**Multicore Programming Project 1**

담당 교수 : 최재승 교수님

이름 : 송경호

학번 : 20191599

1. **개발 목표**

Project 1의 목표는 Linux Shell의 기능을 부분적으로 구현한 myshell을 구현하는 것이다.

총 3개의 phase로 나누어져 개발이 진행되며, 각 각의 phase는 Linux Shell에서의 특정 기능의 구현을 요구한다. 이때, 이전 phase에서 구현된 기능들이 이후의 phase의 기능들과 이어지기 때문에 phase를 거듭해 나가며 점차 발전된 형식의 myshell이 구현된다.

[phase 1] : fork와 exec를 이용 linux shell의 main function을 구현한다.

[phase 2] : pipeline 기능을 구현한다. phase1에서 구현된 main function간의 입력과 출력을 ` | `을 통해 연결할 수 있는 기능을 만든다.

[phase 3] : background에서 명령어를 실행하는 기능을 구현한다. 추가적으로, 프로세스를 background에서 실행 중이거나 멈춰 있던 process를 foreground로 실행하거나 멈춰 있던 background 프로세스를 다시 실행하는 기능을 구현한다. 실행 중인 프로세스들은 항상 관리되고 있어야 하며 그 목록을 확인할 수 있어야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**Phase1 (fork & signal)**

**cd**, **ls**, **mkdir**, **rmdir**, **touch**, **cat**, **echo**, **history**, **exit** 명령어가 linux shell과 동일하게 수행되도록 한다.

아래의 실행 예시는 기본적으로 명세서 따랐으나, execvp() 함수를 사용해 해당 실행 예시에 포함된 명령어 외에도 파일 경로에 존재하는 명령어의 경우 실행된다.

추가적으로, execvp()를 통해 실행되지 않는 **cd**와 **history** 및 **!**관련 명령어들의 기능, 그리고 ` “ `, ` ‘ ` 등을 포함한 전처리가 필요한 명령어들과 관련된 처리가 구현되어 있다.

다음은 출력 예시이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**Phase2 (pipelining)**

phase1에서 구현된 명령어들과 pipeline이 함께 동작하도록 한다. pipeline의 기능은 pipeline 이전 명령어의 output이 이후 명령어의 input으로 입력되게 하는 것이다. 이는 pipe()를 통해 file descriptor들을 잇고, dup2()를 통해 file descriptor들을 표준 입력과 표준 출력으로 대체하며 구현할 수 있다.

또한 한 명령어에 pipeline이 여러 번 들어올 수 있으므로, pipeline을 기준으로 파싱된 여러 명령어를 좌측부터 순서대로 input을 다음 명령어의 output으로 전달하며 실행한다.

이때, 각 명령어들은 phase1과 마찬가지로 fork() 후 child process 내에서 이루어지기 때문에 파이프 명령문을 실행하는 재귀함수 역시 지속적인 child reaping이 필요하다.

다음은 출력 예시이다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

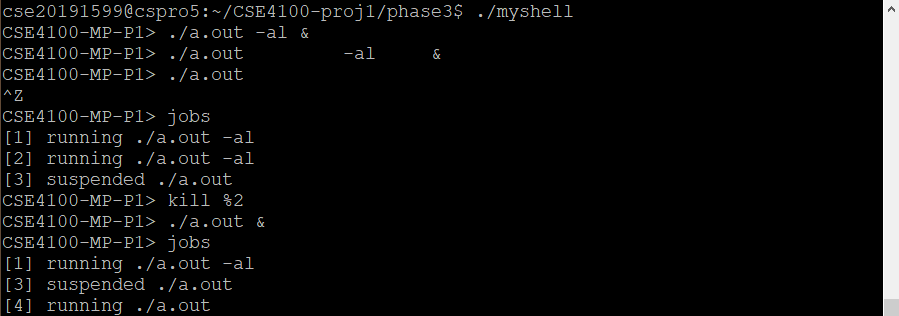
**Phase3 (background process)**

phase1과 phase2에서 구현된 명령어 처리들이 & 기호를 통해 background에서 실행될 수 있도록, 다시 말해 foreground에서 다른 작업을 할 수 있도록 처리한다.

각 각의 프로세스들은 항상 관리되고 있어야 하며, 관리 중인 프로세스들을 확인하는 **job**, 멈춰 있던 background process를 다시 실행시키는 **bg**, background process를 foreground로 옮기는 **fg**, 마지막으로 프로세스들을 종료하는 **kill** 명령어가 구현된다.

또한 실행 중인 foreground 프로세스는 Ctrl+Z 입력에 의해 정지되어 background process로 옮겨져야 한다.

다음은 출력 예시이다.



텍스트이(가) 표시된 사진

자동 생성된 설명

* 1. **개발 내용**

**Phase1 (fork & signal)**

* + **fork를 통해서 child process를 생성하는 부분에 대해서 설명**

명령어를 입력 받은 뒤, 여러 parsing 절차를 거쳐 verify\_cmd() 함수에 parsing이 완료된 command line을 넘겨준다. 해당 명령어가 builtin\_command()에 포함된 **cd**, **history, exit** 명령어의 경우 builtin\_command()에 정의된 각 각의 명령을 수행한다. builtin\_command()에 포함된 명령어가 아닐 경우, fork()를 통해 새로운 child process를 생성하고 해당 child process 안에서 execvp() 명령어를 수행한다.

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

phase1 내에서 builtin\_command()에 포함되지 않아 execvp()를 통해 실행되는 명령어들의 경우 fork()를 통해 child process를 생성하여 execvp()를 실행하기 때문에 reaping을 필요로 한다. 이때 각 child process는 execvp()에 의해 종료되어 SIGCHLD signal을 보내기 때문에, handler함수를 통해 이들을 reaping할 수 있다.

**Phase2 (pipelining)**

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

pipeline을 처리해주기 앞서 먼저 ` ‘ `, 와 ` “ `에 대한 처리, 그리고 명령어와 pipeline이 붙어있는 경우 띄어쓰기를 추가해주는 처리가 선행되어야 한다. 이후에, 전체 명령어를 pipeline 기준으로 나누었을 때 생기는 하나의 명령어를 단위로 실행 및 입출력을 관리해야 하므로, strtok()를 이용해 pipeline을 기준으로 명령어들을 나눈다.

이후에 phase1과 마찬가지로 builtin\_command() 혹은 fork()와 execvp()를 통해 나누어진 명령어들을 실행시켜 준다. 이때, 명령어 실행 전 각 실행의 file descriptor를 pipe()를 통해 연결해주고 상황에 맞게 dup2()를 실행 해 입력과 출력을 조정하여 pipeline을 구현한다.

* + **Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명**

pipeline으로 나누기 전 전체 명령어에 대하여, 최초로 실행되는 명령어, 마지막에 실행되는 명령어 그리고 중간에 실행되는 명령어에 따라 file descriptor를 설정해주는 방식이 다르다.

file descriptor를 각 각 fd[0](입력), fd[1](출력)으로 칭하자.

첫 명령어의 경우 이후 명령어로 그 출력을 넘겨주므로, fd[0]를 닫아주고 dup2(fd[1], STDOUT\_FILENO)를 통해 표준 출력으로 fd[1]을 변경해준다. 이때, 부모 프로세스에서 fd[1]을 다시 닫아준다.

두번째 명령어부터, 마지막 이전 명령어까지는 이전 명령어의 출력을 입력 받고 다음 명령어의 입력으로 출력해야 한다. 따라서, dup2(in\_fd, STDIN\_FILENO)를 통해 이전 출력을 입력 받는다. 이때, in\_fd는 이전 명령어의 입력의 끝을 의미한다. 이후, 첫 명령어와 마찬가지로 fd[0]를 닫아주고 dup2(fd[1], STDOUT\_FILENO)를 수행해, 출력을 다음 명령어의 입력으로 넘겨준다.

마지막 명령어의 경우 두번째부터 마지막 이전 명령어까지, 이전 명령어의 출력을 입력받기 위해 사용한 dup2(fd[1], STDOUT\_FILENO)를 이용한다.

**Phase3 (background process)**

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

& 명령문이 붙지 않은 경우, foreground 프로세스임을 나타낸다. 때문에 무한히 지속되는 코드의 경우 SIGINT 혹은 SIGTSTP signal을 받을 때까지 프로세스가 종료돼서는 안된다. 이를 위해 foreground 프로세스가 실행되고 있음을 나타내는 flag를 통해 main으로 return되는 것을 막는다.

& 명령문이 붙어 있는 경우, background 프로세스임을 나타낸다. foreground와 달리 해당 프로세스는 종료여부와 관계없이 main으로 return되어 다음 작업을 수행할 수 있게 한다.

또한 fg, bg, kill 명령어 및 SIGINT, SIGTSTP, SIGCHLD를 통해 해당 프로세스들을 관리하기 위해 프로세스의 상태와 pid를 담은 구조체를 구현하여 프로세스를 추적한다.

* 1. **개발 방법**

**Phase1 (fork & signal)**

Phase 1의 실행흐름은 다음과 같다.

먼저 main이 실행되면 **ⓐ initFile()**을 호출하여 쉘 프로그램의 초기 설정을 완료한다. 다음으로 무한 반복을 하는 while문 속에서 명령어를 입력받고 **ⓑ historySaver()**에서 해당 명령어를 history.log 파일에 저장한다. 이후 이어지는 main에서 ` ” `와 ` ’ `를 처리하는 과정을 거친다.

따옴표 처리가 완료된 명령어는 **ⓒ verify\_cmd()**에서 parseline()을 통해 실행하기 전 최종형태로 parsing을 완료하며 **ⓓ builtin\_command()**의 return 값에 따라 자식 프로세스를 생성하여 execvp를

Phase 1에 사용된 주요 함수는 다음과 같다.

**ⓐ void initFile()**

main()의 시작과 함께 호출되는 함수이다. 역할은 다음과 같다.

① history.log 생성

myshell.c 파일이 존재하는 경로에 history.log 파일을 생성하고 해당 파일 경로를 전역변수 filesrc에 저장한다. 만약 파일이 이미 존재했다면, 가장 마지막으로 실행한 명령어를 전역변수인 prevcmd에 저장한다.

② handler 함수 연결

phase1과 phase3에서 사용할 handler함수를 각 각의 signal에 연결한다.

**ⓑ void historySaver(char \*buf)**

입력된 명령어인 buf를 initFile()에 의해 생성된 history.log 파일에 저장하는 함수이다.

① 단순 enter 무시

입력된 명령어의 길이가 1인, 다시 말해 하나의 개행문자로 이루어진 명령어는 history.log에 저장하지 않는다.

② `!` 처리

명령어의 첫번째 단어에 `!`가 들어있는지 판단 후, 존재한다면 exclamationParser()함수를 통해 buf의 `!!`, `!#` 문자열을 정해진 명령어로 수정한다. 이때 `!`문자가 존재하지만, 명령어로 수정할 수 없는 형태일 경우 Linux Shell과 마찬가지로 history에 저장하지 않는다.

③ 이전 명령어와 비교

initFile()로부터 얻어진 마지막 명령어를 저장하는 전역변수인 prevCmd와 현재 입력된 명령어를 비교하고, 다르다면 history.log파일에 저장하며 prevCmd를 최신화 한다.

**ⓒ void verify\_cmd(char \*cmdline)**

입력된 명령어인 cmdline을 parseline()을 통해 argument 단위로 나눈뒤, 해당 명령어를 실행한다. 이때, builtin\_command()의 return 값이 0이라면, 즉 builtin\_command()에 정의되지 않은 명령어라면 execvp()를 통해 실행시켜준다.

이때, execvp() 함수의 경우, 성공적으로 실행 시 현재 프로세스의 실행 흐름이 새로운 프로그램으로 전환되어 실행된 뒤 해당 프로그램을 종료하므로, shell의 유지를 위해선 fork()를 통해 독립적인 child process를 생성하고, 해당 child process에서 execvp()를 실행해야 한다.

추가적으로, 해당 child process는 execvp()의 실행 이후 zombie 상태로 전환되는데, 이는 등록한 SIGCHLD 핸들러에 의해서 자동으로 reaping되고 있으나, 관련 설명은 phase3 설명란에 기재하겠다.

주어진 명령어가 builtin\_command()에 정의되어 함수 내에서 주어진 실행을 완료했거나, child process에서의 실행이 완료되고 그 reaping이 완료됐을 경우 다시 main으로 돌아가 새로운 명령어를 입력 받게 된다. 이때, 입력된 명령어가 exit일 경우 builtin\_command()에서 프로세스를 종료하며 쉘이 종료된다.

**ⓓ int builtin\_command(char \*\*argv)**

verify\_cmd()에서 parseline()에 의해 argument 단위로 나뉜 명령어를 입력값으로 받아 첫번째 인자를 strcmp()를 통해 정의된 명령어와 비교한다. builtin\_command에 정의된 명령어는 다음과 같다.

① exit

exit(0)를 호출하여 shell 프로그램을 완전히 종료한다.

② &

하나의 `&`만 들어온 경우 아무런 행동도 취하지 않는다.

③ cd

두번째 argument로 넘어온 경로로 chdir()을 실행한다. 이때 두번째 인자가 존재하지 않거나 `~`일 경우 getenv()를 통해 home directory의 경로를 구하고 해당 경로로 chdir()을 실행한다. 만약 존재하지 않는 경로라면 “bash: cd: 경로: No such file or directory\n”을 출력하고 return 한다.

④ history

initFile(에서 저장한 history.log의 파일 경로인 filesrc를 통해 history.log를 열고

내용을 읽어온다.

**Phase2 (pipelining)**

phase2의 실행흐름은 다음과 같다.

phase1과 마찬가지로 initFile() historySaver()를 호출하고 main에서 ` ” `와 ` ’ `를 처리하는 과정을 거친다. 이와 동시에, 명령어 내에 ‘|’를 포함하는지 확인한다.

만약 **‘**|’가명령어에 포함되어 있는 경우, phase1과 달리 handlePipeline()를 호출한다. handlePipeline()에서 기본적인 문자열 처리를 완료한 후 execute\_pipline()에서 명령어를 parseline()을 통해 실행가능한 형태로 처리하여 재귀적으로 phase1과 동일한 명령어 실행 방식을 진행한다. 이때, 명령어 실행 전, file descriptor을 수정하여 각 명령어 간의 입력과 출력을 연결한다.

Phase 2에 사용된 주요 함수는 다음과 같다.

**ⓐ void handlePipeline(char \*buf)**

입력받은 명령어를 실행하게 용이하게 처리한다.

① ‘ | ‘ 처리

만약 ‘ | ‘ 문자가 명령어와 띄어쓰기 없이 붙어있는 경우, parseline()에서 이를 올바르게 처리하지 못한다. 따라서, 이 경우 명령어와 pipeline 사이에 빈칸을 삽입한다.

② tokenize

후에 부르는 함수인 execute\_pipeline()에서 각 명령어를 parseline()을 통해 처리하기 위해서는 ‘ | ‘를 기준으로 문자열을 분리하는 과정이 필요하다. 이는 strktok()를 통해 처리하며, 그와 동시에 총 명령어의 개수를 계산한다.

**ⓑ int execute\_pipeline(char \*\*args, int num\_cmds, int in\_fd)**

재귀적으로 연속된 pipeline 명령문을 처리한다. num\_cmds는 처리해야 할 명령어의 수로 재귀적으로 호출할 때 마다 1씩 감소한다. 또한 in\_fd는 이전 명령어의 입력의 끝으로 최초 호출 시 STDIN\_FILENO를 값으로 갖는다.

명령어의 실행 방식은 phase1에서 이용한 builtin\_command()를 통해 실행하거나 fork()와 execvp()를 통해 실행하는 방식과 동일하며 이때 실행 전 file descriptor의 수정은 B-phase2의 Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명을 따른다.

**Phase3 (background process)**

phase3의 실행흐름은 다음과 같다.

‘&’를 포함한 명령어의 경우, phase1, 2와 동일하게 해당 명령을 수행한다. 이때, foreground 프로세스의 실행여부를 확인하는 전역변수 flag1가 1로 설정되며 verify\_cmd의 while(flag1)을 통해 main함수로의 return을 막는다. 이때, SIGINT, 혹은 exit에 의해 프로세스가 종료되면 SIGCHLD handler에 의해 해당 프로세스는 reaping되고 flag1이 0으로 설정되어 foreground 프로세스가 종료된다.

만약 ‘&’을 포함하지 않았다면 main함수로의 return이 즉시 수행되며 다음 명령문의 실행으로 이어질 수 있다.

추가적으로, 실행되는 프로세스를 저장할 구조체를 만들어 관리한다. 구조체의 각 요소들은 하나의 프로세스를 나타내므로, 프로세스의 pid, 프로세스 번호, 프로세스의 상태, 프로세스의 실행위치, 명령문을 저장하며 다음 프로세스와 연결하기 위한 \* 변수가 존재한다.

프로세스가 실행되는 경우 foreground, background와 관계없이 add\_job()을 통해 linkedlist에 저장되며, sigchld에 의해 reaping 되는 경우 remove\_job()에 의해 제거된다.

Phase 3에 사용된 주요 함수는 다음과 같다.

**ⓐ void sigint\_handler()**

SIGINT가 발생 시, linkedlist를 순회하며 foreground 프로세스가 존재하면 SIGKILL함수를 통해 종료시킨다. foreground 프로세스가 존재하지 않는 다면 프롬프트를 다시 출력한다.

**ⓑ void sigchld\_handler()**

zombie 상태의 프로세스들을 reaping한다. reaping한 프로세스가 foreground 프로세스 였다면 flag1을 0으로 세팅하며 remove\_job()을 통해 linked list를 관리한다.

**ⓒ void sigtstp\_handler()**

SIGTSTP 발생 시, linkedlist를 순회하며 foreground 프로세스가 존재하면 SIGTSTP signal을 foreground 프로세스에 보낸다. 또한 foreground 프로세스가 background 프로세스로 전환되므로 해당 요소의 상태와 위치를 stopped와 background로 전환한다.

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)**

