**Multicore Programming Project 1**

담당 교수 : 최재승 교수님

이름 : 유종선

학번 : 20191611

1. **개발 목표**

* 기본적인 몇몇 shell commands를 실행할 수 있는 나만의 shell을 구현한다. (ex. cd, ls, history, exit .etc.)
* 명령어 와 명령어 사이의 파이프라인(“|”)이 들어있는 command를 실행할 수 있는 shell을 구현한다.

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

: 외부 명령어와 내부 명령어를 구현하였다. 예를 들면 ls 나 cd 같은 명령어를 입력했을 때, shell과 같은 결과가 나오도록 하였다. 더해서 history 명령어와 !!, !# 명령어를 구현하였다. History command를 입력하면, 내가 입력했던 명령어들이 결과로 나타난다. 중복은 제거하고, history는 infinitely하게 생성된다. !!는 바로 이전의 명령어를 실행하고, !#은 #번째로 입력했던 명령어를 history에서 찾아서 실행한다.

1. Phase 2

: ls | grep *filename* 과 같은 파이프라인이 포함된 명령어를 실행한다. ls의 결과 값을 그 뒤의 명령어인 grep *filenam*e이 받아서 실행한 후의 결과값을 우리에게 출력한다. cat *filename* | grep -i “abc” | sort -r 과 같은 파이프라인이 여러개 있는 경우도 파이프라인이 단독으로 있는 경우와 같이 앞부터 차례대로 실행하여 최종 결과를 출력한다.

1. Phase 3

:

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

: builtin\_command나 history와 cd 명령어를 제외한 명령어들은 execve 명령어를 사용하여 실행한다. 이 함수는 fork()함수를 통해서 child process를 생성한 후에 return 값이 0이면( = child process / pid = 0)이면 execve 함수를 실행하여 명령어를 처리하였다.

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

: 자식 프로세서가 종료되면, 부모 프로세스에서 waitpid 함수를 통해서 reaping하도록 설계되었다. 명령어 실행은 자식 프로세서에서 진행되고 이후에는 부모 프로세서로 return 된다.

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

: 먼저 파이프라인이 들어있는 명령어를 처리하는 함수를 따로 생성했다. 마지막 명령어를 처리하는 경우와, 그 외의 경우로 나누어서 구현하였다. 다음 명령어가 남아있는 경우에는, Fork함수를 통해 child process를 생성한다. Child process에서는 dup2함수를 통해서 fd[0]가 STDIN\_FILENO를 참조하도록 하고, 필요없는 fd는 close한다. Parent process에서는 다음 명령어를 위해서 재귀함수를 실행하고 child process가 종료를 회수하기 위해서 waitpid함수를 실행한다.

마지막 명령어인 경우에는 fork함수를 실행 후 child process에서 dup2함수를 통해 fd[0]가 STDIN\_FILENO를 참조하도록 하고, 사용하지 않는 fd는 close해준다. 그리고 해당 명령어를 실행 한다. Parent process에서는 child process의 회수를 위해서 waitpid함수를 선언하고, fd를 close하고 종료한다. 마지막 명령어이기 때문에 더이상 함수를 호출할 필요가 없다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

: 파이프 개수가 증가함에 따라서 반복문보다 재귀함수를 사용하는 것이 효율적이라고 생각해서, 파이프를 실행하는 함수를 재귀로 구현하였다. 처음 명령어가 입력되면 파이프라인의 갯수와 각각의 명령어를 parseline함수를 통해서 잘라내어서 마지막 실행임을 알려주는 flag를 통해서 파이프라인이 끝날때까지 재귀함수를 실행한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
  1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
  + 1. Phase1: 외부 명령어들의 경우에는 fork함수를 통해서 child process 를 생성 후 명령어들을 수행한다. History는 사용자가 입력했던 명령어들을 저장하여 보여주는 명령어이다. 명령어들을 저장하기 위해서 history.txt파일을 생성하여서 저장해두었다. 쉘을 실행시킬 때마다 파일을 열어서 명령어들을 저장하였다. Cd명령어를 통해서 홈 디렉토리로 가도 파일에 접근할 수 있도록, fopen의 파일 디렉토리를 설정해주었다. Getcwd 함수를 사용하여 실행 디렉토리를 받아서, 받아온 디렉토리 안의 파일을 열고 닫을 수 있도록 구현하였다. History에 저장하는 것과 하지 않는 것을 구별하여 두었다.

!!와 !#의 명령어 또한 조건을 두어 구현하였다. !!의 경우 prev\_command 를 두어, 이전 명령어를 저장하였다가 !!가 입력되면 사용할 수 있게 해주었다. 이는 history에 입력되지 않는다. !#은 #횟수 만큼 fgets함수를 돌려서 history로부터 명령어를 읽어온다. 범위가 넘어가는 숫자에 대해서는 예외처리를 해주었다.

* + 1. Phase2: 파이프라인을 구현하였다. 파이프라인의 명령어를 시행하는 함수 pipe\_exec를 생성했다. 여기서는 명령어를 parse하여 명령어의 갯수만큼 recursive하게 동작한다. Fd[2]를 선언하여 파이프를 열고 닫으며 구현하였다. Dup2함수를 사용해서, 파일 서술자를 변경해주고, 닫고(close), 사용하지 않는 서술자 또한 닫아준다. 마지막 명령어를 child process가 실행하고 나면, parent process는 waitpid함수를 호출하여 reaping 해준다. 그리고 서술자를 닫으면 함수가 종료된다. 이 함수를 구현할 때 중요한 것은 서술자를 열고 닫는 것이다. 앞의 수행 결과가 다음 명령어의 입력이 되기 때문에, 그 방향에 맞추어서 열고, 필요 없는 fd는 닫아주는 것이 중요하다.

또한, 따옴표(“, ‘)에 대해서 parseline에서 처리가 되어 있지 않아 추가적인 구현을 하였다. 입력한 명령어에서 따옴표가 입력될 경우에는 따옴표를 지워서 명령어에 다시 넣어주는 방식을 택하였다. 새로운 함수를 생성하지 않고 parseline함수 안에서 처리하였다. 이렇게 처리 해주면 cat filename | grep -i “abc” 에서 “abc” 같은 부분을 처리할 수 있도록 도와준다. “abc” -> abc로 바꾸어 처리한다. 이렇게 바꾸어주면 execve함수가 이를 읽어서 명령어를 처리해준다.

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

**도표, 개략도이(가) 표시된 사진

자동 생성된 설명**

: parent process에서 fork()를 통해서 child process를 생성 후, child process에서 명령어를 수행한다. Parent process는 child process가 끝나기(return to parent process)를 기다리다가 reaping하면서 프로그램이 종료된다.

1. **Phase 2 (pipeline)**

**도표이(가) 표시된 사진

자동 생성된 설명**

: 처음 parent process에서 fork()함수를 통해서 child process를 생성하고, 파이프라인의 개수에 따라서 fork()함수를 통해서 또 다른 child process 를 생성한다. Child process에서 실행한 결과들은 파이프라인을 통해서 다른 child process로 입력되어 명령어를 수행할 수 있게 된다. 마지막 명령어를 수행하면 return to parent process한다. 그림에서 보면 아래쪽의 화살표가 오른쪽 명령어의 실행결과가 왼쪽 process로 넘어가서 입력으로 들어오는 부분이다. 왼쪽 process에서는 오른쪽의 결과값을 입력으로 받아서 실행하게 된다.

1. **Phase 3 (background)**