Cykor 2주차
리눅스 쉘 구현

2025350002 스마트보안학부 권기완

1. 과제목적

이번 과제의 목적은 리눅스 쉘 프로그램을 직접 구현함으로써 실제 쉘이 어떤 식으로 사용자 입력을 처리하고 다양한 명령어를 실행시키는지를 이해하는 것이다. 이를 통해 fork(), execvp(), pipe(), wait() 등의 시스템 콜이 어떻게 작동하는지를 학습을 목표로 한다.

2. 개념정리(영상기반)

쉘 - 리눅스 커널과 사용자를 연결해주는 인터페이스. 이번 과제는 이러한 쉘의 기본 구조를 직접 재현해보는 작업이었다.
파이프라인(|) - 파이프라인 앞 명령어의 출력값을 파이프라인 뒤 명령어의 입력값으로 사용할 수 있게 해 준다. 이번 과제에서는 pipe() 함수와 dup2()를 통해 구현하였다.
백그라운드 실행(&) - 터미널에서 시작하여 백그라운드에서 사용자 개입 없이 실행되는 프로세스(명령어 뒤에 붙여서 사용)
fork: 부모 프로세스에서 호출되면 새롭게 자식 프로세스를 생성한다. fork에 의해 생성된 자식 프로세스는 부모 프로세스의 메모리를 그대로 복사하여 가진다. 본 과제에서는 외부 명령어 실행 전 항상 fork()를 통해 새로운 프로세스를 생성했다.
exec 계열 함수 - 현재 실행되고 있는 프로세스를 다른 프로세스로 대신하여 새로운 프로세스를 실행한다(이미 생성된 프로세스로 대체하는 것이다)
대부분의 외장 명령어는 fork -> exec계열 함수(새로운 프로세스를 실행 후 해당 프로세스를 초기화) -> 명령어 실행 -> exit 과정을 통해 실행된다
IPC(프로세스 간 커뮤니케이션) - 리눅스가 사용하는 IPC 중 하나인 PIPE(커널영역에 파이프를 생성해 데이터를 주고받는 구조)를 거쳐서 두 개의 프로세스가 서로 커뮤니케이션한다
파일 디스크립터 - 리눅스 혹은 유닉스 계열의 시스템에서 프로세스가 파일을 다룰 때 사용하는 개념으로, 프로세스에서 접근할 때 사용하는 추상적인 값
기본적으로 할당되는 파일 디스크립터 - 0: 표준입력 1: 표준 출력 2: 표준 에러
make - 소프트웨어 개발을 위해 유닉스 계열 운영체제에서 사용되는 프로그램 빌드 도구
Makefile - 프로그램을 빌드하기 위해 make문법에 맞춰 작성하는 문서본. 이 프로젝트는 Makefile을 작성하여 make 명령어로 컴파일 및 빌드를 자동화하였다.

3. 코드구현

(1) 전체 구조

이 쉘은 사용자의 명령어 입력을 의미 단위로 파싱하고, 내부 명령어는 직접 구현으로 외부 명령어는 execvp 명령어를 통해 실행하는 구조로 구성하였다. 입력된 명령어는 파싱 과정을 거쳐 논리 연산자(; && ||)와 파이프(|), 공백을 기준으로 파싱하고, 각 명령어가 내부 명령어인지 외부 명령어인지에 따라 적절한 시스템 콜(gethostname, fork, execvp, pipe, wait 등)을 통해 실행된다. 또한 백그라운드 실행(&)은 별도로 분기 처리하도록 설계하였다. cd, pwd와 같은 내부 명령어는 직접 구현하였으며, hello라는 기존 리눅스에서는 존재하지 않는 내부 명령어를 구현하였다.

(2) 세부 구조

사용자 인터페이스 – getenv, gethostname과 같은 시스템 콜을 통해 운영체제로부터 환경변수를 제공받고 이를 화면에 출력하였다.

파싱 – strtok와 strtok_r을 이용하여 특정 문자(공백, 논리연산자 등)를 기준으로 토큰을 쪼개고 이를 splitCmds같은 배열에 저장함으로써 파싱하였다.

내부 명령어 – cd, pwd라는 별개의 함수를 main함수 밖에 만들어 그 안에서 getcwd나 chdir 같은 시스템 콜을 이용하여 구현하였다.

외부 명령어 – fork함수를 통해 부모 프로세스와 같은 자식 프로세스를 복제한 후 execvp 함수를 이용하여 자식 프로세스를 외부명령어로 덮어씌움으로써 구현하였다.

다중 명령어 – 파싱 과정 이후 goNext 변수를 이용하여 특정 조건을 만족하면 다음 조건이 실행되도록 구현하였다.

파이프 라인 – 파싱 과정 이후 pipe 명령어를 통해 pipe를 생성하고 이 파이프의 FD값을 pipes배열에 저장하였다. pipes의 앞 인덱스는 0과 1을 번갈아가며 활용하였고, 뒤 인덱스는 0은 입력으로 1은 출력으로 활용하였다. 이후 fork를 통해 자식 프로세스를 만들고 dup2 함수를 통해 파이프의 입출력 방식을 변경하여 구현하였다.

백그라운드 실행 – 백그라운드 실행이 아닐 때는 waitpid를 이용하여 부모가 자식 프로세스를 기다리도록 구현하였으나 백그라운드일 때는 부모가 자식을 기다리지 않고 실행되도록 함으로써 구현하였다.

hello(추가 명령어) - pwd, cd 같은 내부 명령어 구현 부분에서 hello라는 기존 리눅스에 없는 새로운 명령어를 만들어 시스템 콜을 통해 사용자의 이름을 부르며 인사하도록 구현하였다.

(3) 전체 코드가 길어져서 각 역할의 코드가 어디에 있는지 찾기 쉽도록 show_prompt, pwd, cd, pipeline 등으로 함수를 쪼갬으로써 전체 코드의 유지보수가 용이하도록 구성하였다.

(4) 파싱 구현 예시

```
char *savePtr;
char *cmdToken = strtok_r(line, ";", &savePtr); // ;를 기준으로 첫번째 주소 반환
while (cmdToken != NULL && splitCount < 10) // ;가 끝날 때까지 반복
{
    splitCmds[splitCount++] = cmdToken; // splitCmds 에 cmdToken의 주소값 저장
    cmdToken = strtok_r(NULL, ";", &savePtr); // ;를 기준으로 다음 주소 반환
}
```

코드 전반에 걸쳐 이와 같은 파싱 구조가 반복적으로 사용되었다.
strtok_r을 통해 입력받은 명령어를 ; && 공백 등을 기준으로 쪼개고 첫 번째 토큰 주소를 cmdToken에 저장한다. 이후 반복문을 통해 splitCmds[splitCount]에 cmdToken 값을 저장하고 두 번째 토큰의 주소로 cmdToken의 값을 갱신되어 저장된다. 이후 다시 이 값을 splitCmds[splitCount]에 저장한다. 이 과정을 반복함으로써 파싱된 문자열들의 시작 주소가 splitCmds 포인터 배열에 차례대로 저장되도록 구현하였다.

4. 과제 중 고민한 점

쉘을 구현하면서 기능 구현만이 아니라 함수의 작동 방식과 시스템콜의 구조에 대한 여러 궁금증이 생겨 이를 찾아보며 이해하게 되었다.
먼저 getcwd와 같은 함수를 활용할 때 항상 인자로 배열의 크기를 제공해야 했는데 이유가 궁금하여 알아보니 경로의 길이가 너무 길 경우 버퍼 오버플로우라는 보안 취약점이 발생하고, 이를 방지하기 위해 버퍼의 크기를 인자로 제공해야 한다는 것을 알게 되었다.

구현 과정에서 getcwd 같은 함수들에서 배열의 크기를 인자로 항상 넣길래 넣어야 하는 이유가 궁금해서 검색해 보았는데 경로의 길이가 너무 길 경우 버퍼 오버플로우라는 보안 취약점이 발생하고, 이를 방지하기 위해 버퍼의 크기를 인자로 제공한다는 것을 알게 되었다. 또한 특이한 자료형을 쓰는 경우가 있길래 살펴 보니 pid를 저장하는 변수는 pid_t라는 별개의 자료형을 사용하는 경우가 많다는 것을 알게 되었다.

파이프 구현 과정에서 pid_t pid = fork(); 에서 pid는 한 개의 변수인데 어떻게 자식의 pid와 0 또는 -1을 동시에 저장하는지 궁금했는데 알아보니 두 개의 부모, 자식 프로세스에 각각 변수 pid가 존재하고 그 안에는 fork함수가 반환하는 값이 저장되는 것을 이해하게 되었다. 이와 연결되어 부모와 자식 프로세스가 동시에 존재하면 그 아래의 명령어는 어느 프로세스에서 실행되는 것이 의문이었지만 두 프로세스에 모두 실행되고 if문 같은 코드를 통해 흐름을 조절해야 하는 것을 알게 되었다.

또한 wait 명령어를 필요성에 대해 고민하였는데 wait을 사용하지 않으면 출력 순

서가 꼬이게 되고 자식 프로세스가 종료되지 않아 좀비 프로세스가 될 수 있다는 것을 알게 되었다. 이 문제를 방지하기 위해 백그라운드 실행이 아닐 경우에는 wait 이나 waitpid를 이용함으로써, 확실히 자식 프로세스가 종료되게 만들어야 한다는 것을 알게 되었다. wait 명령어가 1개의 자식 프로세스만을 기다리기에 fork로 만든 자식 프로세스 수만큼 wait을 반복시켜야 한다는 것을 새롭게 배웠다.

파싱 과정에서 strtok 명령어가 작동 했을 때 문자열의 처음 주소만 반환할 텐데 어떻게 문자열의 끝이 어딘지 어떻게 구분하는지가 궁금했는데, 찾아보니 뒤 인자를 널문자로 대체함으로써 문자열의 끝을 파악하게 된다는 것을 알게 되었다.

쉘 구현 중 가장 어려웠던 부분은 pipe함수가 정확히 어떤 과정을 통해 파이프를 생성하고 코드에서 이 파이프를 어떻게 활용하는지였다. 이 궁금증은 Chatgpt의 도움을 통해 pipe함수는 운영체제에 파이프 생성을 요청한 후 전달 받은 배열에 파이프의 입구와 출구의 FD를 저장함으로써 앞의 명령어의 출력값을 뒤의 명령어에 전달한다는 것을 이해하여 해결하였다.

마지막으로, 파이프 사용 후 close()를 하지 않으면 입력이 계속 들어올 수 있다고 판단되어 프로그램이 대기 상태(블로킹)에 빠지는 문제가 발생한다는 점에서, 파이프를 쓰고 나서 닫아야 하는 이유도 명확히 이해할 수 있었다.

5. 느낀점

처음 쉘을 구현할 때 이 정도로 긴 길이의 코드를 작성해 본 적이 없어서, 전체 흐름을 어떻게 구현해야 할지 막막했다. 그렇지만 미리 차근차근 기능을 하나씩 쪼개서 구현해 나가다 보니 완성할 수 있었다. 이 과제를 통해 다양한 시스템콜 함수들을 배울 수 있었고 내부 명령어와 외부 명령어의 차이를 이해할 수 있었다. 또한, 단순히 돌아가는 코드만 만드는 것이 아니라 wait을 이용한 좀비 프로세스 처리, close를 통한 블로킹 방지 등 보안과 관련된 문제를 고민한 것이 인상깊었다. 이번 과제를 통해 코드와 운영체계가 어떤 식으로 상호 작용하는 지 이해할 수 있었고, 나중에 리눅스를 깊게 파고 들 때 도움이 될 것 같다고 느꼈다.

예시 출력