Operating Systems
Project #1*

소프트웨어학부

2021년 3월 18일

UNIX Simple Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX `fork()`, `exec()`, `wait()`, `dup2()`, and `pipe()` system calls and can be completed on any Linux, UNIX, or macOS system.

Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

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

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (`&`) at the end of the command. Thus, if we rewrite the above command as

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

the parent and child processes will run concurrently. The separate child process is created using the `fork()` system call, and the user's command is executed using one of the system calls in the `exec()` family.

A C program that provides the general operations of a command-line shell is supplied in Figure 1. The `main()` function presents the prompt `osh>` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters `exit` at the prompt, your program will set `should_run` to 0 and terminate.

This project is organized into several parts:

1. Creating the child process and executing the command in the child

*이 과제는 Operating System Concepts 10판 3장 프로그래밍 프로젝트의 일부이다.

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

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

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

      /**
       * After reading user input, the steps are:
       * (1) fork a child process using fork()
       * (2) the child process will invoke execvp()
       * (3) parent will invoke wait() unless command included &
       */
   }

    return 0;
}
```

Figure 1: Outline of simple shell

2. Adding support of input and output redirection
3. Allowing the parent and child processes to communicate via a pipe

Executing Command in a Child Proces

The first task is to modify the `main()` function in Figure 1 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (`args` in Figure 1). For example, if the user enters the command `ps -ael` at the `osh>` prompt, the values stored in the `args` array are:

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

This `args` array will be passed to the `execvp()` function, which has the following prototype:

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

Here, `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0],`

`args)`. Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

Redirecting Input and Output

Your shell should then be modified to support the '>' and '<' redirection operators, where '>' redirects the output of a command to a file and '<' redirects the input to a command from a file. For example, if a user enters

```
osh> ls > out.txt
```

the output from the `ls` command will be redirected to the file `out.txt`. Similarly, input can be redirected as well. For example, if the user enters

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

the file `in.txt` will serve as input to the `sort` command. Managing the redirection of both input and output will involve using the `dup2()` function, which duplicates an existing file descriptor to another file descriptor. For example, if `fd` is a file descriptor to the file `out.txt`, the call

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

duplicates `fd` to standard output (the terminal). This means that any writes to standard output will in fact be sent to the `out.txt` file. You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as `sort < in.txt > out.txt`.

Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

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

has the output of the command `ls -l` serve as the input to the `less` command. Both the `ls` and `less` commands will run as separate processes and will communicate using the UNIX `pipe()` function. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute `ls -l`). This child will also create another child process (which will execute `less`) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the `dup2()` function as described in the previous section. Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.

명령어 파싱과 오류 처리

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

- 명령어 or 명령어 &

- 명령어 > 파일명 or 명령어 > 파일명 &
- 명령어 < 파일명 or 명령어 < 파일명 &
- 명령어1 | 명령어2 or 명령어1 | 명령어2 &
- `exit`

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

Best Coding Practices

바람직하지 않은 코딩 스타일에 대한 감점이 있다.

1. 프로그램의 가독성 (들여쓰기와 형식의 일관성 등)
2. 코드에 대한 설명 (주석 등)
3. 알기 쉬운 변수명 사용과 불필요한 상수 사용 회피 등
4. 프로그램의 확장성
5. 프로그램의 효율성
6. 기타 바람직한 프로그래밍 원칙에 위배되는 경우
   (https://en.wikipedia.org/wiki/Best_coding_practices)

제출물

Simple Shell이 잘 설계되고 구현되었다는 것을 보여주는 자료를 각자가 판단하여 PDF로 묶어서 `이름_학번_PROJ1.pdf`로 제출한다. 여기에는 다음과 같은 것이 반드시 포함되어야 한다.

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

HK