**Task 2**

The output changes every time.

**Task 4**

When sleep is added, the print statement in the work function is not executed. This is because the sleep function causes the thread being executed to sleep until 3 seconds have passed, and this causes the main function to complete before any of the threads are completed.

**Task 5**

After the sleep function is moved, the print statements work as normal, but before the program finishes executing there is a 3 second delay.

**Task 6**

It changes as the “After creating the thread..” statement gets printed first, and then the work function print statements get printed out. This is different from Task 4 where the print statements in the work function are skipped entirely. The statements get printed out in this case because the pthread\_join function ensures each thread completes, before proceeding to the next one.

**Task 7 & 8**

Changing the type to signed char does not print the threadid, but rather just a NULL value. Every time the program is run the value printed out for threadid is NULL.

**Task 9**

When you change f (I changed it to 100) , the change is visible for all f’s. When I change \*g to 100, the change is visible in all \*g’s and the output is the same for all runs. One difference is that when f is changed to 100, the final i printed in the last sentence does not change and stays equal to 10, however when \*g is changed to 100 the final i printed changes to 100, since the variable stored in the memory location itself is changed.

I changed pthread\_t id[NUM\_THREADS] to pthread\_t id[NUM\_THREADS-2], so fewer than NUM\_THREADS threads are created.

**Task 10**

Added a global int variable- global which was set to 0. Global was incremented by 1 for each thread, so each thread got a unique value from the array. Then each element of the array was multiplied by 10. However, this method does not seem to reliably work, since some threads ended up receiving the same array element, since those threads received the same Global variables at the same time. However, when the final array was printed, where each thread multiplied each element by 10, the array printed out perfectly with each element multiplied by 10.

**Task 12**

The thread completes before mutex is unlocked.

**Task 13**

It no longer produces an output – syncing multiple threads with one mutex lock causes issues.

**Task 14**

After adding the sleep function, the prints before barriers still occur before the prints after barriers. Hence the barrier still works. However, one change adding the sleep function makes is that all the threads get created before any printing occurs.

I used an argument i\*taskid for the sleep function. The barrier still works since the print after barrier statements occur after print before barrier statements.

**Task 15**

The pthread join loop is executed after threads have executed, since there is still a mutex lock for taskid 7 before the join loop which gets unlocked after all the other taskids have unlocked.

**Task 17**

The balance is correct, without changing any code. However when a sleep timer is added to the PrintHello function before changing the balance, the balance at the end comes out to 999. When the number of threads is changed to 10,000, without the sleep function, the balance still comes out correct.

**Task 19**

No, the answer is not correct for 1000 iterations. It takes roughly 1000000 iterations to get close to the steady state values. The timer records time close to 2.33s.

When Barriers was set to 1, the values being printed got close to the steady state after about 10000 iterations which is considerably lower than the number of iterations taken when no barriers were used. The time taken for this was 0.56s, which is again lower than the time taken when no barriers were used. Using barriers has a reduced time & iterations because the barriers ensure that all the threads complete for each iteration, i.e. each element in the array is updated in each iteration before moving on to the next iteration. Fewer iterations are needed to reach steady state, as extra iterations are not needed so elements are edited based on updated neighboring values. Using barriers also gave similar answers consistently, as the elements updated in each iteration are consistently the same, whereas when no barriers are used, the neighboring elements being used to update the current element could be from any previous iteration, and the current element and the neighboring elements may not be aligned in terms of the iteration each is on.