HW4 Participants: Piotr Milewski

**Requirements:**

- Server is able to run concurrent jobs with the max job number configuration
- The server is able to track necessary resources for each running process (memory usage, time spent running) and be able to kill/stop/change the priority of each job.
- The server is able to report back to the client with the job status of one or more jobs
- The client is able to issue commands to the server and the server should return the correct info for each command issued.
- The client is able to submit new jobs and have them be run successfully.
- A pipe is correctly set up to allow read/write functionality for both the client and the server.
- A pipe communication protocol with a reasonable format is used between the client and the server.
- The server can wake up, read from the pipe, perform the requested functionality, respond to the client, and go back to sleep to await further input from the pipe.
- The client can read the pipe and process/display the information from the pipe.
- Different client/server command support to return the output of a command.

**Design:**

- Protocol - My system has 2 protocols, one for sending data to the client and the other for sending data to the server. Both are structs that are sent as a sequence of bytes and the recipient can cast the data to a struct and read it. For example, when the server wants to determine the command that the client sent it can simply get the appropriate field from the struct like: toServer.commandType.
- Server protocol:
    - char commandType - This specifies the type of the command that the client is requesting the server to perform. Here are the 8 types of commands that the client can request and their corresponding character:
        - 1 - request to submit a new job, must also specify the command, the environmental variables, how many strings there are in the commands, and the job limits
        - 2 - list all the jobs that the client has submitted along with their details (id, name, pid, status (suspended, running, in queue, aborted, completed)
        - 3 - list one job that the client has submitted along with its details, must also specify the job id in the protocol
        - 4 - kill an existing job, must also specify the job id in the protocol
        - 5 - suspend an existing job, must also specify the job id in the protocol
        - 6 - continue an existing job, must also specify the job id in the protocol
        - 7 - change the nice priority of a job, must also specify the job id and by how much the client would like to change the nice value (in the optional field) in the protocol. E.g. if the client provides a 2 in the optional field then the nice value is increased by 2.

        - 8 - return the exit status of a command that the client ran, must also specify the job id in the protocol
    - int jobid - This specifies the job id of a job on which to perform a specified action.
    - int optional - An optional argument for passing things such as by how much to change the nice value of a specified job.
    - int argc - The number of arguments plus 1 for the command name that are being passed to the server when submitting a new job.
    - char argv[4096] - Holds the name of the command that is being sent to the server
    - char envp[4096] - Holds the environmental variables that are being sent to the server in the form of a string
    - job_limits jobLimits - A struct that provides limits for the job to be run. These are described below.
        - int maxTime - The maximum amount of time the program can be run.
        - int maxMemory - The maximum amount of memory the program can use.
        - int priorityValue - The program's priority value.
- Client protocol:
    - char replyType - This specifies the type of reply that the client is expected to receive from the server. The 2 types of replies are defined below:
        - 0 - reply to the client for a request that isn't a job submission request. This tells the client to ignore the optional element of the struct.
        - 1 - reply to the client for a job submission request. This tells the client that the optional element of the struct will contain an integer that specifies whether the job was successfully started or if it is in queue. If the job was successfully started then the optional element will contain a 0. If the job is in queue to be run (since the maximum number of jobs are currently being run) then the optional element will contain a 1.
    - int optional - This is an optional element in the protocol used by some of the reply types.
    - Char buf[4096] - This contains a reply from the server and it depends on what the client requested. If the client requested a list of the jobs to be run then it will be held here in the form of a string. The client must read from this buffer after every request sent to the server to obtain a message regarding the request.
- Client/Server interaction:
    - The client and the server communicate in the form of UNIX domain sockets.
    - The server will set up a UNIX domain socket called domain_socket. In order to ascertain that it was created, after running ./server you should be able to ls and find a file of type socket called domain_socket in the same directory.
    - The client can then set up its own socket and connect to the domain_socket, since it knows the file path. Upon connection, the server will store the file descriptor of the client's socket for future communication and add it to the list of file descriptors that it is listening to for instructions.
    - Using the select(2) method, the server listens for activity on its domain_socket and the client_sockets.

        - If there is activity on the domain_socket then a client is attempting to connect to the server and the server will wake up, connect to the client and store its file descriptor.
        - If there is activity on one of the client sockets then the server will wake up, read from the active pipe, and fulfill the request from the client.
    - When a client disconnects, the server will send a SIGKILL to all the jobs that the client had running as well as free and remove the jobs from the jobs table that the client had running. This is to ensure that if another client connects and decides to use the same socket that the removed client used, then the newly connected client won't have access to the jobs in the job table that the removed client added.
    - In order for the server to send a protocol to the client, a method exists in protocol.c called createClientProtocol() which will create a struct based on the arguments and return it to the server. The server can then send this struct to the client. This guarantees that the server doesn't instantiate anything in the protocol incorrectly.
    - In order for the client to send a protocol to the server, 2 methods exist in protocol.c called serverProtocolNewJob() and serverProtocolRequest(). These methods work similarly to createClientProtocol() except serverProtocolNewJob() is for setting up the struct to send a job request and serverProtocolRequest() is for any other command.
- Server's job table:
    - The server has a job table where it stores all the information on submitted jobs.
    - Each table entry contains: the name of the command being run, the job's pid, the job's id, the job's status (0 = suspended, 1 = running, 2 = in queue, 3 = completed, 4 = aborted), the file descriptor of the client that owns the job, a timeval struct for when the job was created, the job's exit status, a job_limits struct that contains the maximum memory and time that the job can use, and a pointer to the next job.
    - If a client requests to see the entries of the jobs then the server can tell which job is the client's by comparing the client socket's file descriptor with the file descriptor in each entry. If they are the same then the client owns that job and can perform actions on it.
    - When a job request is finished, the job table continues to store the job in the job table until the client disconnects. This is used for if the client decides it wants to view the exit status of the job later on. Also when a job request is finished, the server will check to see if any jobs are in queue and will start them up.

**Implementation:**

1. I started off this homework by experimenting with my pipe/domain socket implementation. I was quite stumped as to which to use and how each of them differed but in the end I opted to use domain sockets for this homework since they were basically the same in functionality as pipes except they specified the endpoints of communication so it would be much easier to have one socket for the server upon which the clients could send messages, and a socket for each client to facilitate how the server can send messages to each individual client.
2. Next I started implementing a job handler which I basically copied from my hw3 implementation. The client would submit a job to the server, the server would then fork and the child would use execvpe(3) to run the job in the background. In the meantime the server will add the job to the

table and keep tabs on it, making sure it doesn't exceed its memory limits or its time limits that the client specified.

3. Next I started implementing the other commands that the client would be able to specify. These are listed in the server protocol in this document. With every command I made sure that the server would send back a coherent response to the client letting the client know if the command succeeded and if the client requested a piece of information then the server would send that back as well.
4. During this process I had been using client.c, a file which would generate an executable that when connected to the server would send a couple of commands to the server to test if the functionality I had been implementing worked. I then copied that file and made a couple others in a similar format titling them client1.c, client2a.c, etc. and each would create an executable that when run, would ensure that the functionality I implemented worked.
5. After the test scripts were done, I started working on this design document.

**Included Files:**

- baseHeaders.h - contains all the basic headers that are needed for this assignment, gets included in all the other header files
- server.c server.h - sets up the server socket and handles requests upon the socket as well as the requests of clients that have already connected
- exec.c exec.h - handles the execution of requests accepted by the server
- jobs.c jobs.h - stores the job submission requests in a jobs table and handles changes that need to be made to certain jobs (suspending a job, changing the nice value, etc.)
- protocol.c protocol.h - defines the protocols needed for the server and client and provides functions for easily setting up the protocols to be sent.
- client.c client.h - contains a function to handle connecting to the server socket
- client1.c - used to demonstrate that all the command types (1-8) work as intended
- client2a.c client2b.c client2c.c - used to demonstrate that multiple clients can connect to the server and if the server is set to max jobs of 2 then one client's job will be queued up and only started up if one of the other clients' job finished
- client3.c - used to demonstrate that job limits can be set for jobs and they work as intended
- makefile - used for compiling the different parts of the assignment (more on this in usage)

**Usage:**

- Various make choices have been provided and they are described below.
    - make - will create the ./server and all the ./client examples that I have provided
    - make server - will only create the ./server
    - make client - will only create the ./client examples
    - make clean - will cleanup the directory
- In order to run the test scripts, first run the regular 'make' and then startup the server using the command './server'.
    - In order to run the first script simply run ./client1
    - In order to run the second script, run ./client2a, ./client2b, and finally ./client2c. You will see that the job request for ./client2c is put in the queue. In order to see ./client2c's job be

started up kill either ./client2a or ./client2b and the server will print out a message that ./client2c's job is currently running.
    - In order to run the third script, simply run ./client3

**Extra Notes:**

- I did not manage to make a command that will tell the client what output their jobs put in stdout/stderr. I didn't really know how to store/retrieve each stdout/stderr and at this point I'm feeling pretty overwhelmed so I just decided to call it quits on that functionality. Everything else that the professor specified should work just fine.