**CS2106 Operating Systems**

**Lab 4**

**Introduction to Mutexes (ANSWER BOOK)**

|  |  |
| --- | --- |
| **STUDENT NUMBER 1** | **STUDENT NAME 1** |
| **STUDENT NUMBER 2** | **STUDENT NAME 2** |
| **STUDENT NUMBER 3** | **STUDENT NAME 3** |

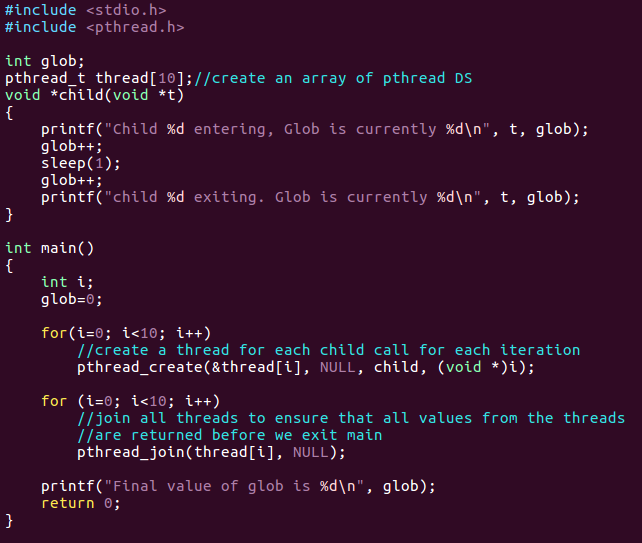
**Question 1** (1 mark)

The value of glob at the end of main is:

20

**Question 2** (3 marks)

My changes are:



Create an array of pthread DS of size 10. In the first for loop, create a thread for each child call for each iteration. Then in the second loop, join all threads to ensure that all values from the threads are returned before we exit main.

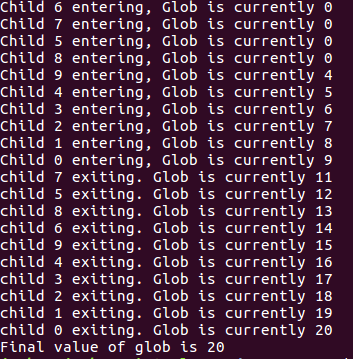
**Question 3** (3 marks)

The value of glob is not correct. This is why:

This is because all the threads are created at around the same time in the for loop (each iteration now creates a thread instead of calling the child method directly). Hence, the threads will be executed “concurrently”, while sharing the same memory space.

As there is a sleep of 1 second in each child method call, the threads are likely to be pre-empted by the scheduler in an indeterministic fashion.

A sample output is shown below:



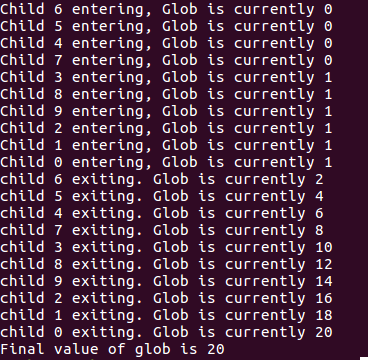
The lines are printed in two chunks.

**Question 4** (3 marks)

The value of glob is now not correct (at best half correct). This is why:

The first value printed by each thread is not correct as this statement is not included in the critical section. Hence, similar to the argument in Question 3, the glob printed for entering (first print) are affected by pre-emptions.

The first thread that reaches the pthread\_mutex\_lock() will lock the mutex. Subsequently, other threads reaching pthread\_mutex\_lock() will be blocked as the mutex is already locked by the first thread. Hence, at this juncture, only the first thread can execute the statements that come after pthread\_mutex\_lock() despite any pre-emptions. Eventually, this first thread will reach the pthread\_mutex\_unlock() which unlocks the mutex. The next thread (resumed by the scheduler) will execute pthread\_mutex\_lock() and lock the mutex for the second round. From here on, the cycle repeats until the last thread completes the execution of child.



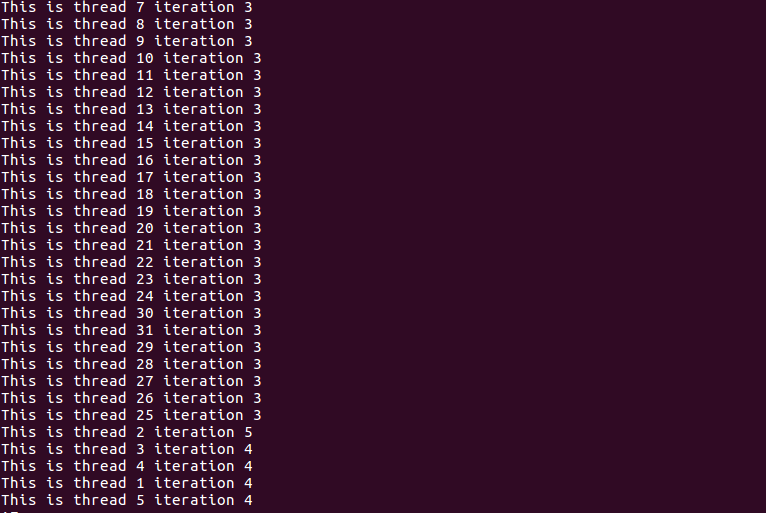
One chunk and one by one….

**Question 5** (4 marks)

Here are my modifications:

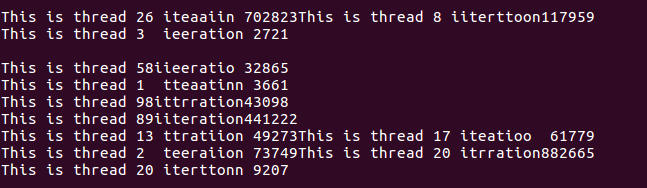
**Question 6** (2 marks)

I see the 32 threads printing “This is thread %d iteration %d\n” with correct incrementing count after each 32 threads printing the same count (in intervals of 70ms). However, there are some threads that print larger counts than the other in each round. For example, in the example below, thread 2 is already printing iteration count 5 while the rest is still printing iteration count 4.



**Question 7** (2 marks)

I see the threads trying to print “This is thread %d iteration %d\n” however with errors.



Some errors are wrong spelling of iteration, missing new line characters, missing whitespaces, wrong numbering of threads and wrong iteration counts.

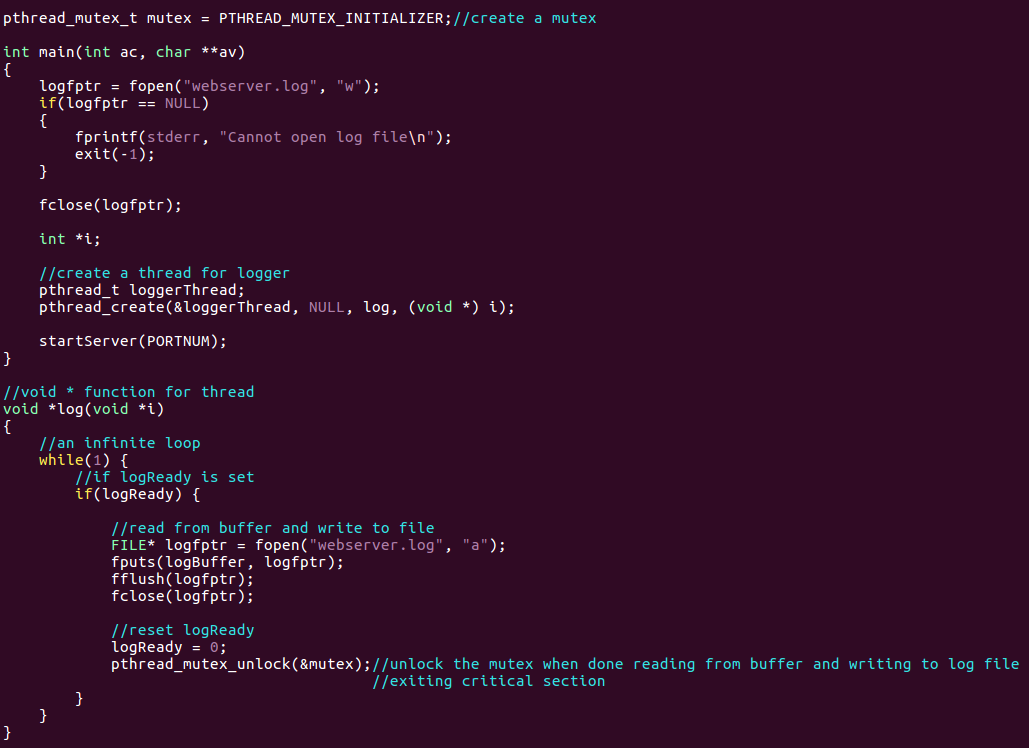
**Question 8** (4 marks)

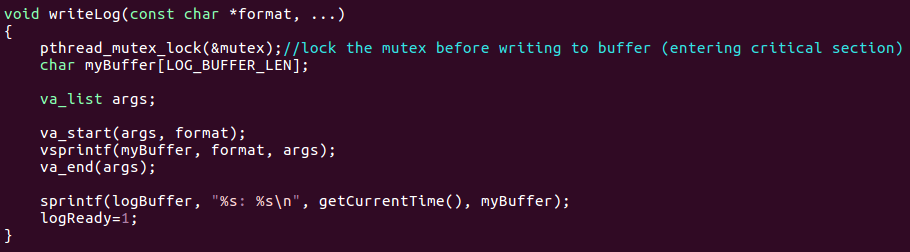
lab4p3.c prints incorrectly because of race conditions. Two or more threads are executing their critical section (writing and reading from the buffer) “at the same time”. This can happen when one of the threads has just loaded a value to a register and gets pre-empted. The next thread executes completely. When the first thread is resumed, it will overwrite the value of the latter thread.

The mutexes fix the problem because locking and unlocking the mutexes are atomic and will block the data and address buses. This prevents race conditions on the locks. Hence, this can effectively ensures that only one thread can access its critical section at one time, while other threads will wait for the former thread to finish executing the critical section and unlock the mutex before one of them can execute its critical section.

**Question 9** (3 marks)

My modifications are:





They work because:

First create a mutex as a global variable. Whenever when call writeLog, we first lock the mutex in the method body as writing and reading from buffer is the critical section where race conditions can occur (accessing same buffer memory). After we are done writing log to log file in \*log(), we then unlock the mutex as we are exiting the critical section. As such, only one thread can access the critical section and hence buffer and log file at one time, thus preventing race conditions that may corrupt results.

**TOTAL: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / 25**