**System Programming Project 4**

이름 : 박찬우

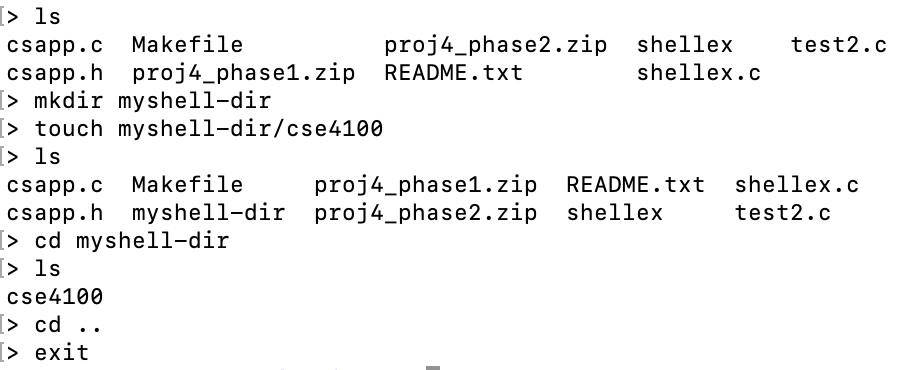
학번 : 20171645

1. **개발 목표**

system-level의 프로세스 제어, 신호 제어, 프로세스 간 통신과 background에서 프로세스를 동작시키고 제어하는 기능 등을 구현하여 간단한 Linux shell을 구현하고, phase 1부터 phase 3에 이르기까지 순차적으로 기능을 구현해 shell 및 system programming에 대한 전반적인 이해를 높인다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1

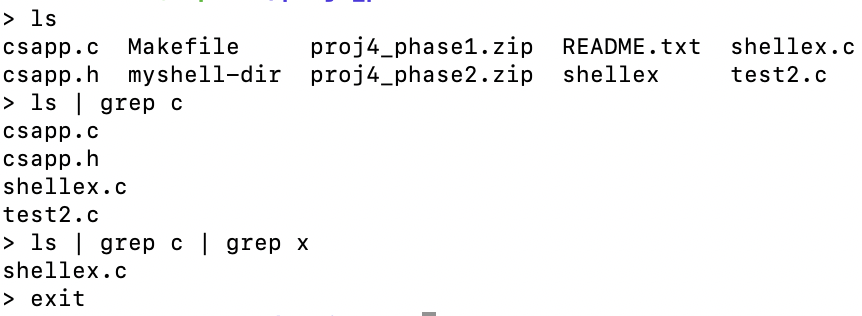
cd, ls, mkdir, rmdir, touch, cat, echo, exit 등의 system call을 입력받으면 그에 맞는 동작을 한다.



ls, mkdir, touch, cd 등에 대해 정상적 shell처럼 동작함을 확인할 수 있다.

1. Phase 2

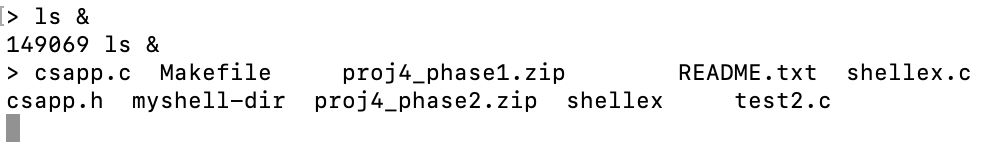
pipeline을 활용한 system call을 수행할수 있는 기능을 구현한다. 특히, ls -al | grep a | grep b | grep c 등의 다중 pipeline 역시 수행할 수 있도록 신경쓴다.

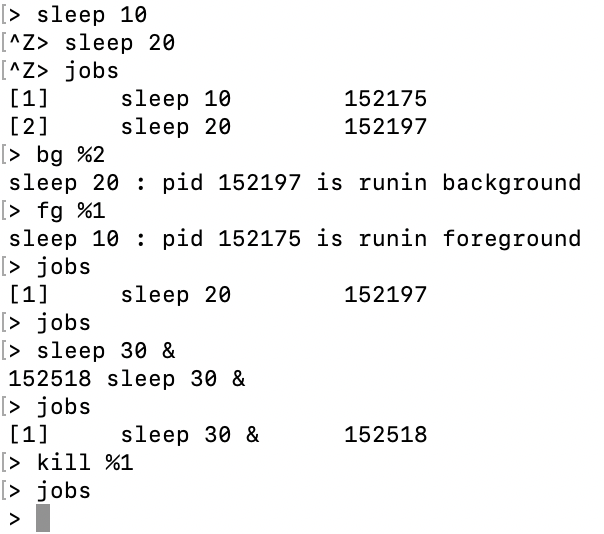


다중 파이프에 대해서도 제대로 적용된 결과나 나옴을 확인할 수 있다.

1. Phase 3

프로세스를 background에서 동작시킬수 있는 기능을 구현하고, 현재 수행중인 프로세스를 확인하는 jobs, 멈춰있거나 background 동작중인 프로세스를 foreground 에서 동작하도록 하는 fg, 멈춰있는 프로세스를 background 에서 동작하도록 하는 bg, 프로세스를 종료하는 kill 명령을 구현한다.

ls 명령을 background에서 실행시켜, > 출력 이후 ls 결과가 나온걸 볼수있다.



sleep 10, 20을 cltr+Z를 통해 중단시킨 뒤 jobs 를 통해 해당 프로세스가 남아있음을 확인하고, bg, fg를 통해 각 프로세스를 background, foreground에서 수행한다. sleep 30 프로세스를 중단한 뒤 kill 명령어를 통해 강제로 종료시킨다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

quit, exit, cd 등의 built-in 명령어가 아니라면, pid = fork() 를 통해 fork를 수행한 후, if (pid == 0) 을 통해 자식 프로세스와 부모 프로세스를 구분한다. 자식 프로세스는 execvp(argv[0], argv) 를 통해 parsing 된 문자열 argv를 활용한 system call을 수행한다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

위와 달리, 부모 프로세스는 waitpid(pid, &status, 0)을 통해서 자식 프로세스가 종료될때까지 대기한다. 자식 프로세스가 execvp() 후 종료되면, 종료되었다는 signal을 보내게 되고, 그 signal을 부모 프로세스가 waitpid()를 통해 받으면 루프문을 돌아 다시 처음으로 돌아가 system command를 입력받게 대기하게 된다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

pipe\_idx[MAXARGS] 배열을 선언하고, parsing된 문자열인 argv[]를 확인해 ‘|’ 인 문자열의 위치를 pipe\_idx[]에 저장하고, 그 수인 pipe\_count 역시 저장한다. 이후 int pipes[MAXARGS][2]를 선언하고, pipe 갯수에 따라 pipe(), close(), dup2() 함수를 통해 연결하도록 구현했다. 자세한 내용은 후술함.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

pipeline 함수를 run 1st command, run 2~final-1 command, final command 3가지 파트로 나누어, run 1st command 파트에서는 fork() 및 pipes[0]에 대한 pipe, dup2를 수행한다. 다음 파트에서는 for문을 통해 i=0부터 i=pipe\_count-1까지 fork() 및 pipes[i]에 대해 pipe, dup2를 수행한다. dup(pipes[i][0], STDIN\_FILENO) 로 입력받아 dup(pipes[i+1][1],STDOUT\_FILEN O)로 다음 파이프에 저장하는 식으로 진행된다. 마지막 파트는 pipes[pipe\_count-1]에 대해 fork(), pipe, dup2를 수행한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

parsing 과정에서 ‘&’를 검사해 background process인지 아닌지 확인해 이를 bg에 반환한다. system call을 수행하며 fork()할때 부모 프로세스는 bg를 확인하고, background process가 아니라면 위 phase1처럼 waitpid(pid,&status,WUNTRACED)를 통해 자식 process를 wait하고, background process라면 wait하지 않게된다. 자세한 구현 및 bg, fg, jobs, kill에 관한 내용은 후술함.

* 1. **개발 방법**

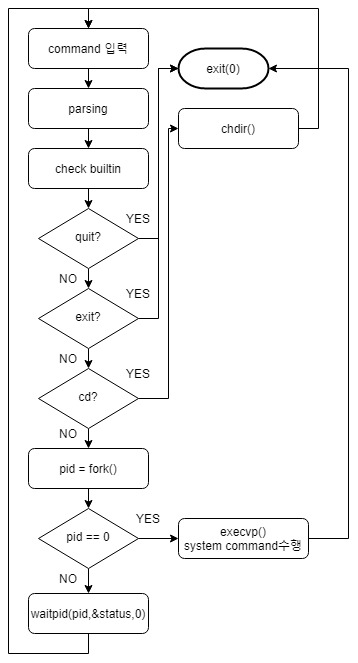
phase1, phase2는 앞서 작성한 내용을 충실히 코드로 작성한다.

phase3의 경우, 우선 process의 list를 관리할 구조체 process\_list를 만든다. 이 구조체에는 각 프로세스의 pid, command 및 활성 상태를 저장한다. 앞서 언급한 system call 호출전 fork() 뒤의 부모 프로세스는 pid를 process\_list에 저장하고, background인지 foreground인지에 따라 활성 상태를 저장한 뒤, background가 아니라면 waitpid(pid,&status,WUNTRACED)를 통해 cltr + z 입력에 의한 프로세스 중단 signal을 받아 해당 process\_list의 활성상태를 수정한다. background process는 pid와 command만 출력한 뒤 wait 하지않고 바로 넘어간다.

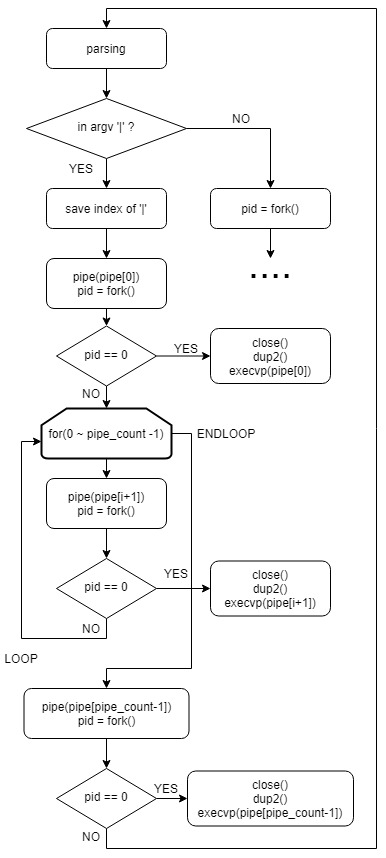
SIGCHILD 신호를 handling할 signal handler를 새로 설정하고, waitpid 값을 process\_list의 pid와 비교해 signal에 따른 process\_list의 활성상태를 조정한다.

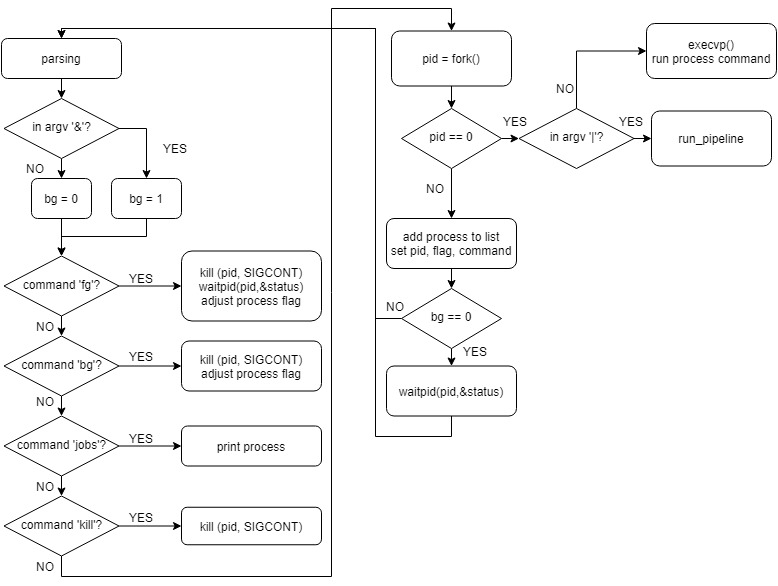
jobs 입력은 process\_list를 확인해 활성된 프로세스를 출력해주고, fg 및 bg 프로세스는 입력받은 pid에 맞는 process\_list의 프로세스 활성 상태를 조정한 뒤, kill(pid, SIGCONT)를 통해 중단된 프로세스면 다시 동작하도록 signal을 보내준다. fg라면 이후 waitpid(pid,&status,WUNTRACED)를 통해 signal에 대한 활성상태를 조정하고, bg라면 wait 없이 넘어간다. kill 명령은 경우는 입력받은 pid에 대해 kill(pid, 9)를 통해 해당 프로세스를 강제로 종료시킨다.

1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**