Hannah Gibson
EC 527
Lab 4
2/24/16

Task 1) See modified test_generic.c code.

Task 2)

Compiling test_create.c generally produces the following output:

```
“Hello from child thread <number>”
“Hello from child thread <number>”
“Hello from child thread <number>”
“Hello from child thread <number>”
“Hello from child thread <number>”
“main() after creating the thread. My id is <number>”.
```

The output is not always the same. Sometimes the thread numbers are in different orders, sometimes main outputs its statement before the last thread, sometimes only 3 or 4 of the child threads execute.

Task 3) See modified test_create.c code.

Task 4)

When the line “sleep(3)” is added to the child thread work function before the print statement in test_create.c none of the child threads print anything out. This is likely because the main thread finishes executing before the child thread can execute.

Task 5)

When the line “sleep(3)” is added to the main function after the print statement and before the return the output is similar to Task 2 except that all five child threads always execute. This is likely because the sleep statement causes main to wait long enough that all the child threads finish executing before main exits.

Task 6)

When the line “sleep(3)” is added to the work function of the child thread before the print statement in test_join.c cases the program to execute more slowly but all child threads finish executing before main prints and exits.

Task 7)

When the program test_param1.c is compiled with the data passed to the thread function cast to a long unsigned int the program prints the following output:

```
“Hello World!  It's me, MAIN!”
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
 Hello World!  It's me, thread # 0 !
In main:  creating thread 3
 Hello World!  It's me, thread # 2 !
 Hello World!  It's me, thread # 1 !
```

```
In main:  creating thread 4
 Hello World!  It's me, thread # 3 !

 It's me MAIN -- Good Bye World!
 Hello World!  It's me, thread # 4 !”
```

When test_param1.c is compiled with the data passed to the thread function not cast as anything the compiler gives a warning that the initialization makes an integer from a pointer without cast but the program still compiles and prints the following output:

```
“Hello World!  It's me, MAIN!
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
 Hello World!  It's me, thread # 2 !
 Hello World!  It's me, thread # 3 !
In main:  creating thread 4

 It's me MAIN -- Good Bye World!
 Hello World!  It's me, thread # 1 !
 Hello World!  It's me, thread # 4 !
 Hello World!  It's me, thread # 0 !”
```

This output is has essentially the same functionality as the cast version. The thread execution order is different but that can vary between multiple runs of the same binary.

When test_param.c is compiled with the data passed to the thread cast as a char the compiler gives a warning that there is a cast from pointer to integer of a different size, but the program still compiles and produces the following output:

```
“In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
 Hello World!  It's me, thread # 2 !
In main:  creating thread 4

 It's me MAIN -- Good Bye World!
 Hello World!  It's me, thread # 3 !
 Hello World!  It's me, thread # 1 !
 Hello World!  It's me, thread # 4 !
 Hello World!  It's me, thread # 0 !”
```

Again, there is variation in the thread execution order, but that variation was not the result of changing the data type. There seem to be no negative effects from incorrect casting but it’s possible that if the number of threads increased so that the ID number was too large to store in a char variable the program would execute incorrectly.

Task 8)

The thread print out order changes each time the program is run. This is true regardless of what data type the input to the thread work function is cast to.

Task 9)

The program test_param.c created 10 threads which use two values: f which is the value being pointed to by the void pointer input to the thread work function, and g which was the address of the value being pointed to by the input to the thread work function. The value i which ranges from 1 to 10 is passed to each thread as it is created. If the threads did not affect each other's values the printed values f and *g would be the same and thread 1 would print out 1 1, thread 2 would print out 2 2, etc. The unmodified version of test_param2.c where none of the threads performed operations on f or g produced the following output when run:

```
“Hello World from 139723212760832 with value 4
Hello World! 4  8
Hello World from 139723191781120 with value 5
Hello World! 5  10
Hello World from 139723149821696 with value 10
Hello World! 10  10
Hello World from 139723233740544 with value 5
Hello World! 5  10
Hello World from 139723170801408 with value 10
Hello World! 10  10
Hello World from 139723202270976 with value 5
Hello World! 5  10
Hello World from 139723223250688 with value 6
Hello World! 6  10
Hello World from 139723139331840 with value 10
Hello World! 10  10
Hello World from 139723181291264 with value 10
Hello World! 10  10
Hello World from 139723160311552 with value 10
Hello World! 10  10
After creating the thread.  My id is 139723233748736, i = 10”
```

Test_param2.c was modified so that each thread incremented f and *g. The modified test_param.c produced the following output:

```
“Hello World from 140240978818816 with value 3
Hello World! 3  6
Hello World from 140240957839104 with value 5
Hello World! 5  11
Hello World from 140240936859392 with value 11
Hello World! 11  12
Hello World from 140240915879680 with value 11
Hello World! 11  13
Hello World from 140240894789376 with value 11
Hello World! 11  14
Hello World from 140240905389824 with value 11
Hello World! 11  15
Hello World from 140240968328960 with value 6
Hello World! 6  16
Hello World from 140240926369536 with value 11
Hello World! 11  17
Hello World from 140240947349248 with value 6
Hello World! 6  18
```

After creating the thread. My id is 140240978827008, i = 18"

The variable f is a local copy of the value passed to the function, so modifying that would have no effect on the other threads. However, the value stored at g is used by the other threads so modifying that affects their input data. Also, in the unmodified test_param2.c and the modified version where only f is incremented by the thread function all 10 threads get executed, but in the version where *g is incremented only 8 threads executed. The output of the unmodified version and the two modified versions changed slightly from run to run (the threads may be executed in different orders) but the end value of i always remained the same.

Task 10)

Test_param2.c was modified again so that the thread work function was passed an array and each thread performed an operation on a different array element.

Task 11)

When the program test_param3.c is compiled and run it produces the following output:

```
 Hello World!  It's me, MAIN!
"In main:  creating thread 0
In main:  creating thread 1
 Hello World!  It's me, thread #0! sum = 28 message = First Message
In main:  creating thread 2
In main:  creating thread 3
 Hello World!  It's me, thread #2! sum = 30 message = Third Message
In main:  creating thread 4
 Hello World!  It's me, thread #3! sum = 31 message = Fourth Message
 Hello World!  It's me, thread #1! sum = 29 message = Second Message
 Hello World!  It's me, thread #4! sum = 32 message = Fifth Message"
```

Test_param3.c was then modified so that it had a sixth thread that it output a sum of 1000.

Task 12)

The program test_sync1.c spawns a thread which cannot reach its print statement until the main thread releases control of the mutex. This is done by entering a character and pressing return. When the unmodified version of test_sync1.c is run it produces the following output:

```
Hello World!  It's me, MAIN!
In main: created thread 1, which is blocked -- press a char and enter
a
   I, thread #0 (sum = 28 message = First Message) am now unblocked!
After joining
GOODBYE WORLD!"
```

When test_sync1.c was modified so that the trylock check in the child thread work function was removed the program produces the following output:

```
"In main: created thread 1, which is blocked -- press a char and enter
   I, thread #0 (sum = 28 message = First Message) am now unblocked!
a
After joining
GOODBYE WORLD!"
```

Which is the same as the unmodified version.

Task 13)

When another child thread was added to test_sync1.c with no trylock check the program requested character input but did not run the rest of the code. This is likely because the first child thread prompted the user for input before the mutex was released by main. The mutex never was released by main because main was still trying to spawn the second thread.

Task 14)

The program test_barrier.c produced the following output:

```
“In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
Thread 2 printing before barrier 1 of 3
Thread 3 printing before barrier 1 of 3
Thread 0 printing before barrier 1 of 3
Thread 1 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 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 4
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3

After creating the threads.  My id is:  140614733809408
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 printing after barrier 1 of 2
Thread 4 printing after barrier 2 of 2
Thread 2 printing after barrier 1 of 2
Thread 2 printing after barrier 2 of 2
Thread 3 printing after barrier 1 of 2
Thread 3 printing after barrier 2 of 2
Thread 0 printing after barrier 1 of 2
Thread 0 printing after barrier 2 of 2
Thread 1 printing after barrier 1 of 2
Thread 1 printing after barrier 2 of 2”
```

All the before barrier messages are printed before the after barrier messages. When test_barrier.c was modified so that the child thread work function had a “sleep(1)” statement before printing output the “before barrier” messages still printed before the “after barrier” messages; the program simply took longer to run.

Task 15)

When the program test_sync2.c was run it printed the following output:

```
 “Hello World!  It's me, MAIN!
```

```
It's me, thread #6! I'm waiting 3 and 4 ...
It's me, thread #5! I'm waiting for 4 ...
It's me, thread #2! I'm waiting for 1 ...
It's me, thread #3! I'm waiting for 1 ...
It's me, thread #4! I'm waiting for 2 ...
It's me, thread #7! I'm waiting 5 and 6 ...
 Created threads 2-7, type any character and <enter>
a

Waiting for thread 7 to finish, UNLOCK LOCK 1

 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 #5! I'm unlocking 5...
It's me, thread #6! I'm unlocking 6...
It's me, thread #7! I'm unlocking 7...
After joining
GOODBYE WORLD!"
```

The threads threads were spawned out of order but they were waiting for the correct mutex to unlock and executed in the correct order after a character was input. The join loop exits after most of the threads have executed because each thread unlocks its mutex after it finishes executing. For this reason the last thread before the join will have waited for the other threads to exit and unlock their mutexes so most of the threads will have exited by the time the join occurs.

Task 16) See modified test_sync2.c code.

Task 17)

The program test_crit.c had a balance variable containing a value. Some threads added to the value and some threads subtracted from the value. When test_crit.c was compiled and run it produced the following output:

```
" Hello World!  It's me, MAIN!

 MAIN --> balance = 1000"
```

The starting balance was 1000, the main function spawned 5 threads which incremented the balance and 5 threads which decremented the balance. The end balance was 1000 so the program executed correctly.

When the number of threads spawned was increased to 10000 the final balance was 999 instead of 1000 so the program executed incorrectly.

Task 18) See modified test_crit.c code.