**System Programming Project 4**

담당 교수 : 김영재 교수님

이름 : 백인찬

학번 : 20150195

1. **개발 목표**

* Linux에서 제공하는 shell 프로그램(bash, csh 등)의 기능을 가진 shell 프로그램을 개발한다. 기본적으로 단일 명령어, pipe로 이어진 다중 명령어 처리와 background 명령어에 대한 처리의 기능을 갖는다.

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

* Phase 1에서 명령어를 입력 받아 parse하고 입력 받은 명령어에 해당하는 파일을 실행해 명령어를 수행한다. fork()와 exec()을 통해 자식 프로세스를 생성한 후 실행하도록 한다.

1. Phase 2

* pipe(|) 로 연결된 다중 명령어를 처리한다. 명령어의 개수만큼 fork()를 통해 병렬적으로 자식 프로세스를 생성하고 pipe()를 통해 입출력을 연결하여 실행하도록 한다.

1. Phase 3

* 명령어의 마지막에 & 이 추가되면 background로 수행하도록 한다. 기본적으로 background에서 수행할 경우 parent process는 child process를 기다리지 않고 main loop으로 return한다. 시그널 핸들링을 통해
  1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork를 통해 child process를 생성하여 parsing한 실행파일의 이름과 인자를 넘겨 execve를 통해 실행하도록 한다. 실행파일을 위해 /bin 하위의 파일과 /usr/bin 하위의 파일을 참조한다. 우선적으로 /bin의 파일을 execve를 통해 실행하고 해당하는 파일이 없을 경우 /usr/bin의 파일을 찾아 실행한다.

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

입력 받은 문자열에서 ‘|’를 찾고 그 개수+1만큼 child process를 fork()하여 생성한다. parsing한 문자열에서 ‘|’를 기준으로 배열에 명령어를 별도로 저장한다. (ex. ls -al | grep abc -> [0] ls -al [1] grep abc) 또 그 개수만큼 pipe를 생성한다. child process끼리 pipe를 통해 입출력을 조절한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

반복문을 통해 child process를 병렬적으로 생성한다. 생성된 child process의 pid를 배열에 저장한다 (ls -al | grep abc | sort -> [0] ls -al의 pid, [1] grep abc의 pid, [2] sort의 pid).

index가 0이라면 pipeline의 첫번째 명령어라고 간주하고 (ls -al) dup2()을 통해 stdout을 pipe의 write end로 옮긴다.

index가 위에서 구한 ‘|’의 개수와 일치한다면, 마지막 명령어라고 간주하고 (sort) stdin을 pipe의 read end로 옮긴다.

이 외의 경우는 pipeline의 중간에 있는 명령어라고 간주하고 (grep abc) stdin을 pipe의 read end로, stdout을 pipe의 write end로 옮긴다.

위의 flow를 통해 pipeline의 개수가 몇 개가 들어오는지에 관계 없이 정상적으로 handling하여 수행할 수 있도록 한다.

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

Foreground(FG)/background(BG) process와 job control를 위해 fork된 process들을 묶은 process group의 구조체 PG를 정의한 후, PG의 멤버 변수인 pid의 배열에 pid를 저장한다.(pipeline으로 연결된 프로세스들의 경우 모두 pid를 배열에 저장해 한 PG로 묶는다.)

또 PG의 doubly linked list인 FGPGs, BGPGs를 선언해 PG를 관리한다.

Background로 실행했다면 BGPGs에 저장하고 parent process는 기다리지 않고 return하여 다시 main함수의 loop을 수행한다. 또 아직 background process가 reap되지 않았을 경우 SIGCHLD를 unblock하여 parent process가 다른 작업 수행 중에 SIGCHLD handler를 통해 child process를 reap할 수 있도록 했다.

foreground process가 SIGTSTP, SIGINT signal을 받은 경우의 handler를 구현해 FG와 BG 사이의 job control이 가능하도록 했다.

또 background process의 명령어가 less, man, more인 경우 shell의 표준 출력을 방해하면 안되므로 /dev/null로 출력하도록 했다.

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

parent process가 child process에 대한 관리를 해야하므로, parent process가 child process의 PG를 list에 추가, 삭제 등의 처리를 하는 도중 SIGCHLD, SIGINT, SIGTSTP 등의 시그널을 통해 데이터가 변경되는 불일치를 막기 위해 PG 관리 중에는 모든 시그널을 block한다. 처리가 완료되면 다시 unblock한다. SIGINT 등으로 connection이 종료되거나 하는 경우 별도의 handler가 존재하지 않으면 parent process인 shell도 종료될 수 있기 때문에 별도의 signal handler를 정의한다.

SIGINT, SIGTSTP handler에서는 foreground child process에 SIGINT, SIGTSTP 시그널을 보내고, FG list에서 제거한다. SIGTSTP의 경우 job control을 위해 BG list에 추가한다. SIGCHLD handler에서는 wait() 시스템 콜을 통해 child process를 reap하고 해당 process가 속한 list(FG/BG)에서 해당 PG의 process count를 하나씩 줄인다. count가 0이 되면 해당 PG에 속한 process가 모두 reap되었다는 뜻이므로 속한 list에서 제거한다.

* 1. **개발 방법**
* **Phase 1**

builtin\_command() : exit의 경우 exit()을 호출해 프로그램이 종료하도록, cd의 경우 cd()를 호출해 working directory 변경이 가능하도록 했다.

cd() : chdir()을 호출해 working directory를 변경한다.

eval() : argv[0]에 해당하는 실행 파일의 이름을 /bin과 /usr/bin에서 참조하도록 했다. parent process는 Waitpid()를 통해 child process의 종료를 기다리게 했다.

* **Phase 2**

parseline() : pipeline(‘|’)도 함께 parsing하도록 한다. pipeline의 개수를 저장하여 반환한다.

exec\_pipe() : pipeline이 존재할 경우 수행하는 함수로, pipeline의 개수만큼 pipe를 동적할당하여 생성한다. parseline()에서 parsing된 명령어들을 pipeline를 기준으로 별도의 배열에 저장한다. (ex. ls -al | grep abc -> [0] ls -al [1] grep abc)

pipe\_fork\_execve() : pipeline+1의 개수만큼 fork하여 child process를 생성하고 그 pid를 배열에 저장한다. 배열에 index에 따라 첫번째와 중간, 마지막 process를 구분하고 그에 맞게 dup2()를 통해 pipe를 연결한다. child process는 exec\_pipe()에서 index에 맞게 저장된 명령어 배열에 따라 해당 명령을 실행한다. parent process는 waitpid()를 통해 child process를 모두 reap할 수 있도록 한다.

search\_and\_execve() : /bin의 파일을 먼저 탐색 및 참조하여 실행하고 실패할 경우 /usr/bin의 파일을 탐색 및 참조하여 실행한다. (ex. sort -> /bin/sort -> /usr/bin/sort)

* **Phase 3**

PG : process group을 위한 구조체로 전체 명령어와 pid의 배열, status, process의 개수 등을 멤버변수로 갖는다.

PG\_list : PG를 관리하기 위한 doubly linked list 구조체이다.

eval(), pipe\_exec() : 단일 명령어에 대해 PG를 생성하여 FG 또는 BG의 list에 삽입한다. FG의 경우 explicit하게 전역 변수를 통한 반복문을 이용해 child process가 reap되기를 기다린다.

SIGCHLD\_handler() : parent process가 SIGCHLD 시그널을 받았을 경우 완료된 child process를 reap하고 해당 pid의 PG를 FG와 BG 중 하나에서 찾는다. 찾은 PG가 FG인 경우엔 전역 변수의 값 변경을 통해 parent process가 반복문을 탈출하도록 한다. PG로 묶인 process가 모두 reap된 경우 FG 또는 BG의 list에서 제거한다.

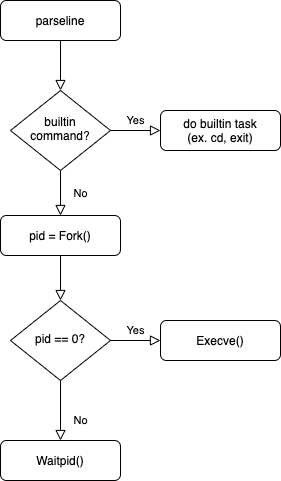
SIGINT\_handler() : parent process가 SIGINT 시그널을 받았을 경우 FG의 list에 있는 모든 PG에 kill()을 통해 SIGINT 시그널을 보낸다. 또 FG의 list에서 제거하며 반복문에서 기다리고 있는 parent process를 탈출하게 한다.

SIGTSTP\_handler() : parent process가 SIGTSTP 시그널을 받았을 경우 FG의 list에 있는 모든 PG에 kill()을 통해 SIGTSTP 시그널을 보낸다. FG의 list에서 제거해 BG의 list로 옮겨주고 status를 STOPPED로 변경한다. 반복문에서 기다리고 있는 parent process를 탈출하게 한다.

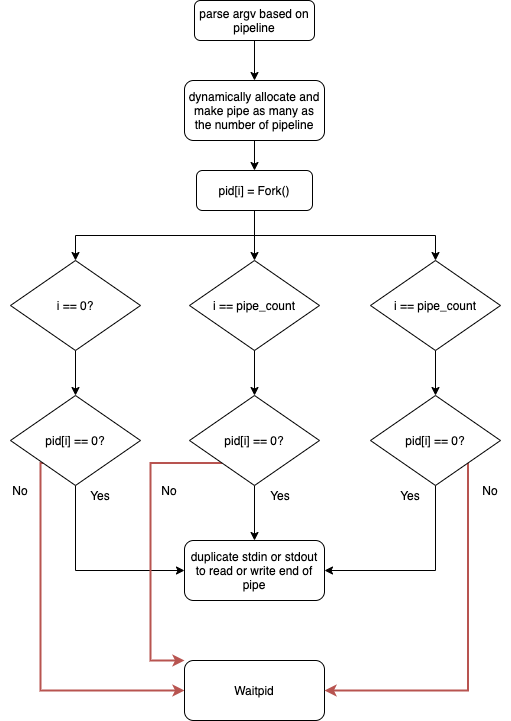
check\_termcap() : 실행할 파일이 more, man, less인지 검사하고 background에서 수행되어야 한다면 1을 리턴해 stdout이 /dev/null로 연결될 수 있도록 한다.

\*\* 어느 함수에서든 PG에 대한 관리를 할 때에는 항상 Sigprocmask를 통해 모든 Signal을 Block한 후 처리가 모두 끝나면 Unblock한다.

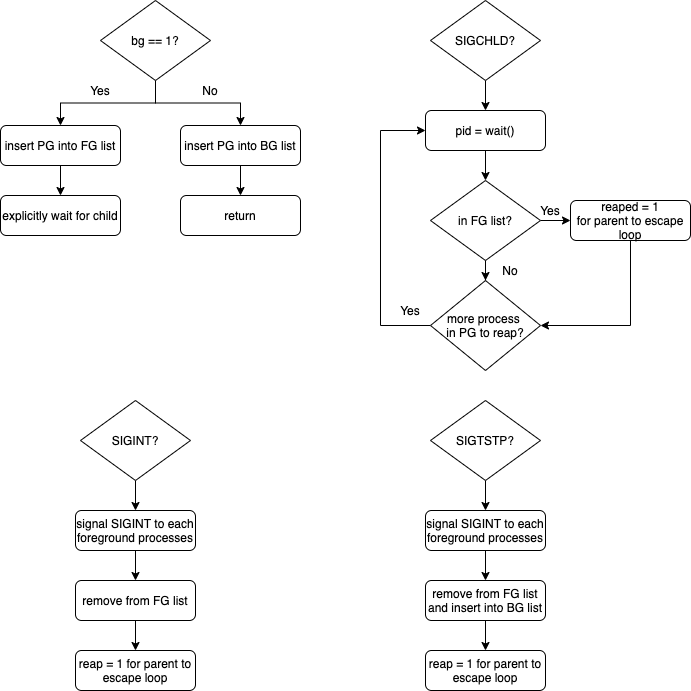
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****