[Protocol]

The first step in establishing this network-like relationship between a client and server is to establish a network-like protocol. Following concepts from protocols like TCP, I created a standard format of communication from client to server. Since the kind of information returned to the client is different per request, I created several formats of responses of communication from server to client.

I established a Request struct by which the client can send to the server so they could process the request. It represents an out-of-band data structure where the size of the payload is described by the size field before it. This provides a versatile means of communication because that payload can transmit variable size commands to the server. The command can then be parsed depending on the code at the head of the struct. That code field can take on the server's current functionalities, such as submitting a job, viewing the statuses of jobs (or a single job), issuing a signal to the process the job runs on, clearing jobs that have finished, or shutting down the server. This way, the server and client can agree on the means of communication and the server knows exactly what to do with the code provided from the client.

I decided to make the job submission done via a string, which is slightly more effort for the user but ensures that I can safely and reliably provide a valid argument array to execute the job with. The same applies for signal sending, which requires a string with the job id to send to alongside the signal to send. Although slightly more work for the user, it avoids the overhead of requiring more structs and complicated tokenization.

```
struct Request // standard format for requests
{
    int code;        // type of request
    int size;        // size of command
    char payload[1]; // command
};
```

I established several response structs which the server can use to send data back to the client. It wouldn't make sense to include a payload to return to the client because all the necessary return data has fixed lengths. Therefore, to be able to distinguish between the types of responses, I created several structs that are easily transmissible and easily read. The JobResponse struct simply confirms the submission of a job. The JobStatus struct carries a status for the corresponding job id and can be returned and read multiples at a time. The JobSignal struct confirms the sending of a signal to the corresponding job id.

```
struct JobResponse // standard format for job submissions
{
    int status_code; // success
    int job_id;
};

struct JobStatus // standard format for status reports
{
    int job_id;
    int status; // current status of job
    int ret;    // return code of job
};

struct JobSignal // standard format for signal request responses
{
    int job_id;
    int signal;
};
```

[Project Structure]

There was a natural divide between client and server, which was the fact that they were different executables. I decided to have a folder for both to include files that they required individually to carry out their respective duties first, and nextly possible files they both require. The data that both required ended up being protocol.h, which established all the structs and macros necessary for client and server communication. Otherwise, they'd have their own separate makefiles because they can require different dependencies, separate includes as a result, and separate macros for ease of use. For example, a custom status had to be introduced in the server to indicate a running job, as no signals indicate an initially running child. The shared file is symlinked in both folders to ensure that both client and server are up to date as per the protocol.

With the client is simply a short program to parse the type of request the user is aiming for and to first create the request struct and populate it with the appropriate codes and payloads. After sending the request, the client then waits to receive a response struct from the server or simply prints out a confirmation of the action if the server does not need to send data. This was performed using a global options set of flags where the client sends the request based on flags set in the first half, and then using those same flags in the second half of the program to determine how to process a response appropriately.

The difference with the server is that it requires to be run in a loop to achieve constant run time and an arsenal of functions to handle and process requests from the client. The primary functionalities are reflected in the server functions to submit a job, display the statuses of jobs, send a signal to jobs, and to erase jobs. Those were abstracted away since they represent core functionality. The rest of the functionality were small helper functions, such as the ones to manage the jobs storage array or to find the next free index for job storage. The server also requires the SIGCHLD handler, which is the only way to know if a child has changed state, and the program always jumps to the handler to set the state immediately.

[Design Choice]

I decided to make the client as simple as possible by only letting it pass flags for the main functionality and to pass complicated commands as a string. Even though slightly more effort for the client to wrap their command in quotes, it makes handling the payload of the request much simpler. It also gives the user more freedom since they can pass more complex commands that would be difficult to parse in individual units such as “ls -l”, where the “-l” would be required to be interpreted.

In terms of communications, the way my established structs would be transmitted was going to be pipes. In order to avoid race conditions and possible collisions, I decided on two separate pipes for client to server where client writes and server reads, and server to client where server writes and client reads. These pipes would be temporary and would be reset per request to ensure that the pipes are clean. At each step in the server loop, the pipes clean up would be easily handled via a custom function that closes pipes or unlink files based on the progress of the loop. For example, if an error occurred because the second pipe failed to open, it would only close the first pipe that was already opened and wouldn’t unlink any files since they were never created.

I decided to have the client close jobs that were finished. The primary reason was because the client might want to view information relevant to the jobs that they requested and wouldn’t be able to do so if they were simply erased automatically. A lot of programs are short, so I allowed the user to keep track of jobs that they’ve requested in the past and erase the ones that they deem are no longer needed. The same applies for the output of these functions, which I left to the user to manipulate. The user can come back to those files as they needed and each job id file also came with a job id error log in the event that something breaks during command execution.

The data structure I chose to use was simply an array. Apart from a linked list, this was the most practical data structure to use. The strategy was to populate the first available spot, which was a matter of finding the first null position in the array, and go left to right. The user can determine the maximum number of jobs to run, which I felt 64 was a decent power of 2 to set the default at. Once the array is fully populated, the user must choose jobs to clear, given the benefit of reviewing past jobs and having the power to select which ones to erase.

[Restrictions and Usage]

The client is only allowed to pass one flag at a time. For job submission and signal sending specifically, the client must provide their command in a string for parsing. Everything else either only requires an extra integer to indicate id or no extra requirement at all.

The max number of jobs can either be set or is a default value of 64. The maximum command length is also 64, since any command that requires more than 64 arguments should be split up or is probably being executed incorrectly.

[Functionality]

Job Submission:

Like with the interactive shell, I decided to handle job submission via forking the requested command into another process. This way, the server can remain running and issues with the command will be logged and will not interfere with the server's running process. Jobs are submitted into an array that the user will be responsible for clearing by themselves. It also features a signal handler that will indicate when the requested process has terminated and will set the appropriate job's status.

Signals:

The user can pick a target process via the job id and the server will find the matching pid corresponding to the process the job's command is running on. It will then send the requested signal via the kill command and the child signal handler will handle the change of state that results from it.

Statuses:

Depending on whether the user provides an index, the server will either print all statuses or selectively the status of the provided job id. To achieve this, the server iterates across the entire array of jobs, fills a response struct, and sends the struct over to the client, or target a specific index to print and repeat the same process.