John Burke
Robert Munafo
EC 527

LAB 4

Task 1

We iterate through it the following way: **`*((float*) the_data+i)`**

First we cast the_data as a float pointer, increment into it (C handles the sizes here), then dereference to get the value.

Task 2

From terminal:

```
[jsburke@vlsi35 given]$ ./create
 Hello World! from child thread 140453911426816
 Hello World! from child thread 140453900936960
 Hello World! from child thread 140453932406528
 Hello World! from child thread 140453921916672

 main() after creating the thread.  My id is 140453932414720
 Hello World! from child thread 140453890447104
[jsburke@vlsi35 given]$ ./create
 Hello World! from child thread 140079809988352
 Hello World! from child thread 140079778518784
 Hello World! from child thread 140079799498496
 Hello World! from child thread 140079789008640

 main() after creating the thread.  My id is 140079820486400
 Hello World! from child thread 140079820478208
[jsburke@vlsi35 given]$ ./create
 Hello World! from child thread 140377332332288
 Hello World! from child thread 140377311352576
 Hello World! from child thread 140377342822144
 Hello World! from child thread 140377321842432

 main() after creating the thread.  My id is 140377342830336
 Hello World! from child thread 140377300862720
[jsburke@vlsi35 given]$
```

Outputs change slightly in thread number from `pthread_self()`.

Task 3

Done right before loop of `pthread_create()` in the code.

Tasks 4 & 5

Sleeping the child threads causes main to exit first so that none of the child threads get a chance to print. Sleeping main allows all threads to print, but program termination is unsurprisingly delayed. Also noted in the submitted code.

Task 6

There are only small differences in behavior. Notably, when sleeping is used, all the child threads print after the line where main prints its id. Without sleeping children, some of them print before that line.

Task 7

Changing to a char doesn't allow the program to compile, reason being that the pointer and char are different sizes.

Task 8

The print orders can change and interleave a little because of how the OS handles this. Some kind of locking mechanism would be needed to force order strictly.

Task 9

Modifying f quickly makes it the same across all threads. Modifying g does the same since for all it will be an address plus some value. The only thing that might alter from run to run is the value g has. We can get differences in the number of threads we see by using a sleep call, having the threads do more busywork than main and letting main terminate with a return call. Using `pthread_exit()` lets all threads get up, work, and terminate.

Task 10

Accomplished in code. We get each thread to pick an index by getting the value of a global and then incrementing it, then storing the index they get to that point of the array that is shared between all threads by main. In testing, it always behaved well. However, this may not guarantee this is always the case. If the assign then increment is not atomic, two threads could both get the same value before an increment, which would leave a zero in the middle of the array.

Task 11

In code

Task 12

The difference is that the child threads print before the `scanf().`

Task 13

No, this causes it to softlock, even when trylock is reimplemented.

Task 14

Barrier does still work for both cases, but the printing orders differ.

Task 15

Main locks all the mutexes, so all the threads need to wait for some mutex, two notably for one. Once main unlocks the first lock, two can get access and unlock the second and a cascade proceeds. All the threads were waiting to unlock before we typed the dummy key. Once unlocked they fly. Then main has to wait for at least thread 7 to finish since it locks on the same mutex before joining. Once 7 releases it, main can join.

Task 16

Implemented, but the eighth thread must unlock in the order in the code or it will guarantee a soft lock with thread 7.

Task 17

It is correct overall, but the behavior should still be concerning. Increasing the number of threads can make this more apparent easily. Getting the timing to do this with `nanosleep()` is tricky and can be a little hit or miss with only ten threads.

Task 18

See code