Operating-System
HW1 Report

DKU 단국대학교

32217072 모바일시스템공학과 김도익

Operating-System HW1

Freeday: 5

목차

1. Instruction

쉘(Shell)은 컴퓨터 시스템의 사용자와 운영 체제 간의 인터페이스 역할을 담당하는 프로그램으로, 사용자가 입력한 명령어를 해석하고 이를 운영 체제가 이해할 수 있도록 전달하는 중개자 역할을 한다. 쉘은 명령어 기반의 인터페이스(CLI)를 제공하여 사용자가 다양한 시스템 작업을 수행할 수 있도록 지원한다.

이 레포트는 유닉스 기반 운영 체제에서 작동하는 간단한 쉘 프로그램인 SiSH(Simplified Shell)의 구현을 다룬다. SiSH 는 명령어 해석 및 실행, 외부 프로그램 실행, 입출력 리다이렉션, 프로세스 생성 및 제어 등의 기능을 수행한다. 이 프로그램을 구현하기 위해 필요한 주요 시스템 호출로는 fork(), execve(), wait() 등이 있으며, 이를 통해 여러 프로그램을 실행하고 자식 프로세스가 정상적으로 종료될 때까지 부모 프로세스가 기다리도록 하였다.

본 레포트는 SiSH 의 동작 방식과 설계 방법, 코드 구현 세부사항을 다루며, 시스템 호출을 이용한 쉘 프로그램의 작동 원리를 설명한다.

–

2. 배경지식

2.1 쉘이란

쉘 프로그램은 유닉스 및 리눅스 운영 체제에서 가장 기본적이면서도 중요한 도구이다. 사용자는 쉘을 통해 시스템에 명령어를 전달하고, 이를 통해 다양한 작업을 수행할 수 있다. 쉘은 사용자가 입력한 명령어를 해석하고 실행할 수 있도록 하며, 외부 프로그램을 실행할 때 시스템 호출을 사용하여 새로운 프로세스를 생성하거나, 기존 프로세스를 관리한다.

이 프로젝트에서 구현한 SiSH(Simplified Shell)는 시스템 호출을 이용해 기본적인 쉘의 기능을 수행하는 간단한 CLI 프로그램이다. 시스템 호출(System Call)은 사용자 프로그램과 운영 체제 커널 간의 인터페이스 역할을 하며, 프로세스 관리, 파일 시스템 접근, 메모리 관리 등의 중요한 작업을 수행할 수 있다. SiSH는 아래와 같은 주요 시스템 호출을 활용하여 프로세스를 생성하고, 명령어를 실행하며, 자식 프로세스의 종료를 기다리는 작업을 수행한다.

- **fork()**: 새로운 프로세스를 생성한다. 부모 프로세스는 자식 프로세스의 PID를 반환받고, 자식 프로세스는 0을 반환받는다. 이를 통해 부모와 자식 프로세스가 동시에 실행되며, 외부 프로그램을 실행할 때 자식 프로세스를 생성해 실행을 담당한다.
- **execve()**: 자식 프로세스 내에서 새로운 프로그램을 실행하기 위해 사용된다. 이 호출은 현재 프로세스의 메모리와 컨텍스트를 지우고 새로운 프로그램을 메모리에 로드하여 실행한다. 쉘 프로그램에서 외부 명령어 실행 시 이 호출이 사용된다.
- **wait()**: 부모 프로세스가 자식 프로세스의 종료를 기다리도록 하는 시스템 호출이다. 자식 프로세스가 종료될 때까지 부모 프로세스는 일시 정지되며, 자식이 종료되면 자원을 해제하

고 제어를 다시 가져온다.

추가적으로, SiSH는 getenv() 함수로 PATH 환경 변수를 읽어 외부 프로그램의 경로를 탐색하며, strtok_r()를 사용하여 명령어와 인자를 파싱한다. PATH는 쉘이 외부 프로그램을 찾는 데 사용하는 환경 변수로, 여러 디렉토리 경로를 콜론(:)으로 구분하여 포함한다.

2.2 쉘 작동 흐름

쉘의 작동 원리는 다음과 같은 흐름을 따른다.

1. **prompt:** 쉘을 프롬프트를 출력한 후 사용자로부터 명령어를 입력받는다. 이때 사용자는 명령어, 인자, 리다이렉션, 파이프 등을 포함할 수 있다.
2. **Parsing:** 사용자가 입력한 명령어 문자열을 파싱하여 각 명령어, 인자, 그리고 리다이렉션 및 파이프 연산을 구분한다. 이를 통해 쉘은 명령어가 어떻게 실행되어야 할지 알수있다
3. **내부 명령어와 외부 명령어 구분:**
   A. 내부명령어: cd, pwd 등은 쉘 자체에서 처리할 수 있는 명령어이다. 이 경우 쉘이 직접 해당 작업을 처리하고 외부 프로그램을 실행하지 않는다. 본 프로젝트에서는 내부 명령어를 직접 구현하고 실행하였다
   B. 외부명령어: 쉘이 자체적으로 처리하지 않는 명령어는 운영체제의 파일 시스템에서 해당 명령어의 실행 파일을 찾아서 실행한다.
4. **프로세스 생성 및 관리:** 쉘은 외부 명령어를 실행할 때, 새로운 프로세스를 생성합니다. 이는 보통 fork() 시스템 호출을 통해 이루어지며, 자식 프로세스를 생성하여 명령어를 실행합니다. 그 후 exec() 계열 함수를 통해 자식 프로세스에서 실제 프로그램이 실행됩니다.
5. **리다이렉션 및 파이프라인:** 쉘을 표준 입력과 표준 출력을 다른 파일이나 명령어로 연결가능하다.
   A. 리다이렉션: 입력< 또는 > 출력을 파일로 리다이렉트한다
   B. 파이프: 명령어 간의 출력을 다음 명령어의 입력으로 연결하는 파이프를 사용한다.
6. **리다이렉션 및 파이프라인:** 외부명령어가 실행되고 나면 쉘은 해당 프로세스가 종료될때까지 기다린다. Wait()함수로 자식프로세스의 종료를 감시한다.

3. SISH 디자인

3.1 SISH 구조

SISH 주요 구조는 크게 세 부분으로 나뉜다:

Main.c: 프롬프트를 출력하여 실행됨을 알리고, 사용자로부터 명령어를 입력받고, execute 함수를 호출하는 루프를 담당한다.

Parsing.c: 사용자가 입력한 명령어 문자열을 파싱하여 각 명령어, 인자, 그리고 리다이렉션을 구분한다.

Execute.c: 명령어를 해석하고 처리하는 로직을 포함한다.

func.c: 명령어 실행과 관련된 함수들을 제공한다.

SISH 프로젝트는 기본적으로 사용자가 입력한 명령어를 분석하여 실행하는 기능을 목표로 하며, 리다이렉션과 같은 추가적인 기능도 지원한다. 이를 위해 명확하게 나눠진 구조를 통해 프로그램이 모듈화되고, 각 부분이 독립적으로 동작하면서도 서로 유기적으로 연결되어 쉘의 주요 기능을 구현하게 된다.

3.2 주요 기능

1. **프롬프트 출력 및 사용자 입력 처리**: SISH는 사용자로부터 명령어를 받기위해 프롬프트를 표시한다. 사용자 환경 변수를 사용하여 사용자 환경 변수를 사용하여 프롬프트를 사용자 맞춤형으로 출력할 수 있다.
2. **명령어 parsing**: 사용자가 입력한 명령어 문자열을 파싱하여, 문자열과 인자를 구분하고, 리다이렉션(<, >, >>)을 처리한다.
3. **내부 명령어 처리**: SISH는 cd, pwd, quit, help, echo, type 과 같은 내부 명령어를 처리할 수 있다.
4. **외부 명령어 처리:** SISH는 외부 프로그램을 실행할 수 있다. Ls, cat과 같은 명령어를 입력했을 때, 시스템의 PATH 환경 변수를 참조해 해당 프로그램을 찾아 수행한다.
5. **프로세스 제어:** 외부 명령어가 실행되는 동안 쉘을 비활성 상태가 되고, 해당 프로세스가 종료될 때까지 기다린다. 부모프로세스는 Wait() 시스템 호출을 통해 자식 프로세스가 종료될 때까지 대기하며, 자식 프로세스가 성공적으로 종료되면 다음 명령어 입력을 받을 수 있게 된다.
6. **오류 처리:** 명령어가 잘못되었거나 실행파일이 없을 경우 적절한 오류 메시지를 사용자에게 제공한다.

4. 코드 설명

4.1 main.c

```c
#include "shell.h"

int main(void){
    char cmd[MAX];

    while(1){
        printPrompt();
        fgets(cmd, sizeof(cmd) - 1, stdin);

        if (execute(cmd) == false) {
            break;
        }
    }
    return 0;
}
```

SiSH는 메인 루프를 통해 지속적으로 사용자 명령어를 입력받는다. fgets() 함수로 입력된 명령어는 처리를 위해 execute() 함수로 넘겨진다. 루프는 사용자가 quit 명령어를 입력할 때까지 반복된다.

pirntPrompt() 함수는 현재 작업 디렉토리, 사용자이름, 시간을 포함한 쉘 프롬프트를 출력한다. 이를 통해 사용자가 명령어를 입력할 수 있음을 알린다.

Execute() 함수는 입력된 명령어를 해석하고, 실행한다. 명령어가 quit인 경우에 false를 반환하여 SISH의 루프가 종료되도록 하였다.

4.2 parsing.c

tokenize함수

사용자가 입력한 명령어 문자열을 파싱하여 개별 토큰으로 분리하고, 입력 및 출력 리다이렉션을 처리하는 역할을한다.

```c
int tokenize(char *cmd, char *argv[100], char **input_file, char **output_file, int *append) {
    int tokensize = 0;
    char *saveptr;
    *input_file = NULL;    // 입력 파일을 초기화
    *output_file = NULL;   // 출력 파일을 초기화
    *append = 0;           // 덮어쓰기로 설정

    argv[tokensize] = strtok_r(cmd, DELIMS, &saveptr);  // 첫 번째 토큰 추출
    while (argv[tokensize] != NULL) {
        if (strcmp(argv[tokensize], ">") == 0) {
            // 출력 리다이렉션 (덮어쓰기 모드)
            argv[tokensize] = NULL;  // ">" 기호를 명령어 배열에서 제거
            *output_file = strtok_r(NULL, DELIMS, &saveptr);  // 출력할 파일 이름을 가져옴
        } else if (strcmp(argv[tokensize], ">>") == 0) {
            // 출력 리다이렉션 (추가 모드)
            argv[tokensize] = NULL;  // ">>" 기호를 명령어 배열에서 제거
            *output_file = strtok_r(NULL, DELIMS, &saveptr);  // 출력할 파일 이름을 가져옴
            *append = 1;  // 추가 모드로 설정
        } else if (strcmp(argv[tokensize], "<") == 0) {
            // 입력 리다이렉션
            argv[tokensize] = NULL;  // "<" 기호를 명령어 배열에서 제거
            *input_file = strtok_r(NULL, DELIMS, &saveptr);  // 입력할 파일 이름을 가져옴
        } else {
            tokensize++;  // 명령어 배열에 토큰 추가
            argv[tokensize] = strtok_r(NULL, DELIMS, &saveptr);  // 다음 토큰을 추출
        }
    }
    argv[tokensize] = NULL;  // 명령어 배열 마지막에 NULL 추가
    return tokensize;  // 토큰 개수 반환
}
```

tokenize() 함수는 입력된 명령어 문자열을 구분자를 기준으로 나누어 각 부분을 **토큰**으로 분리한다. 각 토큰은 argv 배열에 저장되며, 리다이렉션 기호(<, >, >>)를 처리한다. 출력 리다이렉션의 경우, >는 덮어쓰기, >>는 추가 모드로 작동하며, 입력 리다이렉션 <도 처리한다. 리다이렉션 기호는 명령어 배열에서 제거되고, 파일 이름은 별도로 저장된다. 리다이렉션이 없는 토큰은 argv 배열에 계속 추가되며, 마지막에 NULL을 추가하여 종료를 표시한다. 최종적으로 토큰의 개수를 반환하여 명령어 실행단계에서 토큰의 개수를 count할 때 사용된다.

4.2 execute.c

명령어를 처리하는 함수로 내부 명령어와 외부 명령어인지 판단하고, 그에 따른 적절한 처리를 수행한다. fork()와 execve()를 사용해 외부 프로그램을 실행하고, 내부 명령어는 직접 처리한다.

내부 명령어 목록 정의 및 환경 변수 배열 extern으로 가져오기

```
// 내부 명령어 목록 배열 정의
char *builtin_cmd[] = {"quit", "cd", "help", "pwd", "type","echo"};

// 환경 변수 배열을 extern으로 가져오기
extern char **environ;
```

builtin_cmd는 쉘에서 처리할 수 있는 **내부 명령어**들을 저장하는 배열이다. 내부 명령어는 별도의 외부 프로그램을 실행하지 않고, 쉘 자체에서 처리되는 명령어들이다. 이 배열은 쉘이 명령어 입력을 받을 때, 입력된 명령어가 내부 명령어인지 외부 명령어인지 판단하는 데 사용된다.

Environ은 환경 변수들의 배열이다. 환경변수들은 프로그램의 실행 환경을 제어하는 요소로, PATH, HOME, USER등의 변수는 시스템 경로, 사용자 정보 등을 담고 있다.

이 환경 변수들은 쉘이 실행되는 동안 다양한 작업에 활용되며, 특히 execve()와 같은 함수에서 프로그램이 실행될 때 이 환경 변수들이 프로그램에 전달된다. 이를 통해 프로그램은 실행되는 시스템의 설정 및 정보를 알 수 있다.

environ을 extern 키워드를 통해 가져오는 것은 현재 프로그램의 전역 환경 변수 배열을 참조하여, 외부 명령어를 실행할 때 이 정보를 전달하거나 읽을 수 있도록 한다. 쉘은 이 환경 변수들을 통해 외부 명령어를 정확하게 실행하고, 필요한 경우 환경 변수를 수정하여 실행 환경을 제어할 수도 있다.

외부명령어와 오류 처리

이 코드는 입력된 명령어가 내부 명령어가 아닐 경우, 이를처리하는 흐름을 보여준다.

```
// 현재 디렉토리의 명령어 실행 (./로 시작하는 경우)
if (strncmp(argv[0], "./", 2) == 0) {
    // 현재 디렉토리에서 파일 존재 여부 확인
    if (access(argv[0], F_OK | X_OK) == 0) {
        strcpy(pathname, argv[0]);
    } else {
        printf("%s: command not found or no execute permission\n", argv[0]);
        return true;
    }
}
// PATH 환경변수에서 명령어 검색
else if (is_in_path(argv, pathname)) {
    // pathname 설정 완료
} else {
    printf("%s: command not found\n", argv[0]);
    return true;
}
```

```
// 외부 명령어 실행
pid_t pid = fork();
if (pid == -1) {
    perror("fork error\n");
    return false;
} else if (pid == 0) {
    // Redirection
    if (input_file) {
        int fd_in = open(input_file, O_RDONLY);
        if (fd_in == -1) {
            perror("input redirection error");
            exit(EXIT_FAILURE);
        }
        dup2(fd_in, STDIN_FILENO);  // 표준 입력을 파일로 리다이렉트
        close(fd_in);
    }
    if (output_file) {
        int fd_out;
        if (append) {
            fd_out = open(output_file, O_WRONLY | O_CREAT | O_APPEND, 0644);  // 추가
        } else {
            fd_out = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);  // 덮어쓰기
        }
        if (fd_out == -1) {
            perror("output redirection error");
            exit(EXIT_FAILURE);
        }
        dup2(fd_out, STDOUT_FILENO);  // 표준 출력을 파일로 리다이렉트
        close(fd_out);
    }
    execve(pathname, argv, environ);
    perror("execve failed");
    exit(EXIT_FAILURE);
} else {
    wait(NULL);
    return true;
}
```

현재 디렉토리에서 명령어 실행:

strncmp(argv[0], "./", 2)를 통해 명령어가 ./로 시작하는지 확인한다. 이 경우, 명령어는 현재 디렉토리에서 실행 파일을 찾는다는 의미이다.

access() 함수를 사용해 명령어에 해당하는 파일이 존재하는지, 그리고 실행 권한이 있는지 확인한다. 파일이 존재하고 실행 가능할 경우, pathname에 해당 경로를 저장하고 실행 준비를 한다.

파일이 존재하지 않거나 실행 권한이 없으면 오류 메시지를 출력하고 종료한다.

PATH 환경변수에서 명령어 검색:

명령어가 ./로 시작하지 않으면, PATH 환경변수에서 해당 명령어를 검색한다.

is_in_path() 함수는 PATH에 설정된 여러 디렉토리를 순회하며, 실행 가능한 파일이 있는지 확인한다. 실행 가능한 파일을 찾으면 pathname에 그 경로를 저장하고, 파일을 실행할 준비를 한다.

프로세스 생성 및 명령어 실행:

fork() 시스템 호출을 통해 자식 프로세스를 생성한다. 자식 프로세스는 독립된 메모리 공간에서 명령어를 실행하기 위해 생성된다.

자식 프로세스에서는 execve()를 호출하여, 지정된 경로에 있는 외부 명령어를 실행한다. 이때, environ 배열을 통해 환경 변수도 전달된다.

만약 execve()가 실패하면, 오류 메시지를 출력하고 자식 프로세스를 종료한다.

입출력 리다이렉션:

명령어에 리다이렉션(입출력 변경)이 포함된 경우, dup2() 함수를 사용해 표준 입력이나 출력을 파일로 리다이렉트한다. 이를 통해 명령어의 입력은 파일에서 읽고, 출력은 파일로 저장할 수 있다.

출력 리다이렉션은 > 또는 >>로 처리되며, 덮어쓰기 모드 또는 추가 모드로 동작한다.

프로세스 대기:

부모 프로세스는 wait() 시스템 호출을 통해 자식 프로세스가 종료될 때까지 대기한다. 자식 프로세스가 종료되면, 쉘은 다시 사용자 입력을 기다리며 루프를 계속 진행한다

내부명령어 처리

내부 명령어는 셸 자체에서 처리되는 명령어이므로, 별도의 외부 프로그램 실행 없이 직접 구현하였다. 이러한 명령어들은 셸의 주요 기능 중 하나로, 시스템 호출이나 환경 변수 등을 활용하여 셸 내부에서 빠르고 효율적으로 처리된다.

1)quit명령어

```
// quit 명령어 (내부 명령어)
if (strcmp(argv[0], "quit") == 0) {
    return false;
}
```

사용자가 Quit을 입력하면, false를 반환하여 main루프를 벗어나 쉘이 종료되도록 하였다.

2)cd명령어

```
// cd 명령어 (내부 명령어)
if (strcmp(argv[0], "cd") == 0) {
    char *home_dir = getenv("HOME");  // HOME 환경 변수 설정
    if (paramCheck(tokensize) == true) {
        int result = chdir(argv[1]);  // 경로 변경
        if (result == -1) {
            printf("%s: No such file or directory\n", argv[1]);
        }
    } else {
        chdir(home_dir);
    }
    return true;
}
```

사용자가 cd를 입력하면 paramCheck()함수를 이용하여 경로 인자가 있는지 확인한다. 인자가 있는 경우, 해당 경로로 이동하며, 만약 경로가 잘못되었다면 오류메시지를 출력하고 쉘이 계속 작동하도록 처리하였다. 인자가 입력되지 않았을 경우, getenv()함수를 사용해 Home 환경 변수로 설정된 디렉토리로 이동할 수 있도록 구현하였다.

3)pwd명령어

```
// pwd 명령어 (내부 명령어)
if (strcmp(argv[0], "pwd") == 0) {
char *curr_dir = getcwd(NULL, 0);  // 현재 작업 디렉토리 가져오기
{if (curr_dir != NULL) {
    printf("%s\n", curr_dir);  // 현재 작업 디렉토리 출력
    free(curr_dir);  // getcwd로 동적 할당된 메모리 해제
} else {
    perror("getcwd error");  // 에러 발생 시 메시지 출력
}}
return true;
}
```

사용자가 pwd를 입력하면 getcwd를 사용하여 현재 작업디렉토리를 가져온다. Getcwd()함수는 현재 작업 디렉토리의 경로를 가져오는 시스템 호출이다. Getcwd(NULL,0)은 메모리를 동적할당하기 때문에 free로 메모리 해제하여 누수를 방지하였다.

4)type명령어

```
// type 명령어 (내부 명령어)
if (strcmp(argv[0], "type") == 0) {
    for (int token = 1; argv[token] != NULL; token++) {
        char final_path[256];

        // is_in_path() 함수를 사용하여 명령어 경로 확인
        if (is_in_path(argv + token, final_path)) {
            printf("%s is an external command\n", argv[token]);
        } else if (is_in_arr(argv[token], extern_cmd_arr, 5)) {
            printf("%s is a shell builtin\n", argv[token]);
        } else {
            printf("%s: command not found\n", argv[token]);
    }
}
return true;
}
```

type 명령어는 사용자가 입력한 명령어가 외부 명령어인지, 내부 명령어인지, 혹은 존재하지 않는 명령어인지를 판단하여 그 결과를 출력하는 기능을 제공한다. 이를 구현하기 위해 is_in_path() 함수를 활용하여 명령어가 시스템의 PATH 환경 변수에 정의된 경로에 존재하는지 확인한 후, 그에 맞

는 메시지를 출력한다.

5)기타 내부 명령어

```
// help 명령어 (내부 명령어)
if (strcmp(argv[0], "help") == 0) {
    printHelp();
    return true;
}

// echo 명령어 (내부 명령어)
if (strcmp(argv[0], "echo") == 0) {
    for (int i = 1; argv[i] != NULL; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return true;
}

// 명령어가 존재하지 않는 경우
printf("%s: command not found\n", argv[0]);
return true;
```

사용자가 help를 입력하면 prinThelp()함수를 호출하여SISH 사용시 도움말을 출력한다.

사용자가 echo를 입력하면 입력한 문자 그대로 argv[1]부터 배열에 있는 인수들을 순차적으로 출력한다. 사용자가 입력한 문자열을 그대로 출력하는 명령어이다.

사용자가 존재하지 않는 명령어를 입력시 오류 메시지를 출력하여 명령어가 존재하지 않음을 알린다. True를 반환하여 쉘이 계속 실행되도록 한다.

4.3 func.c

func.c는 쉘의 주요 기능을 구현하기 위한 다양한 함수를 정의하고 있다. 각각의 함수는 쉘 명령어 실행을 돕는다.

1)is_in_path

```
// PATH 환경 변수 참조
bool is_in_path(char *argv[100], char *final_path) {
    char *pathenv = getenv("PATH");  // 환경 변수 'PATH' 가져오기
    if (pathenv == NULL) {
        return false;  // PATH 환경 변수가 없는 경우
    }

    char *path_copy = strdup(pathenv);  // strtok 사용을 위한 복사본
    char *dir = strtok(path_copy, ":"); // ':' 구분자로 디렉토리 추출

    while (dir != NULL) {
        // 각 디렉토리에서 명령어의 경로 만들기
        snprintf(final_path, MAX, "%s/%s", dir, argv[0]);

        // 해당 경로에 실행 파일이 있는지 확인
        if (access(final_path, F_OK) == 0) {
            free(path_copy);  // 동적 메모리 해제
            return true;
        }

        // 다음 디렉토리로 이동
        dir = strtok(NULL, ":");
    }

    free(path_copy);  // 동적 메모리 해제
    return false;  // PATH에 없으면 false
}
```

이 함수는 getenv()를 사용해 시스템의 PATH환경변수를 가져온 후 strtok()로 각 디렉토리를 하나씩 추출해 해당 디렉토리에서 명령어 파일이 존재하는지 확인한다.

Access()함수를 사용해 명령어가 있는지 검사하고, 명령어가 있으면 그경로를 final path에 저장한 뒤 true를 반환한다. 모든 경로를 탐색한 후에도 명령어가 없다면, 동적 메모리를 해제하고 false를 반환한다.

2)is_in_arr

```
// 배열 요소 검사
bool is_in_arr(char *element, char *arr[], int size) {
    for (int i = 0; i < size; i++) {
        if (strcmp(element, arr[i]) == 0) {
            return true;  // 요소가 리스트에 있음
        }
    }
    return false;  // 요소가 리스트에 없음
}
```

이 함수는 주어진 명령어가 특정 명령어 목록 배열에 존재하는지 확인한다. 이를 통해 명령어가 내부 명령어 배열에 속하는지 확인할 수 있다. SISH에서 명령어가 내부 명령어인지를 검사할 때 사용된다.

3)paramCheck

```
// 인자 체크
bool paramCheck(int tokensize){
    // 인자 없을경우 false return
    if(tokensize == 1){
        return false;
    }
    // 인자 있을경우 true를 반환
    return true;
}
```

paramcheck함수는 명령어에 인자가 있는지 확인하는 역할을 한다. 인자가 없으면 false를 반환하고 있으면 true를 반환한다. Execute()함수에서 인자가 있는지 확인할 때 사용한다.

4)printPrompt

```c
void printPrompt(void){
    char *pwd = getcwd(NULL,0);    // 현재 작업 디렉토리
    char *user = getenv("USER");  // 사용자 이름
    time_t now = time(NULL);
    struct tm *local = localtime(&now);
    char time_str[9];
    strftime(time_str, sizeof(time_str), "%H:%M:%S", local);  // 현재 시간

    // 사용자 이름, 현재 디렉토리, 시간으로 프롬프트 표시
    if (user && pwd) {
        printf("%s [%s] %s$ ", pwd, time_str,user);
    }     // 오류 메시지를 출력
    else {
        if (!user) {
            fprintf(stderr, "Failed to retrieve user from environment.\n");
        }
        if (!pwd) {
            fprintf(stderr, "Failed to retrieve current directory.\n");
        }
        printf("$ ");  // 오류가 발생해도 기본 프롬프트 출력
    }
    free(pwd);
```

PrintPrompt 함수는 셸에서 사용자에게 표시되는 프롬프트를 출력한다. Getcwd()와 getenv()를 사용하여 디렉토리 절대 경로와 사용자이름을 가져오고 time()과 strftime()함수를 사용하여 현재 시간을 가져와 이 정보들을 조합하여 프롬프트를 형성하고 출력한다.

만약 user나 디렉토리 정보를 가져오는데 실패하면 오류메시지를 출력하고 기본 프롬프트를 표시한다.

5.프로그램 실행방법

이 섹션에서는 Makefile을 사용하여 쉘 프로그램을 컴파일하고, 쉘을 실행하는 방법을 설명한다.

5.1 Makefile 작성 및 컴파일

Makefile은 프로그램을 자동으로 컴파일하고 필요한 파일을 정리하는 도구이다. 각 파일의 종속 관계를 파악해 Makefile에 기술된 대로 컴파일 명령이나 shell명령을 순차적으로 수행한다. 각 파일에 대한 반복적 명령을 자동화 하여 컴파일러 명령어를 수동으로 입력하는 번거로움을 덜어준다.

```
CC = gcc
CFLAGS = -Wall -g
OBJS = main.o execute.o func.o parsing.o

TARGET = SISH

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

main.o: main.c shell.h
    $(CC) $(CFLAGS) -c main.c

parsing.o: parsing.c shell.h
    $(CC) $(CFLAGS) -c parsing.c

execute.o: execute.c shell.h
    $(CC) $(CFLAGS) -c execute.c

func.o: func.c shell.h
    $(CC) $(CFLAGS) -c func.c

clean:
    rm -f *.o $(TARGET)
```

5.2 컴파일 방법

터미널에서 다음 명령어를 입력하여 프로그램을 빌드한다.

- Make SISH

위 명령어를 실행하면 Makefile이 실행되어 소스 파일들이 컴파일되고 SISH 라는 이름의 실행파일이 생성된다.

다음 명령어는 이전에 컴파일 된 객체 파일 및 실행파일을 모두 삭제한 후 새로운 빌드를 진행할 수 있도록 준비하는 명령어이다.

- Make clean

5.3 쉘 실행 방법

프로그램이 컴파일되고 나면 실행파일을 실행하여 SISH을 시작할 수 있다. 다음 명령어를 사용하여 SISH를 실행한다.

- ./SISH

6.프로그램 실행

1)컴파일

```
doik@DESKTOP-9OSLRDI:~/os_hw/os_hw_1$ make SISH
gcc -Wall -g -c main.c
gcc -Wall -g -c execute.c
gcc -Wall -g -c func.c
gcc -Wall -g -c parsing.c
gcc -Wall -g -o SISH main.o execute.o func.o parsing.o
```

2)SISH 실행

```
doik@DESKTOP-9OSLRDI:~/os_hw/os_hw_1$ ./SISH
/home/doik/os_hw/os_hw_1 [20:40:18] doik$
```

3) help 실행

```
/home/doik/os_hw/os_hw_1 [20:55:59] doik$ help
============================================================
쉘에서 지원하는 명령어 및 기능
대부분의 기본 명령어 사용가능하며 외부명령어도 지원
cd      : Change the shell working directory.
quit    : Exit the shell.
help    : Display information about builtin commands.
type    : Display information about command type.
pwd     : Print the name of the current working directory.
echo    : Write arguments to the standard output.
redirection 지원
> [파일]                : 명령어 출력을 해당 파일에 저장합니다 (덮어쓰기).
>> [파일]               : 명령어 출력을 해당 파일에 추가합니다.
< [파일]                : 파일로부터 명령어 입력을 받습니다.
============================================================
/home/doik/os_hw/os_hw_1 [20:56:02] doik$
```

4) ls -al 실행

```
/home/doik/os_hw/os_hw_1 [20:56:02] doik$ ls -al
total 128
drwxr-xr-x 5 doik doik  4096 Sep 29 20:55 .
drwxr-xr-x 3 doik doik  4096 Sep 27 16:10 ..
drwxr-xr-x 8 doik doik  4096 Sep 29 12:34 .git
drwxr-xr-x 2 doik doik  4096 Sep 27 16:10 .vscode
-rw-r--r-- 1 doik doik   412 Sep 29 19:02 Makefile
-rwxr-xr-x 1 doik doik 30472 Sep 29 20:55 SISH
-rw-r--r-- 1 doik doik  4071 Sep 29 20:53 execute.c
-rw-r--r-- 1 doik doik 11568 Sep 29 20:55 execute.o
-rw-r--r-- 1 doik doik  3327 Sep 29 20:48 func.c
-rw-r--r-- 1 doik doik 13024 Sep 29 20:55 func.o
-rw-r--r-- 1 doik doik   226 Sep 28 00:35 main.c
-rw-r--r-- 1 doik doik  6592 Sep 29 20:55 main.o
-rw-r--r-- 1 doik doik  1682 Sep 29 19:00 parsing.c
-rw-r--r-- 1 doik doik  5440 Sep 29 20:55 parsing.o
-rw-r--r-- 1 doik doik   596 Sep 29 19:00 shell.h
-rw-r--r-- 1 doik doik  5654 Sep 27 16:10 test.c
drwxr-xr-x 2 doik doik  4096 Sep 27 16:10 test_dir
/home/doik/os_hw/os_hw_1 [21:09:11] doik$
```

5) cd 실행 후 ls 실행

```
/home/doik/os_hw/os_hw_1/test_dir [21:59:44] doik$ ls
1.txt  test1  test1.c
/home/doik/os_hw/os_hw_1/test_dir [21:59:45] doik$
```

6) 실행 파일 실행

```
/home/doik/os_hw/os_hw_1/test_dir [22:01:35] doik$ ./test1
hello world
/home/doik/os_hw/os_hw_1/test_dir [22:01:42] doik$
```

7) redirection

```
/home/doik/os_hw/os_hw_1/test_dir [22:01:42] doik$ ls > output.txt
/home/doik/os_hw/os_hw_1/test_dir [22:02:50] doik$
```

test_dir > output.txt

```
1  1.txt
2  output.txt
3  test1
4  test1.c
5
```

8) quit 실행

```
/home/doik/os_hw/os_hw_1/test_dir [22:02:50] doik$ quit
doik@DESKTOP-9OSLRDI:~/os_hw/os_hw_1$
```

7.결론

쉘은 운영체제와 사용자 간의 상호작용을 가능하게 하는 매우 중요한 도구이다. 이번 SISH 프로젝트를 구현하면서 쉘의 세부 작동 원리를 보다 깊이 이해할 수 있었다. 특히, 쉘이 단순한 명령어 실행기가 아니라, 프로세스 제어와 자원 관리를 포함한 다양한 기능을 수행하는 복잡한 시스템이라는 점을 실감했다.

이번 구현에서 fork()와 execve()와 같은 시스템 호출을 통해 프로세스 생성 및 외부 프로그램 실행이 어떻게 이루어지는지 알 수 있었으며, 부모 프로세스와 자식 프로세스 간의 상호작용이 쉘의 핵심적인 동작 원리라는 것을 배웠다. 또한, 내부 명령어와 외부 명령어의 구분, 그리고 입출력 리다이렉션과 기능들이 어떻게 구현되는지 경험할 수 있었다.이를 통해 운영체제의 기본적인 작동 원리와 쉘의 역할에 대해 보다 깊이 이해하게 되었다.

하지만 이번 프로젝트에서 파이프(pipes) 기능을 완벽하게 구현하는 데는 실패한 점이 아쉬웠다. 파이프는 명령어 간의 출력을 다른 명령어의 입력으로 연결하는 매우 중요한 기능이지만, 이를 구현하는 과정에서 예상보다 많은 어려움을 겪어 구현에 실패하였다.

결론적으로, 이번 SISH 프로젝트는 쉘의 구조와 그 기능에 대한 전반적인 이해를 높였으며, 실제 프로세스 제어 및 자원 관리를 구현해보면서 쉘이 단순한 명령어 해석기를 넘어서는 중요한 도구임을 다시금 깨닫게 해주었다. 파이프 기능을 완벽하게 구현하지 못한 것은 아쉽지만, 파이프의 기능과 중요성을 알게 되었다.