**System Programming Project 4**

담당 교수 : 김영재

이름 : 남주형

학번 : 20171630

1. **개발 목표**

먼저 bulit in command를 구현하고 fork를 통해 child process를 생성하여 다양한 명령어를 처리해준다. 그리고 구현한 명령어로 pipeline에 대해서도 작동이 가능하도록 해준다. 마지막으로 앞에서 구현한 것들을 foreground와 background로 나누어서 실행이 가능하게 구현한다. 이렇게 shell과 유사한 myshell을 만들어준다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명

<phase 1 실행 결과>

bulit in command와 fork를 사용한 명령어를 구현하여 ls, mkdir, touch, cd 등의 명령어를 수행한 것을 볼 수 있다. 따옴표에 대한 처리도 해주었다.

1. Phase 2

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

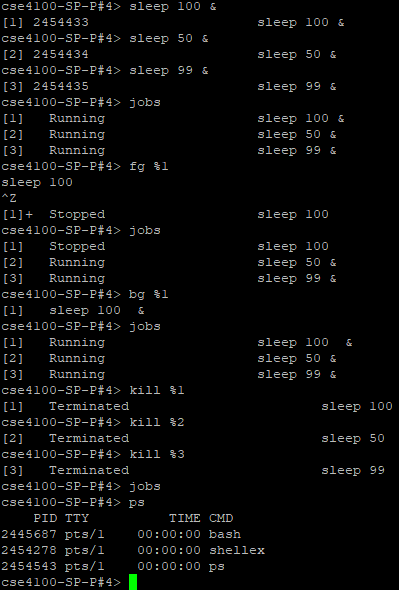
<phase 2 실행 결과>

pipeline에 대한 명령어를 수행한다. 이전 명령어의 출력을 다음 명령어의 입력으로 넘겨주면서 최종적으로 정제된 출력을 screen에 보여준다. recursive를 통해 pipe의 개수에 상관없이 실행되도록 해주었다.

1. Phase 3

텍스트이(가) 표시된 사진

자동 생성된 설명



<phase 3 실행 결과>

명령어에 ‘&’를 붙이면 background에서 실행시킨다. 또한 jobs, fg, bg, kill 함수를 구현하여 이런한 jobs를 관리하였다. sleep함수를 이용하여 background와 foreground 에서 명령어를 수행시키고 CTRL+C와 CTRL+Z를 통해 foreground 명령어를 종료, 중지시키는 등 다양한 실행을 해보았다.

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

builtin\_command가 아닐 때 fork를 한다. child에서는 execvp 명령어를 통해 command를 실행시킨다. 그리고 parent에서 waitpid를 통해 child가 종료되면 reap해준다.

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

fork를 하면 child가 생기고 child에서 명령어를 수행한다. 이때 parent process는 waitpid함수를 이용하여 child의 종료 시그널을 기다린다. 그리고 child process에서 명령어가 종료되면 parent process에서 waitpid를 통해 시그널을 감지하고 child process를 reaping 해준다.

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

먼저 pipeline(‘|’)이 있는 명령어와 없는 명령어로 따로 구분을 해주었다. pipeline이 없으면 phase1과 비슷하게 구현해주었다.

pipeline이 있는 경우는 먼저 fork를 실행하고 run\_pipe\_command를 child process안에서 실행한다. run\_pipe\_command 함수는 recursive로 구현하였다.

STEP 1. 먼저 마지막 명령어가 아니면 fork를 해주고 child process에서는 dup2 명령어를 이용하여 STDIN\_FILENO가 in\_fd(이전 명령어에서 넘겨준 fd[0])가 가리키는 곳을 가리키게 한다. 그리고 STDOU\_FILENO는 현재의 fd[1]를 가리키게하여 이전 명령어의 출력 결과를 현재 명령어의 입력으로 받고 현재 명령어의 출력을 STDOUT\_FILENO를 통해 내보낸다. 이러한 설정들을 해주고 execvp 명령어를 사용하여 명령어를 수행한다. parent process에서는 run\_pipe\_command를 recursive로 다시 실행하고 waitpid 함수를 실행시킨다.

STEP 2. recursive로 run\_pipe\_command를 실행하다가 마지막 명령어에 도달하면 STEP 1과 동일하게 dup2 명령어를 이용하여 STDIN\_FILENO가 in\_fd(이전 명령어에서 넘겨준 fd[0])가 가리키는 곳을 가리키게 한다. 그리고 명령어를 수행하고 종료한다.

STEP 1을 마지막 명령어 이전 명령어까지 실행한 후 STEP 2를 통해 마지막 명령어까지 실행시켜주고 종료한다. 그리고 최초의 fork()를 실행시키는 parent에서 waitpid를 통해 child를 reaping한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Pipeline이 없다면 run\_command를 실행하고 pipeline이 존재하면 run\_pipe\_command를 수행한다.

pipeline 명령어를 수행하는 run\_pipe\_command를 recursive로 구현하여 위에서 설명하였듯이 마지막 명령어가 되기 전까지 위에서 말한 STEP 1을 수행하고 마지막 명령어에 도달하면 STEP 2를 실행하여 pipeline의 개수에 관계없이 작동하도록 구현하였다.

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

‘&’문자가 있는지 파싱 과정에서 확인하여 parseline에서 ‘&’가 있으면 1을 없으면 0을 반환한다. 이 값에 따라 foreground와 background를 나누는데 foreground process에서는 이전에 했던 것과 같이 waitpid를 통해 명령어가 종료되길 기다리다 reaping해준다. 하지만 background process에서는 child\_handler를 통해 child의 signal이나 종료 여부를 받아 waitpid를 통해 reaping해준다. 또한 job을 control하기 위해 job, fg, bg, kill 명령어를 추가해 주었다.

* 1. **개발 방법**

**Phase1**

builtin\_command 함수에 cd와 exit에 대한 기능을 구현하였다. cd는 chdir()을 사용하여 구현하였고 exit은 원래 구현된 quit와 동일하게 구현해주었다.

eval 함수에서 builtin\_command가 아니면 fork를 실행한다. 그리고 fork로 전달받은 pid가 0이면 child process이다. child에선 execve함수를 execvp함수로 바꾸어서 명령을 수행시켰다. 그리고 parent process에선 bg가 1이 아닐 때(foreground 명령) waitpid를 수행하여 child를 reaping 해준다.

**Phase2**

run\_command와 run\_pipe\_command 함수를 구분해주어 명령어에 pipe가 없을 때와 있을 때를 구분하여 수행시켜주었다.

eval 함수에서 pipe가 있는지 없는지 확인하고 있다면 파이프를 기준으로 명령어들을 나누어서 저장해준다. pipe가 있다면 fork를 하고 child process에서 run\_pipe\_command 함수를 호출해준다. parent process에서는 waitpid로 child를 reaping해준다. pipe가 없다면 run\_command 함수를 호출해준다.

run\_pipe\_command 함수에서는 명령어 인자(char \*argv[][MAXARGS])와 전체 명령어(char \*cmdline), 몇번째 명령어가 수행될 차례인지 알려주는 변수(pos), 입력으로 받을 파일 포인터(in\_fd), 백그라운드 여부(bg)를 parameter로 받아준다.

함수내부적으로는 먼저 마지막 명령어인지 확인을 해준다. 마지막 명령어라면 Dup2(in\_fd,STDIN\_FILENO)으로 standard input이 in\_fd가 가리키는 곳을 가리키게 한다. 그리고 execvp를 통해 명령어를 수행한다. 만약 마지막 명령어가 아니라면 pipe함수를 호출하고 이어서 fork를 하여 child process이면 Dup2(in\_fd,STDIN\_FILENO)으로 standard input이 in\_fd가 가리키는 곳을 가리키게 하고 Dup2(fd[1],STDOUT\_FILENO)으로 standard out이 fd[1]이 가리키는 곳을 가리키게 하므로 pipe를 이용하여 process간의 입출력을 연결해준다. 그리고 execvp를 통해 명령어를 수행한다. 그리고 parent process에서 run\_pipe\_command를 recursive하게 호출해주어 파이프의 개수에 상관없이 해당 기능이 잘 수행되게 해준다.

run\_command 함수는 phase 1에서 구현한 명령어 실행 기능을 함수로 묶어주어서 pipe가 아닐 때 수행시켜주는 함수이다.

**Phase3**

먼저 job을 관리하기 위해 구조체를 선언하였다. 요소로는 job의 pid와 id, state, cmdline이 있다. 또한 job의 state를 나타내기 위해 0~5까지 각각의 상태를 define해주었다.

job control을 통해 job들을 관리하기 위해 jobs, fg, bg, kill을 추가로 구현하기 위해 먼저 jobs라는 job들을 관리 하는 배열을 만들어주었다. 그리고 처음 jobs 배열을 초기화 하는 함수(initjobs), job을 jobs배열에 추가해주는 함수(addjob), 명령어가 종료되었을 때 job의 상태를 done으로 변경시켜주는 함수(donejob), job의 요소들을 clear시켜주는 함수(clearjob), jobs를 출력해주는 함수(printjobs), 종료(done)된 job을 jobs에서 제거하고 종료된 job을 출력시켜주는 함수(delNprint\_done) 등의 함수들을 만들어주어 구현해주었다.

또한 SIGINT, SIGTSTP, SIGCHLD에 대한 handler를 각각 구현해주어 앞의 시그널시 받았을 때 각각 상황에 맞는 행동들을 수행 하게 해주었다.

main함수에서는 입력을 받기전에 signal을 install해주었고 입력을 받고 eval을 통해 명령어를 수행하고 delNprint\_done을 통해 끝났거나 종료된 job들을 jobs 배열에서 제거해주고 출력해준다.

eval함수에서는 fork를 하여 child에서 명령어를 수행하고 parent에서 해당 job을 jobs에 추가한다. 그리고 background 인지 foreground인지 판단하여 foreground면 waitpid를 통해 reaping해주고 명령어가 잘 끝났으면 jobs에서 해당 job을 제거해준다. 만약 backgound이면 waitpid를 해주지 않는다.

CTRL+C와 CTRL+Z에 대한 처리도 handler에서 해주었는데 CTRL+C일때는 SIGINT 시그널이 발생하여 해당 프로세서를 종료시킨다. 여기서 두개의 입력은 foreground에만 영향을 미치며 background에는 영향을 미치지 않는다. 따라서 CTRL+C를 입력받았을 때 sigint\_handler에서 job의 state가 FG(foreground)인 job을 찾아서 해당 process와 해당 process의 자식들에게 SIGKILL 시그널을 보내어 종료시킨다. CTRL+Z를 입력받았을땐 sigtstp\_handler에서 job의 state가 FG인 job을 찾아 state를 ST(stop)로 바꾸어주고 해당 process와 해당 process의 자식들에게 SIGTSTP 시그널을 보내어 중지시킨다.

sigchld\_handler에서는 child들이 보내는 signal을 감지한다. waitpid를 사용하여 background에서 종료되는 job들을 reaping해준다. 여기서 정상적으로 종료했거나 시그널을 받고 종료했다면 donejob함수를 호출하여 job의 state를 DONE으로 바꿔준다.

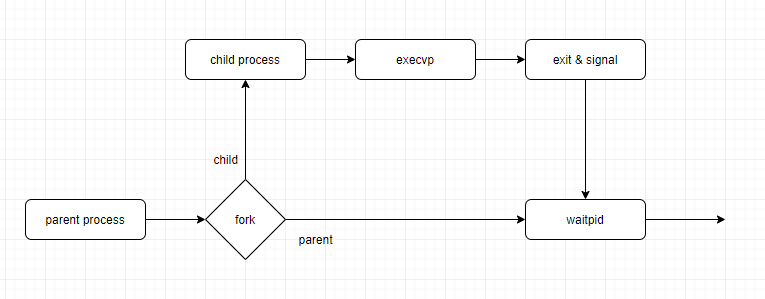
이렇게 jobs에 저장해준 job들을 모두 출력하려면 ‘jobs’명령어를 입력한다. 그러면 printjobs함수를 호출하여 jobs에 존재하는 job들을 출력해준다.

job을 foreground로 수행시키려면 ‘fg <job>’명령어를 입력한다. 그러면 해당 job의 state를 FG로 바꾸고 SIGCONT 시그널을 보내주어 실행시켜준다. 그다음 waitfg라는 함수를 호출하여 해당 job의 상태가 변할때까지 기다려준다. waitfg는 해당 job의 상태가 FG가 아니게 되면 종료하는 함수이다.

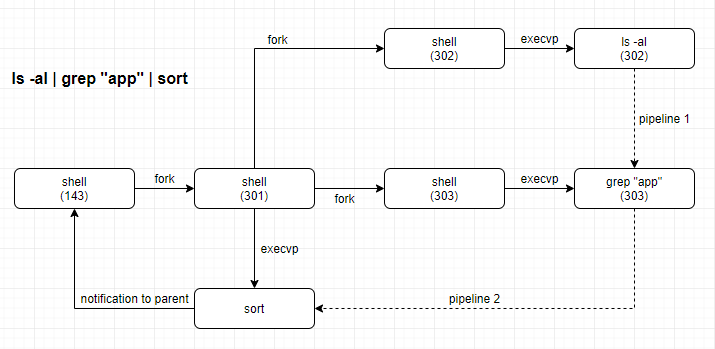
‘bg <job>’ 명령어는 fg와 반대로 background에서 job을 수행하게 하는 명령어이다. fg와 비슷하지만 해당 state를 BG로 바꿔준다음 fg와는 다르게 background에서 수행하므로 waitfg함수를 호출하지 않는다.

‘kill <job>’명령어는 해당 job을 kill해주는 명령어이다. 해당 job을 찾아 SIGKILL 시그널을 보내주어 종료시켜준다. 그리고 state는 KILL로 바꾸어준다.

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



1. **Phase 2 (pipeline)**



1. **Phase 3 (background)**

