유닉스 프로그래밍 쉘구현 프로젝트 과제#3



컴퓨터공학과 3학년

12141600 이인용

제출일 2018-12-09

-1주차 과제 요구 사항

1. Cd 명령이 제대로 먹히지 않는 버그를 수정

조교님이 주신 c파일에서는 fork와 exec로만 명령어를 구현한다. 이렇게 했을 때 어떤 문제점이 생기고 Cd명령어가 제대로 작동하게 고쳐라

1. Exit 명령을 구현

exit이라고 명령어 부분에 입력했을 때 쉘이 종료될 수 있도록 기능을 추가하도록 한다.

1. 백그라운드 실행을 구현

명령어 부분의 마지막에 &를 쳤을 때 그 명령어를 백그라운드 명령으로 인식하고 백그라운드로 처리를 한다.

1. 좀비 프로세스가 생성되는 이유

백그라운드로 명령을 했을 때 좀비 프로세스가 생성되었다. 그 이유를 서술하라

-2주차 과제 요구 사항

1. SIGCHLD로 자식 프로세스 wait()시 프로세스가 온전하게 수행되도록 구현

좀비프로세스가 계속 남아있는 문제를 해결하라

2. SIGINT,SIGQUIT 사용시 쉘이 종료되지 않도록 ,Foreground 프로세스 실행시 SIGINT를

받으면 프로세스가 끝나는 것을 구현

쉘이아닌 자식 process들이 sigint에 dfl기능으로 작동시켜라

-3주차 과제 요구 사항

1. 리다이렉션 구현

2. 파이프와 리다이렉션 같이 구동하게 구현

3. 파이프 리다이렉션 백그라운드로 구현

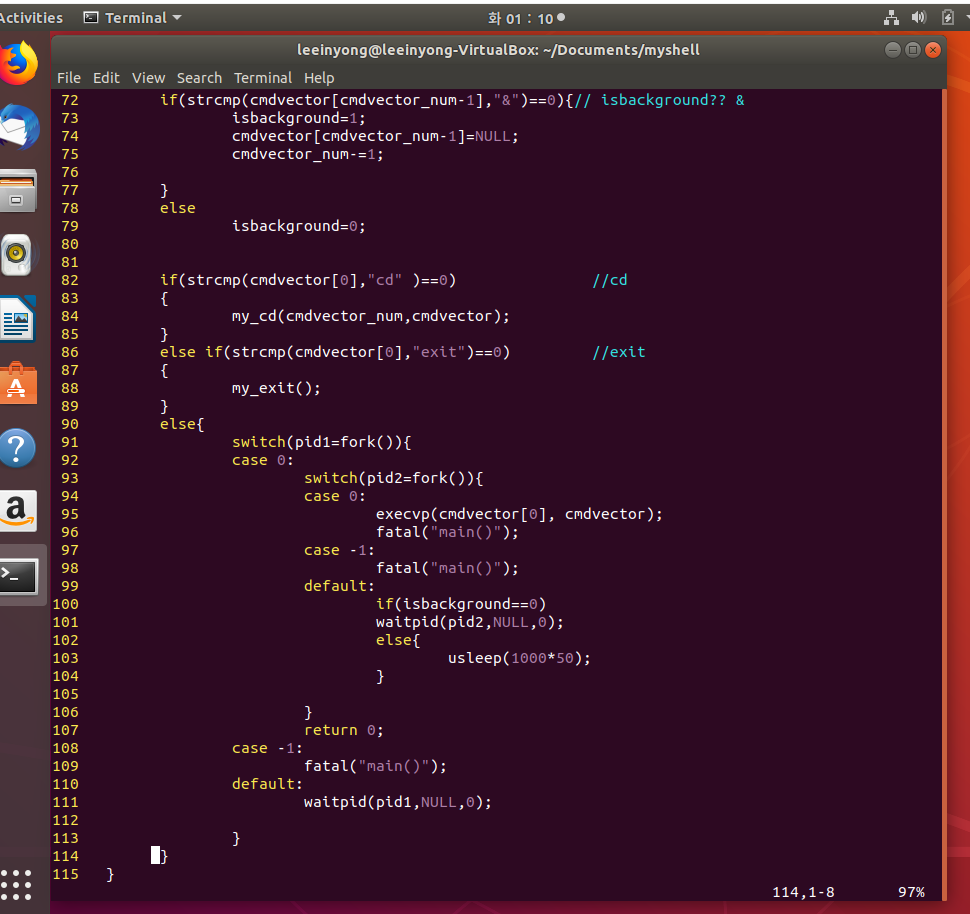
-1주차 구현 방법 서술

1. cd 구현

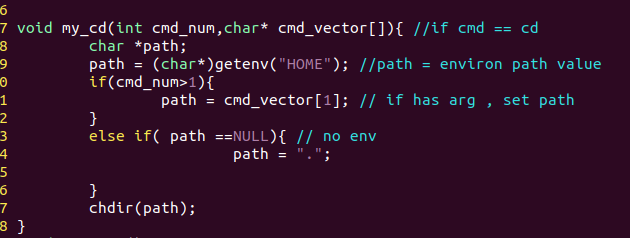
처음에 조교님이 올려주신 c파일에서는 모든 명령어에 대해서 자식 프로세스에서 exec을 써서 명령어를 수행하는 방식으로 되어있었는데

Cd 명령어만 이상하게 작동을 안하였다. 무엇이 문제인가 해서 요번에는 직접 chdir 명령어를 사용해서 구현하였는데 그래도 작동하지 않았다. (문제점)

조금 더 공부해보니 자식프로세스에서 chdir해도 자식프로세스의 current working directory가 바뀌는 것이지 쉘의 cwd가 바뀌는 것이 아니었다. 그래서 fork를 하기전에 cd 명령어를 처리해주기 위하여 fork 앞에 cd 명령어에 대한 예외 코드를 작성하였다. 바로 밑은 코드 사진이다.



중간부분에 fork switch문 들어가기 전에 if else if else등으로 각 명령어마다 상황을 다르게 처리하였다. strcmp이용하여 cmdvector의 첫번째 토큰이 cd일 때 cd 명령어를 실행했다고 보았다. 미리 말하자면 exit도 역시 cd와 마찬가지로 예외적으로 앞에 빼서 구현해 주었다. Background도 strcmp로 cmdvector마지막 토큰이 &일 때 background명령어를 실행했다고 보고 isbackground라는 변수를 1로 설정하는 코드가있다. 이것으로 백그라운드인지 아닌지 판별할 것이다.

My\_cd 라는 cd 명령어 처리함수를 보겠다.

Cd 명령어를 exec해서 프로세스로 구현할게 아니기 때문에 chdir를 사용하였다.

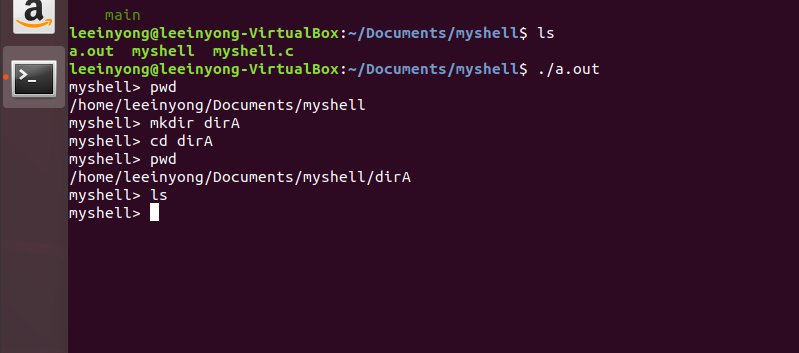
여기서 나는 기본 cd 명령어와 비슷하게 구현하기 위해 여러가지 경우를 생각했다

첫째 cd argument가 없을 때 home으로 chdir 한다.

둘째 environment path가 없을 때 그냥 . 으로서 자기자신으로 한다

셋째 path 에 argument넣어서 chdir한다

세가지 경우에 대해서 각각 chdir(path)에서 path 값이 변한다. 여기서 가장 기본은 셋째 경우이다. 결국 cd 구현에서 제일 중요한 것은 fork 밖으로 cd를 빼는 것이다.

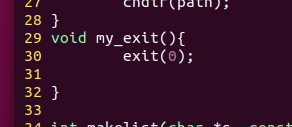


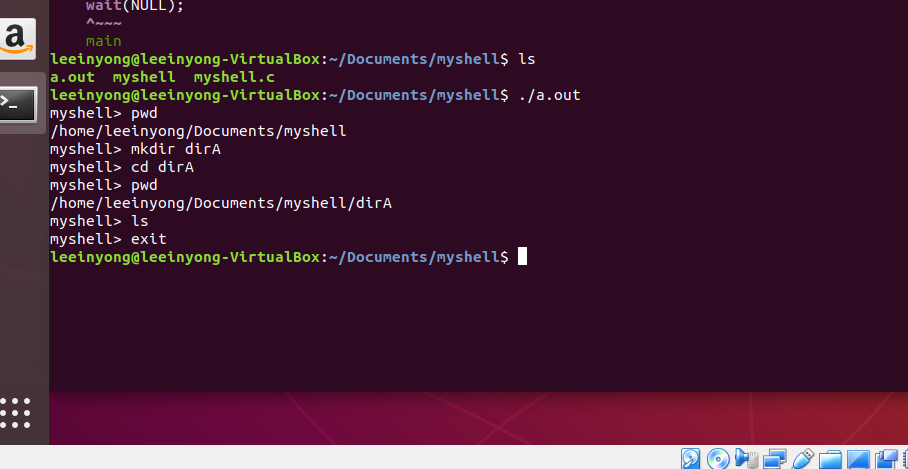
위 사진은 cd에 대해서 실행한 화면이다. Cd dirA를 함으로써 pwd가 바뀐 것을 알 수있다.

2. exit 명령어 구현

exit이라고 명령어를 쳤을 때 쉘이 종료되면 된다. 이것은 앞의 cd처럼 쉘이 종료 되어야 하는 명령어이기 때문에 exit(0)을 호출해주기만 하면 된다.

exit도 그냥 자식프로세스를 생성하여 안에서 실행하면 쉘이 종료 안되는 것을 확인, cd와 마찬가지로 fork전에 예외로서 처리해주었다.(문제점)

.



위 사진은 exit 실행 화면으로 쉘이 잘 종료되는 것을 볼 수 있다.

3.백그라운드 명령어 구현

요번 과제에서 제일 곤란한 문제라고 생각한다. 백그라운드 명령어는 포어그라운드 명령어와 다르게 명령어를 실행하자마자 바로 새로운 shell창이 떠야하는게 특징이다. 그것에 초점을 맞춰서 구현을 해보았는데 가장 기본적으로 백그라운드는 & 명령어가 마지막에 들어가야 실행된다. 이것은 cd 쪽 코드설명하면서 같이 설명한 것 처럼 isbackground 변수로 0,1에 따라 포어그라운드인지 백그라운드인지 나타내주었다. &는 cmd\_vector의 마지막 offset을 구해서 그 문자열이 “&”인지 비교하여 주었다.



위 코드에 대한 자세한 설명은 4번에서 할 것이며 여기서는 간단하게 백그라운드의 기초인 부모가 wait를 하지 않는 것에 초점을 맞추었다.

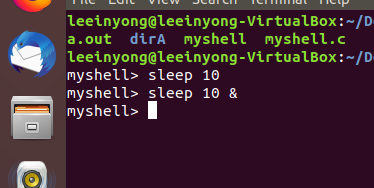
Isbackground ==0 이면 포어그라운드이기 때문에 wait하고

Isbackground ==1 이면 백그라운드이기 때문에 usleep만 해주었다.

여기서 usleep 한 이유는 가끔가다 출력하는 함수를 background로 돌릴 때 미묘하게 느리게 출력되어서 myshell>이 먼저 출력되는 문제가 생긴다.

이러면 보기에 너무 안 좋기에 약간의 딜레이를 넣어서 문제를 해결하였다.

실행 화면



Sleep 10 을 하면 10초후에 myshell> 이 뜨고 sleep 10 & 을 하면 바로 myshell> 이 뜨는 것을 확인하였다.

4. 좀비프로세스가 생기는 이유

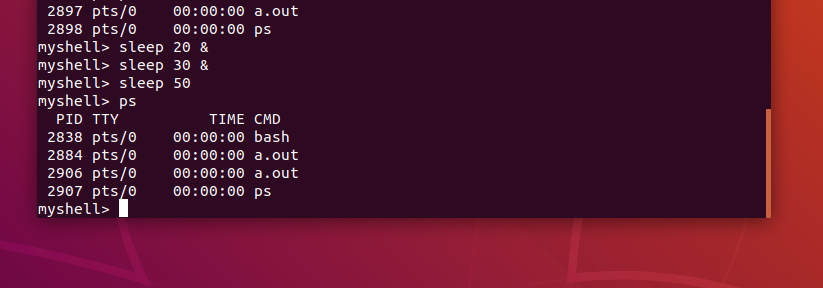
백그라운드로 명령어를 실행하면 당연히 wait를 안하기 때문에 자식은 죽어버려도 부모가 wait를 해줄 때 까지 계속 좀비로서 지속된다.

이것은 좀비프로세스의 정의이다 . 내가 처음에 짠 코드에서는 쉘을 부모로 고 가지기 때문에 부모가 프로그램 종료전까지는 죽지 않는다. 결국 고아프로세스가 생기지 않게 되고 init이 wait 할 기회도 못가져서 좀비프로세스가 회수 되지않게된다. 이것은 조금 곤란한 문제이다. 그래서 나는 코드를 다시한번 짜게 되었고 결국 fork안에 fork를 한번 더 부르고 중간 자식을 죽여버림으로써(return 0해버림) 인위적으로 고아프로세스를 만들었다.



이렇게 하면 고아프로세스가 생겨서 init이 background 명령어가 끝나면 자동으로 회수해가는 완벽한 모습이 되게 된다. 좀비프로세스가 없어지지 않는 것은 자원 낭비이기 때문에 많은 고민을 하게 된 문제였다.

2주차 구현방법 서술



Sleep을 백그라운드로 2번 실행하고 새로운 sleep을 하였을 때 내 코드는 이미 더 이상의 코드 수정 없이 정확히 작동하였다. 1주차 4번의 좀비프로세스가 생기는가에 대한 질문으로 이미 설명을 하였다.fork를 두번 실행하여 쉘 -> 중간자식(마지막자식의 부모가된다)

->마지막 자식 순으로 자식 프로세스들을 생성하고 마지막 자식에서 exec를 실행 ,

중간자식은 새로운 자식을 만들고 나서 return 0 해주어서 자동으로 죽는다. 그러면 백그라운드로 명령어 실행하여도 그것이 고아프로세스가 되기 때문에 init에 의해 자동으로 wait가 호출된다.

이런 원리로 좀비프로세스가 남아있게되는 문제를 해결하였다.

처음에는 SIGCHLD를 어디에 써야하는지 고민하였는데 내 코드에서는 넣을 곳도 넣을 이유도 없었다. 그래서 굳이 하지않고 문제 해결에만 집중하도록 하였다.

(수정본) 조교님의 메일을 받고서 SIGCHLD 를 꼭 사용하라고 하셔서 코드를 수정하였다. 어쩔 수 없이 fork를 하나로 줄이고 코드 맨 앞에 실습 시간에 배운 좀비프로세스 없애는 코드를 첨가하였다.

Struct sigaction act; Sigemptyset(&act.sa\_mask); Act.sa\_flags =SA\_NOCLDSTOP;

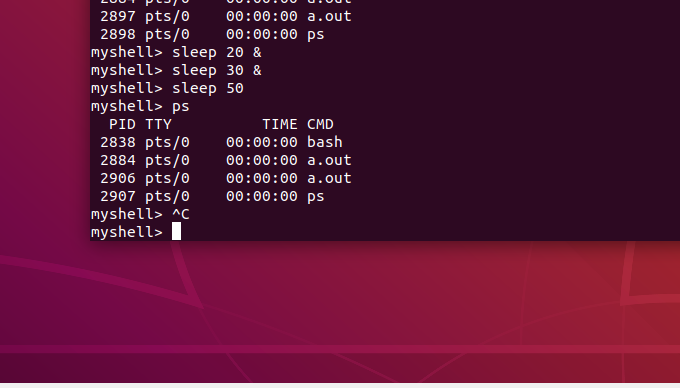
Act.sa\_handler = SIG\_IGN;

Sigaction(SIGCHLD,&act,NULL); //앞대문자는 오타임

실습 코드를 그대로 넣었는데 사실 여기서 중요한 것은 sa\_handler = SIG\_IGN이다.

자식이 죽어도 좀비프로세스가 생성되지 않게 막는다.

2.



쉘에서 ^C를 눌러도 문제가 없다. 정말 간단하게 해결 할 수 있었다. Main문 시작하자마자

Signal(SIGINT,SIG\_IGN);

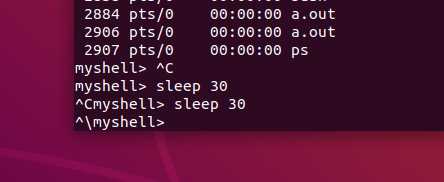
Signal(SIGTSTP,SIG\_IGN);

Signal(SIGQUIT,SIG\_IGN);

을 써주니 그 세가지 명령어를 쉘이 무시하였다.

쉘이 가장 부모인 프로세스이기 때문에 맨 앞에 써주어 signal ignore 하였다.

3.



Signal을 사용하면서 제일 어려운 문제였다. Foreground프로세스를 실행할 때 이것이 SIGINT와 SIGQUIT를 default로 실행을 해야하는 문제였다.

쉘을 짤 때 Foreground프로세스들은 쉘과는 다른 프로세스 그룹을 형성해야 한다. 그리고 그 프로세스 그룹에게 터미널 제어권을 넘겨주어 signal 처리를 해주고 다시 쉘에게 제어권을 넘겨주어야한다. 일단 코드를 보자



Pid2쪽 fork를 하고 자식프로세스에서 signal을 DFL로 바꾸어주어서 ign -> DFL로 바꾸어준다.

그리고 자식 프로세스들은 각자 프로세스 그룹을 생성해주어야한다.

Setpgid(0,0)을 해주면 호출한 pid로 pgid를 바꾸는 함수이고 이것은 프로세스 그룹리더를 뜻한다.

이렇게 새로운 프로세스 그룹을 생성하면 이제 포어그라운드일 때 터미널 제어권을 넘겨주어야 한다.

이때 tcsetpgrp(STDIN\_FILENO,getpgid(0))를 써주어서 쉘에서 제어권을 자식에게 넘겨준다.

여기서 STDIN\_FILENO는 그냥 filedes 0 이고 표준입력을 뜻한다.

Getpgid(0)은 호출한 프로세스의 그룹을 뜻한다.

그리고 부모에서 waitpid가 끝난후에 다시 쉘에게 제어권을 넘겨주면 된다.

이때 tcsetpgrp(STDIN\_FILENO,getpgid(getppid()))를 써주어서 쉘에서 제어권을 자식에게 넘겨준다.

Getpgid(getppid())를 써준 이유는 내 코드에서 이 함수를 호출한 프로세스의 조부모가 쉘이기 때문이다(부모가 쉘아님)

위 코드를 실행하면 정상적으로 프로세스가 ^C ^\ 에 꺼지는 것을 볼 수 있다.

(수정본)

Fork를 다시 한번으로 고치면서 특별히 달라진건없고 tcsetpgrp(STDIN\_FILENO,getpgid(getppid()))를 써주어서 쉘에서 제어권을 자식에게 넘겨주는 부분을 다시 tcsetpgrp(STDIN\_FILENO,getpgid(0))을 써주어서 shell에게 제어권 넘기도록 했다.

3주차 구현 방법 서술

1. redirection 확인

3주차 과제는 1,2주차 과제와 비교해서 구현이 매우 어려웠다.

Redirection은 dup2를 이용해서 간단하게 만들 수 있었다.

Redirection의 개념은 명령어 후에 < ,> 가 나오고 그 다음에 파일명이 나오는 형식을 가지고 있는데 (EX. Cat > test.txt)

“명령어 < 파일명” 형태는 파일을 open해서 dup2(fd,0)을 하여서 표준입력을 파일로 하여서 파일 내용을 cat이 사용하게 된다. 하고 쓸모없는 fd는 close(fd)해줬다.

반대로 “명령어 > 파일명” 형태는 파일open해서 dup2(fd,1)을 하여서 표준출력을 file로 하여서 명령어의 결과가 파일에 저장되게 한다. 하고 쓸모없는 fd는 close(fd)해줬다.

Dup2를 해서 표준입력과 표준출력을 바꿔줬으면 끝나는게 아니다. 제일 중요한 파싱 과정이 남아 있다. 기본적으로 개념자체는 매우 쉬웠지만 파싱에서 많은 에러가 발생하였다. 일단 나는 redirection을 표준 출력 ,입력으로 나누어서 비슷한 함수 2개를 생성하였다. 그리고 기본적으로 파싱된 문자열 배열은 전부 전역변수를 사용하였다.

전역변수를 사용해도 크게 꼬이지 않아서 전역변수를 계속 사용하였다.

파이프 전에는 그저 예전에 makelist 한번만 사용한 배열을 사용하여서 그렇게 문제될거 없이 그냥 사용할 수 있었다. 그런데 파이프를 하면서 파싱을 2번 하게 되어서 거기서 조금 헷갈려서 시간을 많이 낭비하였다.

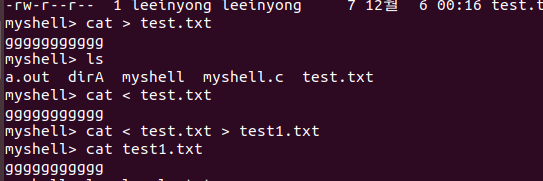
일단 문자열배열을 하나 받는다. 반복문을 통해 명령어 입력문에 < 나 > 가 있는지 찾고 그 위치를 i에 남겼다. 그리고 cmdvector[i] 나cmdvector[i+1] 두개가 NULL이면 예외처리를 해주었고 두개다 있으면 cmdvector[i+1]이 파일 이름일 것이므로 그것을 open하였다. 여기서 > 가 있으면 O\_CREAT까지 넣어주어서 없는 파일은 생성해서 넣어주도록 하였다.

그렇게 파일 open하고 위에서 설명한 dup2를 하면 redirection은 끝이고 마지막으로

I와 i+1자리에 NULL을 넣어 지우고 i+2를 i자리에 넣어서 땡기는 작업을 한다.

>와 파일이름을 지우는 이유는 execvp에 방해되기 때문이다.

리다이렉션은 execvp바로 직전에 함수 호출되어 기능한다.



결과는 잘나왔다. Cat > test.txt 로 ggggggggggg를 파일안에 넣었고

Cat < Test.txt로 그 값을 화면에 나타내주었다.

1. 파이프 구현 방법

파이프 구현은 정말 최고 난이도였다.

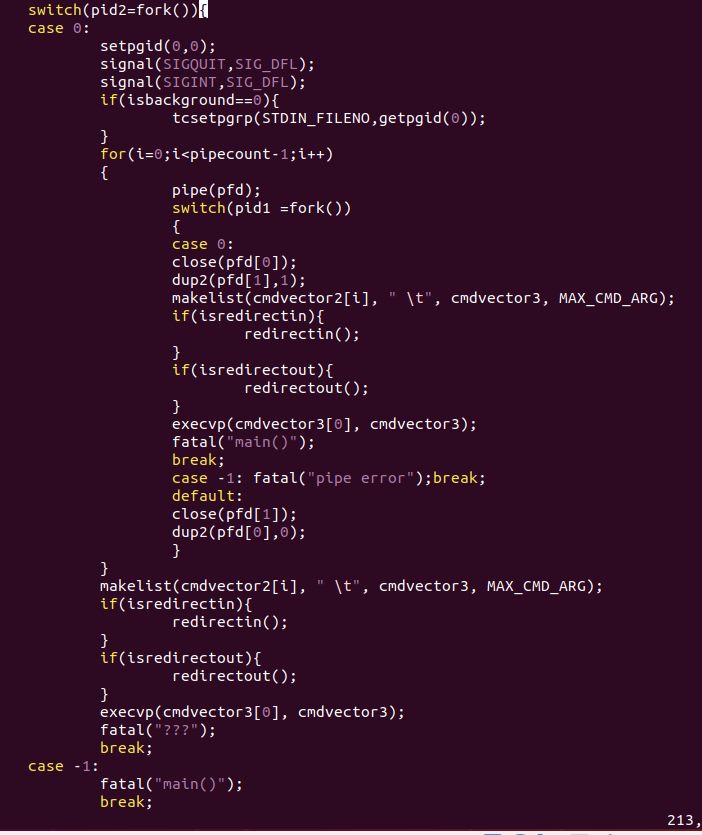
파이프자체의 개념은 그리 어렵지 않게 세울 수 있었다.

파이프는 프로세스 (부모와 자식)사이에서 파이프를 사이에 두고 입력이나 출력 같은 서로 연결하는 방법이다.

쉘에서 파이프는 명령어들끼리 입출력을 연결할 수 있게 한다.

명령어 자체가 결국 execvp를 뜻하므로 나는 명령어 개수만큼 execvp를 해주었다.

이때 맨 처음 fork에서 자식일 때 명령어 개수-1 만큼 for문 돌려가면서 fork -> execvp 해주었고 마지막 명령어는 다시 부모에게 정보를 넘겨줄 필요가 없으므로 첫 번째 fork 자식에서 execvp 해주었다.

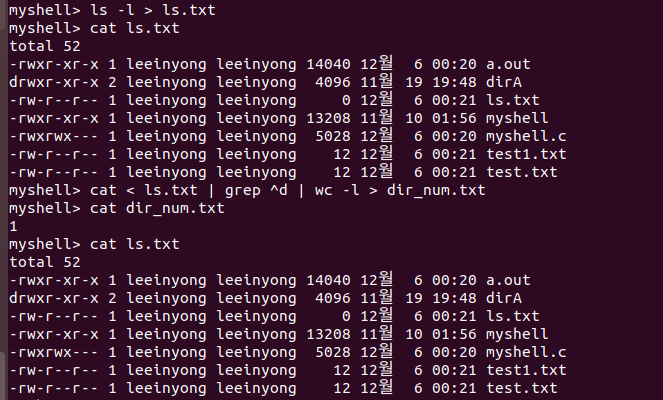
설명을 다시 하자면 

위 코드에서 for문 안에 명령어개수(==pipecount)-1번 만큼 계속 새롭게 pipe를 연결하고 안쪽 fork에서 자식일 때는 표준출력을 파이프 [1]번으로 한다. 결국 파이프에 표준출력을 넣는 것이다.

그리고 부모일 때는 표준입력을 파이프로 한다 .결국 파이프값을 받아온다.(버퍼에 넣음으로서 다음번 명령어가 이용가능 ^^)

이런식으로 계속 진행하면 파이프역할을 할 수 있다. 참고로 redirection이 나오면 파이프가 날아가 버리므로 redirection은 양끝에 나와야 될 것이다.(중간에 나오면 dup2가 겹쳐버릴것이다.)

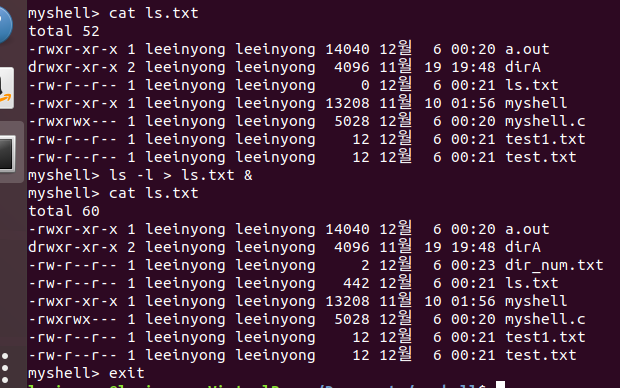
파이프 기능은 구현해주었는데 계속 에러가 난 부분은 makelist부분이었다. “|”로 파싱을 하고 그 파싱된 부분들을 다시 “ \ t”로 파싱하는 것 까지는 이해가 되었는데 처음에 이상하게 makelist를 한번만 넣어서 문제가 생겼었다. Execvp 전에 계속 반복적으로 makelist를 넣어야 잘 작동한다.



꽤나 긴 명령어들이었다. 그래서 MAX\_CMD\_ARG를 20으로 늘려주었다.

정확히 잘 작동하였다.

1. 백그라운드 구현방법



3번 문제는 이해가 잘되지 않았다. 조교님이 주신 testcase를 실행해 보았는데

파이프를 만든 뒤에는 그냥 잘 되었다. 그래서 여기서 더 이상 수정을 하지않았다.

그냥 더욱더 생각을 해보았는데 지금 명령어들은 매우 빠르게 수행되고 겹치지 않아서 문제가 없겠지만 만약 여러 개의 명령어들이 같이 실행된다고 했을 때 모든 프로세스 들이 STDIN STDOUT을 dup2를 통해 건들기 때문에 겹치는 문제가 생길 수도 있다고 생각해보았다. 만약 그렇다면 새로운 파일을 프로세스마다 만들어서 (입력시 깨끗이 지우고 입력 받는)

표준입출력 대신에 그 파일을 open해서 사용하고 프로세스 종료시 그 파일을 지우는 식으로 하면 겹치는 일이 발생하지 않을 것이다.