**System Programming Project 2**

담당 교수 : 박성용

이름 : 조원빈

학번 : 20201644

1. **개발 목표**

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

이번 프로젝트에서는 실제 Shell이 작동하는 원리를 이해하고 최대한 비슷하게 기능하도록 나만의 MyShell 프로그램 만들었다. Shell 자체가 매우 많은 수의 명령어와 기능을 제공하므로 원활하게 프로젝트를 해내기 위해 3개의 phase로 나누어 구현하였다. 첫 번째 phase에서는 쉘과 비슷하게 명령어를 한 줄씩 입력 받고 그 명령어를 수행하도록 구현하였다. 두 번째 phase에서는 쉘의 핵심 기능 중 하나인 파이프를 구현해 여러 프로그램(프로세스)를 이어주고 입출력을 주고받을 수 있도록 구현하도록 구현하였으며, 마지막으로 세 번째 phase에서는 앞의 두 phase에 구현한 것에 더불어 back ground에서도 프로세스를 실행할 수 있도록 했으며, 실행중인 여러 프로세스를 job을 통해 관리하도록 하였다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

* 뼈대 코드에서 while문을 돌면서 명령어를 한 줄 씩 받아온다.
* parseline함수를 이용해 명령어를 공백을 기준으로 분리해주었다.
* Builtin command인 경우에는 직접 구현한 함수를 이용해 처리해주었다. Cd(change directory) 함수에서는 인자 개수가 잘못되었거나 파일이 없는 경우를 예외로 처리해주었다.
* ‘ls, mkdir, rmdir, cat, echo, touch’ 명령어의 경우 이미 /bin폴더에 있으므로 execvp 함수를 이용해 실행하도록 구현했으며, 그 전에 fork를 떠 child process에서 호출한 후 waitpid를 이용해 현재 명령어 수행이 끝날 때까지 parenet process는 기다리도록 구현하였다.

1. Phase 2

* Pipe를 통해 여러 명령어가 한 줄에 입력되었을 때도 처리할 수 있도록 구현하였다.
* Parseline 함수에서 파싱을 할 때 pipe(|)를 따로 처리해주었고, grep 명령어를 쓸 때 따옴표가 뒤의 인자에 올 수 있으므로 이를 처리해주었다.
* Go\_eval 함수를 새로 정의해 현재 명령어 안에 파이프가 존재할 경우 파이프를 기준으로 왼쪽 부분을 fork를 떠서 child process에서 먼저 재귀적으로 수행하고, 그 이후 오른쪽 부분도 똑같이 재귀적으로 실행되도록 구현하였다.

1. Phase 3

* 명령어를 입력할 때 마지막에 &를 붙이면 Process를 back ground에서 실행하도록 구현하였다.
* fg %(job number) 명령어를 통해 backgound의 process를 foreground에서 running할 수 있도록 하였다.
* bg %(job number) 명령어를 통해 backgound suspended된 process를 다시 running 상태가 되도록 구현하였다.
* kill %(job number) 특정 process를 종료할 수 있도록 하였다.
* Sigint, sigtstp signal의 handler를 각각 직접 새로 정의해줌으로써 ctrl+c, ctrl+z 를 입력했을 때 myshell에서 foreground에서 돌아가는 프로그램이 있다면 각각 stop, terminate하도록 구현하였다.
* Jobs 명령어를 통해 backgound에서 stopped 또는 running 상태인 process를 볼 수 있게 하였다.
  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

‘quit, exit, cd’ 가 아닌 명령어, 즉 builtin command가 아닌 경우에는 fork를 하여 parent process를 복제한 child process를 만들었다. 그리고 child process 에서는 fork함수의 return 값이 0이 되는데 이 경우 if문 내에서 명령어를 수행하도록 구현하였다.

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

현재 phase에서는 signal을 딱히 이용하지 않았다. Parent process에서 waitpid 함수를 통해 child process가 명령어를 수행하고 종료될 때까지 기다리도록 하였으며, child process가 종료된 후에 다음 단계로 넘어가도록 구현하였다.

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

파싱을 할 때 우선 pipeline이 있으면 하나의 argument로 따로 처리를 해주었다. 그리고 매 명령어를 수행할 때 마다 가장 왼쪽에 있는 pipeline을 찾는다. 만약 존재한다면 fork를 한 후 child process에서 왼쪽 부분의 명령어를 수행하고 오른쪽 부분의 명령어를 함수를 재귀적으로 호출해 구하였다. 또한 pipeline으로 이어져 있는 경우 process간의 입출력이 가능해야 하므로 file descriptor와 dup2() 함수를 이용해 각 process의 입출력을 pipeline에 연결해주었다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

매번 현재 명령어에서 가장 왼쪽에 있는 pipieline을 찾아 왼쪽 부분을 수행한 후 그 오른쪽 부분을 재귀적으로 호출하기 때문에 Pipeline의 개수는 딱히 중요하지 않다.

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

Background process을 구현하기 위해 job의 개념을 도입하였다. 만약 background에서 process를 실행하는 경우에는 jobs에 추가해주고 계속해서 입력을 받았다. 여기서 background가 종료되면 reap을 해줘야 하는데 이를 위해 signal을 이용하였다. SIGCHLD의 signal handler를 직접 정의해줘서 background process가 끝날 때마다 jobs를 갱신해주었다. foreground process가 들어왔을 때 ctrl+z, ctrl+c를 입력했을 때 myshell 프로그램 자체를 종료시키는 게 아니라 foreground process를 종료하도록 구현하기 위해 signal handler(SIGINT, SIGTSTP)를 직접 정의해주었다. 또한 SIGCHLD signal이 job에 추가하기 전에 전달될 경우 등을 막고, foreground 작업이 끝날 때까지 대기하는 등의 기능을 구현하기 위해 signal을 block하는 부분도 추가하였다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* void go\_eval(char\* argv[]);

pipeline을 통해 한 줄에 여러 명령어가 들어왔을 때 이를 처리하기 위해 새로운 함수인 go\_eval 함수를 정의하였다. 함수 내에서는 pipeline이 없을 경우 명령어를 처리하고 pipeline이 있는 경우 pipeline을 기준으로 왼쪽 부분을 먼저 재귀적으로 호출해 수행하고, 그 결과값을 file descriptor를 통해 오른쪽 부분으로 전달한 후 오른쪽 명령어 부분을 재귀적으로 호출하도록 구현하였다.

* Parseline 함수에서 grep 명령어가 정상적으로 실행될 수 있도록 따옴표(‘, “)를 처리해주었으며, pipeline이 들어온 경우에는 따로 argument로 저장해서 go\_eval함수 내에서 pipeline을 처리하도록 하였다.
* Signal 처리

Background process가 종료되면 reap하기 위해 SIGCHLD signal의 hanlder를 직접 정의했는데, 매 호출마다 wait를 통해 job에 있는 process가 종료되면 job목록에서 제거하도록 구현하였다. 또한 ctrl+c, ctrl+z 입력을 따로 처리하기 위해 SIGINT, SIGTSTP handler를 직접 정의해주었다.

* cur\_pid 변수를 통해 현재 foreground에서 돌아가고 있는 process의 pid를 저장해두었다.
* Jobs를 구현하기 위한 요소들
* char Jobs[MAXJOBS][MAXLINE];
* int job\_cnt, jpid[MAXJOBS];
* /\* 0 : terminated, 1 : running, -1 : suspended\*/
* int job\_state[MAXJOBS];

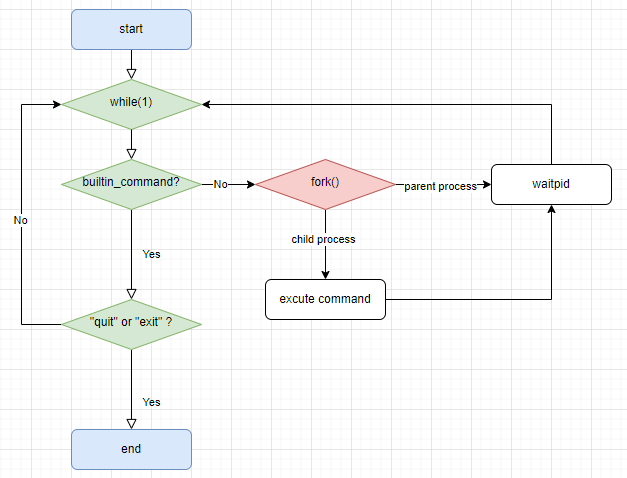
jobs에 background에서 돌아가는 중인 process의 명령어들을 저장해두었으며, job\_cnt에 지금까지 만든 job의 개수, jpid 배열에 i번째 job의 pid를 저장해두었다. 또한 job\_state배열에 각 job의 상태를 저장했는데, terminated 되었으면 0, running이면 1, suspended면 -1 값을 저장하였다.

* Background에서 process가 돌아가는 경우 fork를 해 하나의 child process를 만들고 그 process를 하나의 job처럼 관리하였다. Addjob 함수를 이용해 job을 추가하도록 구현하였다.

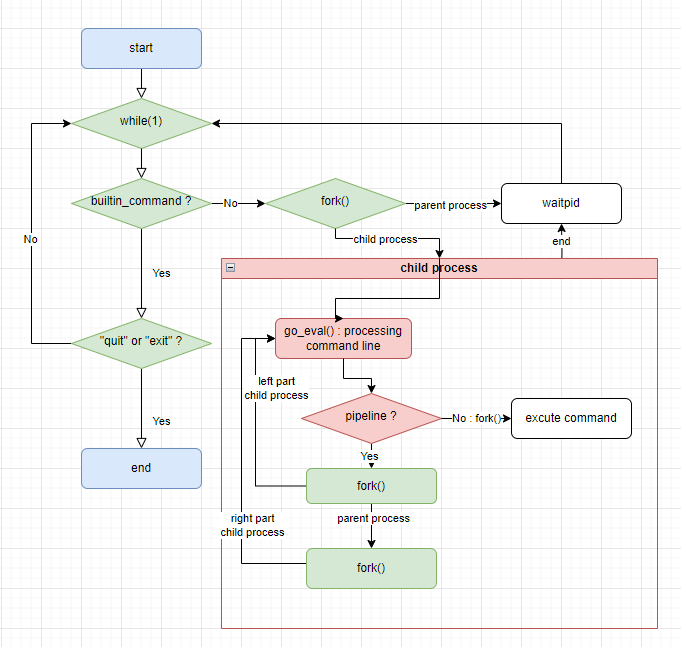
1. **구현 결과**
   1. **Flow Chart**

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

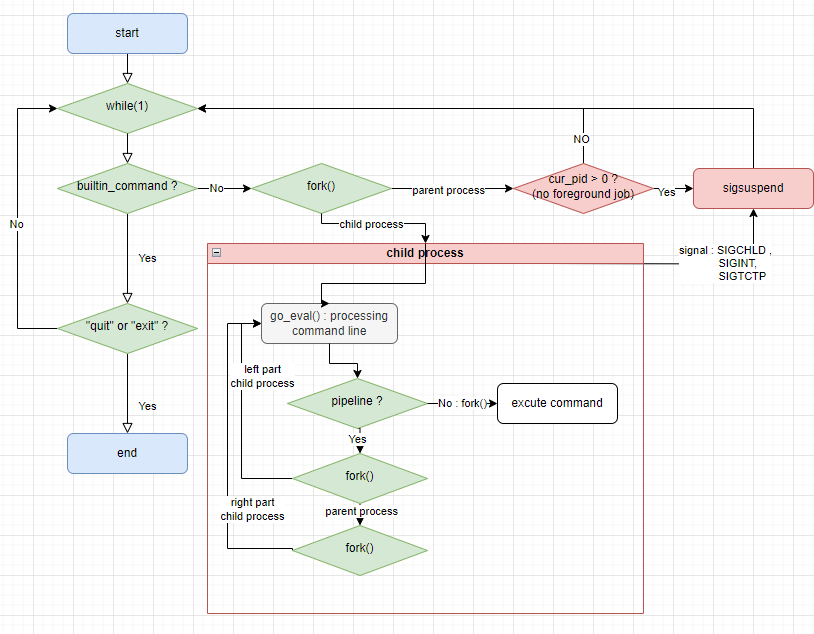
1. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****