**System Programming Project 1**

김영재:

조창희 :

20200118 :

1. **개발 목표**

이번 프로젝트는 리눅스 환경에서 동작하는 자신만의 쉘 프로그램을 구현한다. 쉘이란 명령어/프로그램 실행할 때 사용하는 인터페이스로, 사용자와 커널 사이에서 명령을 해석하여 전달하는 기능을 한다. 구현하는 쉘 프로그램은 기본적으로 리눅스 쉘과 동일하게 작동해야 한다. 해당 프로젝트를 통해 리눅스 쉘의 기본 동작 원리를 이해하며, 수업시간에 배운 process control, inter process communication(pipe), process signaling, background 등을 응용하고 친숙해지는 것을 목표로 한다.

세부적인 목표로는 3가지로 나눌 수 있다. 첫 번째 phase에서는 fork를 이용하여 child process를 생성하고, 이를 이용하여 쉘을 작성한다. 두 번째 phase에서는 pipe 기능을 추가하여 pipe line(|)을 기준으로 한 프로세스의 출력을 다른 프로세스의 입력으로 전달 시킨다. 마지막 phase에서는 프로세스를 백그라운드에서도 실행시킬 수 있도록 하는 것을 목표로 한다.

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

리눅스 쉘과 동일한 명령을 작업할 수 있도록 프로세스를 구성한다. 해당 쉘은 ls, mkdir, rmdir, cd, touch, cat, echo과 같은 리눅스 기본 명령어를 수행할 수 있어야 한다. exit을 누르면 모든 child process를 종료하고 해당 쉘을 나가야 한다.

echo 명령어를 사용할 때 따옴표를 사용하면, 그 따옴표가 제거되고 출력되어야 하며, 따옴표 없을 때에는 띄어쓰기를 무시하지만, 따옴표 안에 있는 띄어쓰기는 출력에 반영되어야 한다.

Myshell은 exit 명령어에 의해서만 종료되므로 ctrl-c, ctrl-z를 통해 보내지는 신호는 무시되어야 한다.

1. Phase 2

Phase1의 연장선 상에서 pipe 기능을 추가한다. 각 파이프라인마다 새로운 프로세스를 생성하고 이전 command가 wait 후 reaping을 수행한다. 한 프로세스의 output이 한 프로세스의 input이 되도록 연결해야 하며, 파이프 수가 2개 이상이더라도 구현되어야 한다. 파이프와 명령어 사이에는 공백이 있을 수도 있고 없을 수도 있는데, 이와 상관없이 처리 가능해야 한다.

1. Phase 3

Phase2에서 백그라운드 기능이 추가되었다. 지금까지 수행 가능한 모든 명령어 뒤에 &를 붙이면 백그라운드 실행으로 넘어가게 된다. 또한 job기능을 수행해야 하는데, 이는 백그라운드 상태의 job들을 출력하는 기능이다. 또한 bg job을 하면 stop 백그라운드상태가 running 백그라운드 상태로 변하고, fg job하면 백그라운드 상태가 running foreground 상태로 변한다. 마지막으로 kill job하면 job 기능을 끝낸다.

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

fork함수는 호출한 프로세스를 복사하는 기능을 하며, 이를 통해 부모와 자식 프로세스가 생성된다. 이 두 프로세스는 같은 코드를 수행하나 독립적인 공간에서 실행된다. 자식 프로세스의 경우 함수 실행이 실패하면 -1, 성공적으로 실행되면 0이 반환된다. 본 프로젝트에서는 이러한 fork함수와 에러 처리를 포함하고 있는 Fork함수를 csapp.h으로부터 가져와서 사용하였다. 무사히 child process가 생성될 경우, execvp함수를 실행하여 프로그램을 실행하였다. execvp()는 unistd.h에 있으며 PATH에 등록되어 있는 디렉토리에 있는 프로그램을 실행하는 함수이다. 이 함수는 해당 프로세스(자식 프로세스)에 덮어씌어지는 형식으로 실행된다. 실패시 -1를 반환하므로, 만약 실패하면 에러 메세지를 출력한 후 exit을 하도록 처리하였다.

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

zombie 상태를 방지하기 위해서는 자식 프로세스가 종료되었을 때, 부모 프로세스가 자식프로세스로부터 SIGNAL을 받아, reaping을 수행하여야 한다. 이를 위해 필요한 함수는 waitpid 함수이다. 자식 프로세스가 종료되면 커널에서 부모 프로세스한테 SIGCHLD 시그널을 보낸다. 이를 이용하여 signal handling을 하는데, 부모 프로세스는 SIGCHLD를 받아 waitpid함수를 호출한다. 성공시 자식 프로세스의 ID를 실패시 -1를 리턴한다. 성공할 경우 정상적으로 자식 프로세스가 종료된다.

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

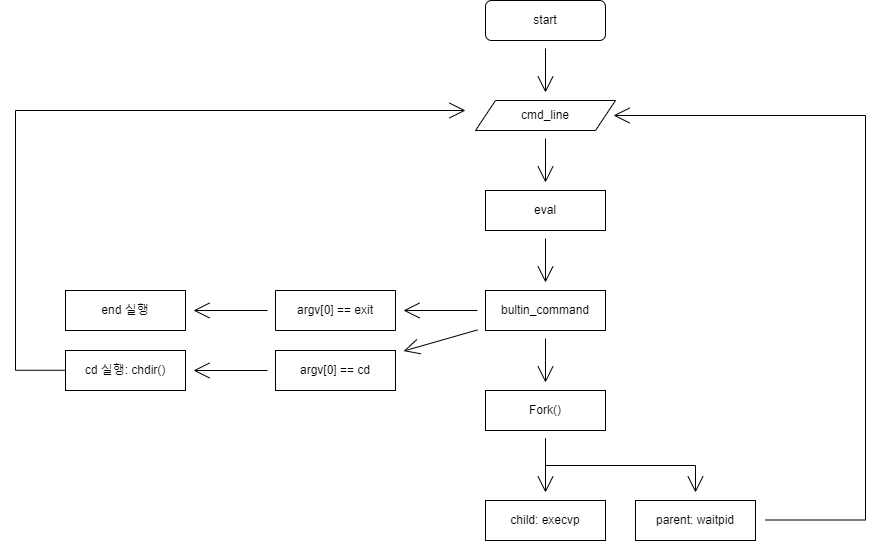
우선, parseline 함수에서 pipeline이 있는 경우를 공백으로 바꿔준 후, 해당 인덱스를 미리 선언해둔 자료구조를 통해 표시해둔다. 그러한 다음 pipe가 존재한다는 flag를 1로 바꾸어주고, 관련 함수를 실행한다. 관련 함수는 두가지가 있는데, token함수에서는 미리 선언해둔 인덱스를 기준으로 문자열을 2개로 나누고, exec함수에서는 나눈 두 문자열을 기준으로 inter process를 수행한다. 함수 안에서는 pipe() 함수를 통해 파이프를 생성한다. 그러한 다음 fd[2]를 선언하여 0은 input, 1은 output으로 이용한다. 각 프로세스 안에서 dup2 함수를 실행하여 한 프로세스에서는 입력만, 한 프로세스에서는 출력만 하도록 나머지는 닫아주도록 한다.

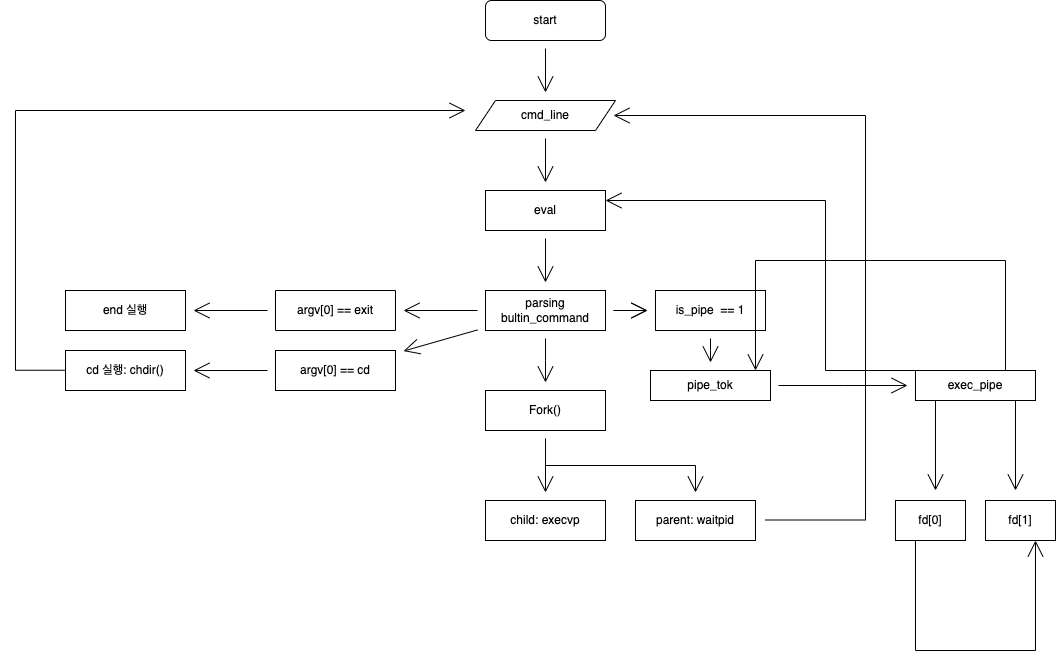
* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Pipeline 기준으로 두 문자열을 받아오는 함수와 해당 pipeline을 처리하는 함수를 분류하였다. Pipeline 개수가 여러 개더라도 연속해서 handling하기 위해 재귀적 방식을 사용하였다. Pipeline을 처리하는 함수에서 두 문자열을 받았을 때, 두번째 문자열이 NULL 형태가 될 때까지 반복하여 동일 함수를 호출하였다.

* 1. **개발 방법**

Base code로 주어진 csapp.h, csapp.c, shellex.c에서 필요한 부분을 가져와서 myshell.c, myshell.h라는 파일명으로 개발하였다.

1. Phase1: 주어진 코드 중 함수 eval에서는 프로그램을 실행시키기 위해서 execve를 사용했는데, execve 첫 번째 인자에는 PATH 경로가 모두 들어가야 하기 때문에 명령어를 입력할 때 경로까지 입력하지 않으면 인식되지 않는다. 따라서 기본적으로 PATH 경로를 가정하고 함수를 실행시키는 execvp로 변경하여 구현하였다. 또한 부모 프로세스에서는 자식 프로세스가 끝날 때까지 waiting을 하고 알맞게 reaping해주어야 하기 때문에 waitpid함수를 추가해주었다. 그리고 cd는 예외적인 상황이므로 builtin\_command에서 따로 chdir을 호출하는 형식으로 추가해주었다. 또한 기존 코드에서는 따옴표에 대한 처리가 되어있지 않았기 때문에, parsing해주는 함수인 parseline을 수정하였다. 만약 따옴표가 있으면 strchr로 문자열 검색을 하여 다음 따옴표가 있는 주소를 찾고, 그 사이 문자열을 한꺼번에 한 배열 안에 넣는 방식으로 parsing하여 따옴표가 알맞게 처리될 수 있도록 하였다.
2. Phase2: 우선, 인자를 parsing하는 부분에서 만약 | 문자열이 있었다면 flag를 사용하여 표시하였다. 그리고 pipe\_arr라는 배열을 선언하여, | 문자열이 있을 때의 토큰의 위치를 표시하였다. evaluate 함수에서 flag가 1인 경우 pipe\_tok함수와 exec\_pipe 함수를 실행하였는데, pipe\_tok함수는 parsing한 argv를 가져와 미리 저장해둔 배열 인덱스를 바탕으로 두 문자열을 나누었다. 그리고 exec\_pipe 함수는 두 문자열을 가져와 pipe()함수를 사용하여 pipe를 생성한 뒤, 각각 한 프로세스에서는 output하는 역할을, 다른 프로세스에서는 read하는 역할을 담당하여 pipe를 구현하였다. 구현한 이후 미리 설정해둔 배열 인덱스가 눌 값인지 확인하고, 그렇지 않다면 재귀적으로 호출하여 다음 파이프도 수행한다. 눌 반환이 된다면 마지막 파이프 부분을 수행하고, 다시 evaluate함수로 돌아와 waitpid를 수행한다. 재귀적으로 호출하는 부분에 있어 인덱스를 처리하기가 까다로웠는데, pipe\_tok함수에서 현재 배열 인덱스에서 그전 인덱스를 빼는 식으로 할당하여 이를 해결하였다.
3. **구현 결과**
   1. **Flow Chart**
4. **Phase 1 (fork)** 
5. **Phase 2 (pipeline)**

****