Evan Lang
Yang Lu
EC527

Lab 4

Task 1:

```
the_data = &ArrayA;       /* make ptr point to a memory location */
printf("the_data points to the float value of the array %f\n", *(float*) the_data);
printf("the_data points to the float value of the array %f\n", *((float*) the_data + 1));
printf("the_data points to the float value of the array %f\n", *((float*) the_data + 2));
```

The altered code, printing the first 3 entries by adding the offset to each of the pointer's in the print statements

Output:

```
evanlang@vlsi37$ ./test_generic
the_data points to the float value 1.100000
the_data points to the float value 2.200000
the_data points to the float value 3.300000
the_data points to the integer value 6
the_data now points to the character a
```

Task 2:

The output changed each time I ran the code. None of the child threads or main ID were preserved between the runs.

Task 3:

```
pthread_t *id = (pthread_t*) malloc(NUM_THREADS);
```

Altered declaration of id using malloc.

Output:

```
evanlang@vlsi37$ ./test_create
Hello test_create.c
Hello World! from child thread 2acc28187700
Hello World! from child thread 2acc27f86700
Hello World! from child thread 2acc2878a700
Hello World! from child thread 2acc28589700
main() after creating the thread.  My id is 2acc275c1600
Hello World! from child thread 2acc28388700
```

Task 4:

```
evanlang@vlsi37$ ./test_create
Hello test_create.c
main() after creating the thread.  My id is 2af96a078600
```

The child threads are never printed and seemingly the pthread is executed before the print statement occurs.

Task 5:

```
Hello test_create.c
Hello World! from child thread 2ac34978f700
main() after creating the thread.  My id is 2ac348dca600
Hello World! from child thread 2ac349f93700
Hello World! from child thread 2ac349d92700
Hello World! from child thread 2ac349990700
Hello World! from child thread 2ac349b91700
```

The order of thread completion changes, with the main thread seemingly completing before the child threads.

Task 6:

```
main() -- After creating the thread.  My id is: 2b23a3c52600
 Hello World!  It's me, thread #2b23a4a19700 --
 Hello World!  It's me, thread #2b23a4c1a700 --
 Hello World!  It's me, thread #2b23a4e1b700 --
 Hello World!  It's me, thread #2b23a4818700 --
After joining, Good Bye World!
ylu149@vlsi35$ vim test_join.c
ylu149@vlsi35$ gcc -pthread test_join.c -o test_join -std=gnu99
ylu149@vlsi35$ ./test_join
Hello test_join.c
 Hello World!  It's me, thread #2b23f4140700 --
 Hello World!  It's me, thread #2b23f4341700 --
 Hello World!  It's me, thread #2b23f4944700 --
 Hello World!  It's me, thread #2b23f4542700 --
 Hello World!  It's me, thread #2b23f4743700 --
main() -- After creating the thread.  My id is: 2b23f377b600
After joining, Good Bye World!
ylu149@vlsi35$
```

Adding a sleep(3) to the child process forces the parent thread to resolve first. Otherwise the parent process can resolve between any of the child processes.

Task 7:

```
ylu149@signals14$ ./test_param
Hello test_param1.c
In main:  creating thread 0
In main:  creating thread 1
PrintHello() in thread # as int 0 !
PrintHello() in thread # as char 0 !
PrintHello() in thread # 0 !
PrintHello() in thread # as int 1 !
PrintHello() in thread # as char 1 !
PrintHello() in thread # 1 !
In main:  creating thread 2
In main:  creating thread 3
PrintHello() in thread # as int 2 !
PrintHello() in thread # as char 2 !
PrintHello() in thread # 2 !
PrintHello() in thread # as int 3 !
PrintHello() in thread # as char 3 !
PrintHello() in thread # 3 !
Segmentation fault (core dumped)
```

It does not seem to work as intended. Passing a negative number seems to lead to a segmentation fault. Based off of this, it can be inferred that the thread id's cannot contain negative values.

Task 8:

```
ylu149@vlsi35$ ./test_param2
in work(): f= 8, k=49999995000000, *g=10
in work(): f= 5, k=49999995000000, *g=10
in work(): f= 6, k=49999995000000, *g=10
in work(): f= 4, k=49999995000000, *g=10
in work(): f= 7, k=49999995000000, *g=10
in work(): f= 2, k=49999995000000, *g=10
in work(): f= 3, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
```

```
ylu149@vlsi35$ ./test_param2
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
in work(): f=10, k=49999995000000, *g=10
```

By adjusting f and g* prior to the print, the output will change from run to run. This is because the threads are grabbing the values of f and g* at different starting times, hence, the values of f and g* will always vary.

Task 9:

```
in work(): f= 7, k=49999995000000, *g=10
in work(): f= 6, k=49999995000000, *g=10
in work(): f= 2, k=49999995000000, *g=10
in work(): f= 4, k=49999995000000, *g=10
in work(): f= 3, k=49999995000000, *g=10
in work(): f= 5, k=49999995000000, *g=10
in work(): f= 6, k=49999995000000, *g=10
```

Threads can be intercepted and deleted prior in the function work itself.

Task 10:

One way to do this is to create a hash map to map each thread to a specific identifier number. Then one can check and see if the thread exists in the map. If it does then return the identifier number, otherwise, the new thread number can be added to the map.

Task 11:

```
Hello test_param3.c
In main:  creating thread 0
In main:  creating thread 1
in PrintHello(), thread #0 ; sum = 27, message = First Message
in PrintHello(), thread #1 ; sum = 28, message = Second Message
In main:  creating thread 2
In main:  creating thread 3
in PrintHello(), thread #2 ; sum = 29, message = Third Message
In main:  creating thread 4
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
In main:  creating thread 5
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
in PrintHello(), thread #5 ; sum = 1000, message = Sixth Message
```

Task 12:

Original Output:

```
Hello sync1
In main: created thread 1, which is blocked
   Press any letter key (not space) then press enter:
a
in thread ID 0 (sum = 123 message = First Message), now unblocked!
After joining
GOODBYE WORLD!
```

Output with commented out trylock:

```
Hello sync1
In main: created thread 1, which is blocked
   Press any letter key (not space) then press enter:
in thread ID 0 (sum = 123 message = First Message), now unblocked!
a
After joining
GOODBYE WORLD!
```

The thread immediately “unblocked” (it was never blocked in the first place). This led to the thread completing before I entered anything.

Task 13:

New output:

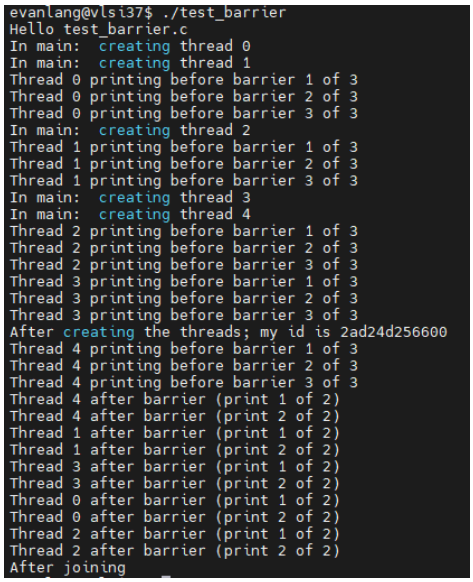
```
Hello sync1
in thread ID 0 (sum = 123 message = First Message), now unblocked!
A
s

^C
```

The code now hangs after printing the unblocked first thread, I would consider this broken.

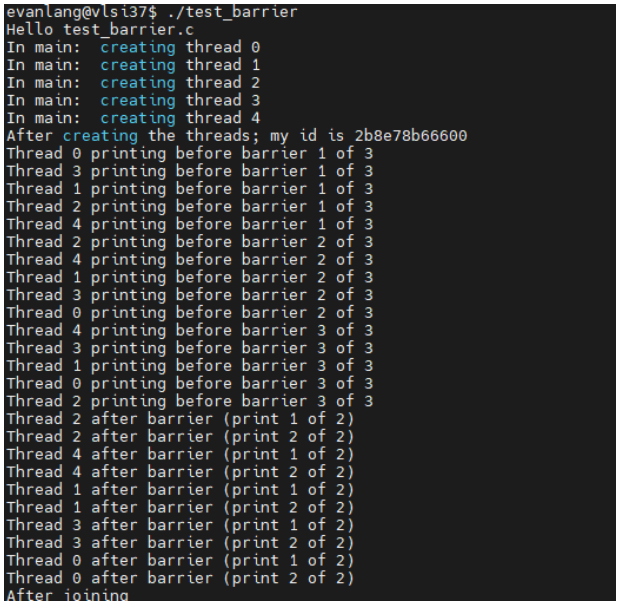
Task 14:

Original output (confirmed before prints before after):



```
evanlang@vlsi37$ ./test_barrier
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 2
Thread 1 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3
In main:  creating thread 3
In main:  creating thread 4
Thread 2 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
Thread 3 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
After creating the threads; my id is 2ad24d256600
Thread 4 printing before barrier 1 of 3
Thread 4 printing before barrier 2 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
After joining
```

With Sleep(1):



```
evanlang@vlsi37$ ./test_barrier
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
In main:  creating thread 4
After creating the threads; my id is 2b8e78b66600
Thread 0 printing before barrier 1 of 3
Thread 3 printing before barrier 1 of 3
Thread 1 printing before barrier 1 of 3
Thread 2 printing before barrier 1 of 3
Thread 4 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 4 printing before barrier 2 of 3
Thread 1 printing before barrier 2 of 3
Thread 3 printing before barrier 2 of 3
Thread 0 printing before barrier 2 of 3
Thread 4 printing before barrier 3 of 3
Thread 3 printing before barrier 3 of 3
Thread 1 printing before barrier 3 of 3
Thread 0 printing before barrier 3 of 3
Thread 2 printing before barrier 3 of 3
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
After joining
```

The threads are no longer printing the before barriers correctly, running 0-3 before 0,1,2 are done respectively.

With Sleep(taskid):

```
evanlang@vlsi37$ ./test_barrier
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 3
In main:  creating thread 4
After creating the threads; my id is 2b2610e27600
Thread 1 printing before barrier 1 of 3
Thread 2 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3
Thread 3 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 4 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
Thread 4 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
After joining
```

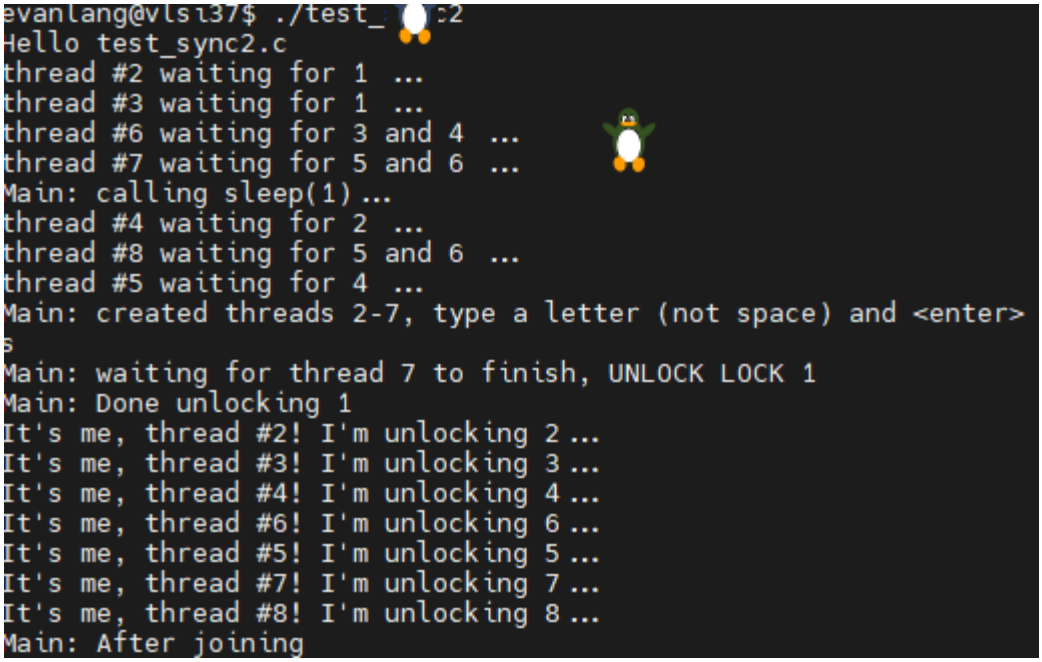
All threads wait till the block from 0 is done but they still occur out of order after.

Task 15:

```
Hello test_sync2.c
thread #2 waiting for 1  ...
thread #3 waiting for 1  ...
thread #6 waiting for 3 and 4  ...
thread #5 waiting for 4  ...
Main: calling sleep(1) ...
thread #7 waiting for 5 and 6  ...
thread #4 waiting for 2  ...
Main: created threads 2-7, type a letter (not space) and <enter>
a
Main: waiting for thread 7 to finish, UNLOCK LOCK 1
Main: Done unlocking 1
It's me, thread #2! I'm unlocking 2 ...
It's me, thread #3! I'm unlocking 3 ...
It's me, thread #4! I'm unlocking 4 ...
It's me, thread #6! I'm unlocking 6 ...
It's me, thread #5! I'm unlocking 5 ...
It's me, thread #7! I'm unlocking 7 ...
Main: After joining
```

The join occurs after due to the final mutex lock on line 152. This prevents it from occurring until the final thread locks.

Task 16:



Task 17:

```
Hello test_crit.c
 MAIN --> final balance = 999
          qr_total = 18.035593
ylu149@vlsi35$ vim test_crit.c
ylu149@vlsi35$   gcc -pthread test_crit.c -o test_crit -std=gnu99
ylu149@vlsi35$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 999
          qr_total = 17458.733753
ylu149@vlsi35$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 998
          qr_total = 17463.977350
ylu149@vlsi35$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1002
          qr_total = 17463.198553
ylu149@vlsi35$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1003
          qr_total = 17463.027875
ylu149@vlsi35$ ./test_crit
Hello test_crit.c
 MAIN --> final balance = 1002
          qr_total = 17464.076758
ylu149@vlsi35$
```

Final balance definitely seems wrong. The final balance value fluctuates around 1000. This is especially so at a higher number of threads. The issue stems from the fact that multiple threads are attempting to access the final balance value and perform arithmetic on it.

Task 18:

```
ylu149@signals14$ gcc -pthread test_crit.c -o test_crit -std=gnu99
ylu149@signals14$ ./test_crit
Hello test_crit.c
Done loop MAIN --> final balance = 1000
          qr_total = 17464.076758
ylu149@signals14$
```

Using the mutex to lock down the threads prevents multiple threads from accessing the balance value at the same time. Hence, producing the result show below.

Task 19:

```
test_sor setup ...
Thread for index 1 starting
Thread for index 2 starting
Thread for index 6 starting
Thread for index 4 starting
Thread for index 3 starting
Thread for index 8 starting
Thread for index 5 starting
Thread for index 7 starting
Thread for index 9 starting
Thread for index 10 starting
  0.00 12.50 18.75 20.86 22.97 23.99 25.00 30.66 36.29 57.48 78.74 100.00

real    0m0.006s
```

This initial run at 1000 iterations is incorrect.

```
test_sor setup ...
Thread for index 1 starting
Thread for index 2 starting
Thread for index 6 starting
Thread for index 4 starting
Thread for index 9 starting
Thread for index 10 starting
Thread for index 5 starting
Thread for index 3 starting
Thread for index 8 starting
Thread for index 7 starting
  0.00  9.09 18.18 27.27 36.36 45.45 54.54 63.63 72.73 81.82 90.91 100.00

real    0m0.187s
```

This run using 150000 iterations is finally correct

Task 20: At 10000 iterations and barriers on:

```
test_sor setup ...
Thread for index 1 starting
Thread for index 4 starting
Thread for index 7 starting
Thread for index 5 starting
Thread for index 8 starting
Thread for index 10 starting
Thread for index 3 starting
Thread for index 9 starting
Thread for index 6 starting
Thread for index 2 starting
  0.00  9.09 18.18 27.27 36.35 45.45 54.54 63.63 72.72 81.81 90.91 100.00

real    0m0.195s
```

With barriers it consistently took longer than by a marginal amount when compared to without barriers, as the barriers forced the threads to wait. However, I only needed 10000 iterations when compared with the 150000 for no barriers. At this amount of iterations both constantly reported the correct answer, however at slightly less iterations than optimal the nonbarrier method was less accurate by a large margin. This is due to the fact that without a barrier the threads are able to complete without regarding the status of the other threads, skewing the results.