```
/*
 * @file hw4_jobs.c
 * @author David Holmes
 * @brief
 * @version 0.1
 * @date 2022-11-20
 *
 * A program to run and show running jobs in the command line
 * Compiles with 'gcc -Wall jobstuff.c hw4_jobs.c -lpthread -o hw4'
 * Links the pthread library and the queue functions.
 * Runs with './hw4 <number of cores>' e.g. './hw4 3'
 *
 * Excuse the mass of comments everywhere, they're mostly for my own figuring out code
placement
 * Fun fact: It took me a very long time to realize that I had to edit the queue to work with jobs
 * A very long time.
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>
#include <string.h>
#include <fcntl.h>
#include <stdint.h> // found out about this for int typcasts

#include "jobstuff.h"

#define TRUE 0
#define FALSE 1
#define BUFFERSIZE 1024 // Not going to lie, I just looked up "what is a good buffer size" to
determine this
                        // and was recommended use of a base 2 number, of which 1024 was suggested
#define MAXJOBS 100
#define MAXQUEUE 50

queue *jobQueue; // Made this a global variable due to being unsure of how to update it across
methods
int j_running; // To keep track of how many jobs are running
job jobList[MAXJOBS]; // Array of all jobs; made this global for use in job_handler
```

```
// job structure to hold information about each job being run
// Implementation inspired by work on Lab 12, which also used structures to run a program on threads
/*
/
/
*/

// Job-centric commands
// runs each job
void *job_runner(void *arg) {
    job *currJob = (job *)arg; // Typecast back to get structure
    char **argv = malloc(sizeof(char *)); // Command for the job
    pid_t ch_pid; // Child's pid (the process id)
    int status;
    int fdout, fderr;

    j_running += 1; // Add one job to the number of currently running jobs
    currJob->status = "Working...";

    ch_pid = fork(); // Fork for execvp

    // Child Process
    if (ch_pid == TRUE) {
        // 0755 is read, execute, write access for all, only owner explicitly can write
        // Based off example from lecture 28
        if ((fdout = open(currJob->outFile, O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
          printf("Error opening %s for redirection and output.\n", currJob->outFile);
        }
        if ((fderr = open(currJob->errFile, O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
          printf("Error opening %s for redirection and output.\n", currJob->errFile);
        }
        dup2(fdout, 1);
        dup2(fderr, 2);

        char *tempc = malloc(sizeof(char) * (strlen(currJob->job_comm) + 1));
        strcpy(tempc, currJob->job_comm);

        // Consulted the below link for behavior involving tokenizing a string into an argv array
        // https://www.gnu.org/software/libc/manual/html_node/Finding-Tokens-in-a-String.html
        char *token;
        int i = 0; // Current position in argv array
        while((token = strtok(tempc, " ")) != NULL) {
            argv[i] = malloc(sizeof(char) * (strlen(token) + 1));
```

```
            strcpy(argv[i], token);
            ++i;
            tempc = NULL; // NULL char is at the beginning of the future tokens, set tempc to NULL
to look for next
                        // string there.
        }

        // RUn the job
        execvp(argv[0], argv);

        // Only runs if execvp fails
        fprintf(stderr, "Failed execution of given command '%s'\n", argv[0]);
        exit(-1);
    }
    // Parent process
    else if (ch_pid > 0){
        waitpid(ch_pid, &status, WUNTRACED);
        currJob->status = "Complete."; // Change job status to completed
        // Report if the child process didn't exit normally
        if (status == -1) {
            fprintf(stderr, "Error running the child process '%d'.\n", ch_pid);
        }
    }
    else {
        fprintf(stderr, "Error performing the fork() operation.\n");
        exit(-1);
    }

    j_running -= 1;

    return (NULL);
}

// Handles the jobs, lol
// In seriousness: performs operations on each job in the queue
void *job_handler(void *arg) {
    int maxrun = (intptr_t) arg;
    job *currJob;

    // continously check for jobs
    while (1) {
        // Do job stuff as long as there isn't max jobs running and the queue isn't empty
        if (j_running < maxrun && jobQueue->count > 0){
            currJob = queue_delete(jobQueue); // Pop first job off the top of the queue
```

```
            pthread_create(&currJob->jtid, NULL, &job_runner, (void *)&currJob); // make a thread to run the job
            pthread_detach(currJob->jtid); // Frees up resources; Use this over exit due to no threads needing to be joined
        }
        sleep(3);
    }

    return (NULL); // Doing this because that's how it's done in the lab12 examples - why do we need to return NULL?
}

// Main function, initializes program and handles user input
int main(int argc, char **argv) {
    // User input stuff
    int p_cores;
    int isActive; // Determines if program should still be running
    char input[BUFFERSIZE]; // entirety of user input
    char *usr_cmd; // variable to store user command
    char *arg_cmd; // Program specified to be used by the sumbit command
    char *arg_cmdArg; // I think this variable name is funny; used if the argument has arguments
    pthread_t tid; // obligatory tid

    // Job stuff
    jobQueue = queue_init(MAXQUEUE); // initialize queue with job amount cap od MAXQUEUE (50 here)

    // Predefined stuff
    // char *poss_cmds[3] = {"submit", "showjobs", "quit"};

    // Program starting only accepts one argument.
    if (argc > 2) {
        printf("Too many arguments given. Usage: %s <number of cores>", argv[0]);
        exit(-1); // Exit with -1 to show issue with program
    }
    else if (argc < 2) {
        printf("No core input given, defaulting to usage of P = 2 (two cores).\n");
        p_cores = 2;
    }
    else {
        p_cores = atoi(argv[1]); // p_cores is argv[1], which is the number of core specifed by the user
    }
```

```
    isActive = TRUE;
    printf("Welcome. \nAvailable commands: \nsubmit <program> <arguments> | showjobs\n");
    printf("Use command 'quit' to exit.\n");

    pthread_create(&tid, NULL, &job_handler, (void *)&p_cores); // Do stuff with all the jobs while the "UI" is running

    // The horrific block of user input handling
    int i = 0;
    do {
        printf("Enter command. >> ");
        if ((fgets(input, BUFFERSIZE, stdin) != NULL) && ((usr_cmd = strdup(strtok(input, " \n"))) != NULL)){
            // User command provided is "submit"
            if (strcmp(usr_cmd, "submit") == TRUE && i < 100) {
                arg_cmd = strdup(strtok(NULL, " \n"));
                if ((arg_cmdArg = strdup(strtok(NULL, " \n"))) != NULL) {
                    strcat(arg_cmd, " ");
                    strcat(arg_cmd, arg_cmdArg);
                }
                jobList[i] = init_job(i, arg_cmd);
                queue_insert(jobQueue, jobList + i);
                printf("Job %d has been added to the queue.\n", i++);
            }
            // User command provided is "showjobs" or "quit"
            else if (strcmp(usr_cmd, "showjobs") || strcmp(usr_cmd, "quit")) {
                if (strcmp(usr_cmd, "showjobs") == TRUE) {
                    show_jobs(i, jobList);
                }
                else if (strcmp(usr_cmd, "quit") == TRUE) {
                    printf("Exiting program.\n");
                    isActive = FALSE;
                }
                // If this shows at all, something went incredibly wrong
                else {
                    printf("An error has occured. Exiting...\n"); // You literally should not see this
                    exit(-1);
                }
            }
            else {
                if (i > 99 && strcmp(usr_cmd, "submit") == TRUE) {
                    printf("The maximum number of jobs in memory (%d) has been reached. Please restart the program if you would like to add more.\n", MAXJOBS);
                }
```

```
            else {
                printf("Unrecognized and/or unsupported command.\n");
            }
        }
    }

} while(isActive == TRUE);

return 0;
}
```