**System Programming Project 2**

담당 교수 : 박성용

이름 : 임나현

학번 : 20211582

1. **개발 목표**

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

우리가 자주 사용하는 리눅스 시스템을 직접 구현해본다는 의의가 있다. 이 프로젝트에서는 리눅스 쉘 프로그램에서 동작하는 다양한 명령어들을 똑같이 처리하고 실행하는 기능을 구현할 것이다. 이를 구현하기 위해 process signal, handling, background process의 개념을 이해하고 있어야 한다. 크게 세 가지 phase로 구분하고 있다.

Phase 1에서는 cd, ls 등의 기본적인 단일 명령어를 구현하는 것이 목표이다. 이를 구현하기 위해서는 사용자가 창에 입력한 명령어를 담는 배열이 필요하고 명령어를 조건에 맞게 parsing하는 과정이 필요하다. 또한, fork함수를 사용하여 child proccess에서 명령어를 실행시키고 parent에서 wait함수를 통해 생성한 child process를 reaping하는 과정에 대해 이해가 필요하다. 사용자가 exit이나 quit 명령어를 입력하면 실행되는 쉘 프로그램이 종료되어야 한다. mkdir와 rmdir과 같은 directory를 생성하고 삭제하는 명령어, cd 명령어를 통해 directory를 이동하는 명령어 등을 구현하는 것이 목표이다.

Phase 2에서는 기본적인 단일 명령어에서 확장되어 여러 명령어들이 pipe로 연결되어 있을 때 실행이 되도록 해야 한다. pipe로 연결된 명령어는 단순히 명령어를 여러 개 처리하는 것이 아니라 앞의 명령어의 출력값을 뒤의 명령어가 input값으로 받아서 처리해야 한다. 이를 구현하기 위해서는 pipe함수, pipe의 읽기 쓰기 디스크립터, dup2함수에 대한 이해가 필요하다.

앞의 phase에서 구현한 명령어들은 foreground에서 실행된 명령어였다면 phase 3에서의 목표는 명령어를 background에서 실행시키는 것이다. 사용자가 명령어를 입력할 때 뒤에 & 기호를 사용하면 background에서 실행이 되어야 한다. 또한, jobs 명령어를 입력받으면 현재 존재하는 background process의 목록을 볼 수 있고, bg 명령어와 fg 명령어를 입력받으면 background에서 멈춰 있는 process를 foreground로 옮기거나 background에서 다시 실행시키는 동작을 구현해야 한다. Kill 명령어는 해당하는 background job을 삭제할 수 있다. Phase 2와 마찬가지로 pipe로 연결되어 있는 명령어 또한 처리를 해야 하는데, 파이프로 연결된 명령어 중 &가 붙어 있는 명령어가 있다면 그 명령어는 background에서 실행될 수 있도록 하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

-단일 쉘 명령어를 입력했을 때 기존 쉘과 동일하게 실행이 되어야 한다. main에서 while 루프로 명령어를 받고 처리하는 동작을 계속 하기 때문에 사용자에게 quit이나 exit을 받기 전까지 계속 실행되어야 한다. 잘못된 명령어를 입력 받은 경우에는 에러 메시지를 출력한다. Execv의 ls, mkdir 등의 내장 함수들과, cd, quit 등의 built-in 명령어들로 구분이 된다.

1. Phase 2

-phase 1에서 여러 명령어들을 pipe로 연결하여 실행하는 pipeline 기능을 더한 범위이다. 파이프의 개수는 제한이 없도록 구현했으며 입력받은 명령어의 pipeline의 앞뒤 공백 여부에 상관없이 실행되도록 했다. Grep 명령어나 sort 명령어 들은 뒤의 -I 등의 특수 조건을 덧붙이면 그 조건에 맞는 실행 결과를 출력해야 한다.

1. Phase 3

-명령어를 background에서 실행시킬 수 있도록 한 최종 myshell의 형태이다. 사용자가 입력한 명령어 뒤에 &기호가 붙어있으면 background에서 실행을 하고 실행되고 있거나 정지되어 있는 background process의 목록을 볼 수 있는 jobs 명령어가 존재한다. 이 외에도 bg, fg, kill 명령어들을 통해 background에서 생성된 process를 삭제하고 정지하고 재실행시킬 수 있어야하며 signal을 handling 해야 한다. 파이프라인으로 연결된 경우에도 마찬가지로 & 기호가 명령어 뒤에 붙어 있으면 background에서 실행되어야 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

fork() 함수는 기존 process에서 child process를 생성하는데 parent process와 모든 것을 똑같이 복제하여 만들어 진다. Fork 함수가 실행이 되면 child process에는 새로운 pid가 부여된다. 부모 process에는 child process의 pid가 전달되고 child process에는 pid에 0이 저장되기 때문에 이를 활용해서 child process에서 다른 동작을 수행할 수 있다. 이 프로젝트에서는 사용자에게 받은 명령어를 child process에서 수행하게 하는데 execve() 시스템 콜을 호출한다. Execve를 실행하게 되면 자식 프로세스를 모든 것을 덮어쓰는 새로운 프로그램을 실행하게 되고 실행 결과에 따라 값을 반환한다. Parent process에서는 child process가 종료될 때까지 waitpid 함수를 통해 기다렸다가 종료되었을 때 child process가 zombie process가 되지 않도록 reaping해준다.

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

child process가 종료되면 SIGCHILD signal을 받게 된다. 이는 signal handler에서 처리할 수도 있는데 parent process에서는 sginal(SIGCHILD, sigchild\_handler)를 통해 SIGCHILD signal을 받으면 handler로 이동할 수 있도록 한다. 이후 handler에서는 waitpid함수를 사용하여 종료된 자식 프로세스의 pid를 받고 처리할 수 있다.

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

먼저, 입력받은 명령어에 pipeline이 있는지 확인을 하고 pipeline이 있다면 파이프를 기준으로 명령어들을 parsing한다. 이차원 배열을 사용하여 parsing한 명령어를 다시 공백에 따라 분리한 뒤 저장한다. 명령어를 처리하는 방식은 phase 1과 동일하나 명령어를 실행시키기 전에 약간의 처리과정이 필요하다. 만약 현재 명령어 뒤에 다음 명령어가 연결되어 있다면, 현재 명령어는 자신의 출력값을 STDOUT이 아닌 pipe로 연결을 해야 한다. 다음 명령어는 입력값을 STDIN이 아닌 이전 명령어가 넘긴 pipe로 연결을 해야 한다. 이를 위해 file descriptor가 필요하고 각각의 file descriptor는 pipe에 연결이 되어 있어야 한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

먼저 pipe가 몇 개 존재 하는지 세고 pipe가 없는 경우는 따로 빼서 처리했다. pipe 개수만큼 parsing 해서 배열에 저장하고 파일 디스크립터를 생성한다. 그 후 pipe\_num만큼 fork를 하여 각각의 child process가 서로 연결되게 구현을 했다. 이를 위해 pid는 pipe\_num만큼의 배열로 처리를 하였다. 맨 첫 번재 명령어는 파이프에 자신의 출력값을 쓰기만 하고, 중간에 있는 명령어는 파이프에서 입력값을 받아 다음 파이프에 자신의 출력값을 쓰고, 가장 마지막에 있는 명령어는 입력값만을 파이프에서 받는다.

**Phase3 (background process)**

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

명령어가 background에서 실행되는 방법은 사용자가 명령어 뒤에 &기호를 사용하는 것이다. Foreground에서 명령어가 실행될 때는 하나의 명령어가 실행이 완료된 이후에 다음 명령어가 실행이 될 수 있는데 background에서는 여러가지 job들이 동시에 돌아갈 수 있다는 장점이 있다. 이를 구현하기 위해 우선 &가 있는 경우 bg =1로 설정하였고, background process의 목록을 확인할 수 있는 jobs 명령어를 구현하였다. Background process는 parent process가 wait을 할 수 없기 때문에 종료가 됐을 때 signal로 처리를 해야 한다. 이를 위해 signal\_handler가 필요하고 handler안에서 waitpid를 통해 reaping한다. 또한, bg, fg, kill 등의 명령어를 구현하여 background process를 다룰 수 있다.

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

main함수에서는 while문으로 사용자에게 명령어를 계속 입력받는 구조이며 eval함수를 통해 명령어를 처리하고 실행한다. eval함수에서는 입력받은 cmdline을 우선 parsing하는 과정이 필요하며 이는 parseline 함수를 사용한다. 그 과정에서 명령어 뒤에 &가 붙은 경우는 bg를 1로 설정한다. 이후, builtin command는 builtin 함수에서 따로 처리를 하고 그 이외의 명령어를 실행하기 위해서 fork 함수를 사용하였다. fork함수를 통해 생성된 child process에서 execve 함수를 통해 실행 프로그램을 실행시킨다. 반면에 parent process에서는 종료된 child process를 기다렸다가 reaping시켜준다.

명령어가 두개 이상이 pipeline으로 연결되어 있는 경우에는 parseline으로 명령어 안의 단어를 분리하기 전에 명령어 자체를 pipeline 단위로 parsing하는 과정이 필요하다. 이를 위해 새로운 \*new\_buf[]라는 배열을 만들어 각각의 명령어를 저장하였고, pipe의 개수를 세었다. 그 이후 명령어를 연결하여 실행하기 위해서 pipe 개수만큼 fork()를 호출하였다. 각각의 child process는 명령어를 실행하기 전에 pipe를 생성하고 file 디스크립터를 생성해야 한다. 만약, 가장 첫번째 명령어라면 자신의 출력값을 pipe의 쓰기 디스크립터로 바꿔야 하기 때문에 dup2(fds[i][1], STDOUT\_FILENO)를 사용한다. 중간에 있는 명령어는 입력값을 pipe의 읽기 디스크립터로 바꿀 수 있는 dup(fds[i-1][0], STDIN\_FILENO)를 사용하고, 다음 명령어에게 자신의 출력값도 전달해야하기 때문에 dup2(fds[i][1], STDOUT\_FILENO)를 사용한다. 제일 마지막 명령어는 입력값만 pipe에서 받아오면 되기 때문에 dup(fds[i-1][0], STDIN\_FILENO)를 사용한다. 사용이 완료된 file 디스크립터는 꼭 close를 해주는 것이 중요하다.

Background process 목록을 만들기 위해서 프로세스의 pid를 저장하는 pid\_arr[]배열과 stopped인지 running인지를 구분하기 위한 state\_arr[], 명령어를 저장하는 com\_arr[]가 존재하며 background process의 개수를 세는 pid\_num 또한 필요하다. foreground에서 실행되다가 ctrl-z에 의해 중단되는 경우 background에 저장되어야 하기 때문에 이를 처리하기 위한 forpid\_arr, forcom\_arr, forstate\_arr배열이 존재한다.

SIGCHID signal을 handling하기 위해서 sigchild\_handler 함수를 선언하였다. 이 handler안에서는 다른 signal에 의해 영향을 받지 않아야 하기 때문에 sigprocmask 함수를 사용하여 signal들을 전부 block 시킨다. 그 후, waitpid함수를 while문으로 받아 종료된 child의 pid를 받고, 그 pid가 들어있는 pid\_arr배열의 값을 -5로 바꾼다. 값을 변경하는 이유는 jobs를 실행할 때 -5의 값이 들어있는 배열은 이미 종료된 background process이기 때문에 삭제하고 출력하기 위해서이다.

SIGTSTP signal을 handling하기 위해서 sigtst\_handler 함수를 구현하였다. 마찬가지로 다른 signal에 의한 영향을 없애기 위해 sigprocmask함수를 통해 모든 signal을 block하였다. kill함수를 사용하여 해당하는 foreground process 전체 배열 arr에 SIGTSTP signal을 전송한다.

Builtin\_command에서 jobs의 경우 저장한 background process만큼 for문을 통해 sigchild\_handler에서 -5로 설정된 값이 있는 것을 찾아 strcpy를 통해 그 배열의 내용을 지우고 나머지를 연결시켜준다. 그 이후에 pid\_num만큼 running상태인지 stopped 상태인지를 구분하여 출력한다.

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)**

도표, 라인, 평면도, 화이트이(가) 표시된 사진

자동 생성된 설명

1. **Phase 2 (pipeline)**

도표, 라인, 화이트, 평면도이(가) 표시된 사진

자동 생성된 설명

1. **Phase 3 (background)**

도표, 라인, 화이트, 평면도이(가) 표시된 사진

자동 생성된 설명