**Multicore Programming Project 1**

담당 교수 : 최재승 교수님

이름 : 임해진

학번 : 20180223

1. **개발 목표**

이번 프로젝트의 목표는 쉘 프로그램을 구현하는 것이다. 프로젝트에서 구현되어야 하는 쉘 프로그램의 기능은 아래와 같다.

* 사용자의 명령어를 입력받아 명령어를 수행한다.
* 명령어에 따른 외부 프로그램을 실행해야 한다.
* history 명령어를 통해, 사용자의 명령을 기록하고 다시 실행할 수 있도록 한다.
* pipeline을 구현하여, 하나의 명령어의 결과가 다른 명령어의 입력으로 주어질 수 있도록 한다.
* background 프로세스를 구현하여, 명령어가 백그라운드에서 실행되도록 한다.

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

myshell 프로그램을 실행하면, 무한루프를 돌면서 사용자의 명령어를 입력받고 명령어를 처리한다. 사용자가 입력한 명령어는 history에 관리되며, history 명령어를 통해서 다시 실행할 수 있다.

1. Phase 2

명령어에서 pipe기능을 붙여 사용할 수 있다. 한 명령어의 결과값이 다른 명령어의 입력 값이 되도록 명령할 수 있다.

1. Phase 3

명령어를 백그라운드로 실행할 수 있다. fore ground 프로세스를 정지하고 background 프로세스로 변경할 수 있고, 다시 실행을 재개하고 fore ground 프로세스로 바꿀 수 있다. 그리고 실행중인 프로세스를 종료할 수 있다.

* 1. **개발 내용**

**Phase1 (fork & signal)**

* + fork를 통해서 child process를 생성하는 부분에 대해서 설명

명령어가 내부 프로그램이라면, 명령어에 해당하는 함수를 호출해서 명령어를 처리하였다. 하지만 외부 프로그램이라면, 외부 프로그램을 실행시키기 위해 Fork함수를 호출했다. Fork()의 결과가 0이라면, child process이기 때문에 execvp함수를 호출해서 환경변수 PATH에 있는 프로그램이라면 실행되도록 구현했다.

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

Fork()를 한 이후, parent process는 Waitpid()함수를 사용해서, child process가 종료될 때까지 대기한다. Waitpid()함수로 child process를 성공적으로 reaping 했다면, 다시 명령어 입력을 대기한다.

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

pipeline은 pipe\_execute()함수에 구현을 했다.

명령어에서 “|”문자가 먼저 있는지 판단한다. 그리고 “|” 앞부분의 명령어를 fork()를 호출한 후 실행한다. 이 명령어 전에 파이프가 사용되었다면, dup2()함수를 사용해서 함수의 인자로 넘겨진 my\_fd[0]에서 입력값을 읽도록 등록해준다. 파이프가 필요한 명령이라면, fd[1]에 결과값을 쓸 수 있도록, dup2()함수를 호출해 표준 출력때신 fd[1]을 사용하도록 등록해준다.

이후에는 내부 함수를 호출하거나 execvp()함수를 호출해 프로그램을 실행한다. parent process는 child process가 종료될 때까지 대기한 후 reaping한다. 이때, 아직 뒤에 실행할 명령어가 남았다면 다시 pipe\_execute()함수를 호출해 남은 명령어를 실행한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

재귀 함수를 통해 pipeline을 구현했다. pipe의 개수에 상관없이, 앞부분의 명령을 실행한 후 뒤에 명령이 있다면 재귀로 함수를 다시 호출에 뒷부분의 명령이 실행되도록 했다. 재귀로 구현했기 때문에 앞부분부터 차례대로 명령이 실행되고, 차례대로 pipeline이 연결된다. 또한 pipeline의 개수에 제한이 없도록 구현했다. (하지만 pipeline의 개수가 무수하게 많으면, 함수가 stack에 쌓이면서 stackoverflow의 위험이 있다)

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

& 문자가 명령어에 포함되어 있다면 fork를 한 후, child process의 종료를 기다리지 않고 shell은 다음 작업을 한다. signal handler를 정의해서, SIGCHLD 시그널이 오면, 백그라운드 프로세스를 reaping 해준다.

* 1. **개발 방법**

<Phase 1>

명령어를 실행하는 부분을 구현하기 위해 myshell\_execute()이라는 함수를 작성했다. myshell\_execute()에서는 내부 명령어인지 판단하는 builtin\_command()함수가 호출된다. 내부 명령어라면 builtin\_command()함수에서 명령어를 실행한다. 외부 명령어라면 fork()로 새로운 프로세스를 만든 후에 외부 프로그램을 로드해 실행한다. myshell\_execute()에서는 fork한 child process가 종료되길 기다린 후 reaping함으로써 명령어 실행을 끝낸다.

명령어의 history를 구현하기 위해, 구조체 hisNode를 정의했다. 이 구조체를 통해 명령어의 history를 Linked List방식으로 저장한다.

History Linked List를 관리하기 위해 add\_history()와 print\_history()함수를 구현했다. 명령어를 입력할 때마다, 위해 add\_history()가 호출되며, 명령어를 hisNode에 저장하고 Linked List의 원소로 추가한다. history를 조회할 때는 print\_history() 함수를 호출해서, Linked List를 순회하며 명령어들을 출력해준다.

명령어의 history는 프로그램이 종료되어도 사라지면 안되기 때문에 프로그램이 종료될 때 history.txt파일에 명령어를 기록하도록 했다. 이를 위해서 save\_history(), load\_history()함수를 구현했다. 프로그램이 로딩되면 load\_history()함수가 호출되고, history.txt파일을 읽어와 Linked List를 구성한다. 프로그램이 종료될 때는 save\_history()함수가 호출돼서, Linked List의 원소들을 history.txt에 저장한다.

history 명령어 !!, !#를 구현하기 위해 myshell\_translate\_input()함수를 구현했다. 이 함수에서는 명령어의 1번째 부분에서 !! or !#가 있는 찾은 후, 해당 위치에 들어갈 명령어를 history에서 찾은 후 대체한다.

<phase 2>

pipeline을 구현하기 위해 pipe\_execute()함수를 정의했다. pipeline에 사용될 fd는 chd\_fd라는 정수 배열로 정의했고, pipe()함수를 통해 fd를 배열에 할당해주었다. pipe\_execute()함수에서는 pipeline을 감지하고, 프로그램의 입력과 출력을 pipeline에 사용되는 fd로 할당해준다. 또한 재귀 호출로, pipeline의 개수에 상관없이 명령어가 실행되도록 한다.

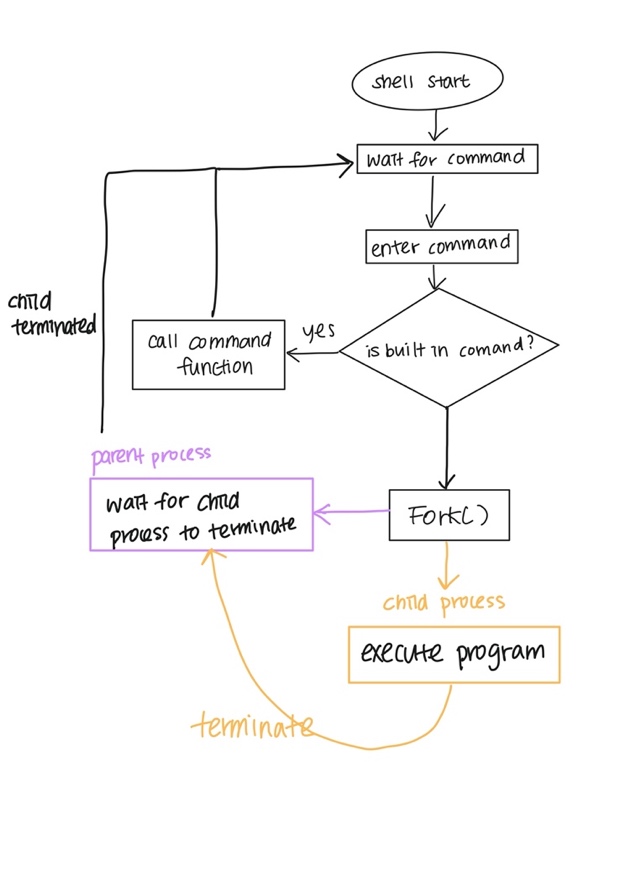
<phase 3>

background job을 구현하기 위해 구조체 BgNode를 정의하고, 백그라운드 프로세스들의 목록을 Linked List 형태로 관리했다. 백그라운드 프로세스가 추가되면 Linked List에 원소를 추가하고, 프로세스가 종료되거나 fore ground로 변경되면 삭제한다. Linked List를 관리하기 위해 node를 추가하고 삭제하는 함수 add\_job(), delete\_job()를 정의했다.

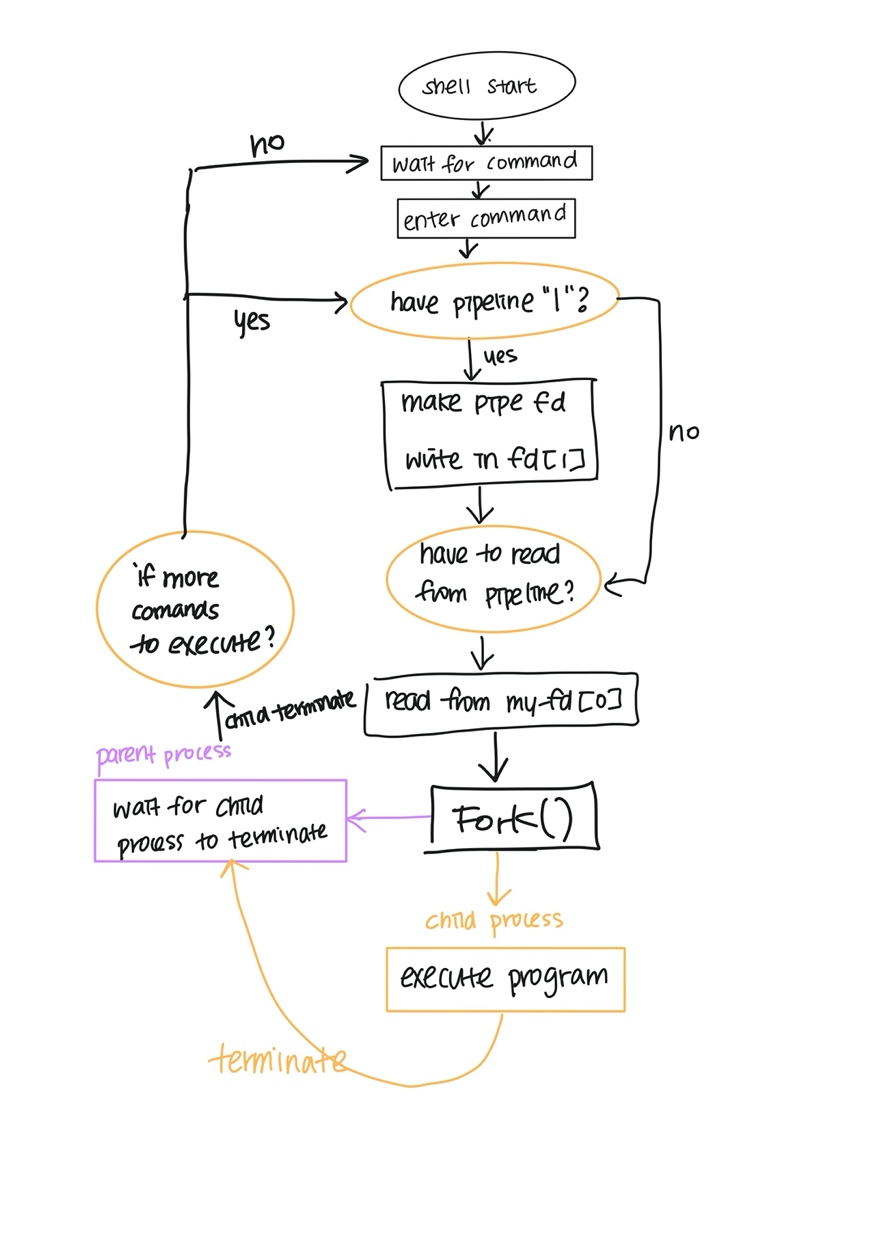
child process를 reaping하기 위해 SIGCHD handler를 정의하고, 프로세스가 종료되면 background process list에서 삭제하기 위해 delete\_job()을 호출했다. ‘control+z’로 foreground process의 작동을 정지하고, background job에 추가하기 위해 SIGTSTP handler를 정의했다.

background process 관련 명령어인 ‘jobs’, ‘bg’, fg’, ‘kill’ 을 구현하기 위해 command\_jobs(), command\_bg(), command\_fg(), command\_kill() 함수를 구현했고, id로 프로세스의 Node를 찾는 get\_bgNode\_by\_id() 함수를 구현했다. 각 노드를 순회하면서, 프로세스 관련 정보를 출력하는 print\_jobs() 또한 구현했다.

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

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

