myshell_202350024_서기

과제

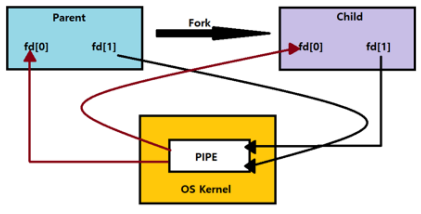
- 조건
  1. 현재 디렉토리 명을 쉘에 표시
  2. 일반적인 리눅스 bash 쉘의 사용자 인터페이스 형태를 띌 것
  3. cd, pwd 명령어를 반드시 구현 (상대경로 사용 가능해야 함, 단순히 exec을 사용해서는 안됨, 나머지 명령어는 exec 시스템콜을 이용해 구현해도 됨)
  4. 파이프라인 (멀티 파이프라인도 가능해야 함)
  5. 다중 명령어 (; , &&, ||)
  6. 백그라운드 실행(&)
  7. exit 입력 시 프로그램 종료
  8. 추가 명령어 및 옵션 직접 구현 시 가산점 부여
  9. 좀비 프로세스나 오버플로우 등 보안위협이 되는 요소들에 대한 고려가 있을 경우 가산점 부여 (보고서에 기록) ⇒ 고민의 흔적 자체를 긍정적으로 평가

리눅스 쉘 구현

- fork(), exec()
- pipe(), dup2()
- waitpid()
- 효율적인 구현 -> 재귀호출

리눅스 쉘의 동작원리 - IPC

- OS가 제공하는 IPC 은 여러 가지가 있음
- 그러한 Support 중 하나가 PIPE
  - 파이프는 커널 영역에 생성, 파이프를 생성한 프로세스는 파일 디스크립터를 통해 파이프에 접근 가능



파일 디스크립터: 리눅스 혹은 유닉스 계열의 시스템에서 프로세스(process)가 파일(file)을 다룰 때 사용하는 개념으로, 프로세스에서 특정 파일에 접근할 때 사용하는 추상적인 값이다.

<기본적으로 할당되는 파일 디스크립터>
0: 표준 입력
1: 표준 출력
2: 표준 에러

0. cd, pwd 구현했다고 가정

1. 명령어 입력받기 -> 명령어를 (| , ; , &&, || , &) 구분자로 전부 split하기

2. split한 명령어가 쭉 있을텐데, 어차피 0번 인덱스만 명령어고 space로 구분되어있기 때문에, 다시 한번 split 가능

   2-1. 구분자도 따로 저장해야됨.

   2-2. space도 기준으로 구분하려고 했는데, `cat a.txt|grep a` 처럼 space 가 생략되어도 작동해야되기 때문에 고려

3. ex) cat a.txt 우선 cat을 인식을 했을테니까, fork()로 자식프로세스 생성, exec 계열 함수를 이용해 프로그램 실행. 자식 프로세스는 이를 실행하고 출력 (근데 만약 뒤에 구분자가
`|` 라면 출력이 아니라 IPC로 값을 넘겨 주어야 하기 때문에 pipe 및 dup2 사용)

4. 출력을 마치고 나면 exit함수로 자식 프로세스를 종료하고, 자식 프로세스가 종료하기 전에 wait 하고 있었던 부모 프로세스의 wait 해제
5. 파이프는 따로 구현하는게 편할 것 같은데, 다중 파이프는 재귀함수로 구현하는 방법 생각
   a. 구분자를 저장해놓은 변수에서 연속된 파이프의 개수 세고(명령어의 개수 = 파이프 개수 + 1)
   b. 명령어 개수가 1개가 될 때 까지 재귀함수
6. 일회성 코드가 아니라 명령어를 계속 받을 수 있어야 하기 때문에 반복문 사용

7. strcmp(line, exit) == 0 → return
8. pwd, cd 구현

   8-1. cd가 파이프에서 실행될때. cd 이전 명령어를 전부 무시하고 cd다음 파이프부터 처리하도록 구현.
9. 초기 인터페이스, `abcd@user:pwd$` 구현

1. 입력받기

1. scanf는 공백을 입력의 끝으로 인식하기 때문에 적절하지 않다.
2. `getline()` 함수 : 공백 포함한 전체 줄을 동적으로 받아오기 때문에 적당하며 메모리 관리에도 효율적
3. `getline()` 함수는 동적 할당이기 때문에 반드시 free를 해주어야 메모리 누수를 막을 수 있다.

2. split

1. 구분자 split
2. 동적할당 해서 split 하려고 했는데 너무 복잡해져서 2차원배열을 사용하기로 함. command[256][256]; 마지막에 NULL만 잘 넣어주면 됨. 할 때마다 memset으로 초기화 해서 재사용.
3. excvp에 값을 넣어주려면 공백 기준으로 split해야하는데, 여기에는 strtok 함수를 사용.

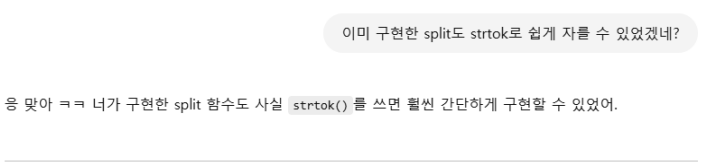
```
char *strtok(char *str, const char *delim);
```

주어진 문자열을 구분자(delimiter)를 기준으로 여러 개의 토큰으로 나누는 함수.

`str` : 토큰을 분리할 문자열. 첫번째 호출에서는 문자열을 전달하고, 이후에 NULL을 전달하여 이전 상태에 이어서 토큰을 분리

`delim` : 구분자로 사용할 문자열. 여기서는 스페이스를 기준으로 문자열을 나눔.

나눈 문자열을 반환하며, 더 이상 토큰이 없다면 NULL 반환



3. fork, waitpid, exec 시스템

1. 가장 작은 프로세스 작동 단위 구현
2. 명령어 한 줄과 다음 separator을 입력 받음

`WIFEEXITED(status)`

: 자식 프로세스가 정상적으로 종료되었는지 판단

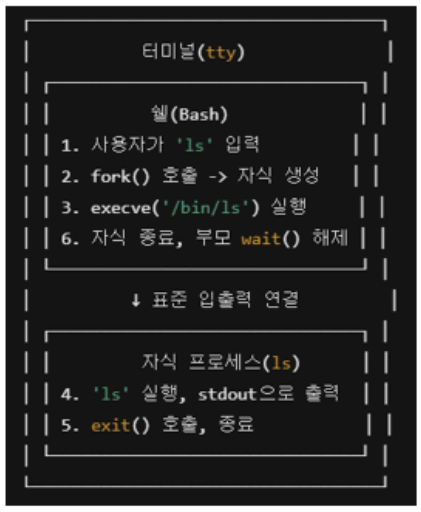
- true면 정상 종료
- flase면 비정상 종료

`WEXITSTATUS(status)`

- `WIFEXITED(status)`가 true일 때만 사용 가능
- 자식 프로세스가 exit 또는 return 으로 종료했을 때, 그 **종료 코드(int)** 를 얻는 함수
- 0 이면 정상 종료 , 아니면 오류

리눅스 쉘의 동작원리 - 프로세스

- 생성한 자식 프로세스에서 명령어를 실행하는 방법
  - exec 계열 함수: 현재 실행되고 있는 프로세스를 다른 프로세스로 대신하여 새로운 프로세스를 실행
  - 프로세스를 대체하는 것이지 새로 생성하는 것이 아니므로 fork와는 다른 개념
  - 보통 fork로 자식프로세스를 생성하고 exec으로 특정 실행파일을 실행하는 것이 일반적



입력받은 명령어를 이미 다 split했기 때문에, 그대로 execvp함수로 값을 넣어주면 된다.

```
int execvp(const char *file, char *const argv[]);
```

`file` : 실행할 프로그램의 경로.

`argv`: 실행할 프로그램에 전달할 인자의 배열. 배열의 첫 번째 요소는 실행할 프로그램의 이름, 배열의 마지막 요소는 `NULL`

`execvp`는 성공하면 반환하지 않고, 프로그램을 새로운 프로그램으로 교체

`fork()` 를 통해 자식 프로세스를 만들고, 자식 프로세스가 명령어를 실행 및 출력

부모는 기다렸다가 자식이 종료되면 다시 wait해제 후 작동

4. 다중명령어 (; , &&, ||) 구현

1. sep == 2 (;)

   다음 명령어 그냥 실행
2. sep == 3 (&&)

   현재 명령어가 실행 되면 다음 명령어 실행
3. sep == 4 (||)

   현재 명령어가 실행 안되면 다음 명령어 실행

명령어가 제대로 실행되었는지 안되었는지 확인하기 위해, run_command 함수와 run_pipe 함수에서 리턴값을 받아온다.

제대로 실행되었다면 리턴값은 0, 아닐 경우 리턴값은 1

5. 파이프 구현 (다중 파이프 까지 → 아마 재귀함수로?)

https://velog.io/@hidaehyunlee/minishell-5.-파이프Pipe-처리

https://architectophile.tistory.com/9

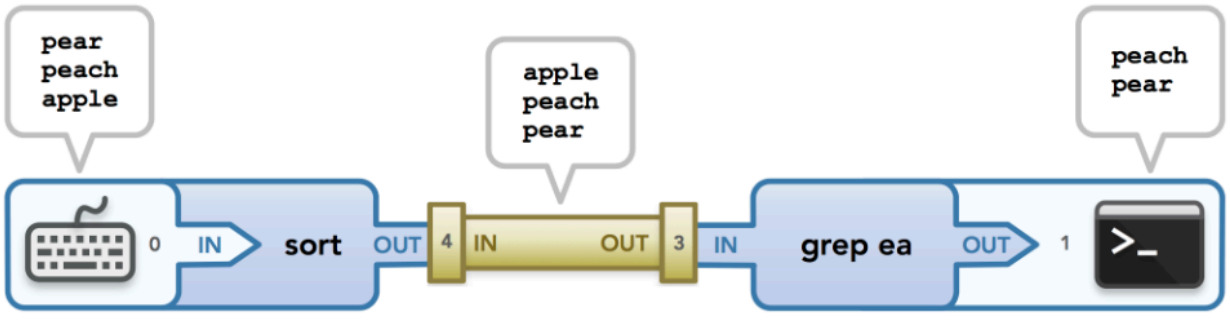
1. dup2()함수

```
# include <unistd.h>

int dup2( int fd, int fd2 );
```

파일 식별자를 복제해 fd2를 fd1으로 변경.

ex) `int dup2(fd, stdout);` → 모든 출력이 fd로.



- 그리고 사용할 fd, 즉 파이프를 연결할 때는
  - **fd[1] 에 쓰고**
  - **fd[0] 으로 읽는다**

- ***첫 번째 요소는*** `자식 프로세스`에서 `입력(input)`을 ***읽는데(read) 사용되고, 두 번째 요소는*** `부모 프로세스`로부터 `출력(output)`을 ***쓰는데(write) 사용된다.***

- **pipe는 반드시** `fork()` **이전에 만들어야 한다.**
- 그래야 자식이 유효한 파일 디스크립터를 갖게 된다.

파이프만 실행시키는 함수도 제대로 실행되었는지 아닌지를 반환해야 한다.

ex) `ls - l | grep text.txt | wc -l && ls -l`

```
$ a | ls
makefile  myshell.c  myshell.o          myshell_interface.o  myshell_utils.o  split_line.o
myshell   myshell.h  myshell_interface.c  myshell_utils.c      split_line.c     test.txt
a: command not found
```

```
$ ls | a
a: command not found
```

파이프에서 잘못된 명령어가 들어왔을 경우 그냥 실패 오류를 띄우고 실패처리 하기로 했다.

```
kseo@user:/mnt/c/Users/kyseo/Desktop/cykor/myshell$ ./myshell
kseo@user:/mnt/c/Users/kyseo/Desktop/cykor/myshell$ a | ls
exec fail 2: No such file or directory
kseo@user:/mnt/c/Users/kyseo/Desktop/cykor/myshell$ ls | a
exec fail 1: No such file or directory
kseo@user:/mnt/c/Users/kyseo/Desktop/cykor/myshell$
```

6. 반복문 구현

일회성으로 사용하는 코드가 아니라, 명령어가 들어올 때마다 처리를 해야된다.

while문으로 구현하려고 했는데, 파이프가 값으로 들어오면 고장이 난다.

`getline` 함수에서 값을 stdin으로 받는데, 파이프 구현 함수에서 이 값이 부모 프로세스에서 초기화되지 않고 남아있기 때문에 다음 반복에서 이 값이 그대로 들어가면서 무한반복이 된다.

`getline()` 함수가 line을 동적할당 하기 때문에, free를 반복문마다 해주어서 메모리 누수도 막고, 함수가 무한반복되지 않도록 수정.

`run_pipe` 함수에서 `strtok` 를 이용해 파싱을 할때, 기존의 문자열의 스페이스 대신에 널문자로 바뀐다. 그래서 재귀함수가 실행되면 `grep txt` 같은 명령어에서 `grep` 만 인식하는 문제점이 생긴다. `tmp_command` 변수를 만들어 명령어를 복제한 후 실행하여 문제 해결.

7. pwd 구현

```
char * getcwd (char *buffer, size_t size)
```

현재 작업 디렉토리 경로 저장

`char * buf` : 현재 작업 디렉토리 경로를 저장할 버퍼의 포인터

`size_t size` : 버퍼의 크기(보통 `PATH_MAX`)

8. cd 구현

```
#include <unistd.h>

int chdir(const char *path);
```

현재 프로세스의 작업 디렉토리를 변경하는 함수

상태를 변경해야 하기 때문에 자식 프로세스가 아니라 부모 프로세스에서 실행해야 한다.

```
$ pwd | cd ..
   → 아무 값도 출력되지 않으며, 디렉토리 또한 바뀌지 않음.
$ cd.. | pwd
   → pwd 명령어 실행.
```

cd 명령어는 파이프에서는 정상적으로 작동하지 않는다.

cd 명령어가 파이프에 있는지 검사하고, cd 다음 명령어부터 처리

cd 명령어가 다중 파이프의 제일 뒤에 있을 경우, 파이프가 제대로 실행된 명령어인 것처럼 작동한다. 그렇기 때문에 `pwd | cd .. || ls` 같은 경우 아무 값도 출력되지 않으므로 예외처리.

```
cd .. a
-bash: cd: too many arguments
```

cd 명령어에는 단 하나의 인자만 전달되어야 한다. 인자가 두개 이상일경우 다음과 같이 처리되며,

```
 cd /a
-bash: cd: /a: No such file or directory
```

다음처럼 없는 디렉토리로 이동하려는 경우에도 오류가 발생한다.

위 두 상황의 경우에는 예외처리를 했다.

```
cd .. a | ls
-bash: cd: too many arguments
makefile   myshell.h          myshell_interface.o  split_line.c
myshell    myshell.o          myshell_utils.c      split_line.o
myshell.c  myshell_in-πterface.c  myshell_utils.o      test.txt
```

다음처럼 cd에 잘못된 인자 전달 + 파이프 조합은 ls 명령어부터만 실행하는 것으로 처리했다. 애초에 파이프에 cd명령어를 넣는것부터가 의미 없는 행위이기 때문에 따로 예외처리를 하지 않았다.

9. 인터페이스 구현

```
char *getenv(const char *name);
```

`name` : 환경변수의 이름을 나타내는 문자열. user의 이름을 불러올때 사용

```
int gethostname(char *name, size_t len);
```

`name`: 호스트 이름을 저장할 문자열.

`len`: `name` 버퍼의 크기.