2025_CyKor_Week2

1. 쉘 명령어 처리 구조

Shell을 만들 때 라인을 입력 받아 명령어를 나누는 과정을 함수 별로 나누었다.

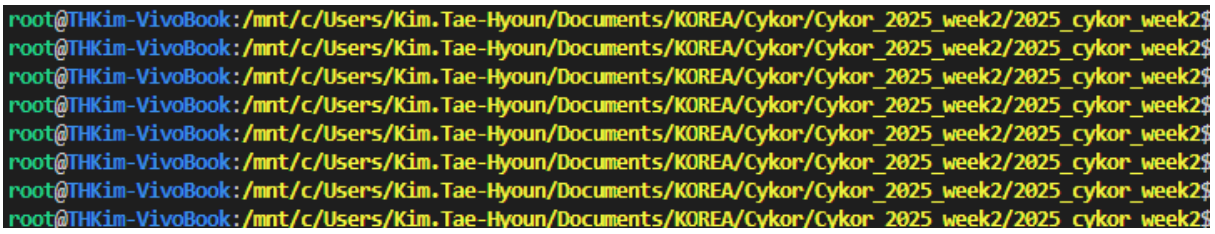
먼저 명령어를 나누는 기준은 아예 다른 명령이라 할 수 있는 ; 를 기준으로 스플릿 하고 명령어들을 이어주는 &&, ||, &를 기준으로 스플릿 한 뒤, 다른 함수에서 단순 명령어를 실행시키거나 파이프라인이 있다면 명령어로 차례대로 나누어 처리하였다.

1-1. main

```
#define MAX_CMD_LEN 1024

void print_directory() {
    char cwd[1024];
    char hostname[1024];
    struct passwd *pw = getpwuid(getuid());
    char *username = pw ? pw→pw_name : "unknown";
    getcwd(cwd, sizeof(cwd));
    gethostname(hostname, sizeof(hostname));

    printf("\033[1;32m%s\033[0m@\033[1;34m%s\033[0m:\033[1;33m%s\033
}

int main() {
        signal(SIGCHLD, sigchld_handler);
    char line[MAX_CMD_LEN];

    while (1) {
        print_directory();
        memset(line, 0, sizeof(line));
        if (!fgets(line, sizeof(line) - 1, stdin)) break;
        line[strcspn(line, "\n")] = 0;
        if (strncmp(line, "exit", 4) == 0) break;
        char *saveptr;
        char *cmd = strtok_r(line, ";", &saveptr);
        while (cmd) {
```

```
            parse_cmd(cmd);
            cmd = strtok_r(NULL, ";", &saveptr);
        }
    }
    return 0;
}
```

main 에서는 exit을 입력하면 나가게 하거나 ;을 기준으로 나누어 parse_cmd로 보낸다. 리눅스 쉘에서는 ; 은 다른 줄, 다른 명령이나 다름없기 때문에 가장 먼저 명령어를 나누는 기준으로 채택하였다. 또한 한 라인의 최대 문자열 수를 1024로 정하고 fgets를 size - 1 만큼 입력 받아 \n을 포함하여 오버플로우가 일어나지 않도록 설계하였으며 매 라인을 받을 때마다 memset을 통하여 line을 초기화하여 이전 명령어들이 유출되거나 악용되는 사고를 막고자 하였다. 또한 라인을 받을 때마다 print_directory가 실행되게 하여 gethostname과 getcwd로 현재 사용자와 현재 실행 디렉토리를 가져와 출력하였다.



1-2. parse_cmd

```
typedef enum { CMD_NORMAL, CMD_AND, CMD_OR, CMD_BG } OpType;

typedef struct {
    char* cmd;
    OpType op_type;
    struct op* next;
} op;

void parse_cmd(char* cmd) {
    op *head = NULL;
    op *tail = NULL;
    int len = strlen(cmd);
    int ret_val = 0;

    for (int i = 0; i < len;){
```

```
    int j = i;
    OpType type = CMD_NORMAL;

    while (j < len){
        if (j + 1 < len && cmd[j] == '&' && cmd[j + 1] == '&') {
            type = CMD_AND;
            break;
        }
        if (j + 1 < len && cmd[j] == '|' && cmd[j + 1] == '|') {
            type = CMD_OR;
            break;
        }
        if (cmd[j] == '&') {
            type = CMD_BG;
            break;
        }
        j++;
    }
    int cmd_len = j - i;
    while (cmd_len > 0 && cmd[i + cmd_len - 1] == ' ') cmd_len--;
    while (cmd_len > 0 && cmd[i] == ' ') { i++; cmd_len--; }

    char *cm = strndup(cmd + i, cmd_len);
    op *node = malloc(sizeof(op));

    node->cmd = cm;
    node->op_type = type;
    node->next = NULL;

    if (tail) tail->next = node;
    else head = node;
    tail = node;

    if (type == CMD_AND || type == CMD_OR) i = j + 2;
    else if (type == CMD_BG) i = j + 1;
    else break;
}
```

```
    int keep = 1;
    for (op* iter = head; iter != NULL; iter = iter→next){
        if (keep) ret_val = execute_subcmd(iter→cmd, iter→op_type==CMD_BG);
        if (keep && iter→op_type == CMD_AND && ret_val != 0) keep = 0;
        else if (keep &&iter→op_type == CMD_OR && ret_val == 0) keep = 0;
        free(iter→cmd);
        free(iter);
    }
}
```

parse_cmd에서는 main에서 나눈 명령어를 &&, ||, &를 기준으로 나눈 subcmd를 linked list로 만들어 execute_subcmd에서 차례대로 실행시키는 역할을 한다. 또한 만약 &로 실행된 명령어라면 background로 실행됨을 나타내는 인자를 전달한다. 이때 &&일 경우 명령어가 성공해야 다음 명령어가 실행되고 || 일 경우 명령어가 실패하면 다음명령어가 실행되고 &는 상관없이 실행되기 때문에 연산자를 저장하기 위해 linked list 자료구조를 사용하였다.

이를 위해 명령어를 나타내는 enum과 명령어 문자열, 명령어 종류, 다음 명령어 구조체의 포인터를 속성으로 가지는 명령어 구조체를 선언하여 사용하였다.

linked list 자료구조를 사용할 때 명령어 구조체 자체와 subcmd의 문자열 공간을 동적으로 할당 받았다. (malloc, strndup) 따라 free하여야 메모리를 올바르게 관리할 수 있다. 처음에는 한 명령어를 실행시키며 free를 돌도록 하였지만 for 문이 끝나고 free된 iter의 next 메모리에 접근하는 것은 UAF 취약점에 해당한다. 하지만 이를 해결하기 위해서는 모든 노드의 주소를 저장하고 이후에 free 하여야 하고, free된 iter의 next의 접근하는 것에 대하여 보안 취약점이 없어보여 그대로 쓰기로 하였다.

for 문에 있는 조건문들은 &&와 ||를 관리하기 위해 정상적으로 실행이 된다면 0이 리턴되고 아니라면 0이 아닌 값이 리턴되는 것을 이용하여 keep의 값으로 다음 명령어들을 실행 여부를 결정하게 하였다. 또한 중간에 명령어가 멈추더라도 끝까지 for문을 돌게 만들어 free되지 않은 동적 메모리가 없도록 하였다.

또한 명령어의 크기를 제한하여 사용하는 것이 아닌 동적으로 할당 받았기 때문에 오버플로우나 범용성 문제에 관하여는 안전해보인다.

1-3. execute_subcmd

```
int execute_subcmd(char *subcmd, int is_bg) {
    int ret = 0;

    if (!strchr(subcmd, '|')) {
```

```
    char *args[64];
    int i = 0;
    char *token = strtok(subcmd, " ");
    while (token && i < 63) {
        args[i++] = token;
        token = strtok(NULL, " ");
    }
    args[i] = NULL;
    if (args[0] == NULL) return 0;

    if (run_builtin(args, &ret) == 0) return ret;

    pid_t pid = fork();
    if (pid == 0) {
        execvp(args[0], args);
        perror("execvp");
        exit(1);
    } else if (pid > 0) {
        if (is_bg) return 0;
        int status;
        waitpid(pid, &status, 0);
        return WEXITSTATUS(status);
    } else {
        perror("fork");
        return 1;
    }
}

if (is_bg){
    pid_t pid_bg = fork();
    if (pid_bg > 0) return 0;
    else if (pid_bg == -1) {
        perror("fork");
        return 1;
    }
}

char *cmds[16];
```

```
    int count = 0;
    char *saveptr;
    char *part = strtok_r(subcmd, "|", &saveptr);
    while (part && count < 15) {
        while (*part == ' ') part++;
        char *end = part + strlen(part) - 1;
        while (end > part && *end == ' ') *end-- = '\0';
        cmds[count++] = part;
        part = strtok_r(NULL, "|", &saveptr);
    }
    cmds[count] = NULL;

    int prev_fd = -1;
    for (int i = 0; i < count; i++) {
        int pipefd[2];
        if (i < count - 1) pipe(pipefd);

        pid_t pid = fork();
        if (pid == 0) {
            if (prev_fd != -1) {
                dup2(prev_fd, 0);
                close(prev_fd);
            }
            if (i < count - 1) {
                dup2(pipefd[1], 1);
                close(pipefd[0]);
                close(pipefd[1]);
            }

            char *args[64];
            int j = 0;
            char *tok = strtok(cmds[i], " ");
            while (tok && j < 63) {
                args[j++] = tok;
                tok = strtok(NULL, " ");
            }
            args[j] = NULL;
            if (args[0] == NULL) return 1;
```

```
            int ret_val;
            if (run_builtin(args, &ret_val) == 0) exit(1);
            execvp(args[0], args);
            perror("execvp");
            exit(1);

        } else {
            if (prev_fd != -1) close(prev_fd);
            if (i < count - 1) {
                close(pipefd[1]);
                prev_fd = pipefd[0];
            }
            waitpid(pid, NULL, 0);
        }
    }
    return 0;
}
```

execute_subcmd에서는 파이프라인이나 단순 명령어를 실행시키는 역할을 한다. 먼저 strchr을 활용해서 | 가 없다면 옵션(최대 64개) 처리를 위해 띄어쓰기를 기준으로 잘라 builtin된 함수에 있는지 확인하고 없다면 fork와 execvp를 통해 실행한다. execvp는 PATH를 활용하기 때문에 우리가 자주 사용하는 명령어인 ls를 실행시킬 때 /bin/ls를 입력하지 않아도 정상적인 실행이 되고 환경변수 알아서 찾는 이점이 있어 execvp를 사용하였다. 또한 &를 활용한 백그라운드 실행일 경우에 pid > 0, 즉 부모 프로세스에서 즉시 정상 실행이 된것으로 판단하여 return하게 하였다. 백그라운드가 아닌 경우에도 wexitstatus를 활용하여 리턴값을 얻어와 다중 명령어를 처리하게 하였다.

파이프라인이 있는 경우에는 가장 먼저 백그라운드라면 그냥 fork하고 리턴하여 기다리지 않고 백그라운드에서 파이프라인이 실행되도록 하였고, 이후 자식 프로세스가 파이프라인을 처리하도록 하였다.

파이프라인의 경우 최대 파이프라인 명령어를 16개로 하였고 | 를 기준으로 나누어 배열에 저장해둔다. 그리고 fork를 사용하여 계속 자식 프로세스에서 명령어를 실행하고 부모 프로세스에서 연속적으로 실행하는 것을 명령어가 없어질 때까지 반복한다. 또한 waitpid와 dup2를 활용하여 표준 입출력 0, 1을 파이프로 바꾸어 한 명령어를 처리하고 prev_fd를 사용해 결과값을 파이프를 통해 기억한 뒤, 다음 멍령어를 처리할 때 prev_fd(이전 pipe[0])를 표준 입력으로 바꾸고 pipe[1]을 표준 출력으로 바꾸어 실행될 수 있도록 하여 연속적인 프로세스 간 통신이 가능하게 하였다.

모두 perror를 사용하므로써 에러가 났을 때 에러에 관한 메세지를 출력할 수 있도록 하였다.

```
execvp(args[0], args);
perror("execvp");
exit(1);
```

또한 execvp 가 실행되면 완전히 프로세스가 args[0]의 것으로 치환되게 되는데 이때 execvp 함수가 실패할 수 있다. 그리고 그냥 ret 하게 되면 이전에 사용한 동적 메모리가 COW에 의해 새로운 메모리로 복제되고 이는 메모리 낭비로 이어질 수 있다. 따라서 실패할 경우 perror와 exit을 활용하여 바로 빠져나오도록 했다.

2. 명령어 구현

2-1. 실행 코드

```
typedef struct {
    const char *name;
    int (*func)(char **args);
} cmds;

cmds *command_list[] = {
    &(cmds){ "cd", run_cd },
    &(cmds){ "pwd", run_pwd },
    NULL
};

int run_builtin(char **args, int *ret) {
    for (int i = 0; command_list[i] != NULL; i++) {
        if (strcmp(args[0], command_list[i]→name) == 0) {
            *ret = command_list[i]→func(args);
            return 0;
        }
    }
    return -1;
}
```

먼저 함수 이름과 함수 주소를 가진 구조체를 만들었다. 함수 포인터로 선언하여 추후 다른 명령어를 사용하더라도 command_list에 추가만 하면 된다. 이를 위해 run_builtin이라는 함수에서 command_list에 있는지 없는지 확인하여 있다면 실행하여 정상 종료가 되었는지 따로 코딩해둔 리턴 값을 반환하고 없다면 돌아가 exec 함수로 실행되게 된다.

2-2. cd

```
// cd.h
#pragma once
int run_cd(char **args);

// cd.c
#include "cd.h"
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <errno.h>

int run_cd(char **args) {
    int L = 1;
    char *target = NULL;

    int i = 1;
    while (args[i] && args[i][0] == '-') {
        if (strcmp(args[i], "-L") == 0) {
            L = 1;
        } else if (strcmp(args[i], "-P") == 0) {
            L = 0;
        } else {
            fprintf(stderr, "cd: invalid option %s\n", args[i]);
            return 1;
        }
        i++;
    }

    if (args[i] == NULL) {
```

```
        target = getenv("HOME");
    } else if (args[i][0] == '~') {
        char path[PATH_MAX];
        snprintf(path, sizeof(path), "%s%s", getenv("HOME"), args[i] + 1);
        target = path;
    } else {
        target = args[i];
    }

    if (!L) {
        char real_target[PATH_MAX];
        if (realpath(target, real_target) == NULL) {
            perror("cd -P");
            return 1;
        }
        target = real_target;
    }

    if (chdir(target) != 0) {
        perror("cd");
        return 1;
    }

    char new_pwd[PATH_MAX];
    if (getcwd(new_pwd, sizeof(new_pwd)) != NULL) {
        setenv("PWD", new_pwd, 1);
    }
    return 0;
}
```

cd 구현코드이다.

```
Options:
  -L        force symbolic links to be followed: resolve symbolic
            links in DIR after processing instances of `..'
  -P        use the physical directory structure without following
            symbolic links: resolve symbolic links in DIR before
            processing instances of `..'
  -e        if the -P option is supplied, and the current working
            directory cannot be determined successfully, exit with
            a non-zero status
  -@        on systems that support it, present a file with extended
            attributes as a directory containing the file attributes
```

help cd 를 쳐서 나온 옵션을 참고하였는데 e, @ 옵션은 구현하지 않았다.

또한 구현하고보니 물리적, 논리적 심볼릭 링크에 관한 것으로 realpath라는 함수로 간단히 구현되었다.

cd를 구현하며 ~ (home) 기능을 구현하였다. ~가 앞에 있다면 환경변수에서 home의 위치를 가져와 이를 ~가 있던 곳에 넣어 디렉토리를 변경하도록 하였다. 또한 마지막에는 pwd도 현재 위치로 바꿔주어 쉘 안에서 현재 디렉토리에서 바로 작업을 할 수 있도록 하였다.

2-3. pwd

```
// pwd.h
#pragma once
int run_pwd(char **args);

// pwd.c
#include "pwd.h"
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int run_pwd(char **args) {
    int L = 1;

    if (args[1] != NULL) {
        if (strcmp(args[1], "-P") == 0) {
            L = 0;
        } else if (strcmp(args[1], "-L") != 0) {
            fprintf(stderr, "pwd: invalid option %s\n", args[1]);
```

```
            return 1;
        }
    }

    if (L) {
        char *pwd = getenv("PWD");
        if (pwd){
            printf("%s\n", pwd);
            return 0;
        }
        else{
            perror("PWD");
            return 1;
        }
    } else {
        char cwd[1024];
        if (getcwd(cwd, sizeof(cwd)) != NULL){
            printf("%s\n", cwd);
            return 0;
        }
        else{
            perror("getcwd");
            return 1;
        }
    }
}
```

pwd 구현코드이다.

```
Options:
  -L        print the value of $PWD if it names the current working
            directory
  -P        print the physical directory, without any symbolic links

By default, `pwd' behaves as if `-L' were specified.
```

옵션처리를 위해 L 옵션에서는 pwd의 값을 가져오는 getenv를 사용하였고 P 옵션에서는 물리적 경로를 가져오기 위해 getcwd를 사용하였다.

또한 cd, pwd 모두 exit 같은 에러시 즉시 종료를 사용하지 않고 반환값을 사용해서 성공, 실패를 직접 분리할 수 있도록 하였다.

3. Signal Handler

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

int main() {
    signal(SIGCHLD, sigchld_handler);
    ...
}
```

위에서 exit을 시켰더라도 부모 프로세스가 wait이나 waitpid를 호출하기 전까지 자식 프로세스의 자원은 회수되지만 프로세스 목록에 계속 남아있다. 이를 좀비 프로세스라고 부른다. 따라서 자식 프로세스가 종료되면 부모 프로세스에게 SIGCHLD를 보내는 것을 활용해서 시그널이 오면 waitpid를 호출하게 하였다. while을 활용하여 구현한 이유는 쉘에서 사용하는 보통의 명령어는 굉장히 빠른 시간내에 종료가 되기 때문이 여러개의 자식 프로세스가 종료되었더라도 시그널이 한번만 올 수 있다. 따라서 while을 통해 waitpid를 호출하였고, 인자로 -1, null, wnohang을 주어 어떤 자식 프로세스든 상태값은 상관없이 즉시 return을 하도록 하였다.

그런데 main에서 백그라운드 실행이 아닐 시, 직접 waitpid로 기다리게 되는 부분이 있는데 이것이 핸들러와 충돌하지 않을까 생각이 들었다. 하지만 부모 프로세스가 직접 waitpid로 기다리고 있다면 먼저 수거되고 나중에 핸들러가 실행되어 waitpid의 리턴값이 -1이 되어 충돌하지 않는다.

4. Makefile

```
CC = gcc
OBJ = main.o cd.o pwd.o
TARGET = shell

all: $(TARGET)

$(TARGET): $(OBJ)
```

```
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) $(TARGET)
```

makefile은 확장성을 위해 명령어 파일만 입력하면 가능하도록 하였다.