**System Programming Project 1**

담당 교수 : 김영재 교수님

이름 : 이재진

학번 : 20181671

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**

리눅스의 bash와 같이 동작하는 Shell 프로그램을 구현하는 것을 프로젝트의 개발 목표로 설정하였다.

우선 첫 번째로 ls, cd와 같은 간단한 명령어를 Shell의 프롬프트에 입력하였을 때, 자식 프로세스를 생성해서 명령어에 해당하는 프로그램을 실행하도록 한다. 그러면 부모 프로세스에서는 자식 프로세스가 끝나면 Reaping을 해 주게 되고, 사용자가 또 다른 명령어를 입력할 수 있도록 쉘 프롬프트로 돌아가야 한다. 명령어는 builtin 형식인 것과 실제로 자식 프로세스에서 실행시킬 수 있는 경우로 분류해야 할 것이다.

두 번째로, 하나의 명령어의 출력을 다른 명령어의 입력으로 받아 실행시킬 수 있도록 Pipe redirection 기능을 구현한다. 예를 들어 ls -al | grep abc 와 같은 명령어를 수행할 수 있도록 한다.

마지막으로, ‘&’ 문자를 명령어의 끝에 같이 입력했을 때 그 명령어를 background에서 실행할 수 있도록 한다. 그 말은 곧 background로 실행되고 있는 프로세스가 있을 때 사용자 입장에서는 그 프로세스가 종료될 때까지 기다리는 것이 아니라, 또 다른 명령어를 받아서 foreground로 실행할 수 있어야 한다는 의미이다. 더 나아가서 background에서 실행되고 있는 프로세스나 멈춘 프로세스를 foreground job으로 전환하고, 멈춘 프로세스를 background job으로 전환하는 fg, bg 등 작업을 효율적으로 관리할 수 있는 여러 가지 명령어들이 실제로 작동하도록 구현한다.

2. **개발 범위 및 내용**

* 1. **개발 범위**
* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

ls, cd, mkdir, rmdir, exit 등과 같은 간단한 명령어들을 입력으로 받아서 실행할 수 있도록 한다.

1. Phase 2

Phase 1에서 구현한 명령어들뿐만 아니라 그것들을 파이프 문자로 조합한 파이프 명령어를 입력으로 받아서 실행할 수 있도록 한다.

1. Phase 3

앞에서 구현한 명령어들을 & 문자를 통해 백그라운드 작업으로 실행할 수 있도록 하고, fg/bg/kill/jobs 등의 명령어로 작업을 효율적으로 관리할 수 있도록 한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

Wrapper 함수 Fork()를 사용하여 사용자가 입력한 명령어에 해당하는 프로그램을 실행할 수 있도록 자식 프로세스를 생성한다. 생성한 자식 프로세스의 pid는 Fork()가 성공적으로 실행되었을 때 부모 프로세스에게 리턴되고, 자식 프로세스가 종료되기까지 부모 프로세스는 waitpid 함수에 이 pid를 넘겨줌으로써 기다린다.

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

부모 프로세스는 Fork()를 통해 리턴된 자식 프로세스의 pid를 waitpid의 인자로 넘겨주고 자식 프로세스가 종료될 때까지 계속 기다린다. 자식 프로세스가 종료되면 부모 프로세스는 자식을 reap하게 된다.

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

- 우선 Pipe가 명령어의 마지막에 붙어서 나오는 경우 (즉, ls -al | 과 같은 입력), 뒤이은 명령어를 계속 입력받도록 하고, 파이프 명령어가 한 줄에 다 들어올 때와 똑같이 처리되도록 따로따로 들어온 파이프 문자열을 모두 합쳐서 parse\_pipe()라는 함수에 전달하여 parsing을 한다.

- 파이프 앞의 명령어가 그 출력을 파이프 뒤의 명령어의 입력으로 전달하는 기능은 dup2 함수를 이용하였다. STDIN이 파이프 앞의 명령어의 출력이 되도록 하였고, STDOUT이 파이프 뒤의 명령어의 입력이 될 수 있게 하였다.

- 파이프 앞의 명령어는 Fork()를 통해 생성된 자식 프로세스에서 dup2()를 통해 위 기능이 시행된 이후에 execvp를 통해 실행된다. 부모 프로세스는 자식 프로세스가 끝날 때까지 기다린다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

- Pipeline 개수가 여러 개 있어도 앞의 과정을 반복하도록 recursive하게 함수를 작성하였다. (run\_pipe)

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

만약 parseline 함수를 통해 명령어의 맨 끝에 ‘&’가 있음이 확인된다면 (bg==1), 부모 프로세스는 자식 프로세스가 끝날 때까지 waitpid와 같은 함수에서 hang 해 있지 않는다. 대신 부모 프로세스는 계속 자신의 작업을 수행하게 된다. 자식 프로세스가 비로소 끝나서 SIGCHLD를 보내게 되면, SIGCHLD Signal 핸들러가 자식 프로세스를 reap해 주게 된다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

- Phase 1

ls, mkdir, touch와 같은 명령어들 대다수는 /bin 디렉토리에 이미 그 명령어들에 해당하는 프로그램이 있다. 그래서 builtin command가 아닌 경우 command에 해당하는 프로그램 이름(argv[0]) 에 strcat 함수로 “/bin/”을 덧붙여서 실행하도록 수정하였다. 또한 /bin 디렉토리에 존재하지 않는 프로그램들 중 현 디렉토리에 존재하는 ./a.out과 같은 프로그램들도 실행될 수 있도록 이 경우에는 strcat을 하지 않았다.

builtin command를 구현하기 위해 cd, exit에 대한 내용을 추가하였다. builtin command가 입력되었을 때 Fork()가 실행되지 않고 eval 함수가 return 되어 새로운 프롬프트를 입력받게 된다.

Quotation mark를 포함하는 명령어가 입력으로 들어왔을 때, quotation mark를 모두 삭제한 명령어를 실행하도록 parseline 함수를 수정하였다.

어떤 프로세스를 실행하고 있는 도중에 Ctrl + C를 누르면 쉘 전체가 종료되지 않도록 sigint\_handler SIGINT 핸들러 함수를 추가하였다.

- Phase 2

파이프 문자가 명령어 끝에 들어오는 경우를 다루기 위해 main 함수에서 입력의 마지막이 ‘|’일 때 계속 뒤에 오는 명령어를 받게 하였다. 그리고 뒤에 이어지는 명령어를 모두 하나로 합쳐서 결과적으로 한 줄에 파이프 명령어가 한꺼번에 들어오는 경우와 같도록 구현하였다.

앞에서의 파이프 명령어를 이제 parse\_pipe()라는 함수에 인자로 넘겨서 parsing을 하게 된다. 파이프 문자를 만날 때마다 그 앞의 명령어를 미리 구현되어 있는 parseline 함수에 넘겨서 처리한 뒤에 pipe\_args라는 3차원 배열에 저장하였다. parsing이 다 끝나면 parse\_pipe 함수는 pipe\_args를 리턴한다.

parse\_pipe 함수가 리턴한 pipe\_args에는 실행할 명령어들이 파이프를 기준으로 하여 차례차례 나뉘어 있다. (예를 들어 명령어가 ls -al | grep abc | sort -r 이라면

parse\_pipe의 리턴값 = { {”ls”, “-al”}, {”grep”, “abc”}, {”sort”, “-r”}} 의 포인터값이 된다.) 만약 명령어에 파이프 문자가 포함되어 있는지 확인하는 변수 is\_pipe 값이 1이라면, parse\_pipe를 실행하여 그 리턴값을 pipe\_res에 저장한다. 그리고 그 pipe\_res 값을 파이프 명령어를 실제로 실행시키는 run\_pipe() 함수에 인자로 넘긴다.

추가한 run\_pipe() 함수는 recursive 함수이고, previous\_fd를 통해 바로 이전에 실행된 명령어의 출력값에 해당하는 pfd(파이프 file descriptor)의 file descriptor 값을 하나의 명령어가 recursive하게 실행될 때마다 저장한다. 그 저장된 file descriptor의 값을 STDIN\_FILENO로 복제하고, 프로그램의 실행 결과가 출력될 pfd[1]의 file descriptor의 값을 STDOUT\_FILENO로 복제하는 과정을 반복한다.

pipe 명령어일 때와 pipe 명령어가 아닐 때를 eval함수에서 pid\_p (pipe), pid\_np (not pipe)로 구분하였다.

eval 함수의 인자를 추가하였다. pipe\_cmd는 메인함수에서 입력된 파이프 문자열을 전달한 것이고, is\_pipe는 명령어에 파이프가 들어가 있는지를 알기 위해 추가하였다.

- Phase 3

- SIGCHLD, SIGTSTP, SIGINT, SIGTTOU, SIGTTIN 시그널 핸들러 함수를 추가하였다.

- background job을 구현할 수 있도록 eval 함수에서 bg==1인 경우에 자식 프로세스를 wait하지 않도록 했고 자식 프로세스가 SIGCHLD Signal을 보낸 경우에 SIGCHLD 핸들러 함수에 waitpid(WNOHANG 옵션)를 사용하여 자식 프로세스를 reap하도록 하였다.

- job 기능을 구현하기 위해 proc tracked\_jobs라는 새로운 구조체 배열을 사용하였다. proc은 구조체로서, 프로세스의 pid와 프로세스 이름(process\_name), 프로세스 상태(stat : 0이면 백그라운드 프로세스, 1이면 정지된 프로세스, 2이면 포그라운드로 실행되고 있는 프로세스), 프로세스의 pgid(process group id), 프로세스가 파이프를 포함하는 프로세스인지 (is\_piped)의 속성을 포함한다. 또한 현재 job list에 있는 job의 개수를 tracked\_num에 저장하였다.

- 프로세스 실행 도중에 Ctrl+Z를 눌렀을 때 그 프로세스에 SIGTSTP을 보내서 멈출 수 있도록 eval 함수를 수정하였다.

- eval 함수에서 foreground job을 waitpid(WUNTRACED 옵션)로 기다려서 waitpid의 status에 따라 job을 추가하거나 삭제하거나 eval함수를 return하는 기능을 추가하였다. Foreground job의 경우에도 끝나거나 멈췄을 때 SIGCHLD를 보내지만, 본인이 작성한 코드에서는 Sigchld handler 함수를 실행하게 되더라도 아무것도 하지 않고 eval 함수에서 직접 Signal Handling을 하게 된다.

- 반대로 background job의 경우에는 SIGCHLD를 보내면 Sigchld handler 함수가 실행되어 만약 자식 프로세스가 멈춘 이유가 SIGTSTP을 받아서라면 job의 상태(stat)를 1(suspended)로 바꾸고 종료한다. 만약 정상적으로 종료되었거나 SIGINT를 받아서라면 job을 삭제한다.

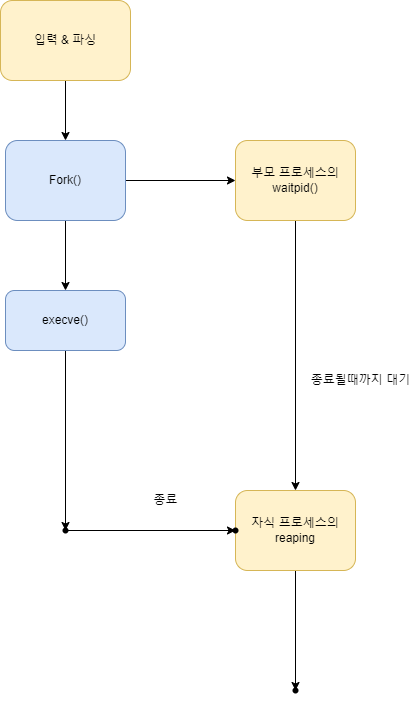
- job list에서 job를 추가하거나 삭제하는 add\_job, del\_job 함수를 구현하였다.

- builtin command에 kill, bg, fg를 추가하였다. fg의 경우 백그라운드로 실행되거나 정지되어 있는 프로세스를 foreground로 바꾸어 실행하도록 하는데, 이 과정에서 tcsetpgrp 함수를 사용하여 터미널의 STDIN에 대한 권한을 잠시 그 프로세스에게 주도록 하였다. 그리고 SIGCONT 시그널을 그 프로세스에 보내서 정지되어 있는 프로세스를 다시 foreground에서 실행되도록 하였다. 그리고 foreground process가 되었으므로 waitpid에서 이 프로세스가 끝날 때까지 기다렸다가 reaping을 하거나, 만약 중간에 사용자가 Ctrl+Z를 눌러 SIGTSTP를 보내게 되면 프로세스를 다시 정지시킨다. bg의 경우 멈춰있는 프로세스에 SIGCONT만 보내주어 구현하였고, kill의 경우 프로세스에 SIGKILL을 보내주어 구현하였다.

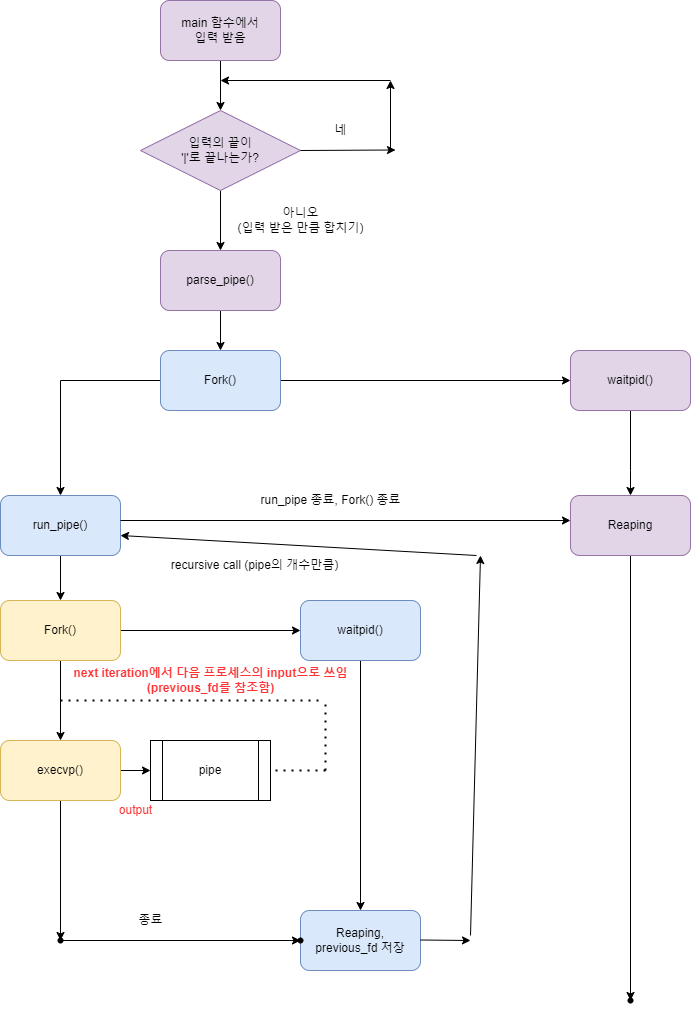
1. **구현 결과**
   1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

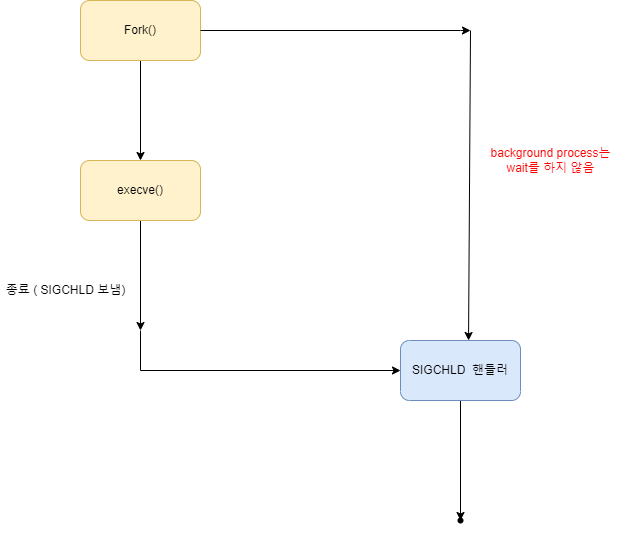
1. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****