<유닉스 프로그래밍 쉘 프로젝트>

1. 요구사항 정의

1.1 'cd'명령이 작동하도록 수정하기

- 주어진 simple\_myshell.c 파일을 실행하면, cd명령어가 잘 작동하지 않았다. 이를 잘 작동하도록 수정하는 것이 목표이다.

1.2 'exit' 명령을 구현하기

- 내가 만든 쉘에서 exit명령을 입력했을 시, 쉘이 종료되도록 구현해야 한다.

1.3 백그라운드 실행을 구현하기

- 명령 뒤에 &를 붙였을 때, 해당 명령이 백그라운드에서 실행하도록 해야한다.

1.4 시그널 처리

- SIGCHLD로 자식 프로세스를 wait()할 때 프로세스가 온전하게 수행되도록 구현해야한다. 또한, SIGINT와 SIGQUIT 사용 시 쉘이 종료되지 않도록 하고 Foreground 프로세스 실행 중에 SIGINT를 받으면 프로세스가 끝나는 것을 구현해야 한다.

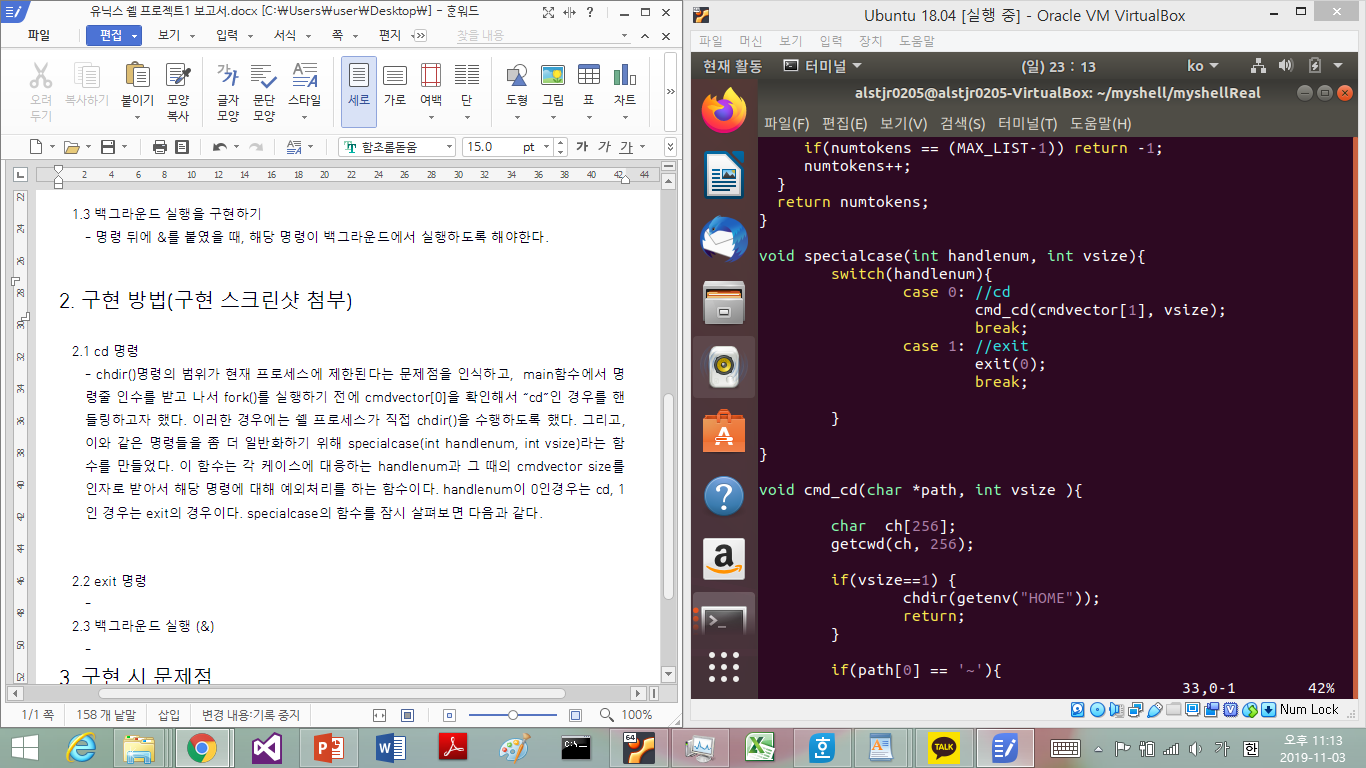
1.5 파이프, 리다이렉션

- '>', '<' 을 구분하여 표준출력이나 입출력을 조정하여 파일을 입력으로 받거나, 명령어의 출력을 파일로 옮길 수 있어야 한다. 또한, '|'를 구분하여 프로세스 간에 입출력을 주고받을 수 있어야 한다.

2. 구현 방법(구현 스크린샷 첨부)

2.1 cd 명령

- 먼저 chdir()명령의 범위가 현재 프로세스에 제한된다는 문제점을 인식했다. 이를 해결하기위해 main에서 fork()를 실행하기 전에 cmdvector[0]을 확인해서 “cd”인 경우를 핸들링 했다. 이후에 쉘 프로세스가 직접 chdir()을 수행하도록 했다. 그리고, 이와 같은 명령들을 좀 더 일반화하기 위해 specialcase(int handlenum, int vsize)라는 함수를 만들었다. 이 함수는 각 케이스에 대응하는 handlenum과 그 때의 cmdvector size(argc와 같은 역할)를 인자로 받아서 해당 명령에 대해 예외처리를 하는 함수이다. handlenum이 0인 경우는 cd, 1인 경우는 exit의 경우로 설정했다. specialcase의 함수를 잠시 살펴보면 다음과 같다.

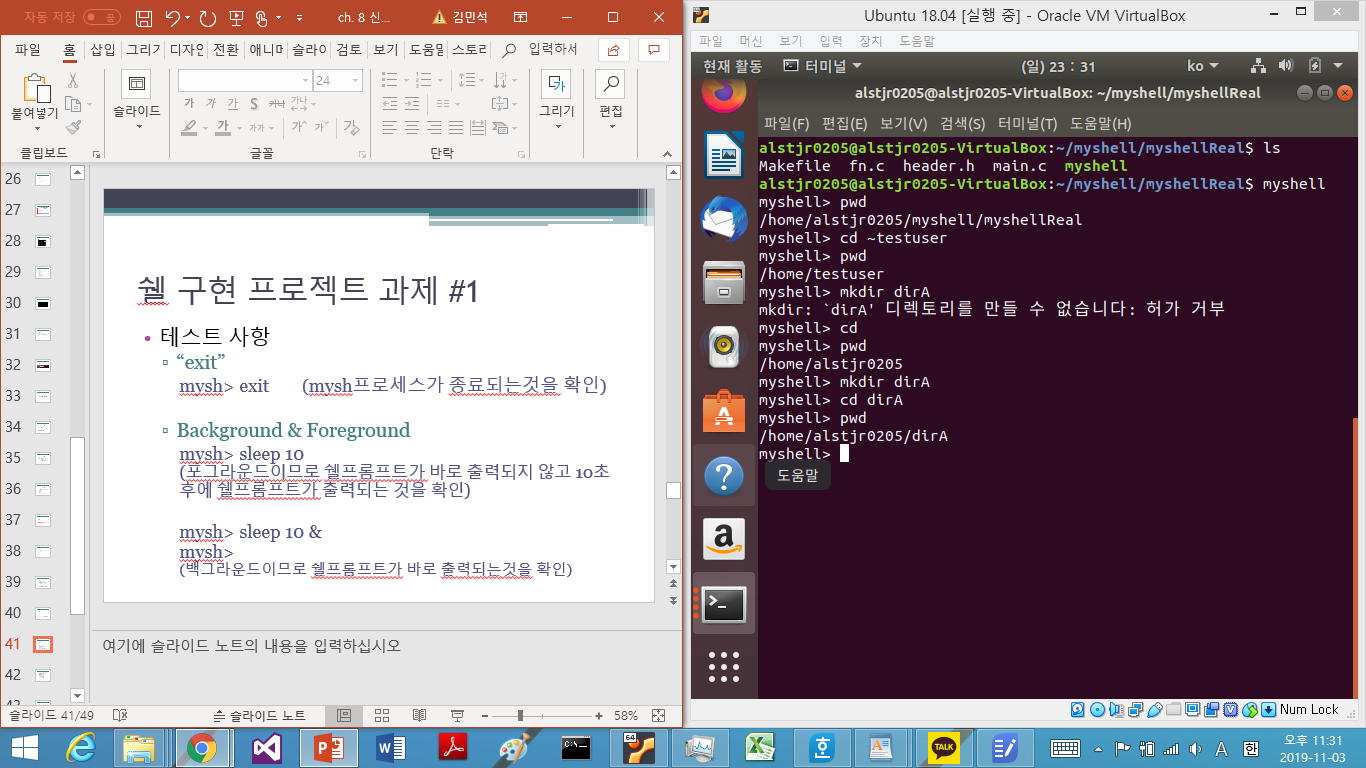


cmd\_cd함수는 해당 프로세스에서 chdir()을 통해 cd명령어를 구현한 것이고, cd만 입력한 경우와 '~'문자 입력에 대한 여러 경우들도 구현했다. 특히 '~'에 대해 구현하기 위해 환경변수에 대한 함수 getenv(”HOME”)를 이용했다.

main함수의 구현을 살펴보면, 먼저 cmdline을 입력받고 makelist를 실행했다. 이후에 cmdvector[0]과 commandlookup의 변수들을 strcmp로 검사했다.(commandlookup에는 “cd”, “exit”이 순서대로 저장되어 있다.) 그리고, 같은 문자열이 나오면 specialcase로 가야함을 알리는 flag 를 1로 설정하고, handlenum을 설정했다. 그리고 specialcase함수로 넘어가서 해당 명령을 쉘 프로세스에서 처리하고, main으로 돌아왔을 때, continue를 통해 다시 쉘이 또다른 명령을 입력 받을 수 있게 프로그램을 만들었다.

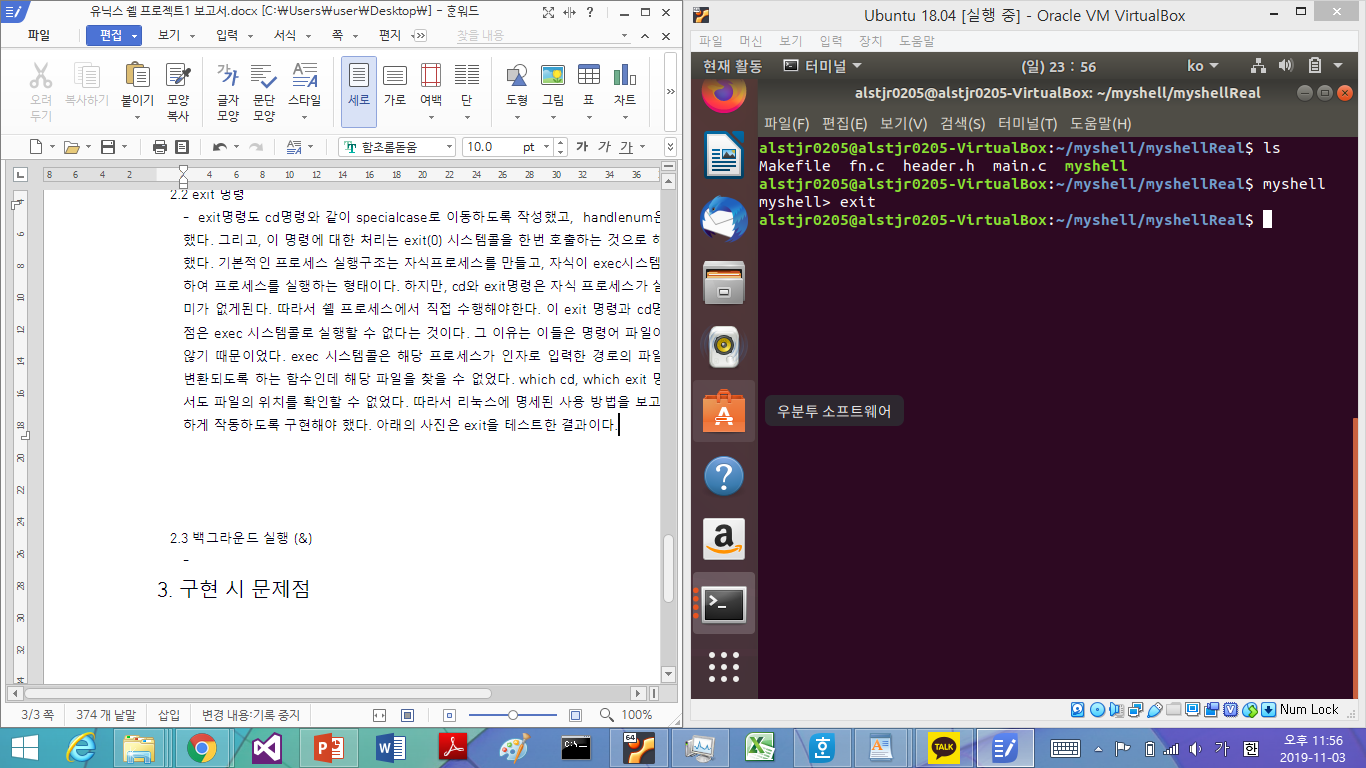
아래의 사진은 cd의 테스트 결과이다. make 명령어를 통해 fn.c, main.c, header.h를 컴파일하고, myshell 실행파일을 생성해놓은 상황이다.

`'~' 를 테스트하기 위해 임의의 유저 testuser를 만들어서 cd가 되는지를 테스트해보았고, 해당 디렉토리 내의 쓰기권한이 없어서 디렉토리 생성에는 실패했다.



2.2 exit 명령

- exit명령도 cd명령과 같이 specialcase로 이동하도록 작성했고, handlenum은 1로 설정했다. 그리고, 이 명령에 대한 처리는 exit(0) 시스템콜을 한번 호출하는 것으로 해결이 가능했다. 일반적인 프로세스 실행구조는 자식프로세스를 만들고, 자식이 exec시스템 콜을 호출하여 프로세스를 실행하는 형태이다. 하지만, cd와 exit명령은 자식 프로세스가 수행하면 의미가 없게된다. 따라서 쉘 프로세스에서 직접 수행해야한다. 이 exit 명령과 cd명령의 공통점은 exec 시스템콜로 실행할 수 없다는 것이다. 그 이유는 이들은 명령어 파일이 존재하지 않기 때문이었다. exec 시스템콜은 해당 프로세스가 인자로 입력한 경로의 파일의 코드로 변환되도록 하는 함수인데 해당 파일을 찾을 수 없었다. which cd, which exit 명령을 통해서도 파일의 위치를 확인할 수 없었다. 따라서 구글링을 통해서, 또는 리눅스에 명세된 사용 방법을 보고 직접 유사하게 작동하도록 구현해야 했다. 아래의 사진은 exit을 테스트한 결과이다.



2.3 백그라운드 실행 (&)

- 백그라운드 실행은 main에서 구현하였다. cmdline에 접근을 많이 할 것으로 예상해서 코드를 편리하게 보면서 작성하고자 했다. 또한, linux의 쉘과 유사하게 “sleep 5&ps”와 같이 '&'가 끝이 아닌 위치에 오는 경우도 처리하고자 했다. 리눅스 쉘에서 몇번 테스트해본 결과, '&'를 구분문자로 보고 '&'이전에 위치한 명령을 백그라운드로 실행한다는 것을 알게 되었다. 즉, '&'가 중간에 위치할때, 중간의 '&' 다음에 오는 명령은 또다른 프로세스를 실행시켜야 한다. 또한 백그라운드 실행구조는 쉘이 자식 프로세스를 만들고, 그 자식이 또 자식을 만들어서 exec를 실행하고, 쉘의 자식은 waitpid(pid, NULL, WNOHANG)을 통해 바로 종료되도록 구현했다. 그림으로 표현하면 아래와 같다.

3. 쉘의 자식이 wait하지 않고종료한다.

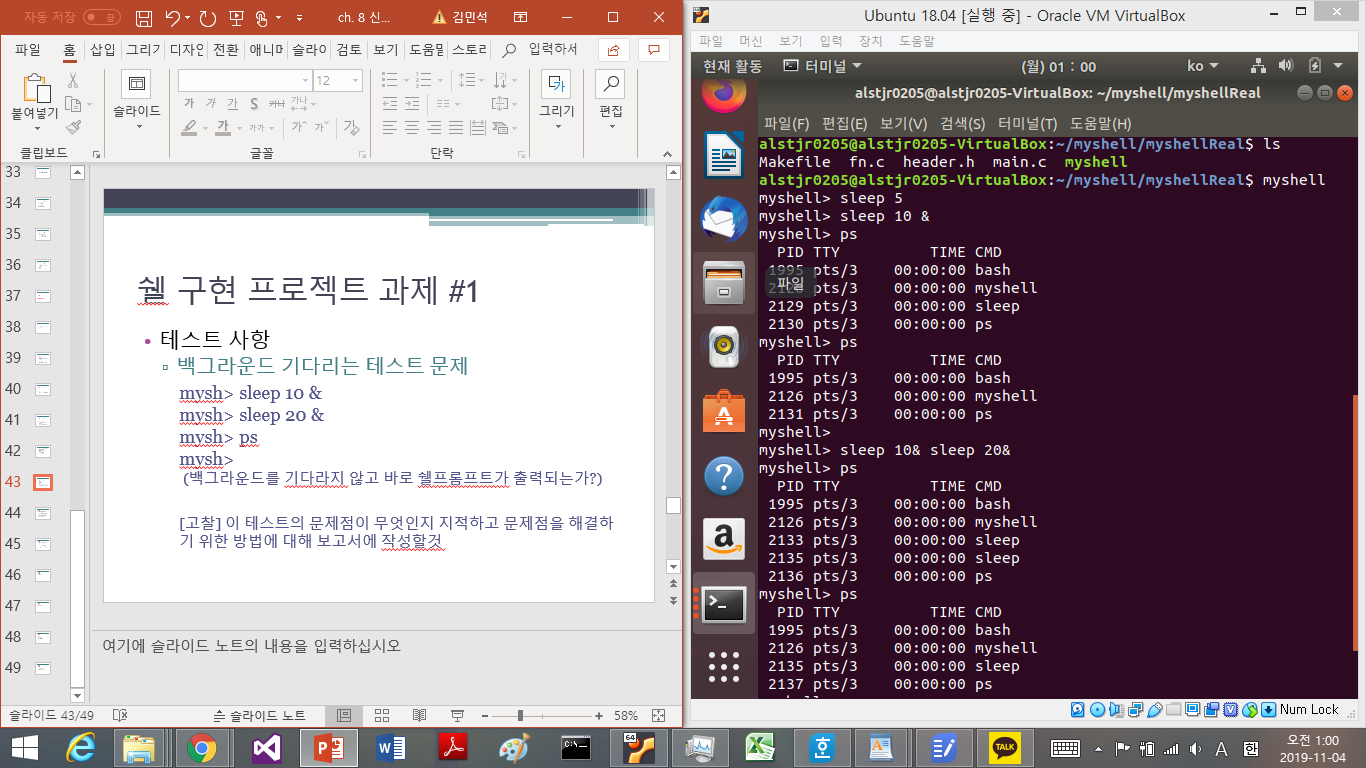
4. 쉘의 자식이 만든 자식은 고아 프로세스가되며, 이후에 백그라운드로 실행하게 된다.

1. 쉘이 자식프로세스를 생성한다.

2. 쉘의 자식이 자식프로세스를 생성한다. 그리고 exec를 실행한다.

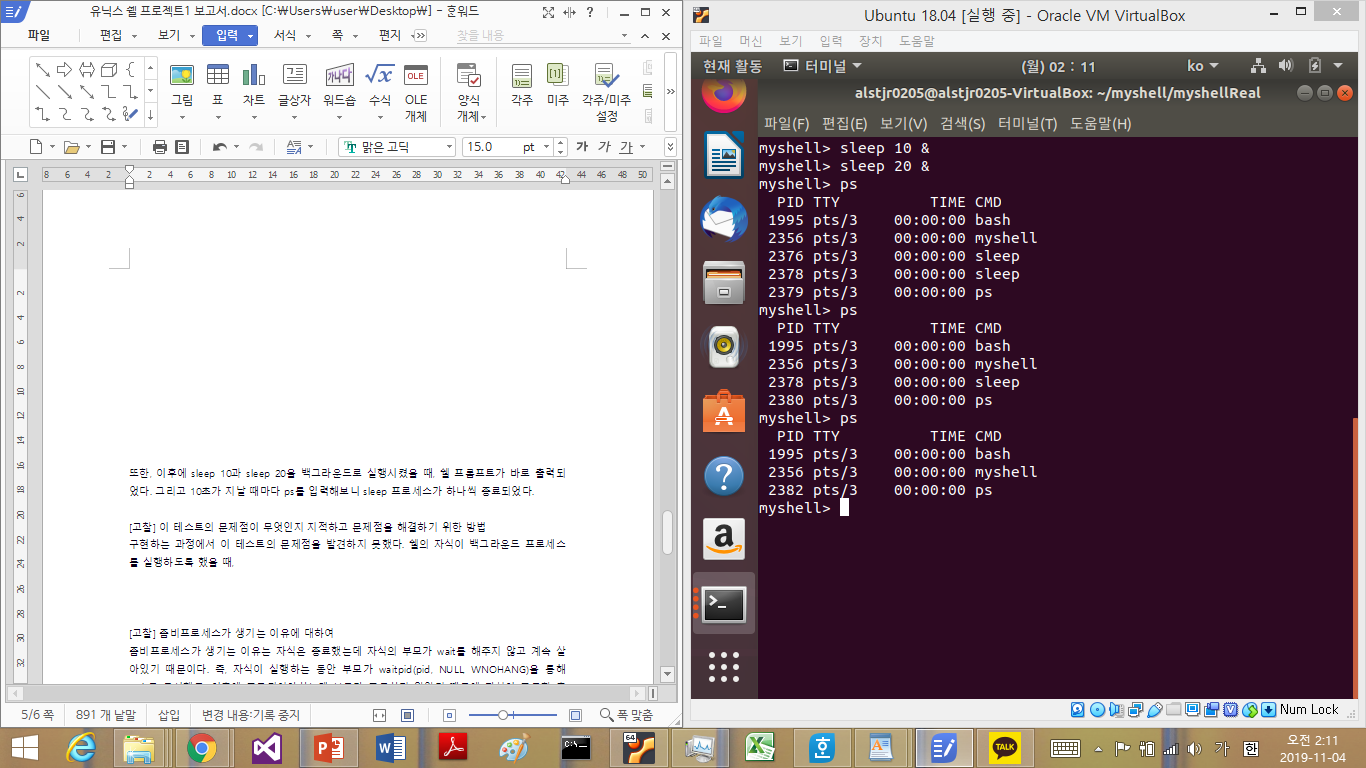
sh

구현 방법을 살펴보면, 먼저 main에서 cmdline을 받고, cmdline에서 '&'를 검색한다. '&'가 검색이 안되었을 경우, 그 명령어는 그대로 다음단계로 이동해서 makelist, specialcase 체크 및 fork실행을 하게 된다. '&'가 검색되었을 경우, 검색된 위치를 저장하고, 그곳을 '\0'문자로 교체한다. 그리고 cmdline을 makelist로 cmdvector에 토큰화하여 저장한다. 그러면 '&'이전의 문자열까지 토큰화가 진행된다. 이 cmdvector을 가지고 fork()를 두번 실행해서 자식과 그 자식의 자식을 생성하고, 쉘의 자식은 waitpid(pid, NULL, WNOHANG)을 실행하고, 자식의 자식은 exec(cmdvector[0], cmdvector)을 실행한다. 그리고 쉘 프로세스 코드 부분에서 '&'가 검색된 위치의 다음으로 cmdline의 시작위치를 옮긴다. 그리고, 변경된 cmdline의 마지막 위치를 '\0'으로 설정해준다. 이 과정을 마치고나면, cmdline은 검색된 '&'이전의 문자열과 '&'를 없애고, 그 이후의 문자열만 담게 된다. 이 과정을 '&'가 검색이 되지 않을 때까지 반복해서 중간에 오는 모든 '&'와 끝에 오는 '&'에 대해 백그라운드 실행을 하도록 구현했다. 테스트 결과는 아래의 사진과 같다.



-

먼저, 포그라운드로 실행 시 5초 뒤에 쉘프롬프트가 출력되었고, 백그라운드로 실행했을 때 쉘프롬프트가 바로 출력되었다. 이후 ps를 통해 sleep 프로세스가 10초 뒤에 종료된 것을 테스트했다.



또한, 이후에 sleep 10과 sleep 20을 백그라운드로 실행시켰을 때, 쉘 프롬프트가 바로 출력되었다. 그리고 10초가 지날 때마다 ps를 입력해보니 sleep 프로세스가 하나씩 종료되었다.

[고찰] 이 테스트의 문제점과 문제점을 해결하기 위한 방법

구현하는 과정에서 발생했던 이 테스트의 문제점은 좀비프로세스가 생성되는 것이었다. sleep 10과 sleep 20이 백그라운드로 실행되었고, 이를 기다리지 않고 바로 쉘프롬프트가 출력되었다. 그러나 20초가 지난 후에 ps를 통해 프로세스 목록을 봤을 때, 두 백그라운드 프로세스가 사라지지 않고 defunct상태로 남아있었다. 그 이유를 분석해보면, 구현 초기에 백그라운드 실행을 구현하기 위해 fork()를 한 번만 실행했기 때문이라고 생각한다. 쉘의 자식프로세스가 바로 exec를 통해 백그라운드로 실행하도록 했고, 부모인 쉘에서 waitpid(pid, NULL, WNOHANG)를 통해 자식을 wait하지 않으면 자식이 고아 프로세스가 될 것이라 생각했다. 하지만, 이렇게 구현하면 쉘이 종료되지 않을 시에 백그라운드 프로세스의 부모가 계속 살아있는 형태가 되어서 백그라운드 프로세스가 모든 일을 끝내도 부모가 wait를 하지않아서 좀비가 된다. 이를 해결하기 위해서는 fork()를 두 번 실행해서 쉘의 자식의 자식, 즉 손자 프로세스가 백그라운드 프로세스가 되어야하고, 쉘의 자식은 종료되어야 한다.

[고찰] 좀비프로세스가 생기는 이유에 대하여

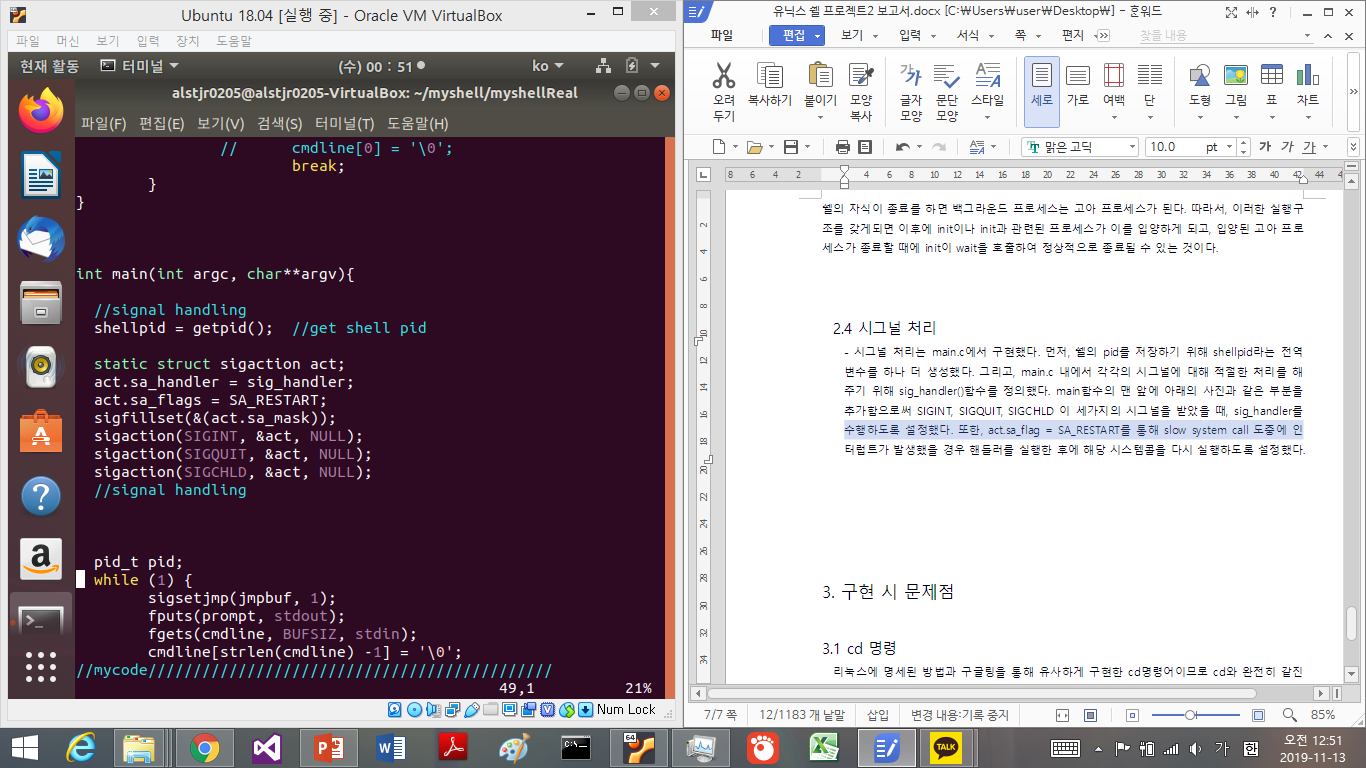
좀비프로세스가 생기는 이유는 자식은 모든 일을 마쳤는데 자식의 부모가 wait를 해주지 않고 계속 살아있기 때문이다. 즉, 자식이 실행하는 동안 부모가 waitpid(pid, NULL, WNOHANG)을 통해 wait를 무시했고, 이후에 부모가 종료하지 않았기 때문에 자식이 일을 마친 후에 좀비로 남아있는 것이라 생각한다.

내가 처음에 백그라운드 실행을 구현했을 때는 쉘프로세스가 fork()로 자식을 만들고, 쉘의 코드 부분이 waitpid(pid, NULL, WNOHANG)을 실행하도록 했었다. 하지만 이를 실행했을 때, sleep이 백그라운드로 잘 실행 되지만 실행 후에 좀비로 남아있는 것을 ps를 통해 확인할 수 있었다. 이는 쉘이 자체적으로 종료하지 않았기 때문에, 즉 부모가 계속 살아있는데 wait를 무시했으므로 쉘이 종료되기 전까지 백그라운드 프로세스는 좀비로 남아있는 것이었다. 이를 해결하기 위해 고안한 것이 fork()를 두 번 실행하는 것이었다. 자식의 자식이 백그라운드 프로세스가 되고, 쉘의 자식이 종료를 하면 백그라운드 프로세스는 고아 프로세스가 된다. 따라서, 이러한 실행구조를 갖게되면 이후에 init이나 init과 관련된 프로세스가 이를 입양하게 되고, 입양된 고아 프로세스가 종료할 때에 init이 wait을 호출하여 정상적으로 종료될 수 있는 것이다.

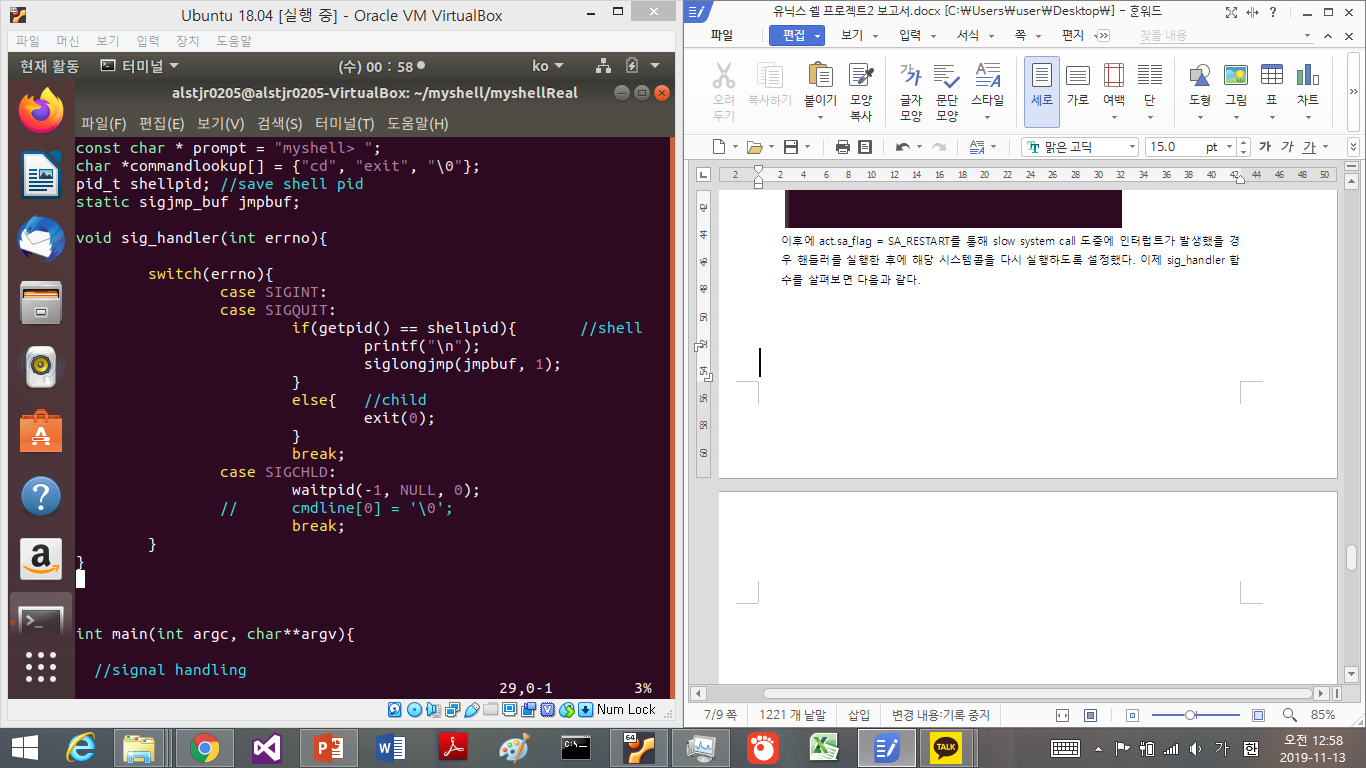
2.4 시그널 처리

- 시그널 처리는 main.c에서 구현했다. 먼저, 새로 생성된 변수들에 대해 설명하면 쉘의 pid를 저장하기 위해 shellpid라는 전역변수를 하나 더 생성했다. 이 변수는 임의의 프로세스가 쉘인지 아닌지 확인하는 용도로 사용된다. 또한, sigsetjmp와 siglongjmp를 사용하기 위해 header.h에 <setjmp.h>를 선언하고 main에서 static sigjmp\_buf jmpbuf 변수를 생성했다. 이는 핸들러함수 내부에서 main함수의 시작지점으로 돌아가기 위해 사용되었다.

그리고, main.c 내에서 각각의 시그널 처리를 해주기 위해 sig\_handler()함수를 정의했다. main함수의 맨 앞부분에 아래의 사진 부분을 추가함으로써 SIGINT, SIGQUIT, SIGCHLD 시그널을 받았을 때, sig\_handler를 수행하도록 설정했다.



이후에 act.sa\_flag = SA\_RESTART를 통해 slow system call 도중에 인터럽트가 발생했을 경우, 핸들러 실행 후에 해당 시스템콜을 다시 실행하도록 설정했다. 이제 sig\_handler 함수를 살펴보면 다음과 같다.

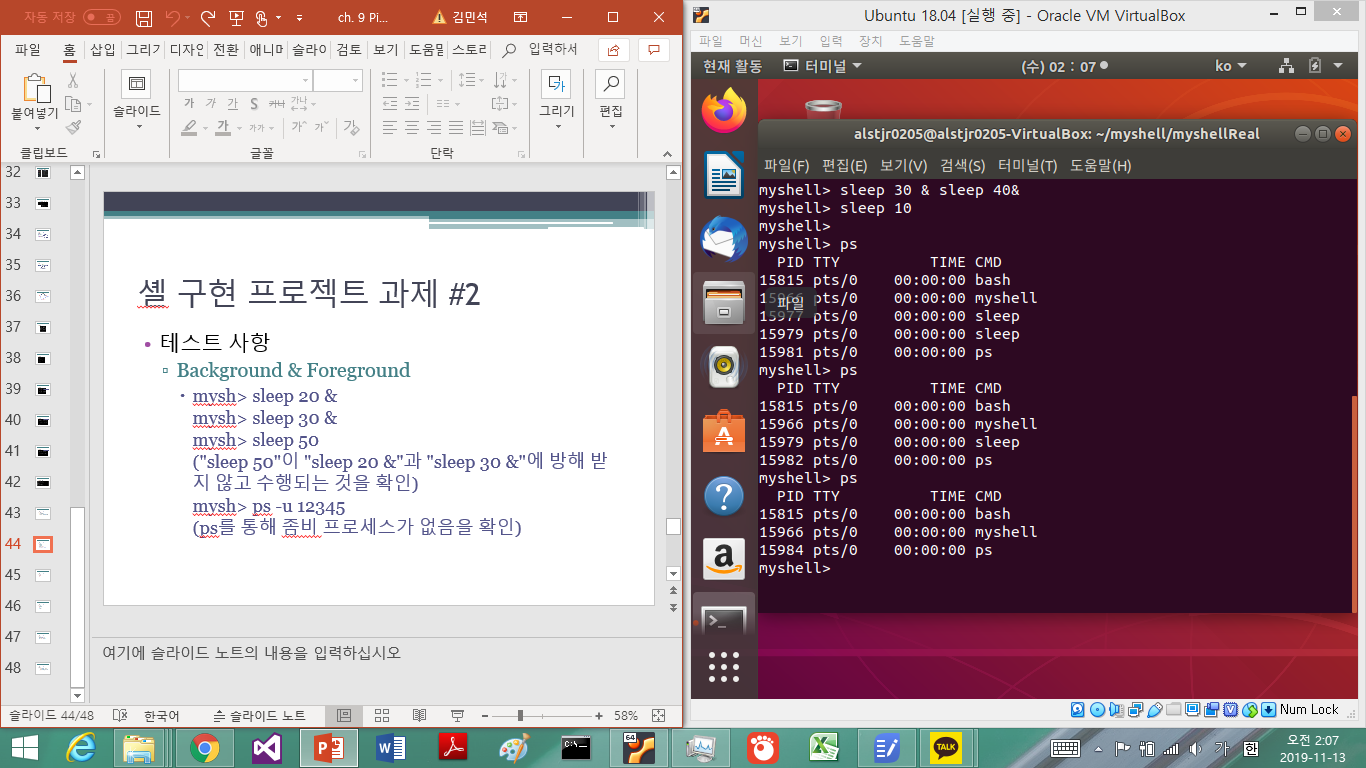


- SIGINT나 SIGQUIT시그널을 받았을 때, 같은 행동을 하도록 구현했다. 만약 현재 프로세스가 쉘이라면 개행문자를 한번 출력하고 main에 설정해놓은 sigsetjump(jmpbuf, 1)로 이동한다. 이 위치로 이동함으로써 main의 while문 안에 있는 fputs와 fgets를 통해 명령어 입력을 다시 받게 된다. SIGINT와 SIGQUIT에 대해 현재 프로세스가 쉘이 아닌 경우는 자식프로세스인 경우이므로, 바로 exit(0)을 호출해서 종료되도록 구현했다.

다음은 SIGCHLD 시그널의 경우이다. SIGCHLD를 받는 경우는 자식프로세스가 종료했을 때이다. 따라서 좀비프로세스가 생성되는 것을 방지하기 위해 waitpid로 수행이 끝난 임의의 자식프로세스를 종료시키도록 구현했다.

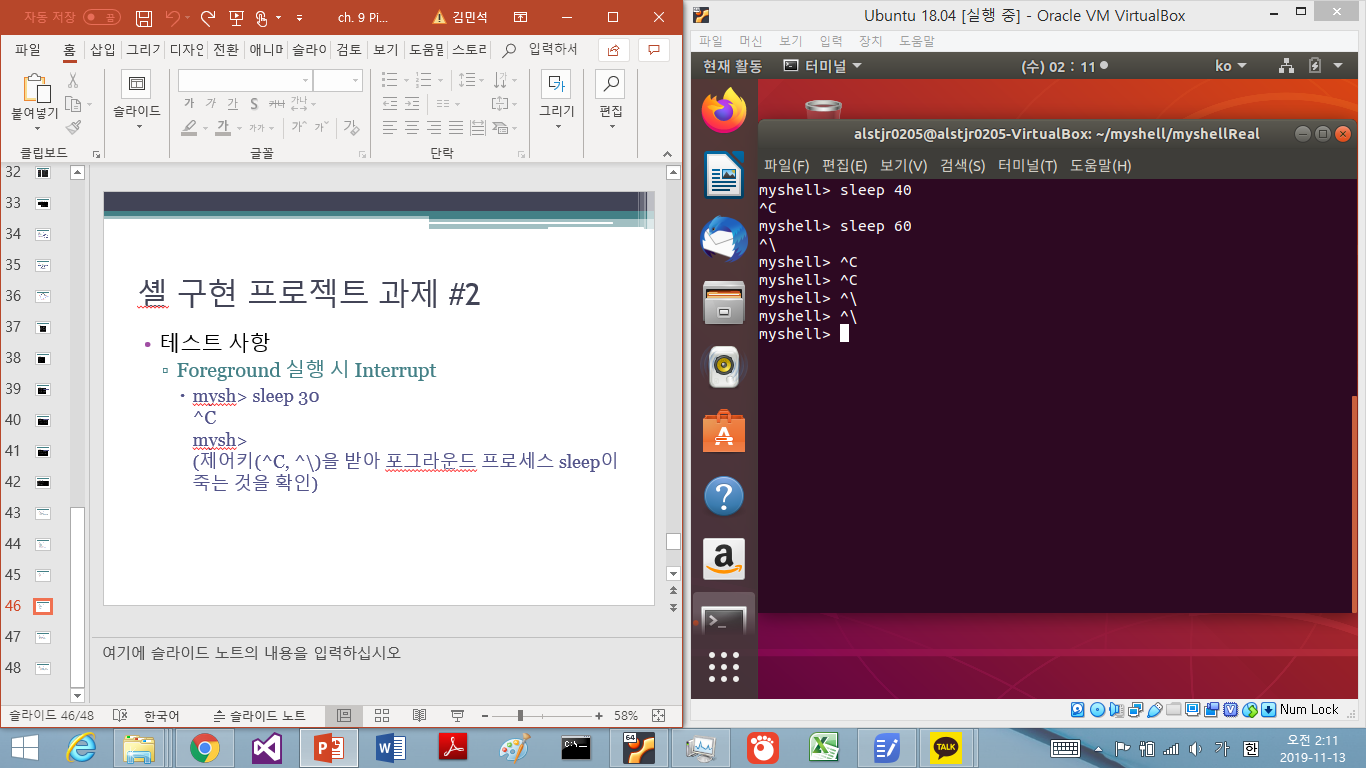
다음으로, 백그라운드로 실행중인 프로세스의 경우이다. 위의 과정까지 구현을 마치고 나서 실험을 해본 결과, 쉘에서 ^C키를 입력하면 기존에 실행되던 백그라운드 프로세스가 모두 종료되는 현상이 발견되었다. 이는, exec로 코드가 변환된 프로세스에 대해서는 sigaction이 동작하지 않기 때문이었다. 코드가 변환된 후에는 핸들러를 인식하지 못하므로, default action을 수행해서 모두 종료되는 것이었다. 따라서 백그라운드 프로세스의 경우에는 SIGINT와 SIGQUIT 시그널을 block처리한 후에 exec를 실행하도록 구현했다. SIGINT와 SIGQUIT을 블럭시켜주는 함수 blockintquit()를 만들었고, 백그라운드 구현부분으로 가서 grand child의 실행부분에 blockintquit();를 추가함으로써 백그라운드 프로세스의 시그널 블럭을 구현했다. 아래의 사진은 여러가지 테스트에 대한 결과이다.

-Background & Foreground



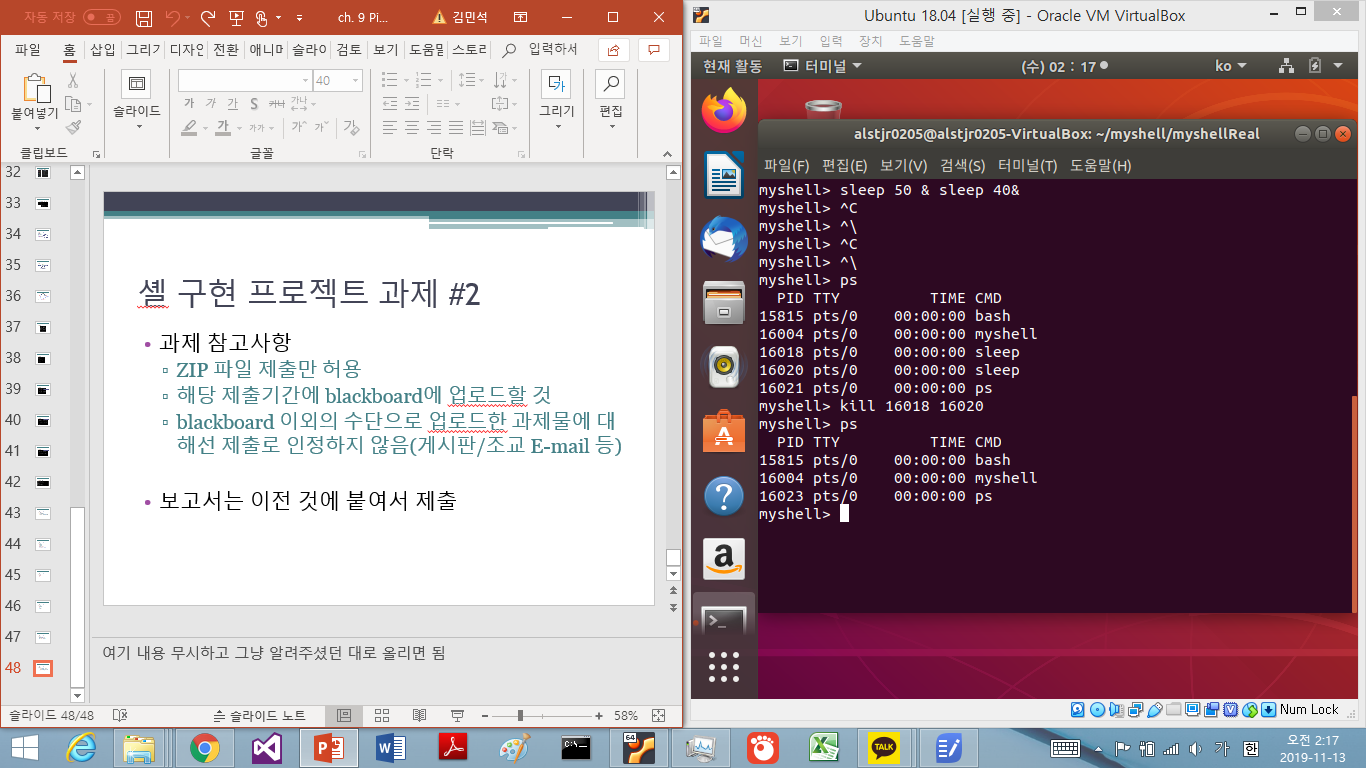
sleep 10이 sleep 30&과 sleep 40 &에 방해받지 않고 수행되었고, 각각 30초 후와 40초 후에 좀비가 생성되지 않고 백그라운드 프로세스가 정상종료되었다.

- Foreground 실행 시 Interrupt



제어키 ^C, ^\ 를 받아서 sleep 40, sleep 60이 죽는 것을 확인했고, 쉘 프롬프트 상에서는 SIGINT와 SIGQUIT 신호를 무시하는 것을 알 수 있었다.

- Background 실행 시 SIGINT, SIGQUIT 무시

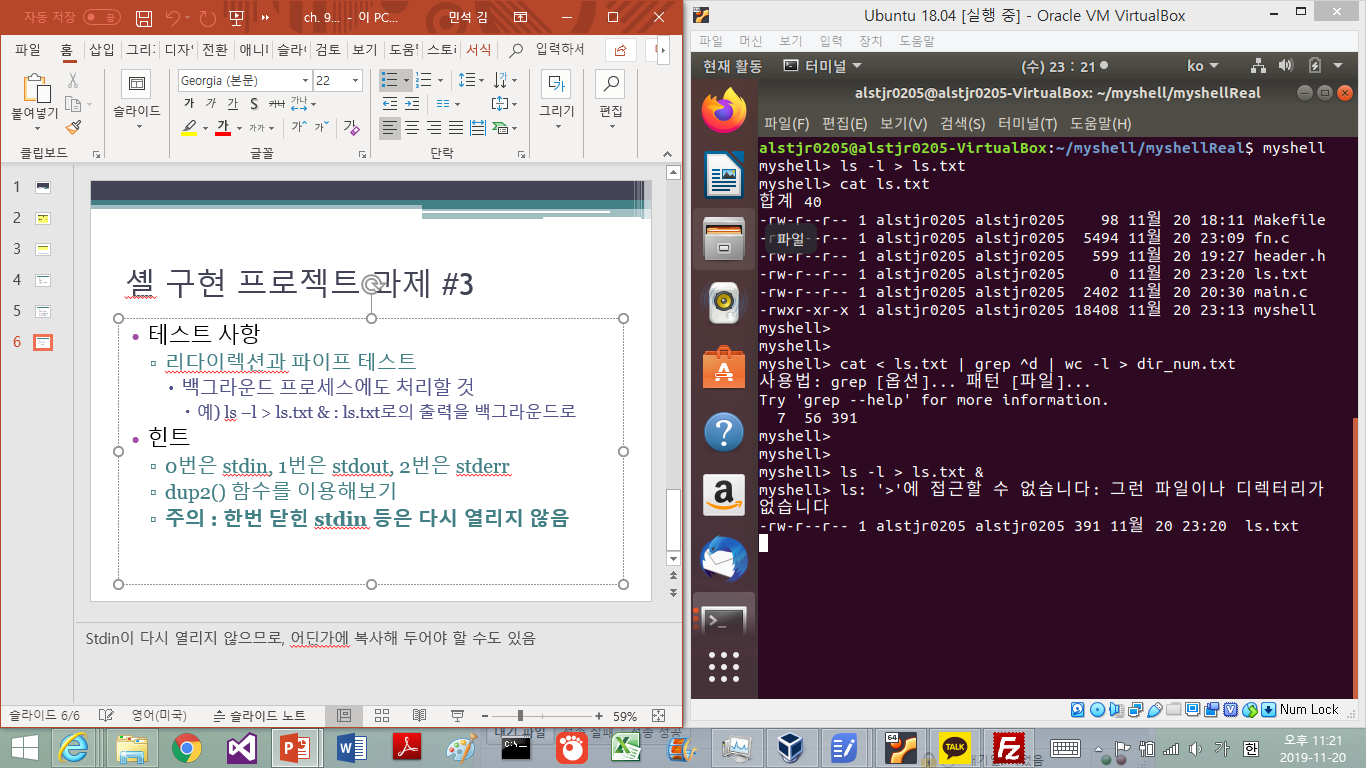
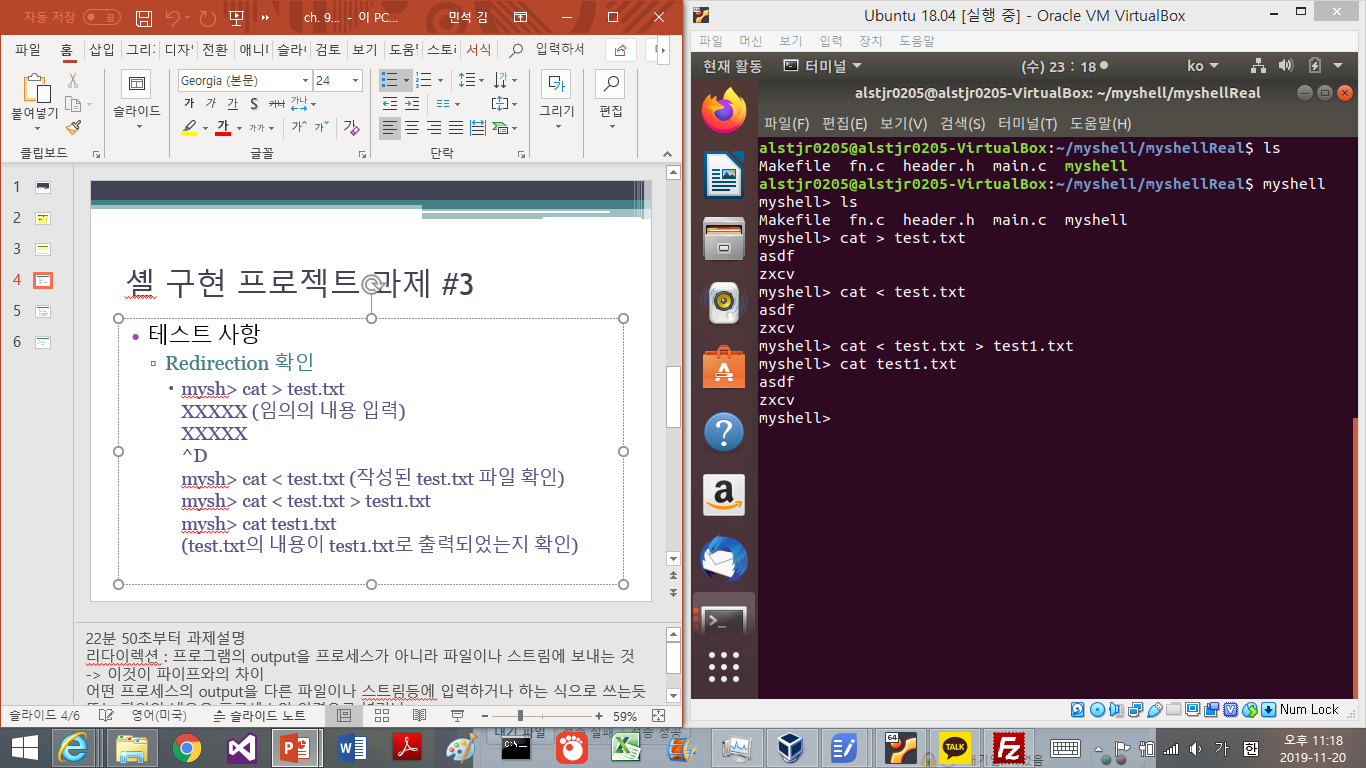


`

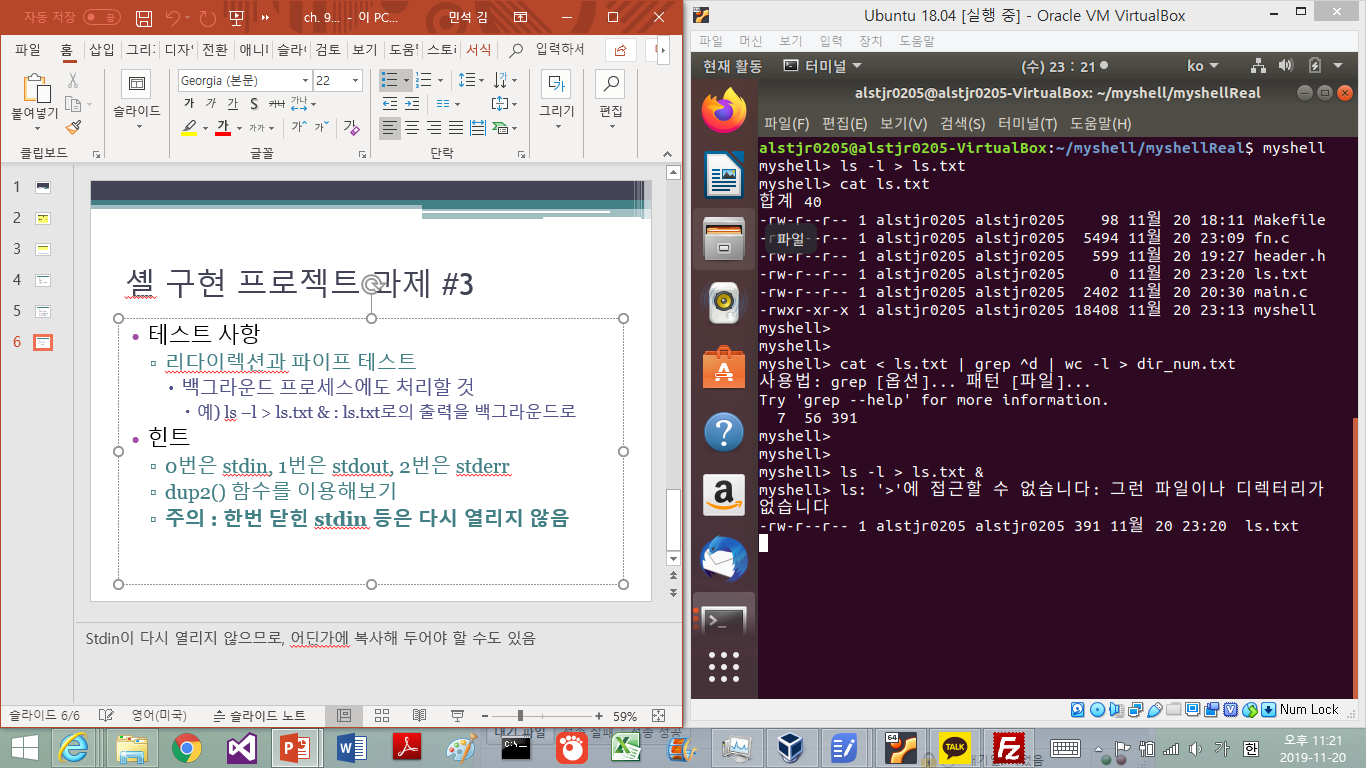
sleep 50& 과 sleep 40&이 SIGINT, SIGQUIT 신호를 블럭시켜서 종료되지 않는 것을 실험했다.

2.5 파이프, 리다이렉션

- 먼저, 리다이렉션을 구현하기 위해 전반적으로 코드의 구조를 수정했다. main에 있는 specialcase 처리와 fork() 수행을 하는 부분을 execute\_small(char\* cmd)함수에 넣었다. 이 함수는 전달받은 명령줄 cmd에 대해 specialcase, 리다이렉션을 처리해서 fork()를 통해 실행해주는 함수이다. 또한, execute\_pipe(char \*cmd) 함수는 cmd의 각 문자열을 '|' 를 구분문자로 해서 토큰화한 후에, 각 명령을 수행하는 함수이다. 리다이렉션 부분은 잘 수행되었고, 이에 따라 execute\_small 함수가 리다이렉션까지 처리해주는 명령어로 잘 동작하게 되었다. 아래의 두 사진은 리다이렉션의 실험결과이다.



하지만, 파이프 구현 후에 파이프 명령어를 진행하는 도중에 프로세스가 멈춰버리는 것을 확인했다. ls | cat | wc 명령어를 수행했을 시에 cat을 수행하는 과정에서 입출력의 오류가 발생했다. 또한, 백그라운드에서의 리다이렉션 구현은 execute\_small, execute\_pipe 함수와는 별개로 main에서 구현해서 백그라운드로 수행되지 않았다. 아래의 사진은 파이프와 백그라운드의 실험결과이다.



3. 구현 시 문제점

3.1 cd 명령

리눅스에 명세된 방법과 구글링을 통해 유사하게 구현한 cd명령어이므로 cd와 완전히 같진 않다. 예를 들어, 리눅스 쉘에서의 cd명령어에 있는 옵션은 myshell에서 구현되어 있지 않다.

3.2 exit 명령

구현할 때 부모에서 실행하도록 예외처리를 해주면 되는 부분이어서 큰 문제점은 아직 발견되지 않았다.

3.3 백그라운드 실행 (&)

리눅스에서는 백그라운드 명령 실행 성공시 “[번호] PID” 형태로 출력을 해주는데 myshell에서는 구현하지 않았다. 백그라운드 프로세스의 개수를 관리하는 방법이 애매했고, 이후에 백그라운드 프로세스가 종료되면, 그 시점으로부터 다음 명령어 입력 시 “[번호]+ 완료 명령어” 형태를 출력하는 것 또한 구현이 불가능했기 때문이다. 종료 시에 출력이 불가능하다고 생각한 이유는 백그라운드 프로세스가 exec를 통해 코드를 변환시키므로 내가 원하는 대로 출력하도록 코드를 수정할 수 없기 때문이었다. 또한, 백그라운드 프로세스가 종료될 때 시그널을 받는 프로세스는 init이어서 쉘 프로세스에게 종료를 알리기가 어려울 것이라 생각했다.

3.4 시그널 처리

원하는 대로 구현하는 것에는 성공했지만, main상에서 모든 구현이 이루어져서 main.c가 조금 복잡해졌다. 이후에 fn.c에 sig\_handler와 blockintquit를 옮기고, 전역변수들을 header.h에서 선언해서 코드를 분리해봤지만, 세그멘테이션 오류가 발생했었다. 결국 정확한 이유는 알아내지 못했고, 추측으로는 전역변수를 사용하는 부분에서 미숙한 점이 있었거나, sigsetjmp와 siglongjmp의 각각 위치가 서로 다른 파일에 존재했기 때문일 것이라는 생각을 해봤다.

3.4 파이프, 리다이렉션

리다이렉션에 대한 명령을 처리하는 부분에서는 문제가 없었지만, 파이프를 구현하는 과정에서 문제가 발생했다. 기본적인 실행구조는 cmdline에서 '|' 발견시, makelist를 구분문자 '|'을 통해 토큰화해서 각각 표준입출력을 조정하면서 토큰화된 명령어들을 수행하는 구조였다. 하지만, 구현이 가능했던 단일파이프에 비해 여러개의 명령을 파이프로 처리하는 부분은 구현에 한계가 있었다. dup2() 사용이 완벽하지 못했고, 한번 바꾼 표준입출력을 다시 복구하는 방법 등을 잘 몰라서 문제가 발생했을 것이라 추측했다.

또한, 초기의 설계 구조에서도 한계점을 느꼈다. 알고 있는 시스템콜들을 최대한 활용하여 해당 주차마다 주어진 과제의 기능만을 쉘과 거의 같게 구현하고자 하였다. 결과적으로 백그라운드를 main함수에서 구현함에 따라 main의 코드길이가 점점 늘어났다. 이로 인해 백그라운드 실행이 파이프나 리다이렉션과 같은 다른 요소들과 어우러질 수 없었다고 생각한다. 규모가 생각보다 큰 쉘 구현에서 프로그램의 전체적인 실행구조를 설계하는 것이 미숙했다고 생각한다.

비록 완벽한 구현을 하지는 못했지만, 아주 오랜 시간동안 고민하고 코드를 들여다보면서 기반 설계의 중요성과 여러 함수들을 잘 활용하는 것이 중요하다는 것을 깨달았다.