**Multicore Programming Project 1**

담당 교수 : 최재승

이름 : 이우진

학번 : 20181669

1. **개발 목표**

본 프로젝트에서는 리눅스 혹은 유닉스 운영체제에서 사용되는 shell 프로그램을 C언어를 이용하여 개발한다.   
 Shell 프로그램은 다음과 같이 동작한다. 먼저, 프로그램을 실행하고 난 이후 사용자로부터 명령어 입력을 받기 위해 대기한다. 사용자가 명령어를 입력하고 엔터 키를 입력하면 해당 문자열을 적절하게 파싱 하여 대응하는 기능이 존재한다면 해당 기능을 실행한다. 실행이 종료하게 되면 사용자로부터 다음 명령어를 입력 받기 위해 대기한다. Shell 프로그램은 위 과정을 반복하면서, 사용자가 종료 명령어를 입력할 때 까지 실행된다.  
 이 프로젝트에서도 위와 같은 과정을 통해 실행되는 myShell 프로그램을 개발한다. 이를 위해 문자열 입력, 파싱, 명령어 실행 등의 과정을 구현하며 shell 프로그램에서 사용되는 많은 명령어들이 실행될 수 있도록 개발한다.

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

Phase 1에서는 입력 대기, 문자열 파싱, 명령어 실행의 세 가지 동작을 하는 shell의 기본적인 기능을 구현한다. 입력을 받았을 때 적절한 입력인지 확인하고, “…”와 같이 특수한 문자들을 고려하여 적절하게 문자열을 파싱한다. 파싱된 문자열을 이용하여 리눅스의 기본적인 명령어들을 실행할 수 있도록 구현하는데 이에는 *cd*, *ls*, *mkdir*, *rmdir*, *touch*, *cat*, *echo*, *history*, *exit*이 포함된다. 명령어의 실행 결과는 실제 리눅스 shell의 실행 결과와 동일하도록 구현한다. 이 때, 리눅스에서 프로그램으로 제공하는 명령어들은 해당하는 프로그램을 실행하여 구현하고 그 외에는 직접 구현한다.

1. Phase 2

Phase 2에서는 shell에서의 pipeline 기능을 구현한다. Pipeline은 “statement1 | statement2 | …”와 같이 입력되는데, 앞서 입력한 명령어의 결과값이 그 뒤 명령어의 입력값으로 사용된다. Linux shell에서 구현된 바와 같이 myshell에서도 여러 개의 pipeline을 이용하여 recursive하게 실행할 수 있도록 구현한다.

1. Phase 3

Phase 3에서는 shell에서의 background process기능을 구현한다. 이를 위해 이전에는 foreground process만 있었던 명령어 실행 방법을 foreground와 background 실행의 두 가지 방법 모두 실행 가능 하도록 구현한다. 사용자는 “statement &”, “statement&” 와 같은 형태로 명령어를 background process로 실행할 수 있다. Shell에서 프로세스가 종료될 때까지 대기하고 있던 foreground 프로세스와는 달리, background process는 프로세스가 실행되고 있더라도 사용자가 다른 명령어를 입력할 수 있다. Background process를 관리하기 위한 명령어 jobs, bg, fg, kill 또한 사용 가능한 명령어에 추가하고, SIGINT, SIGTSTP와 같은 시그널 관리를 추가한다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

리눅스에서 프로그램 형태로 제공되는 명령어들은 *fork()*와 *execve()* 함수를 이용하여 구현한다. Fork()를 호출하면 현재 실행되고 있는 프로세스의 메모리 공간, 레지스터 등이 그대로 복사되어 새로운 child process가 만들어진다. 이 때, fork()의 반환값이 0일 때는 child process, 0이 아닐 때는 parent process를 의미하기 때문에 parent process에서는 호출한 child process의 reap를 위해 waitpid()함수를 이용하여 잠시 대기시키고 child process에서는 execve() 함수를 호출하여 리눅스 프로그램 파일을 실행시킨다. Child process는 execve()함수를 이용하여 호출된 프로그램으로 대체되는데, 해당 프로그램이 종료되면 parent process에 SIGCHLD 시그널을 보내며 대기하고 있던 parent process는 child process를 reap 시켜준다.

Phase1에서는 하나의 child process를 foreground process로 실행하기 때문에 waitpid()함수를 이용하여 child process가 종료될 때까지 parent process가 대기하면 되지만, 여러 개의 process가 동시에 수행되거나, background process가 존재할 때에는 SIGCHLD 시그널을 이용하여 해당 child process를 제대로 reap 시켜줄 필요가 있다. C에서는 sigaction()함수를 이용하여 signal handler를 등록할 수 있는데, 이 때 안정적인 프로그램을 위하여 csapp.h 헤더의 wrapper 함수인 Signal 함수를 사용한다. 등록한 signal handler 함수에서는 SIGCHLD 함수가 parent에 전달되면 child를 reap시켜주는 작업을 수행하는데 이 때 여러 개의 child가 reap되기를 기다리고 있을 수 있기 때문에 while문을 이용하여 가능한 모든 child를 reap 시켜준다.

* **Phase2 (pipelining)** 
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

프로그램의 개발에서 pipeline은 pipe(), dup2() 함수를 이용하여 구현한다. Pipe()함수는 두 개의 새로운 file descriptor을 생성하는데 이 두 file descriptor는 출력과 입력의 기능을 담당하며 서로 연결되어 있다. 이 file descriptor를 dup2()함수를 이용하여 stdin, stdout에 해당하는 file descriptor위치에 복사해주고 해당 위치에 출력을 하면, terminal에 출력되는 것이 아닌 해당 file에 출력이 되고, 그 출력을 읽어들일 수 있는 상태가 된다. Myshell에서 구현하는 명령어들은 모두 file descriptor 0, 1 위치에서 입력과 출력을 수행하기 때문에 해당 위치에 새로 생성한 file descriptor을 복사하여 해당 file에서 입력과 출력이 이루어지도록 한다.

여러 개의 pipleline을 입력하였을 경우에는, pipeline 개수 만큼의 file을 pipe()를 이용해 생성하여 구현한다. 이 때, 명령어를 파싱한 문자열 배열의 종료가 null문자를 이용하여 표시된다는 것을 이용하여, while 문을 사용해 “|” 가 존재하는 위치에 null문자를 삽입하고 그 전 문자열을 실행하는 과정을 반복한다. 이 때, file descriptor을 close하는 과정을 반드시 수행한다.

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

Background process는 & 문자가 명령어의 마지막에 입력되면 수행하는데, 문자열을 파싱하는 과정에서 입력이 background process로 들어왔는지 확인하여 표시한다. 만약 background process라면, fork()를 이용해 새로운 child process를 생성하여 해당하는 명령어를 수행하도록 한다. Child process가 실행되는 과정은 foreground process가 실행되는 과정과 같다. fg, bg, SIGTSTP 등을 사용해 foreground process와 background process를 변경하는 작업은 c에서 함수로 제공하는 nonlocal jump를 이용하여 구현한다.

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

프로그램의 전체적인 뼈대를 잡기 위해 입력된 문자열을 실행하는 함수 eval(), 문자열을 파싱하는 함수 parseline(), 명령어를 실행하는 함수 builtin\_command()를 추가한다. 이 때, 문자열이 “”, ‘’ 등을 고려하여 제대로 입력되었는지 확인하기 위한 함수 check\_special() 함수를 추가하여 제대로 된 입력이 들어올 때 까지 입력을 받고, 제대로 된 입력을 받았다면 remove\_special을 이용하여 문자열을 적절한 형태로 고쳐준다. 예를 들어, cat “hello 와 같은 입력이 들어왔다면 쌍따옴표가 하나밖에 등장하지 않았기 때문에 입력이 모두 들어오시 않았다고 판단하여 다른 쌍따옴표가 입력될 때까지 입력을 받는다. 만약 cat “hello world”와 같이 제대로된 입력이 들어왔다면 문자열 파싱 할 시에 {cat, hello world} 형태로 파싱이 되어야 하므로 쌍따옴표를 제거하고 띄어쓰기 문자를 보존 처리한다. 이 과정을 하나의 함수로 담아내기 위한 함수 readcmd() 함수를 추가하였다.

프로그램의 실행은 builtin\_command()를 이용하여 이루어지는데 입력된 명령어가 myshell에서 지원하는 명령어인지 확인하기 위해 eval\_command() 함수를 추가하였는데, 이 함수는 명령어의 유형을 나타내는 enum 타입 변수 cmd\_id를 반환한다. 명령어의 유형에 따라 프로그램이 다르게 실행되는데 history, cd와 같은 경우에는 실행 코드가 길어져서 load\_history(), save\_history(), change\_dir() 함수를 추가하였다. 또한, linked\_list로 구현되는 history 명령어를 위한 struct type인 his\_nd를 추가하였다. 마지막으로 history file, linked list등을 free해주기 위한 cleanup() 함수를 추가하여 exit명령어 실행 시에 수행되도록 한다.

* **Phase 2**

파이프라인을 구현하기 위해 추가한 함수는 없지만, pipeline 등과 같은 코드로 인해 eval() 함수의 코드가 너무 길어져서 execute() 함수를 추가하여 입력된 문자열의 실행은 execute() 함수에서 모두 담당하도록 하였다. Execute() 함수에는 pipe()로 생성된 file descriptor를 담기 위한 변수 말고도 stdin, stdout을 위한 임시 변수를 추가하였다. Dup2()로 인해 stdin, stdout이 사라지기 때문에 해당하는 임시 변수에 다른 곳에 복사한 stdin, stdout의 file descriptor를 저장해두고 명령어의 실행이 종료되면 임시 변수를 이용해 다시 표준 입출력으로 되돌린다.

* **Phase 3**

Background process를 추가하면서 SIGCHLD, SIGTSTP, SIGQUIT 등의 signal handler를 추가하여 프로그램이 child process에서 보내는 signal을 관리할 수 있도록 하였다. 또한 jobs 명령어를 위해, his\_nd와 유사한 struct type job\_nd를 추가하고 이를 관리하기 위한 함수 add\_job, change\_job, rmv\_job, print\_job, find\_job을 추가하였다. 마지막으로 background process와 foreground process를 추가하면서 현재 실행하고 있는 process의 상태를 표시하기 위한 flag ischild, for\_pid를 추가하였다.

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

**도표이(가) 표시된 사진

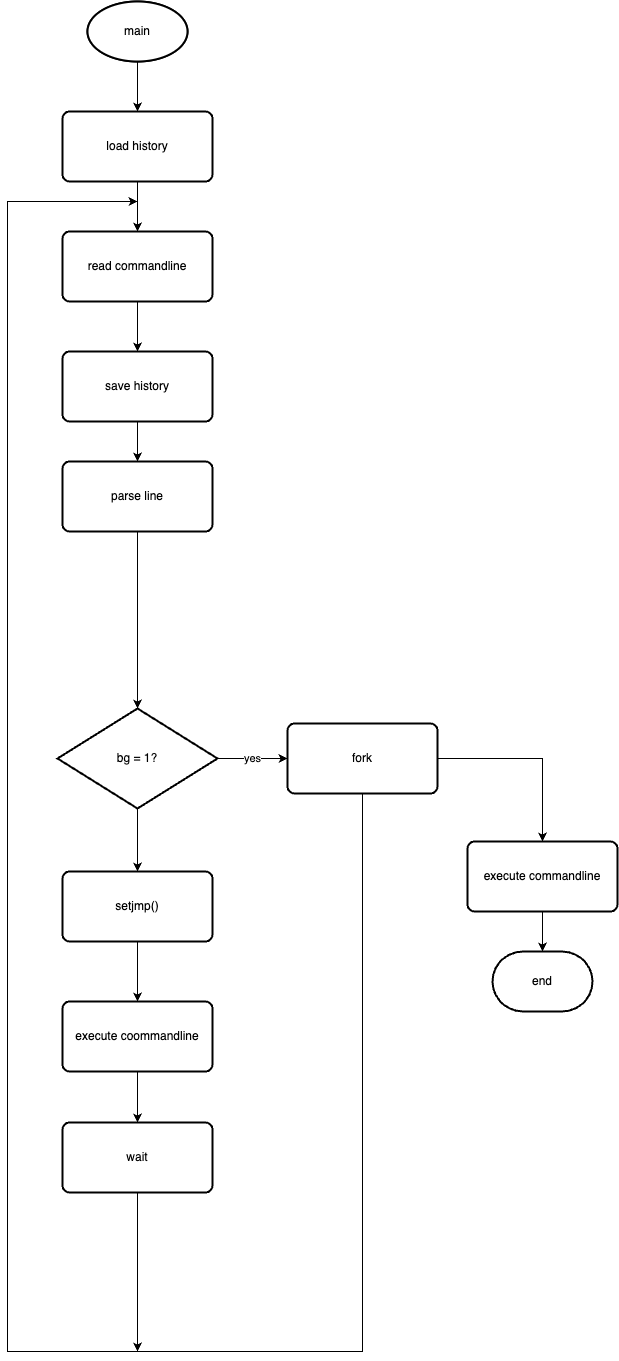
자동 생성된 설명**

1. **Phase 2 (pipeline)**

**도표이(가) 표시된 사진

자동 생성된 설명**

1. **Phase 3 (background)**

****