The purpose of this lab is to work with thread applications. First, I typed the sudo nano count_first.c command to type make the file. Then, I have typed the command to compile and to execute it later. That would be the gcc -Wall -g -std=c99 -Werror count_first.c -o count_first. This compiles our c program file and let the count_first.c file to execute.

```
[06/01/23]seed@VM:~$ export PS1="Eric_Kim@vm:"
Eric_Kim@vm:sudo nano count_first.c
Eric_Kim@vm:gcc -Wall -g -std=c99 -Werror count_first.c -o count_first
```

Observation: This is the gcc compiler that was used.

Within the count_first.c file we have this code written.

```
  GNU nano 4.8                    count_first.c
#include <stdio.h>

#define NUM_LOOPS 600000000
long long sum = 0;

void counting_function(int offset)
{
  for (int i = 0; i < NUM_LOOPS; i ++ ){
        sum += offset;
        }
}

int main(void)
{
        counting function(1);
        printf("Sum = %lld\n", sum);
        return 0;
}erickim
```

Observation: disregard the name that was written and this prints the sum that was added from the iteration of the loop.

Then we type in the executable command. We type the command ./count_first. Then we also type the command time ./count_first.

```
Eric_Kim@vm:sudo nano count_first.c
Eric_Kim@vm:gcc -Wall -g -std=c99 -Werror count_first.c -o count_first
Eric_Kim@vm:
Eric_Kim@vm:./count_first
Sum = 600000000
Eric_Kim@vm:time ./count_first
Sum = 600000000

real    0m0.903s
user    0m0.889s
sys     0m0.000s
```

Observation: The time command showed that it took .903s to execute the file.
Doubling the length of a single thread.

We will go back to edit our code using the sudo nano count_first.c command once again. In that file we will see our code like this.

```c
#include <stdio.h>

#define NUM_LOOPS 600000000
long long sum = 0;

void counting_function(int offset)
{
  for (int i = 0; i < NUM_LOOPS; i ++ ){
        sum += offset;
        }
}

int main(void)
{

        counting_function(1);
        counting_function(-1);
        printf("Sum = %lld\n", sum);
        return 0;
}
```

Observation: this shows that it is counted up towards 600000000 times and downward 60000000 times. We ran the counting function twice.

Then we will run the command that will compile the

```
Eric_Kim@vm:sudo nano count_second.c
Eric_Kim@vm:gcc -Wall -g -std=c99 -Werror -pthread count_second.c -o count_secon
d
Eric_Kim@vm:time ./count_second
Sum = 0

real    0m1.845s
user    0m1.775s
sys     0m0.010s
Eric Kim@vm:
```

program once again and execute it.

Observation: It worked as we had expected. The sum was zero because the function that counted up went back down. As expected the time it took to run this file was almost twice the amount of time.

Now, we will count up by a thread. We have to go back into the sudo nano count_first.c file to edit this again.

This is where we would change the codes. We will examine the code changes within the file.

We are passing in a thread in this function. Int offset will be casted into pointer type so this function can run.

```
void counting_function(void *arg)
{
        int offset = *(int *) arg
```

In here, we examined the code we have to insert pthread_exit(NULL); this means to end the thread once done as well as to return nothing which would be null.

```
void counting_function(void *arg)
{
        int offset = *(int *) arg
  for (int i = 0; i < NUM_LOOPS; i ++ ){
        sum += offset;
        }
        pthread_exit(NULL);
}
```

This would be the full code to run the thread. We create the thread id, also passed in values within the thread that the value was set to 1. Pthread_join makes the thread wait.

This is the full code that we would write for running our thread that is counting up. Make sure to add the library for thread as well.

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_LOOPS 600000000
long long sum = 0;

void* counting_function(void *arg)
{
        int offset = *(int *) arg;
        for (int i = 0; i < NUM_LOOPS; i ++ ){
        sum += offset;
        }
        pthread_exit(NULL);
}

int main(void)
{
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function,&offset1);

        pthread_join(id1, NULL);

        //counting_function(-1);
        printf("Sum = %lld\n", sum);
        return 0;
}
```

Observation: we have deleted the counting_function(); instead we will be passing in threads with pointers and dereferencing.

We will compile the code once again, and also add the -pthread onto it before executing the file.

```
Eric_Kim@vm:sudo nano count_first.c
Eric_Kim@vm:gcc -Wall -g -std=c99 -Werror -pthread count_first.c -o count_first
Eric_Kim@vm:time ./count_first
Sum = 600000000

real    0m0.901s
user    0m0.870s
sys     0m0.010s
Eric_Kim@vm:
```

Observation: pthread was added onto here.

Here, we have added the same function from thread 1. The thread one counts up and the second thread will count downwards.

```c
#define NUM_LOOPS 600000000
long long sum = 0;

void* counting_function(void *arg)
{
        int offset = *(int *) arg;
        for (int i = 0; i < NUM_LOOPS; i ++ ){
        sum += offset;
        }
        pthread_exit(NULL);
}

int main(void)
{
        //erickim thread spawn
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function,&offset1);
        pthread_join(id1, NULL);

        pthread_t id2;
        int offset1 = -1;
        pthread_create(&id2, NULL, counting_function,&offset2);
        pthread_join(id2, NULL);

        //counting_function(-1);
        printf("Sum = %lld\n", sum);

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
```

Observation: The first thread will execute, then the second thread will execute.

Here, we have compiled the file again, then we see that the time took twice as long and the sum is zero from running two threads.

```
Eric_Kim@vm:sudo nano count_second.c
Eric_Kim@vm:gcc -Wall -g -std=c99 -Werror -pthread count_second.c -o count_secon
d
Eric_Kim@vm:time ./count_second
Sum = 0

real    0m1.845s
user    0m1.775s
sys     0m0.010s
Eric Kim@vm:
```

Observation: this thread was created in a different file. It took twice as long and the sum as expected was zero because it counted up and counted down.

This time we will also use sudo nano count_fifth.c to edit the file again. This time we will separate the thread to finish from when the thread is spawned.

```c
int main(void)
{
        //erickim thread spawn
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function,&offset1);


        pthread_t id2;
        int offset2 = -1;
        pthread_create(&id2, NULL, counting_function,&offset2);

        //waiting thread to finish
        pthread_join(id1, NULL);
        pthread_join(id2, NULL);

        printf("Sum = %lld\n", sum);
        return 0;
```

Observation: This will mess up the sum as it will produce different sums everytime the code is being executed.

Once again, we have compiled and executed this file again.

```
Eric_Kim@vm:./count_fifth
Sum = 22913682
Eric_Kim@vm:./count_fifth
^[[ASum = 47678304
Eric_Kim@vm:./count_fifth
Sum = 39899142
Eric_Kim@vm:./count_fifth
Sum = -27867482
Eric_Kim@vm:./count_fifth
Sum = 19268104
Eric_Kim@vm:time ./count_fifth
Sum = 1980245

real    0m1.874s
user    0m1.725s
sys     0m0.061s
Eric_Kim@vm:
```

Observation: the time took longer and it produced different sums because of race conditions.

Now we need to solve the race condition. In this case, the global variable that is being accessed is the sum. That will be the critical section that we need to protect from the resource that is being shared by different threads.

We will change the code once again by accessing the file. The solution is mutex. The mutex is initialized to the default. Since the critical section that we need to protect is the sum, that section will be where the mutex lock and unlock section happens. This means that only one thread can execute at a given time.

```
#include <stdio.h>
#include <pthread.h>

#define NUM_LOOPS 600000000
long long sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* counting_function(void *arg)
{
        int offset = *(int *) arg;
        for (int i = 0; i < NUM_LOOPS; i ++ ){
        //start critical section
        pthread_mutex_lock(&mutex);

        sum += offset;
        //end critical section
        pthread_mutex_unlock(&mutex);
//erickim
        }
        pthread_exit(NULL);
```

Observation: Once again, the critical section is within the for loop because the sum is our critical section.

As seen, when we ran the command once again to execute, it was added then subtracted.

```
Eric_Kim@vm:gcc -Wall -g -std=c99 -Werror -pthread count_fifth.c -o count_fifth
Eric_Kim@vm:time ./count_fifth
Sum = 0

real    0m32.240s
user    0m31.825s
sys     0m0.020s
```

Observation: We ran a two system call, that is an overhead. That is why it took a very long time for this file to execute.

This time, we will examine another race condition. This time we will use the same offset memory address. This will create some issues in the order the thread gets executed.

```
        for (int i = 0; i < NUM_LOOPS; i ++ ){
        //start critical section
        pthread_mutex_lock(&mutex);

        sum += offset;
        //end critical section
        pthread_mutex_unlock(&mutex);
//erickim
        }
        pthread_exit(NULL);
}

int main(void)
{
        //erickim thread spawn
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function,&offset1);


        pthread_t id2;
      //int offset2 = -1;
        offset1 = -1;
        pthread_create(&id2, NULL, counting_function,&offset1);

        //pthread_create(&id2, NULL, counting_function,&offset2);
        //waiting thread to finish
        pthread_join(id1, NULL);
        pthread_join(id2, NULL);

        printf("Sum = %lld\n", sum);
        return 0;
        }
```

Observation: In most cases it will create two threads and execute it. However, sometimes the order gets tangled up because the value of the offset is passed down so from the first thread it goes to the function to get executed a little bit and then creates thread 2 that will change the outcome.

We execute this command after compiling it once more.

```
Eric_Kim@vm:./count_tenth
Sum = -12000000
Eric_Kim@vm:./count_tenth
Sum = -12000000
Eric_Kim@vm:./count_tenth
Sum = -12000000
Eric_Kim@vm:
Eric_Kim@vm:./count_tenth
Sum = -12000000
```

Observation: most of the time it will execute it by creating two threads which is the case here.

Now we go back into the file and fix it.

```c
int main(void)
{
        //erickim thread spawn
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function,&offset1);


        pthread_t id2;
        int offset2 = -1;
        pthread_create(&id2, NULL, counting_function,&offset2);

        //waiting thread to finish
        pthread_join(id1, NULL);
        pthread_join(id2, NULL);

        printf("Sum = %lld\n", sum);
        return 0;
```

Observation: In here, we have changed the offset value for both of the offset values for the threads.

Then we run this code again to execute. This is the proper answer, and we have executed the program correctly.

```
Eric_Kim@vm:gcc -Wall -g -std=c99 -W
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
Eric_Kim@vm:./count_tenth
Sum = 0
```

Observation: it gave us a consistent answer.

Here, we have gotten rid of the mutex. We commented out the pthread mutex.

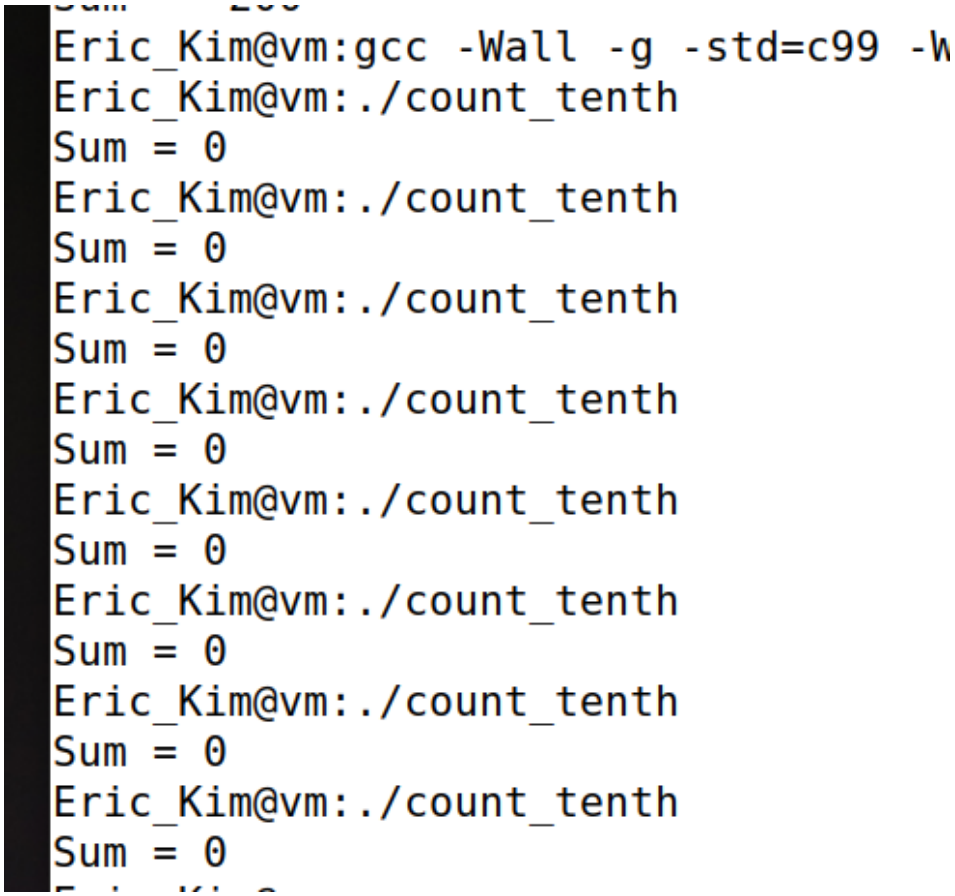
```
#define NUM_LOOPS 6000000
long long sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* counting_function(void *arg)
{
        int offset = *(int *) arg;
        for (int i = 0; i < NUM_LOOPS; i ++ ){
        //start critical section
        //pthread_mutex_lock(&mutex);

        sum += offset;
        //end critical section
        //pthread_mutex_unlock(&mutex);
//erickim
        }
        pthread_exit(NULL);
}

int main(void)
```

Observation: we have gotten rid of the mutex.

```
#define NUM_LOOPS 100
long long sum = 0;
```

Here, we have changed the number of loops to a smaller number.

We ran the code execution again, and we get the correct sum without the mutex.



Observation: It did not work on a higher number of loops. It worked on a smaller number of loops without the mutex because the first thread runs through a short amount of burst before the second loop executes. This works for smaller amounts of loops but gives a random value when there is a higher number of loops.