Cykor Week2 보고서

2025350215 사이버국방학과 김용성

개요

본 과제에서는 리눅스 셸의 핵심 메커니즘을 가상 파일 시스템 위에 재현함으로써, 운영체제의 프로세스 제어, I/O 리다이렉션, 문자열 파싱, 신호 처리(signal handling) 등 주요 개념을 직접 구현해 보았다. 특히 파이프(|)와 논리 연산자(&&, ||), 명령 시퀀스(;)뿐 아니라 백그라운드 실행(&)까지 지원하여, 실제 셸에서 사용하는 복합 문법을 최소 기능 범위 내에서 모두 다뤘다. 이로써 "쉘이 어떻게 명령을 분리·해석하고, 언제 프로세스를 생성하며, 언제 부모가 기다리고, 언제 즉시 리턴하는지"를 체득할 수 있었다.

시스템 구조 및 흐름

1. 프롬프트 생성:

   main() 루프 진입 시 현재 디렉토리(current_dir)를 검사하고, 홈 디렉토리일 때는 [%s@%s:~]$, 그 외에는 전체 경로(/home/kaleido/etc 등)를 표시하도록 구현.입력 수신: fgets()로 한 줄 입력을 받아 끝의 ₩n 제거

   이 과정에서 pwd() 재귀 호출을 통해 루트부터 현재 노드까지 트리 순회를 수행하므로, 깊이가 깊어져도 올바르게 경로를 구성할 수 있다.

2. 백그라운드 감지:

   입력 문자열 끝에 &가 한 개 이상 붙어 있는지 검사하여 bg 플래그 설정.

   백그라운드(bg = 1)인 경우, 별도의 자식 프로세스를 fork()로 띄워 run_logical_chain()을 실행시키고, 부모는 continue하여 즉시 새 프롬프트 출력.

3. 시퀀스 분리:

   split_sequence()에서는 strtok(..., ";")로 명령 블록을 나누고, 각 토큰 앞뒤의 공백을 제거.

   이 단계를 거치면 "ls ; pwd"처럼 띄어쓰기 유무에 상관없이 정확히 2개의 명령 블록으로 분리된다.

4. 논리 연산 처리:

   split_logical()에서 && 또는 || 위치를 탐색하여, 해당 위치에 '₩0'을 삽입하고 명

령(cmd)과 연산(op)을 Logical 배열에 저장.

run_logical_chain()에서는 이전 명령의 종료 코드(prev)가 0인지/아닌지를 기준으로 &&(성공 시에만 다음 실행), ||(실패 시에만 다음 실행) 로직을 적절히 건너뛴다.

5. 파이프라인 처리: 각 논리 체인의 단일 명령이 |을 포함하면 execute_pipe()에서 다중 프로세스, pipe(), dup2()로 표준 입출력 연결
6. 명령 실행: 파이프가 없으면 execute_command()로 내장 명령(cd, ls, pwd) 또는 "not found" 처리
7. 자원 정리: wait() 또는 SIGCHLD 무시로 좀비 방지 + 프로그램 종료 시 동적 할당된 가상 파일 시스템 메모리 해제

```c
typedef struct Directory{
    char name[MAX_SIZE];
    struct Directory *parent;
    struct Directory *subdirectories[10];
    int subdirectory_count;
} Directory;

Directory* create_directory(char *name, Directory*parent){
    Directory *new_dir = (Directory*)malloc(sizeof(Directory));
    strcpy(new_dir->name,name);
    new_dir->parent = parent;
    new_dir->subdirectory_count = 0;
    if(parent != NULL){
        parent-> subdirectories[parent->subdirectory_count] = new_dir;
        parent-> subdirectory_count ++;
    }
    return new_dir;
}
```

Directory 트리 자료구조

Struct 문법을 이용하여 directory 자료를 구현

char name[MAX_SIZE] : 디렉토리 이름

Directory *parent : 부모 노드 포인터 (최상위는 NULL)

Directory *subdirectories[10] : 최대 10개의 하위 디렉토리 포인터 배열

int subdirectory_count : 현재 몇 개가 사용 중인지 카운트

단, 하위 디렉토리 수 고정(10개) 제약이 있으며, 동적으로 더 늘리려면 malloc 기반 리스트 구조 활용이 필요하다.

```
34
35
36  void split_sequence(char *input, char **seq) {
37      char *tok = strtok(input, ";");
38      int i = 0;
39      while (tok) {
40          while (*tok==' '){
41              tok++;
42          }
43          char *end = tok + strlen(tok)-1;
44          while (end>tok && *end==' '){
45              *end--='\0';
46          }
47          seq[i++] = tok;
48          tok = strtok(NULL, ";");
49      }
50      seq[i] = NULL;
51  }
52
```

입력 파싱 모듈 시퀀스 분리 (split_sequence)

- ;로 명령 블록 나누고 앞뒤 공백 제거 논리 연산 분리 (split_logical)
- &&, || 위치에서 문자열 자르고 Logical(cmd, op) 배열에 저장 파이프 분리 (split_pipeline)
- |로 분리하고 공백 제거

```c
int execute_command(char* input , Directory **current_dir , Directory *root , Directory *home){
    char argument[MAX_SIZE] = {0,};
    char command[MAX_SIZE] = {0,};

    memset(argument , 0 , MAX_SIZE);
    sscanf(input, "%s %s" , command , argument);

    if(strcmp(command , "exit") == 0){
        exit(0);
    }
    else if(strcmp(command, "cd") == 0){
        return cd(current_dir,argument,root,home);
    }
    else if(strcmp(command, "ls") ==0){
        ls(*current_dir);
        return 0;
    }
    else if(strcmp(command , "pwd") == 0){
        pwd(*current_dir , 1);
        printf("\n");
        return 0;
    }
    else{
        printf("%s: command not found\n" , command);
        return 1;
    }

    return 0;
}
```

내장 명령어 처리 (execute_command)

- command와 argument를 sscanf로 분리
- exit, cd, ls, pwd 지원
- 지원되지 않는 명령은 "command not found" 출력 후 1 반환

```c
int execute_pipe(char *input, Directory **current_dir, Directory *root, Directory *home) {
    char *cmds[10];
    split_pipeline(input, cmds);

    int num_cmds = 0;
    while (cmds[num_cmds] != NULL) {
        num_cmds++;
    }

    int fd[10][2];
    int in_fd = STDIN_FILENO;

    for (int i = 0; i < num_cmds; i++) {
        if (i < num_cmds - 1) {
            if (pipe(fd[i]) == -1) {
                perror("pipe");
                exit(EXIT_FAILURE);
            }
        }

        pid_t pid = fork();
```

```c
        if (pid < 0) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        else if (pid == 0) {


            if (in_fd != STDIN_FILENO) {
                dup2(in_fd, STDIN_FILENO);
                close(in_fd);
            }

            if (i < num_cmds - 1) {
                dup2(fd[i][1], STDOUT_FILENO);
                close(fd[i][0]);
                close(fd[i][1]);
            }

            for (int j = 0; j < i; j++) {
                close(fd[j][0]);
                close(fd[j][1]);
            }

            char *argv[MAX_SIZE / 2];
            int argc = 0;
            char *token = strtok(cmds[i], " ");
            while (token) {
                argv[argc++] = token;
                token = strtok(NULL, " ");
            }
            argv[argc] = NULL;
```

```
            if (!strcmp(argv[0],"cd")||!strcmp(argv[0],"ls")||!strcmp(argv[0],"pwd")) {
                execute_command(cmds[i], current_dir, root, home);
                exit(0);
            }

            execvp(argv[0], argv);
            perror("execvp");
            exit(EXIT_FAILURE);
        }
        else {

            if (in_fd != STDIN_FILENO) {
                close(in_fd);
            }

            if (i < num_cmds - 1) {
                close(fd[i][1]);
                in_fd = fd[i][0];
            }
        }
    }

    int status = 0;
    for(int i = 0 ; i < num_cmds; i++){
        int w;
        wait(&w);
        status = w;
    }

    if(WIFEXITED(status)){
        return WEXITSTATUS(status);
    }
    return 1;
}
```

파이프라인 처리 (execute_pipe)

- split_pipeline로 cmds 배열 생성
- n-1개의 파이프 생성
- 각 명령마다 fork
  - 자식: 이전 입력을 dup2로 STDIN, 다음에 dup2로 STDOUT, 불필요 파이프 닫기, 내장 명령 처리 후 exit
  - 부모: 이전 입력 닫고 다음 파이프 읽기 끝 저장
- 모든 자식 wait, 마지막 종료 코드 반환

```c
int run_logical_chain(char *line, Directory **cdp, Directory *root, Directory *home) {
    Logical chain[10];
    int n = split_logical(line, chain);
    int prev=0;
    for(int i=0; i<n && chain[i].cmd; i++){
        if (i > 0) {
            if (chain[i-1].op == OP_AND && prev != 0){
                continue;
            }
            if (chain[i-1].op == OP_OR  && prev == 0){
                continue;
            }
        }
        if (strchr(chain[i].cmd, '|')){
            prev = execute_pipe(chain[i].cmd, cdp, root, home);
        }
        else{
            prev = execute_command(chain[i].cmd, cdp, root, home);
        }
    }

    return prev;
}
```

논리 연산 체인 실행 (run_logical_chain)

- split_logical로 (cmd, op) 배열 생성
- 순차 실행, 이전 op와 이전 결과(prev)로 실행 여부 결정
- 파이프 포함 시 execute_pipe, 그 외 execute_command

```c
    int bg = 0;
    size_t len = strlen(input);
    if(len > 0 && input[len-1] == '&'){
        bg = 1;
        input[--len] = '\0';
        while(len > 0 && input[len -1] == '&'){
            input[--len] = '\0';
        }
    }
    if (bg) {
        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
        } else if (pid == 0) {
            run_logical_chain(input, &current_dir, root, home);
            exit(0);
        }
        continue;
    }
```

백그라운드 실행 &

- 메인 루프에서 & 감지 후 fork로 별도 실행
- SIGCHLD SIG_IGN으로 좀비 방지
- 자식 프로세스를 fork로 생성하여 명령 체인 실행, 부모는 즉시 프롬프트 복귀