

2017313213 PA2 보고서

2017313213 박경태

개요

1. Executable Code 구현설명: *.c
2. exit, cd 구현 설명: builtin_command 함수
3. Built in Command 구현설명 및 path 상의 임의의 바이너리 구현 설명: my_execv
4. /bin 의 Command 실행 구현설명: my_execv
5. pipe 설명: eval()
6. redirection 설명: eval()
7. SIGINT, SIGSTOP 으로 멈추지 않게 하는 기능 설명: main()
8. background process 설명: while(waitpid(-1,&status,WNOHANG))

1. Executable Code 설명

1. head:

argc 가 모자라면 not enough parameter 오류를 출력하도록 설정

path 문자열을 설정한 뒤, getcwd 와 strcat 을 통해 현재 pwd 에 들어있는 파일에 접근토록 처리.

-n 옵션을 받으면 최상단부터 n 개의 라인을 출력하게 하는 기능 구현

기본값으로 n 은 10 으로 설정

파일디스크립터를 연 뒤, open 성공여부를 체크하고

추가적으로 file descriptor 의 디렉토리여부를 체크하여 디렉토리라면 EISDIR 에러를 반환토록 구현

파일 무사히 open 시, 한 글자씩 표준출력을 통해 write 하도록 구현.

line_cnt 를 하여 줄의 수를 셈.

2. tail:

argc 가 모자라면 not enough parameter 오류를 출력하도록 설정

path 문자열을 설정한 뒤, getcwd 와 strcat 을 통해 현재 pwd 에 들어있는 파일에 접근토록 처리.

-n 옵션을 받으면 최하단부터 n 개의 라인을 출력하게 하는 기능 구현

기본값으로 n 은 10 으로 설정

파일디스크립터를 연 뒤, open 성공여부를 체크하고

추가적으로 file descriptor 의 디렉토리여부를 체크하여 디렉토리라면 EISDIR 에러를 반환토록 구현

파일 무사히 open 시, 한 글자씩 표준출력을 통해 write 하도록 구현.

line_cnt 를 하여 줄의 수를 셈

tail 의 경우, head 와 다르게 줄의 끝부터 세어야 하므로, 먼저 전부 세서 total line number 를 획득한 뒤 lseek 을 통해 file descriptor 를 rewind 함.

3. cat:

argc 가 모자라면 not enough parameter 오류를 출력하도록 설정

path 문자열을 설정한 뒤, getcwd 와 strcat 을 통해 현재 pwd 에 들어있는 파일에 접근토록 처리.

파일디스크립터를 연 뒤, open 성공여부를 체크하고

추가적으로 file descriptor 의 디렉토리여부를 체크하여 디렉토리라면 EISDIR 에러를 반환토록 구현

파일 무사히 open 시, 한 글자씩 표준출력을 통해 write 하도록 구현.

4. cp:

argc 가 모자라면 cp: missing file operand 또는 cp: missing destination operand after FILE1

이라는 독특한 오류를 출력하도록 설정.

filename, filename2 를 설정하고 전자가 from, 후자가 to 의 의미를 갖는다.
전자는 RDONLY, 후자는 WRONLY 로 연 뒤, 한 글자씩 read->write 해서 써내려간다.

5. mv:

argc 인자가 모자라면 mv: missing file operand 또는 mv: missing destination file operand after 'SOURCE' 라는 독특한 에러를 출력토록 처리한다.

mv 는 이름을 바꾸기도 하지만, 디렉토리를 옮기는 기능도 있다.

이름을 바꾸는 기능의 경우, rename 으로써 단순히 처리하면 되지만 디렉토리의 경우 약간 다르다.

만약 절대경로임이 확실하다면 생각할게 적은데, 그냥 새 이름에 낡은 이름을 덮어써버리면 된다.

절대경로가 아니라면, 우선 디렉토리를 여는 부분을 추가적으로 구현해야 하기 때문에 DIR 객체선언 및 opendir 함수를 사용한 뒤 일반파일명이나 상대경로 디렉토리명이나에 따라 케이스를 나눈다.

일반파일명이라면 이름을 바꾸는 단순작업이므로 역시 그냥 이름을 덮어쓴다. 다만, 상대경로 디렉토리명이라면 우선 현재경로를 getcwd 로 저장한 뒤, strcat /newname/oldname 을 통해 "폴더를 옮기는" 작업을 실행하도록 한다.

6. rm:

인자 수가 적으면 Not enough Parameter 오류를 내도록 처리했다.

rm 에는 여러 인자가 들어올 수 있으므로, 들어온 인자 하나하나마다 전부 unlink 를 처리하도록 for 루프를 돌게 하였고 에러가 발생시 perror 호출 뒤 exit 토록 처리하였다.

7. pwd:

현재 디렉토리를 확인해주는 함수이므로, 매우 간단히 구현할 수 있다.

getcwd 함수를 통해 현재 디렉토리를 표준출력으로 printf 한 뒤 종료한다.

2. exit, cd 구현 설명: builtin_command 함수

exit, cd 의 경우, 일반적으로 redirection 될 경우도 없고 독특한 기능을 수행하므로 따로 builtin_command 로 구현했으며 이는 minishell.c 의 builtin_command 함수로 구현되었다.

exit: 일단 뒤에 숫자가 따라붙는지의 여부에 따라 반환값이 달라진다. 그런데, 할당안된 메모리를 괜히 참조했다가는 SEGMENTATION FAULT 가 발생하므로 argc 를 추가적으로 받게끔 했고, argc 가 1 이라면 기본값인 exit 0, 2 라면 2 번째 인자를 atoi 로 변환한 뒤 그 값을 exit 시 매개변수로 반환토록 하였다.

cd:

일단 인자가 2 개가 아니라면(cd DIRECTORY) not enough parameter 오류를 내도록 처리했다. 일단 상대경로만 고려하므로 구현은 비교적 간단하다.

mv 와 유사하게 path 를 getcwd 로 우선 받고 /argv[1] 를 strcat 해서 path 를 완성시킨다.

chdir 을 통해 디렉토리를 변경한 뒤 에러여부를 fprintf(stderr) 로 체크하고 종료한다.

```
/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv, int argc) {
    // printf("argc: %d\n", argc);
    // print_argv(argv);
    if (strcmp(argv[0], "exit") == 0) {
        int ret;
        if (argc == 1)
            ret = 0;
        else
            ret = atoi(argv[1]);
        fprintf(stderr, "exit(%d)", ret);
        exit(ret);
    } else if (strcmp(argv[0], "cd") == 0) {
        if (argc != 2) {
            fprintf(stderr, "cd: not enough parameter\n");
            return 1;
        }
        char path[MAXLINE];
        path[0] = 0;
        getcwd(path, MAXLINE);
        strcat(path, "/\0");
        strcat(path, argv[1]);
        strcat(path, "\0");
        int status = chdir(path);
        if (status != 0) {
            fprintf(stderr, "%s: %s\n", argv[0], strerror(errno));
            return 1;
        }
        getcwd(path, MAXLINE);
        return 1;
    }
    return 0;
}
```

3. Built in Command 구현설명 및 path 상의 임의의 바이너리 구현 설명: my_execv

앞서 직접 구현한 head, tail, cat 등등의 7 개 함수는 내가 직접 구현한 것이므로, /bin 이 아니라 ./pa2 가 있는 디렉토리에 있는 직접 빌드한 결과물의 head, tail, 등등의 프로그램을 참조하도록 하여야 한다.

따라서, original_path 를 main 함수 실행 직후 저장한다. 왜냐하면 pa2 와 동일한 디렉토리에 해당 7 개 프로그램이 존재하므로 이 7 개 함수를 실행하고자 한다면 cd 로 어느 디렉토리를 순방중이더라도 제각 pa2 의 디렉토리로 접근해야하기 때문이다.

is_implemented() 함수를 통해 명령어가 7 개 함수에 속한다면, 또는 ./ 로 시작하는 현재 디렉토리의 executable 을 의미한다면 1 을, 아니라면 0 을 반환토록 한다.

0 을 반환 시에는 /bin 에서 끌어와서 쓰게 되는데, 이에 관련된 이야기는 4 장에서 하도록 한다.

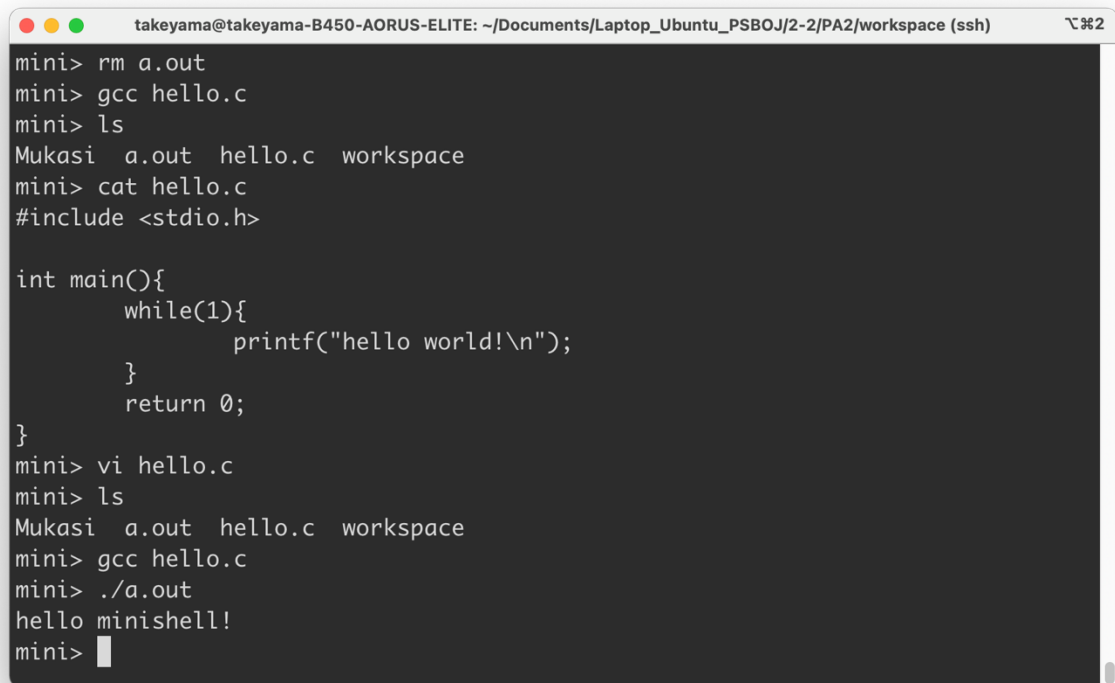
1 을 반환 시에, 내가 직접 구현한 커맨드 또는 현재 디렉토리에 있는 커맨드라는 것인데 myexecv 함수에서 is_implemented 결과가 참일 경우, 우선 ./ 로 시작하는지를 판별한다.

./ 로 시작할 경우, 현재 디렉토리의 executable 임이 확실하므로 getcwd 를 통해 현재 디렉토리의 path 를 저장한 뒤, ./ 를 없애야 하므로 argv[0]+2 를 한다.

아니라면, 7 개 함수라는 뜻이므로 original_path 를 참조하도록 한다.

그 뒤, 일괄적으로 path 에 strcat /argv[0] 하여 최종경로를 완성한다.

그리고 execv 시킨다.

A terminal window titled 'takeyama@takeyama-B450-AORUS-ELITE: ~/Documents/Laptop_Ubuntu_PSBOJ/2-2/PA2/workspace (ssh)' with a temperature icon showing 22°C. The terminal shows a series of commands in a 'mini' shell: 'rm a.out', 'gcc hello.c', 'ls' (output: 'Mukasi a.out hello.c workspace'), 'cat hello.c' (output: C code for a 'hello world' program), 'vi hello.c', 'ls' (output: 'Mukasi a.out hello.c workspace'), 'gcc hello.c', './a.out' (output: 'hello minishell!'), and 'mini>' with a cursor.

```
takeyama@takeyama-B450-AORUS-ELITE: ~/Documents/Laptop_Ubuntu_PSBOJ/2-2/PA2/workspace (ssh) 22°C
mini> rm a.out
mini> gcc hello.c
mini> ls
Mukasi a.out hello.c workspace
mini> cat hello.c
#include <stdio.h>

int main(){
    while(1){
        printf("hello world!\n");
    }
    return 0;
}
mini> vi hello.c
mini> ls
Mukasi a.out hello.c workspace
mini> gcc hello.c
mini> ./a.out
hello minishell!
mini> 
```

위와 같이, minishell 내부에서 vi 를 통한 문서편집 및 gcc 컴파일 및 컴파일된 임의의 프로그램까지 minishell 내부에서 실행가능함을 확인했다.

4. /bin 의 Command 실행 구현설명: my_execv

builtin_command 에서도 exit, cd 가 아님이 판명되었으며
is_implemented(argv) 의 결과가 거짓이라면, 직접 구현한 커맨드가 아니라는 뜻이다.

그렇다면 자연스럽게 /bin 경로의 해당하는 프로그램이 있나 검색을 하게 되고, 에러가 발생한다면
에러메시지를 출력하게 된다.

/bin/ 의 경로를 참조하게 되므로, path 의 전처리는 sprintf (path, /bin/%s, argv[0]) 으로 간단하게
할 수 있으며, 구현 역시 양호하게 작동한다.

```
void my_execv(char *file, char **argv) {
    char path[MAXLINE];
    path[0] = 0;
    if (is_implemented(argv)) { // 직접구현했다면
        if (argv[0][0] == '.' && argv[0][1] == '/') { // 현재폴더라면
            getcwd(path, MAXLINE);
            argv[0] += 2;
        } else {
            strcat(path, original_path);
        }
        strcat(path, "/\0");
        strcat(path, argv[0]);
        argv[0] = path;
    } else { // 미구현이라 bin/bash 에서 끌어쓴다면
        sprintf(path, "/bin/%s", argv[0]);
        argv[0] = path;
    }
    if (execv(argv[0], argv) == -1) {
        fprintf(stderr, "%s\n", strerror(errno));
        return;
    }

    return;
}
```

5. pipe 설명: eval()

eval 로부터 raw_cmd 라는 매우 긴 string 인자를 받아옴을 전제로 한다.

raw_cmd 를 우선 argv 라는 2 차원 char 배열에 공백, 탭, 개행에 대해 strtok 하여 파싱한다.

그 뒤, argv 의 마지막 인자를 NULL 로 한다.

그 뒤, 파싱된 argv 를 순회하며 <일 때, > 일 때, >> 일 때, | 일 때로 구분하여 해당 특수문자를 또한 NULL 로 바꾸고, pipeline 의 시발점이라면 이를 pipeline_index[p_cnt+1] 에 저장한다.

예를 들어,

grep is < input.txt | grep it | grep you > output.txt 라면

grep is \0 input.txt \0 grep it \0 grep you \0 output.txt

0 1 2 3 4 5 6 7 8 9 10 11

이 되고, pipeline 은 0, 5, 8 번 int 를 가리키게 된다.

그 뒤, input_redirection 또는 output_redirection 이 있다면 0 에서 1 또는 2 로 설정한 뒤 redirection path 또한 따로 저장해놓는다.

추가적으로 & 이 있다면 이는 background 를 true 로 설정하는 작업을 거친다.

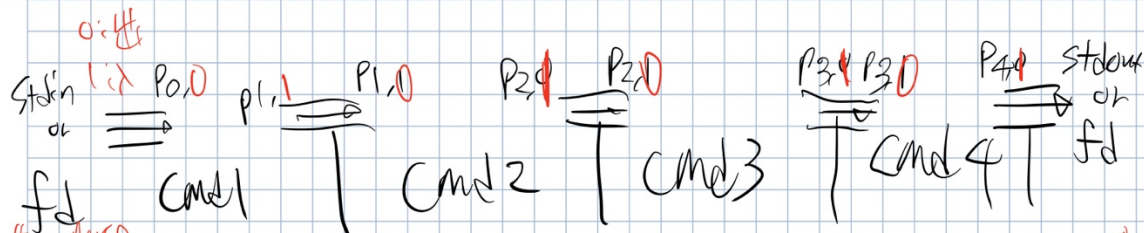
그 뒤, 모든 파이프라인에 대해 작업을 수행하게 되는데

파이프라인이 1 개 이상이면서 i 가 마지막 파이프라인이 아닌 경우, pipe () 를 통해 파이프를 생성하게 되며

fork 를 해서 자식프로세스와 부모프로세스를 분리시킨다. 이러한 주요분기점에 switch 문을 사용하면 if-else-if-if-else 식의 if 문의 연쇄보다 훨씬 더 가독성높은 코드작성이 가능할 뿐더러 추가적으로 빠르다.

아래 첨부한 사진과 같이, 케이스는 3 가지로 나뉜다.

1. 첫 파이프라라인일 때
 2. 마지막 파이프라라인일 때
 3. 첫 번째와 마지막 사이의 파이프라라인일 때
- 각각의 경우 아래와 같이 처리하기로 했다.



#cc=parce
for [cmd x] x: 1 ~ n
Pipe(P[x])

fork => CHLD: if prev fd
dup2(P[x][0], stdin)
close(P[x][0])
close(P[x][1])
if next fd
close(P[x][1])
dup2(P[x][1], stdout)
close(P[x][1])
if cmd[x] on <, >, >> fd

~~P[x][0] = P[x][1]~~
P[x][0] cmd[x] P[x][1]

grep aaa <input.txt | grep bbb

if >: fdw=open, dup2(fdw, 1)
if >>: fdw=open, dup2(fdw, 1)
if <: fdw=open
dup2(fdw, 0)

P[x][0] → P[x][1]
exec cmd[x]
exit(0) → cmd[x] = char**

cmd[0][0] = grep /o
cmd[0][1] = aaa /o
cmd[0][2] = < /o
cmd[0][3] = input.txt /o
cmd[0][4] = NULL

PRT: wait(NULL)
if prev fd
close(P[x][0])
close(P[x][1])
if next fd
P[x] = P[x+1]

(closeAllfds)

파일 디스크립터 및 파이프 처리가 전부 끝나면, my_execv 함수를 호출하고 만일 my_execv 함수에서 exit 되지 않고 다음단계로 넘어갔다면 이는 에러가 발생했다는 것이므로 mini:command not found 에러를 출력한다.

```
if (p_cnt > 0) { //파이프가 1개 이상
    if (i != 0) {
        dup2(l_pipe[STDIN_FILENO], STDIN_FILENO);
        close(l_pipe[STDIN_FILENO]);
        close(l_pipe[STDOUT_FILENO]);
    }
    if (i != p_cnt) {
        dup2(r_pipe[STDOUT_FILENO], STDOUT_FILENO);
        close(r_pipe[STDOUT_FILENO]);
        close(r_pipe[STDIN_FILENO]);
    }
}
my_execv(argv[pipe_index[i]], &argv[pipe_index[i]]);
fprintf(stderr, "mini: command not found\n");
exit(0);
```

부모의 경우, 우선 switch 이전에 열린 pipe 를 close 해야하지만 이 또한 케이스가 나뉜다.

1. 처음 파이프일 때 -> 애초에 파이프가 안열렸으므로 닫을 필요가 없다
 2. 두 번째 이상 이면서 다음 파이프가 있을 때 -> 다음 파이프에 연결시킨다.
- 2 번의 경우, 추가적으로 예외처리를 해야하지만 사실 어차피 마지막 파이프의 경우 애초에 파이프라인이 열리지 않은 빈 int r_pipe[2] 이기에 큰 영향이 없어서 크게 문제가 발생하지 않는다.

추가적으로, background 구현 또한 부모단에서 돌아가게 되는데, 이에 관한 자세한 설명은 8 장에서 하도록 한다.

6. redirection 설명: eval()

사실 5 장에서 말한 "3 가지의 경우" 는 3 가지가 아니라 엄밀히 말하면 5 가지가 된다.

1. 첫 파이프라인일 때

1-1. 첫 파이프라인에 input redirection 이 있을 때

1-2. 첫 파이프라인에 input redirection 이 없을 때

2. 마지막 파이프라인일 때

2-1. 마지막 파이프라인에 output redirection 이 있을 때

2-2. 마지막 파이프라인에 output redirection 이 없을 때

3. 첫 번째와 마지막 사이의 파이프라인일 때

input redirection 및 output/append direction 의 여부는 parsing 단계에서 flag 를 통해 true/false 로 알 수 있고, 경로 또한 input_path, output_path 로써 알고있으므로 매우 쉬운 작업이 된다.

추가적으로, file descriptor 복원문제 또한

자식프로세스 분기 이후 생성된 open 이라 부모측에서 따로 close 하고 또 STDIN/OUT 을 복원하지 않아도 되며(부모는 STDIN/OUT 을 그대로 유지하고 있으므로)

자식프로세스에서 pipe 단에서 close 가 매우 잘 이루어지고 있기 때문에 걱정할 필요가 없다.

말한대로, 첫 파이프라인과 마지막 파이프라인일 때만 신경써서 redirection 분기를 태워주면 된다.

```

switch (pid = fork()) {
    case -1:
        perror("fork failed");
        break;
    case 0: //자식
        if ((i == 0) && input_redirection == 1) {
            int input_fd = open(input_path, O_RDONLY);
            if (input_fd < 0) {
                fprintf(stderr, "mini: No such file or directory\n");
                exit(0);
            }
            dup2(input_fd, STDIN_FILENO);
            close(input_fd);
        }
        if ((i == p_cnt) && (output_redirection > 0)) {
            if (output_redirection == 1) {
                int output_fd = open(output_path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                if (output_fd < 0) {
                    fprintf(stderr, "mini: No such file or directory\n");
                    exit(0);
                }
                dup2(output_fd, STDOUT_FILENO);
                close(output_fd);
            } else if (output_redirection == 2) {
                int output_fd = open(output_path, O_WRONLY | O_CREAT | O_APPEND, 0666);
                if (output_fd < 0) {
                    fprintf(stderr, "mini: No such file or directory\n");
                    exit(0);
                }
                dup2(output_fd, STDOUT_FILENO);
                close(output_fd);
            } else {
                fprintf(stderr, "mini: No such file or directory\n");
                exit(0);
            }
        }
    }
    if (p_cnt > 0) { //파이프가 1개 이상
        if (i != 0) {
            dup2(l_pipe[STDIN_FILENO], STDIN_FILENO);
            close(l_pipe[STDIN_FILENO]);
            close(l_pipe[STDOUT_FILENO]);
        }
        if (i != p_cnt) {
            dup2(r_pipe[STDOUT_FILENO], STDOUT_FILENO);
            close(r_pipe[STDOUT_FILENO]);
            close(r_pipe[STDIN_FILENO]);
        }
    }
    my_execv(argv[pipe_index[i]], &argv[pipe_index[i]]);
    fprintf(stderr, "mini: command not found\n");
    exit(0);
}

```

7. SIGINT, SIGSTOP 으로 멈추지 않게 하는 기능 설명: main()

원래는 SIGINT, SIGSTOP 의 신호를 받으면 내가 만든 쉘도 멈춰야하지만, 만일 그렇게 된다면 minishell 내에서 executable 을 돌릴 때 Ctrl C 시, executable 과 동시에 minishell 도 꺼지는 문제점이 있기에 이를 면역시키는 것이 불가피하다. SIGNAL 처리는 main 시발점에서 signal 함수를 통해 설정하였으며, void sig_handler(int signo) 함수가 아무것도 동작하지 않도록 설정했고 signal(SIGINT, *void*)sig_handler), signal(SIGSTOP, *void*)sig_handler) 에 대해 모두 돌아가게 하였으므로 해당 시그널 무시기능은 무난히 구현할 수 있었다.

```
void sig_handler(int signo) {
    return;
}

int main() {
    char cmdline[MAXLINE]; /* Command line */
    char *ret;

    getcwd(original_path, MAXLINE);
    // printf("original path = %s\n", original_path);
    signal(SIGINT, (void *)sig_handler);
    signal(SIGSTOP, (void *)sig_handler);

    while (1) {
        /* Read */
        printf("mini> ");
        ret = fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin) || ret == NULL)
            exit(0);

        /* Evaluate */
        eval(cmdline);
    }
}
```

8. background process 설명: while(waitpid(-1,&status,WNOHANG))

5장에서 설명했듯이,

부모의 경우, 우선 switch 이전에 열린 pipe 를 close 해야하지만 이 또한 케이스가 나뉜다.

1. 처음 파이프일 때 -> 애초에 파이프가 안열렸으므로 닫을 필요가 없다

2. 두 번째 이상 이면서 다음 파이프가 있을 때 -> 다음 파이프에 연결시킨다.

2 번의 경우, 추가적으로 예외처리를 해야하지만 사실 어차피 마지막 파이프의 경우 애초에 파이프라인이 열리지 않은 빈 int r_pipe[2] 이기에 큰 영향이 없어서 크게 문제가 발생하지 않는다.

```
default: //부모
    if (i > 0) {
        close(l_pipe[STDIN_FILENO]);
        close(l_pipe[STDOUT_FILENO]);
    }
    l_pipe[STDIN_FILENO] = r_pipe[STDIN_FILENO];
    l_pipe[STDOUT_FILENO] = r_pipe[STDOUT_FILENO];
    if (!background) {
        if (waitpid(pid, &status, 0) < 0)
            fprintf(stderr, "Error occured: %d\n", errno);
    } else {
        while (waitpid(-1, &status, WNOHANG)) {
            ;
        }
    }
    break;
```

이 때, 5 장의 parsing 때 & 가 발견되어 background 가 true 가 되었다면, background 에서 돌아가도록 해야하는데

평상시에는

```
if(waitpid(pid,&status,0)<0)
```

error 출력

식으로 대기토록 하면 되지만, background 에서 돌아가야 하므로

```
while(waitpid(-1,&status,WNOHANG)){
```

```
    ;
```

```
}
```

로 대체하면 된다. 이렇게 하면, 모든 회수되지 않은 자식을 일괄회수한 뒤 더 이상 회수할 자식이 없다면 넘어가게 됨을 대기없이 반복하게 되기 때문이다. 따라서 좀비프로세스도 예방할 수 있다.

실제로 10 주차에 배운 echo_server client 프로그램을 백그라운드로 실행시키고 확인해보면

위: 실제 shell

아래: 자작 shell

```
takeyama@takeyama-B450-AORUS-ELITE: ~/Documents/Laptop_Ubuntu_PSB0J/2-2/week10/mukashi (ssh)
a      client      echo_server      w10_client      w10_server.c
a_copy 'client copy.c' echo_server.c w10_client.c w10_server.o
takeyama@takeyama-B450-AORUS-ELITE:~/Documents/Laptop_Ubuntu_PSB0J/2-2/week10/mukashi$
./echo_server 10000 &
[1] 11441
takeyama@takeyama-B450-AORUS-ELITE:~/Documents/Laptop_Ubuntu_PSB0J/2-2/week10/mukashi$
./echo_client localhost 10000
a
got 2 bytes from client.
a
apple
got 6 bytes from client.
apple
apple
got 6 bytes from client.
apple
apple
got 6 bytes from client.
apple
apple
got 6 bytes from client.
apple
banana
got 7 bytes from client.
banana
█
```

```
takeyama@takeyama-B450-AORUS-ELITE: ~/Documents/Laptop_Ubuntu_PSB0J/2-2/PA2/workspace (ssh)
mini> ./echo_client localhost 10000
a
a
^Cmini> ./echo_client localhost 10010
connect() failed
mini> ./echo_server 10010 &
mini> ./echo_client localhost 10010
a
got 2 bytes from client.
a
apple
got 6 bytes from client.
apple
apple
got 6 bytes from client.
apple
apple
got 6 bytes from client.
apple
banana
got 7 bytes from client.
banana
█
```

동일한 결과로써, 서버가 백그라운드에서 돌아가고 got %d bytes from client 를 잘 보내고 있음을 확인할 수 있다.