Cykor_week2_2024350024_이근목

처음에는 makefile을 제대로 이해를 못해서 c언어 파일 하나에 모든 기능을 넣으려고 노력했었다. 그러다가 gpt에게 도움을 받으며 파일을 나누는 것을 시작했다.

1. 과제 개요 및 목표

이번 과제의 목표는 리눅스 환경에서 작동하는 간단한 셸(shell) 프로그램을 직접 구현함으로써 운영체제의 핵심 원리 중 하나인 명령어 해석기(command interpreter)의 작동 방식을 이해하는 것이다. 기본적으로 사용자가 입력한 명령어를 분석하고, 이를 적절한 방식으로 실행하는 셸을 작성하며, 다음과 같은 기능들이 반드시 구현되어야 했다:

- 내장 명령어 cd, pwd 구현 (exec 사용 금지)
- 외부 명령어 실행 (exec 사용 허용)
- 다중 명령어 실행 (;, &&, ||)
- 파이프라인 및 멀티 파이프라인 (|)
- 백그라운드 실행 (&)
- exit 명령어로 종료

또한, 과제에서는 리눅스 bash 셸 스타일의 프롬프트 형식(사용자@호스트:디렉토리$)을 출력하도록 요구되었다.

2. 기능 구현 설명 및 주요 코드 구조

-프롬프트 출력:

사용자명, 호스트명, 현재 디렉토리를 함께 보여주는 bash 스타일 프롬프트는 main.c에서 getpwuid(), gethostname(), getcwd()를 사용해 다음과 같이 구현하였다:

```
char cwd[1024], hostname[256];
struct passwd *pw = getpwuid(getuid());
getcwd(cwd, sizeof(cwd));
gethostname(hostname, sizeof(hostname));
printf("%s@%s:%s$ ", pw→pw_name, hostname, cwd);
```

-명령어 입력 및 파싱:

fgets()로 입력을 받고, 우선순위에 따라 파이프 → 논리 연산자 → 일반 명령 순서로 파싱한다. strtok_r()을 중첩 사용해 우선순위에 따라 명령어를 분해하고, 실행 함수 execute() 또는 handle_pipe()로 분기한다.

내장 명령어 처리:

execute.c 파일 내부에서 strcmp()를 통해 "cd", "pwd", "exit" 여부를 판단한다. cd는 chdir()로, pwd는 getcwd()로, exit은 exit(0)으로 처리하였다.

```
if (strcmp(args[0], "cd") == 0) {
    chdir(args[1]);
    return 0;
}
```

-외부 명령 실행:

내장 명령이 아닌 경우 fork()로 자식 프로세스를 생성하고 execvp()로 명령어를 실행한다.

```
pid_t pid = fork();
if (pid == 0) {
    execvp(args[0], args);
    perror("exec 실패");
    exit(1);
}
```

-백그라운드 실행:

args 배열의 마지막 인자가 &이면 이를 제거하고 wait()을 생략한다. 이를 통해 부모 프로세스는 기다리지 않고 다음 명령을 받을 수 있다.

다중 명령어 처리:

한 줄에 여러 명령어가 있는 경우 ;, &&, || 기준으로 분리하여 실행 흐름을 제어한다. ;는 순차 실행, &&는 앞 명령 성공 시 실행, ||는 실패 시 실행하도록 조건을 나눈다.

-파이프 및 멀티 파이프:

| 를 기준으로 나눈 명령어들을 배열에 저장하고, 각 구간에 대해 fork()와 dup2()를 이용해 stdout → pipe → stdin 형태로 입출력을 연결한다.

```
pipe(fd);
fork();
dup2(fd[1], STDOUT_FILENO);
execvp(...);
```

이 구조를 반복문으로 확장하여 n개의 명령어가 파이프로 연결된 구조도 처리할 수 있도록 설계하였다.

3. 어려웠던 점 및 해결 과정

   cd 명령어는 외부 명령이 아닌 내장 명령으로 부모 프로세스의 현재 디렉토리에 직접 영향을 주어야 했다. 자식 프로세스에서 실행하면 디렉토리 이동이 유지되지 않기 때문에 부모 안에서 직접 처리하도록 구조를 분기하였다.

   멀티 파이프라인 구현 시 dup2()와 pipe()의 연결 순서를 정확히 맞춰야 하며, 자식 프로세스가 파이프의 어느 쪽을 사용할지에 따라 입력과 출력을 적절히 연결해야 했다. 또한 모든 자식에서 사용하지 않는 pipe를 close()하지 않으면 프로그램이 멈추거나 출력이 꼬이는 문제가 발생하였다.

   다중 명령어 처리도 단순히 순서대로 실행하는 것이 아니라, &&와 ||의 조건 흐름에 따라 다음 명령 실행 여부를 판단해야 했다. 이 때 명령어 실행 결과로부터 종료 코드를 받아오는 waitpid()와 WEXITSTATUS 매크로를 활용하였다.

   <Makefile 구성에 대한 고민 및 각 파일의 역할 설명>

   초기에는 단일 파일로 전체 코드를 작성하고 gcc shell.c -o shell 형태로만 컴파일하였다. 그러나 기능이 점차 많아지고 구조가 복잡해지면서 소스를 main.c, execute.c, pipe.c, shell.h 등으로 나누었다. 이후에는 Makefile을 구성하여 다음과 같은 고민을 통해 작성하였다:

main.c, execute.c, pipe.c가 각각 다른 역할을 하므로 개별 .o 파일로 컴파일한 후 링크하는 구조로 설정

make, make run, make clean 등으로 개발과 테스트, 정리를 빠르게 할 수 있도록 규칙 구성

의존성 관리를 위해 main.o: main.c shell.h 같은 형식으로 헤더파일 변경 시 자동 재컴파일 되도록 설정

main.c: 사용자 입력과 프롬프트를 관리하며 프로그램의 흐름을 담당한다. 파이프, 논리 연산자 등을 판단하여 적절히 실행 함수로 분기한다.

execute.c: cd, pwd, exit 같은 내장 명령어 처리와 외부 명령어 실행을 포함한다. 백그라운드 여부도 여기서 판단한다.

pipe.c: 여러 개의 명령어가 |로 연결된 경우 각 명령어마다 파이프 입출력을 구성하고, dup2()와 fork()를 사용해 병렬 실행을 구현한다.

shell.h: 모든 함수 선언을 포함하여 모듈 간 인터페이스를 정의한다.

이를 통해 코드 구조가 깔끔해졌을 뿐 아니라, 기능별로 디버깅이 수월해지고 역할 분리가 명확해졌다. 부분 수정 시 전체 파일을 다시 컴파일하지 않아도 되는 효율성도 얻을 수 있었다. 또한 run 명령어를 통해 바로 실행해볼 수 있는 구조로 설정하여 디버깅도 쉬워졌다.