**Multicore Programming Project 1**

담당 교수 : 박성용 교수님

이름 : 유 시 원

학번 : 20191610

1. **개발 목표**

본 프로젝트는 Unix 환경에서의 Process와 Job의 실행, pipe와 signal을 활용한 프로세스 간 통신에 대한 이해를 바탕으로, system-level process control을 활용하는 Shell을 구현해 보는 것을 목표로 한다. Unix 환경에서의 시스템 컨트롤이 가능한 C언어를 활용해, 리눅스 기본 쉘 bash의 동작을 baseline으로 두는 MyShell을 개발한다.

본 프로젝트는 3개의 Phase로 구성되어 있고, 각 Phase별 개발 목표는 다음과 같다.

* Phase I

Linux Shell의 기본 커맨드가 실행 가능하도록 한다. Basic command인 ‘cd’, ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’, ’cat’, ’echo’, ‘history’, ‘exit’의 명령어가 정상적으로 수행되어야 한다. History의 경우 ‘!!’와 ‘!#’의 명령어가 있는데, ‘!!’는 직전에 수행되었던 명령어를, ‘!#’는 #에 들어오는 숫자에 해당하는 과거 명령어를 실행한다. System call 중 fork()와 exec() 함수를 사용하고, ‘ls’, ‘cat’ 등의 프로그램으로 구현되어 있는 명령어의 경우 fork()를 통해 자식 프로세스에서 exec()으로 실행한다. 그렇지 않은 명령어의 경우 동작을 직접 구현한다.

* Phase II

Linux Shell의 유용한 기능 중 하나인, Pipe를 구현한다. Pipe는 ‘|’의 기호로 여러 개의 명령어를 잇는다. 앞선 명령어의 출력을 뒤따라오는 명령어의 입력으로 보낸다. 하나의 명령어에 대한 입력과 출력은 모두 터미널에서 수행된다. Pipe 구현을 위해서는, stdin과 stdout으로 연결되어 있던 입/출력을 pipe로 변경해 준다. 이를 위해 dup()과 dup2() system call을 활용한다. 구현되는 Pipe는 연결하는 명령어의 개수 제한이 없고, 마지막 명령어에 대한 출력은 stdout을 통해 터미널에 출력한다.

* Phase III

Linux Shell의 핵심 기능 중 하나인, job control을 구현한다. 수행하려는 작업을 foreground 혹은 background에서 수행하고, background에서 수행 중이라면 동시에 새로운 커맨드를 입력해 동작 가능하도록 한다. Background job은 Ctrl-C, Ctrl-Z 등의 키보드 입력이 수행되지 않아야 한다. Foreground job은 수행되는 동안 터미널에 새로운 커맨드 입력이 불가능하지만, Ctrl-C, Ctrl-Z 등의 키보드 입력이 수행 가능하다. Ctrl-Z가 수행된 job은 background로 바뀌며, 수행이 일시 정지된다. Ctrl-C가 수행된 job은 종료된다. 실행 중인 쉘 프로그램은, 키보드 입력으로 인해 종료되거나 멈추지 않는다. ‘jobs’ 명령을 수행하면, 실행 중인 background job들의 상태를 볼 수 있고, ‘fg’를 통해 일시 정지된 job의 재실행 및 foreground로 전환이 가능하다. ‘bg’를 통해 foreground로 전환하지 않고, background에서 그대로 재실행이 가능하다. ‘kill’을 통해 실행 중인 job을 종료할 수 있다. ‘fg’, ‘bg’, ‘job’ 명령어는 #에 숫자를 넣어 ‘%#’와 함께 사용된다. SIGINT, SIGTSTP, SIGCONT 등의 시그널을 보내 위와 같이 동작할 수 있도록 구현한다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

위 캡처와 같이 구현한 모든 명령어가 올바르게 수행되는 것을 확인할 수 있다. ‘cd’, ‘exit’, ‘history’, ‘!!’, ‘!#’의 명령어는 fork() 없이 부모 프로세스에서 바로 수행되는 builtin\_ command로 구현했고, 나머지 명령어는 fork()와 execvp(), waitpid()로 자식 프로세스에서 수행되고 종료되면 부모 프로세스에서 reaping되도록 구현했다.

1. Phase 2

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

위 캡처와 같이, pipeline이 올바르게 동작하는 것을 확인할 수 있다. Pipeline이 여러 개인 경우에도, 각각의 command에 따른 출력을 다음 command의 입력으로 넘긴다. 마지막 command의 출력을 STDOUT으로 출력해 터미널에 결과가 표시된다. 앞선 Phase I에서 구현했던 모든 명령어가 정상적으로 사용되고, 따옴표(“) 처리도 완료되어 pipeline에 따라 원하는 출력이 정상적으로 나오도록 구현했다.

1. Phase 3

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

위 캡처와 같이, 모든 요구되는 foreground와 background 동작에 대한 job control을 구현 완료했다. ‘jobs’, ‘fg’, ‘bg’, ‘kill’에 따른 기대 동작이 올바르게 수행된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Specification 상 명시된 pipeline에 대한 background job 테스트를 위와 같이 수행했다. ‘a.out’ 파일은 위와 같은 간단한 소스 코드로 구현된, 5초동안 수행되며 1초마다 한번씩 print하는 프로그램이다. ./a.out | grep The & 커맨드 실행 시 Foreground에서는 다른 명령어를 사용할 수 있고, 5초 후 위 job이 끝나면 터미널에 결과가 표시된다.

텍스트이(가) 표시된 사진

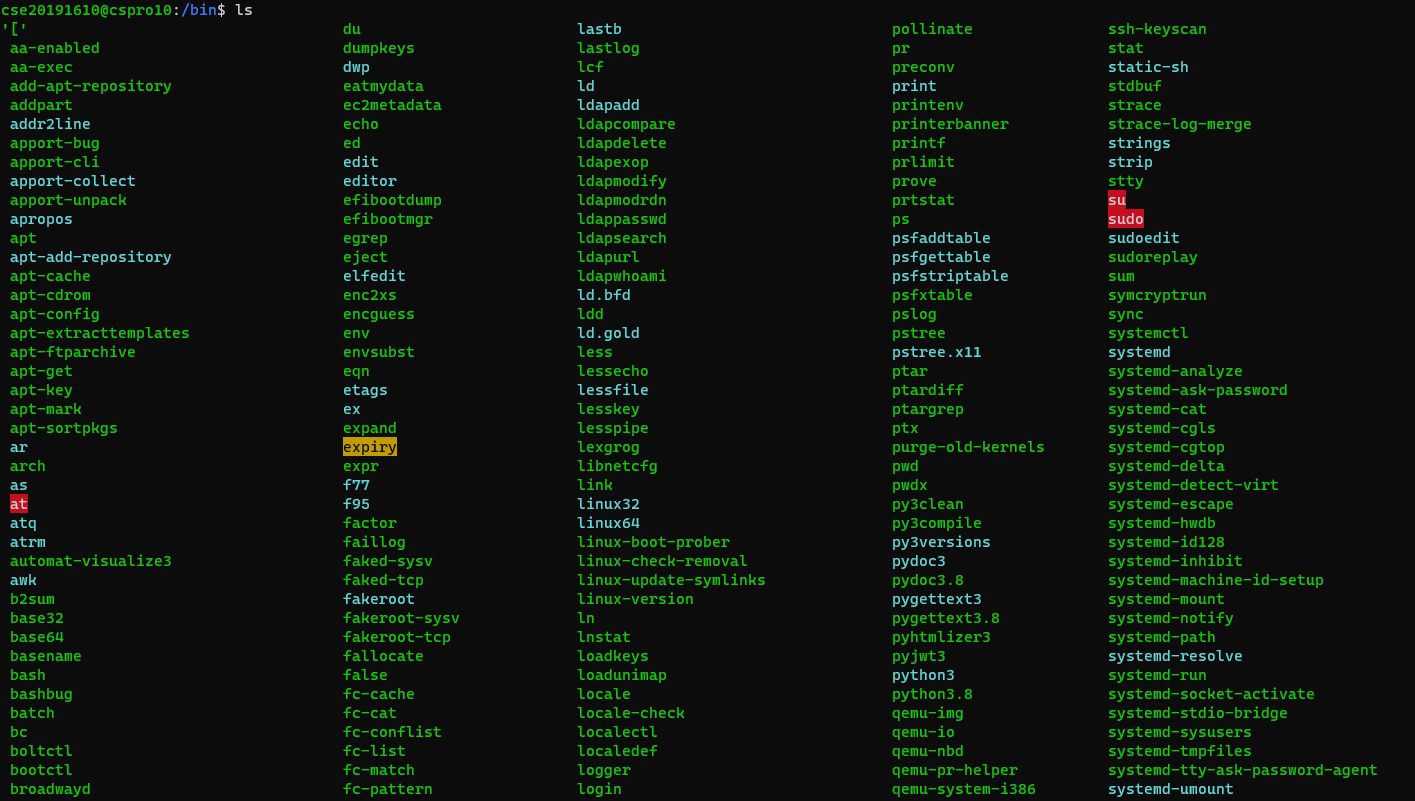
자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

‘bg’와 ‘fg’를 통해 job을 Background와 Foreground로 자유롭게 이동시킬 수 있고, Ctrl-Z와 Ctrl-C를 전달함에 따라 job이 올바르게 멈추거나 종료되는 것을 확인할 수 있다. ‘ps’ 명령어를 통해 프로세스가 실제로 종료된 것을 확인할 수 있다.

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

‘fork()’ 함수를 호출하면, 부모 프로세스의 메모리를 그대로 복사한 자식 프로세스가 하나 더 생성된다. 본 프로젝트의 ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’ 등의 명령어는 이 자식 프로세스에서 수행된다. 위와 같은 명령어들은 기본적으로 구현이 완료된 하나의 프로그램으로, 리눅스의 /bin 디렉토리 혹은 /usr/bin 디렉토리 등의 경로에 저장되어 있다.



쉘에서 명령어를 입력했을 때 해당하는 프로그램을 실행하기 위해서는, exec 명령어를 사용해 프로그램을 실행해야 한다. Exec 명령어는 현재 프로세스의 메모리 공간에, 파라미터로 전달되는 경로의 프로그램을 로드하고, 현재 프로세스에서 실행할 수 있도록 한다. 쉘의 부모 프로세스에서 바로 exec 명령어를 실행하게 되면 실행되고 있던 쉘이 덮어써지기 때문에, 프로세스를 하나 새로 생성하고 이 자식 프로세스를 덮어써 실행하도록 구현한다. Exec 명령어는 execve(), execvp() 등 다양한 종류가 있다. 기본적으로 프로그램을 실행하는 것으로 그 역할은 비슷하지만, execve()는 직접 경로를 입력해 줘야 하고, execvp()는 파라미터로 전달되는 프로그램의 이름을 전달하면 직접 경로를 찾아 실행한다. ‘./a.out’으로 프로그램을 실행하거나, ‘vi’ 명령어로 편집기를 여는 등, 다양한 경로에 흩어져 있는 프로그램을 쉘에서 자유롭게 사용할 수 있도록 하기 위해 본 프로젝트에서는 execvp() 함수를 사용했다.

프로그램의 main() 함수에서 fgets()로 command line을 받아오면, eval() 함수가 실행되는데, 이 안에서 현재 입력된 command에 대한 파싱을 수행한다. 이후 built-in 명령어가 아니라면 구현되어 있는 프로그램이므로, 이 프로그램을 수행하기 위해 fork()해 자식을 생성하고 execvp()로 프로그램을 실행한다. 만약 실행에 실패해 음수 값을 리턴했다면, 에러 메시지를 출력하고 자식 프로세스를 종료한다.

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

자식 프로세스에서 프로그램 실행이 끝나면, connection을 종료해야 한다. ‘ls’ 명령어를 수행했다고 가정하자. 자식 프로세스에서 execvp()로 ‘ls’ 프로그램이 위치한 경로를 찾고, 수행한다. ‘ls’ 프로그램이 실행되어 터미널에 현 경로의 파일 정보를 출력하고, 프로그램이 끝날 것이다. 이렇게 끝난 프로그램은, 자식 프로세스를 fork()했던 부모 프로세스가 reaping하기 전까지 좀비 프로세스 상태가 되며, 사라지지 않는다. 자식 프로세스의 수행이 완료된 후 좀비 프로세스가 된 자식 프로세스를 종료하기 위해, 부모 프로세스에서는 reaping을 수행해 주어야 한다. 이 과정은 wait(), waitpid() 등의 함수를 통해 진행된다.

프로그램 상에서 Fork() 함수를 실행하는 순간, 메모리가 복사되고 자식 프로세스가 생긴다. 부모 프로세스와 자식 프로세스는 Fork() 실행 이후부터 각자 그 아래의 코드를 실행한다. 이 때 부모 프로세스에서는 Fork()에 대한 리턴 값이 생성된 자식 프로세스의 PID가 되고, 자식 프로세스에서는 0이 된다. 따라서 if (Fork() ==0 ) {} else {}와 같은 구문으로 자식 프로세스에서만 수행되거나, 부모 프로세스에서만 수행될 구문을 지정할 수 있다. 혹은 자식 프로세스만 수행되는 if 구문 내에서 exec 함수가 실행되거나, exit 함수가 실행되면 자식 프로세스는 그 이후의 코드를 실행하지 못하기 때문에, 그 아래 부분에 부모 프로세스에서만 실행할 코드 내용을 작성할 수도 있다. 자식 프로세스에 대한 reaping은 부모 프로세스에서만 일어나야 하기 때문에, 언급한 부분에서 wait(), 혹은 waitpid() 함수를 호출해 자식 프로세스에 대한 reaping을 수행할 수 있다.

자식 프로세스는 멈추거나 종료될 때 부모 프로세스에 SIGCHLD 시그널을 전송한다. 부모 프로세스는 wait(), 혹은 waitpid() 함수가 실행 중, 자식 프로세스로부터 시그널이 도달할 때까지 프로세스를 블로킹한다. 이 때 블로킹은, OS의 리소스를 사용하지 않은 채 시그널이 발생할 때까지 프로세스를 멈추는 것이다. 자식 프로세스가 수행이 완료되어 terminate될 때, 즉 execvp() 로 실행된 프로그램이 종료되었거나 자식 프로세스에서 exit했을 때 SIGCHLD가 부모 프로세스에게 전달되고 wait 상태 또한 끝이 난다.

본 프로그램에서는 waitpid(pid, &status, 0)을 통해 생성한 자식 프로세스가 종료될 때까지 명시적으로 기다려 주었다.

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

Pipeline이 입력되는 경우는 여러 개의 명령어가 입력되고, 그 사이가 ‘|’로 이어져 있다. 따라서, 입력되는 command line에 ‘|’가 있다면 pipeline에 따른 파싱 및 로직 수행을 달리 해야 하고, 그렇지 않다면 기존과 동일하게 로직을 수행하면 된다. 따라서, main() 함수에서 command line을 입력 받고 실행하기 전에 strchr() 함수를 사용하여, 입력에 ‘|’가 있는지 확인한다. 이에 따라 분기되어 eval()을 수행하거나, eval\_pipeline()을 수행한다.

eval\_pipeline() 함수의 경우, 커맨드를 파싱해 각 명령어에 대해 eval() 함수를 실행시킨다. pipeline이 적용되었을 때는, 앞선 명령어에 대한 출력을 뒤따르는 명령어에 대한 입력으로 넘겨 주어야 한다. 이를 구현하기 위해 pipe(), dup2() 함수를 사용했다. pipe() 함수를 통해 파일 디스크립터를 연결하는 파이프를 구성하고, 파이프로 구분된 명령어들이 어떤 입력과 출력을 받아야 하는지 경우의 수를 구분해 그에 맞는 eval() 함수를 호출한다.

호출된 각 eval() 함수 내에서는, fork()된 자식 프로세스가 수행될 때에, 경우에 따라 STDIN 혹은 파일 디스크립터에서 입력을 받고, STDOUT 혹은 파일 디스크립터로 출력할 수 있도록 dup2() 함수를 통해 입출력 파일 디스크립터를 변경한다. Pipe를 사용해 프로세스 간 입출력을 전송할 때에는 입력 디스크립터가 닫혀 있는 쪽으로 출력하고, 출력 디스크립터가 닫혀 있는 쪽으로 입력을 받기 때문에 자식 프로세스에서는 사용하지 않는 파일 디스크립터를 close해 주어야 올바르게 데이터를 입력, 출력 받을 수 있다.

파일 디스크립터는 fd1, fd2 두 개를 사용했는데, 각각 크기 2를 갖는 정수형 배열이고, pipe(fd1);과 같은 형식으로 파일 디스크립터의 read-end와 write-end를 파이프라인으로 연결한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Pipeline의 개수에 따라 명령어는 최소 2개부터, 최대 개수는 제한 없이 연결될 수 있다. 파이프라인이 명령어 2개를 연결하는 경우, 가령 A | B 와 같은 구조로 명령어가 연결되는 경우가 있다고 하자. A 명령어는 표준 입력 STDIN으로부터 입력을 받아 출력을 Pipeline의 write-end로 출력한다. B 명령어는 Pipeline의 read-end로부터 입력을 받아 출력을 표준 출력 STDOUT으로 출력한다.

파이프라인이 명령어 3개를 연결하는 경우, 가령 A | B | C와 같은 구조로 명령어가 연결되어 있는 경우가 있다고 하자. 명령어에 대한 입력, 출력이 일어나는 파일 디스크립터를 {입력 - 출력}과 같이 표기하면, A 명령어는 위와 동일하게 *{STDIN – 파일 디스크립터1}*이 된다. B 명령어는 *{파일 디스크립터1 – 파일 디스크립터2}*가 된다. C 명령어는 *{파일 디스크립터2 – STDOUT}*이 된다. 명령어 3개를 연결할 때, A, C는 위 명령어 두 개를 연결할 때의 A, B와 동일한 파일 입출력 방식을 사용한다. 파이프라인의 개수가 3개 이상으로 많아지는 경우, 가운데 명령어들은 B에 해당하는 *{파일 디스크립터1 – 파일 디스크립터2}*와 같은 형식으로 입출력을 수행한다. 이렇게 두 개의 파일 디스크립터를 적절히 사용해 입출력을 수행하면, 아무리 많은 개수의 명령어를 연결해도 이전 명령어의 출력 결과를 다음 명령어의 입력으로 넘겨줄 수 있다.

또, *{파일 디스크립터1 – 파일 디스크립터2}*의 경우 입력 받는 파일 디스크립터와 출력하는 파일 디스크립터가 각각 다른 것을 사용한다. 이는 입력을 받는 파이프라인과 출력해야 하는 파이프라인이 다르기 때문이고, 이와 같이 파이프라인 간 데이터를 옮겨 주기 위해 두 개의 파일 디스크립터만을 사용해서 구현이 가능하다. 위 항목에서 언급한 fd1, fd2를 이런 식으로 사용해 Pipeline 개수에 따른 입출력을 handling했다.

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

자식 프로세스에서 exec되어 실행되는 각 job들은, 부모 프로세스에서 fork되었기 때문에 부모 프로세스와 동일한 프로세스 그룹에 속해 있다. Background에서 job을 수행시키기 위해서는, Ctrl-C / Z 등의 터미널을 통한 키보드 입력이 막혀야 하고, 또 터미널을 바로 사용할 수 있도록 부모 프로세스에서 wait가 수행되지 않아야 한다. 키보드 입력을 막기 위해 SIGINT, SIGTSTP 시그널을 Block할 수도 있지만, 이 방법을 사용하는 경우 추후 exec되어 실행되고 있는 Background 프로세스를 Foreground로 가져와도 Ctrl-C / Z에 따른 시그널을 받지 못하는 이슈가 발생한다. SIGKILL, SIGSTOP을 대신 사용하는 방법도 있지만, 실제 쉘과 동일하게 구현할 수 있도록 **새로 생성되는 프로세스에 대해 프로세스 그룹을 분리하고, 터미널 제어권을 이동하는 방식**으로 Phase 3을 구현했다. 터미널 제어권은 tcsetpgrp() 함수를 사용해 변경했고, 그 과정에서 백그라운드 프로세스에 의해 포그라운드 프로세스가 블로킹 상태에 빠지지 않도록 SGITTOU 시그널을 Block했다.

Ctrl-C / Z 등을 터미널에 입력하면, 현재 터미널 제어권을 갖고 있는 프로세스 그룹 내 모든 프로세스에 시그널이 전달된다. 따라서 현재 터미널 제어권은 늘 Foreground에 있는 프로세스 그룹이 가지고 있어야 한다. 아무런 프로세스도 실행하지 않았을 경우, 실행된 myshell이 Foreground 프로세스이다. 여기서 새 프로세스를 Foreground로 실행시키면, 해당 프로세스가 Foreground 프로세스가 되고, 터미널 제어권을 갖도록 한다. Foreground 프로세스가 실행되는 동안은, myshell은 종료될 때까지 기다린다. 본 프로젝트에서는 Sigsuspend() 함수를 사용해 시그널이 발생할 때까지 기다리도록 했다. 이후 해당 Foreground 프로세스가 종료되면, 터미널 제어권을 myshell의 프로세스 그룹으로 돌려받도록 한다. Foreground 프로세스가 실행 중 Ctrl-C / Z 등의 Interrupt가 발생하면, 이 프로세스는 부모 프로세스인 myshell과 다른 프로세스 그룹에 속해 있으므로 Foreground 프로세스 그룹에만 시그널이 전달된다. 따라서 실행 중인 프로세스에 SIGINT / SIGTSTP가 발생하고, 종료 혹은 정지된다. 이 프로세스가 멈췄을 경우 부모 프로세스로 SIGCHLD가 전달되므로, 부모 프로세스에서는 상황에 맞게 적절히 reaping 등의 작업을 수행한다.

Background 프로세스의 경우, Foreground 프로세스와 동일하게 독립된 프로세스 그룹으로 분리하지만, 터미널 제어권을 부여하지는 않는다. 이를 통해 백그라운드 프로세스는 다른 프로세스 그룹에서 실행됨과 동시에 Foreground의 myshell에서 터미널을 사용해 다른 커맨드를 입력할 수 있다. 마찬가지로 Background의 프로세스가 종료되면, 부모 프로세스인 myshell의 프로세스에 SIGCHLD가 발생하므로, 상황에 맞게 reaping 등을 수행한다. Background에서 실행 중인 프로세스를 Foreground로 가져오기 위해서는, SIGCONT 등의 시그널을 보내 정지된 프로세스를 재실행하고, Foreground 프로세스에서 했던 것과 같이 Sigsuspend() 함수를 사용해 시그널이 발생할 때까지 기다린다.

* 1. **개발 방법**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

본 프로젝트 구현을 위해, 위와 같은 구조체 및 전역 변수를 사용했다. Phase 3 구현 시 실행되는 각 프로세스를 관리하는 구조체 ‘job’을 선언했다. 각 실행되는 프로세스마다 job에 대한 정보를 저장하고, 각 job을 원소로 갖는 구조체 배열 형태로 사용해 모든 프로세스에 대한 정보를 관리한다. 구조체 내의 ‘status’ 변수는 종료, 실행 중, 정지를 나타내고, ‘pid’에는 프로세스의 pid를 나타낸다. ‘bg\_job\_idx’ 변수는 해당 job이 백그라운드로 돌고 있으면 갖는 고유한 index를 나타낸다. ‘bg %1’, ‘kill %2’, ‘fg %3’과 같은 명령어 수행 시 % 뒤에 오는 수가 여기서 bg\_job\_idx로 관리된다. 각 job에 대한 실행시킬 때의 command를 job\_name으로 저장한다. Job을 관리하기 위해 total\_jobs, total\_bg\_jobs 변수를 사용했다. 각 job들에 idx를 할당하기 위해 사용하는 변수이다. SIGCHLD 핸들러 동작 시 자식에 대한 reaping을 수행하면 자식을 올바르게 처리했는지 부모 프로세스의 eval() 함수 내에서도 알 수 있도록, reaped\_pid 변수를 업데이트하며 사용한다. Foreground job 종료 후 부모 프로세스에서 터미널의 제어권을 돌려받을 수 있도록, 최상단 부모 프로세스인 myshell의 pgid를 myshell\_pgid 변수에 저장해 사용한다. 이 변수들은 시그널 핸들러 안에서 값이 변경되기도 하기 때문에, 안전한 값 assignment를 위해 volatile sig\_atomic\_t로 선언되었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Phase 1, 2, 3의 구현을 위해 main() 함수와, 시그널 핸들러 함수 sigchld\_handler, sigint\_ handler, sigtstp\_handler을 제외하면 위 캡처에 프로토타입으로 명시된 다섯 개의 함수를 사용하였다. 순서대로 eval() 함수는 각 개별 command를 실행하는 함수로, 필요 여부를 확인해 fork() 및 execvp()를 수행한다. eval() 함수는 주어진 template 코드에서 많이 변경되었는데, Phase 2의 pipeline 구현을 위해 부모와 자식 프로세스에서 올바른 파일 디스크립터로 입/출력할 수 있도록 Dup2() 함수를 사용한다. 또 Phase 3의 background process 구현을 위해 부모 프로세스에서 터미널의 제어권을 부여하고 회수하는 로직, 무시할 시그널을 마스킹하는 로직, 자식 프로세스의 실행 동안 부모 프로세스를 블로킹하는 Sigsuspend 로직 등이 추가되었다. 자식 프로세스에서 프로그램으로 구현되어 있지 않은 ‘jobs’, ‘history’, ‘!!’, ‘!#’ 등의 구현도 추가되었다. Pipeline을 거치려면 입출력이 자식 프로세스에서 수행되어야 올바르게 동작할 수 있으므로 기존 builtin\_command()에서 수행하는 것과 다르게 Fork() 후 자식 프로세스 내에서 실행되고, 입출력이 완료되면 자식 프로세스를 exit(0)으로 종료한다.

eval\_pipeline() 함수는 command line에 ‘|’가 표함되어 있을 경우 파싱을 수행하고, 적절하게 eval() 함수를 호출한다. parseline() 함수는 command line을 파싱하는 함수로, argv에 command line의 인자들을 공백 기준, 따옴표가 있을 경우 따옴표 기준으로도 파싱해 담아 주고, command line의 마지막에 ‘&’이 포함되었는지 확인해 명령의 background 수행 여부를 리턴한다. builtin\_command 함수는 실행하려는 명령이 builtin\_command인지, 아닌지를 확인하고 맞는 경우 해당하는 로직을 수행하는 함수이다. 자식 프로세스에서 수행할 필요 없이 부모 프로세스에서 바로 동작을 수행하는 경우, 예를 들어 쉘을 종료하는 ‘exit’ 혹은 현재 경로를 이동하는 ‘cd’와 같은 명령어들은 부모 프로세스에서 실행해야만 의도하는 결과를 얻을 수 있다. 이 경우가 아니라면 eval() 함수 내에서 fork를 통해 새 프로세스를 실행하게 된다. Linux shell에서는 history를 관리하는 파일에 입력했던 커맨드를 저장하고, 필요한 경우 불러와 사용한다. save\_history() 함수는 history 기록을 위해 입력하는 command를 저장하는 함수이다. 파일에 open해 가장 마지막에 새로 들어온 command line을 작성한다.

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)**

**도표이(가) 표시된 사진

자동 생성된 설명**