```
/*Author: Samuel Casto
 *PantherID: 6330314
 *Description: This program will accept a single int input and perform the collatz algorithm on it
and the number that is the input + 6. These should run at  *the same time and update with their
current value in their algorithm. The input will need to be greater than zero and less than 40 and
the program will     *verify the input is correct.
 *
 *Task 2:The processes to me did not always finish in the order in which they were forked even
for the same input value, but I am basing that on the
 *output I received in my terminal. I think this is a rendering “issue” when using Putty as even
though I coded the children to run concurrently my output
 *was often divided into one child going first and finishing before the other child printed their
initial value. Sometimes, the output would be mixed like
 *the example and other times they would be as I described. I also believe that if I read more
about how concurrency is implemented then I might know why my
 *output is like that. My operating systems class has given me enough knowledge to think that
the CPU switches between the two tasks and simply swaps to
 *one child, finishes that recursion stack, and then switches back to the other child and finishes
that stack but I’m not sure since there are forks
 *involved and I need to check my notes if I am even remembering correctly what I learned last
week.
 *
 * */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>


void collatz(int input, int child){
        //might do our printing in here as well so we will need to know which child along with input
```

```
    if(input == 1){
        //base case
        printf("child %d: %d\n",child,input);

    }
    else if(input % 2 == 0){
        //number was even
        printf("child %d: %d\n",child,input);
        collatz(input / 2,child);

    }
    else if(input % 2 == 1){
        //number was even
        printf("child %d: %d\n",child,input);
        collatz(input * 3 + 1,child);

    }

}

int main(int argc, char **argv) {
    //adding command line parsing code
    extern char *optarg;
    extern int optind;
    int input,input2;//variables for holding user input

    //using a switch statement because this is modified code for an assignment from a previous unix class. Code can be found on my github,
    // I think assignment 5. Probably would have been easier to just watch a YouTube video to reremember how to parse a single value and validate it
    // but here we are. Wow didn't even need the switch statement. I am so smart. I didn't need have the code I tried to make work from previous
    // assignments. I am so smart.


    //checking for an input value to perform operations on
    if (optind >= argc) {
        fprintf(stderr, "Expected intval input\n");
        fprintf(stderr, "usage: collatz intval\n", argv[0]);
        return(1);
    }


    //if we are here then we have an input that we should try to assign
    input = atoi(argv[1]);
    //used input2 because I'm afraid of C
    input2 = input + 6;
    //verifying our input meets our wanted criteria
    if((input <= 0) || (input >= 40)){
        //if true then input is not larger than zero or less than 40
        printf("Input should be larger than zero and less than 40\n");
        exit(1);
    }


    //creating our child processes and forking them
    pid_t child1, child2;
    child1 = fork();


    if(child1 == 0){
```

```
    //fork executed succesfully
    printf("child1: init input = %d\n",input);
    //need to call a recursive function as a switch statement won't work
    collatz(input,1);
    //when this is finished we need to print a statement printing that
    printf("child1: I finished pid: %d\n",getpid());
    //we need to call a return value to close the fork I think
    exit(0);
}

child2 = fork();
if(child2 == 0){
    //fork executed successfully
    printf("child2: init input = %d\n",input2);
    //calling our recursive function
    collatz(input2,2);

    printf("child2: I finished pid: %d\n",getpid());
    //sending a return value
    exit(0);
}



//parent code starting here
printf("This is the parent waiting\n");
printf("Parent: Child1 process created with PID %d\n",child1);
printf("Parent: Child2 process created with PID %d\n",child2);
//waiting for the processes to finish
```

```
    wait(NULL);
    wait(NULL);

    printf("Parent: Child processes finished.\n");



    return 0;
}
```