Objectives:
This is a program which introduces us to threading, as well as mutual exclusion/semaphores. The goal of the assignment is to make a program which will use a data file that has resources, as well as tasks inputted into it. After interpreting the data file, we setup a thread for each task, as well as a thread for the monitor. To ensure that all threads won't experience a deadlock, or any sort of race conditions, semaphores are implemented for when a thread is accessing shared data (global variables or resources).

After interpreting all the resources and their respective values, we know what the bounds are for the tasks. Thus, we need to create a semaphore for each resource, where the number of concurrent threads using the semaphore is equivalent to the resource value. Once that has been completed, task threads simply need to check the semaphore to see if there's any available resources.

The monitor thread only checks the status of each of the task threads. Thus, there's a status global variable which both monitor and tasks can access. In order to ensure there's no deadlocks or race conditions here, a binary semaphore is implemented to ensure mutual exclusion between the task threads and monitor thread. Thus, when a task changes its status, it opens up the binary semaphore, and only passes its status after it has gained access to the semaphore.

Design Overview:
- Split up into multiple files where each file has a specific task.
- Multiple files make it easy to read and less cumbersome.
- Implements semaphores for all possible race conditions/deadlock possibilities.
- Stores all the information of tasks and resources into nice structs
- Uses a tuple struct to pass around information on resources and tasks between files and functions
- Globals.h has all the modules and structs needed, thus each header file uses globals to ensure each file has the modules and structs needed to run.
- Ensures the main thread is finished last through pthread_join

Project status:
For the most part, this project is complete. It could easily have code shortened, and made to be more optimal, but overall I'm quite happy with it. Initially, I had issues trying to figure out how to pass multiple variables through files. Especially the task and resource structs. In the end, I decided to make a header file which contains all global variables, modules, as well as a struct Tuple which can act as a holder for the resource and task structs. Another issue encountered was figuring out how to pass data around between multiple threads. In the end, I made it as simplistic as possible by making multiple global variables which the threads all have access to. For any of the global variables that might get changed during execution, semaphores were implemented.

Testing and Results:
As I built the program, I would continuously check to see if it still runs. The make command in my makefile sets up the program to be ran with gdb as well by using the -g flag for gcc. Thus, when there was values that didn't seem right, or the flow of the program seems unexpected, I would simply debug using gdb to find what the values are, as well as programs flow. I also ensured that the argv[] is inputted properly through the check_args.c file. This makes sure that the file was ran properly. Thus, we know that if the flow got past that point, we don't have to worry about values that are wrong.

After all the error checking of inputs have finished, I needed to implement threading. Error checking multiple threads running concurrently was tricky because I had to ensure a task didn't exceed any of the resource values. To ensure things went smoothly, I used printf to know all the values of each thread at any given time.

Acknowledgements:
Man pages, course slides, AUPE, and
https://www.geeksforgeeks.org/use-posix-semaphores-c/ to help understand semaphores and pthreads better.