UNIX 단순 셸

운영 체제 프로젝트 #1*

소프트웨어학부

2021 년 3 월 18 일

Unix Simple Shell

이 프로젝트는 사용자 명령을 수신하고 각 명령을 별도의 프로세스에서 실행하는 셸 인터페이스 역할을 하는 C 프로그램을 설계하는 것으로 구성됩니다. 구현에서는 입력 및 출력 리디렉션을 지원할 뿐만 아니라 명령 쌍 간의 IPC 형식도 지원합니다. 이 프로젝트를 완료하려면 유닉스 fork(), exec(), wait(), dup2() 및 pipe() 시스템 호출(system call)을 사용해야 하며 리눅스, 유닉스 또는 MacOS 시스템에서 완료할 수 있습니다.

Overview

셸 인터페이스는 사용자에게 프롬프트를 제공하고 그 후에 다음 명령을 입력합니다. 아래 예제는 프롬프트 ossh> 와 사용자의 다음 명령인 cat prog.c 를 보여줍니다 (이 명령은 UNIX cat 명령을 사용하여 터미널에 prog.c 파일을 표시합니다).

```
osh> cat prog.c
```

셸 인터페이스를 구현하는 한 가지 기법은 부모 프로세스가 먼저 사용자가 명령줄에 입력하는 내용(이 경우 cat prog.c)을 읽고 명령을 수행하는 별도의 자식 프로세스를 만드는 기법입니다. 별도로 지정하지 않는 한 상위 프로세스는 하위 프로세스가 종료될 때까지 기다렸다가 계속합니다. 그러나 UNIX 셸은 일반적으로 하위 프로세스를 백그라운드에서, 즉 동시에 실행할 수도 있습니다. 이를 위해 명령 끝에 앰퍼샌드(&)를 추가합니다. 따라서 위의 명령을 다음과 같이 다시 쓰면,

```
osh> cat prog.c &
```

상위 및 하위 프로세스가 동시에 실행됩니다. 별도의 자식 프로세스는 fork() 시스템 호출을 사용하여 생성되며, 사용자의 명령은 exec() family 의 시스템 호출 중 하나를 사용하여 실행됩니다. 명령줄 셸의 일반적인 작동을 제공하는 C 프로그램은 [그림 1]에 나와 있습니다. main() 함수는 프롬프트 osh>를 제공하고, 사용자의 입력을 읽은 후 수행할 단계를 표시합니다. main() 함수는 should_run 이 1 이면 계속 루프됩니다. 사용자가 프롬프트에서 exit 를 입력하면 프로그램이 should_run 이 0 으로 설정하고 종료됩니다.
이 프로젝트는 다음과 같은 여러 부분으로 구성됩니다.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80

int main(void)
{
char *args[MAX_LINE/2 + 1];
int should_run = 1;

        while (should_run) {
                printf("osh>");
                fflush(stdout);

                /**
                사용자 입력을 읽은 후 단계는 다음과 같습니다.
                (1) 자식 프로세스를 fork() 명령어를 통해 포크합니다.
                (2) 자식 프로세스가 execvp() 를 호출합니다.
                (3) & 명령이 포함되지 않는 한, 부모 프로세스는 wait() 를 호출합니다.
                */
        }

        return 0;
}
```

[그림 1] : 단순 쉘(Simple shell)의 개요

1. 하위 프로세스 생성 및 하위에서 명령 실행
2. 입력 및 출력 리디렉션 지원 추가
3. 상위 및 하위 프로세스가 파이프를 통해 통신하도록 허용

Excuting Command in a Child Process

첫 번째 작업은 그림 1 의 main() 함수를 수정하여, 하위 프로세스를 포크 처리하여 사용자가 지정한 명령을 실행하는 것이다. 이를 위해서는 사용자가 입력한 내용을 별도의 토큰에 파싱하여 넣고, 그 토큰을 문자열 배열([그림 1]의 args[])에 저장해야 합니다. 예를 들어 사용자가 osh> 프롬프트에서 ps -ael 명령을 입력하면 args 배열에 저장된 값은 다음과 같습니다.

osh> ps -ael 를 입력한 경우 :

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

이 args 배열은 execvp() 함수에 전달되며, 이 함수의 프로토타입은 :

```
execvp(char *command, char *params[])
```

입니다. 여기서 command 는 수행할 명령을 나타내며, params 에는 이 명령을 위한 매개변수들을 저장합니다. 이 프로젝트의 경우 execvp() 함수를

```
execvp(args[0], args)
```

로 호출해야만 합니다. (should be invoked as)
사용자가 부모 프로세스가 자식 프로세스가 종료될 때까지 대기하는지를 결정하기 위해 & 를 입력했는지 확인해야 하는 것을 명심하세요.

Redirecting Input and Output

그런 다음 > 및 < 리디렉션 연산자를 지원하도록 셸을 수정해야 합니다. 여기서 > 는 명령의 출력을 파일로 리디렉션하고 < 는 파일에서 명령으로 입력을 리디렉션합니다.
예를 들어 사용자의 입력이

```
osh> ls > outtxt
```

일 때, ls 명령의 출력이 out.txt 파일로 리디렉션됩니다. 마찬가지로, 입력도 다음과 같이 리디렉션 될 수 있습니다. 예를 들어 사용자의 입력이

```
osh> sort <in.txt
```

일때, in.txt 라는 파일은 정렬 명령(sort)에 대한 입력으로 사용됩니다. 입력과 출력의 리디렉션을 관리하는 데는 dup2() 함수를 사용해야 하는데, dup2() 함수는 기존 파일 디스크립터를 다른 파일 디스크립터로 복제합니다.

* file descriptor : 할당 받은 파일을 대표하는 값 / POSIX 표준에서는 표준입력(0), 표준출력(1), 표준에러(2)를 기본적으로 할당한다고 함. 참고만하셈

예를 들어, fd 가 out.txt 라는 파일 출력에 대한 파일 디스크립터인 경우 다음 호출

```
dup2(fd, STDOUT_FILENO);
```

는 fd 라는 파일 디스크립터를 터미널의 표준 출력에 복제합니다. 이는 표준 출력에 대한 어떠한 쓰기라도 실제로 out.txt 로 전송된다는 것을 의미합니다. 당신은 명령에 하나의 input 또는 하나의 output 리디렉션이 포함되고, 둘 다 포함되지 않는다고 가정할 수 있습니다. 즉, sort <in.txt> out.txt 와 같은 명령을 고려하지 않아도 됩니다.

Communication via a Pipe

당신의 셸에 대한 마지막 변경은 파이프를 사용하여 한 명령의 출력이 다른 명령의 입력으로 작용할 수 있도록 하는 것입니다. 예를 들어 다음 명령

```
osh> ls -l | less
```

은 ls -l 명령의 출력을 less 명령의 입력으로 사용합니다. ls 와 less 명령어 모두 별도의 프로세스로 실행되며 UNIX pipe() 기능을 사용하여 통신합니다. 이러한 개별 프로세스를 생성하는 가장 쉬운 방법은 상위 프로세스가 (예제에서 ls -l 을 실행하는)하위 프로세스를 생성하도록 하는 것입니다. 또한 이 하위 프로세스는 (예제에서 less 를 실행하는)다른 하위 프로세스를 생성하고, 그 프로세스(ls)와 생성된 하위 프로세스(less) 사이에 파이프를 설정합니다. 파이프 기능을 구현하려면 앞 절에서 설명한 dup2() 기능을 사용해야 합니다. 마지막으로 비록 여러 명령을 여러 파이프를 사용하여 함께 연결할 수 있지만, 여기에서는 명령에 파이프 문자가 하나만 포함되고 어떠한 리디렉션 연산자와도 결합되지 않는다고 가정할 수 있습니다.

명령어 파싱과 오류 처리

이번 프로젝트는 프로세스 생성과 프로세스간 통신을 이해하는 것이 주요 목적입니다. 입력된 명령어를 파싱해서 형식에 어긋나는 명령어 오류를 유연하게 처리하는 것은 필요하지만, 이번 주제와는 별개의 문제이므로 여기서는 고려하지 않겠습니다. 문제를 간단하게 하기 위해서 Simple Shell 사용자는 다음 형식의 명령어만 올바르게 사용한다고 가정하고 명령어를 스캔합니다.

- 명령어 or 명령어 &
- 명령어 > 파일명 or 명령어 > 파일명 &

- 명령어 < 파일명 or 명령어 < 파일명 &
- 명령어 1 | 명령어 2 or 명령어 1 | 명령어 2 &
- exit

만일 앞 형식을 따르지 않은 명령어가 입력되면 전체를 무시하거나 또는 인식된 부분까지만 처리하고 나머지는 버린다고 가정합니다. 다만 명령어는 옵션을 포함할 수 있어야 합니다.

Best Coding Practices

바람직하지 않은 코딩 스타일에 대한 감점이 있습니다.

1. 프로그램의 가독성 (들여쓰기와 형식의 일관성 등)
2. 코드에대한설명(주석등)
3. 알기쉬운변수명사용과불필요한상수사용회피등
4. 프로그램의 확장성
5. 프로그램의 효율성
6. 기타 바람직한 프로그래밍 원칙에 위배되는 경우 (https://en.wikipedia.org/wiki/Best_coding_practices)

제출물

Simple Shell 이 잘 설계되고 구현되었다는 것을 보여주는 자료를 각자가 판단하여 PDF 로 묶어서

이름_학번_PROJ1.pdf 로 제출한다. 여기에는 다음과 같은 것이 반드시 포함되어야 한다.

- 본인이 설계한 Simple Shell 알고리즘 (1 쪽 분량)
- 프로그램 소스파일
- 컴파일과정을보여주는화면캡처
- 위에서 언급한 명령어 형식을 최소한 한 번씩을 포함하여 여러가지 명령어를 실행한 결과물과 그에 대한 간단한 설명 등

파이팅!