**Multicore Programming Project 1**

담당 교수 : 최재승 교수님

이름 : 이승연

학번 : 20211569

1. **개발 목표**

* 수업시간에 배운 process control, signal 등의 개념을 이용하여 리눅스 쉘의 기능을 수행하는 간단한 쉘을 작성한다.

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

ls, mkdir, rmdir, touch, cat, echo, cd, history등의 기본 내부 쉘 명령을 실행할 수 있다.

Bash 내에서의 명령어 실행 구현 결과와 같은 결과를 출력한다.

1. Phase 2

쉘 기능을 확장해 '|' 기호에 대한 pipeline 기능을 실행한다.

여러 pipeline chain을 명령 줄 인수로 가질 수 있다.

1. Phase 3

'&'를 사용해 background에서 process를 실행한다.

user가 작업의 process 상태를 변경할 수 있으며, background와 foreground 간에 작업을 이동시킬 수 있다.

jobs, bg, fg, kill 명령을 실행할 수 있다.

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

쉘이 실행되면 사용자가 종료하기 전까지 명령어를 입력받는다. 입력 받은 명령어를 builtin\_command 함수에서 쉘의 내장 명령어인지 판단하고 만약 내장 명령어라면 builtin\_command 함수 내에서 처리한다. 내장 명령어에는 cd, history, exit, quit가 있다. 만약 아니라면 Fork 함수로 child process를 생성한다. child process 내의 execve 함수를 사용해 사용자가 입력한 명령어를 실행한다. parent process는 child process를 reaping 하기 위해 waitpid 함수로 child process가 종료할 때까지 기다리도록 구현했다. 이 때 child process의 pid를 사용한다. 사용자가 'exit' 또는 'quit'을 입력하면 쉘을 종료한다.

* **Phase2 (pipelining)**

쉘에서 사용자가 명령어를 입력할 때마다 입력 명령어에 '|'가 있는지 판단하여 파이프 명령어라면 eval\_pipe 함수에서 명령을 처리하도록 했다. 이후 parse\_pipe 함수를 실행해 '|' 기준으로 명령어를 parsing 하고 그 결과를 이차원 배열에 저장하고 파이프 개수와 명령어 개수를 저장한다. eval\_pipe 함수에서는 for문을 통해 명령어 개수만큼 반복한다. 매번 fork 함수를 통해 child process를 호출하고 Dup2 함수를 호출하여 stdin, stdout을 적절히 연결한다. 파이프라인으로 구분된 첫번째 명령어와 마지막 명령어는 따로 처리를 해주었다. child process에서 execvp 함수로 사용자가 입력한 명령을 실행한다. 이 결과는 바로 출력되지 않고 Dup2에서 적절히 연결해 다음 for문의 입력값으로 들어가게 된다. 이후 parent process에서는 더 이상 사용하지 않는 파일 디스크립터를 닫아준다. 마지막 명령어까지 실행됐다면 waitpid로 child process를 reaping한다.

* **Phase3 (background process)**

쉘에서 사용자가 입력한 명령어에 '&'이 포함되어 있는지 판단하여 맞다면 parseline 함수의 리턴값이 1이 되어 background를 처리한다. job\_list 구조체를 만들어서 모든 background process 정보를 저장했다. 현재의 job에 대한 정보를 job\_list에 저장한 후 다시 사용자의 입력을 받을 수 있도록 한다. foreground에서 실행하는 경우 SIGTSTP 신호를 받으면 process를 suspend 시키고 background로 보낸다.

builtin\_command 함수에서 fg, bg, kill, jobs 명령어를 처리한다. fg 명령어로 job\_list[n]을 foreground로 보낸다. process를 다시 실행하기 위해 SIGCONT 시그널을 보낸다. bg 명령어로 job\_list[n]을 background로 보낸다. suspend 되었던 process를 다시 실행한다. 마지막으로 kill 명령어로 background의 process를 종료할 수 있다. 이 때 parent, child process 모두 reaping된다.

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

추가한 함수는 insert\_get\_history, count\_history\_inst, read\_history\_file, history이다.

입력 받은 명령어는 builtin command가 아닌 것과 builtin command인 것으로 나눈다. builtin command에 해당하는 명령어들은 cd, history가 있고 이는 builtin\_command 함수 내에서 처리된다.

cd 명령어 : chdir 명령어를 이용해 디렉토리를 변경해 주었다.

history 명령어 : 쉘을 실행하면 history.txt파일을 열어 count\_history\_inst 함수를 사용해 현재까지 몇 개의 명령어가 텍스트 파일에 저장되어 있는지 history\_inst\_num에 저장한다. 연다. 만약 history.txt.파일이 없다면 새로 만들어준다. history 명령어를 수행시키기 위해 get\_history 라는 이차원 배열을 만들었다. 이후 명령어를 입력받을 때마다 insert\_get\_history 함수를 이용해 history.txt파일에 명령어를 추가하고 만약 직전 명령어와 중복이 아니고 느낌표 명령어가 아니라면 get\_history 배열에 추가해준다. 사용자가 history 명령어를 입력하면 매순간 계산해놓은 history\_inst\_num(배열에 저장된 명령어 개수)로 for문을 사용해 history를 출력한다. 만약 '!!'가 실행됐다면 바로 이전에 수행했던 명령문을 재실행하고 '!n'이 실행됐다면 get\_history의 n번째 줄에 있는 명령어를 실행하고 이 명령어를 다시 get\_history에 넣는다. 사용자가 쉘을 종료할 때 열려있던 history.txt 파일이 닫힌다.

이외의 명령어들은 execve 함수를 통해 /usr/bin/ 또는 /bin/ 디렉토리에 위치한 명령어 경로를 실행할 수 있게 구현했다. Fork 함수를 통해 child process를 생성한다. parent process는 child process가 종료될 때까지 기다린 다음 reaping해주는 작업을 처리해준다. 이후 eval 함수를 리턴하여 다시 main의 while문으로 돌아가서 사용자가 입력을 받을 수 있는 상태로 만들어준다.

* **Phase2 (pipelining)**

phase1의 내용에 eval\_pipe 함수를 추가하여 사용자의 입력 속에 '|'가 포함되어 있었다면 eval\_pipe 함수로 명령어 수행을 처리하는 작업을 추가했다.

추가한 함수는 eval\_pipe, parse\_pipe이다.

main의 while문 안에서 strstr함수를 이용해 명령어에 '|' 가 있는지 판단하고 eval\_pipe 함수를 실행한다. 여기서 parse\_pipe 함수를 실행하는데 이 함수는 주어진 명령어를 '|을 기준으로 parsing하여 argv\_pipe 이차원 배열에 저장하는 역할을 한다. 예를 들어 argv = “cat filename | grep -i “abc” 였다면

argv\_pipe[0] = “cat filename”

argv\_pipe[1] = “grep -i 'abc'”로 저장되는 것이다.

성공적으로 parsing을 마쳤다면 명령어 개수를 저장하고(argv\_idx) 파이프 개수(pipe\_idx)를 리턴한다. eval\_pipe로 돌아와서 argv\_idx만큼 for문을 반복하며 명령어를 수행한다. 반복문마다 Fork 함수를 실행하며 child process를 생성하고 첫번째 명령어가 아닌 경우 이전 명령어의 stdout을 현재 명령어의 stdin으로 연결한다. 그리고 마지막 명령어가 아닌 경우 현재 명령어의 stdout을 다음 명령어의 stdin으로 연결한다. 이후 현재 명령어를 execvp 함수를 이용해 실행하고 사용한 파일 디스크립터를 전부 닫아준다. 이 과정을 마치면 마지막 파이프 명령어가 수행된 결과만 터미널로 출력되게 된다. parent process는 child process가 종료될 때까지 기다린 다음 reaping해주는 작업을 처리해준다. 이후 eval 함수를 리턴하여 다시 main의 while문으로 돌아가서 사용자가 입력을 받을 수 있는 상태로 만들어준다.

* **Phase3 (background process)**

추가한 함수는 시그널 핸들러인 sigint\_handler, sigtstp\_handler, sigchld\_handler, 실행중인 job을 출력하는 print\_job, job을 초기화해주는 clear\_job, job을 추가해주는 add\_job, job 완료시 메세지를 출력해주는 done\_job, 명령어의 마지막 '&'문자를 제거해주는 remove\_ampersand 까지이다.

쉘에서 관리하기 쉽도록 job 구조체를 선언했다. job 타입의 배열 job\_list 선언해서 여기에 job을 넣기로 한다. jobs에는 process id인 **pid**, job의 index를 알려주는 **id**, procsee 상태를 나타내는 **state**, 입력받은 명령어인 **cmdline**이 있다. state는 알아보기 편하도록 define을 이용해 0이면 NONE, 1이면 FG, 2이면 BG, 3이면 SUSP, 4이면 DONE, 5이면 KILL라고 정의했다.

프로그램 시작시 main에서 SIGTSTP, SIGCHLD에 대한 handler를 지정했다. 이는 사용자가 foreground 상태에서 Ctrl+Z를 눌렀을 때 sigtstp\_handler를 실행시켜 foreground에서 실행중이던 process를 SUSP 상태로 바꿔주고 kill 함수를 통해 중지시킨다.

eval 함수에서 사용자 입력 문자열의 마지막 문자가 '&'라면 remove\_ampersand 함수를 이용해 마지막 '&'문자를 지워 add\_char 배열에 복사하고 background에서 실행되므로 bg=1로 설정한다. Fork 함수를 통해 child process를 생성하고 add\_job 함수를 통해 현재 진행중인 job을 job\_list에 추가한다. 이 때 add\_char배열을 넣어서 jobs 관련 메세지를 출력할 때 '&'문자열이 출력되지 않도록 한다. 이후 parent process에서는 foreground인 경우 waitpid를 통해 child process가 종료될 때까지 대기하고 정상적으로 종료되면 job\_list의 job들 중 foreground 상태인 것들에 대해 clear\_job 함수를 호출한다. background인 경우엔 리턴하고 다시 사용자가 입력을 받을 수 있는 상태로 돌려준다. 만약 bg=0인 process, 즉 foreground process가 실행 중에 Ctrl+Z 명령어를 입력 받으면 sigtstp\_handler에 의해 process가 중지되어 job\_list에 SUSP 상태로 변경된다. 사용자가 입력한 명령어에 따른 수행 처리를 마친 후 done\_job 함수를 수행하여 process가 DONE 혹은 KILL 상태이면 작업이 종료되었다는 메세지를 출력한다. eval\_pipe함수에서도 동일한 로직으로 background process를 처리하도록 구현했다.

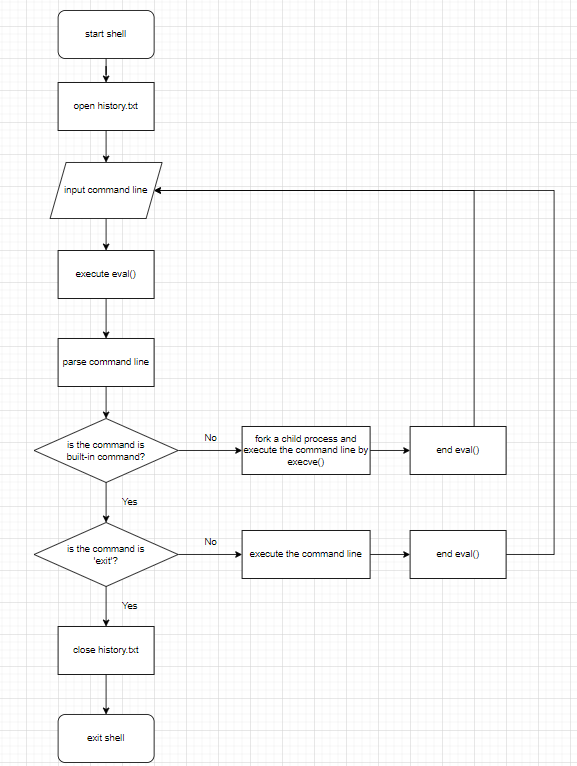
jobs 명령어는 print\_job 함수를 호출한다. print\_job 함수는 현재 실행중인 background 작업과 suspend 작업의 목록을 출력한다.

kill %n 명령어는 실행 중인 작업을 종료한다. 입력받은 n을 통해 job\_list 배열에서 해당 작업을 찾는다. 작업이 존재하지 않으면 “No Such Job”을 출력하고 존재하면 해당 작업의 pid에 SIGKILL 시그널을 보내어 작업을 강제종료시킨다. 이후 sig\_list의 state를 KILL로 변경한다. 이후 done\_job 함수가 수행될 때 state가 KILL에서 NONE으로 변경된다.

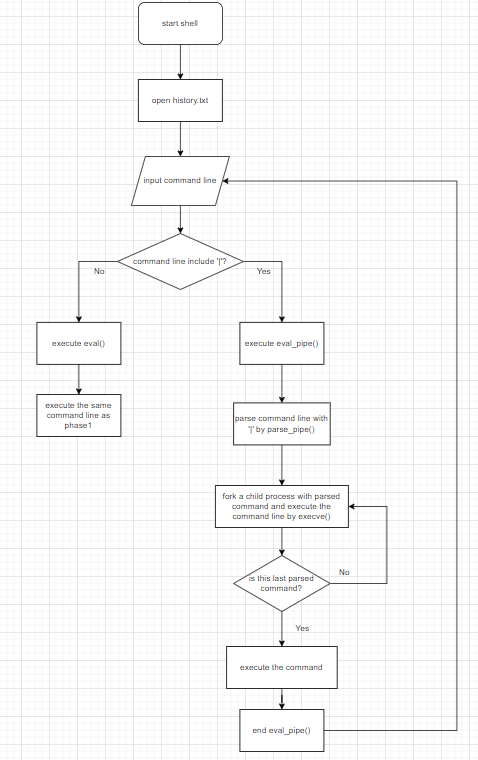
fg %n 명령어는 background에서 실행 중인 작업을 foreground로 전환한다. 입력받은 n을 통해 job\_list 배열에서 해당 작업을 찾고 존재하지 않거나 상태가 DONE, KILL, NONE인 경우 “No Such Job”을 출력한다. 존재한다면 state를 FG로 변경하고 해당 pid에 kill 함수를 통해 SIGCONT 시그널을 보내서 작업을 실행할 수 있도록 한다. 마지막으로 작업 상태가 FG인 동안 while문을 통해 작업이 완료될 때까지 대기한다. 이는 sigchld\_handler에 의해 state가 DONE으로 바뀌는 경우 while문이 종료되도록 하기 위함이다.

bg %n 명령어는 foreground에서 실행 중인 작업을 background로 전환한다. 입력받은 n을 통해 job\_list 배열에서 해당 작업을 찾고 존재하지 않거나 상태가 DONE, KILL, NONE인 경우 “No Such Job”을 출력한다. 존재한다면 state를 BG로 변경하고 해당 pid에 kill 함수를 통해 SIGCONT 시그널을 보내서 작업을 실행할 수 있도록 한다. 이를 통해 background에서 process state를 running으로 만들어준다.

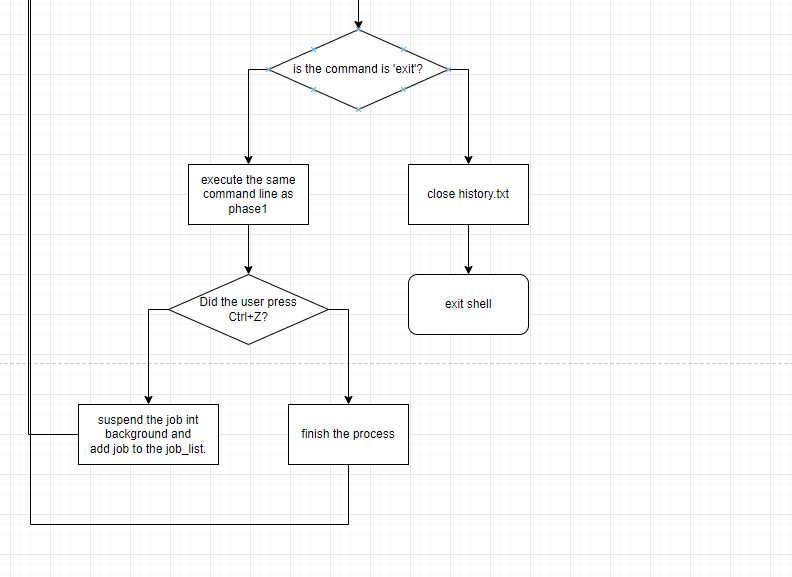
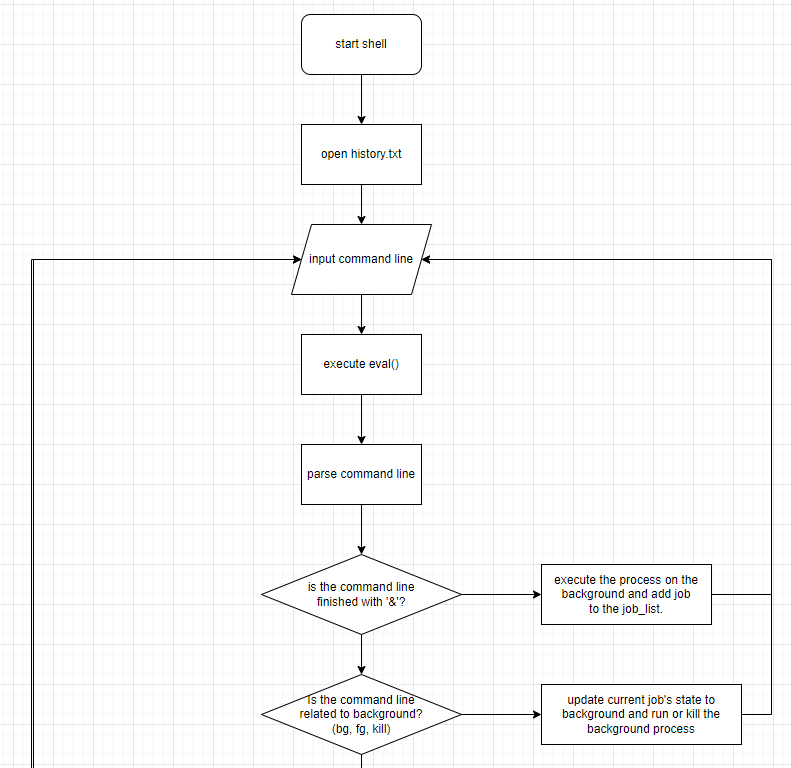
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****