shell 구현

2023350031 스마트보안학과 김은채

Introduction

이번 시스템 해킹 2주차 과제인 쉘 프로그램 구현하기에서 중점을 뒀던 부분은 다음과 같습니다

- 기본적인 과제의 조건 사항을 만족하는가?
- 실제 쉘을 이용하는 것처럼 사용할 수 있는가?
- 좀비 프로세스 등 보안 위협 요소들을 고려하였는가?

위의 사항들을 고려하면서 여러 사람들이 어떻게 쉘을 구현하였는가 살펴봤을 때 https://github.com/renkangchen/myShell/tree/master에서 구현에 용이한 readline 등의 라이브러리를 사용하는 것을 확인했습니다. 또한 라인을 입력받고 라인을 분석하는 함수를 수행 후 어떤 명령인지에 따라 어떤함수를 실행해야할지 나눠주는 과정이 이해하기 쉬운 구조라고 생각했습니다. 따라서 해당 소스를 쉘 구현할 때 참고하되 여러 보안 취약점들을 고려하면서 과제를 진행했습니다.

Explanation

구조는 다음과 같습니다.

- shell.h : .c 파일에서 공통으로 사용되는 것들을 정의해놓은 헤더파일
- history.c : 커맨드 히스토리를 저장하고 불러오는 역할
- prompt.c : 프롬프트를 표시하고, 문자열 파싱, 자식 프로세스 좀비 방지 등의 역할을 함
- execommnad.c : 사용자 입력의 구조에 따라 어떤 함수를 실행할지 결정
- builtin.c : exit, cd, pwd 등 빌트인 명령어 처리
- shellfunct.c : 파이프라인, 백그라운드 등등 다양한 명령을 처리하는 함수들이 모인 c 파일
- main.c : 프로그램 진입점

하나하나 자세히 설명해보도록 하겠습니다.

shell.h

```
#ifndef SHELL_H
#define SHELL_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pwd.h>
#include <wait.h>
#include <fcntl.h>
#include <sys/types.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <signal.h>

#define MAX_ARG 100
#define MAX_CMD 1024
```

```
void sigchld_handler(int sig);
void set_prompt(char *prompt);
void execute_command(char *cmd);
void history_setup();
void history_finish();
int run_builtin(char **args);
int parse_args(char *cmd, char **args);

int execute_single(char *cmd);
int execute_pipeline(char *line);
int execute_logical(char *line);
int execute_background(char *line);
int execute_sequential(char *line);

#endif
```

shell.h는 .c 파일에서 공통으로 사용되는 것들을 정의해놓은 헤더파일입니다. (각 함수는 추후 자세히 설명하겠습니다) 이 과제에서는 readline 라이브러리를 사용했습니다. readline은 GNU에서 제공하는 라이브러리로 커맨드라인 인터페이스를 구현할 때 용이하게 사용할 수 있는 함수들이 있습니다. 대표적으로 위, 아래 방향키로 입력했던 커맨드라인 히스토리들을 이동할 수 있고 add_history(), using_history() 등으로 히스토리들을 관리할 수 있는 기능을 제공합니다. 더불어 readline 함수를 통해 입력줄을 읽고 문자열을 반환하는 과정을 간단하게 구현할 수 있었습니다. 따라서 관련 함수들을 이용하기 위해서 include 해주었습니다. MAX_ARG는 사용자가 커맨드라인에 다양한 인자를 사용하여 입력을 줄 때 최대로 사용할 수 있는 인자 수를 지정해놓은 것입니다. MAX_CMD는 입력 가능한 최대 커맨드라인의 길이를 정의해놓은 것입니다. 그리고 헤더 파일에서 함수 선언부를 모아놓고 추후 c 파일에서 함수를 정의했습니다.

참고자료 ; https://tiswww.case.edu/php/chet/readline/rltop.html

history.c

```
#include "shell.h"

void history_setup() {
    using_history();
    stifle_history(50);
    read_history("/tmp/msh_history");
}

void history_finish() {
    append_history(history_length, "/tmp/msh_history");
    history_truncate_file("/tmp/msh_history", 50);
}
```

history_setup는 히스토리 기능을 셋업하는 함수입니다. using_history는 readline의 히스토리 사용을 위한 초기화 함수입니다. 히스토리 리스트를 위한 메모리 공간을 할당하게 됩니다. stifle_history는 히스토리의 최대 개수를 지정해서 최근 50개의 히스토리만 저장될 수 있도록 합니다. read_history는 과거 히스토리 목록을 불러와서 적재하는 역할을 합니다. 이를 통해 사용자는 위아래 방향키를 통해 히스토리를 탐색할 수 있게 됩니다.

history_finish는 쉘 종료 전에 명령어 기록을 저장하는 마무리 역할을 하는 함수입니다. append_history 함수는 현재 히스토리 명령어 개수 (history_length)를 인자로 해서 /tmp/msh_history에 append 합니다. 이렇게 append된 기록을 history_truncate_file을 통해 오래된 줄부터 삭제, 최근 50개의 명령어만 저장될 수 있도록 합니다.

prompt.c

```
#include "shell.h"
```

```
void sigchld_handler(int sig) {
    (void)sig;
    while (waitpid(-1, NULL, WNOHANG) > 0);
}

void set_prompt(char *prompt) {
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    struct passwd *pw = getpwuid(getuid());
    sprintf(prompt, "[%s@localhost:%s]$ ", pw->pw_name, cwd);
}

int parse_args(char *cmd, char **args) {
    int argc = 0;
    char *token = strtok(cmd, " \t\n");
    while (token && argc < MAX_ARG - 1) {
        args[argc++] = token;
        token = strtok(NULL, " \t\n");
    }
    args[argc] = NULL;
    return argc;
}
```

prompt.c는 프롬프트를 표시하고, 문자열 파싱, 자식 프로세스 좀비 방지 등의 역할을 하는 함수들을 모아놓은 파일입니다.

sigchld_handler는 좀비 프로세스 방지를 위해 구현한 코드입니다. 자식 프로세스가 종료되면 부모 프로세스가 종료 상태를 처리해야하는데 이를 처리 하지 않아서 발생하는 문제를 말합니다. 이런 좀비 프로세스가 계속해서 쌓이게 된다면 전반적으로 시스템의 성능 저하를 일으킬 수 있기 때문에 이를 적절히 처리하는 과정이 필요합니다. 자식 프로세스가 종료될 때 SIGCHLD 시그널이 발생하는데 sigchld_handler는 이 시그널을 처리하는 역할을 합니다. 이 함수는 추후 main에서 signal 함수를 이용할 때 사용되는데, 이 signal 함수를 사용할 때 핸들러 함수는 sig를 인자로 받는 형태여야 합니다. 하지만 실제로는 sig를 사용하지 않기 때문에 void 처리를 해줘서 에러가 뜨지 않도록 우선 처리해줍니다. 이후 while 문을 돌면서 waitpid를 통해 종료된 자식 프로세스들을 수거합니다. waitpid는 자식 프로세스가 종료되길 기다리는 함수로 임의의 자식 프로세스(-1) 대상으로 어떻게 종료되었는지는 무시하고(NULL) 종료된 자식이 있다면 즉시 반환처리(WNOHANG) 동작을 지시합니다. 이를 통해 좀비 프로세스의 누적을 방지합니다.

참고자료 ;

https://jaduwvad.tistory.com/139

https://github.com/jaduwvad/SystemSoftwareSecurity/blob/master/process/shandler.c

set_prompt는 쉘에 표시되는 프롬프트를 설정하는 함수입니다. cwd는 현재 디렉토리를 표시하는 문자열이고 getcwd 함수를 통해 현재 위치한 경로를 얻어서 cwd에 저장합니다. 이후 현재 사용자 ID를 getuid 함수로부터 가져오고 그에 해당하는 사용자 정보 struct를 getpwid 함수로부터 설정합니다. 여기에는 이름이나 홈 디렉토리, 쉘 정보등이 포함되게 됩니다. 최종적으로 이렇게 얻은 정보들을 sprintf함수를 통해 프롬프트 형식으로 출력합니다. 인자는 pw로부터 pw_name, cwd를 가져오게 됩니다. 저는 리눅스 가상머신 환경을 이용하고 있기 때문에 localhost로 표시하도록 해주었습니다.

parse_args는 커맨드를 단위별로 쪼개고 각 토큰을 args[] 배열에 저장하는 역할을 합니다. 우선 int argc=0으로 초기화 해줍니다. argc는 인자 개수를 카운트하는 역할을 하게 됩니다. 이후 strtok으로 공백, 탭, 엔터를 기준으로 첫 토큰을 분리, 이를 포인터 처리합니다. 그리고 while문을 돌면서 argc가 인자 제한 (MAX_ARG)을 넘지 않을때 토큰들을 args 배열에 계속 추가하면서 파싱을 반복합니다. 그리고 배열 마지막에 NULLL을 추가해주고 파싱된 토큰의 개수 argc를 리턴하고 함수는 종료됩니다.

execommand.c

```
#include "shell.h"

void execute_command(char *line) {
    if (strchr(line, ';')) {
        execute_sequential(line);
```

```
    } else if (strstr(line, "&&") || strstr(line, "||")) {
        execute_logical(line);
    } else if (strchr(line, '|')) {
        execute_pipeline(line);
    } else if (strchr(line, '&')) {
        execute_background(line);
    } else {
        execute_single(line);
    }
}
```

execommand.c는 라인 분석에 따라 어떤 명령을 실행할 것인지 결정하는 execute_command가 위치한 파일입니다. 여기서는 strchr을 이용해 문자열에서 ;, && 등에 해당하는 첫번째 표시가 있는지 찾습니다. ;를 찾았을 경우 execute_sequential로, &&이나 ||을 찾았을 경우 execute_logical로, | 을 찾았을 경우 execute_pipeline, &을 찾았을 경우 execute_background, 해당하는 것이 없을 경우 execute_single로 분기처리합니다. execute로 시작하는 각 함수들은 기호 표시에 따라 파이프라인, 백그라운드 등 다양한 명령을 처리하게 됩니다. 분기문을 작성할때 &을 먼저 두면 &&이라도 &으로 처리될 가능성이 있기 때문에 &&을 먼저 분기할 수 있도록 처리해두었습니다.

참고자료 ; https://www.ibm.com/docs/ko/i/7.3.0?topic=functions-strchr-search-character

builtin.c

```
#include "shell.h"

int run_builtin(char **args) {
    if (!args[0]) return 1;

    if (strcmp(args[0], "exit") == 0) {
        exit(0);
    } else if (strcmp(args[0], "cd") == 0) {
        const char *path = args[1];
        if (!path || strcmp(path, "~") == 0) {
            path = getenv("HOME");
        }
        if (chdir(path) != 0) {
            perror("cd failed");
            return 1;
        }
        return 0;
    } else if (strcmp(args[0], "pwd") == 0) {
        char cwd[1024];
        getcwd(cwd, sizeof(cwd));
        printf("%s\n", cwd);
        return 0;
    }

    return -1;
}
```

bash 쉘에는 실행파일이 따로 없이 자체적으로 지원하는 명령어가 있는데 이를 처리하기 위한 파일이 builtin.c입니다. 여기서는 exit, cd, pwd 등을 구현해 놓았습니다. 여기서는 위에서 토크나이즈한 args 배열을 이용합니다.

이 배열의 !args[0]일 경우 유효한 커맨드가 아니므로 return 1을 하게 됩니다.

args[0]와 “exit”를 strcmp를 이용해 비교했을때 같으면 exit(0)을 통해 쉘 프로그램을 종료처리합니다.

args[0]와 “cd”를 비교했을 때 같으면 args[1]을 path 포인터 처리하고 인자가 없거나, ~면 home 디렉토리로 이동하도록 처리합니다. (cd ~ 입력했을 때 홈으로 이동해야함) 그리고 chdir(path)를 통해 디렉토리를 이동하도록 하고 만약 이동이 실패시 “cd faild” 처리가 되고 1을 리턴합니다. cd 명령어 성공시에는 0을 리턴합니다.

args[0]와 “pwd”를 비교했을 때 같으면 경로를 담을 cwd 배열을 선언하고 getcwd 함수를 통해 현재 디렉토리의 위치를 cwd에 담습니다. 이후 cwd를 print함으로써 pwd 명령어를 구현했습니다.

이후 exit, cd, pwd 그 어떤 것도 아니면 빌트인 명령어에서 처리할 일이 아니므로 -1을 반환합니다.

참고자료 ; https://zetawiki.com/wiki/Bash_빌트인_명령어

shellfunct.c

```
#include "shell.h"

int execute_single(char *cmd) {
    char tmp[1024];
    char *args[MAX_ARG];

    strncpy(tmp, cmd, sizeof(tmp) - 1);
    tmp[sizeof(tmp) - 1] = '\0';

    char *redir = strchr(tmp, '>');
    int redirect = 0;
    char *outfile = NULL;

    if (redir) {
        *redir = '\0';
        redir++;
        while (*redir == ' ') redir++;
        outfile = strtok(redir, " \t\n");
        redirect = 1;
    }

    parse_args(tmp, args);
    int builtin_result = run_builtin(args);
    if (builtin_result >= 0) return builtin_result;

    pid_t pid = fork();
    if (pid == 0) {
        if (redirect && outfile) {
            int fd = open(outfile, O_CREAT | O_WRONLY | O_TRUNC, 0644);
            if (fd < 0) {
                perror("open failed");
                exit(1);
            }
            dup2(fd, STDOUT_FILENO);
            close(fd);
        }
        execvp(args[0], args);
        perror("execvp failed");
        exit(1);
    } else {
        int status;
        waitpid(pid, &status, 0);
        return WIFEXITED(status) ? WEXITSTATUS(status) : 1;
```

```
    }
}

int execute_pipeline(char *line) {
    char *cmds[MAX_ARG];
    int count = 0;
    char *cmd = strtok(line, "|");

    while (cmd && count < MAX_ARG - 1) {
        cmds[count++] = cmd;
        cmd = strtok(NULL, "|");
    }
    cmds[count] = NULL;

    int in_fd = 0, fd[2];

    for (int i = 0; i < count; i++) {
        pipe(fd);
        pid_t pid = fork();
        if (pid == 0) {
            dup2(in_fd, 0);
            if (i < count - 1)
                dup2(fd[1], 1);
            close(fd[0]);

            char tmp[1024];
            strncpy(tmp, cmds[i], sizeof(tmp) - 1);
            tmp[sizeof(tmp) - 1] = '\0';

            char *args[MAX_ARG];
            parse_args(tmp, args);
            execvp(args[0], args);
            perror("execvp failed");
            exit(1);
        } else {
            wait(NULL);
            close(fd[1]);
            in_fd = fd[0];
        }
    }

    return 0;
}

int execute_logical(char *line) {
    char *copy = strdup(line);
    char *ptr = copy;
    int last_status = 0;

    while (*ptr) {
        char *cmd = ptr;
        char *next = NULL;
        int is_and = 0, is_or = 0;

        while (*ptr) {
```

```
            if (strncmp(ptr, "&&", 2) == 0) {
                is_and = 1;
                *ptr = '\0';
                next = ptr + 2;
                break;
            } else if (strncmp(ptr, "||", 2) == 0) {
                is_or = 1;
                *ptr = '\0';
                next = ptr + 2;
                break;
            }
            ptr++;
        }

        while (*cmd == ' ') cmd++;
        char *end = cmd + strlen(cmd) - 1;
        while (end > cmd && (*end == ' ' || *end == '\n')) {
            *end = '\0';
            end--;
        }

        if (*cmd) {
            last_status = execute_single(cmd);
        }

        if (is_and && last_status != 0) break;
        if (is_or && last_status == 0) break;

        if (!next) break;
        ptr = next;
    }

    free(copy);
    return last_status;
}

int execute_sequential(char *line) {
    char *copy = strdup(line);
    char *rest = copy;
    char *cmd;
    int status = 0;

    while ((cmd = strsep(&rest, ";"))) {
        while (*cmd == ' ') cmd++;
        char *end = cmd + strlen(cmd) - 1;
        while (end > cmd && (*end == ' ' || *end == '\n')) {
            *end = '\0';
            end--;
        }
        if (*cmd) {
            execute_command(cmd);
        }
    }

    free(copy);
```

```
    return status;
}

int execute_background(char *line) {
    char *cmd = strtok(line, "&");
    while (cmd) {
        pid_t pid = fork();
        if (pid == 0) {
            setsid();
            execute_single(cmd);
            exit(0);
        } else {
            printf("[Background pid %d]\n", pid);
        }
        cmd = strtok(NULL, "&");
    }
    return 0;
}
```

다양한 명령을 처리하는 함수들이 모인 c 파일로 builtin.c에서 처리하지 않는 다수의 명령어들을 처리하기 위한 목적입니다.

execute_single은 단일 명령어 처리를 위한 함수입니다. 인자로 받아온 cmd 를 바로 이용하면 cmd가 변경되어 다른 함수에서 이용되기 어려울 수 있기 때문에 이를 tmp에 strcpy 해서 안전하게 이용합니다. 이렇게 복사한 tmp에 > (리다이렉션) 기호가 있으면 >가 처음 나타나는 위치를 찾아 redir 포인터 변수에 저장합니다.(>가 포함되어 있으면 후에 if문이 작동됩니다) redirect 플래그를 0으로 초기화, 그리고 output으로 저장될 파일 포인터 변수 *outfile을 선언하고 초기화합니다. if 문은 리다이렉션을 처리하기 위한 전처리 부분입니다. 우선 >을 \0(null)로 바꿔서 >을 기준으로 문자열을 두 부분으로 나눕니다. > 다음 부분 부터는 저장될 파일명을 나타내기 때문에 redir ++ 후 공백이 없을 때 까지 while문을 통해 스킵합니다(공백이 여러개 있을 수 있음) 이후 문자가 나타나는 부분까지 이동했다면 strtok을 이용해 공백, 탭, 엔터를 기준으로 파일명이 되는 부분만 파싱해서 이것을 outfile로 넣어줍니다. 최종적으로 redirect =1로 설정함으로써 리다이렉 준비를 합니다. 이후 tmp를 parse_args를 이용해 파싱해줍니다. 이렇게 파싱한 문자열을 run_builtin(args)의 인자로 넣어주고 함수의 결과값을 builtin_result에 저장합니다. 여기서 builtin_result가 0, 1인 경우는 builtin 명령어가 맞다는 뜻이며 리턴됩니다. 이때 0과 1은 성공, 실패를 지시합니다. 만약 builtin_result가 -1면 내장 명령어가 아니기 때문에 리턴되지 않고 리다이렉션 과정이 수행됩니다. 리다이렉션을 위해 우선 fork로 자식 프로세스를 생성합니다. 이 자식 프로세스(pid==0)에서 redirect&&outfile을 확인해서 리다이렉션이 설정되어 있는지 확인하고 맞다면 open 함수를 이용해 파일 디스크립터 fd를 생성합니다. 이때 fd<0이면 파일 열기에 실패했다는 뜻으로 open failed 에러를 출력하며 종료됩니다. 열기에 성공한 경우 dup2 함수를 통해 표준 출력을 의미하는 STDOUT_FILENO 옵션을 지정해주어 파일로 출력이 들어갈 수 있게 해줍니다. 즉, 이 과정이 리다이렉션 과정입니다. 리다이렉셔닝이 완료되면 fd는 더이상 사용할 필요가 없으므로 file descriptor leak을 방지하기 위해 close처리 후 현재 프로세스를 execvp 함수를 통해 프로세스를 args[0]에 해당하는 명령어로 교체, 실행됩니다. execvp가 실패했을 경우 error 가 뜨며 에러처리되고 비정상 종료됩니다. else를 살펴보면 waitpid를 통해 부모 프로세스는 자식 프로세스가 종료될 때까지 기다립니다. WIFEXITED(status) ? WEXITSTATUS(status) : 1; 를 통해 자식이 정상 종료된 경우 종료 코드를 반환, 비정상 종료된 경우 1을 반환합니다.

execute_pipeline은 파이프라인 명령어를 처리하는 함수입니다. 우선 파이프라인에 연결된 명령어를 저장할 배열 cmds를 선언해주고 strtok을 통해 |을 기준으로 라인을 나눕니다. 이후 while문을 돌면서 토크나이즈를 계속 해주고 분리된 토큰들을 cmds에 저장해줍니다. 즉 strtok(NULL,"|")는 다음 명령어를 계속 추출하면서 멀티 파이프라인을 처리할 수 있도록 해주었습니다. cmd 마지막에는 NULL을 추가해줍니다. 예를 들면, a|b|c인 경우 cmds에는 a, b, c, NULL이 들어가는 것입니다. count는 a,b,c 세 가지의 명령어를 처리해야 하기 때문에 3입니다. 이제 파이프라인을 처리하기 위해 in_fd와 fd[2]를 선언해줍니다. in_fd는 이전 명령어의 파이프 읽기 쪽 디스크립터를 의미하며 현재 명령어의 입력으로 처리됩니다. fd[2]는 pipe 함수에서 이용될 read/write 디스크립터를 의미합니다. 이전에 토크나이즈 할때 count 값을 이용하며 for문을 돌 동안 각 명령어마다 새로운 pipe를 생성하고 fork로 자식 프로세스를 생성해줍니다. 자식 프로세스에서는 dup2(in_fd,0)을 통해 현재 명령어의 입력이 이전 파이프의 읽기라고 처리해줍니다. i<count-1 이면, 즉 현재의 명령어가 마지막이 아니라면, 이전 명령어의 출력은 다음 명령어의 입력으로 동작하므로 dup2(fd[1],1)을 통해 표준 출력을 새 파이프로 보낼수 있도록 처리해줍니다. 그러면 fd[0]은 사용하지 않으므로 이 디스크립터를 close 처리를 해줍니다. 파이프라인 처리가 끝나면 본격적으로 명령어를 실행합니다. 인자로 받아온 cmds가 변경되지 않도록 strncpy를 이용해 tmp를 이용해줍니다. 그리고 parse_args를 수행해 args 배열을 생성합니다. args[0]이 의미하는 명령어를

execvp 함수를 통해 교체, 수행합니다. 여기서도 마찬가지로 에러가 뜰 경우 에러 처리되고 비정상 종료 됩니다. 부모 프로세스의 경우 자식 프로세스가 끝날때까지 wait을 호출하 기다리게 되며 close(fd[1]) 라인을 통해 파이프 쓰기 쪽을 닫고 in_fd=fd[0]으로 지정해주면서 다음 명령어가 출력을 읽기로 처리할 수 있도록파이프를 계속 이어주는 역할을 수행하게 됩니다.

execute_logical 함수는 &&, || 명령어를 처리하는 함수입니다, 여기서 핵심은 &&은 앞 명령어가 성공했을 때 뒤의 명령어가 실행, ||은 앞 명령어가 실패했을 때 뒤 명령어가 실행되는 것입니다. 인자로 받은 line을 안전하게 이용할 수 있도록 strdup 함수를 통해 복사본 copy를 생성합니다. 그리고 문자열을 순회할 포인터 ptr을 선언해주고 각 명령어의 실행 결과 상태를 지정해줄 last_status를 선언해줍니다. ptr 포인터는 &&, ||을 찾아 이를 기준으로 명령어를 나누는데 사용되고, last_status는 명령어의 결과 상태를 지시함으로써 다음 명령어가 실행될지 말지 판단하는 것에 사용됩니다. 우선 copy의 시작부분부터 while문을 돌며 cmd, next, is_and, is_or 등의 변수를 선언해줍니다. ptr은 다시 while문을 돌면서 strncmp 함수를 통해 &&이 있는지 찾고, 있을 경우 is_and=1로 지정해줍니다. 그리고 해당 위치를 \0 처리 하고 그 다음에 처리할 명령어가 있는 포인터를 ptr+2로 지정해줍니다 (&&, || 은 두칸). 혹은 ||이 있을 경우 is_or을 1로 처리, 해당 위치를 \0로 처리하고 다음에 실행할 명령어가 있는 위치를 next=ptr+2로 지정해줍니다. 현재 위치에서 &&, ||가 없으면 ptr++을 해주며 다음 문자로 이동합니다. &&, ||을 기준으로 명령어를 찾는 과정이 끝났다면 while (*cmd == ' ') cmd++; 라인을 통해 앞쪽 공백을 처리합니다. end는 현재 명령어의 마지막 문자 위치를 가리키게 되는데 end 선언 후 while문을 통해 뒤에서부터 루프를 돌면서 공백, 엔터를 제거하면서 앞으로 이동합니다. 이 과정은 문자열 앞, 뒤의 불필요 공백을 제거함으로써 커맨드 처리과정의 오류를 방지하는 역할을 합니다. 공백 처리가 끝났다면 execute_single(cmd)를 통해 해당 명령어를 실행하고 그 실행 결과를 last_status에 저장합니다. &&연산일 경우 앞 명령어가 성공해야 뒷 명령어가 실행될 수 있으므로 is_and&&last_status가 0이 아닐경우 break 처리합니다. ||의 경우 앞 명령어가 실패한 경우에 뒷 명령어가 실행되므로 is_or&&last_status==0인 경우 break 처리합니다. 명령어 처리가 끝나면 next로 계속 이동되며 다음 명령어를 처리하며 next=NULL로 더이상 처리할 명령어가 없으면 break됩니다. 모든 실행이 끝나면 더이상 copy를 사용할 필요가 없으므로 free를 통해 공간 할당을 해제해줌으로써 memory leak를 방지합니다. 최종적으로 last_status를 리턴하면서 함수는 종료됩니다.

execute_sequential은 ;로 연결된 커맨드라인을 처리하기 위한 함수입니다. 원본 line을 안전하게 하기 위해서 마찬가지로 strdup 함수를 이용해 copy를 이용합니다. 그리고 copy문자열을 순회할 rest 포인터 변수를 선언해주고 ;로 구분된 명령어들을 저장해줄 포인터 변수 cmd를 선언해줍니다. 그리고 상태 변수 status를 선언해줍니다. while문을 돌며 strsep함수를 이용해 rest 문자열에서 ;을 기준으로 앞에서부터 하나씩 명령어를 분리, 이를 cmd에 저장하는 과정을 거칩니다. 이후 execute_logical에서처럼 명령어의 앞뒤 공백을 제거해줍니다. 분리된 명령어 cmd를 execute_command를 통해 실행하고 free를 통해 copy에 할당된 메모리를 반환시켜줍니다. 최종적으로 status를 리턴하면서 함수는 종료됩니다.

execute_background는 백그라운드 명령어를 실행하기 위한 함수입니다. 우선 strtok을 이용해 &을 기준으로 line을 나누고 이를 포인터 변수 cmd에 저장해줍니다. sleep 5 & ls & 같이 여러개의 백그라운드 명령어가 실행될 것을 대비한 것입니다. while(cmd), 즉 포인터 변수 cmd에 명령어가 있으면 fork를 통해 자식 프로세스를 생성합니다. 자식프로세스에서는 setsid() 함수로 새 세션을 생성하고 부모 프로세스와 자식 프로세스를 분리시킵니다. 즉 분리의 과정이 백그라운드 프로세스에서도 쉘을 이용할 수 있게 하는 핵심이라고 볼 수 있습니다. 이후 execute_single(cmd)로 명령어를 실행하고 작업이 끝나면 exit(0)을 통해 프로세스가 종료될 수 있게 합니다. 부모프로세스에서는 wait 계열 함수가 없는 것을 확인할 수 있는데, 이는 부모가 자식이 실행되는 동안 기다리지 않고 있다는 것을 의미합니다. (이때 좀비 프로세스 등의 문제가 발생할 수 있으므로 prompt.c에서 정의한 sigchld_handler가 이용됩니다.)대신 백그라운드에서 실행되는 pid의 메시지를 출력하는 역할을 합니다. while문은 cmd=strtok(NULL, “&”)으로 다음 & 구분 명령어로 이동하면서 cmd에 더이상 실행할 명령어가 없을 때까지 while문이 반복됩니다. while문이 끝난 후 함수는 0을 리턴하면서 종료됩니다.

참고자료 ;

https://ko.wikipedia.org/wiki/C_문자열_처리

https://eastc.tistory.com/entry/C-Linux-open-함수

https://reakwon.tistory.com/104

https://eastc.tistory.com/entry/execl3-execv3-execle3-execve2-execlp3-execvp3

https://blog.naver.com/endfirst/20018509790

main.c

```
#include "shell.h"

int main() {
    char prompt[1024];
    char *line;

    signal(SIGCHLD, sigchld_handler);

    history_setup();

    while (1) {
        set_prompt(prompt);
        line = readline(prompt);

        if (!line) break;
        if (*line) add_history(line);

        execute_command(line);
        free(line);
    }

    history_finish();
    return 0;
}
```

main은 프로그램의 시작점이 되는 부분으로, 사용자의 입력을 받고 히스토리를 처리하는 등의 역할을 합니다. 우선 프롬프트가 들어갈 배열 prompt를 선언하고 사용자로부터 입력받을 line 포인터 변수를 선언해줍니다. signal 함수를 통해 백그라운드로 실행된 자식 프로세스가 정리될 수 있도록 핸들러를 설정해 좀비 프로세스를 방지합니다. (자식 종료 대기 없이 리소스만 정리) 이후 history_setup()을 통해 히스토리 기능을 설정해줍니다. 쉘은 exit 입력 전까지 계속해서 사용자의 입력을 받으며 커맨드를 처리해야하므로 while(1)로 무한 반복 처리를 해줍니다. 이 반복동안 set_prompt로 쉘에 표시될 프롬프트 문자열을 생성해주고 readline 함수를 통해 프롬프트를 띄우고 사용자 입력을 받아 이걸을 line 포인터 변수에 저장해줍니다. 이때 line이 NULL일 경우 break되어 종료되며, line이 NULL이 아니면 add_history 함수를 통해 히스토리에 추가하고 해당 라인을 execute_command함수에 인자로 넣어 커맨드를 처리해줍니다. 커맨드 처리가 끝나면 free를 통해 해당 라인 공간을 할당 해제해줍니다. 이 과정을 계속 반복합니다. 반복하다가 exit나 break 등으로 쉘이 종료되는 상황에 놓이면 history_finish 함수가 실행되어 명령어 히스토리를 저장하는 마무리 과정을 거치고 최종적으로 종료됩니다.

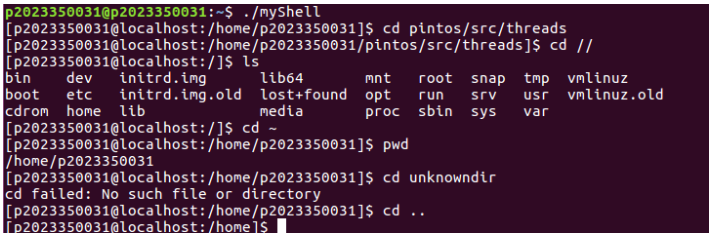
전체적인 내용을 크게 설명하자면

main에서 라인 받음 → 라인이 적절하면 execute_command → 여기서 어떤 커맨드인지 확인 후 함수 지정 → shellfunct.c에 있는 함수에서 커맨드 처리(이 과정에서 builtin.c, prompt.c 등에 있는 다양한 함수 이용) → 처리 완료후 main에서 공간 할당 해제 → 다시 커맨드 입력받음 ..

위와 같은 과정을 반복하게 됩니다../ㅡ

Test

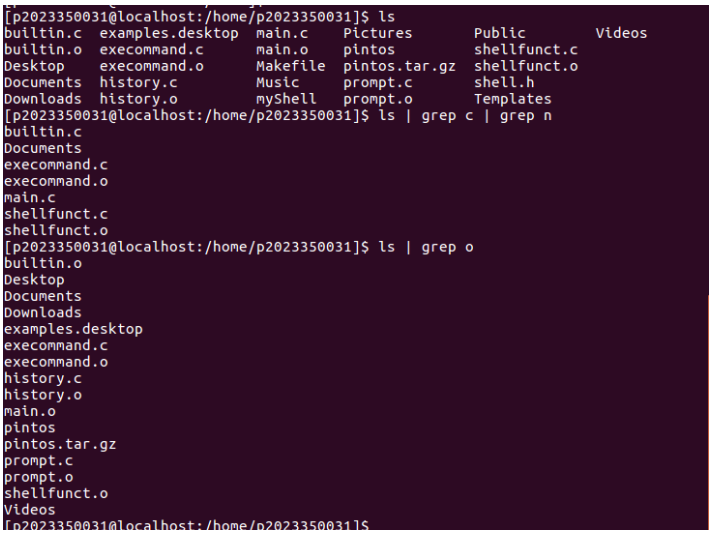
리눅스 가상머신에서 테스트한 모습입니다



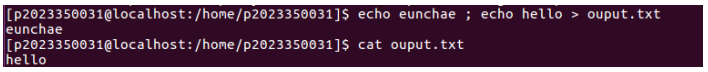
실행을 하게 되면 username, host, 그리고 directory path가 뜨는 것을 확인할 수 있습니다.

cd 명령어를 이용해서 절대경로로 이동하는 것이 가능함은 물론 cd .. 등으로 상대 경로 사용도 가능 한 것을 확인할 수 있습니다. cd ~ 로 home으로 이동하는 것 또한 확인이 가능합니다

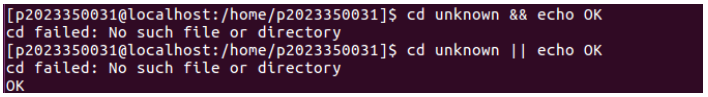
pwd로 현재 위치한 경로를 확인할 수 있으며 없는 디렉토리로 이동하려 할때는 cd failed가 뜨는 것도 확인할 수 있습니다.



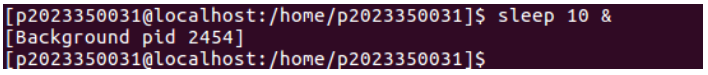
다음으로는 파이프라인과 멀티 파이프라인을 아주 간단하게 테스해본 모습입니다. 해당 경로의 디렉토리에는 여러 파일들이 들어있습니다 ls | grep c | grep n 명령어를 통해 파일명에 c,n이 들어간 것들이 잘 출력되는 것을 확인할 수 있습니다. ls | grep o 명령어도 잘 작동하는 것을 확인할 수 있습니다. 멀티 파이프라인과 파이프라인이 잘 동작되는 모습입니다.



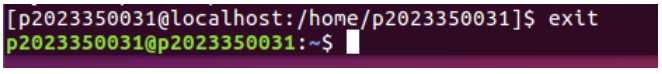
마찬가지로 간단한 예제를 통해 다중명령어와 리다이렉트를 확인해보겠습니다. ; 전 echo 명령어가 잘 실행된 모습을 확인했습니다. 이후 cat을 통해 생성된 ouput.txt의 내용을 확인해보니 마찬가지로 ; 이후의 리다이렉트 명령어도 잘 실행된 것이 보입니다.



다중명령어 &&과 ||입니다. unknown이라는 없는 폴더로 이동하려고 할 때를 가정해보고 명령어를 입력해본 모습입니다. &&의 경우 이동이 실패했기 때문에 echo OK가 실행되지 않는 모습입니다. 반면에 ||의 경우 unknown으로 이동되지 않았기에 뒤에 있는 echo OK가 실행된 모습이 보입니다.



sleep 10 & 명령어를 사용해 백그라운드를 실행해본 모습입니다. 백그라운드로 실행된 프로세스의 pid가 출력이 되고, 백그라운드 실행 중에도 커맨드를 입력할 수 있습니다.



exit를 입력하면 실행중이던 myShell에서 빠져나와 원래의 bash Shell로 이동할 수 있습니다.