Mikhail Andreev

EC527: Assignment 5

**Task 1: test\_generic.c**

Using the following code, ArrayA can print the float values.



**Task 2: test\_create.c base code**

The output from the base code is given below. The id’s on the thread have definitely changed:







**Task 3: change id to be a pointer**

Using the following code, this is achieved:



**Task 4: sleep(3)**

Code for work:



The output is a single line stating:

main() after creating the thread. My id is 140664013453056

with different id values. The child threads do not print anything. This is happening because the main thread terminates before the child threads reach the print statements. This causes them to also be terminated.

**Task 5: sleep(3) in main**

Code has been changed to:



This causes all the thread statements to be printed, as well as the main thread statement. Execution terminates 3 seconds after the last statement. This happens because all the threads now have time to finish executing.

**Task 6: sleep with join**

The code has been changed to the following:



The output starts by outputting the main thread statement. Then after a 3 second delay, all the child threads output their print statements. After this the main thread outputs its final statement and terminates.

This occurs because since the main thread blocks while the child threads are running, it has to wait for the threads to complete before finishing and joining. This causes the full body of the child threads to be executed, resulting in the delays and prints.

**Task 7: passing parameter**

Using the code:



We get the output:



Clearly, in this case the output is the same regardless of what the pointer is cast to.

If we change the code use the signed char:



The code does compile and the output is:

Hello World**!** It's me, MAIN!

In main**:** creating thread 0

In main**:** creating thread 1

**-**4

Hello World**!** It's me, thread # fffffffffffffffc !

In main**:** creating thread 2

**-**4

Hello World**!** It's me, thread # fffffffffffffffc !

In main**:** creating thread 3

**-**4

Hello World**!** It's me, thread # fffffffffffffffc !

In main**:** creating thread 4

**-**4

Hello World**!** It's me, thread # fffffffffffffffc !

**-**4

Hello World**!** It's me, thread # fffffffffffffffc !

It's me MAIN -- Good Bye World!

The variable is printed out correctly, but when converting to an unsigned long, we get the unsigned representation of the variable instead.

**Task 8: print out**

The printout will sometimes output messages in different orders, just based on what manages to finish processing first. However, the content is the same.

**Task 9: changing variables**

Without any changes we have:



Changing f in the code with:



We get the output:



Changing the code to modify g, we have:



The output after this is:



As more threads are created, segmentation faults start to appear. This occurs because the changes to the loop variable eventually propagate down to the main thread which is still creating threads. If the loop variable exceeds the length of the pthread\_t array, a segmentation fault will occur. This also causes less than NUM\_THREADS threads to be created.

**Task 10: passing an array**

Using the following code we can pass an array to work:



This gives the output:



As can be seen the values increment. The way this happens is because we pass the pointer to the array with an offset to the correct location. This method will work as long as the array is continuous.

**Task 11: passing a struct**

The output from the given code is:



Using this code we can make a 6th thread:



Which gives us the output:



**Task 12:**

Output without changes:



After commenting out the lock, we have the output:



In both cases the s is the character entered. So in the first case, we had to wait to enter the character to pass the lock. In the second case the child thread ignores the lock and just runs, then we have to enter the character to finish the main thread.

**Task 13:**

After changing the thread count, the output from the main thread no longer responds and everything is blocked. This indicates having multiple threads on the same mutex does not work.

**Task 14:**

Code output before edits verify that all before barrier statements occur prior to after barrier prints:

After sleep is uncommented, the print statement sequence is still the same, however, the statements are outputted in stages where each second 1 of 3 statements are outputted for each thread:



After changing the sleep time to tasked we have the lower id threads finishing execution much faster than higher id threads:



**Task 15:**

Output without any changes:



The main thread waits to join the remaining threads because a lock is set to wait for the 7th thread to unlock before unlocking, which causes the join statements to wait.

**Task 16:**

Code that adds 8th thread:



**Task 17:**

Under normal code execution the balance is the same as the start. However, when upgrading to 10000 threads causes the balance to get skewed. The reason the normal code is fine is because the accesses to the shared variable are not frequent enough to trigger corruption. However, when the number of threads increases, this causes an error.

**Task 18:**