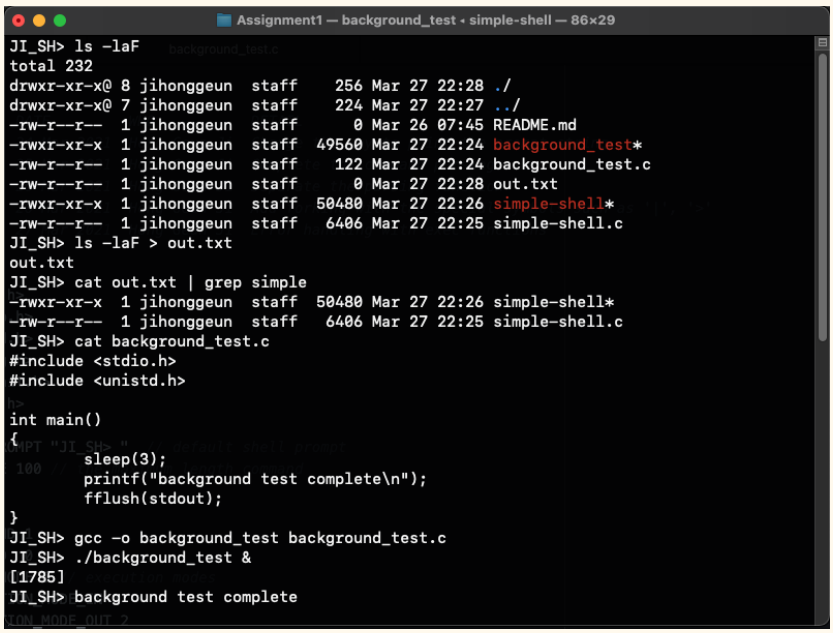
Operating Systems Spring Semester
Dr. HEEKUCK OH

Operating Systems Project 1
The JI Shell

—

By HONGGEUN JI [2019042633], Division of Computer Science, HYU ERICA

27.03.21



Introduction

This report is dealing with the first assignment given by Dr. HEEKUCK OH. The assignment was to implement and design a C program to serve as a shell interface. Most of the codes are based on the POSIX system calls so that it can be run on Linux, UNIX or even the macOS.

The base prompt is “JI_SH> ” This will allow the user to input some commands. You can invoke the process as a background by using the ‘&’ sign at the end. File redirection and pipe also can be done by using the ‘>’, ‘|’ respectively. Most of the commands, just like we use in the bash or zsh, can be used in JI Shell, and it will properly fork and wait the child processes. However, there are some downsides of this program when the user did not input the proper command form (You may check this form at the “proj1.pdf”). If the user did not use the proper command form, JI Shell cannot give the right answer as the user expected.

How does it work? - The Algorithm

1. The JI Shell pops up the shell prompt and ready to take the user command

   To take the user's input, the shell should let the user know it is ready to get some commands. Doing this, JI Shell will print the prompt "JI_SH> " The shell will not be terminated unless the user input the exit command or there is an error while executing the command.

2. The JI Shell parses the user command

   After the user input the command, the JI Shell will check for the validity of the command, and parse it to give tokens to the child process. If the command given by the user was meaningless, then the JI Shell will not give the user any result. The commands are separate by whitespace, and each of the tokens will be saved in the token string-list.

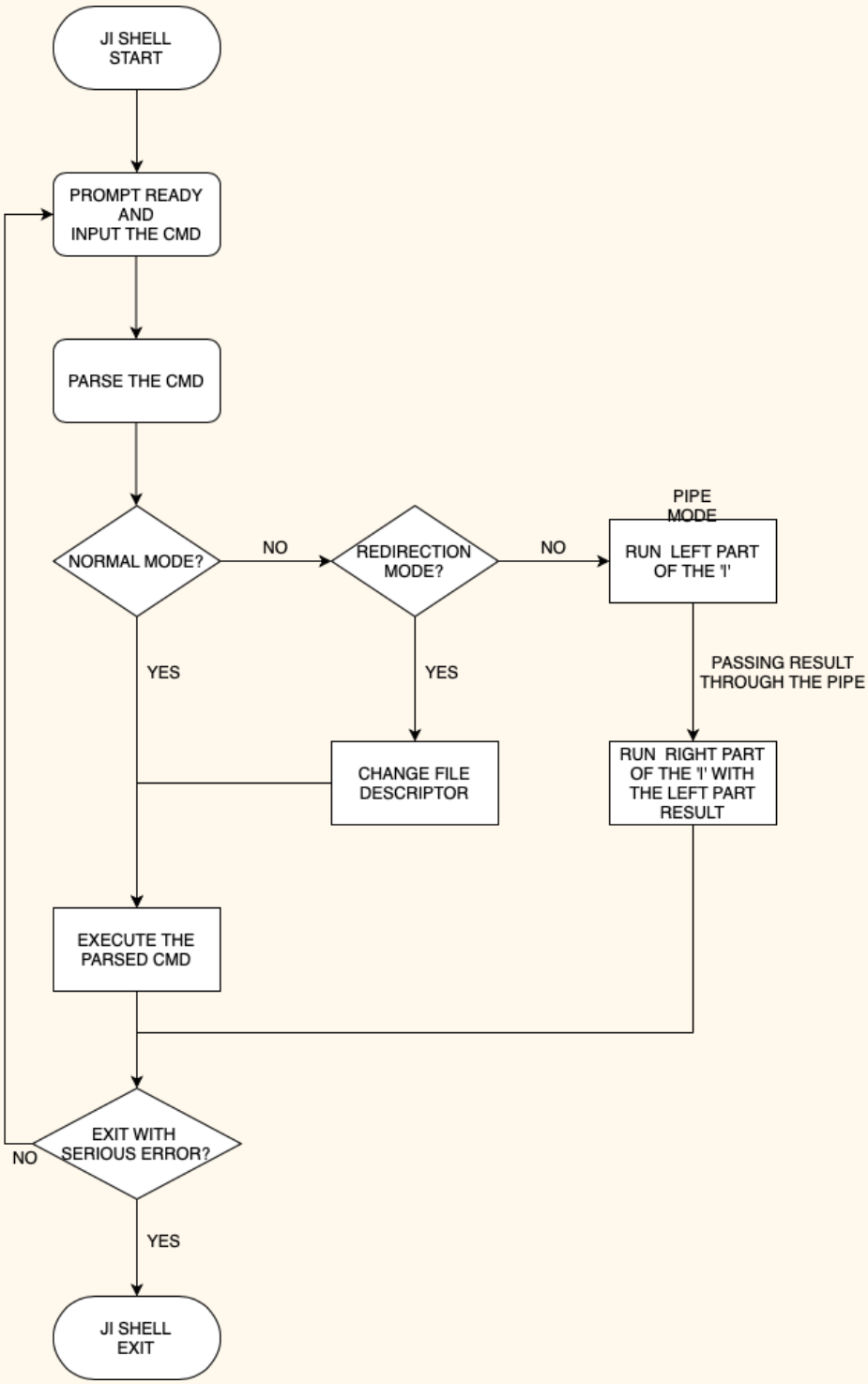
3. The JI Shell program acts differently by the command mode

   JI Shell will then use the fork() system call to make a new process, and let the child process implement the given commands. The child process will choose what types of execution mode to execute.
   There are three execution modes. REDIRECTION_MODE_OUT and REDIRECTION_MODE_IN are the modes when the user uses the '>' or '<' symbol to change the file redirection. PIPE_MODE will be used when the user types the '|' symbol to use the command with the pipe. The child process will use only one mode at a time to execute the given tokens. Also, there is a background mode that can run the command in a background mode. Users can use this with the '&' sign at the end of the command. The child process will run in the background with this '&' sign; however, the default execution will happen in the foreground.

4. The JI Shell checks the process exit status and goes back to the first step

   After the child process executes the commands, it will give some status information to the JI Shell. JI Shell will simply check the exit status, and if the status was a normal exit or exit with a non-serious error, it will go back to the first step.

Diagram for JI Shell



The Codes

```
/*
FILENAME : ji_shell.c

DESCRIPTION :
  Designing a C program accepting user commands and serving in sort of shell's way

NOTES :
  This C file is designed for assignment1 given by Dr. Hee Kuck Oh

AUTHOR :  Hong Geun Ji      START DATE : 23 Mar 2021

CHANGES :
  NO   VERSION   DATE        WHO          DETAIL
  1    1.1       24 Mar 2021  Hong Geun Ji  Change the way to print the prompt string
  2    1.2       25 Mar 2021  Hong Geun Ji  Complete the command parsing
  3    1.3       26 Mar 2021  Hong Geun Ji  Separate the parsing function
  4    1.4       26 Mar 2021  Hong Geun Ji  Add forking with the special symbols such as '|', '>'
  5    1.5       27 Mar 2021  Hong Geun Ji  Error handling with exec functions
  6    1.6       28 Mar 2021  Hong Geun Ji  Remove the unused vars and revise the repetitive codes
  7    1.7       28 Mar 2021  Hong Geun Ji  Revise and add some comments
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <fcntl.h>
#define SHELL_PROMPT "JI_SH> "  // default shell prompt
#define MAX_LINE 100 // the maximum length command
#define TRUE 1
#define FALSE 0
#define WRITE_END    1
#define READ_END    0
#define NORMAL_MODE 0 // execution modes
#define REDIRECTION_MODE_IN 1
#define REDIRECTION_MODE_OUT 2
#define PIPE_MODE 3

// the length of the prompt string
// need to change this as well if the prompt string is changed
#define PROMPT_BUFF_SIZE 7
```

```
// parses the command and saves some info. (this returns the args location of '>', '<', and '|')
int MyParse(char* usr_cmd, char** args, int* background, int* mode) ;

// fork by mode
int Execution(char** args, int mode, int background, int symbol_loc);

int main(void)
{
    char** args = (char**)malloc(MAX_LINE*sizeof(char*)/2+sizeof(char*));  // cmd line arguments
    char* usr_cmd = (char*)malloc(MAX_LINE*sizeof(char)+sizeof(char));    // user command line ready
    char*  prompt = SHELL_PROMPT; // string for the prompt
    int shell_is_alive;  // shell status
    int mode;        // execution mode when fork
    ssize_t cmd_len;    // the number of chars from user command input
    int special_symbol_loc; // the symbol location such as '|' or '>'
    int background;

    // prompt runs until it is killed by something or someone
    shell_is_alive = TRUE;
    while(shell_is_alive)
    {
        /**************** user command input area start *****************/
        // use system call to print right away
        if(!write(STDOUT_FILENO, prompt, PROMPT_BUFF_SIZE))
        {
            // use stderr to let fprintf prints error right away
            fprintf(stderr, "prompt error\n\n");
            exit(EXIT_FAILURE);
        }
        memset(usr_cmd, '\0', MAX_LINE);  // reset the command line buffer

        if((cmd_len = read(STDIN_FILENO, usr_cmd, MAX_LINE+1)) > MAX_LINE)
        {
            // use stderr file to let fprintf prints error right away
            fprintf(stderr, "command error! may be too long?\n\n");
            exit(EXIT_FAILURE);
        }
        if(*usr_cmd == '\n') continue;  // do nothing with the single newline
        *(usr_cmd+cmd_len-1) = '\0';    // swap the '\n' to null character
        if(strcmp(usr_cmd, "exit\0") == 0) break; // exit condition
        /**************** user command input area end *****************/

        special_symbol_loc = MyParse(usr_cmd, args, &background, &mode);
        if(Execution(args, mode, background, special_symbol_loc)  == 0)
            continue; // fork followed by mode
        else // execution failed
        {
```

```
            printf("something went wrong...\n");
            shell_is_alive = FALSE; // kill this shell process on execution error
        }
    } // while

    return 0;
}

int Execution(char** args, int mode, int background, int symbol_loc)
{
    pid_t pid;  // process id for non PIPE_MODE
    pid_t pid2; // process id for PIPE_MODE
    int status;

    pid = fork();
    if(pid < 0) // error
    {
        fprintf(stderr, "Fork failed");
        return 1;
    }
    else if(pid == 0) // child
    {
        int fd;
        int fdl[2];

        // use redirection according to the mode
        if(mode == REDIRECTION_MODE_OUT)
        {
            printf("%s\n",*(args+symbol_loc+1));  // let the user knows the output file name
            fflush(stdout);
            fd = open(*(args+symbol_loc+1),
                    O_CREAT|O_RDWR|O_TRUNC,
                    0666);
            dup2(fd, STDOUT_FILENO);
        } // end if REDIRECTION_MODE_OUT
        if(mode == REDIRECTION_MODE_IN)
        {
            fd = open(*(args+symbol_loc+1),
                    O_CREAT|O_RDWR,
                    0666);
            dup2(fd, STDIN_FILENO);
        } // end if REDIRECTION_MODE_IN

        // child use pipe on PIPE_MODE
        if(mode == PIPE_MODE)
        {
            pipe(fdl);
```

```
            pid2 = fork();  // create a child which will run the right part of the pipe symbol

            if(pid2 < 0) // error
            {
                fprintf(stderr, "Fork2 failed");
                return 1;
            }
            else if(pid2 == 0) // child again
            {
                dup2(fdl[READ_END], STDIN_FILENO);  // take the result from the parent
                close(fdl[WRITE_END]);
                execvp(*(args+symbol_loc+1), args+symbol_loc+1);  // execute the right part of the '|'
                exit(1);  // exit on pipe error
            }
            else  // child as a parent
            {
                dup2(fdl[WRITE_END], STDOUT_FILENO);  // give the result to the child
                close(fdl[READ_END]);
                sleep(1); // give some time to ready for next command
            }
        } // end if PIPEMODE

        execvp(*args, args);
        exit(1);  // exit on fail (such as meaningless command)
    } // else if
    else  // parent
    {
        if(background) {
            fprintf(stdout, "[%d]\n", pid);
            fflush(stdout);
        }
        else{     // wait only if it is a foreground
            waitpid(pid, &status, 0);
        }
    } // else
    return 0;
}

int MyParse(char* usr_cmd, char** args, int* background, int* mode)
{

    int i = 0;
    int symbol_loc = 0;
    int args_len;

    // save the each token to args
    *args = strtok(usr_cmd, " ");
```

```
    while(*(args+i))
    {
        i++;
        *(args+i) = strtok(NULL, " ");  // parse the next location from right before
    }
    args_len = i; // save the number of args elements
    i = 0;
    *mode = NORMAL_MODE;
    *background = FALSE;

    // check whether the command should run on background or not
    if(strcmp(*(args+args_len-1), "&") == 0) {
        *background = TRUE;
        *(args+args_len-1) = NULL;
    }

    while(*(args+i))  // look up the every args until it finds what mode should be used
    {
        // printf("!%s!\n", *(args+i));  // testing what the tokens are
        if(strcmp(*(args+i), ">") == 0)  // found “>”
        {
            *mode = REDIRECTION_MODE_OUT;
            *(args+i) = NULL;
            symbol_loc = i;
            break;
        }
        if(strcmp(*(args+i), "<") == 0) // found “<”
        {
            *mode = REDIRECTION_MODE_IN;
            *(args+i) = NULL;
            symbol_loc = i;
            break;
        }
        if(strcmp(*(args+i), "|") == 0) // found “|”
        {
            *mode = PIPE_MODE;
            *(args+i) = NULL;
            symbol_loc = i;
            break;
        }
        i++;
    } // while
    return symbol_loc;
}
```

How to compile and run?

Simple. Use the GCC to compile it and run it on the default shell.

```
jihonggeun:~/Library/Mobile Documents/com~apple~CloudDocs/Study/HYU/2021/Operating Systems/a1
[ ls -laF
total 16
drwxr-xr-x@  3 jihonggeun  staff    96 Mar 28 00:39 ./
drwxr-xr-x@ 12 jihonggeun  staff   384 Mar 28 00:39 ../
-rw-r--r--   1 jihonggeun  staff  6406 Mar 27 23:27 ji_shell.c
jihonggeun:~/Library/Mobile Documents/com~apple~CloudDocs/Study/HYU/2021/Operating Systems/a1
[ gcc -o ji_shell ji_shell.c
jihonggeun:~/Library/Mobile Documents/com~apple~CloudDocs/Study/HYU/2021/Operating Systems/a1
[ ls -laF
total 120
drwxr-xr-x@  4 jihonggeun  staff    128 Mar 28 00:40 ./
drwxr-xr-x@ 12 jihonggeun  staff    384 Mar 28 00:40 ../
-rwxr-xr-x   1 jihonggeun  staff  50480 Mar 28 00:40 ji_shell*
-rw-r--r--   1 jihonggeun  staff   6406 Mar 27 23:27 ji_shell.c
jihonggeun:~/Library/Mobile Documents/com~apple~CloudDocs/Study/HYU/2021/Operating Systems/a1
[ ./ji_shell
JI_SH>
```

Show us the RESULT

1. Executing the normal commands

Let's run it with the simple one. JI Shell can run basic programs such as ls, ps or even the vi editor. The processor id will be printed when you type '&' to run a program in the background. You can also type the single newline if you don't want to execute anything.

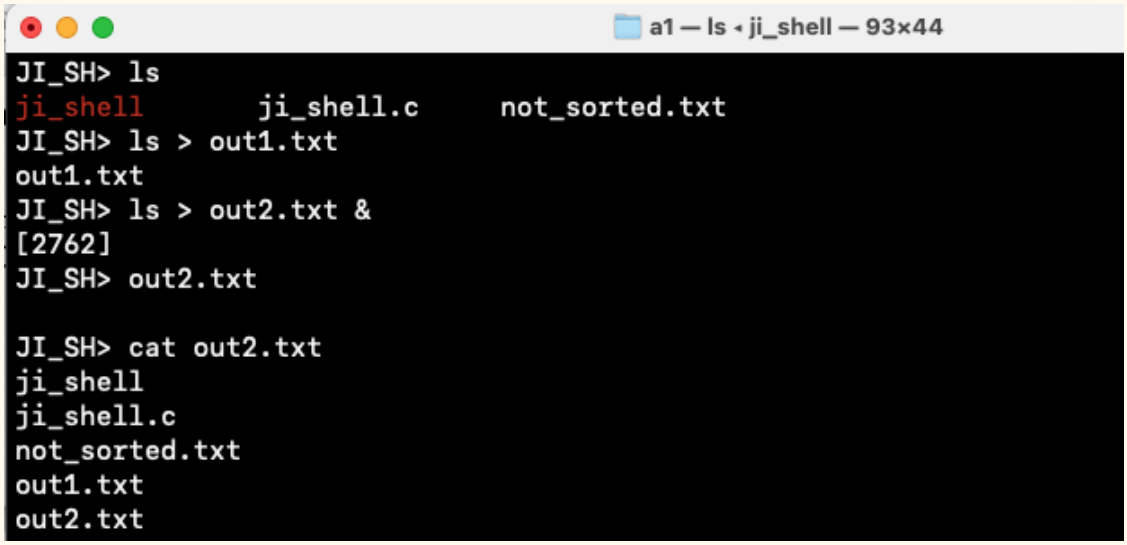
Here are some examples.

```
a1 — ps ◂ ji_shell — 93×33
JI_SH> clear

JI_SH> ls -laF
total 120
drwxr-xr-x@  4 jihonggeun  staff    128 Mar 28 00:40 ./
drwxr-xr-x@ 12 jihonggeun  staff    384 Mar 28 00:40 ../
-rwxr-xr-x   1 jihonggeun  staff  50480 Mar 28 00:40 ji_shell*
-rw-r--r--   1 jihonggeun  staff   6406 Mar 27 23:27 ji_shell.c
JI_SH> ls -laF &
[2657]
JI_SH> total 120
drwxr-xr-x@  4 jihonggeun  staff    128 Mar 28 00:40 ./
drwxr-xr-x@ 12 jihonggeun  staff    384 Mar 28 00:40 ../
-rwxr-xr-x   1 jihonggeun  staff  50480 Mar 28 00:40 ji_shell*
-rw-r--r--   1 jihonggeun  staff   6406 Mar 27 23:27 ji_shell.c

JI_SH> ps &
[2658]
JI_SH>   PID TTY           TIME CMD
  969 ttys000    0:00.95 -zsh
 2618 ttys000    0:00.01 ./ji_shell
 2640 ttys000    0:00.00 (ls)
 2657 ttys000    0:00.00 (ls)

JI_SH> ps
  PID TTY           TIME CMD
  969 ttys000    0:00.95 -zsh
 2618 ttys000    0:00.01 ./ji_shell
 2640 ttys000    0:00.00 (ls)
 2657 ttys000    0:00.00 (ls)
JI_SH>
JI_SH>
JI_SH> vi not_sorted.txt
```

The vi editor can be used via the JI Shell.

2. Redirecting input and output

Use '>' or '<' to change the file descriptor table. If you want to redirect your stdout to a file, you may use it like this.



Also, this is an example of using the stdin redirection.

```
a1 — sort ◂ ji_shell — 109×47
JI_SH> cat not_sorted.txt
banana
apple
desk
orange
cola
kakao
books
phone
watch
water
watch
games
codes

JI_SH> sort < not_sorted.txt

apple
banana
books
codes
cola
desk
games
kakao
orange
phone
watch
watch
water
JI_SH> sort < not_sorted.txt &
[2785]
JI_SH>
apple
banana
books
codes
cola
desk
games
kakao
orange
phone
watch
watch
water
```

3. Communication via a Pipe

The pipe can be used in this shell through the pipe symbol '|'

```
a1 — ji_shell — 128×30
(base) jihonggeun:~/Library/Mobile Documents/com~apple~CloudDocs/Study/HYU/2021/Operating Systems/a1
 ./ji_shell
JI_SH> ls -l | less

JI_SH> cat ji_shell.c | less

JI_SH>
```

```
(base) jihonggeun:~/Library/Mobile Documents/com~apple~C
 ./ji_shell
JI_SH> ls -l | less

JI_SH> ls -l | less &
[3910]
```

When you have a short result with a pipe, (whether it is a foreground or background)

```
a1 — ji_shell — 112×32
total 144
-rwxr-xr-x  1 jihonggeun  staff  50480 Mar 28 00:40 ji_shell
-rw-r--r--  1 jihonggeun  staff   6406 Mar 27 23:27 ji_shell.c
-rw-r--r--  1 jihonggeun  staff     79 Mar 28 00:55 not_sorted.txt
-rw-r--r--  1 jihonggeun  staff     44 Mar 28 01:18 out1.txt
-rw-r--r--  1 jihonggeun  staff     53 Mar 28 01:19 out2.txt
(END)
```

When you want to run in a background,

```
a1 — cat ◂ ji_shell — 128×30
JI_SH> cat ji_shell.c | grep AUTHOR &
[3884]
JI_SH> AUTHOR :  Hong Geun Ji      START DATE : 23 Mar 2021

JI_SH>
```

When you have a long result with a pipe,

```
a1 — ji_shell — 128×30
/*
FILENAME : ji_shell.c

DESCRIPTION :
  Designing a C program accepting user commands and serving in sort of shell's way

NOTES :
  This C file is designed for assignment1 given by Dr. Hee Kuck Oh

AUTHOR :  Hong Geun Ji      START DATE : 23 Mar 2021

CHANGES :
  NO   VERSION   DATE         WHO           DETAIL
  1    1.1       24 Mar 2021  Hong Geun Ji  Change the way to print the prompt string
  2    1.2       25 Mar 2021  Hong Geun Ji  Complete the command parsing
  3    1.3       26 Mar 2021  Hong Geun Ji  Separate the parsing function
  4    1.4       26 Mar 2021  Hong Geun Ji  Add forking with the special symbols such
  5    1.5       27 Mar 2021  Hong Geun Ji  Error handling with exec functions
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <fcntl.h>

#define SHELL_PROMPT "JI_SH> "  // default shell prompt
#define MAX_LINE 100 // the maximum length command
:
```

When you want to find a word in a foreground,

```
a1 — -zsh — 112×32
(base) jihonggeun:~/Library/Mobile Documents/com~apple~CloudDocs/Study/HYU/2021/C
[ ./ji_shell
JI_SH> cat ji_shell.c | grep AUTHOR
AUTHOR :  Hong Geun Ji      START DATE : 23 Mar 2021
JI_SH> ls -l | less

JI_SH> cat ji_shell.c | less
```

4. etc.

To strongly show that the JI Shell has the ability to run a program in the background, here is the example with the custom C file. Also, you can use the GCC right away.

First, create the *background_test.c*

```
JI_SH> ls -laF
total 160
drwxr-xr-x@  8 jihonggeun  staff    256 Mar 28 01:59 ./
drwxr-xr-x@ 12 jihonggeun  staff    384 Mar 28 01:59 ../
-rw-r--r--@  1 jihonggeun  staff   6148 Mar 28 00:55 .DS_Store
-rwxr-xr-x   1 jihonggeun  staff  50480 Mar 28 00:40 ji_shell*
-rw-r--r--   1 jihonggeun  staff   6402 Mar 28 01:50 ji_shell.c
-rw-r--r--   1 jihonggeun  staff     79 Mar 28 00:55 not_sorted.txt
-rw-r--r--   1 jihonggeun  staff     44 Mar 28 01:18 out1.txt
-rw-r--r--   1 jihonggeun  staff     53 Mar 28 01:19 out2.txt
JI_SH> vi background_test.c
JI_SH>
```

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        sleep(3);
        printf("background test complete!\n");
        return 0;
}
~
```

Then let the JI Shell compile it.

```
JI_SH> gcc -o background_test background_test.c
JI_SH> ls -laF
total 272
drwxr-xr-x@ 10 jihonggeun  staff    320 Mar 28 02:04 ./
drwxr-xr-x@ 12 jihonggeun  staff    384 Mar 28 02:04 ../
-rw-r--r--@  1 jihonggeun  staff   6148 Mar 28 00:55 .DS_Store
-rwxr-xr-x   1 jihonggeun  staff  49472 Mar 28 02:04 background_test*
-rw-r--r--   1 jihonggeun  staff    117 Mar 28 02:03 background_test.c
-rwxr-xr-x   1 jihonggeun  staff  50480 Mar 28 00:40 ji_shell*
-rw-r--r--   1 jihonggeun  staff   6402 Mar 28 01:50 ji_shell.c
-rw-r--r--   1 jihonggeun  staff     79 Mar 28 00:55 not_sorted.txt
-rw-r--r--   1 jihonggeun  staff     44 Mar 28 01:18 out1.txt
-rw-r--r--   1 jihonggeun  staff     53 Mar 28 01:19 out2.txt
JI_SH>
```

Test it in the background.