**System Programming Project 4**

담당 교수 :　김영재

이름 : 장수길

학번 : 20161634

1. **개발 목표**

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

본 프로젝트의 목표는 시스템 레벨의 명령어들을 이용하여 단계적인 과정을 거쳐 기본적인 리눅스 쉘 (shell)의 동작 원리를 파악하고 그것을 구현함에 있다.

결과물은 명세서의 단계별 목표에 명시되어있는것과 동일하게, 기본적인 리눅스 built-in command들과 internal shell command들을 수행해야 하며, 파이프를 이용한 프로세스 간 통신을 사용하여 파이프를 포함한 다양한 명령어를 처리해야 한다.  
추가적으로 & 옵션을 사용한 백그라운드 프로세스와 부가적인 built in command 들에 대해서도 올바른 결과를 출력해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

Phase 1을 구현한 결과 myshell 프로그램은 성공적으로 internal command들인 cd, ls, mkdir, touch, cat, echo 등의 명령어들을 수행할 수 있었다.

1. Phase 2

Phase 2를 구현한 결과 myshell 프로그램은 성공적으로 Phase 1 에서 언급한 command들에 대해서 파이프를 사용해 조합된 다양한 명령어들에 대해서 올바른 결과를 출력하였다.

1. Phase 3

미구현

* 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

myshell에 명령어가 주어질 경우, 가장 우선적으로 처리하게 되는 것은 명령어들의 parsing 과정이다. 예를 들어 cat test 라는 명령어가 주어질 경우, 프로그램은 공백으로 구분된 단어들을 전처리하여 cat 과 test라는 각 단어들을 argv 배열에 옮겨준다.   
이후 built-in command의 여부를 확인해준다. 만약 built – in command라면 fork할 필요가 없으므로 main process에서 명령어를 처리해주고, build-in command 가 아닐 경우에는 fork 하여 자식 프로세스를 하나 생성해준다.

생성한 자식 프로세스에서는 execve 명령어를 통해 /bin 디렉토리 내부의 명령어들을 수행시켜 주어야만 하기 때문에 새로운 명령어들을 /bin/이라는 prefix 뒤로 복사해주었다. 이후 execve를 호출하는 것을 통해서 결과를 출력해주었다.

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

internal command들의 프로세스가 종료되고 connection이 끊어지는 경우에는 이미 fork 된 자식 프로세스의 내부에서 일어나는 일들이기 때문에 단순히 execve함수에서 종료 시그널을 보내준다.

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

파이프를 구현하기 위해서는 다양한 전처리가 필요했다. 일단 파이프로 구분되는 명령어의 집합들을 각각 따로 분리해줄 필요가 있었다. (ex: cat test | grep -v “abc” | sort -r 의 경우, 1. (cat test) 2. (grep -v “abc”), 3.(sort -r))

기본적인 디자인은 파이프를 만나거나 명령어의 끝에 도달 할 때 까지 새로운 집합에 단어들을 추가해주고, 파이프에서 읽어온 값들에 대해서 집합에 해당하는 명령어를 수행시켜 준 후, 다시 파이프에 출력시켜주는 것을 반복하는 재귀함수를 이용하여 작동하도록 구성하였다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

단순히 하나의 파이프라인만 존재할 경우 하나의 file descriptor array를 이용하여 간단하게 구현이 가능하지만, 이번 프로젝트의 경우 파이프라인의 개수에 제한이 없었기 때문에 파이프의 개수에 비례하도록 file descriptor array를 맞춤형으로 생성하거나, 재귀적인 방법을 이용해 파이프를 만날때마다 새롭게 file descriptor array를 생성해야만 했다.

이번 프로젝트에서는 후자를 선택하여 진행하였다.

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

우선적으로 fork를 하고 foreground process는 background process를 기다리지 않고 실행하고, background process는 foreground process를 기다리지 않도록 서로 독립적으로 실행되어야 한다. 따라서 background process는 실행하는 순간에 jobs list에 추가되어야 하고 sigchld handler를 수정하여 신호를 받을 때 waitpid의 return 값으로 받은 pid 값에 따라서 jobs list를 업데이트 해야 한다.

fg, bg 등의 명령어는 tcsetpgrp()등의 함수를 사용하면 가능할것으로 보인다.

개요는 파악했지만 시그널 간의 race문제등을 해결하지 못하였다.

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

Phase1의 구현을 위해서는 기본 템플릿인 shellex.c 파일의 built-in 함수에서 cd 등의 기본적인 명령어를 구현해줄 필요가 있었다. 따라서 B부분에서 언급한 바와 같이 execve를 수행하기 위한 경로 수정을 해준 뒤, built-in command 들에 대해서는 cd 명령어를 구체적으로 수행하기 위해서 추가적으로 함수를 수정하였다. cd 명령어는 일반적으로 뒤 인자에 따라서 chdir 명령어를 수행시키는 것을 통해 알아서 수행하지만, 단순 cd의 경우 home directory로 이동하는 부분을 추가적으로 구현하였다. 이를 위해서 getenv(“HOME”) 함수를 사용하여 유저의 기본 디렉토리를 구해주고, 그곳으로 이동시켰다.

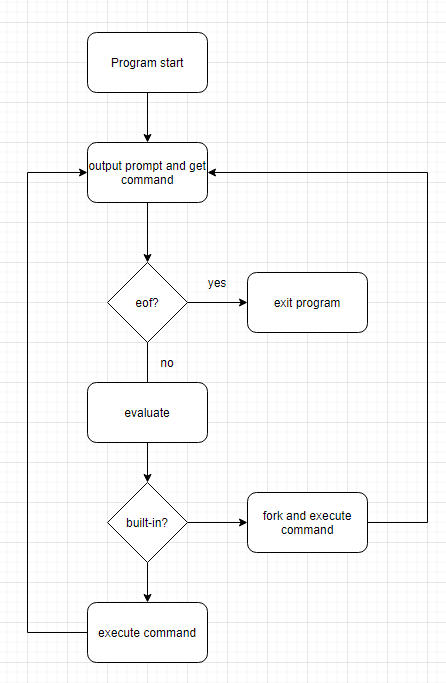
Phase2의 구현을 위해서는 일단 조합된 명령어를 수행해야 하기 때문에 pipe\_execute라는 함수를 추가하였다 .  
B에서의 설명과 마찬가지로 pipeline 단위로 끊어진 명령의 집합을 구성해주어야 했기 때문에 적당한 전처리를 통해서 pipeline 단위로 명령들을 끊어서 새로운 arg배열에 넣어주었고, pipeline을 지날 때마다 새로운 재귀함수를 호출하면서 파이프를 생성해주었다. 생성된 파이프를 통해서 구성한 명령어 집합을 수행한 결과값을 통과시켜주고, 호출된 재귀함수에서는 결과를 읽어와 다시 새로운 명령어 집합을 수행시켰다.

특이사항으로는 grep 명령어의 경우, “abc”와 같이 따옴표가 포함된 문장은 지원지원 않은 것을 확인할 수 있었다. 따옴표의 존재 이유는 grep “ab ab”와 같이 여러 단어로 이루어진 문자열들을 탐색할 때 사용되는데, 이것을 구현하기 위해서 따옴표를 발견할 경우, 끝맽는 따옴표를 발견할 때 까지 문자열을 하나의 배열에 구성해주었다. 만약 따옴표가 끝맽지 않아질 경우 오류를 출력하였다.

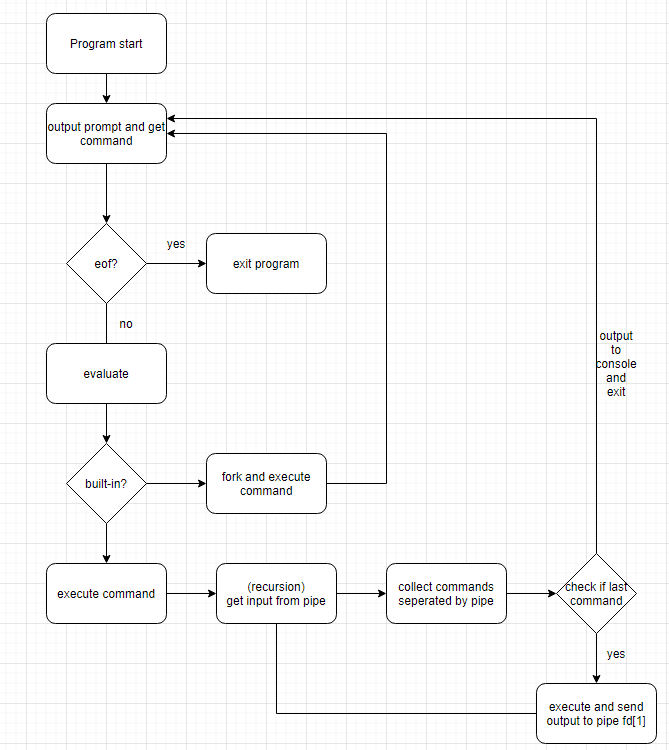
1. **구현 결과**
   1. **Flow Chart**

* **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)**