Programming Assignment #3

Due : 15th Nov. (Thur),5:59 PM

1. Introduction

이번 과제에선 리눅스 환경을 위한 셸인 swsh를 제작한다. 이 과제의 주요 목표는 수업 전반기에 배운 주제인 파일, 프로세스, 시그널, IPC를 다루는 프로그램을 제작하는 것이다.

이번 과제는 여타 과목의 중간고사와 비슷한 비중을 지닌다. 이 과제를 제출하지 않으면 큰 불이익을 받을 수 있다.

2. Problem specification

셸은 사용자의 1) 입력을 받아, 2) 해석하고, 3) 명령을 실행하는 프로그램이다. swsh은 아주 작은 기능만 구현한 셸로 정해진 특정 명령어를 실행시킬 수 있으며, input/output redirection, pipeline 기능을 지원한다.

다음 표는, 사용자의 입력이 될 수 있는 문자열을 BNF(Backus-Naur Form)와 유사한 형태로 기술한 것이다. (이하, <표>라고 표현)

```
input         ::=   commands
                    commands | input
commands      ::=   command
                    command < [filename]
                    command > [filename]
                    command >> [filename]
                    command < [filename] > [filename]
command       ::=   cmd_type1 [options/arguments]
                    cmd_type2 [options/arguments]
                    cmd_type3 [arguments]
                    cmd_type4
cmd_type1     ::=   { ls, man, grep, sort, awk, bc }
                    path
path          ::=   { pathname with leading "./" }
cmd_type2     ::=   { head, tail, cat, cp }
cmd_type3     ::=   { mv, rm, cd }
cmd_type4     ::=   { pwd, exit }
```

<표>에서 사용한 표현의 일부는 영단어 그대로의 의미로 해석하면 된다. 간단히 설명하면, filename은 임의의 [파일명]을 뜻하고, options/arguments는 해당 프로그램의 [옵션과 인자]

를 뜻하는 표현이다. 그리고, <표>에서 사용된 일부 기호는 bash에서 쓰이는 기호와 동일하다.

|	Pipeline
<	Input redirection
>	Output redirection
>>	Output redirection (appending)

각 기호의 기능은 Pipe 수업 시간에서 배운 것과 같고, 간략하게 다시 설명하면 다음과 같다.

- Pipeline: 앞 프로세스의 표준 출력을 뒤 프로세스의 표준 입력으로 **파이프**를 통해 연결
- Input redirection: 해당 프로세스의 표준 입력(fd 0)을 [파일명] 파일로 대체
- Output redirection: 해당 프로세스의 표준 출력(fd 1)을 [파일명] 파일로 대체
  - > 으로 redirect 되었을 경우, 해당 파일을 초기화하고 새로 쓴다.
  - >> 으로 redirect 되었을 경우, 해당 파일의 뒤에서부터 덧붙인다.

또한 구현의 편의를 위해, **<표>의 각 키워드 사이는 무조건 한 칸의 space로 구분된다**고 가정한다. 만약 입력이 command < filename의 경우, [명령어] 와 '<' 문자 사이엔 한 칸의 space, '<' 와 [파일 이름] 사이에도 한 칸의 space로 구성된다.

2.1. Interpreting inputs

사용자는 200바이트 이내의 문자열을 입력한다. 만약 사용자의 입력이 <표>의 input으로부터 도출(derive)할 수 있는 문자열이면, 유효한 입력이다. 예를 들면, 다음 명령의 경우,

```
cd dir3
```

- input ➔ commands ➔ command ➔ cmd_type3 [arguments]
  - cmd_type3 ➔ cd
  - [arguments] ➔ dir3

순으로 해석될 수 있기 때문에 유효한 입력이다.

다음 명령 또한 input으로부터 유도될 수 있기 때문에 유효하다.

```
ls -al /etc | sort -r > ls_sorted.txt
```

하지만 다음 명령은, less라는 프로그램을 input으로부터 유도할 수 없기 때문에 유효하지 않다.

```
ls -al /etc | less
```

<표>의 path 경우, 셸에서 현재 디렉토리 내의 프로그램을 실행하는 것을 의미하는 표현이다. 현재 디렉토리의 a.out 파일을 실행시킨 땐 통상적으로 다음 명령을 사용한다.

```
./a.out
```

즉, [명령어]가 "./" 로 시작하는 문자열(경로)일 경우 cmd_type1 에서 유도될 수 있다고 본다. (과제의 단순화를 위해, 절대경로 등의 방식은 고려하지 않는다. 또한, 홈 디렉토리를 의미하는 '~' 또한 사용하지 않는다.)

입력이 유효할 경우 **입력을 평가**하고, **유효하지 않은 경우** 다음과 같은 에러 메시지를 **표준 에러** (fd 2) 에 출력한다.

```
swsh: Command not found
```

2.2. Evaluating inputs

입력값을 평가하는 과정을 구현하는 방식은 사람마다 다르겠지만, 일반적으로 다음과 같은 방식을 취할 수 있다.

1) 명령어의 개수를 파악 (pipeline 등으로 명령어들이 연결된 경우)
   (각 명령에 대해)
2) Input/output redirection이 사용되었는지 파악
3) 명령어 종류를 분석하여,
   A. 직접 구현한 경우, (필요하다면) 자식 프로세스를 생성하여 해당 루틴을 호출
   B. 구현하지 않은 경우, 자식 프로세스를 생성하여 해당 프로그램을 로드하여 실행
4) 자식 프로세스를 생성했을 경우, 종료될 때까지 기다림

2.2.1. Input/output redirection

셸의 명령으로 redirection이 사용될 경우, 사용자가 명시한 파일이 존재하는지, 혹은 존재하더라도 해당 파일이 정상적인 파일인지(디렉토리가 아닌지), 해당 파일을 사용할 권한이 있는지 등을 확인할 필요가 있다. 하지만, swsh에선 아래 다음과 같은 경우만 파악한다.

만약 input redirection으로 명시된 파일이 파일이 존재한다면, 무조건 읽기 권한이 있는 정상적인 파일이고, 존재하지 않으면 다음과 같은 문자열을 출력한다.

명령이 파이프로 연결되었다고 생각하자.

```
A | B | C | D
```

Input redirection는 제일 앞의 A 프로세스에서만 존재할 수 있고, output redirection는 제일 뒤의 D 프로세스에서만 존재할 수 있다. (B와 C에서 redirection이 존재하는지 검사할 필요가 없음)

```
A < file1 | B | C | D > file2
```

2.2.2. command

swsh에서 처리할 명령은 크게 4가지 종류로 구분된다. 간단히 설명하자면,

- cmd_type1: 해당 프로그램을 로드(fork - exec*)해서 실행해야 하는 프로그램
- cmd_type2: 로드하거나, 구현할 지 선택할 수 있는 프로그램
- cmd_type3: 구현해야 하는 프로그램 (인자가 있음)
- cmd_type4: 구현해야 하는 프로그램 (인자가 없음)

cmd_type1 에 해당하는 명령은, 옵션이나 인자를 추출해 자식 프로세스를 생성해 exec계열 함수를 통해 해당 바이너리를 로드하여 실행한다.
cmd_type2 의 경우, 스스로 구현하던지 이미 존재하는 exec를 통해 실행할 지 선택할 수 있다. **당연히, 직접 구현한 경우 높은 평가를 받는다**. 직접 구현하더라도, 파이프 등을 통해서 다른 프로세스와 연결될 수 있다는 점을 고려한다.
cmd_type3 와 cmd_type4 에 해당하는 명령은 직접 구현해야 한다.

구현해야 하는 명령에 대해선 아래 3장에서 각 명령의 기능을 정의하고 필요한 시스템 콜/라이브러리 함수를 언급할 것이다.

2.3. Process group

swsh가 새로운 자식 프로세스를 생성할 경우, 해당 프로세스를 새로운 process group에 속하도록 한다. (해당 자식 프로세스의 pgid는 자기 pid와 일치하도록 만든다)
만약, 파이프의 사용으로 인해 두 개 이상의 프로세스를 생성할 경우, 모든 프로세스는 제일 먼저 생성한 프로세스의 그룹에 속하도록 한다. (프로세스의 pgid는 제일 먼저 생성한 자식 프로세스의 pid와 일치해야 함)

2.4. Reaping child processes

swsh가 자식 프로세스를 생성했다면, wait계열 시스템 콜를 이용해 자식 프로세스를 확인하고 제거하여 좀비 프로세스가 생기지 않도록 한다.

프로세스가 실행 도중 Ctrl+Z 키를 통해 생성하는 SIGTSTP 시그널을 수신하면, 일반적인 경우 프로세스 실행을 멈추고 중단된다. 동시에 부모 프로세스는 SIGCHLD 시그널을 수신하고, 이 때 부모가 다음과 같이 waitpid 를 호출해서 얻는 status 값에 WIFSTOPPED 매크로 함수를 실행하면 true가 반환된다. (waitpid 시스템 콜의 옵션 WUNTRACED)

```
waitpid(-1, &status, WNOHANG | WUNTRACED)
```

(참고: $ man 2 waitpid)

만약 자식 프로세스가 중지되었을 경우 (WIFSTOPPPED가 true일 때), 해당 자식이 속한 **프로세스 그룹**에 SIGKILL 시그널을 보내 모든 연관 프로세스를 강제로 종료시킨다.

2.5. Signals

swsh는 SIGINT, SIGTSTP 시그널을 수신해도 종료되지 않는다.

3. Programs

구현할 프로그램은 기존 프로그램과 동일한 작업을 수행하지만, 쉽게 구현할 수 있도록 단순화된 버전이다.

3.1. head

SYNOPSYS

head [OPTION] file

DESCRIPTION

file 파일의 위에서부터 10 라인을 표준 출력으로 출력한다.

-n K

10 라인 대신 K 라인을 출력한다.

file 파일이 존재하지 않는 경우는 없다.

3.2. tail

SYNOPSYS

tail [OPTION] file

DESCRIPTION

file 파일의 아래에서부터 10 라인을 표준 출력으로 출력한다.

-n K

10 라인 대신 K 라인을 출력한다.

`file` 파일이 존재하지 않는 경우는 없다.

3.3. `cat`

SYNOPSYS

`cat file`

DESCRIPTION

`file` 파일을 표준 출력으로 출력한다.

`file` 파일이 존재하지 않는 경우는 없다.

3.4. `cp`

SYNOPSYS

`cp file1 file2`

DESCRIPTION

`file1` 파일의 복사본 `file2` 파일을 만든다.

`file1` 파일이 존재하지 않는 경우는 없다. `file2` 파일을 덮어쓸 수 없는 경우는 없다.

3.5. `mv`

SYNOPSYS

`mv file1 file2`

DESCRIPTION

`file1` 파일의 이름을 `file2`로 바꾼다.

SEE ALSO

`rename(2)`

3.6. `rm`

SYNOPSYS

`rm file`

DESCRIPTION

`file` 파일을 삭제한다.

SEE ALSO

`unlink(2)`

3.7. `cd`

SYNOPSYS

`cd dir`

DESCRIPTION

현재 working directory를 `dir`로 변경한다.

SEE ALSO

chdir(2)

3.8. pwd

SYNOPSYS

pwd

DESCRIPTION

현재 working directory를 표준 출력으로 출력한다.

SEE ALSO

getcwd(3)

3.9. exit

SYNOPSYS

exit [NUM]

DESCRIPTION

표준 에러로 exit란 문자열을 출력한 뒤, swsh를 종료한다.

NUM이 명시되었을 경우 프로그램의 종료 값으로 NUM을, 아닐 경우 0을 반환한다.

SEE ALSO

exit(3)

3.10. Errors

mv, rm, cd 명령 수행 중 에러가 발생한 경우, 에러의 종류에 따라 아래 표에 명시된 메시지를 다음과 같이 조합하여 표준 에러로 출력한다.

EACCESS	Permission denied
EISDIR	Is a directory
ENOENT	No such file or directory
ENOTDIR	Not a directory
EPERM	Permission denied
Other errors	Error occurred: <ERRNO> (에러 번호 출력)

```
command: ERROR_MESSAGE
e.g.)
mv: Permission denied
cd: Not a directory
```

4. Background

4.1. strace

어떤 프로그램을 실행할 때 발생하는 시스템 콜을 추적하는 프로그램이다. 간단히,

`$ strace ls`
명령을 수행하면, `ls` 명령을 실행하기 위해 사용된 시스템 콜의 내역을 볼 수 있다.

또한, `-p` 옵션을 사용하면, 현재 실행 중인 프로세스의 시스템 콜을 확인할 수 있다. `bash`에서, 현재 자신의 pid를 보는 명령은 `echo $$` 이다. 새로운 셸을 띄워, 이렇게 확인한 `bash` 프로세스의 pid를 `strace -p` 옵션으로 넘길 경우, 셸을 동작시키기 위해 사용되는 시스템 콜을 확인할 수 있다.

4.2. whereis // which

어떤 파일의 위치를 알고 싶을 때, whereis나 which 명령을 사용할 수 있다.

```
$ which ls
/bin/ls
```

5. Restrictions

- 과제 구현을 위해 여태까지 배운 리눅스 시스템 콜/라이브러리 함수를 이용한다.
- 어떤 자원을 동적으로 할당 받았다면, 프로그램 종료 전에 반드시 해제해야 한다.
  - 여기서 자원이란 파일, 메모리, 자식 프로세스를 뜻한다.

6. Hand in instruction

- 작성한 코드 상단의 주석에 이름과 학번을 작성한다.
- **`make` 명령을 통해 프로그램이 제대로 만들어지는지 확인한다.**
  - **제출한 코드가 컴파일되지 않을 경우, 경우에 따라서 해당 제출은 채점되지 않거나 심한 페널티를 받게 될 것이다.**
- 프로그램 코드를 "학번.tar.gz" 형태로 압축한다.
- 본 과제 수행 시 구현 방법과 디자인을 설명하는 보고서를 PDF 포맷으로 작성하여 "학번.pdf" 이란 이름을 붙인다

7. Logistics

- 본 과제는 혼자 수행한다.
- 과제 제출 시간은 메일 도착 시간을 기준으로 하며, 과제를 지연 제출하면 **기한 직후**엔 10%가 감점되고, **매 24시간마다** 10%씩 추가로 감점된다.
- **Copy 할 경우, 연구실 자체 규정에 따라 처벌하며, 상당한 불이익이 있을 수 있다.**

Have fun!