2024350209 송민서

사이코 2주차 시스템해킹
쉘 프로그램 구현하기 보고서

쉘을 구현하라고 하셔서 처음에 mkdir부터 구현을 하고싶어 저걸 먼저 구현하기 시작했다. 하지만, mkdir은 디렉토리를 만드는 명령어이다. 그래서 구현하면, 계속 내가 코드를 테스트하면서 무수히 많은 디렉토리가 생성되는게 아닐까? 생각했다.

그래서 차라리 내가 root부터 home과 같은 디렉토리를 구현을 하자고 생각했었다. 하지만 이게 잘못된 생각이라는 것을 깨닫기까지 얼마 걸리지 않았다. 디렉토리를 구현하기 위해서 처음에 양방향 연결 리스트를 사용했다가 몇 시간 뒤에 디렉토리는 트리가 제일 적합하다는 것을 깨닫게 되었다. 트리로 디렉토리를 구현하려고 했으나 디렉토리는 본편적으로 보면 동적 메모리이기 때문에 그것에 맞게 트리를 구현해야 했다, 하지만, 동적 메모리를 구현하는 것은 필히 어려운 것이었고 이 때, 과제에 의문이 들기 시작하면서 내가 왜 root와 home까지 구현을 해야하는 것인가 생각하게 되었다.

그리고 깨달았다. 내가 이때까지 하던 것은 OS 구현이었다. 과제의 목표는 쉘 프로그램 구현이지 OS 구현이 아니었다.

이때부터 난 방향을 잡기 시작하면서 코드를 작성하기 시작했다.

처음에는 멀티 파이프라인, 다중 명령어, 백그라운드 실행은 무시했다.

제일 처음 구현한 것은 일반적인 리눅스 bash 쉘의 사용자 인터페이스 형태를 띄게 구현하는 것이다.

이렇게 구현하기 위해선 사용자의 정보와 현재 디렉토리가 필요했다.

이것은 getuid와 gethostname으로 

이것을 구현할 수 있었다.

```
pw = getpwuid(uid);
gethostname(hostname, sizeof(hostname));
```

으로 사용자의 정보들을 수집했다.

그리고 현재 디렉토리 명을 쉘에 표시해야 했다. 그러기 위해선 getcwd가 필요하다.

getcwd는 현재 디렉토리의 주소를 그대로 가져온다. 그리고 '/'가 그대로 붙어나오기 때문에 '/'을 기준으로 folder 이름을 정리해야 한다.

```c
typedef struct node {
    char folder[256];
    struct node *next;
} Node;
```
f

디렉토리를 연결 리스트로 정리를 해서 만약 다른 명령어에 쓰일 수 있을 것 같아 정리하게 된 것이다. 그래서

```c
Node *load_current_path() {
    char cwd[1024];
    getcwd(cwd, sizeof(cwd));
    Node *head = NULL;
    Node *tail = NULL;
    char *folder = strtok(cwd, "/");
    while (folder != NULL) {
        Node *new_node = create_node(folder);
        if (head == NULL) {
            head = new_node;
            tail = new_node;
        } else {
            tail->next = new_node;
            tail = new_node;
        }
        folder = strtok(NULL, "/");
    }
    return head;
}
```

이렇게 '/'을 기준으로 폴더명을 리스트화 시켰다.

그리고 다시 리스트를 받아서 현재 디렉토리를 출력하는 함수를 만들면, 현재의 디렉토리 위치를 출력할 수 있다.

```c
void print_path(Node *head) {
    Node *current = head;
    while (current != NULL) {
        printf("/%s", current->folder);
        current = current->next;
    }
}
```

그리고 이걸 구현하고 보니, 이 함수는 pwd와 그냥 똑같다는 것을 깨닫게 되었다.

```
if (strcmp(command, "pwd") == 0) {
    print_path(path_list);
    printf("\n");
    continue;
}
```

그래서 실제로 command가 사용자가 입력받은 명령어라고 생각할 때, 이게 pwd면,

printf_path함수에 path_list를 줘서 그것을 출력한다. path_list는 현재 디렉토리를 담고있다. 그렇게 이상하게도 내부 명령어인 pwd를 구현하게 되었다.

그렇게 이어서 cd와 exit를 구현하기로 했다. Exit는 다른 쉘 프로그램을 살펴보았을 때, exit라는 명령어로 현재 쉘을 종료시키는 함수가 있으면 좋겠다고 해서 추가하게되었다.

cd는 chdir을 사용하면 된다. chdir함수는 자기가 받은 인자가 현재 디렉토리에서 이동가능하면 이동하고 불가능하면 0을 출력하게 된다. 이것을 가지고 cd와 똑같이 구현이 가능하게 되는것이다.

```
if (strncmp(command, "cd ", 3) == 0) {
    char *path = command + 3;
    ltrim(path);
    if (chdir(path) != 0) {
        perror("cd failed");
    } else {
        free_path_list(path_list);
        path_list = load_current_path();
    }
    continue;
}
```

여기서 char *path에 command + 3을 주는 것에 의아할 수 있다. 하지만 이것은 생각해보면 쉽게 알 수 있다. cd는 하나의 인자 값을 받는다. 즉, 이동할 디렉토리를 적어둔 곳은 "cd "다음에 저장이 되어있다. 그래서 공백 다음의 주소를 path에 할당을 시킨 것 이다. 그리고 사용자가 공백을 더 추가해서 "cd /mnt" 이렇게 적을 경우가 발생할 수 있다. 이것을 해결하기 위해 ltrim함수를 만들었다. ltrim은 문자열의 왼쪽 부분의 공백을 제거하는 함수로 "cd /mnt"이렇게 적어도 path에는 "/mnt"이렇게 저장이 되는 것이다. 그래서 주소가 변경되면, 현재 리스트에도 업데이트를 해줘야 하니 기존 경로는 지우고 새로운 경로를 업데이트 받는다.

```
if (strcmp(command, "exit") == 0) {
    printf("Exit the shell.\n");
    break;
}
```

Exit는 단순히 exit를 받으면 break로 while문을 빠져나와 기존 경로가 동적 메모리로 되어있으니 free하고 프로그램을 종료시킨다.

이렇게 cd, pwd, exit를 구현했다. 이제 필요한 것은 외부 명령어를 어떻게 구현할지 문제이다. 외부 명령어를 구현하기 위해선 파이프 라인, fork를 구현해야 했다.

그리고 stdin, stdout으로 멀티 파이프라인을 구현하면서 파이프 라인이 제대로 구현하기 위해 필요했다. 이제 외부 명령어는 exec 시스템콜으로 구현이 가능하기 때문에 좀 수월할 뻔 했다. 일단 외부 명령어는 부모 프로세스에서 fork를 하고 자식 프로세스에서 명령어를 수행한다음, 부모 프로세스의 wait을 푼다. 하지만 명령어가 한 개만 실행될 때에는 간단하다. 하지만 우리가 구현해야하는 것은 멀티 파이프라인이며 멀티 파이프라인을 구현하기 위해선 콜 스택에 자식프로세스를 넣으면서 하나씩 실행결과를 다음 자식 프로세스에 넘겨줘야 했다. 그리고 이걸 구현하기 위해서 dup2()함수를 stdin, stdout을 구현해서 멀티파이프라인이 가능하게 한다.

```
+ void pipe_command(char *command) {
+     char *args[20];
+     int cnt = 0;
+     args[0] = strtok(command, "|");
+     while (args[cnt] != NULL) {
+         ltrim(args[cnt]);
+         args[++cnt] = strtok(NULL, "|");
+     }
+     run_command(args);
+ }
```

일단, pipe_command함수를 만들어서 '|' 기준으로 명령어들을 쪼갠다. 그리고 ltrim으로 '|'을 입력할 때의 공백을 다 없앤다. 그리고 이것을 run_command에 보내면서 본격적인 pipeline이 실행되는 것이다.

```c
if (pid == 0) {
    close(pipefd[0]);
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[1]);

    char *cmd_args[20];
    int j = 0;
    cmd_args[j] = strtok(args[i], " ");
    while (cmd_args[j] != NULL) {
        cmd_args[++j] = strtok(NULL, " ");
    }
    execvp(cmd_args[0], cmd_args);
    perror("execvp failed");
    exit(1);
} else {
    close(pipefd[1]);
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]);
    run_command(&args[i + 1]);
    waitpid(pid, NULL, 0);  // 부모가 자식 종료 기다림
}
```

여기서 pid가 0이면 자식 프로세스를 가리킨다. 그리고 dup2로 stdin, stdout을 설정하고 다시 cmd_args로 명령어와 인자를 구분한다. 그래서 execvp으로 외부 명령어를 실행시고 만약 오류가 발생하면 perror로 execvp failed가 발생한다.

그리고 pid가 0보다 크면 부모 프로세스로 부모 프로세스에서 다시 자식 프로세스를 실행시키는 멀티 파이프라인을 구현한다. 이것은 일종의 call_stack이며, stack으로 자식 프로세스를 차곡차곡 쌓으면서 하나씩 다시 실행시켜 stack을 정리한다.

위에 사진은 run_command의 일부 코드이다. 즉, run_command가 재귀함수로 돌아가면서 멀티파이프 라인을 구현할 수 있게 된 것이다.

그 다음 다중 명령어와 백그라운드를 구현했다.

다중 명령어엔 &&, ||, ; 그리고 백그라운드인 &이가 있다.

;는 앞뒤 상관없이 순서대로 실행시키는 것이고 &&는 앞 명령이 성공하면 뒤에 있는 명령어도 실행한다. ||는 앞이 성공하면 뒤는 실행 안하고 앞이 실패하면 뒤를 실행하는 명령어이다.

```c
void parse_and_execute(char *command) {
    char *token = strtok(command, ";");
    while (token != NULL) {
        // && 처리
        char *and_pos = strstr(token, "&&");
        if (and_pos) {
            *and_pos = '\0';
            char *next = and_pos + 2;
            ltrim(token);
            ltrim(next);
            int status = run_pipeline(token);
            if (status == 0) run_pipeline(next);
            token = strtok(NULL, ";");
            continue;
        }
```

이건 다중 명령어를 실행하기 위한 코드로 먼저 ';'가 있는지 확인한다. ';'무조건 실행이기 때문에 이것은 하나의 블록을 나누는 경계선으로 생각할 수 있다. 즉, ;을 기준으로 실행하는 순서를 나누는 것이다. ';'이 나타나기 전의 명령어들이 먼저 실행이 되어야 하기 때문이다. 그래서 먼저 ';'이 있는지 확인하여 그 전까지 명령어를 token으로 저장하는 것이다. 그리고 and_pos, or_pos, bg_pos가 있는데 이것들은 &&, ||, &이 있는지 확인하는 변수들이다. 그래서 while문이 실행될 때 마다 변수들이 초기화하면서 예전에 들어있던 메모리 값을 초기화 시킨다. 그 다음 and_pos에 값이 들어가면, 즉 &&이 확인되면 if문에 들어가서 strstr함수는 특정 문자를 찾으면 그 문자가 시작하는 주소를 반환한다. 그렇기 때문에 and_pos에는 &&이 시작하는 주소를 반환한다. 따라서 *and_pos에 널 문자로 수정하는 것은 명령어가 run_pipeline을 실행할 때 어디까지 실행해야하는지 표시하기 위해서 하는 것이다. 그리고 next는 "&& "에서 공백 부분을 가리키게 해서 ltrim으로 자동적으로 공백이 제거되기 때문에 그 다음 명령어를 가리키게 할 수 있는것이다. 여기서 status는 run_pipeline의 반환값을 받는데,

```c
    return WIFEXITED(status) ? WEXITSTATUS(status) : -1;
}
```

여기의 삼항연사자의 결과에 따라 넘어간다. WIFEXITED는 프로세스가 정상적으로 종료되었는지 확인하는 함수이다. 그래서 정상적으로 종료가 되면, WEXITSTATUS가 실행되는 것이다. 그리고 이 함수는 status의 종료 코드를 추출하는 함수이다. 그래서 여기서 정상적으로 종료되면 WEXITSTATUS(status)에서 0이 반환되기 때문에

parase_and_execute에서 status에 0을 받으면서 &&에서 앞 부분이 참이므로 뒤에 있는 명령어가 실행되는 것이다.

그리고 만약 ||이면 이중 if문에서 안에 status가 0이 아닐경우 실행되게 만든다.

또한 특이한 명령어인 백그라운드 실행은 parse_and_execute에서 먼저 포크해서 자식 프로세스를 만든 다음 다시 run_pipline에서 fork를 진행한다.

이것을 이렇게 구성한 이유는 **좀비 프로세스**에 대해 방지하기 위해서이다.

좀비 프로세스는 자식 프로세스가 exit로 종료를 했는데, 부모가 프로세스를 wait으로 회수하지 않아서 발생한다. 그리고 좀비 프로세스는 백그라운드 작업을 할 때, 제일 많이 발생한다. 그래서 백그라운드 실행을 제일 유심해서 코드를 작성해야 한다.

좀비 프로세스는 부모 프로세스가 wait을 안해서 발생하는 것이기 때문에 fork하고 또 다시 자식 프로세스에서 fork해서 그 하위 프로세스를 또 만들고 자식 프로세스를 즉시 종료시킨다. 그리고 run_pipeline으로 인해 명령어가 실행되고 wait을 안한다. 자식 프로세스의 하위 프로세스는 종료시 init이 정리하게 된다. 그럼 부모 프로세스가 따로 정리를 안해도 알아서 정리가 되게된다. 간단하게 말하면 백그라운드를 실행하기위해선 자식 프로세스에서 명령어를 수행하는 것이 아니라 자식 프로세스의 하위 프로세스에서 명령어를 수행하고 자식 프로세스는 즉시 exit하고 부모 프로세스는 wait을 걸어서 자식 프로세스의 정보를 회수해서 좀비 프로세스가 생기지 않게 관리하는 것이다.

그리고 다중 명령어를 만들 때, pipe_command와 run_command가 합쳐서 작동하는 것이 효율적이라 생각해 run_pipeline으로 새로 함수를 만들었다.

이때까지 하나의 C파일으로 쉘을 구현했다. 하지만 과제에선 여러 파일로 나눠서 하나의 실행파일로 만들라고 했다. 그래서 나는 일반적인 함수들과 구조체를 gen_func.h으로 헤더파일을 만들어서 관리하기로 했다. 정리하면서 굳이 전역변수에 있을 이유가 없던 사용자 정보, 디렉토리 위치를 담은 path_list를 main함수에 넣어서 수정을 했다.

또한, 실제 쉘은 사용자의 정보가 담긴 부분과 현재 디렉토리를 나타내는 부분에는 다른 색상을 사용해서 구분을 했다. 그래서 이것을 구현하기 위해서 color.h이라는 헤더파일을 만들어서 사용자의 정보는 초록색, 현재 디렉토리는 파랑색, 그 다음 쉘이름은 노랑색으로 구분을 했다.