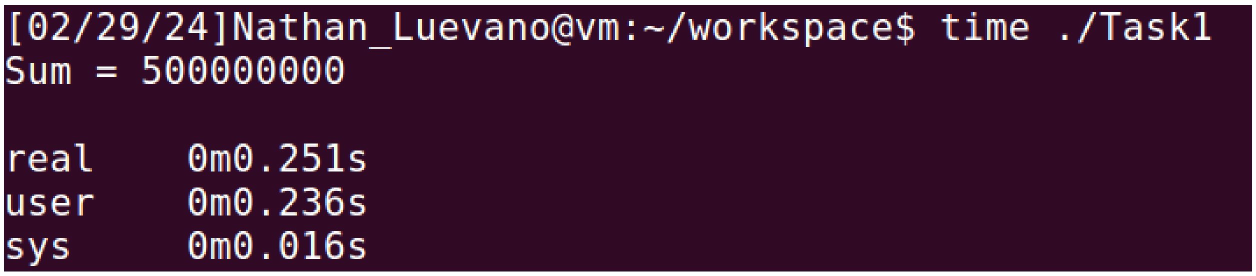
Thread Synchronization Lab

Nathan Luevano

Task 1: Running a Single Thread Counting Program



```
[02/29/24]Nathan_Luevano@vm:~/workspace$ time ./Task1
Sum = 500000000

real    0m0.251s
user    0m0.236s
sys     0m0.016s
```

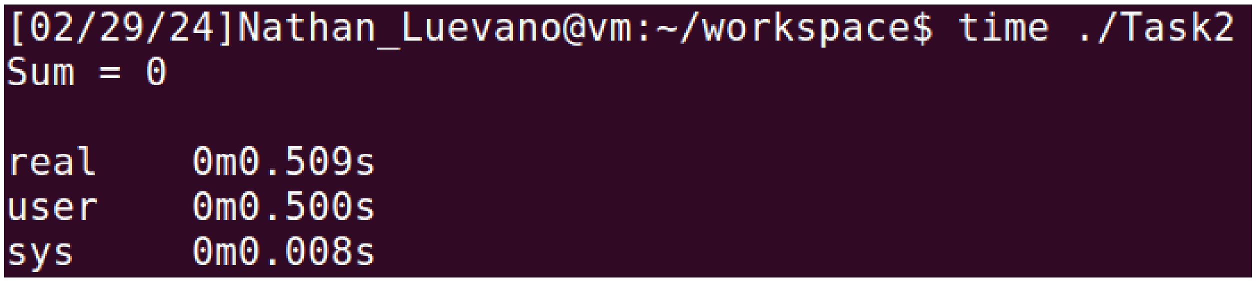
Code Breakdown:

- #include <stdio.h> is to include standard input and output library
- #define NUM_LOOPS 500000000 creates a macro that allows us to use it as a constant through the code. Defines the number of iterations for the loop.
- Long long sum = 0; this is a global variable which is a long long integer, this will hold a large value
- The function counting_function(int offset) takes an offset as the argument and runs a for loop that iterates the number of times that is specified in NUM_LOOPS. It increments the sum by the offset each time.
- Main is just the entry point of the program where the function is called and counted and then printed once done
- Return 0 is just a standard for if the program ran correctly or not

Screenshot:

- I ran 'time ./Task1' where Task1 is the compiled code for the task and time is Linux function to measure the execution time for the process. The sum = 500000000 is an indication that the program ran smoothly and iterated till it hit 500000000.

Task 2: Doubling the Length of a Single Thread Program



```
[02/29/24]Nathan_Luevano@vm:~/workspace$ time ./Task2
Sum = 0

real    0m0.509s
user    0m0.500s
sys     0m0.008s
```

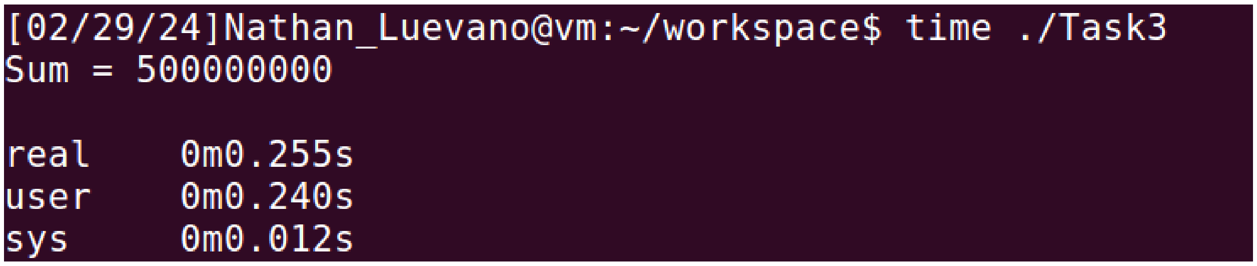
Code:

- This is the same code as Task 1 but I added another function call to count down once it counts up to 500000000

Screenshot:

- The sum comes to 0 which is correct according to the programs logic
- The time it took seems to be double the time it took to just count up to 500000000 which also seems to comply because it should take the about the same time to count and then back down.

<u>Task 3</u>: Counting Up With a Thread



```
[02/29/24]Nathan_Luevano@vm:~/workspace$ time ./Task3
Sum = 500000000

real    0m0.255s
user    0m0.240s
sys     0m0.012s
```

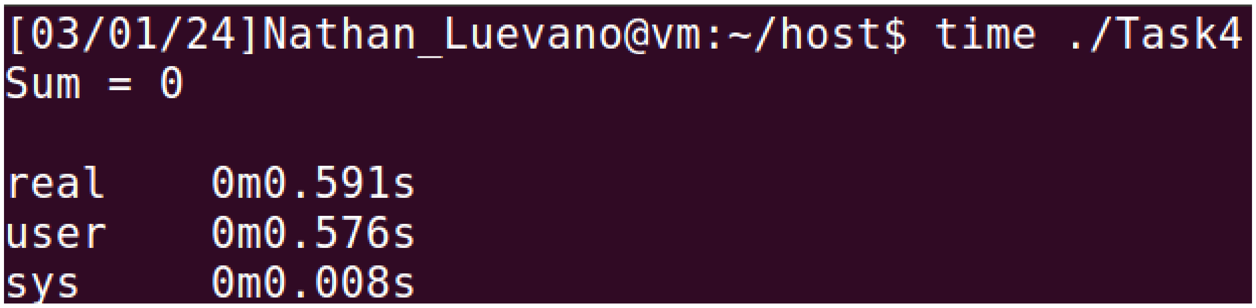
Code:

- In this task I included the POSIX thread library, #include <pthread.h>, which will allow me to do multi-threading in the program
- I had to modify the function counting_function, for now it returns a void* and takes only one parameter, this is because we are creating a thread to run the function instead of just calling it
- Through the use of pointers I casted the argument to int* which is then dereferenced, meaning I got the memory location pointed by the pointer, to get the offset value
- The pthread_creat() makes a new thread running the counting_function with the argument &offset1

Screenshot:

- The time takes to iterate up to 500000000 is about the same as in task one if anything a tiny bit slower.
- The difference between this iteration up and Task1's iteration up is that here we used a thread to count

<u>Task 4</u>: Counting Up and Down With a Thread



```
[03/01/24]Nathan_Luevano@vm:~/host$ time ./Task4
Sum = 0

real    0m0.591s
user    0m0.576s
sys     0m0.008s
```

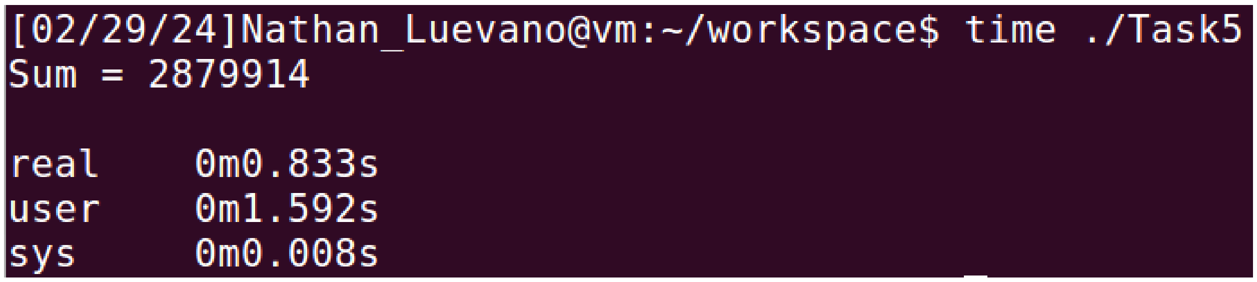
Code:

- In this code I created a second thread, id2, with an offset of -1
- The first thread will increment the sum, and the second thread will decrement it
- Both threads are not concurrently executing which gives better accuracy

Screenshot:

- As opposed to task 2 we are using 2 thread to count up and down
- The time it takes to execute is almost the same as it was in task 2

Task 5: Running Two Coinciding Threads



```
[02/29/24]Nathan_Luevano@vm:~/workspace$ time ./Task5
Sum = 2879914

real    0m0.833s
user    0m1.592s
sys     0m0.008s
```

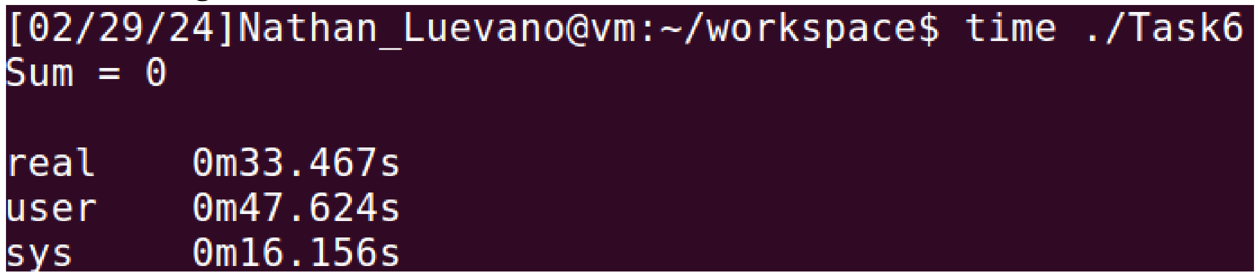
Code:

- In this code I added a mutex, pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER, which is used for locking and to prevent a race condition
- Unlike in task 4 I start both thread at the same time

Screenshot:

- The screenshot displays an inaccurate number and shows that there is something missing from the program that led to a race condition because I create both thread at the same time and did not add in the mutex unlock and lock

Task 6: Solving the Race Condition



```
[02/29/24]Nathan_Luevano@vm:~/workspace$ time ./Task6
Sum = 0

real    0m33.467s
user    0m47.624s
sys     0m16.156s
```

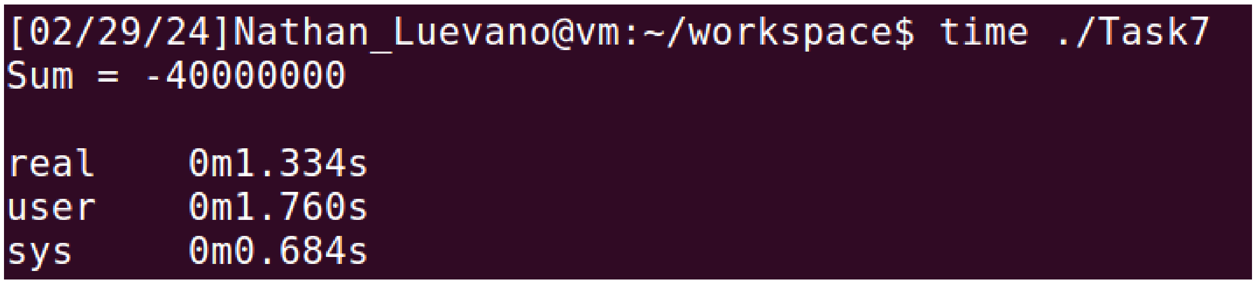
Code:

- I implemented the code to fix the race condition that came up in task 5
- I identified the critical section and the end of the critical sections
- I then added a loop lock, pthread_mutex_lock(&mutex), and unlock, pthread_mutex_unlock(&mutex)

Screenshot:

- This shows us the correct value because now with the added loop locks the program will ensure that only one thread can modify the sum value at a time

Task 7: Task 6 but With 20000000 Instead of 500000000



```
[02/29/24]Nathan_Luevano@vm:~/workspace$ time ./Task7
Sum = -40000000

real	0m1.334s
user	0m1.760s
sys	0m0.684s
```

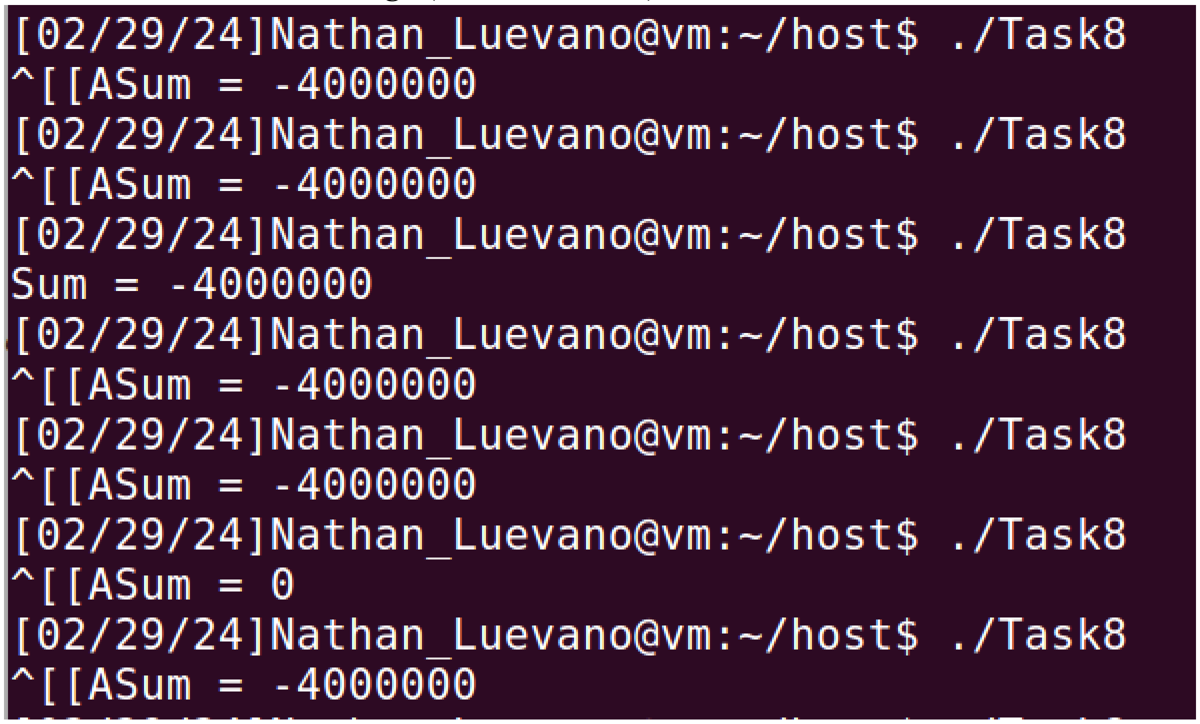
Code:

- Instead of having 2 offset variables for each thread I reused the offset1 for both threads

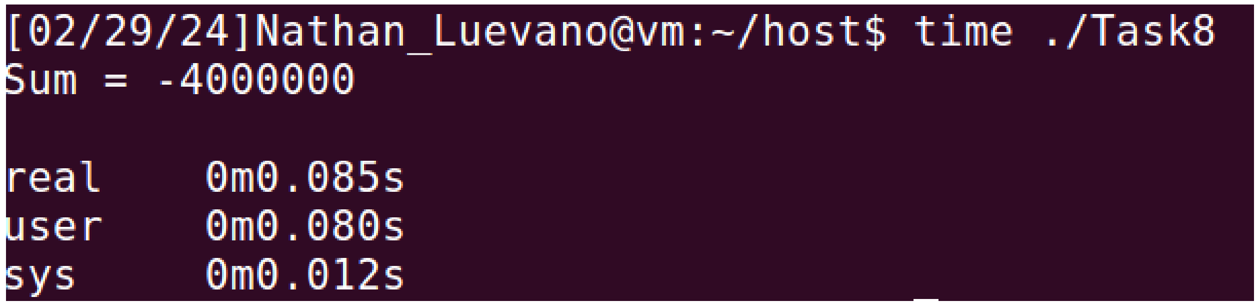
Screenshot:

- The sum is -40000000 which is not what was expected. The expect value was 0
- This shows us that during the execution of the code the decrementing operation was executed more times than the incrementing operation
- This happened most likely due to a timing issue in thread execution

Task 8: What Can Go Wrong? (Race Condition)



```
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
^[[ASum = -4000000
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
^[[ASum = -4000000
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
Sum = -4000000
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
^[[ASum = -4000000
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
^[[ASum = -4000000
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
^[[ASum = 0
[02/29/24]Nathan_Luevano@vm:~/host$ ./Task8
^[[ASum = -4000000
```



```
[02/29/24]Nathan_Luevano@vm:~/host$ time ./Task8
Sum = -4000000

real    0m0.085s
user    0m0.080s
sys     0m0.012s
```

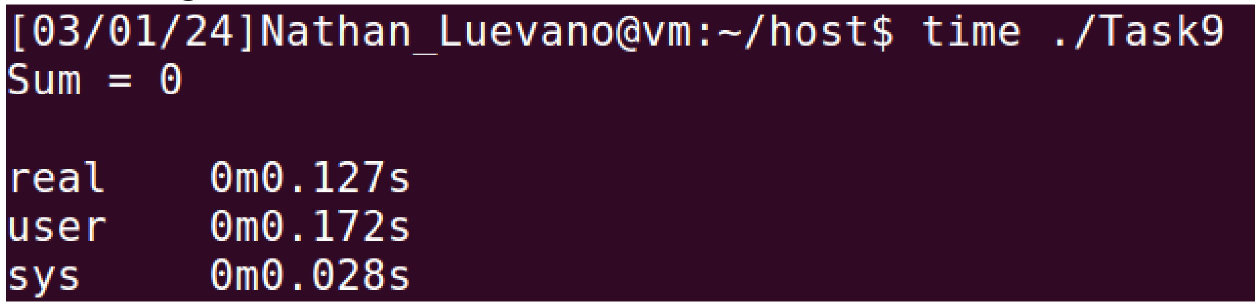
Code:

- For task 8 the code is exactly the same as in task 7 except for me lowering the NUM_LOCK by a value of 10 so it runs faster

Screenshot:

- The screenshots show that there is one inconsistency that the value does comes to a 0 which is due to the threads grabbing the local variable and then we execute the second thread and execute it
- This causes a race condition because the order of executions affects the outcome

Task 9: Fixing the New Race Condition



```
[03/01/24]Nathan_Luevano@vm:~/host$ time ./Task9
Sum = 0

real    0m0.127s
user    0m0.172s
sys     0m0.028s
```

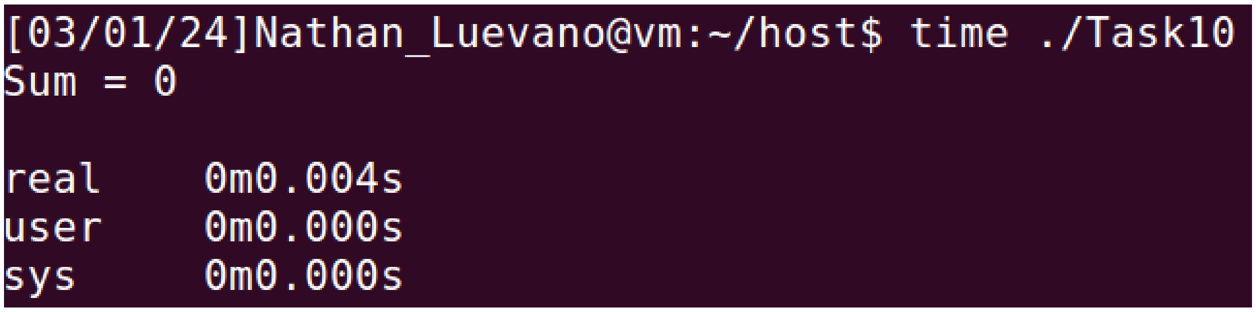
Code:

- Fixed it so that both threads create and point to 2 different variables not the same anymore.

Screenshot:

- Since the offset is 1 on the first thread and -1 on the second it should return a sum of 0 as shown in the picture

Task 10: A Condition That Does Not Seem to Need a Mutex



```
[03/01/24]Nathan_Luevano@vm:~/host$ time ./Task10
Sum = 0

real	0m0.004s
user	0m0.000s
sys	0m0.000s
```

Code:

- Changed the NUM_LOCK to a small number like 100

Screenshot:

- This doesn't need a mutex lock because the count is so low and can run so fast that it only needs one CPU cycle but as the number goes higher it will take a longer time and needs synchronization threads.