**CS2106 Operating Systems**

**Lab 4**

**Introduction to Mutexes (ANSWER BOOK)**

|  |  |
| --- | --- |
| **STUDENT NUMBER 1**  **A0161868R** | **STUDENT NAME 1**  **Joshua Lee Kai Sheng** |
| **STUDENT NUMBER 2**  **A0161551M** | **STUDENT NAME 2**  **Ryan Tan Wei Keat** |
| **STUDENT NUMBER 3**  **A0155156E** | **STUDENT NAME 3**  **Ng Jun Wei** |

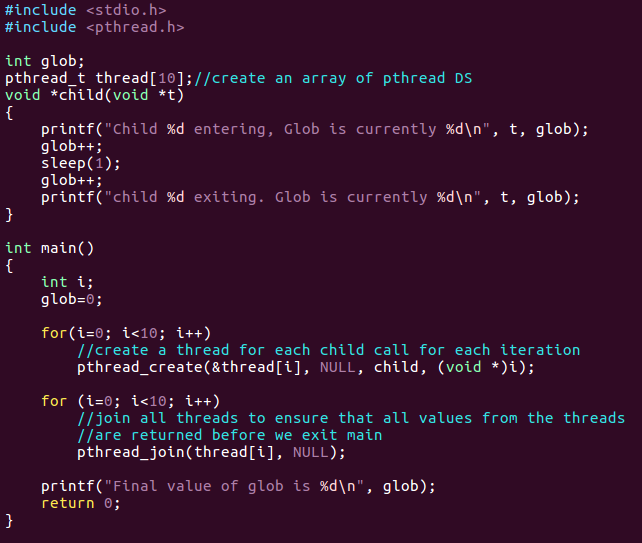
**Question 1** (1 mark)

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.**

**Inside main, create 2 for loops. 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.**

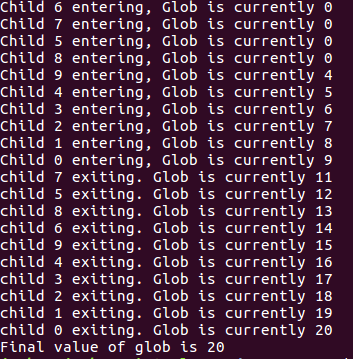
**Question 3** (3 marks)

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” due to multitasking, 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 before printing the second value of glob before exiting.**

**A sample output is shown below:**



**The lines are printed in two chunks that are separated by roughly 1 second (due to the sleep statement).**

**In the first chunk printed (first ten lines), the first few glob values are 0 as the threads load the same initial glob value of 0. This is because before the threads can increment the glob value, they are pre-empted, causing subsequent threads to load the same glob value of 0.**

**For the second chunk, each glob value increments by one starting from 11. This is not as what we expected, i.e. each glob value should increment by 2 as there are 2 glob++ statements in each thread. This is primarily due to the pre-emptions when the threads are sleeping. The sleeping period of 1 second allows all threads to have sufficient time to pre-empt one another and execute the first glob++. Hence, for the first thread to wake from the sleep, the glob value has already been incremented by 10. This thread will then increment it by 1, printing glob value of 11. Since, the second glob++ and printf are executed back to back, they are unlikely to be separated by pre-emptions. As such, each thread thereafter increments the glob value by 1 and prints it. This is not as what we expected.**

**Question 4** (3 marks)

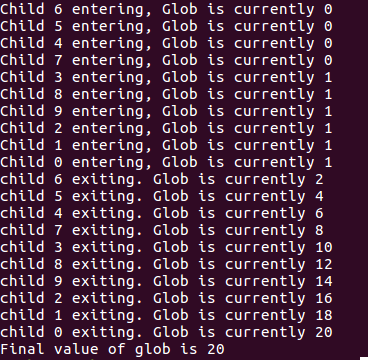
The value of glob is now **not correct (or half correct)**. This is why:

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

**The second value printed by each thread is correct (incrementing by 2). The explanations are as follows:**

**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. In the critical section, this thread will increment the glob value by 2. Eventually, this first thread will reach the pthread\_mutex\_unlock() which unlocks the mutex. Since the second printf follows immediately after, it is unlikely to be affected by pre-emptions. Hence, the first thread will print a glob value that is incremented by 2. The next thread (resumed by the scheduler) will execute pthread\_mutex\_lock() and lock the mutex for the second round. The glob value read by this thread is the already updated glob value of 2. Hence, this thread will correctly increment the glob by 2 to 4 and print the correct glob value of 4 eventually. From here on, the cycle repeats until the last thread completes the execution of child.**

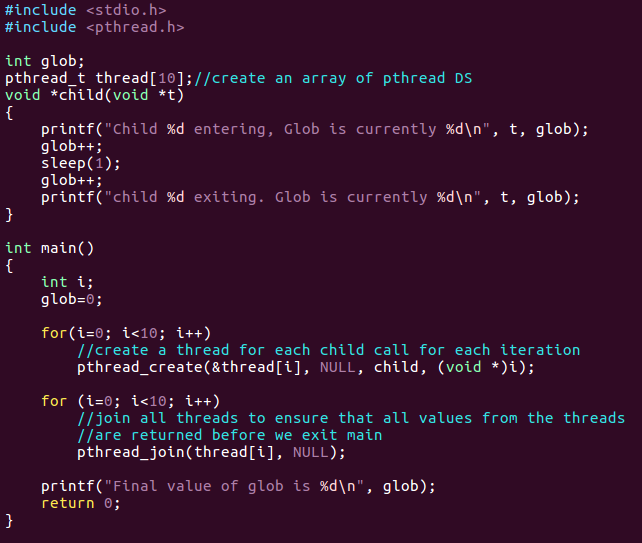
**A sample output is shown below:**



**The lines are printed firstly as a chunk of first ten lines, then line by line with an interval of 1 second in between. As we have explained above, the glob values in the first ten lines are wrong due to pre-emption as the first printf is not included in the critical section. However, for the second half, all the values printed are correct as each thread will take turn to increment the glob value twice in the critical section without any other threads entering this critical section (mutex). Hence, each subsequent glob value printed is incremented by 2, which is what we expected.**

**Question 5** (4 marks)

**We have already included a for loop with pthread\_join() for each thread in our initial implementation (as shown below). This is because had we not done so, the main will exit even before all the threads return, which will give us a final glob value of ≈0, as we don’t block the execution to wait for the threads to return. This necessitates pthread\_join.**

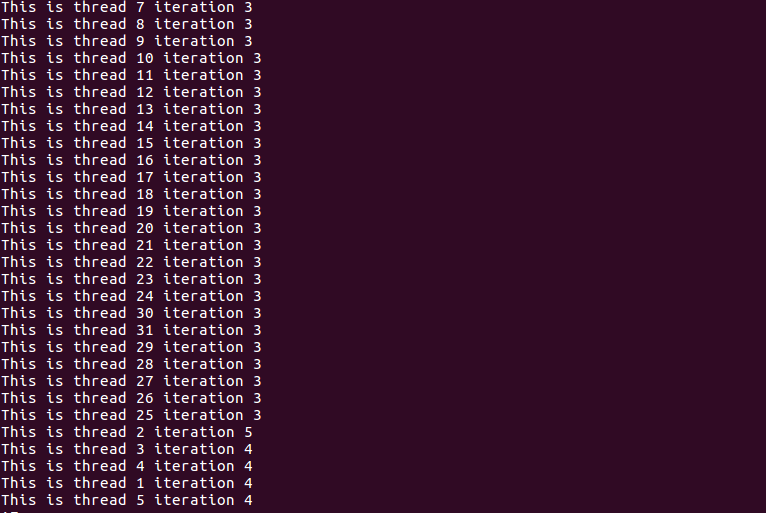


**Another more primitive way to solve this problem is of course to include an infinite while loop as the last statement in main() to prevent it from exiting.**

**Question 6** (2 marks)

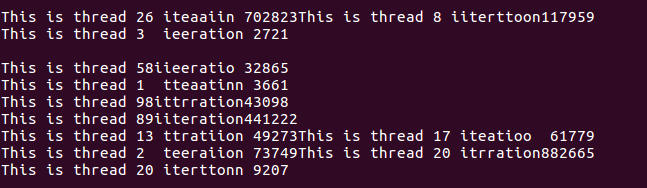
**I see the 32 threads printing “This is thread %d iteration %d\n” with correct incrementing counts (in intervals of 70ms) after all 32 threads have printed the same count in the previous round.**

**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”, but with many errors.**



**Some of the errors include 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 is due to multitasking and pre-emptions.**

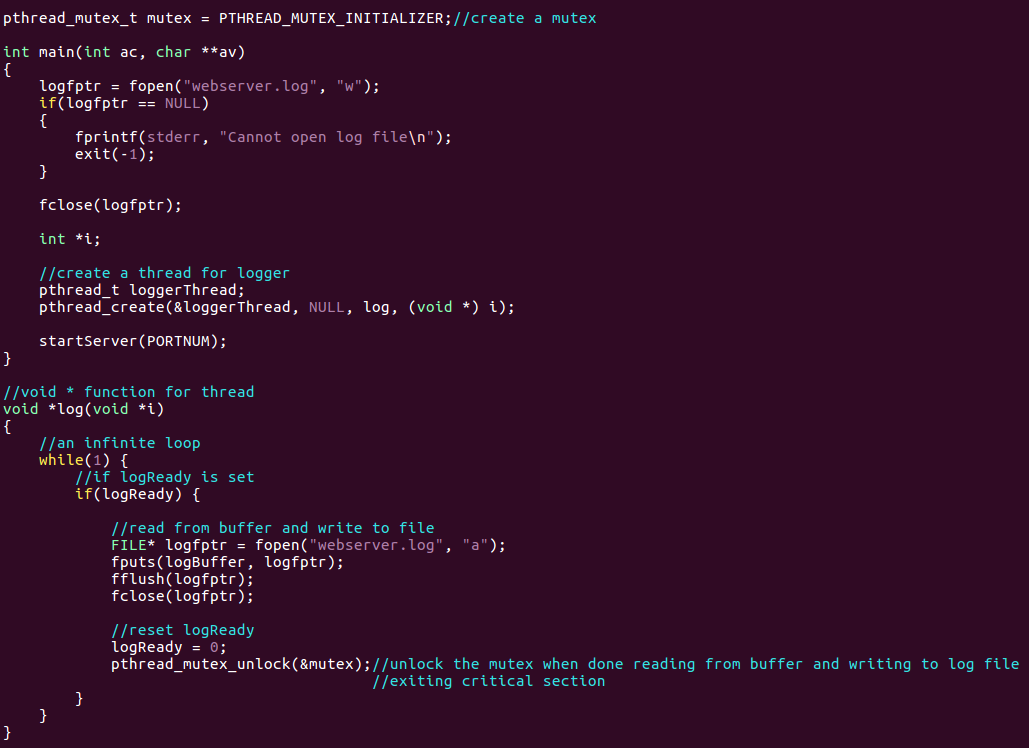
**An example of how this can happen is when one of the threads is in the middle of updating the buffer in \*threadCode() and gets pre-empted. Since the buffer is declared as a global variable, it can be accessed by the next thread as threads share memory space. Hence, the next thread will write to the same buffer (with the values from the previous thread unflushed) and eventually print the wrong buffer value in \*writeSerialThread().**

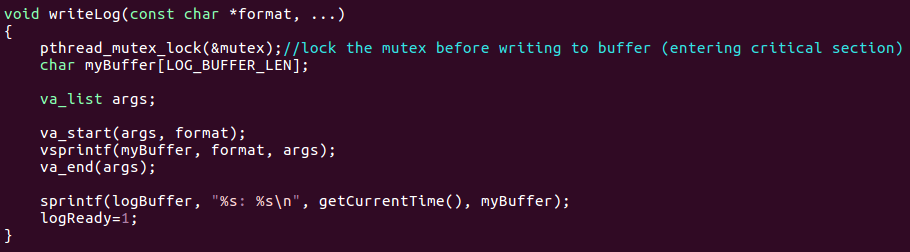
**The mutexes fix the problem because the locking and unlocking of the mutexes are atomic and will block the data and address buses. This prevents race conditions on the locks and enforce a “queue process”. Hence, this can effectively ensure that only one thread can access its critical section at one 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.**

**As such, the threads now behave synchronously, in the sense that all values are correctly written to and read from the buffer and the \*writeSerialThread prints out the correct message, as this whole process is now placed inside a critical section guarded by mutex.**

**Question 9** (3 marks)

My modifications are:





They work because:

**First create a mutex as a global variable. Whenever we 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 shared buffer memory). After we are done writing to log file in \*log() and having the buffer contents flushed, we then unlock the mutex as we are exiting the critical section.**

**As such, only one thread can access the critical section, hence buffer and log file, at one time. This prevents race conditions that may corrupt the logs written to the log file.**

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