

운영체제론 프로젝트1

소프트웨어학부

컴퓨터전공

2019040519 김태형

0. 메인 함수

```
int main(void){
    char *args[MAX_LINE / 2 + 1]; // 파싱 명령어 저장 배열
    int should_run = 1;
    char cmd[MAX_LINE]; // 명령어 입력받을 배열
    int b; // Background 명령어 판별 변수
    int r; // Redirect 타입(Input, Output) 판별 변수
    int p; // Pipe 명령어 판별 변수
    char r_cmd[20]; // Redirect 명령어 이후 문자열 저장 변수
    char **p_after; // 파이프 명령어 이후 문자 저장 포인터

    while (should_run){
        printf("osh>");
        fflush(stdout); // 출력 버퍼를 비움
        fgets(cmd, MAX_LINE, stdin); // 명령어 입력받음

        b = parseCmd(cmd, args); // 명령어 파싱 + Background 명령어 체크
        r = chkRedir(args, r_cmd); // redirect 명령어 체크
        p = chkPipe(args, &p_after); // pipe 명령어 체크

        if (!strcmp(args[0], "exit")) { // exit 명령어 입력시 종료
            should_run = 0;
            continue;
        }
        if (p == 1) { // Pipe 명령어가 있는 경우
            pipeExecute(b, args, p_after);
        }
        else { // Pipe 명령어가 없는 경우
            if (r > 0){ // Redirect 명령어가 있는 경우
                redirExecute(r, b, args, r_cmd); // Redirect 타입(Output, Input)과 Background 명령어 유무를 인자로 전달
            }
            else { // 일반 명령어의 경우
                execute(b, args);
            }
        }
    }
    return 0;
}
```

명령어를 입력받기 전, fflush(stdout)을 통해 출력 버퍼를 비워준다. 이후, fgets()를 이용해 MAX_LINE만큼 cmd 배열에 사용자 명령어 입력을 받아준다.

parseCmd(char cmd[], char *args[]) 함수에서는 명령어를 공백을 기준으로 파싱하여 args 배열에 저장하고, background 명령어 유무를 변수 b에 저장한다. background 명령어가 있는 경우, & 문자와 개행 문자를 args 배열에서 삭제한다.

chkRedir(char *args[], char r_cmd[]) 함수에서는 redirect 명령어 유무와 redirect의 타입(Input, Output)을 체크하여 redirect 명령어가 없는 경우 0, output redirect 명령어가 있는 경우 1, input redirect 명령어가 있는 경우 2를 변수 r에 저장한다. redirect 명령어가 있는 경우 >, < 문자를 args 배열에서 삭제한다.

chkPipe(char *args[], char ***p_end) 함수에서는 pipe 명령어 유무를 변수 p에 저장한다. pipe 명령어가 있는 경우, | 문자를 args 배열에서 삭제하고, | 이후 명령어를 가르키는 포인터를 지정한다.

만약 사용자가 "exit" 명령어를 입력했다면 should_run을 1로 변경시켜, shell을 종료시킨다.

이후 변수 b, r, p의 값을 기반으로 각각의 케이스에 맞게 pipe, input redirect, output redirect, 일반 명령어 작업들을 수행한다.

1. 함수 설명

• int parseCmd(char cmd[], char *args[])

입력받은 명령어를 공백을 기준으로 파싱하여 args 배열에 저장하는 함수. 포인터 두개를 이용해 명령어 파싱 작업과 공백, 개행 문자 삭제 작업을 수행한다.

문자열의 끝을 나타내기 위해 백그라운드 명령어(&)가 있다면 & 문자가 위치한 index를 NULL 처리하고, 1을 리턴한다. 있지 않다면 args 배열의 다음 index를 NULL 처리하고, 0을 리턴한다.

```
// 명령어 파싱 + background 명령어 체크 함수
int parseCmd(char cmd[], char *args[]){
    char *ptr = cmd; // 파싱된 명령어를 저장하기 위한 포인터
    char *dptr = strchr(ptr, ' '); // 공백, 개행 문자를 삭제하기 위한 포인터
    int i = 0; // args 배열의 index

    while (*ptr == ' '){ // 첫번째 공백을 가르킴
        ptr++;
    }

    while (dptr != NULL){ // 사용자 입력의 모든 공백을 기준으로 명령어 파싱
        *dptr = '\0'; // 공백 문자 삭제
        args[i++] = ptr; // 공백 앞의 명령어를 args에 저장
        ptr = dptr + 1; // 공백의 다음 문자를 가르킴
        dptr = strchr(dptr+1, ' ');
    }

    dptr = strchr(ptr, '\n');
    *dptr = '\0'; // 개행 문자 삭제
    args[i] = ptr;

    if (!strcmp(args[i], "&")){ // '&' 문자가 있는 경우, 1을 리턴
        args[i] = NULL; // NULL 처리
        return 1;
    }
    else { // '&' 문자가 없는 경우, 0을 리턴
        args[i+1] = NULL; // NULL 처리
        return 0;
    }
}
```

• int chkPipe(char *args[], char ***p_end)

파싱된 문자열에 pipe 명령어(|)가 포함되어 있는지 확인하고, | 다음 문자의 포인터를 넘겨주는 함수.

while문으로 배열을 순회한 뒤, 포인터의 NULL 여부로 pipe 명령어 유무를 판단한다.

pipe 명령어가 있다면 |를 args 배열에서 삭제하고 1을 리턴한다. 있지 않다면 0을 리턴한다.

```
// pipe 명령어 체크 함수
int chkPipe(char *args[], char ***p_after){
    int i = 0; // '|' 문자의 index 저장 변수
    while (args[i] != NULL){ // 배열의 끝까지
        if(!strcmp(args[i], "|")) // '|' 문자를 만나면 종료
            break;
        i++; // index 증가
    }

    if (args[i] != NULL){ // 포인터가 NULL이 아닌 경우 -> Pipe 문자가 있음
        args[i] = NULL; // NULL 처리
        *p_after = args + i + 1; // '|' 다음 문자를 가르킴
        return 1; // 1을 리턴
    }
    else { // 포인터가 NULL인 경우 -> Pipe 문자가 없음
        return 0; // 0을 리턴
    }
}
```

- `int chkRedir(char *args[], char r_cmd[])`

파싱된 문자열에 redirect 명령어(>, <) 문자가 포함되어 있는지 확인하고, redirect의 type(Output, Input)을 리턴하는 함수. while문으로 배열을 순회한 뒤, 포인터의 NULL 여부로 redirect 명령어 유무를 판단한다. output redirect의 경우, redirect 이후의 명령어를 r_cmd 배열에 복사하고 1을 리턴한다. input redirect의 경우, 마찬가지로 뒷부분 명령어를 복사한 뒤, 2를 리턴한다. redirect 명령어가 존재하지 않는 경우에는 0을 리턴한다.

```
// redirect 명령어 체크 함수
int chkRedir(char *args[], char r_cmd[]){
    int i = 0; // '<', '>' 문자 index 저장 변수
    while (args[i] != NULL) { // 배열의 끝까지
        if (!strcmp(args[i], ">") || !strcmp(args[i], "<")) // '>', '<'를 만나면 종료
            break;
        i++; // index 증가
    }

    if (args[i] != NULL){ // 포인터가 NULL이 아닌 경우 -> redirect 문자가 있음
        if (!strcmp(args[i], ">")){ // Output Redirect의 경우
            strcpy(r_cmd, args[i + 1]); // 뒷부분 명령어를 복사
            args[i] = NULL; // '>' 문자 삭제
            return 1; // 1을 리턴
        }
        else if (!strcmp(args[i], "<")){ // Input Redirect의 경우
            strcpy(r_cmd, args[i + 1]); // 뒷부분 명령어를 복사
            args[i] = NULL; // '<' 문자 삭제
            return 2; // 2를 리턴
        }
    }
    else // 포인터가 NULL인 경우 -> redirect 문자가 없음
        return 0; // 0을 리턴
}
```

- `void execute(int b, char *args[])`

main문에서 chkRedir(), chkPipe() 함수의 리턴 값이 모두 0으로 나왔을 때 실행되는 일반 명령어 함수.

fork()를 통해 자식을 생성한 후, 자식의 경우에는 execvp() 함수를 이용해 명령어를 실행한다. 부모의 경우, 백그라운드 실행 명령어 유무를 나타내는 b를 parameter로 받아 foreground process라면, 부모가 자식의 종료를 기다리도록 wait() 함수를 사용한다. background process라면, 자식의 종료를 기다리지 않고 PID를 출력하고 종료한다.

```
// 일반 명령어 실행 함수
void execute(int b, char *args[]){
    int status;
    pid_t pid;
    pid = fork();
    if(pid < 0) {
        perror("Fork Error\n");
    }
    // 부모 프로세스일 경우
    else if(pid > 0) {
        if(b == 0){
            pid = wait(&status); // Foreground Process
        }
        else { // Background Process
            printf("PID: %d\n", getpid()); // pid 출력
        }
    }
    // 자식 프로세스일 경우
    else {
        execvp(args[0], args); // args 실행
        // Execvp 에러시 에러메시지 출력
        perror("Execvp Error\n");
        exit(1);
    }
}
```

- pipeExecute(int b, char *args[], char **p_after)

파이프 명령어 실행 함수. 총 두 번의 fork()를 진행하여 부모 자식 간 데이터를 전달한다.

첫번째 fork()의 부모 프로세스의 경우, foreground process라면 waitpid()로 자식의 종료를 기다린다. background process인 경우에는 자식의 종료를 기다리지 않고, pid를 출력하고 곧바로 종료한다. 자식 프로세스의 경우 2번째 fork()를 하여, 자식이 fd[1]에 쓰고 부모가 fd[0]으로 읽도록 하였다.

두번째 fork()의 부모 프로세스의 경우, 사용하지 않는 fd[1]를 닫아주고, fd[0]을 표준 입력으로 파이프를 연결하여 execvp() 함수를 통해 파이프 이후 명령어인 p_after를 실행한다. 자식 프로세스의 경우, fd[0]를 닫아주고 fd[1]을 표준 출력으로 파이프를 연결하여 execvp() 함수를 통해 파이프 이전 명령어인 args를 실행한다.

```
// Pipe 실행 함수
void pipeExecute(int b, char *args[], char **p_after){
    int fd[2];
    pid_t pid1;
    pid_t pid2;
    pid1 = fork(); // 첫번째 fork
    if (pid1 < 0) {
        perror("First Fork Error\n"); // 에러 처리
        exit(1);
    }
    if (pipe(fd) < 0){
        perror("Pipe Error\n");
        exit(1);
    }
    if (pid1 > 0){ // 첫번째 fork의 부모 프로세스의 경우
        if (b == 0){ // Foreground Process
            waitpid(pid1, NULL, 0);
        }
        else { // Background Process
            printf("PID: %d\n", getpid()); // pid 출력
        }
    }
    else { // 첫번째 fork의 자식 프로세스의 경우
        pid2 = fork(); // 두번째 fork
        if (pid2 < 0){
            perror("Second Fork Error\n"); // 에러 처리
            exit(1);
        }
        else if (pid2 > 0){ // 두번째 fork의 부모 프로세스의 경우
            close(fd[1]); // fd[1] 닫기
            dup2(fd[0], STDIN_FILENO); // fd[0] 표준 입력 파이프 연결
            close(fd[0]); // fd[0] 닫기
            execvp(p_after[0], p_after);
            // Execvp 에러시 에러메시지 출력
            perror("Execvp Error\n");
            exit(1);
        }
        else { // 두번째 fork의 자식 프로세스의 경우
            close(fd[0]); // fd[0] 닫기
            dup2(fd[1], STDOUT_FILENO); // fd[1] 표준 출력 파이프 연결
            close(fd[1]); // fd[1] 닫기
            execvp(args[0], args);
            // Execvp 에러시 에러메시지 출력
            perror("Execvp Error\n");
            exit(1);
        }
    }
}
```

- void redirExecute(int r, int b, char *args[], char *r_cmd)

input Redirect와 output Redirect를 수행하는 함수.

parseCmd, chkRedir 함수의 리턴값을 저장한 변수 r, b를 parameter로 받아 fork()를 통해 자식 프로세스를 생성한다. 부모 프로세스의 경우, foreground process라면 wait()를 이용해 자식 프로세스가 종료될 때까지 대기한다. background process라면 자식 프로세스의 종료를 기다리지 않고 종료한다. 자식 프로세스의 경우, output redirect라면, dup2() 함수를 이용해 fd를 표준 출력으로 output redirect를 수행한다. input redirect라면, dup2() 함수를 이용해 fd를 표준 입력으로 input redirect를 수행한다.

```
// Redirect 실행 함수
void redirExecute(int r, int b, char *args[], char *r_cmd){
    int fd;
    int status;
    pid_t pid;

    pid = fork();
    if (pid < 0) { // 에러 처리
        perror("First Fork Error\n");
        exit(1);
    }
    // 부모 프로세스의 경우
    else if(pid > 0){
        if (b == 0){ // Foreground Process
            pid = wait(&status); // 자식 프로세스가 종료될 때까지 대기
        }
        else { // Background Process
            printf("PID: %d\n", getpid()); // pid 출력
        }
    }
    // 자식 프로세스의 경우
    else {
        if (r == 1){ // Output Redirect 명령어가 있는 경우
            fd = open(r_cmd, O_WRONLY | O_CREAT | O_TRUNC, 0644); // 쓰기전용, 파일생성, 기존파일 내용삭제 옵션
            dup2(fd, STDOUT_FILENO); // 표준 출력으로 redirect
            close(fd);
        }
        else if (r == 2){ // Input Redirect 명령어가 있는 경우
            if((fd = open(r_cmd, O_RDONLY)) == 1) { // 읽기전용 옵션
                perror(args[0]);
                exit(2);
            }
            dup2(fd, STDIN_FILENO); // 표준 입력으로 redirect
            close(fd);
        }
        execvp(args[0], args); // args 실행
        // Execvp 에러시 에러메시지 출력
        perror("Execvp Error\n");
        exit(1);
    }
}
```


2. 컴파일 과정

```
os@os-VirtualBox: ~/proj1
File Edit View Search Terminal Help
os@os-VirtualBox:~$ cd proj1
os@os-VirtualBox:~/proj1$ ls
osh.c
os@os-VirtualBox:~/proj1$ gcc osh.c -o osh.o
os@os-VirtualBox:~/proj1$
```

별다른 오류 메시지 없이, 컴파일 되는 것을 확인할 수 있다.

3. 실행 결과물 설명

• 명령어+옵션

```
osh>ls -l
total 28
-rw-rw-r-- 1 os os 79 3월 30 12:22 hello.c
-rw-rw-r-- 1 os os 6818 3월 30 21:15 osh.c
-rwxrwxr-x 1 os os 13432 3월 30 21:19 osh.o
```

▲ ls -l 명령어 실행 모습

일반 명령어인 ls + -l 옵션을 실행하는 것을 확인할 수 있다.

```
osh>ls
hello.c osh.c osh.o
osh>gcc hello.c -o hello.o
osh>ls
hello.c hello.o osh.c osh.o
```

▲ gcc hello.c -o hello.o 명령어 실행 모습

hello.c 컴파일 명령시 정상적으로 컴파일이 되어 hello.o 파일이 생성된 것을 확인할 수 있다.

• 명령어+옵션 &

```
osh>ls -l &
PID: 2346
osh>total 28
-rw-rw-r-- 1 os os 79 3월 30 12:22 hello.c
-rw-rw-r-- 1 os os 6818 3월 30 21:15 osh.c
-rwxrwxr-x 1 os os 13432 3월 30 21:19 osh.o
```

▲ ls -l & 명령어 실행 모습

백그라운드 명령어 실행시, 부모 프로세스는 자식 프로세스를 fork한 이후 자식 프로세스의 종료를 기다리지 않고 pid를 출력하고 종료되었고, 자식 프로세스는 ls -l 명령어를 수행한 것을 확인할 수 있다.

```
osh>ls
hello.c  hello.o  osh.c  osh.o
osh>gcc hello.c -o hello2.o &
PID: 2471
osh>ls
osh>hello2.o  hello.c  hello.o  osh.c  osh.o
```

▲ gcc hello.c -o hello2.o & 명령어 실행 모습

백그라운드 명령어 실행시, 부모 프로세스는 자식 프로세스를 fork한 이후 자식 프로세스의 종료를 기다리지 않고 pid를 출력하고 종료되었고, 자식 프로세스는 gcc hello.c -o hello2.o 명령어를 수행하여 컴파일 결과 hello2.o 파일이 새롭게 생성된 것을 확인할 수 있다.

• 명령어+옵션 > 파일명

```
osh>ls
hello2.o  hello.c  hello.o  osh.c  osh.o
osh>ls -al > output.txt
osh>ls
hello2.o  hello.c  hello.o  osh.c  osh.o  output.txt
osh>cat output.txt
total 64
drwxrwxr-x  3 os os  4096  3월 31 16:54 .
drwxr-xr-x 23 os os  4096  3월 31 16:10 ..
-rwxrwxr-x  1 os os  8304  3월 31 16:37 hello2.o
-rw-rw-r--  1 os os    79  3월 30 12:22 hello.c
-rwxrwxr-x  1 os os  8304  3월 31 16:35 hello.o
-rw-rw-r--  1 os os  7023  3월 31 16:29 osh.c
-rwxrwxr-x  1 os os 13432  3월 31 16:31 osh.o
-rw-r--r--  1 os os     0  3월 31 16:54 output.txt
drwxrwxr-x  2 os os  4096  3월 29 23:14 .vscode
```

▲ ls -al > output.txt 명령어 실행 모습

명령어 실행 결과 output.txt 파일이 생성되었고,

정상적으로 ls -al 명령의 결과를 output.txt로 Output Redirect 수행한 것을 확인할 수 있다.

```
osh>ls
hello2.o  hello.c  hello.o  osh.c  osh.o  output.txt
osh>pwd > pwd.txt
osh>ls
hello2.o  hello.c  hello.o  osh.c  osh.o  output.txt  pwd.txt
osh>cat pwd.txt
/home/os/proj1
```

▲ pwd > pwd.txt 명령어 실행 모습

명령어 실행 결과 pwd.txt 파일이 생성되었고,

정상적으로 pwd 명령의 결과를 pwd.txt 파일로 Output Redirect 수행한 것을 확인할 수 있다.

• 명령어+옵션 > 파일명 &

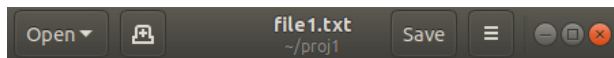
```
osh>ls
a.out hello.c hello.o osh.c osh.o output.txt
osh>ls -al > output2.txt &
PID: 2295
osh>ls
osh>a.out hello.c hello.o osh.c osh.o output2.txt output.txt

osh>cat output2.txt
total 72
drwxrwxr-x  3 os os  4096  3월 31 16:23 .
drwxr-xr-x 23 os os  4096  3월 31 16:10 ..
-rwxrwxr-x  1 os os 13432  3월 30 23:02 a.out
-rw-rw-r--  1 os os   79   3월 30 12:22 hello.c
-rwxrwxr-x  1 os os  8304  3월 30 23:02 hello.o
-rw-rw-r--  1 os os  6907  3월 31 16:14 osh.c
-rwxrwxr-x  1 os os 13432  3월 31 16:20 osh.o
-rw-r--r--  1 os os    0   3월 31 16:23 output2.txt
-rw-r--r--  1 os os   436  3월 31 16:16 output.txt
drwxrwxr-x  2 os os  4096  3월 29 23:14 .vscode
```

▲ ls -al > output2.txt & 명령어 실행 모습

명령어 실행 결과 부모 프로세스는 자식 프로세스를 생성한 뒤 pid를 출력하고 종료되었고,
자식 프로세스는 ls -al의 결과값을 output2.txt로 Output Redirect를 수행한 것을 확인할 수 있다.

• 명령어+옵션 < 파일명



example of standard input redirection

▲ file1.txt의 내용

▲ cat < file1.txt 명령어 실행 모습

```
osh>cat < file1.txt
example of standard input redirection
```

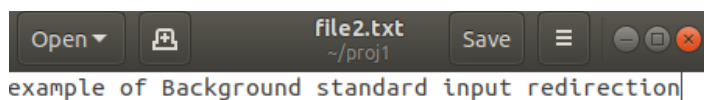
Input Redirect를 수행하여 file1.txt의 내용을 cat 명령어의 표준 입력으로 받아 출력한 것을 확인할 수 있다.

```
osh>cat output.txt
total 68
drwxrwxr-x 3 os os 4096 3월 31 17:35 .
drwxr-xr-x 23 os os 4096 3월 31 16:10 ..
-rw-r--r-- 1 os os 38 3월 31 17:35 file1.txt
-rwxrwxr-x 1 os os 8304 3월 31 16:37 hello2.o
-rw-rw-r-- 1 os os 79 3월 30 12:22 hello.c
-rwxrwxr-x 1 os os 8304 3월 31 16:35 hello.o
-rw-rw-r-- 1 os os 7023 3월 31 16:29 osh.c
-rwxrwxr-x 1 os os 13432 3월 31 16:31 osh.o
-rw-r--r-- 1 os os 0 3월 31 17:39 output.txt
drwxrwxr-x 2 os os 4096 3월 29 23:14 .vscode
osh>sort -r < output.txt
total 68
-rwxrwxr-x 1 os os 8304 3월 31 16:37 hello2.o
-rwxrwxr-x 1 os os 8304 3월 31 16:35 hello.o
-rwxrwxr-x 1 os os 13432 3월 31 16:31 osh.o
-rw-rw-r-- 1 os os 79 3월 30 12:22 hello.c
-rw-rw-r-- 1 os os 7023 3월 31 16:29 osh.c
-rw-r--r-- 1 os os 38 3월 31 17:35 file1.txt
-rw-r--r-- 1 os os 0 3월 31 17:39 output.txt
drwxr-xr-x 23 os os 4096 3월 31 16:10 ..
drwxrwxr-x 3 os os 4096 3월 31 17:35 .
drwxrwxr-x 2 os os 4096 3월 29 23:14 .vscode
```

▲ sort -r < output.txt 명령어 실행 모습

Input Redirect를 수행하여 output.txt의 내용을 sort-r 명령어의 표준 입력으로 받아 내림차순으로 정렬해 출력한 것을 확인할 수 있다.

• 명령어+옵션 < 파일명 &



▲ file2.txt의 내용

```
osh>cat < file2.txt &
PID: 2784
osh>example of Background standard input redirection
```

▲ cat < file2.txt & 명령어 실행 모습

부모 프로세스는 자식 프로세스를 생성한 뒤 pid를 출력하고 종료되었고, 자식 프로세스는 file2.txt의 내용을 cat의 표준 입력으로 받아 출력한 것을 확인할 수 있다.

• 명령어1+옵션 | 명령어2+옵션

```
osh>ls -al
total 60
drwxrwxr-x  3 os os  4096  3월 30 21:31 .
drwxr-xr-x 23 os os  4096  3월 30 20:49 ..
-rw-rw-r--  1 os os    79  3월 30 12:22 hello.c
-rwxrwxr-x  1 os os  8304  3월 30 21:25 hello.o
-rw-rw-r--  1 os os  6818  3월 30 21:15 osh.c
-rwxrwxr-x  1 os os 13432  3월 30 21:19 osh.o
-rw-r--r--  1 os os   442  3월 30 21:31 output2.txt
-rw-r--r--  1 os os   389  3월 30 21:29 output.txt
drwxrwxr-x  2 os os  4096  3월 29 23:14 .vscode
osh>ls -al | wc -l
10
```

▲ ls -al | wc -l 명령어 실행 모습

ls -al 명령의 결과를 파이프를 통해 wc -l의 입력으로 하여 ls 명령어의 라인수를 출력하는 것을 확인할 수 있다.

```
osh>ps -ef | grep bash
os          2259  2253  0 16:20 pts/0    00:00:00 bash
os          2530  2502  0 16:46 pts/0    00:00:00 grep bash
```

▲ ps -ef | grep bash 명령어 실행 모습

ps -ef 명령의 결과를 파이프를 통해 grep bash의 입력으로 하여 현재 실행중인 bash 프로세스를 full format으로 출력하는 것을 확인할 수 있다.

• 명령어1+옵션 | 명령어2+옵션 &

```
osh>ls -al | wc -l &
PID: 2478
osh>10
```

▲ ls -al | wc -l & 명령어 실행 모습

부모 프로세스의 경우 fork()를 통해 자식 프로세스를 생성한 뒤 자식의 종료를 기다리지 않고 pid를 출력하고 종료하고, 자식 프로세스의 경우 정상적으로 파이프 백그라운드 명령어를 실행하여 ls 명령어의 라인수를 출력하는 것을 확인할 수 있다.

```
osh>ps -ef | grep bash &
PID: 2502
osh>os          2259  2253  0 16:20 pts/0    00:00:00 bash
os          2700  2502  0 17:19 pts/0    00:00:00 grep bash
```

▲ ps -ef | grep bash & 명령어 실행 모습

부모 프로세스의 경우 fork()를 통해 자식 프로세스를 생성한 뒤 자식의 종료를 기다리지 않고 pid를 출력하고 종료하고, 자식 프로세스의 경우 정상적으로 파이프 백그라운드 명령어를 실행하여 현재 실행중인 bash 프로세스를 full format으로 출력하는 것을 확인할 수 있다.

- exit

```
osh>exit  
os@os-VirtualBox:~/proj1$
```

exit 명령어 입력시, shell이 정상적으로 종료되는 것을 확인할 수 있다.

4. 과제 수행하면서 경험한 문제점과 느낀 점

모듈화를 위해 각각의 케이스를 전부 분리하여 작업을 하려다 보니, 과제 수행 초반 약간의 어려움이 있었다. 특히 명령어 파싱 과정에서 NULL 포인터 처리를 제대로 진행하지 않아 segmentation fault 오류를 비롯한 다양한 오류들을 경험하였다. 디버깅을 하려해도 어느 부분에서 오류가 난 것인지 바로 확인하기가 어려웠다. 이러한 경험을 통해 처음 코드를 짤 때의 오류 처리의 중요성을 다시금 깨닫게 되었다.

수업시간에 이론으로 배웠던 개념들을 직접 함수로 구현하면서 프로세스의 생성과 부모 자식 간의 통신방식에 대해 이해할 수 있게 되었던 것 같다.