
What is the value of glob printed at the end of main?

Question 2 (3 marks)
Now modify main so that it spawns each call to "child" as a thread, giving us 10 threads in total. Describe the changes you made to the program.

Question 3 (3 marks)
Are the values of glob now correct? Explain why or why not.

Now modify your program as shown below. Statements in **bold underline** are newly added.

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

int glob;

void *child(void *t)
{
      // Increment glob by 1, wait for 1 second, then increment by 1 again.
      printf("Child %d entering. Glob is currently %d\n", t, glob);
      pthread_mutex_lock(&mutex);
      glob++;
      sleep(1);
      glob++;
      pthread_mutex_unlock(&mutex);
      printf("Child %d exiting. Glob is currently %d\n", t, glob);

      ... Other code you may have added ...
}

int main()
{
      int i, quit=0;
      glob=0;

      ... Codes and declarations that make your program multi-threading

      printf("Final value of glob is %d\n", glob);
      pthread_mutex_destroy(&mutex);
      ... Other code you may have added ...
}
```

Question 4 (3 marks)
Do your threads now update glob correctly? Explain your answer, and why the updates are correct/incorrect.

Question 5 (4 marks)

You will notice that the statement "printf("Final value of glob=%d\n", glob);" in main often executes before all the threads are done, causing the wrong value of glob to be printed. Modify your program so that this statement executes only after all threads complete.

Describe the modifications made, and cut and paste the code into your answer book. You will demo that your program works at the start of the next lab session.

ACTIVITY 2. SHARING BUFFERS BETWEEN THREADS

We will now look at the hazards of sharing a single buffer between threads. You are provided with two source files lab4p3.c and lab4p3fixed.c:

Step 1.

We begin first by seeing what our expected output should look like. Compile and run lab4p3fixed.c using:

```
gcc lab4p3fixed.c –pthread –o lab4p3fixed
./lab4p3fixed
```

Question 6 (2 marks)

Describe what you see on the screen.

Step 2.

Now compile and run lab4p3.c:

```
gcc lab4p3.c –pthread –o lab4p3
./lab4p3
```

Question 7 (2 marks)

Describe what you see on the screen.

Step 3.

Now open up lab4p3.c and lab4p3fixed.c. You will notice that lab4p3fixed.c uses mutexes. The differences in lab4p3fixed.c are circled in the next slide. You can verify against lab4p3.c that these lines are indeed missing.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

#define NUM_THREADS         32

void *writeSerialThread(void *p)
{
    // This thread simulates the behavior of a thread
    // communicating with another computer over a 115200bps
    // serial communications link.
    //
    // 115200 bits/sec = 14,400 bytes/s
    // There is therefore a delay of 70 microseconds between
    // each byte simulated with a 70 microsecond sleep.

    while(1)
    {
        if(len > 0)
        {
            int i;
            for(i=0; i<len; i++)
            {
                printf("%c", buffer[i]);
                usleep(70);
            }
            len=0;
            pthread_mutex_unlock(&mutex);
        }
    }

}

void *threadCode(void *p)
{
    int num = (int) p;
    int count=0;

    while(1)
    {
        pthread_mutex_lock(&mutex);
        sprintf(buffer, "This is thread %d iteration %d\n", num, ++count);
        len = strlen(buffer);
    }
}
```

Question 8 (4 marks)

Explain why lab4p3.c prints incorrectly, and how the use of mutexes in lab4p3fixed.c fix the problem.

ACTIVITY 3. PATCHING YOUR WEB SERVER

In Lab 3 we have a solution where all the web server threads write to a buffer, and a logging thread writes the contents of the buffer to a log file. In this lab you will find that it always works correctly.

This is because both the server threads and the logging thread execute fast enough to not get in each other's way. In addition the POSIX file write functions guarantee that data from multiple threads get written correctly.

In other words, "you were rucky".

If you ever had to modify your logging thread to write over a serial port (Google "Universal Asynchronous Receiver Transmitter" to find out more), particularly over a slow but common 9600 bit-per-second (bps) link, your ruck will run out.

Therefore in this activity you will modify your web server from lab 3 to ensure correctness (safety) of the logging operation:

Question 9 (3 marks)

Use what you have learnt above to modify your web server so that it logs safely. Cut and paste your modifications into your answer book and explain.

Note: This method of logging is still unsatisfactory. In the next lab we will learn how to properly write buffer code for use in multithreading systems.