Project 3 Unix Shell

In this project our task was to create an Unix shell. The shell should be able too execute commands from a given path and it should have three built in commands cd, path and exit. The shell should also allow parallel execution with “&” operator and redirection to a file with the “>” operator. The shell should be able to be used in either interactive mode where the shell reads input from stdin or in batch mode where the shell reads input from a file. In this project I also used the GitHub Copilot AI to help generate comments for my functions. All the functions were built by me but some of the comments were generated by AI based on the code I wrote.

Implementation

How I implemented this was with a dynamically allocated array of strings. First of all I simply implemented the reading of an input stream using the getline function. Most of the functionality is built into the shell function. It takes the output stream as an parameter if it is stdin we print the prompt otherwise not. Then II focused on string tokenization to get the individual arguments and commands from the line for this the strok function was used. All of this is shown in picture 1.

```c
void shell(char ***paths, FILE * stream) {
    char *command = NULL;
    char ***commandArray = NULL;
    char *token = NULL;
    char **argArray = NULL;
    size_t size = 0;
    __ssize_t line;

    while(1) {
        if (stream == stdin) {
            fprintf(stdout, "wish> ");
        }

        line = getline(&command, &size, stream); // Read a line from the input stream
        if (line == -1 ) {
            // EOF reached
            break;
        }
        if (line == 1) {
            // Empty command
            continue;
        }

        command[line - 1] = '\0'; // Remove the newline character

        // Tokenize the command
        token = strtok(command, " ");
        while (token != NULL) {
            addArgument(&argArray, token);
            token = strtok(NULL, " ");
        }
```

Picture 1. the first part of the shell function

Then I add the tokenised arguments into the argument array (argArray) using the addArgument function shown in picture 2. This is a dynamically allocated array of strings. I got help for implementing this from a Stack Overflow post (“c - Dynamically create an array of strings with malloc - Stack Overflow,” n.d.). Then we check if we find an inbuilt command and execute it.

```c
void addArgument(char ***array, char *arg) {
    // This post helped with creating a dynamic array https://stackoverflow.com/questions/5935933/dynamically-create-an-array-of-strings-with-malloc

    if ((*array) == NULL) {
        (*array) = malloc(sizeof(char*) * 2);
        (*array)[0] = malloc(sizeof(char) * (strlen(arg) + 1));
        if (((*array) == NULL) || ((*array)[0] == NULL)) {
            fprintf(stderr, "malloc failed");
            exit(1);
        }
        strcpy((*array)[0], arg);
        (*array)[1] = NULL;
    } else {
        int elements = arrayLength((*array));

        // realloc the array to add a new element
        (*array) = realloc((*array), sizeof(char*) * (elements + 2));
        (*array)[elements] = malloc(sizeof(char) * (strlen(arg) + 1));

        if (((*array) == NULL) || ((*array)[elements] == NULL)) {
            fprintf(stderr, "realloc failed");
            exit(1);
        }

        strcpy((*array)[elements], arg);
        (*array)[elements + 1] = NULL;
    }

    return;
}
```

Picture 2. The addArgument function

After this the argument array is split into a command array if we find the “&” operator for parallel execution. If it is found the command before the operator and after it will be split into two different arrays and added to the command array. The construction of the command array is shown in picture 3.

```c
    // if not a built-in command, we will start finding the command in the paths
    // and execute it if it is found
    for (int i = 0; i < arrayLength((*paths)); i++){
        // allocating memory for the command array
        // Maximum number of commands is the number of arguments
        commandArray = malloc(sizeof(char**) * (arrayLength(argArray) + 1));
        commandArray[0] = malloc(sizeof(char*) * (arrayLength(argArray) + 1));
        if (commandArray == NULL || commandArray[0] == NULL) {
            fprintf(stderr, "malloc failed");
            exit(1);
        }
        commandArray[1] = NULL;

        int commandCount = 0;
        int argCount = 0;

        // Loop through the arguments and split them into commands
        // based on the "&" character
        for (int j = 0; j < arrayLength(argArray); j++) {
            if (strcmp(argArray[j], "&") == 0) {
                commandCount++;
                commandArray[commandCount] = malloc(sizeof(char*) * (arrayLength(argArray) + 1));
                if (commandArray[commandCount] == NULL) {
                    fprintf(stderr, "malloc failed");
                    exit(1);
                }
                commandArray[commandCount + 1] = NULL;
                argCount = 0;
                continue;
            }

            commandArray[commandCount][argCount] = malloc(sizeof(char) * (strlen(argArray[j]) + 1));
            if (commandArray == NULL) {
                fprintf(stderr, "malloc failed");
                exit(1);
            }

            strcpy(commandArray[commandCount][argCount], argArray[j]);
            commandArray[commandCount][argCount + 1] = NULL;
            argCount++;
        }
```

Picture 3. The command array construction

After the command array is built we execute each command in parallel by creating a new process for each command where each process calls the executeCommand function on its own and exits once the command has been run. The execute command is shown in picture 4. For executing a number of forks in parallel I used the following Stack Overflow source to help me (“c - fork() 4 children in a loop - Stack Overflow,” n.d.). All sources are also shown in the context in the source code.

```c
void executeCommand(char **argv, char *path) {
    char *commandPath = NULL;

    commandPath = malloc(sizeof(char) * (strlen(path) + strlen(argv[0]) + 2)); // +2 for '/' and '\0'
    if (commandPath == NULL) {
        fprintf(stderr, "malloc failed");
        _exit(1);
    }

    // Construct the command path
    strcpy(commandPath, path);
    strcat(commandPath, "/");
    strcat(commandPath, argv[0]);
    // Check if the command is executable
    if(access(commandPath, X_OK) == 0) {
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            // https://www.youtube.com/watch?v=5fnVr-zH-SE thsis video helped me with redirection
            if (arrayLength(argv) > 1) {
                if (strcmp(argv[arrayLength(argv)-2], ">") == 0) {
                    // Redirect stdout to a file
                    FILE *file = fopen(argv[arrayLength(argv) - 1], "w");
                    if (file == NULL) {
                        fprintf(stderr, "error: cannot open file '%s'\n", argv[arrayLength(argv)-1]);
                        _exit(1);
                    }
                    dup2(fileno(file), STDOUT_FILENO);
                    fclose(file);

                    free(argv[arrayLength(argv)-2]); // remove the redirection part from the argument list
                    free(argv[arrayLength(argv)-1]); // remove the file name from the argument list
                    argv[arrayLength(argv)-2] = NULL; // remove the redirection part from the argument list
                    argv[arrayLength(argv)-1] = NULL; // remove the file name from the argument list

                }
            }
            execv(commandPath, argv);
            fprintf(stderr, "error: execv failed");
            _exit(1);
        } else if (pid < 0) {
            // Fork failed
            fprintf(stderr, "error: fork failed");
            exit(1);
        } else {
            // Parent process
            wait(NULL);
            free(commandPath);
            commandPath = NULL;
        }
    } else {
        free(commandPath); // not found in this path check next path or the command does not exist
        commandPath = NULL;
    }
    return;
}
```

Picture 4. The executeCommand function.

The executeCommand function also checks if there is the redirection operator to redirect the output to a file. For this I got help from a YouTube video (*Redirecting standard output in C*, 2020). This overwrites the stdout and replaces it with the file I choose. The executeCommand function also creates a new process for the command it executes using the execv function.

There is also a helper function called array length which I built. It takes in a NULL terminated array and returns the length of the array. I used this function in many places in this program.

References

c - Dynamically create an array of strings with malloc - Stack Overflow [WWW Document], n.d. URL https://stackoverflow.com/questions/5935933/dynamically-create-an-array-of-strings-with-malloc (accessed 5.10.25).

c - fork() 4 children in a loop - Stack Overflow [WWW Document], n.d. URL https://stackoverflow.com/questions/65202550/fork-4-children-in-a-loop (accessed 5.10.25).

Redirecting standard output in C, 2020.