**System Programming Project 1**

담당 교수 : 김영재 교수님

이름 : 이건영

학번 : 20181662

1. **개발 목표**

수업 시간에 배운 process와 signal 등의 개념을 활용하여 리눅스 Shell의 기능을 수행하는 Myshell 프로그램을 구현한다.

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

ls, cd, echo 등의 shell 내부 명령어를 수행하는 기능을 구현한다.

Bash 내에서의 명령어 실행 결과와 동일한 결과를 출력한다.

1. Phase 2

‘|’ 기호를 사용하여 shell의 pipeline 기능을 구현한다.

‘|’ 기호를 기준으로 앞에 있는 process의 결과가 뒤의 process의 입력으로 전달되어 뒤의 process의 인자로 활용된다.

1. Phase 3

명령어를 background process로 실행하는 기능과 background / foreground process를 관리하는 기능을 구현한다.

Jobs 명령어를 통해 현재 background process의 종류와 상태를 볼 수 있으며, kill 명령어를 통해 process들을 종료시킬 수 있고, bg와 fg 명령어를 통해 해당 프로세스를 background / foreground에서 진행되도록 변경할 수 있다. 또한 Ctrl+Z 커맨드를 통해 foreground process를 멈추어 background로 보낼 수 있다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**

Shell이 실행되면 종료되기 전까지 사용자로부터 명령어를 입력 받는다. 입력 받은 명령어가 built-in command에 해당된다면 builtin\_command 함수 내에서 처리한다. 만약 그 외의 명령에 해당된다면 child process를 Fork 함수를 통해서 생성하고, child process 내에서 execve 함수를 통해 사용자가 입력한 명령어를 실행한다. 이 때 parent process는 waitpid 함수를 통해 child process의 pid를 이용해 해당 process가 종료되면 reaping 하도록 구현하였다. 이후 다시 사용자의 입력을 받는 단계로 돌아간다. 만약 사용자가 ‘quit’ 또는 ‘exit’을 입력했다면 shell을 종료한다.

* **Phase2 (pipelining)**

사용자로부터 받은 입력에 ‘|’ 문자가 존재한다면 기존 명령어 처리 부분에서 pipe 부분으로 넘어간다. 이후 ‘|’ 문자를 기준으로 명령어를 parsing 해주는데, ‘|’ 문자를 발견할 때마다 개수를 기록한다. 이후 pipe\_command 함수를 실행한다.

pipe\_command 함수는 재귀적으로 구현했으며 앞의 명령어를 실행하여 출력한 fd를 다음 pipe\_command 호출 시에 넘겨주었다. 이후 마지막 명령어까지 실행된 이후에 waitpid로 reaping 후 사용자 입력 대기 상태로 돌아간다.

* **Phase3 (background process)**

사용자의 입력에 ‘&’이 존재하는지 확인한다. 만약 존재한다면 parseline 함수의 return 값이 1이 되어 background 처리 부분으로 넘어간다. 해당 command line, process id를 jobList에 추가하고 사용자의 입력을 받는 상태로 돌아가도록 구현했다. 또 foreground에서 작업이 진행중일 때 Ctrl+Z를 입력한다면 해당 process를 중지시키고 background로 보낸다.

bg 명령어를 사용하여 background에서 중지된 process를 다시 run 시킬 수 있으며, fg 명령어를 사용해 background의 process를 foreground에서 run 시킬 수 있다. 또한 kill 명령어를 사용해 background의 process를 종료시킬 수 있다.

* 1. **개발 방법**
* **Phase1 (fork & signal)**

우선 명령어는 built-in command인 것과 아닌 것으로 나뉜다. 우선 bulit-in command에 해당하는 명령어(exit, quit, cd)는 builtin\_command 함수 내에서 실행된다.

exit / quit : exit(0)으로 넘어가게 해 프로그램을 종료시킨다.

cd : chdir 함수를 이용하여 디렉토리를 변경한다.

이외의 명령어는 execve 함수를 이용해서 /bin 또는 /usr/bin 에 있는 프로그램을 실행하도록 구현하였다. 이를 위해 Fork 함수를 이용해서 자식 프로세스를 생성했다. 이 때 자식 프로세스에서는 interrupt와 stop signal을 받아야 하므로 signal 함수로 handler를 재설정해줬다. 부모 프로세스에서는 자식 프로세스의 process id를 받아서 waitpid 함수로 reaping 해주고 eval 함수를 리턴하여 다시 main의 루프 속으로 돌아갈 수 있도록 처리했다.

/bin/과 /usr/bin/ 경로는 PATH, PATH2 배열에 저장하여 입력받은 argv[0]에 더해 execve 함수를 실행할 때 사용되도록 구현하였다.

추가적으로 따옴표 안의 공백을 잘 받아올 수 있도록 parseline 함수 내에서 내용을 추가했다. 처음 발견하는 따옴표의 위치를 기준으로 다음 따옴표를 발견한 후 그 사이의 내용을 보존하는 식으로 구현하였다.

* **Phase2 (pipelining)**

Phase1의 구현 내용에 덧붙여 eval 함수에 입력받은 command line에 파이프 기호 ‘|’ 가 있는지 확인하는 부분을 추가했다. 또 argv 배열을 3중 배열로 만들어, 파이프를 기준으로 나뉜 명령어들을 추가적으로 구분해주었다.

이후 프로젝트 명세의 Hint를 참고하여 pipe 처리를 재귀적으로 해결하기 위해 pipe\_command라는 함수를 추가했다. 이 함수는 argv와 count, file descriptor를 인자로 받는다. pipe\_command 함수에서는 우선 전달받은 명령어가 파이프의 마지막에 해당하는지 체크한다. 만약 마지막 명령어가 아니라면 pipe 함수로 fd 배열에 file descriptor를 전달받는다. 이후 Fork 함수로 자식 프로세스를 생성하고, 자식 프로세스의 입력 fd를 함수의 인자로 전달받은 fd로 바꾸어준다(최초 호출 시에는 STDIN 이다. )출력 fd는 pipe의 출력 fd로 바꾼다. 이후 eval 함수에서처럼 built-in command와 execve 과정을 거친다.

부모 프로세스에서는 waitpid와 더불어 pipe\_command 함수를 재귀적으로 호출한다. 이 때 인자는 pipe 함수로 생긴 입력 fd와 count + 1을 넘겨주는데, count 변수는 이 명령어가 파이프에서 몇 번째 명령인지를 알려준다.

만약 마지막 명령어일 경우 입력 fd만 변경해준 후 built-in command와 execve 실행을 통해 최종적으로 명령을 실행한다. 이후 eval 함수의 waitpid로 인해 reaping 된다.

* **Phase3 (background process)**

우선 프로세스를 쉘에서 관리하기 쉽게 만들기 위해 jobs라는 구조체를 선언하였다. Jobs에는 process id(pid), job id(id), 입력받은 명령어(cmd), 프로세스의 상태(state)가 포함되어 있다. 프로세스의 상태 state는 NONE(없는 프로세스), RUNF(fg에서 실행중), RUNB(bg에서 실행중), SUSB(bg에서 중지), DONE(정상적으로 종료됨), KILL(임의로 종료됨)으로 구분된다. 이 job 타입의 배열 jobList를 이용해 job들을 관리하고자 하였다.

추가적으로 jobList에 job을 추가하는 addjob, 해당 job id의 jobList를 비우는 clearjob, job 목록을 출력하는 printjob(jobs 명령어에 사용된다. ), job이 완료되었는지 확인해서 완료 메세지를 출력해주는 donejob을 구현하였다.

우선 main의 반복문에서 SIGCHLD 시그널에 대한 핸들러를 지정해주었다. 이 SIGCHLDhandler 함수에서는 종료된 프로세스의 pid를 가진 job을 jobList에서 찾아 그 상태를 Done으로 변경한다.

그리고 Ctrl+z를 처리하기 위해 SIGTSTPhandler를 구현하였는데, 이는 Ctrl+z로 SIGTSTP 시그널이 들어오면 해당 pid를 가진 프로세스를 kill 함수를 통해 중지시키고 jobList에 SUSB의 state로 추가한다.

Background 프로세스를 처리하는 방법은 기본적으로 사용자의 명령어의 마지막 글자가 ‘&’이면 bg 변수를 1로 설정한다. Bg가 0인 경우는 phase2 까지의 처리와 동일하다.

Bg가 1이라면 해당 명령을 addjob 함수를 통해 jobList에 추가한다. 그리고 waitpid 함수를 호출하지 않는다. (reaping은 핸들러가 담당한다) 이를 통해 background에서 명령어가 실행되게 된다.

만약 bg가 0인 프로세스가 실행 중 Ctrl+z 명령어를 입력받는다면 핸들러에 의해 프로세스가 중지되어 jobListd에 추가된다.

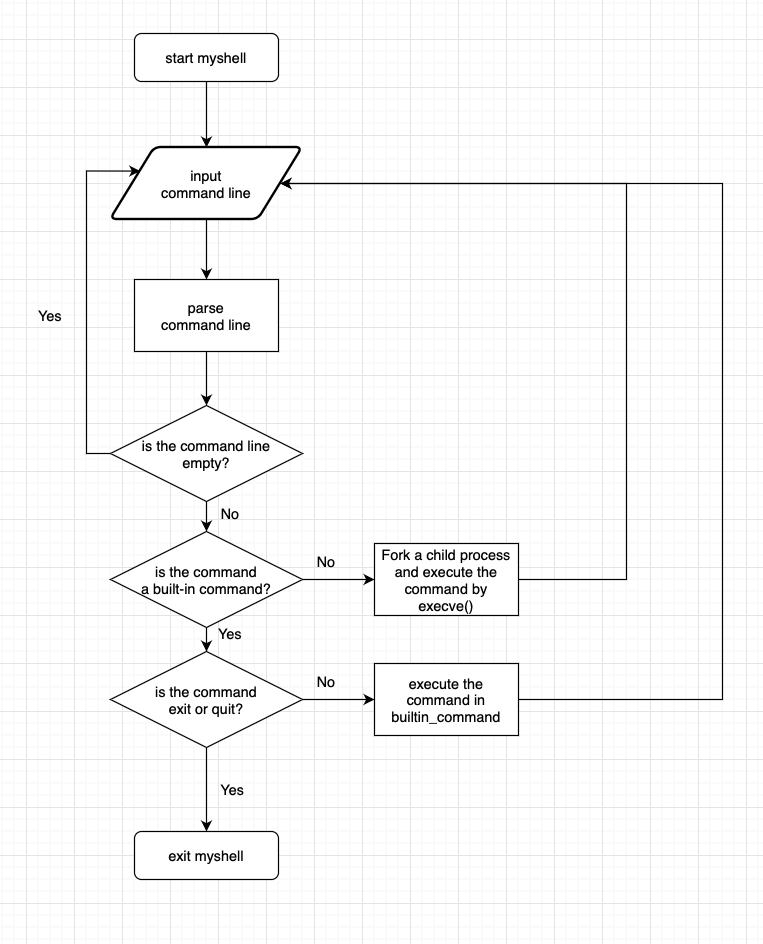
Bg %jobid 명령어를 사용하면 jobList에서 jobid가 일치하는 프로세스를 찾는다. 만약 해당 프로세스의 상태가 SUSB가 아니라면 에러메세지를 출력한다. 만약 SUSB 상태의 프로세스라면 해당 job의 pid를 받아와 Kill 함수로 SIGCONT 시그널을 보내준 후, state를 RUNB로 수정한다. 이를 통해 background에서 프로세스를 running 상태로 만들어줄 수 있다.

Fg %jobid 명령어를 사용하면 jobList에서 jobid가 일치하는 프로세스를 찾는다. Bg와 비슷하나 fg의 경우 background에서 이미 running 중인 프로세스도 대상이기에 state가 RUNB 이거나 SUSB인 프로세스가 아니면 에러메세지를 출력한다. 만약 RUNB 이거나 SUSB 상태라면 마찬가지로 Kill 함수로 SIGCONT 시그널을 보내준 후 state도 RUNF로 수정한다. 이후 무한 루프속으로 들어가 jobList에서 해당 jobid의 state가 RUNF가 아니게 된다면 break로 탈출한다. 이는 SIGCHLDhandler에 의해서 state가 done으로 바뀔 경우가 종료조건에 해당되도록 구현한 것이다.

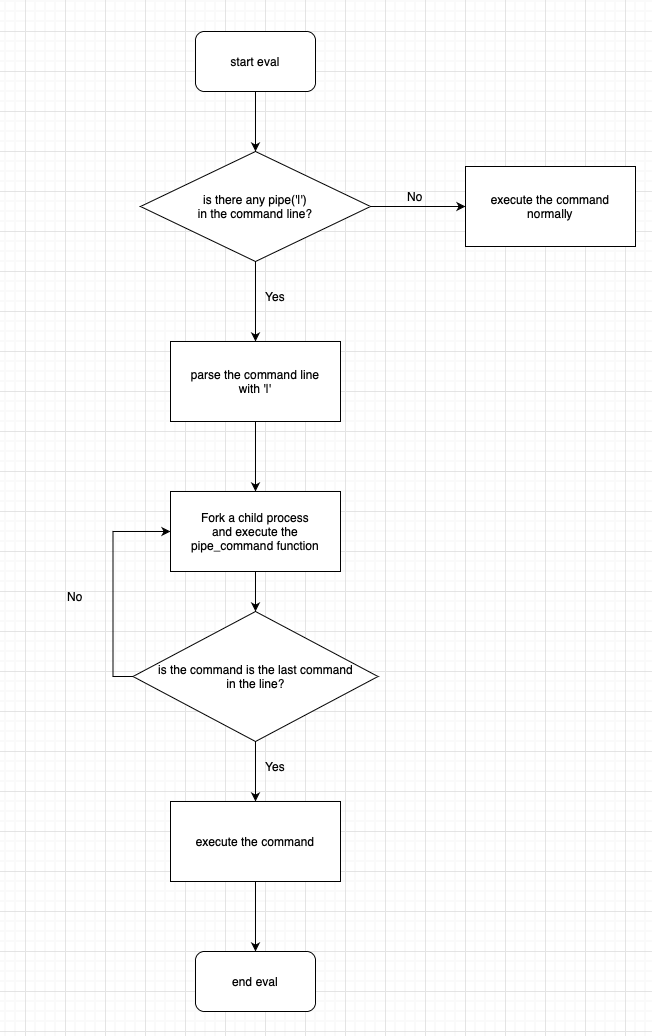
Kill %jobid 명령어를 사용하면 jobList에서 jobid가 일치하는 프로세스를 찾는다. 해당 프로세스가 종료 가능한 상태(SUSB, RUNB)라면 Kill 함수로 SIGKILL 시그널을 해당 프로세스에 전달한 뒤 state를 KILL로 변경한다. 이후 donejob 함수가 실행될 때 KILL에서 NONE으로 state가 변경된다.

Jobs 명령어를 사용하면 현재 printjob 함수를 사용하여 현재 jobList를 출력한다. 이 때 출력하는 job의 state는 RUNB(running), SUSB(suspended) 두 종류로, DONE의 경우 donejob 함수에서 프로세스가 종료될 경우 매번 출력해주고, KILL의 경우 바로 NONE으로 바꾸어주므로 출력될 경우가 없다.

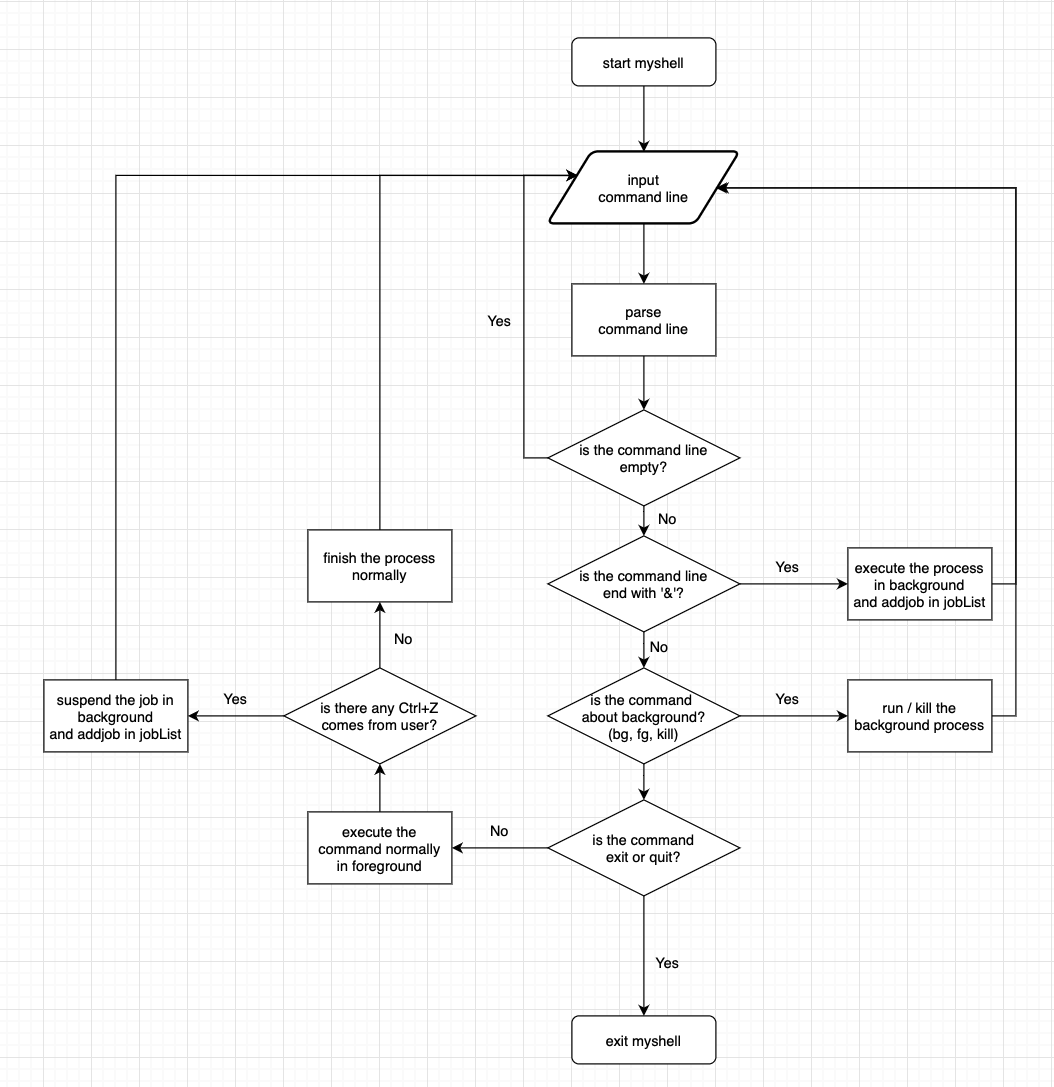
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****