**System Programming Project 1**

담당 교수 : 김영재

이름 : 정지석

학번 : 20181687

1. **개발 목표**

* 본 프로젝트에서는 수업 시간에 다룬 processing, signaling, background, pipe line, fork 등의 개념을 사용하여 자신 만의 리눅스 기반 shell 프로그램을 c언어를 사용하여 구현한다.

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

리눅스 shell의 기본적인 명령어들을 수행할 수 있다. Shell 프로그램은 ‘exit’ 명령어를 받기 전까지 계속 입력을 받는다. cd, ls, mkdir, rmdir, touch, cat, echo 와 같은 명령어들을 입력 받아, 그에 알맞은 기능을 수행한다.

1. Phase 2

Phase1의 기능에 더해 pipe line 기능이 추가된다. 기호 ‘|’를 사용하여 pipe line을 나타내고, 이를 명령어들 사이에 입력하면 해당 명령어 들을 서로 연결해 준다.

1. Phase 3

Phase1,2의 기능에 background/foreground process 의 실행 및 관리를 하는 기능이 추가된다. ‘&’ 기호를 명령어 뒤에 입력하면 해당 명령어를 background에서 실행하는 것이 가능해진다.

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

shell에서 입력 받은 명령어가 built in command(따로 예외처리를 한 command)에 해당이 될 경우, 따로 만든 함수에서 이를 처리한다. 그렇지 않은 경우는 fork 함수를 사용하여 자식 process를 생성한다. Fork 함수를 실행하면 이에 대한 pid를 확인할 수 있는데, 만약 성공적으로 자식 process를 생성한 경우 pid가 0이 된다. 따라서 fork 함수를 실행한 후 pid를 확인하여 자식 process가 제대로 생성이 되었는지를 확인하고, 잘 생성이 되었다면 자식 process에서 execvp 함수를 실행하여 입력 받은 명령어를 실행한다.

부모 process의 경우 waitpid 함수를 사용하여 자식 process가 모두 종료될 때 까지 기다리도록 구현하였다. 만약 process가 제대로 종료가 되지 않으면 에러 메시지를 출력하고, 제대로 종료가 되었다면 다시 사용자의 입력을 받는다. 오류로 인한 종료나 shell에 exit가 입력되어 종료되기 전까지 위의 과정을 반복한다.

* **Phase2 (pipelining)**

입력한 명령에 pipe line(‘|’)이 없다면 phase1에서 했던 대로 명령어를 처리한다. Pipe line이 존재하는 경우에는 ‘|’ 기호를 기준으로 하여 명령어를 나누어 준다. 나누어진 명령어는 each\_command 라는 배열에 저장해주고, 각각의 each\_command는 phase1에서 명령어를 처리했던 방법과 동일하게 실행된다. 그리고 이에 더해 file descriptor와 dup2 함수를 사용하여 pipe line을 통해 나누어진 명령어들을 서로 연결해 준다.

While 문을 사용하여 전체 명령어가 모두 실행될 때 까지 위 과정이 반복되므로, pipe line의 개수가 여러 개라 할지라도 위와 동일한 방법대로 명령어를 처리한다.

* **Phase3 (background process)**

먼저 입력한 command line이 background process인지 확인하기 위해 명령어 제일 뒤에 ‘&’ 기호가 있는지를 parseline 함수에서 확인한다. ‘&’ 기호가 존재할 경우 1을 return 하고 , 이 값과 입력 받은 command line, pid를 job\_list structure에 추가한다. Foreground process가 실행 중일 때에 ctrl-z를 입력하면, 해당 process는 중단된 상태로 background로 넘어가게 된다.

이외에도 jobs, kill, fg, bg와 같은 background 관련 명령어들을 처리할 수 있게 구현하였다.

* 1. **개발 방법**
* **Phase1**

먼저 입력 받은 명령어를 pasrseline 함수에 넘겨 space를 제거하고 각각의 명령어를 하나의 배열에 저장한다. “ 나 ‘ 같은 따옴표의 경우, 따옴표 사이에 있는 문자열은 space를 제거하지 않고, 따옴표만 제거한 채로 그대로 하나의 문자열로 저장한다.

이렇게 parsing한 명령어를 먼저 builtin\_command 함수를 실행하여 따로 예외처리를 하였는지 확인한다.

Built in command에는 exit와 cd가 있다.

Exit이 입력될 경우 exit(0)를 실행하여 shell 프로그램을 종료한다.

Cd가 입력될 경우 chdir 함수를 실행하여 원하는 directory로 이동하는 것이 가능해진다.

위의 경우에 해당하지 않는 명령어들의 경우, fork를 실행하여 자식 process를 생성한 뒤, execvp 함수를 사용하여 명령어를 실행한다.

부모 process는 waitpid를 통해 자식 process가 종료될 때까지 기다리고, 모든 종료가 되면 다시 main으로 돌아가 입력을 받을 수 있는 상태로 돌아온다.

* **Phase2**

기본적으로 명령어를 처리하는 방식은 phase1과 동일하다. 다만 입력에 pipe line이 들어오면 추가적으로 처리해야 하는 부분이 생긴다. Pasrseline을 통해 space와 따옴표를 제거해준다.

이렇게 만들어진 argv 배열은 while문에 들어가게 되는데, 이 곳에서는 pipe line(‘|’)을 기준으로 명령어를 나누는 작업을 수행한다. 나누어진 명령어는 임시 배열 each\_command에 저장되며, 이 배열은 phase1과 동일하게 명령어를 실행해 준다.

다만 이렇게 명령어를 pipe line 별로 잘라서 실행해 줄 경우 각각의 명령어가 따로 실행 되기 때문에, 각각의 명령어를 연결해주기 위해서는 추가적인 작업이 필요하다.

이를 위해 사용하는 것이 바로 file descriptor와 dup2 함수이다. File descriptor에 이전 명령어의 결과를 저장하여 현재 실행하려는 명령어에 전달해 주면, 현재 실행하는 명령어가 그 입력 받아 명령어를 실행한다.

이 과정은 pipe로 연결된 모든 명령어가 다 실행될 때 까지 반복되며, 이렇게 해서 나온 결과가 화면에 출력된다.

만약 pipe line이 명령어와 붙어 있어 하나의 문자열로 인식이 된다면(ex) cat ex.txt|grep “bus” 와 같이 space로 pipe와 명령어를 구분해 놓지 않은 경우), strtok 함수를 사용하여 해당 문자열을 ‘|’ 기호 기준으로 잘라서 각각 each\_command에 저장하는 식으로 문제를 해결하였다.

* **Phase3**

기본적인 명령어 수행은 phase1,2와 동일하나, process 들을 관리해야 하기 때문에 job이라는 structure를 선언하였다. 이 structure 안에는 명령어 자체와 pid, 상태(foreground 인지 background 인지), 고유의 id가 저장되어 있다.

명령어를 입력 받아 parseline을 실행하면 명령어 끝에 ‘&’ 기호가 있는지를 체크하여 background 인지 여부를 확인하여 return 한다. 그 값을 받아 pid, cmdline과 함께 job 타입으로 선언된 job\_list 배열에 넣어준다(이 과정은 직접 구현한 insert\_job 함수가 실행한다). 이런 식으로 process 들을 저장하여 관리하는 것이 phase3의 기본적인 구조이다.

Jobs, kill, bg, fg 명령어의 경우 builtin\_command 함수에 해당 명령어 들을 처리하는 부분을 추가해 주어야 한다.

Jobs의 경우 실행 중인/중지된 background를 모두 출력해주는 명령어이다. 각 job\_list 배열의 state를 확인하여 background/stop 상태인 것들을 출력해준다.

Kill %n(n은 자연수) 를 입력하면 id가 n인 process를 삭제한다. 해당 job의 state를 kill로 바꾸고 해당 process를 kill 함수를 이용해 없앤다.

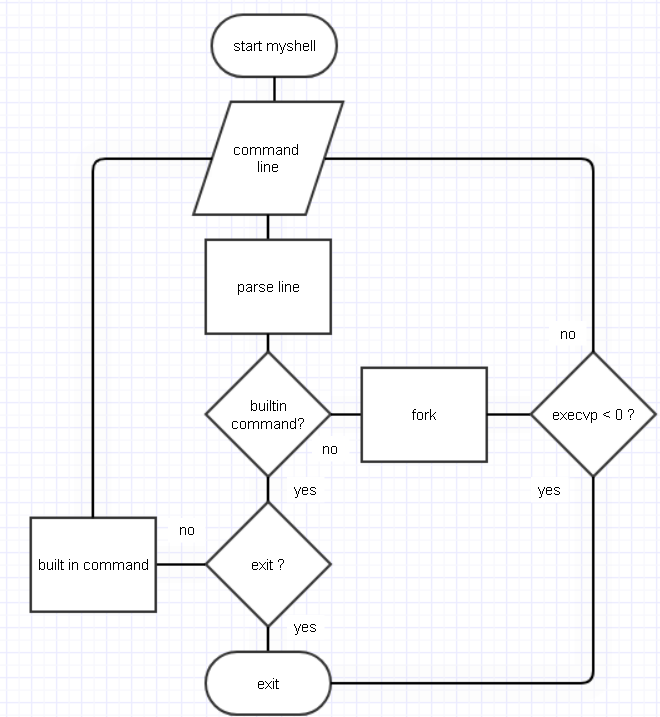
Bg, fg는 id가 n인 process를 background/foreground에서 running하게 해주는 명령어이다. SIGCONT 시그널을 보내 멈춰 있던 process가 다시 작동하도록 한다.

이외에도 특정 signal을 관리하기 위한 signal handler들을 작성해주어야 한다.

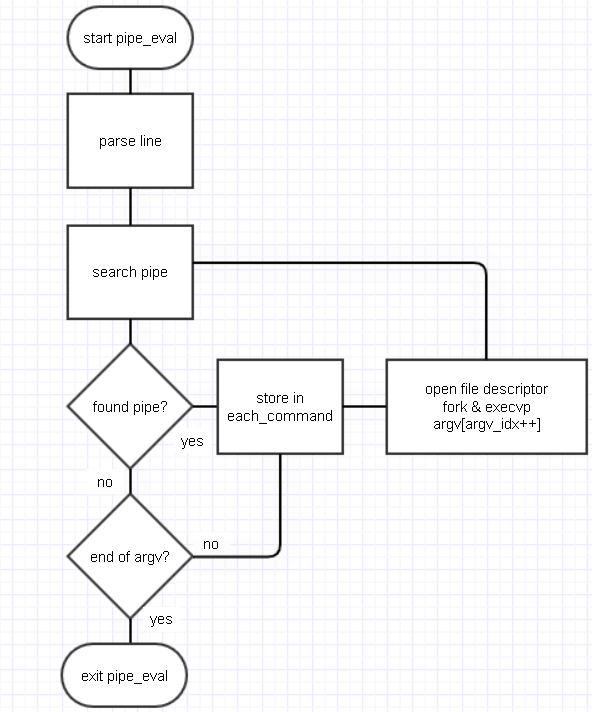
Sigtstp\_handler를 구현하여 입력으로 ctrl-z가 들어왔을 경우 현재 foreground에서 실행 중인 process의 실행을 멈추고 stop 상태로 만들어 준다.

Sighchld\_handler는 종료된 process를 찾아 done 상태로 만들어 주는 역할을 한다.

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

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

