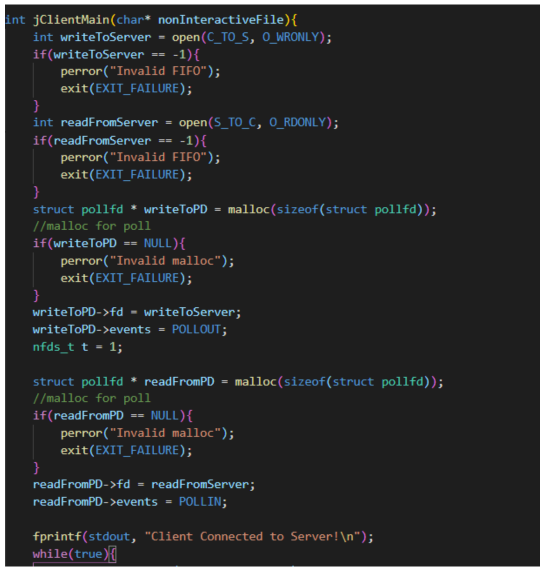
CSE 376 Homework 4 Zhenghao (Kevin) Zhu
5/4/2021 111736904

File Structure:

The project is organized in a very simple to understand structure. We start at the most important folder which is the "src" folder. The src folder contains main.c, jClient.c, and jServer.c. The main file only includes the main method and is used as the intermediary for the client and server. It serves so the program is compiled into one single executable named ajs, which stands for "A Job Scheduler". The jClient.c and jServer.c files contain their respective code which we will go over in the Code Logic section. The include folder has three header files which are used in their respective .c files explained previously in the src folder. The jServer.h file contains a struct definition for the job_status struct which saves information pertaining to the specific job process. All of the jobs executed by clients using the server have a node in a linked list made up of nodes of these job_status structs. The tests folder includes 3 shell scripts with their respective txt files to test the program in a non-interactive manner. All the tests execute different functionalities of the program. The output files of the tests are placed in the root folder of the project. The tmp file generated during the tests contains .cToS and .sToC which are "client to server" and "server to client", respectively. These files are the FIFO pipes used throughout the program execution.



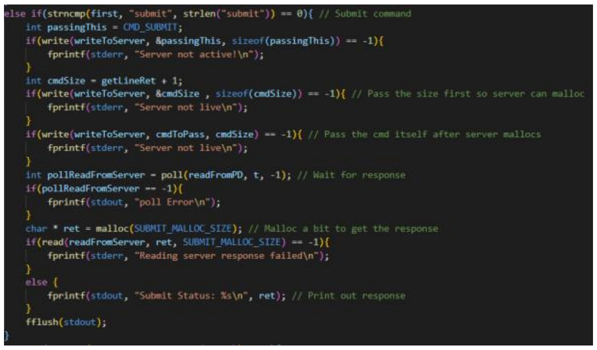
```c
int jClientMain(char* nonInteractiveFile){
    int writeToServer = open(C_TO_S, O_WRONLY);
    if(writeToServer == -1){
        perror("Invalid FIFO");
        exit(EXIT_FAILURE);
    }
    int readFromServer = open(S_TO_C, O_RDONLY);
    if(readFromServer == -1){
        perror("Invalid FIFO");
        exit(EXIT_FAILURE);
    }
    struct pollfd * writeToPD = malloc(sizeof(struct pollfd));
    //malloc for poll
    if(writeToPD == NULL){
        perror("Invalid malloc");
        exit(EXIT_FAILURE);
    }
    writeToPD->fd = writeToServer;
    writeToPD->events = POLLOUT;
    nfds_t t = 1;

    struct pollfd * readFromPD = malloc(sizeof(struct pollfd));
    //malloc for poll
    if(readFromPD == NULL){
        perror("Invalid malloc");
        exit(EXIT_FAILURE);
    }
    readFromPD->fd = readFromServer;
    readFromPD->events = POLLIN;

    fprintf(stdout, "Client Connected to Server!\n");
    while(true){
```

Code Logic:

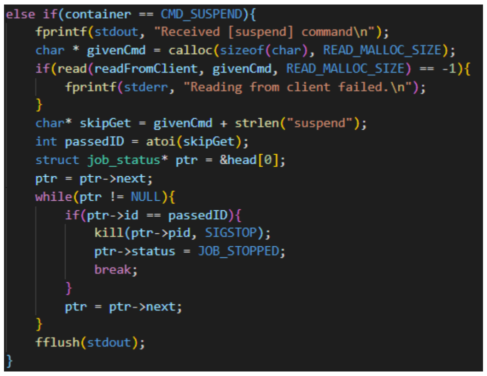
The setup of the FIFO pipes, in the jClient.c file, is shown in the image in the left side. I'm trying to check that all pipes are readily available and creating poll file descriptors so I'm able to read/write to them. This is all done and checked thoroughly before entering the infinite while loop shown I the last line. The setup is very similar in the jServer.c file. The next picture shows the logic of a write/read cycle for one of the most important commands, submit. The logic goes like this, the client writes a single number to the server to tell it what other information it should expect, in this case we are writing an integer 3 to the server. From there on, the client sends the actual command to the server so it can be processed and waits (polls) until the server responds back with the status of the command. The logic is similar in writing and reading as we need to



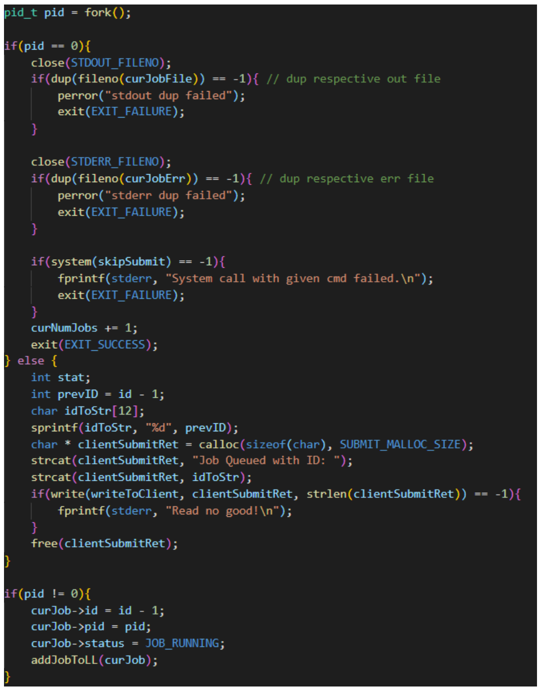
```c
else if(strncmp(first, "submit", strlen("submit")) == 0){ // Submit command
    int passingThis = CMD_SUBMIT;
    if(write(writeToServer, &passingThis, sizeof(passingThis)) == -1){
        fprintf(stderr, "Server not active!\n");
    }
    int cmdSize = getLineRet + 1;
    if(write(writeToServer, &cmdSize , sizeof(cmdSize)) == -1){ // Pass the size first so server can malloc
        fprintf(stderr, "Server not live\n");
    }
    if(write(writeToServer, cmdToPass, cmdSize) == -1){ // Pass the cmd itself after server mallocs
        fprintf(stderr, "Server not live\n");
    }
    int pollReadFromServer = poll(readFromPD, t, -1); // Wait for response
    if(pollReadFromServer == -1){
        fprintf(stdout, "poll Error\n");
    }
    char * ret = malloc(SUBMIT_MALLOC_SIZE); // Malloc a bit to get the response
    if(read(readFromServer, ret, SUBMIT_MALLOC_SIZE) == -1){
        fprintf(stderr, "Reading server response failed\n");
    }
    else {
        fprintf(stdout, "Submit Status: %s\n", ret); // Print out response
    }
    fflush(stdout);
}
```

preface the big "chunk" of the data with a length variable to tell the other side what type of data to expect and what size it must allocate to store it. This logic is the same throughout all command processing in the client.

The logic for the command processing in the server is different though we take for example the command **suspend** which stops a process from running but doesn't completely terminate it like **kill**. The command iterates through the linked list trying to the find the node with the matching ID, if it finds then it issues a SIGSTP to the PID associated to the node which in turn suspends the process.



```c
else if(container == CMD_SUSPEND){
    fprintf(stdout, "Received [suspend] command\n");
    char * givenCmd = calloc(sizeof(char), READ_MALLOC_SIZE);
    if(read(readFromClient, givenCmd, READ_MALLOC_SIZE) == -1){
        fprintf(stderr, "Reading from client failed.\n");
    }
    char* skipGet = givenCmd + strlen("suspend");
    int passedID = atoi(skipGet);
    struct job_status* ptr = &head[0];
    ptr = ptr->next;
    while(ptr != NULL){
        if(ptr->id == passedID){
            kill(ptr->pid, SIGSTOP);
            ptr->status = JOB_STOPPED;
            break;
        }
        ptr = ptr->next;
    }
    fflush(stdout);
}
```



```c
pid_t pid = fork();

if(pid == 0){
    close(STDOUT_FILENO);
    if(dup(fileno(curJobFile)) == -1){ // dup respective out file
        perror("stdout dup failed");
        exit(EXIT_FAILURE);
    }

    close(STDERR_FILENO);
    if(dup(fileno(curJobErr)) == -1){ // dup respective err file
        perror("stderr dup failed");
        exit(EXIT_FAILURE);
    }

    if(system(skipSubmit) == -1){
        fprintf(stderr, "System call with given cmd failed.\n");
        exit(EXIT_FAILURE);
    }
    curNumJobs += 1;
    exit(EXIT_SUCCESS);
} else {
    int stat;
    int prevID = id - 1;
    char idToStr[12];
    sprintf(idToStr, "%d", prevID);
    char * clientSubmitRet = calloc(sizeof(char), SUBMIT_MALLOC_SIZE);
    strcat(clientSubmitRet, "Job Queued with ID: ");
    strcat(clientSubmitRet, idToStr);
    if(write(writeToClient, clientSubmitRet, strlen(clientSubmitRet)) == -1){
        fprintf(stderr, "Read no good!\n");
    }
    free(clientSubmitRet);
}

if(pid != 0){
    curJob->id = id - 1;
    curJob->pid = pid;
    curJob->status = JOB_RUNNING;
    addJobToLL(curJob);
}
```

Next, we have the logic of the **submit** command which uses the system(2) system call to execute the command provided by the client. Before executing the command though, it dup(2) both the stdout and stderr so files can be created from the command execution. This way, the user will be able to query the command output and errors even after the command is done running.

Running Instructions:

To run the program, cd all the way to root folder. From there, type "make tests" and hit enter. This will prompt the Makefile to execute the shell scripts located in the tests folder. You will be able to see output in the terminal and many output files will spawn in the root directory.

If you want to run the program normally, you will need to execute the ajs file. First set up the server by executing the file like this: "./ajs -s [Max Number of Jobs That Can Run Concurrently], after the server is ready and waiting, then execute "./ajs -c [**Optional** file path for non-interactive mode]". **NOTE**: This must be done in two separate terminals or by putting the server in the background using "&". You will then be prompted by some text in both terminals or a singular terminal if you are running the server in the background. From there on, you can any of the following commands: **submit** [command you want to submit], **get** [ID of jobs given to client], **list** (to list out all the jobs that have been ran/scheduled), **suspend** [ID] to suspend a job, **continue** [ID] to continue a job, **kill** [ID] to kill a job, and **exit** to exit out of the client.

Other Things and Extra Credit:

The function for the "get" command which displays output was mainly taken from this link with some very slight modifications. https://stackoverflow.com/questions/2029103/correct-way-to-read-a-text-file-into-a-buffer-in-c This is a common way of reading a file to a buffer which simplified the logic of the "get" command.

The server supports multiple clients connected to it. They all share the same linked list of job_status structs but each individually is able to schedule jobs. This is part of my EXTRA_CREDIT.