EI338 Computer Systems Engineering

Project 1

November 3, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Project 2

Exercise 1

In this exercise, we are asked to design a simple shell interface program that accepts the input commands and then executes them by assigning a new process. It is organized into several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

Basic Structure

To start with, we first illustrate how the program is implemented as a whole. After some initializations, we obviously need to read the user input, and parse it so that it can be executed. And then, depending on the type of the command, we will execute it in a desired way. To be more detailed, the structure of **main()** looks like this:

```
int main(void)
{
    /* some initializations */
    /* ———some lines——— */

    /* present the prompt repeated */
    while (should_run) {
        printf("osh>");
        fflush(stdout);

        /* read the user input */
        fgets(cmd, MAX_LINE, stdin);
        cmd[strlen(cmd) - 1] = '\0'; /* delete line-feed */

        /**
         * After reading user input, the steps are:
         * (1) fork a child process
         * (2) the child process will invoke execvp()
         * (3) if command included &, parent will invoke wait()
         */
```

```
        /* deal with the history command as a special case */
        if (!strcmp(cmd, "!!")) {
            /* ----some lines---- */
        }
        else strcpy(cmd_history, cmd);

        /* parse the command */
        wait_flag = parse_cmd(cmd, args, ifile, ofile, &split_pos);

        /* execute the command by creating a child process */
        exe(args, wait_flag, ifile, ofile, split_pos);

        /* clear values */
        for (int i = 0; i < MAX_LINE/2 + 1; ++i)
        {
            free(args[i]);
            args[i] = NULL;
        }
        ifile[0] = '\0';
        ofile[0] = '\0';
    }

    return 0;
}
```

I. Executing Command in a Child Process

First, it is necessary for us to parse what the user has entered into separate tokens and store the tokens in an array of character strings. It is quite easy for us to parse the command by using **strtok(char *str, const char *delim)**. The function returns the first substring parsed by **delim**.

And also, we tackle redirection, concurrent, and pipe here separately:

- If redirection occurs, **ifile** and **ofile** will be set.
- If concurrent occurs, the function **parse_cmd** will reture the flag to indicate whether the parent process needs to wait.
- If pipe occurs, **split_pos** will be assigned a nonzero value.

```
int parse_cmd(char *cmd, char **args, char *ifile, char *ofile, int *split_pos) {
    char *token;
    int i = 0;
    char redirect;

    token = strtok(cmd, " "); /* take the first token */

    while (token) {
        if (!strcmp(token, "<") || !strcmp(token, ">")) { /* redirect */
            redirect = token[0];
        }
        else {
            switch (redirect) {
                case '<':
                    strcpy(ifile, token);
```

```
                    break;

                case '>':
                    strcpy(ofile, token);
                    break;

                default:
                    args[i] = (char*) malloc(sizeof(char) * (strlen(token)));
                    strcpy(args[i], token);

                    if (!strcmp(token, "|")) *split_pos = i;

                    ++i;
            }
        }

        token = strtok(NULL, " "); /* take the next token */
    }

    if (!strcmp(args[i - 1], "&"))
    {
        args[i - 1] = NULL; /* clear '$' */

        return 0; /* wait flag is set to 0 (concurrent) */
    }

    return 1; /* wait flag is set to 1 */
}
```

Once we obtain the parsed command **args** as well as other optional values, we can call the function **exe()** execute it. The basic structure of it is as follows:

```
int exe(char **args, int wait_flag, char *ifile, char *ofile, int split_pos) {
    /* some initializations */
    /* ———some lines——— */

    pid = fork(); /* create child process */

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed.\n");
        return 1;
    }
    else if (pid == 0) { /* child process */
        if (strlen(ifile)) { /* redirect to input file */
            fd = open(ifile, O_RDWR);

            if (fd < 0) {
                printf("Error occurs when opening %s.\n", ifile);
                exit(0);
            }

            dup2(fd, STDIN_FILENO);
        }
        else if (strlen(ofile)) /* redirect to output file */
        {
```

```
            fd = open(ofile, O_RDWR|O_CREAT, S_IRWXU);

            if (fd < 0) {
                printf("Error occurs when opening %s\n", ofile);
                exit(0);
            }

            dup2(fd, STDOUT_FILENO);
        }

        if (split_pos == 0) { /* no pipe */
            /* execute the command in the child process */
            exe_err = execvp(args[0], args);

            if (exe_err < 0) printf("The command is not executable.\n");
        }
        else {
            /* this part will be discussed in next exercise */
        }

    }
    else { /* parent process */
        if (wait_flag) wait(NULL); /* the parent process waits */
    }
}
```