CSE 3033
OPERATING SYSTEMS

Programming Assignment # 2

150119858-Yasin Çörekci

150119683-Busenur Yılmaz

150119066-Ertan Karaoğlu

PART A:

We are taking the command as input and executing that in a new process. When our program gets the program name, it creates a new process using fork() system call, and the new process (child) executes the program.

We have used execv(), so we have read the PATH environment variable, then search each directory in the PATH for the command file name that appears on the command line.

```
// Running execv command
execv(path, args);

perror("No such command\n");

return -1;
```

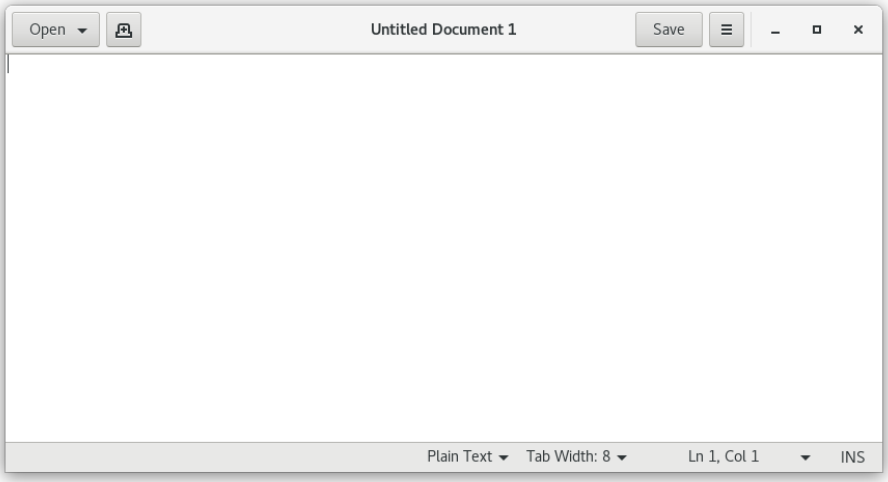
parsePath function searches command in the PATH directories:

```
// Search command in the PATH executable directories and return full path of the command if command exists
char* parsePath(const char *cmd)
{
    static char path[PATH_MAX];
    char *token;

    // Get path environmental parameter
    char *path_env = getenv("PATH");

    if (path_env != NULL) {
        token = strtok(path_env, ": \r\n");

        while (token != NULL) {
            struct stat _stat;
            snprintf(path, PATH_MAX, "%s/%s", token, cmd);

            // Check if full path exists for this command
            if (stat(path, &_stat) == 0)
                return path;

            token = strtok(NULL, ": \r\n");
        }
    }
    else {
        fprintf(stderr, "Cannot get path environmental variable\n");
    }

    // Executable not found in the paths return the command itself
    snprintf(path, PATH_MAX, "%s", cmd);

    return path;
}
```

For foreground process, parent process is waiting for the child while for the background process, the parent does not wait for the child process and wait for a new command. The background process id is stored in the background process array.

```
if (background)
{
    // Add child process to the background array and don't wait to finish
    printf("[1] %d\n", pid);
    addBackground(pid);
}
else
{
    // Foreground
    int status;
    curr_pid = pid;

    // Wait child process until finished
    do {
        if ((pid = waitpid(curr_pid, &status, WNOHANG)) == -1) {
            perror("wait() error");
        }
        else if (pid == 0) {
            // Child process is not finished
            usleep(10000);
        }
        else {
            // Child process is finished
            curr_pid = -1;
        }
    } while (pid == 0);
}
```

```
myshell: gedit &
[1] 9294
myshell:
```



PART B:

We have implemented some built-in commands in this part

history: list and execute the last 10 commands. We have written some helper functions to achieve for it.

addHistory adds command to the history array and print history prints commands in the history array.

```
// Add command to the history
void addHistory(const char *cmd)
{
    int i;

    // Move commands in the history to one next index
    if (hist_len < MAX_HIST_COUNT) {
        for (i = hist_len - 1; i >= 0; i--) {
            snprintf(history[i+1], MAX_CMD_LEN, "%s", history[i]);
        }

        hist_len++;
    }
    else {
        for (i = hist_len - 2; i >= 0; i--) {
            snprintf(history[i+1], MAX_CMD_LEN, "%s", history[i]);
        }
    }

    // New command is in the 0th position in the history
    snprintf(history[0], MAX_CMD_LEN, "%s", cmd);
}

// Print all commands in the history
void printHistory()
{
    int i;

    for (i = 0; i < hist_len; i++) {
        printf("\t%d %s\n", i, history[i]);
    }
}
```

```
myshell: ls
CSE3033_Project2.pdf  mainSetup  mainSetup.c
myshell: ps
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:00:56 java
  9258 pts/0    00:00:00 mainSetup
  9269 pts/0    00:00:00 ps
myshell: gedit
myshell: history
        0 gedit
        1 ps
        2 ls
        3 clear
myshell: history -i 1
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:00:57 java
  9258 pts/0    00:00:00 mainSetup
  9276 pts/0    00:00:00 ps
myshell: history
        0 ps
        1 gedit
        2 ps
        3 ls
        4 clear
myshell:
```

^Z (Ctrl + Z): Stops the currently running foreground process, as well as any descendants of that process. We are handling here SIGTSTP signal in the signalHandler method. We kill foreground process if exists.

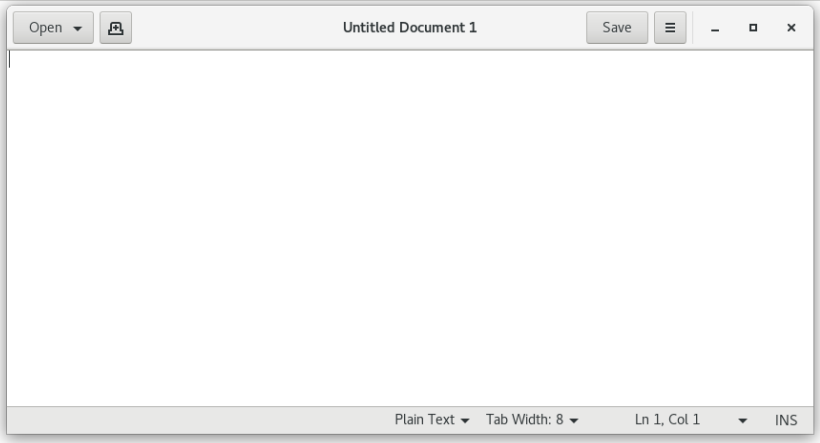
```
// Signal handler to catch SIGINT and SIGTSTP
void signalHandler(int signum)
{
    signal(SIGINT, signalHandler);
    signal(SIGTSTP, signalHandler);

    if (curr_pid == -1)
        return;


    printf("\n");
    fflush(stdout);

    // Kill current foreground child process
    kill(curr_pid, SIGKILL);
}
```

```
myshell: gedit &
[1] 9294
myshell: ^Z
myshell: ^Z
myshell: gedit
^Z
myshell:
```

fg %num: Move the background process with process id num to the foreground. We keep track all the background processes.

```
else if (strcmp(args[0], "fg") == 0) {
    // fg build-in command
    if (args[1] != NULL) {
        int num = atoi(args[1]);

        // Check if the process is in the background array
        if (checkBackgroud(num)) {
            int status;

            // Wait this child process to be finished
            do {
                if ((pid = waitpid(num, &status, WNOHANG)) == -1) {
                    perror("wait() error");
                }
                else if (pid == 0) {
                    usleep(10000);
                }
                else {
                    curr_pid = -1;
                }
            } while (pid == 0);
        }
    }
    else {
        fprintf(stderr, "Usage: fg <process id>\n");
    }

    continue;
}
```

```
myshell: gedit &
[1] 9707
myshell: fg 9707
```



exit: Terminate shell process. If the user chooses to exit while there are background processes, we notify the user that there are background processes still running and do not terminate the shell process unless the user terminates all background processes.

```
else if (strcmp(args[0], "exit") == 0) {
    // exit build-in command
    // First check if there are still background processes
    int count = getBackgroudCount();

    // If there are background processes don't exit
    if (count == 0) {
        exit(0);
    } else {
        fprintf(stderr, "Background count %d, cannot exit\n", count);
    }

    continue;
}
```

```
myshell: gedit &
[1] 9884
myshell: exit
Background count 1, cannot exit
myshell: ls -al &
[1] 9890
myshell: total 124
drwxr-xr-x. 2 mete mete    70 Dec 19 02:17 .
drwxr-xr-x. 7 mete mete    65 Dec 14 11:51 ..
-rw-------. 1 mete mete 87077 Nov 25 18:28 CSE3033_Project2.pdf
-rwxrwxr-x. 1 mete mete 18544 Dec 19 02:17 mainSetup
-rw-------. 1 mete mete 14882 Dec 19 02:15 mainSetup.c

myshell: exit
Background count 1, cannot exit
myshell: fg 9884
```

PART C:

Our shell is supporting I/O-redirection on either or both stdin, stdout, stderr and it can include arguments as well. To achieve that we have used dup2 system call. For each redirection we use open system call with the following FLAGS and used in the code as below:

```
#define FLAG_WRITE      O_CREAT | O_WRONLY | O_TRUNC
#define FLAG_APPEND     O_CREAT | O_WRONLY | O_APPEND
#define FLAG_READ       O_RDONLY
```

```
    while (args[i] != NULL && args[i+1] != NULL) {
        redirect = TRUE;

        if (strcmp(args[i], ">") == 0) {
            // redirect standard output by creating a new file
            fd = open(args[i+1], FLAG_WRITE, 0664);

            dup2(fd, STDOUT_FILENO);
            close(fd);
        }
        else if (strcmp(args[i], "2>") == 0) {
            // redirect standard error by creating a new file
            fd = open(args[i+1], FLAG_WRITE, 0664);

            dup2(fd, STDERR_FILENO);
            close(fd);
        }
        else if (strcmp(args[i], ">>") == 0) {
            // redirect standard output by appending to the file
            fd = open(args[i+1], FLAG_APPEND, 0664);

            dup2(fd, STDOUT_FILENO);
            close(fd);
        }
        else if (strcmp(args[i], "<") == 0) {
            // redirect standard input from the file
            fd = open(args[i+1], FLAG_READ, 0);

            dup2(fd, STDIN_FILENO);
            close(fd);
        }
        else {
            ++i;
            redirect = FALSE;
        }
```

```
myshell: ls > a.txt
myshell: cat a.txt
a.txt
CSE3033_Project2.pdf
mainSetup
mainSetup.c
myshell: ls >> a.txt
myshell: cat a.txt
a.txt
CSE3033_Project2.pdf
mainSetup
mainSetup.c
a.txt
CSE3033_Project2.pdf
mainSetup
mainSetup.c
myshell: cat < a.txt 2> b.txt > c.txt
myshell: cat b.txt
myshell: cat c.txt
a.txt
CSE3033_Project2.pdf
mainSetup
mainSetup.c
a.txt
CSE3033_Project2.pdf
mainSetup
mainSetup.c
myshell:
```

S

```
myshell: ps > a.txt
myshell: cat a.txt
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:01:09 java
 10100 pts/0    00:00:00 mainSetup
 10102 pts/0    00:00:00 ps
myshell: ps >> a.txt
myshell: cat a.txt
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:01:09 java
 10100 pts/0    00:00:00 mainSetup
 10102 pts/0    00:00:00 ps
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:01:09 java
 10100 pts/0    00:00:00 mainSetup
 10108 pts/0    00:00:00 ps
myshell: cat < a.txt > b.txt
myshell: cat b.txt
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:01:09 java
 10100 pts/0    00:00:00 mainSetup
 10102 pts/0    00:00:00 ps
   PID TTY          TIME CMD
  4598 pts/0    00:00:00 bash
  5246 pts/0    00:01:09 java
 10100 pts/0    00:00:00 mainSetup
 10108 pts/0    00:00:00 ps
myshell: cat < a.txt 2> c.txt > d.txt
myshell: cat c.txt
myshell:
```