

리눅스 프로그래밍 과제 보고서

나만의 Shell 만들기

제출일 : 2021.12.01

1 요구사항 정의

1.1 PROJECT (1)

1. cd(change directory) 기능 구현
2. exit 기능 구현
3. 백그라운드 실행 구현(명령 뒤에 & 표시)

1.2 PROJECT (2)

1. SIGCHLD 로 자식 프로세스 wait() 시 프로세스가 온전하게 수행되도록 구현
2. ^C(SIGINT), ^W(SIGQUIT) 사용시 셸이 종료되지 않도록, Foreground 프로세스 실행 시 SIGINT 를 받으면 프로세스가 끝나는 것을 구현

1.3 PROJECT (3)

1. Redirection(>, <) 구현
2. Pipe(|) 구현

2 구현 방법 서술

2.1 PROJECT (1)

2.1.1 cd, exit 기능 구현

Directory 를 바꾸기 위해 chdir system call 을 호출하고 싶지만 문제가 있다. 이를 호출하기 위해 fork 를 하고 exec 를 하는 순간, 원래 의도했던 프로세스가 아닌 그 프로세스의 자식이 Directory 를 바꾸는 사태가 발생한다. 따라서 cd 명령어가 의미가 없다.

Exit 명령어 역시 마찬가지인데, fork 를 하고 exec 를 해서 Exit 를 한다면 이는 자식 프로세스가 Exit 하게 되므로 의미가 없다.

이를 방지하기 위해, fork 를 하기 전 해당 프로세스 내에서 system call 을 호출할 필요가 있다. 따라서, fork 를 하기 전에 명령어를 먼저 탐색하여 해당 명령어가 fork 전에 처리할 필요가 있는지 판단하는 코드를 추가하였다.

```
if( (temp_index = my_shell_internal_instruction(cmdline)) >= 0){  
    internal_instruction(temp_index, cmdvector);  
}
```

fork 하기 전 판단하는 if 문

```

int my_shell_internal_instruction(char* instruction){
    int index = 0;
    for(; my_internal_list[index] != NULL && index < MAX_INT_INST ; index++){
        //printf("%d\n" , strcmp(my_internal_list[index] , instruction));
        if(strcmp(my_internal_list[index] , instruction) == 0)
            return index;
    }

    return -1;
}

int internal_instruction(int index, char** arg){
    int result = 0;
    switch(index){
        case 0:
            change_directory(arg[1]);
            break;

        case 1:
            my_exit();
            break;

        default:
            break;
    }
    return result;
}

```

My_shell_internal_instruction 함수에서 명령어가 *cd*, *exit* 같이 *fork* 전에 처리할 필요가 있는 명령어인지 확인한다. 만약 그런 명령어가 맞다면, *internal_instruction* 함수에서 해당 명령어에 맞는 함수를 호출하도록 한다.

```

char* my_internal_list[MAX_INT_INST] = {
    "cd" , "exit"
};

```

```

int change_directory(char* arg){
    if(chdir(arg) != 0)
        printf("Fail to change directory:(\n");
    else
        printf("The directory change was successful:)\n");

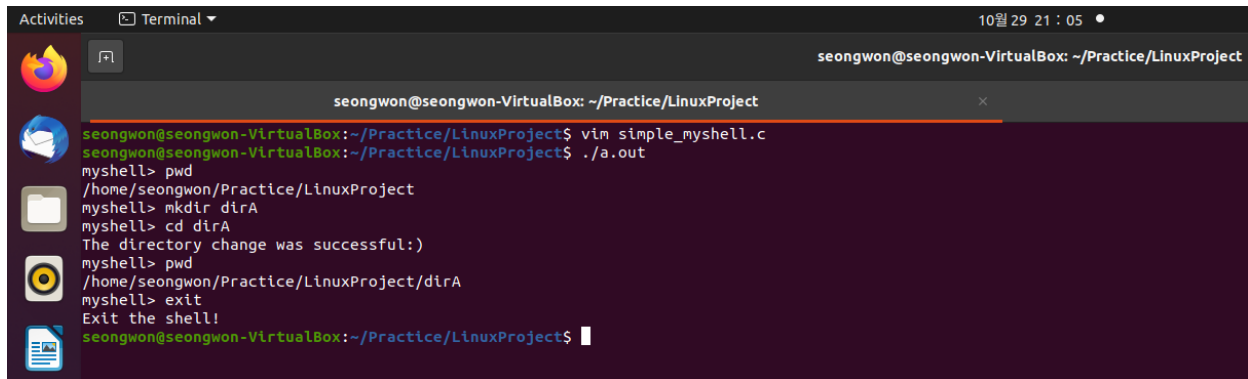
    return 0;
}

int my_exit(){
    printf("Exit the shell!\n");
    exit(0);
}

```

cd 와 *exit* 기능을 하는 함수

이렇게 구현하고 실제 해당 기능을 수행한 결과, 의도한 대로 잘 작동하는 것을 확인하였다.



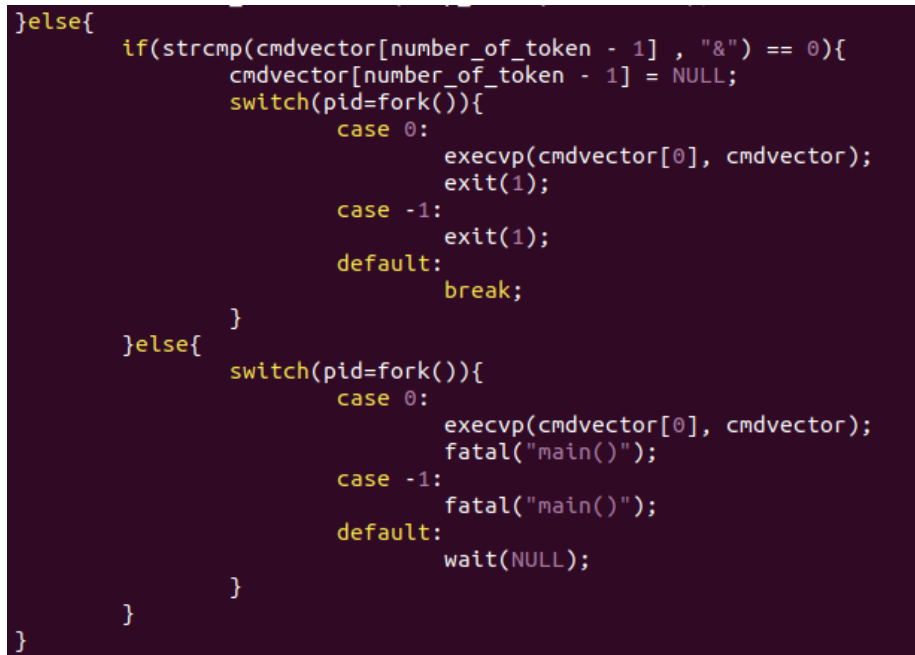
```
seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ vim simple_myshell.c
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> pwd
/home/seongwon/Practice/LinuxProject
myshell> mkdir dirA
myshell> cd dirA
The directory change was successful:)
myshell> pwd
/home/seongwon/Practice/LinuxProject/dirA
myshell> exit
Exit the shell!
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$
```

cd 와 exit 가 의도한 대로 잘 수행되는 결과 사진(테스트 사항)

2.1.2 백그라운드 실행 구현

프로세스를 백그라운드에서 실행시키기 위해 좀비 프로세스의 생성 원리를 이용하였다.

주어진 명령어에 대하여 fork 와 exec 를 해서 프로세스를 생성한다. 그리고 wait() 함수를 일부러 호출하지 않아 자식 프로세스를 기다리지 않는다. 이 경우, 자식 프로세스가 기능을 수행하는 동안 부모 프로세스는 기다리지 않고 다른 명령어를 수행할 수 있다.



```
}else{
    if(strcmp(cmdvector[number_of_token - 1], "&") == 0){
        cmdvector[number_of_token - 1] = NULL;
        switch(pid=fork()){
            case 0:
                execvp(cmdvector[0], cmdvector);
                exit(1);
            case -1:
                exit(1);
            default:
                break;
        }
    }else{
        switch(pid=fork()){
            case 0:
                execvp(cmdvector[0], cmdvector);
                fatal("main()");
            case -1:
                fatal("main()");
            default:
                wait(NULL);
        }
    }
}
```

만약 명령어 맨 뒤에 &가 붙었을 경우, 일부러 wait()를 호출하지 않는다.

2.1.2.1 테스트 사항 1

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> sleep 10000 &
myshell> sleep 10
```

의도한 대로 백그라운드에서 명령어를 수행이 가능하다.

2.1.2.2 테스트 사항 2

```
1549 ?          00:00:00 gvfsd-metadata
1552 ?          00:00:00 update-notifier
3152 pts/1      00:00:00 bash
3378 pts/0      00:00:00 a.out
3383 pts/0      00:00:00 ps <defunct>
3384 pts/0      00:00:00 ps <defunct>
3385 pts/0      00:00:00 ps
```

ps -u 1000 &를 여러 번 수행한 후 ps -u 1000 을 수행하였을 때 좀비 프로세스가 발생한 것을 알 수 있다. 왜냐하면 fork 후 exec 를 해서 자식 프로세스가 명령어를 성공적으로 수행했으나, 부모 프로세스가 wait()를 호출해 주지 않았기 때문이다.

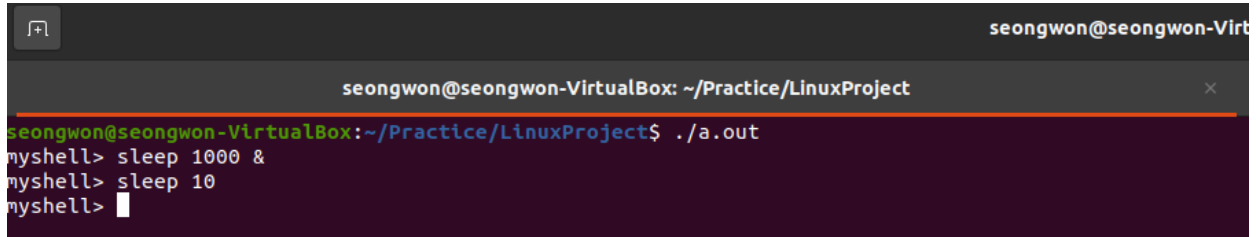
이 경우, 백그라운드에서 프로세스를 수행하는 데에는 성공했으나 그 과정에서 좀비 프로세스가 생성되므로 좋은 구현이라고는 생각하기 힘들다.

2.1.2.3 테스트 사항 3

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> sleep 10 &
myshell> sleep 20 &
myshell> ps
  PID TTY          TIME CMD
 1540 pts/0      00:00:00 bash
 3428 pts/0      00:00:00 a.out
 3431 pts/0      00:00:00 sleep
 3432 pts/0      00:00:00 sleep
 3433 pts/0      00:00:00 ps
myshell>
```

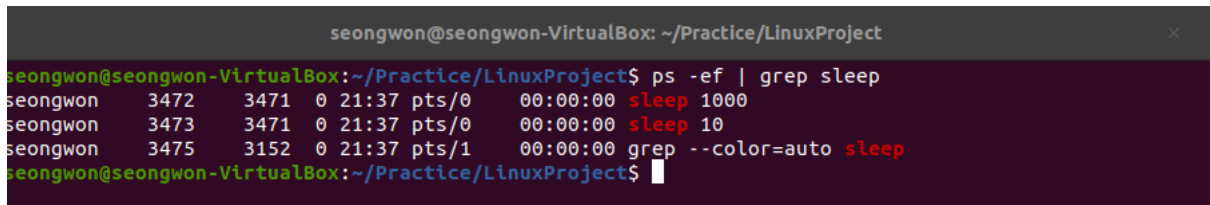
백그라운드에서 Sleep 을 하는 동안 포어그라운드에서 바로 쉘이 출력된다.

2.1.2.4 테스트 사항 4



```
seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> sleep 1000 &
myshell> sleep 10
myshell>
```

직접 만든 셸에서 sleep 1000 &과 sleep 10 호출



```
seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ps -ef | grep sleep
seongwon  3472    3471  0 21:37 pts/0    00:00:00 sleep 1000
seongwon  3473    3471  0 21:37 pts/0    00:00:00 sleep 10
seongwon  3475    3152  0 21:37 pts/1    00:00:00 grep --color=auto sleep
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$
```

두 sleep 이 수행되고 있는 동안 다른 창에서 process 확인

두 개의 창을 띄운 후, 직접 만든 셸에서 sleep 을 백그라운드에서 하나, 포어그라운드에서 하나 수행한다. 그 후, 다른 창에서 해당 sleep 이 수행되고 있는지 확인한다. Sleep 은 정상적으로 수행히 되었다.

2.2 PROJECT (2)

2.2.1 SIGCHLD 이용하여 백그라운드 실행 구현

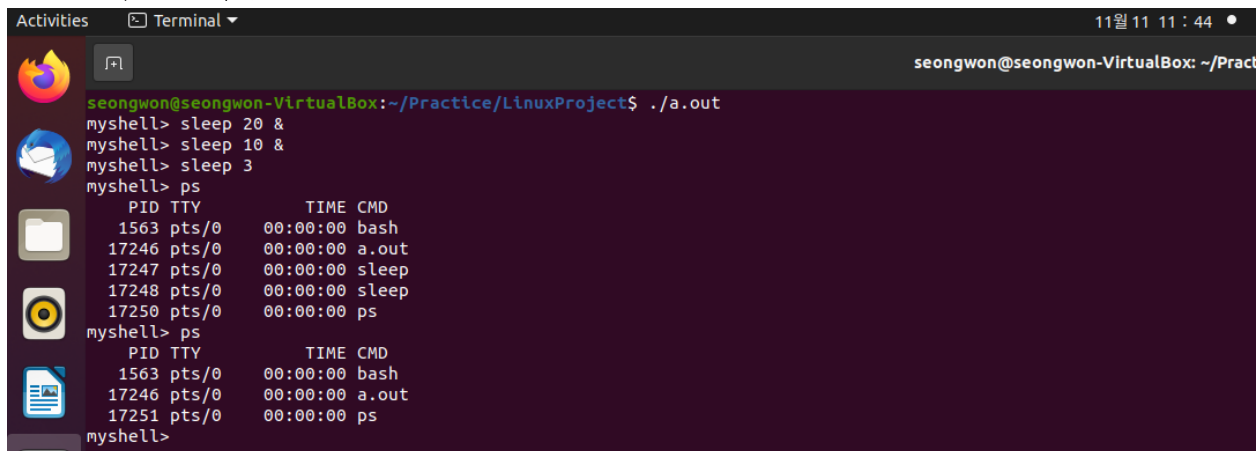
wait()를 호출하지 않는 방식으로 백그라운드 실행을 구현하면 문제가 있다. 시그널을 이용하면 이 문제를 해결할 수 있다. fork 후 exec 까지 수행한 후, 마찬가지로 wait()를 호출하지 않은 채 다음으로 넘어간다. 그 후, 자식이 작업을 마치고 나서 SIGCHLD 를 전달하면 부모 프로세스가 이를 받아 catch_child 함수를 수행한다. 이 함수는 백그라운드에서 작업을 마친 자식을 종료할 수 있게 해 준다. 아직 백그라운드에서 작업을 마치지 않은 자식 프로세스에 대해서는, 옵션을 WNOHANG 으로 주어 기다리지 않고 넘어갈 수 있게 한다.

```
if(strcmp(cmdvector[number_of_token - 1] , "&") == 0){
    cmdvector[number_of_token - 1] = NULL;
    switch(pid=fork()){
        case 0:
            sigaction(SIGINT , &temp_act , NULL);
            sigaction(SIGQUIT , &temp_act , NULL);
            sigaction(SIGTSTP , &temp_act , NULL);
            execvp(cmdvector[0], cmdvector);
            exit(1);
        case -1:
            exit(1);
        default:
            break;
    }
}
```

```
void catch_child(int signo){
    while(waitpid(-1 , NULL , WNOHANG) > 0) {}
    strcpy(cmdline, "\0");
}
```

부모 프로세스가 SIGCHLD 를 받았을 경우 해당 시점에서 작업을 마친 자식을 모두 종료할 수 있게 한다.

2.2.1.1 테스트 사항



The screenshot shows a terminal window with the following content:

```
seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject$ ./a.out
myshell> sleep 20 &
myshell> sleep 10 &
myshell> sleep 3
myshell> ps
  PID TTY          TIME CMD
 1563 pts/0        00:00:00 bash
 17246 pts/0        00:00:00 a.out
 17247 pts/0        00:00:00 sleep
 17248 pts/0        00:00:00 sleep
 17250 pts/0        00:00:00 ps
myshell> ps
  PID TTY          TIME CMD
 1563 pts/0        00:00:00 bash
 17246 pts/0        00:00:00 a.out
 17251 pts/0        00:00:00 ps
myshell>
```

좀비 프로세스가 더 이상 발생하지 않는다.

Sleep 프로세스 두 개를 백그라운드에서 실행하도록 한다. 해당 프로세스들이 작업을 마친 후 확인했을 때 좀비 프로세스가 발생하지 않는 걸 확인할 수 있다.

2.2.2 Foreground 프로세스 실행 시 SIGINT 를 받으면 프로세스가 끝나는 것을 구현
셸은 SIGINT, SIGQUIT 에 종료되면 안 되므로 핸들러를 이용하여 이를 무시할 수 있게 한다.

```
void catch_signal(int signo){
    strcpy(cmdline , usr_signal);
    printf("\n");
}
```

셸이 SIGINT 나 SIGQUIT 을 받았을 경우 이를 수행한다. 사실상 시그널을 무시하는 것과 다름없다.

```
act.sa_handler = catch_signal;
ch_act.sa_handler = catch_child;
temp_act.sa_handler = SIG_IGN;

sigaction(SIGTSTP , &act , NULL);
sigaction(SIGINT , &act , NULL);
sigaction(SIGQUIT , &act , NULL);
sigaction(SIGCHLD , &ch_act , NULL);

while (1) {
    if(flag == false){
        flag = false;
        fputs(prompt, stdout);
    }

    fgets(cmdline, BUFSIZ, stdin);

    if(strcmp(cmdline , usr_signal) == 0){
        strcpy(cmdline , "");
        continue;
    }else if(cmdline[0] == '\0'){
        flag = true;
        continue;
    }
}
```

Sigaction 을 통해 셸이 특정 시그널을 받았을 때 다른 기능을 수행할 수 있도록 한다.

명령어를 쓰던 도중 시그널을 받았을 때 줄을 바꾸고 다시 명령어를 쓸 수 while 문 안에 추가적인 if 문을 작성하였다.

또한, SIGINT 나 SIGQUIT 에 백그라운드에서 수행되던 프로세스가 반응하지 않도록, 백그라운드 프로세스는 이를 무시하게 하였다.


```
if(strcmp(cmdvector[number_of_token - 1] , "&") == 0){
    cmdvector[number_of_token - 1] = NULL;
    switch(pid=fork()){
        case 0:
            sigaction(SIGINT , &temp_act , NULL);
            sigaction(SIGQUIT , &temp_act , NULL);
            sigaction(SIGTSTP , &temp_act , NULL);
            execvp(cmdvector[0], cmdvector);
            exit(1);
        case -1:
            exit(1);
        default:
            break;
    }
}
```

백그라운드에서 실행되던 프로세스가 종료되지 않도록 한다. temp_act 의 handler 는 SIG_INT 다.

2.2.2.1 테스트 사항 1

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> ^C
myshell> ^\
myshell> exit
Exit the shell!
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$
```

셸은 SIGINT 와 SIGQUIT 을 무시한다. 셸에서 빠져나오려면 exit 명령어를 수행해야 한다.

2.2.2.2 테스트 사항 2

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> sleep 1000
^C
myshell> ps
  PID TTY          TIME CMD
  1563 pts/0        00:00:00 bash
  17337 pts/0        00:00:00 a.out
  17339 pts/0        00:00:00 ps
myshell> sleep 10000
^C
myshell> █
```

포어그라운드에서 실행되던 프로세스는 SIGINT 와 SIGQUIT 에 정상적으로 반응하여 종료된다.

2.3 PROJECT (3)

2.3.1 Redirection 구현

Redirection 의 경우 한 명령에 각각 하나씩, 최대 두 개까지 올 수 있다. 따라서 명령을 확인하다 '<' 또는 '>'가 나타날 시, 먼저 해당 연산자와 그 뒤에 있는 파일 이름을 명령에서 제거한다(redirection 후 올바른 명령을 수행하기 위해).

```
seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject
void my_redirection(){
    int fd = -1;
    for(int j = 0 ; cmdvector[j] != NULL ; j++){
        if( strcmp(cmdvector[j] , ">" ) == 0 ){
            if(cmdvector[j + 1] == NULL){
                fatal("main()");
            }else{
                if( (fd = open(cmdvector[j + 1] , O_RDWR | O_CREAT | O_TRUNC, 0644)) == -1 ){
                    fatal("main()");
                }

                dup2(fd , 1);
                close(fd);

                //delete '>' and the file name from the vector
                for(int k = j ; cmdvector[k] != NULL ; k++){
                    if(cmdvector[k + 2] == NULL) break;
                    cmdvector[k] = cmdvector[k + 2];
                }
                j--;
            }
        }else if( strcmp(cmdvector[j] , "<" ) == 0 ){
            if(cmdvector[j + 1] == NULL){
                fatal("main()");
            }else{
                if( (fd = open(cmdvector[j + 1] , O_RDONLY | O_CREAT, 0644)) == -1 ){
                    fatal("main()");
                }

                dup2(fd , 0);
                close(fd);

                //delete '<' and the file name from the vector
                for(int k = j ; cmdvector[k] != NULL ; k++){
                    cmdvector[k] = cmdvector[k + 2];
                }
                j--;
            }
        }
    }
}
```

Redirection 기능을 수행하는 함수. 명령어에 "<"나 ">"가 있는지 확인한 후, 있다면 dup2 를 통해 redirection 하고 명령어를 "<"나 ">"를 제외한 올바른 형태로 재구성한다.

```

if(strcmp(cmdvector[number_of_token - 1] , "&") == 0){
    cmdvector[number_of_token - 1] = NULL;
    switch(pid=fork()){
        case 0:
            sigaction(SIGINT , &temp_act , NULL);
            sigaction(SIGQUIT , &temp_act , NULL);
            sigaction(SIGTSTP , &temp_act , NULL);
            my_redirection();
            pipe_connect(cmdvector);
            execvp(cmdvector[0], cmdvector);
            exit(1);
        case -1:
            exit(1);
        default:
            break;
    }
}else{
    switch(pid=fork()){
        case 0:
            my_redirection();
            pipe_connect(cmdvector);
            execvp(cmdvector[0], cmdvector);
            fatal("main()");
        case -1:
            fatal("main()");
        default:
            wait(pid);
            break;
    }
}

```

fork 하기 전 my_redirection 함수를 호출하는 것으로 redirection 문제를 해결한다. 백그라운드에서도 물론 이를 수행할 수 있도록 해 준다.

2.3.1.1 테스트 사항 1

```

seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> cat > test.txt
hello!
my
name
is
Linux
myshell> cat < test.txt
hello!
my
name
is
Linux
myshell> cat < test.txt > other_test.txt
myshell> cat other_test.txt
hello!
my
name
is
Linux
myshell>

```

Cat 명령어를 활용한 redirection 테스트

테스트 결과, redirection 이 정상적으로 수행되는 것을 확인할 수 있다.

2.3.1.2 테스트 사항 2

```
seongwon@seongwon-VirtualBox: ~/Practice/LinuxProject
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
mysHELL> ls > ls.txt
mysHELL> cat < ls.txt
a.out
dirA
ls
ls.txt
other_test.txt
simple_mysHELL.c
test.txt
mysHELL> ls -l > ls.txt &
mysHELL> cat < ls.txt
total 48
-rwxrwxr-x 1 seongwon seongwon 22640 12월 1 12:03 a.out
drwxrwxr-x 2 seongwon seongwon 4096 10월 29 21:04 dirA
-rw-r--r-- 1 seongwon seongwon 47 11월 30 22:15 ls
-rw-rw-r-- 1 seongwon seongwon 0 12월 1 12:07 ls.txt
-rw-r--r-- 1 seongwon seongwon 24 12월 1 12:03 other_test.txt
-rw-rw-r-- 1 seongwon seongwon 6351 12월 1 12:03 simple_mysHELL.c
-rw-r--r-- 1 seongwon seongwon 24 12월 1 12:03 test.txt
mysHELL> █
```

백그라운드 프로세스에서 cat 명령어 수행 결과

백그라운드에서도 역시 redirection 이 정상적으로 수행되는 것을 확인할 수 있다.

2.3.2 Pipe 구현

Pipe 의 경우 하나의 명령어에 여러 개가 올 수 있기 때문에 이를 전부 확인할 필요가 있다. 따라서 우선 명령어에 "|"가 존재한다면, 모든 "|"와 그 뒤에 붙는 명령어까지 모두 제거한다(이 때, 뒤에 붙는 명령어는 배열에 따로 저장해 둔다). Pipe_connect 함수의 전반부가 이를 수행한다.

```

int pipe_connect(char** cmdvector){
    char* string[MAX_CMD_ARG][MAX_CMD_ARG];
    int j = 0, k = 0, l = 0, m = 0;

    for(k = 0 ; cmdvector[k] != NULL ; k++){
        // printf("%s\n" , cmdvector[k]);
        if( strcmp(cmdvector[k] , "|") == 0 ){
            string[j][l] = NULL;
            j++;
            l = 0;
        }
        else
            string[j][l++] = cmdvector[k];
    }

    if(j == 0) return 0;

    for(; strcmp( cmdvector[m] , string[0][0] ) != 0 ; m++){
        // printf("%s\n" , cmdvector[m]);
    }

    for(k = m; strcmp( cmdvector[k] , string[j][l - 1] ) != 0 ; k++ ){
        // printf("%s\n" , cmdvector[k]);
    }
    // printf("%s\n" , cmdvector[k]);
    for(int t = k - m ; t > 0 ; t--){
        cmdvector[m] = cmdvector[k + 1];
        m++;
        k++;
    }
    cmdvector[m] = NULL;
}

```

명령을 탐색하여 파이프를 제거하고 파이프 뒤에 붙는 명령어를 따로 저장하는 곳까지의 코드. 먼저 String 배열에 파이프 뒤에 붙는 명령어를 각각 저장해 둔 후, 원래의 명령어 배열에서 필요없는 부분을 모두 제거한다.

그 후, 파이프의 수만큼 파이프를 선언한다. 그 다음 각 명령어를 처리하면서, 파이프를 순서대로 사용하여 출력을 다음 프로세스의 입력으로 사용할 수 있게끔 구현한다.

```

int p[j][2];
pid_t pid[MAX_CMD_ARG];

for(k = 0; k < j ; k++){
    pipe(p[k]);
}

```

"|"의 개수만큼 파이프 선언

```
for(; l < j ; l++){  
  
    switch( pid[l] = fork() ) {  
    case -1:  
        perror ("fork call");  
        exit(2);  
    case 0:  
        dup2(p[l - 1][0] , 0);  
        dup2(p[l][1] , 1);  
        for(k = 0 ; k < j ; k++){  
  
            close(p[k][0]);  
            close(p[k][1]);  
        }  
        execvp(string[l][0] , string[l]);  
    default:  
        break;  
    }  
  
}
```

fork 하여 각 명령어를 수행하는 부분. 이 때, 파이프 전의 프로세스의 출력을 입력으로 받는다. 출력 또한 다음 프로세스가 입력으로 받을 파이프로 출력한다. 이를 파이프의 개수만큼 반복한다.

```
if(strcmp(cmdvector[number_of_token - 1] , "&") == 0){
    cmdvector[number_of_token - 1] = NULL;
    switch(pid=fork()){
        case 0:
            sigaction(SIGINT , &temp_act , NULL);
            sigaction(SIGQUIT , &temp_act , NULL);
            sigaction(SIGTSTP , &temp_act , NULL);
            my_redirection();
            pipe_connect(cmdvector);
            execvp(cmdvector[0], cmdvector);
            exit(1);
        case -1:
            exit(1);
        default:
            break;
    }
}else{
    switch(pid=fork()){
        case 0:
            my_redirection();
            pipe_connect(cmdvector);
            execvp(cmdvector[0], cmdvector);
            fatal("main()");
        case -1:
            fatal("main()");
        default:
            wait(pid);
            break;
    }
}
```

My_redirection 함수 호출 후 *pipe_connect* 함수를 호출하는 것으로, *redirection* 과 *pipe* 가 적절히 수행될 수 있도록 한다.

2.3.2.1 테스트 사항 1

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> ls -l > ls.txt
myshell> cat ls.txt
total 56
-rwxrwxr-x 1 seongwon seongwon 22640 12월  1 12:03 a.out
drwxrwxr-x 2 seongwon seongwon  4096 10월 29 21:04 dirA
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:23 dir_num2.txt
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:22 dir_num.txt
-rw-r--r-- 1 seongwon seongwon   47 11월 30 22:15 ls
-rw-rw-r-- 1 seongwon seongwon    0 12월  1 12:24 ls.txt
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:03 other_test.txt
-rw-rw-r-- 1 seongwon seongwon 6351 12월  1 12:03 simple_myshell.c
-rw-r--r-- 1 seongwon seongwon    0 12월  1 12:07 sleep
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:03 test.txt
myshell> cat < ls.txt | grep ^d | wc -l > dir_num.txt
myshell> cat dir_num.txt
1
myshell> cat < ls.txt | grep ^- | wc > dir_num.txt
myshell> cat dir_num.txt
    9    81   556
myshell>
```

Cat 과 ls 등을 활용하여 redirection 및 Pipe 테스트

모든 기능이 정상적으로 작동되는 것을 확인할 수 있다.

2.3.2.2 테스트 사항 2

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> ls > ls.txt
myshell> cat ls.txt
a.out
dirA
dir_num2.txt
dir_num.txt
ls
ls.txt
other_test.txt
simple_myshell.c
sleep
test.txt
myshell> ls -l | grep ^- > ls.txt &
myshell> cat ls.txt
-rwxrwxr-x 1 seongwon seongwon 22640 12월  1 12:03 a.out
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:23 dir_num2.txt
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:24 dir_num.txt
-rw-r--r-- 1 seongwon seongwon   47 11월 30 22:15 ls
-rw-rw-r-- 1 seongwon seongwon    0 12월  1 12:27 ls.txt
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:03 other_test.txt
-rw-rw-r-- 1 seongwon seongwon 6351 12월  1 12:03 simple_myshell.c
-rw-r--r-- 1 seongwon seongwon    0 12월  1 12:07 sleep
-rw-r--r-- 1 seongwon seongwon   24 12월  1 12:03 test.txt
myshell> █
```

백그라운드에서도 역시 Pipe 와 Redirection 이 정상적으로 수행되는 걸 확인할 수 있다.

3 구현 시 문제점

3.1 PROJECT (1)

3.1.1 cd, exit 의 비효율성

cd, exit 같은 명령어를 판단하기 위해서는 유저가 준 명령어를 미리 만들어 놓은 리스트와 대조해야 하는데, 만약 그 리스트에 있는 명령어가 많을 경우 대조에 많은 시간이 소모될 수 있다.

3.1.2 백그라운드 수행 시 좀비 프로세스 생성

위와 같은 코드로 백그라운드 실행을 구현할 경우 좀비 프로세스가 생성되기 때문에 프로세스 관리에 문제가 생긴다. 따라서 백그라운드에서 자식이 명령을 수행한 후 무사히 종료할 수 있도록 다른 처리를 해 주어야 할 필요가 있어 보인다(강의에서 배운 Session 을 이용하면 구현이 가능하지 않을까?).

3.1.3 백그라운드 수행 시 줄바꿈

```
seongwon@seongwon-VirtualBox:~/Practice/LinuxProject$ ./a.out
myshell> sleep 3 &
myshell> ps
  PID TTY          TIME CMD
 1540 pts/0    00:00:00 bash
 1591 pts/0    00:00:00 a.out
 1659 pts/0    00:00:00 sleep
 1660 pts/0    00:00:00 ps
myshell> ps
myshell>      PID TTY          TIME CMD
 1540 pts/0    00:00:00 bash
 1591 pts/0    00:00:00 a.out
 1661 pts/0    00:00:00 ps
sleep 1
myshell> sleep 1
myshell> pwd
myshell> /home/seongwon/Practice/LinuxProject
```

줄바꿈 문제...

백그라운드 수행이 끝난 후, 자식은 좀비 프로세스가 되지만 스크린에 출력하는 건 바뀌지 않았기 때문에, 이 경우 줄 정리에 문제가 생긴다. 백그라운드 프로세스가 스크린에 출력하지 않거나, 프로세스 종료 후에 줄을 정리하거나 하여 이 문제를 해결해야 할 것 같다.

3.2 PROJECT (2)

구현 시 문제점이 발견되지 않음.

3.3 PROJECT (3)

구현 시 문제점이 발견되지 않음.