**System Programming Project 4**

**담당 교수 : 김영재 교수님**

**이름 : 김민식**

**학번 : 20171609**

1. **개발 목표**

Linux shell은 kernel과 사용자간의 다리역할을 하여 사용자로부터 명령을 받아 해석하고, 하드웨어를 위한 지시어로 바꾸어 프로그램을 실행하는 역할을 한다.

이번 project4 에서는 이러한 Linux shell을 구현하는 것이 개발목표라고 할 수 있다. 총 3단계로 나누어 linux shell을 구현하고자 한다.

Phase 1 : linux shell의 기본 internal shell command를 구현한다.

Phase 2 : pipe 기능을 추가하여 여러 명령어끼리 연결해주어 앞에서 실행한 명령의 결과를 파이프 뒤의 명령어의 입력으로 사용되도록 구현한다.

Phase 3 : background에서 프로세스를 실행할 수 있도록 구현하며, 명령어를 통해 백그라운드와 포그라운드 간의 이동이 가능하고, 프로세스를 중지시킬 수 있도록 linux shell 을 구현한다.

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

Linux shell의 basic internal shell command를 구현한다.

구현 명령어 : cd, cd.., ls, mkdir, rmdir, touch, cat, echo, exit

텍스트이(가) 표시된 사진

자동 생성된 설명

위 명령어는 프로젝트 명세서에 있는 명령어를 입력한 결과이다. 이를 통해 ls, cd, cd .., mkdir 명령어가 성공적으로 구현됨을 알 수 있다. 또한 exit 명령어로 myshell을 종료한 후 다시 실행시켰을 때, 생성하였던 폴더가 유지된다는 것을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 결과를 통해 mkdir, touch 등을 사용하여 만들어진 폴더와 파일을 삭제하는 명령어인 rmdir, rm 명령어가 성공적으로 실행됨을 알 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 결과를 통해 cat 명령어와 echo 명령어를 통해 파일과 문자 등을 출력할 수 있음을 알 수 있었다. 이 역시 myshell 종료 후 다시 실행시켰을 때 파일이 유지된다는 것을 알 수 있다.

1. **Phase 2**

Phase2에서는 phase1에서 만든 simple shell에 pipe 기능을 추가하였다. “|” 를 이용하여 명령어끼리 연결해주어 앞에서 실행한 명령의 값을 파이프 뒤의 명령어의 입력으로 사용되도록 구현하였다.

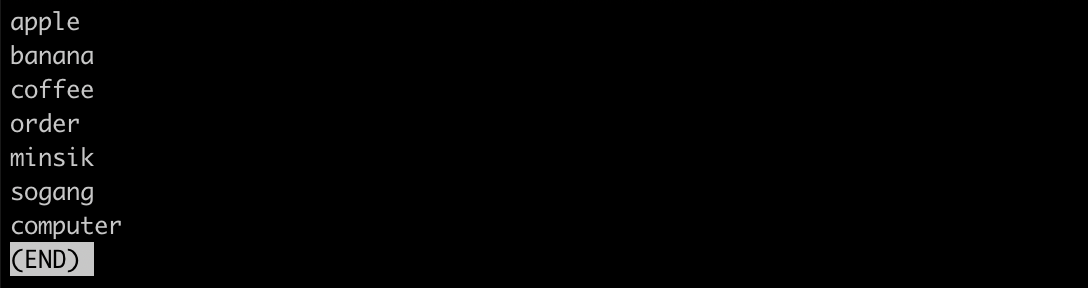
텍스트이(가) 표시된 사진

자동 생성된 설명

“ls -al | grep m” 명령어를 통해 ls -al 결과값을 grep m 명령어의 입력으로 사용하여 “m”이 포함된 파일을 찾아 출력할 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명



Cat text 명령어를 통해 파일을 shell에 출력하였다면, cat text | less 명령어를 통해 내용을 스크롤할 수 있도록 출력되는 것을 볼 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 결과를 통해 3개의 pipe 또한 실행되다는 것을 알 수 있었다.

Cat text | grep -v a 명령어를 통해 file text의 내용 중 “a”를 가지고 있지 않은 내용을 출력하였다. 그 결과값을 sort -r 명령어의 입력값으로 넣어 내림차순으로 출력된다는 것을 볼 수 있었다. 따라서 여러개의 파이프라인도 성공적으로 구현함을 알 수 있었다.

1. **Phase 3**

Phase3 에서는 background에서 프로세스를 실행할 수 있도록 구현하였다.

“&” 명령어를 통해 명령어를 백그라운드에서 실행할 수 있도록 구현하였으며 bg, fg 명령어를 통해 Background와 foreground 간에 이동할 수 있도록 하였고, kill 명령어를 통해 프로세스를 중지 시킬 수 있도록 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위의 결과를 통해 명령어 뒤에 “&”를 함께 입력하였을 때 백그라운드에서 실행되는 것을 알 수 있으며, 파이프라인 명령어 또한 올바르게 구현됨을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 결과를 통해 jobs 명령어를 입력하면 현재 running 중인 명령어를 출력함을 볼 수 있다. 또한 Kill (%process id)를 이용하여 running 중인 프로세스를 중지시킴을 알 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위의 결과를 통해 백그라운드로 실행시킨 명령어를 fg 명령어를 통해 foreground로 전환시킬 수 있다는 것을 알 수 있다.

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

“CSE4100-SP-P4>” 프롬프트 이후 입력받은 명령어는 fork() 함수를 통해 child process에서 수행이 된다.

Fork() 함수가 실행되면 parent process 의 모든 데이터와 구조를 상속받는 child process 가 생성되며, child process는 pid가 0이 되면 종료한다. Parent process는 child process의 동작이 완료될 때까지 wait 했다가 수행이 종료되면 다시 프롬프트를 띄우는 역할을 하게 된다.

Fork() 함수를 통해 만들어진 child process는 exec 계열의 함수를 통해 실행된다. Ls, grep 등 경로가 고정되어 있지 않은 명령어의 경우 execv() 를 사용하며, 해당 실행파일이 위치하고 있는 경로를 포함 시켜주어야 한다.

/bin 에 있는 명령어를 사용할 경우, 이미 설정되어있는 디렉토리이기 때문에 execvp() 를 사용하여 실행시킨다.

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

myshell 이 수행중일 때 터미널을 종료시키는 명령어를 받았을 때 터미널이 종료되는게 아니라 Myshell만 종료되도록 변경해주어야 한다. 이를 위해 signal 함수를 이용하여 handling 해주어야 한다.

일반적인 process의 경우 SIGINT (Ctrl+c) 시그널을 통해 수행중인 process를 종료할 수 있지만 myshell의 경우 SIGINT 시그널을 무시하도록 설정하였다.

fork() 를 이용하여 command 를 수행하는 child process의 경우 수행 도중에 중단시킬 수 있어야 하기 때문에 child process의 경우에만 SIGINT 를 설정하였다. 하지만 background로 실행 중이라면 SIGINT 시그널을 무시하도록 설정하였다. 이를 구현하기 위해 pid에 따라 다르게 작동하는 signalhandler 함수인 sigint\_handler 함수를 사용하였다.

또한 SIGCHLD 로 child process wait 할 때 프로세스가 온전하게 수행되도록 구현하였다. 즉 child process가 좀비 프로세스가 되는 것을 막아주었다. 이를 구현하기 위해 SIGCHLD 시그널을 parent process가 받을 때 sigchildhandler 를 통해 wait 를 호출하여 구현하였다.

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

Pipeline 명령어에서 ‘|’ 를 기준으로 좌측 명령의 표준 출력을 우측 명령의 표준 입력으로 설정하였다. Pipeline에 연결된 각각의 command는 개별적인 Proces로 수행되며 좌측 명령이 child process, 우측 명령이 parent process로 수행된다.

“Pipe(fdarr[2]);” 을 사용하여 pipe 연결을 하였으며, parent process는 fdarr[0]을 표준 입력으로, child process는 fdarr[1]을 표준 출력으로 하여 구현하였다. 위의 내용을 구현하기 위해 파일 디스크립터를 복제하는 dup2(int fd1, int fd2)를 기본 원리로 사용하였다. dup2(fdarr[0], 0)으로 fdarr[0]을 표준 입력으로, dup2(fdarr[1], 1)으로 fdarr[1]을 표준 출력으로 설정하였다.

Child process는 dup2( ) 를 호출하여 자신의 stdin을 자신의 왼쪽 끝에 연결시키고, fdarr[0]을 입력받는다. Parent process는 모든 데이터가 본인의 왼쪽 끝에 쓰여질 때까지 wait 한다. Parent process로부터 모든 데이터가 파이프 오른쪽 끝으로 전달되면 그 결과는 stdout을 통해 출력한다.

Parent process는 fork( ) 호출되면 fdarr[0]을 닫는다. 그 후 child process에게 전달한 데이터를 fdarr[1]에 write 한다. 모두 전송하며 fdarr[1]을 닫는다. Child process가 command를 수행할 때 까지 parent process는 wait 함수를 이용하여 기다리며, child process가 종료되면 함께 종료된다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Command1 | command 2 | command 3 | .. 와 같이 여러 개의 파이프라인이 있을 때 “commmand1 | command2”를 실행해서 표준 출력이 생기면 이를 command3의 표준입력으로 재지향하는 방법으로 구현하였다.

각 입력과 출력 값을 사용하여 위 질문과 같이 Pipeline에 연결된 각 command는 개별적인 process로 수행되며, 파이프를 기준으로 좌측 명령이 child process, 우측 명령이 parent process로 수행된다.

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

명령어를 입력할 때 마지막에 ‘&’를 입력하여 process를 백그라운드로 실행시키도록 구현하였다. 일반적인 foreground 명령의 경우 parent process가 child process의 작업이 완료될 때 까지 wait 한다.

하지만 background process의 경우 parent process가 child process를 기다리지 않고 서로 개별적으로 수행하도록 구현하였다.

* 1. **개발 방법**

**< 구조체 >**

**Struct Myshells**

Shell의 기본 정보를 저장할 때 사용하는 구조체이다.

하나의 명령어를 받아 실행될 때, 함께 입력되는 인자들, 명령어, 명령어의 종류(background, foreground, pipe), process id 등을 저장하는데 사용된다.

**Struct JobStruct**

Jobs, bg, fg 명령어가 입력되었을 때 사용되는 구조체이다.

명령어와 함께 그 명령어의 상태(running, done, terminated 등) 을 저장할 수 있다.

**Const char \*SetArray[]**

JobStruct 구조체에 사용되며, jobs, bg, fg 명령어에 사용되는 문자열 배열이다.

Running, Done, Stopped, Continued, Terminated가 저장되어 있으며, 각 명령어의 상태에 따라 다르게 표현되기 위해 만들어졌다.

**< Phase 1 >**

**char\* Func\_InputLine()**

getchar( ) 를 통해 명령어를 입력 받는 함수이다.

EOF 혹은 엔터까지 문자열을 입력받고 끝 글자를 null값으로 변경시켜준다. 그 후 그 값을 리턴시켜준다.

만약 명령문이 없거나, 메모리를 초과하여 입력이 될 경우 에러를 출력하고, 명령문을 다시 입력 받는다.

**struct JobStruct\* Parse\_Command(char \*line)**

Func\_inputline()을 통해 입력받은 명령문을 표준 명령문으로 변환시켜주는 함수이다. 명령문 사이에 띄어쓰기가 여러번 있거나, 명령문 앞에 여러 띄어쓰기가 있는 명령문을 표준 명령문으로 변경시켜준다.

**int Func\_Command\_Set(char \*cmd)**

입력된 명령어에 대하여 각각에 알맞는 정수를 반환해주는 함수이다.

Shell built-in 명령어가 아닌 exit, cd 를 입력 받았을 때, exit는 1을, cd 는 2를 입력받아 이에 알맞는 함수가 실행되도록 하였다.

**int Func\_cd(int argc, char\*\* argv)**

명령어로 “cd”가 입력되었을 때 실행되는 함수이다.

Cd .. 혹은 cd foldername 이 입력되었을 경우 알맞는 디렉토리로 이동하도록 설정되어있다. 문법에 맞지 않는 명령어가 들어올 경우 에러를 출력한다.

**< Phase 2 >**

**struct JobStruct\* Parse\_Command(char \*line) of Phase1**

문자열에 ‘ | ‘이 있는지 탐색하는 코드를 추가한다.

만약 파이프가 존재하면 파이프를 기준으로 두 명령어를 나누어 구조체에 저장하고, 새로운 구조체를 다시 리턴한다.

**void Func\_Check\_Zombie()**

명령어가 끝났는지 끝나지 않았는지 판별하는 함수이다.

Pipe 를 사용할 때 parent process는 child process가 종료될때 함께 종료된다. 이때 parent process가 wait 할 때 사용된다.

**struct Myshells\* Parse\_command\_segment(char \*segment)**

>, < 이 포함된 명령문이 입력될 경우 redirection 또한 시켜준다.

**< Phase 3 >**

**struct JobStruct\* Parse\_Command(char \*line) of phase 1**

입력된 명령문에 ‘&’가 있을 경우 명령어를 set을 background로 설정하여 구조체에 저장시키도록 코드를 추가하였다.

**int Func\_Command\_Set(char \*cmd) of phase 1**

jobs, fg, bg, kill 이 입력되었을 때 리턴되는 정수값을 3,4,5,6으로 추가하는 소스코드를 작성하여 각 명령문에 대한 함수가 호출될 수 있도록 수정하였다.

**int Func\_Wait\_Job(int id)**

백그라운드 명령어가 종료되었는지 확인하는 함수이다.

Sleep 10 & 등 의 종료되는데 시간이 걸리는 명령어의 경우 jobs 명령어를 입력하였을 때 정보가 입력되어야 하기 때문에 구조체에 입력시켜주는 역할을 한다.

**int Func\_jobs(int argc, char \*\*argv)**

jobs 명령어 입력시 실행되는 함수이다.

명령어와 그 정보가 저장되어 있는 구조체를 탐색하여 현재 실행중인 명령어를 화면에 출력한다.

**int Func\_Print\_JobSet(int id)**

Func\_jobs 내부에서 호출되는 함수이다.

매개변수로 받은 id 값을 가지는 구조체의 정보 (pid, process 상태) 등을 화면에 출력시켜준다.

**int Func\_fg(int argc, char \*\*argv)**

fg %num 형식으로 입력되는 명령문에 대한 함수이다.

Background로 Running 중인 process가 존재할 때, 그 pid를 입력하면 foreground 명령으로 변환시켜주는 함수이다.

**int Func\_bg(int argc, char \*\*argv)**

bg &num 형식으로 명령문이 입력될 경우 실행되는 함수이다.

Foreground에서 실행중인 명령문의 Pid를 입력하면 이 명령문을 background로 변환시켜준다.

**int Func\_kill(int argc, char \*\*argv)**

kill %num 형식으로 입력될 경우 실행되는 함수이다.

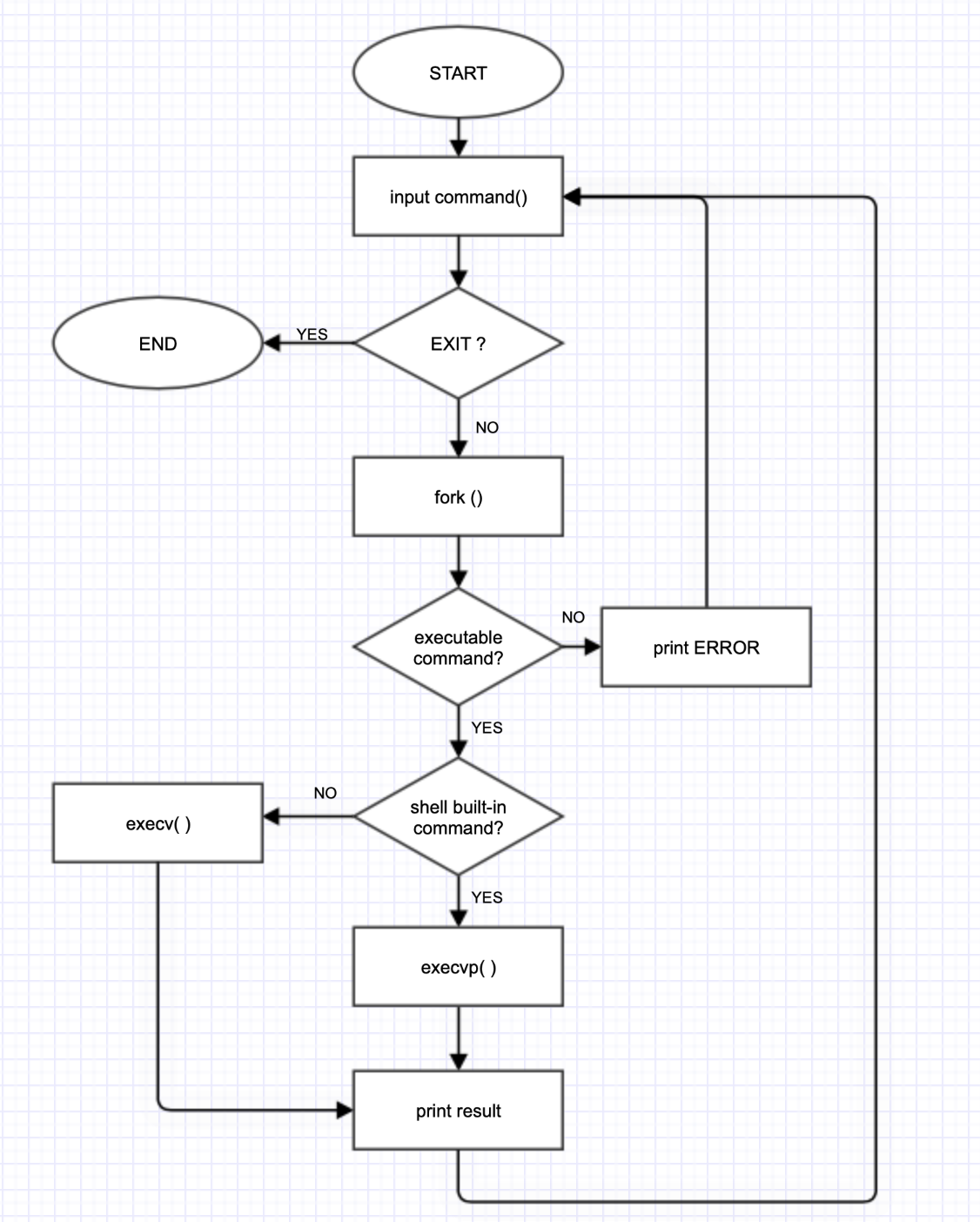
Running 중인 process의 번호를 입력하게되면 process가 종료된다.

**int Func\_Free\_Job(int id)**

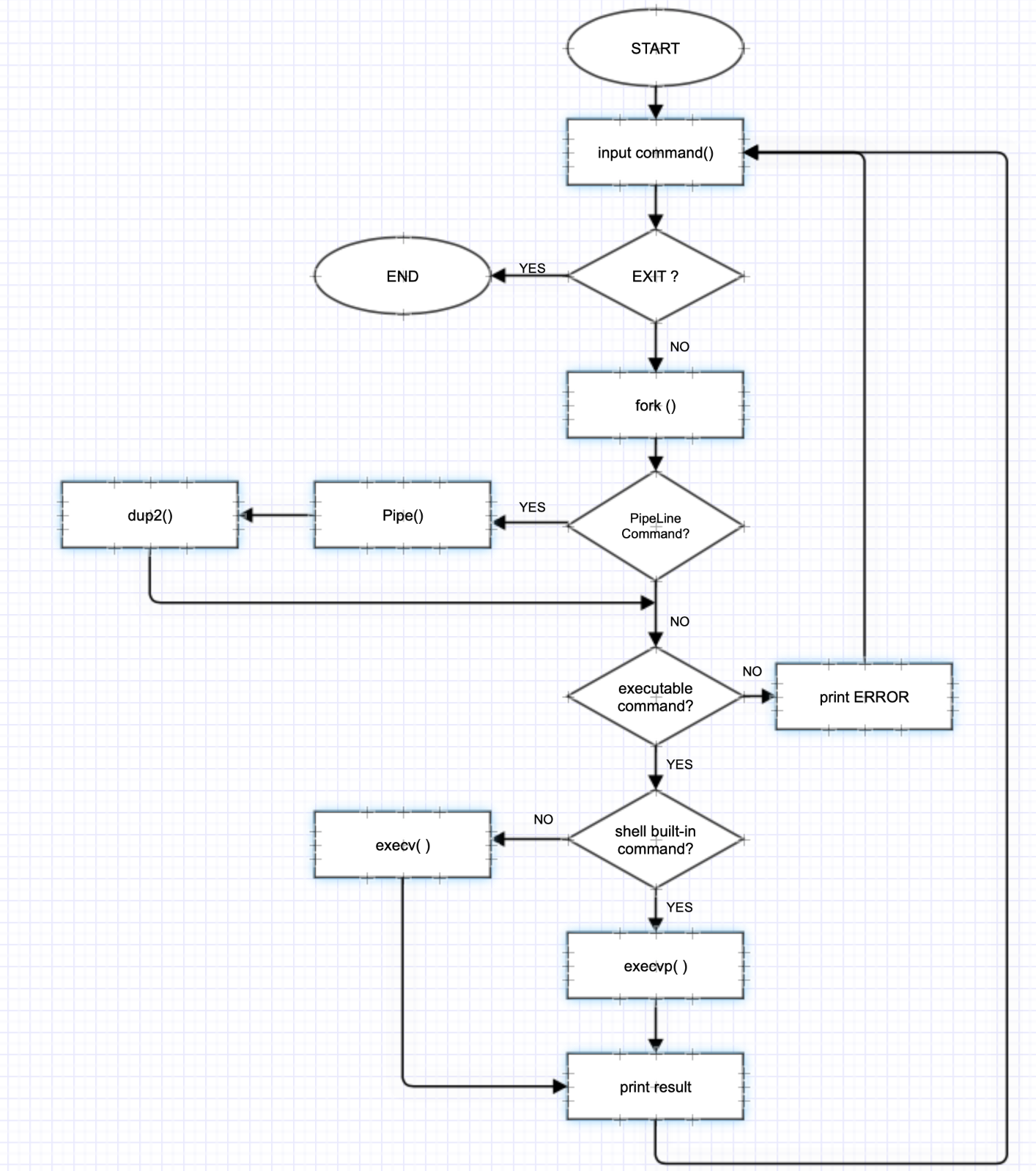
job이 저장되어있는 구조체를 메모리 해제 시켜주는 함수이다.

구조체를 탐색하여 모두 free 시켜준다.

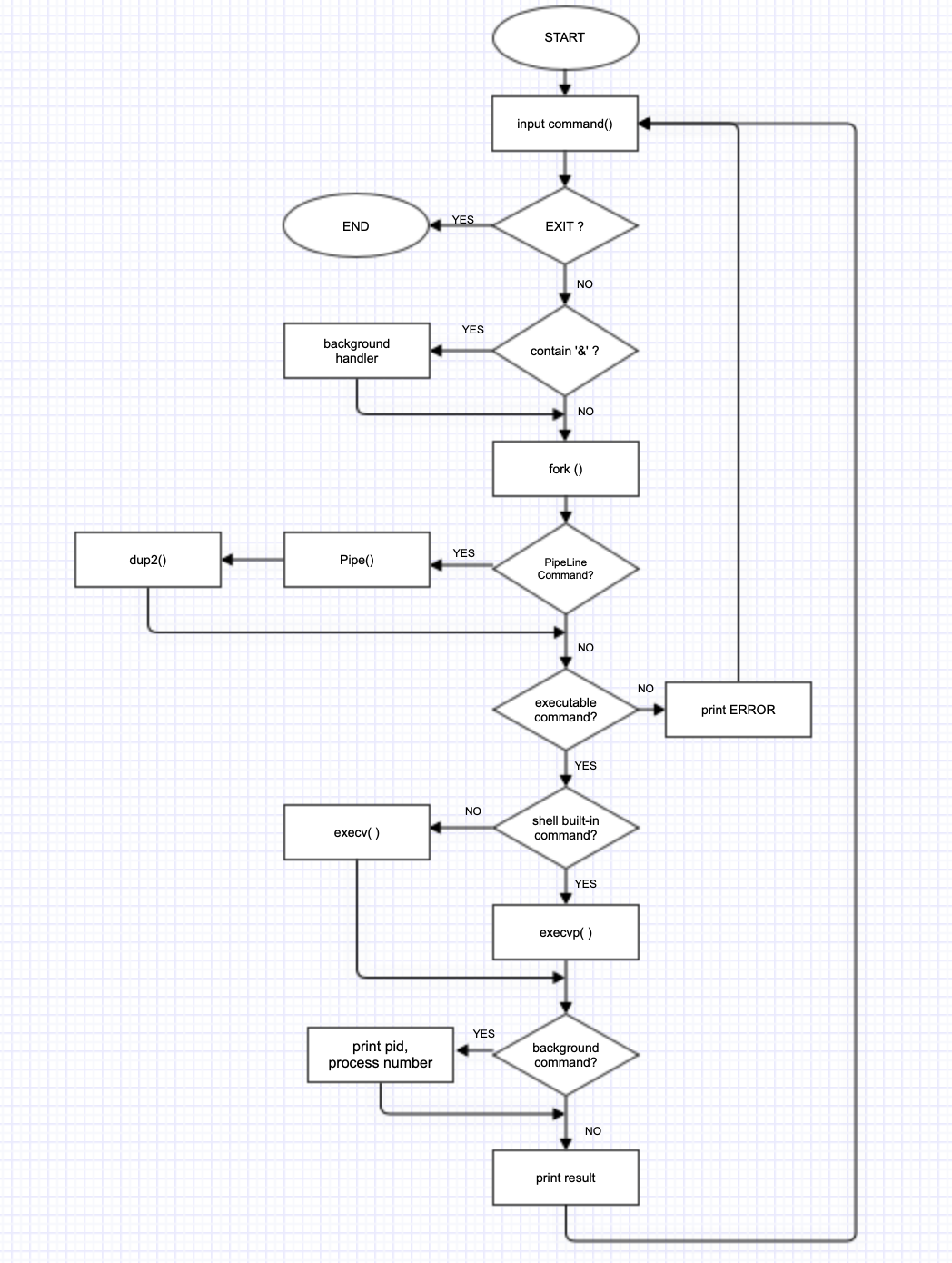
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****