**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 김도현

학번 : 20181256

1. **개발 목표**

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

멀티 코어 프로그래밍 수업의 가장 첫번째 프로젝트로 shell programming 을 통해 shell 을 직접 만들어 보아 system\_level process, process singaling, pipe, background, forground 에서의 프로세스가 작동되는 형태에 대해 이해하는 과제이다.

Phase1 에서는 기본적인 명령어인 cd, ls, mkdir, rmdir, touch, cat, echo, history, exit 등이 잘 작동하도록 적절하게 fork, build-in command 함수, Parsing 함수를 작성하는 것이다. 이때 history에 대한 구현은 skeleton code에 없기 때문에 직접 구현을 해야 한다.

Phase2 에서는 여러 명령어가 긴밀하게 연결되어 있는 ‘|’ 파이프에 대해 구현한다. 이때 역시 phase1 에 대한 내용이 모두 포함되어야 한다.

Phase3 에서는 지금까지 프로세스가 forground 에서 실행되고 있었는데 이를 ‘&’ 명령어를 통해 background에서도 실행시키도록 해보고 jobs 명령어로 어떤 프로세스가 실해오디고 있는지, 해당 프로세스가 어떤 상태인지 알 수 있어야 하고 sigint signal, sigstp signal, sigcont signal 등을 이해하고 해당 signal 이 foreground에서 일어날 수 있도록 구현해야 한다. 여기서도 ‘&’, ‘|’, ‘ ’ ’, ‘ ” ’ 등에 대한 parsing 을 적절히 해야 실제 shell처럼 구현이 될 것 이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

우선 ./myshell 입력 시 다음과 같이 input을 할 수 있게 된다. Parsing 함수에서 따음표의 경우 무시하도록 구현 한 것을 알 수 있다. 예시처럼 기본적인 명령어들을 모두 인식 할 수 있고 build-in command 에 대한 처리를 해주고 특히 history 명령어인 경우 지금까지 쓴 명령어들을 모두 기억하고 있어야 하기 때문에 가장 처음 실행한 곳의 directory location 에 history.txt 파일을 두고 해당 파일에서 history에 관련된 명령어인 !!, !#, history 를 구현해준다. 이때 ! 가 들어간 명령어는 해당 명령어에 대해 치환 후 history에 들어갔음을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

해당 사진은 history가 잘 작동하는지에 대한 확인하는 과정이다.

1. Phase 2

텍스트이(가) 표시된 사진

자동 생성된 설명

Pipeline 에 대한 명령어를 주로 구현 하는 것이다. 여기서는 pipeline을 기준으로 왼쪽에 대한 명령어를 다시 input 으로 받아 오른쪽에 있는 명령어에서 최종 결과를 출력될 수 있도록 해준다.

이때 “hello” 를 보면 원래는 “hello”에 대해서만 search 작업이 일어난다. 하지만 따옴표 parsing 을 통해 hello 전체에 대해 검색을 함을 다음 명령어를 통해서 확인 가능했다.

두 개 이상의 pipeline 명령어에서도 잘 작동함을 알 수 있다.

마지막 명령어 exit 하나 전의 명령어를 살펴보면 sort 라는 파일에 대해 ‘abc’ =>abc 를 검색하고 이를 reverse 하게 sorting 을 해주라는 의미이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Sort 파일은 다음과 같기 때문에 해당결과가 잘 출력됨을 알 수 있다.

1. Phase 3

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

명령어의 가장 끝 부분에 ‘&’ 를 붙으면 해당 명령어는 foreground 에서 시행되지 않고 background 에서 시행 된다. 그 후 jobs 명령어를 통해 현재 진행 중인 모든 프로세스에 대해 현재 상태, command 를 확인 할 수 있다. 이 상태를 바꾸고자 fg %# 원하는 프로세스 번호의 명령어를 입력하면 다시 foreground에서 진행 됨을 알 수 있다. 만약 background에서 중지된 프로세스를 다시 실행시키고자 한다면 bg %# 로 실행 시킨다. Foreground에서의 명령어 중지를 원하면 ctrl+z(SIGSTP), 없애기를 원하면 ctrl+c (SIGINT) 등 역시 잘 실행되고 kill 명령어도 작동하는 것을 확인 할 수 있다.(Terminated)

세부 사항과 비교하자면 jobs에 대해 ‘ [number] 상태 command ‘ 가 잘 나오고 들어오는 순서대로 jobs 에 쌓임을 알 수 있다. 그리고 fg, bg 실행 시에도 그 출력 값이 정상적으로 나오고 kill %2 를 보면 정상적으로 죽었기 때문에 아무 출력 결과가 나오지 않는다. 아래 사진을 보면 fg %2 를 했는데 해당하는 job이 없기 때문에 no such job이 뜬다.

* 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

우선 input으로 받은 명령어에 대해 parsing 작업을 통해 argv 배열로 넣어 준다. 이 후 Argv[0] 이 built in command 인지 아닌지 여부를 strcmp 로 직접 하나씩 비교를 해준다. 만약 build in command 가 아니라면 eval 함수에서 Fork() 함수를 통해 return 값으로 pid를 받아 child process를 생성한다. 이후 child process 에서 Execve 함수로 해당 명령어를 시행 시켜주면 된다.

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

fork 함수에서 child process 가 생성이 되는데 이때 parent process 에서 waitpid 함수를 시행 시켜서 child process 가 종료 될 때까지 계속 기다린다. 이후 waitpid 함수가. Return 값을 주면 reaping 작업을 한다.

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

pipeline 과 일반적인 명령어를 구분하기 위해서 cmd 명령어에서 pipeline의 개수부터 세준다. 그리고 해당 명령어를 parsing 하고 각 명령어를 따로 저장해준다. 만약 pipeline이 없다면 phase1 그대로 명령어를 시행한다. 만약 pipeline이 있다면 반목문을 돌면서 계속 fork로 child process 를 만들어 내며 parsing 한 명령어들을 순차적으로 시행 시킨다. 이때 해당 명령어들은 긴밀하게 연결되어 있기 때문에 미리 만들어 놓은 \*\*fd(file descriptor 배열)에서 i, i+1 의 명령어들을 이렇게 묶어서 Dup2 함수(dup2(a,b) 면 a 에 대한 정보를 b에 넣는다), close 함수(fd를 닫아준다) 를 시행하도록 한다. 예를 들어 처음 stdout에 fd[0][1] 을 넣어주고 fd[0][0] 은 닫는다. 이런 식으로 계속 dup2, close 함수를 번갈아 쓰다가 마지막 파이프에서 read 를 dup 하고 write 을 close 한다. Parent 함수에서는 마지막 fd 에 대한 write을 close 함수를 통해 닫아주고 명령어에 대해 실행하도록 해서 stdout에 찍힘을 알 수 있다. 필자는 반복문을 통해 구현했지만 보고서를 쓰는 시점에서 recursive 하게 구현을 할 수도 있음을 인지하였다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

위에서 언급한 것 처럼 처음부터 pipelien의 개수를 먼저 세어서 pipeline+1의 개수만큼(명령어는 항상 1개 더 많기 때문) 반복문을 돈다. pipe parsing 을 통해 명령어들을 따로 나누어 배열에 저장하고 파이프 기준 왼쪽에 해당하는 명령어의 출력이 파이프 오른쪽의 입력이 되도록 구현하였다. 여기서는 dup2(a,b) 함수로 a를 b에 복사해주고 close 함수로 해당 fd를 닫아주면서 진행했다. 마지막 결과값을 stdout 하면 된다. 이를 parent process 에서 stdout에 출력하도록 하는 것이다. 그 외에는 parent 에서는 pipe 값을 child process 에 넘기는 작업을 해준다.

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

Background 입력은 가장 뒤쪽부터 나올 가능성이 높기 때문에 뒤에서 부터 순차적으로 ‘&’ 기호가 있는지 여부를 체크한다. 만약 있을 경우 bg=1, 없으면 bg=0 으로 하여 foreground 에서 돌리는지 background에서 돌리는지 체크한다. 만약 bg=1 이라면 signal(SIGINT, SIG\_IGN), signal(SIGSTP, SIG\_IGN) 로 ctrl+c, ctrl+z 키를 block 시켜 혹시 모를 불상사를 방지한다. 만약 background job이 아니면 waitpid 함수를 통해 기다리지만 background 일 경우 wait 을 하지 않아 계속해서 다음 명령어를 시행할 수 있도록 해준다. 만약 background의 작업이 정상적으로 종료된다면 해당 결과에 대해 출력할 수 있도록 해준다.

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

Phase1.

기본적으로 skeleton code가 잘 작성이 되어있었기 때문에 명령어 실행에 대해서는 크게 건드릴 부분이 없었다. Builtin\_command 함수에서 우선 ‘cd’가 포함된 명령어인지 확인을 해준다. 만약 cd가 들어있다면 pasring 된 argv에서 chdir를 돌며 입력 받은 directory로 이동 할 수 있는지 없는지 해준다. 그리고 ‘exit’이면 해당 shell 을 종료 할 수 있도록 해주었다. 여기서 중요한 builtin\_command 함수는 history 이다. History 는 내가 적는 모든 명령어들을 저장하고 있어야 한다. 그래서 현재 main이 돌아가는 directory root를 절대 경로로 지정하기 위해서 main 함수에서 getcwd 함수를 통해 현재의 directory 위치를 가져왔다. 거기서 history.txt 파일을 생성해서 해당하는 명령어들을 적절하게 시행 시켰다.

텍스트이(가) 표시된 사진

자동 생성된 설명

History 명령어가 들어오면 해당 history.txt 파일의 모든 명령어들을 해당 번호와 함께 stdout 한다.

만약 !! !# 같은 명령어가 들어온다면 builtin\_command 함수가 들어오기 전에 처리를 해주어야 한다. 그래서 eval 함수에서 이를 처리하였다. 항상 history에 대한 가장 최근 명령어를 저장하고 있다가 !! 가 들어오면 해당 명령어를 cmdline 에 넣어 주고 !#는 반복문으로 # 까지 갔다가 해당하는 명령어를 저장해서 cmdline 에 넣어준다. 이때 input 이 char 이기 때문에 48~57 사이의 ascii 코드가 들어올 때까지만 숫자로 인식하도록 했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이후 가장 최근 명령어와 해당 명령어가 다를 경우 history에 계속해서 추가로 기입하였다. 그리고 parseline 함수를 통해 parsing 하여 built\_in command 함수가 시행된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Parseline 에서는 따옴표를 제거하는 parsing 이 필요하다. 그래서 두개의 포인터 point1, point 2를 따로 설정해 두어 buf를 가리키게 하고 point 1이 하나씩 문자를 돌면서 만약 ‘ ’ ’ ,‘ ” ’ 둘 다 아니면 해당 point1 포인터의 내용을 Point 2에 적어둔다. 이를 \*point1 이 끝날 때까지 반복문을 돌고 마지막에 널 문자를 추가해서 buf 에 따옴표 들어오는 것을 모두 없애 준다. (이 포인터 들은 아무 값을 가지지 않는다. 즉 바로 buf 사용하면 된다.)

텍스트이(가) 표시된 사진

자동 생성된 설명

Builtin\_command 가 아닌 명령어는 /bin/ 이나 /usr/bin 에 대부분 들어있다. 그래서 해당 문자열을 명령어 앞에 붙여서 execve 함수로 실행을 하였다. 이때 ./a.out 과 같은 실행 파일은 앞에 어떠한 것도 붙어서는 안된다. 그래서 이 경우 execve(argv[0],argv,environ) 으로 적어준다. 해당 일들은 모두 eval 함수에서 구현했다.

Phase2

여기서는 phase 1 에 비해 추가적으로 pipe에 대한 개념이 도입됐다. 따로 함수를 추가하지는 않고 모두 eval 함수 내에서 진행하였다. 먼저 numofpipe int 형 변수에 cmdline을 순차적으로 하나씩 돌며 pipe의 개수를 센다. 그리고 해당 pipe의 개수를 통해 parsing 작업을 한다. 필자는 char [200][200] pipe\_command 라는 변수에 저장하였다. 이때 strchr 함수를 사용하면 pipe 포인터가 자동으로 나오기 때문에 이를 이용하여 parsing 을 한다. 이후 가장 마지막 명령어 까지 적어준다.

이때 우리는 history에 가장 최근 명령어와 중복되지 않았다면 해당 명령어를 다시 적어줘야하기 때문에 다시 buf에 cmdline 을 복사해서 비교 후 history에 적는 과정이 필요하다. History 에 관한 명령어 실행이면 Stdout에 명령어를 출력해준다. 이미 numofpipe 변수가 있기 때문에 만약 해당 값이 0 이면 파이프가 없다는 뜻이다. 그래서 우리가 알던 방식 그대로, Phase 1 처럼 시행하면 된다.

하지만 pipe 가 있다면 해당 pipe 수+1 만큼 반목문을 돌며 command 를 수행할 수 있도록 해준다. 여기서는 pipe 기준으로 왼쪽의 명령어가 시행되고 그 결과를 오른쪽의 pipe에 넘기는 과정(쓰는 과정)이 필요하다. 이를 dup2 함수를 통해서 해주었다. 이때 가장 처음에는 읽기는 필요 없고 쓰기만 필요해서 close로 닫았고 가장 마지막에는 pipe에 대해 읽어 들어야 하기 때문에 해당 명령어를 읽고 pipe 쓰기를 close 했다. Parent 에서는 close(fd[][1]) 을 해 쓰기 기능을 닫아주고 waitpid 함수를 통해 child process를 reaping 하도록 한다. 최종적으로는 모든 명령어들이 모두 합쳐진 최종 명령어가 수행된다.

Phase 3.

여기서는 프로세스 상태가 여러 종류가 있기 때문에 우선 정의부터 내렸다. Zero는 아무것도 안하는 상태, r 은 background run, f 는 foreground run, d 는 명령어가 끝난 상태, s 는 stop, suspened 된 상태, k 는 kill 당한 상태 이다. 그리고 job에 대해 구조체를 설정 했는데 해당 구조체에는 job의 id(몇번째 Job에 해당하는지), status 프로세스 상태, pid 로 process 의 pid, commandline 으로 해당 job에 대한 명령어를 저장할 수 있도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Job\_num 은 job에 대한 index 변수이다.

우선 sigcld\_handler 함수 부터 작성했다. 여기서는 sigchild signal 에 대한 시그널로 인해 작동한다. WNOHANG, WCONTINUED 옵션값을 줘서 waitpid 한다. 만약 정상적으로 wait 이 되면 return 값이 0보다 크게 되고 bg 에서 일어난 일들이 정상적으로 끝났다면 해당 job 의 status 를 ‘d’으로 바꿔 나중에 job\_ended 함수에서 이를 catch 할 수 있도록 해준다.

Sigstp\_handler 는 모든 job을 돌면서 foreground 에서 실행되고 있는 job 들을 모두 확인해본다. 이때 kill 함수에 SIGSTOP 시그널이 발생한다면 해당 job 의 상태를 ‘s’ 로 바꾸고 해당 job 은 정지되었음을 stdout 에 출력한다. 만약 시그널이 발생하지 않았으면 아무 일도 일어나지 않는다.

Sigint\_handler 는 모든 job을 돌면서 해당 job이 끝난 상태이거나 아무 job이 없을 경우를 제외한 status 에서 일어날 수 이도록 조치하였다. 이때 kill 함수에 SIGINT 시그널이 발생한다면 해당 job에 대한 status를 zero로 바꾸어 해당 Job 을 초기화 하도록 한다.

Job\_ended 함수는 eval 함수 다음에 실행되는 함수로 최종적으로 끝난 job 이 있다면 해당 job을 stdout 하고 job\_num을 다시 재설정 하는 함수이다. 이때 kill 된 프로세스의 경우는 아무 상태도 출력하지 않고 그냥 status를 zero 로 초기화 시키고 정상 종료된 프로세스는 종료되었음을 stdout에 출력했다. 그리고 계속 job에 대해 탐색을 했을 때 모든 job status가 종료 되었거나 zero 값을 가진다면 job\_num=0 으로 초기화 해서 다시 index가 0부터 시작할 수 있도록 해주었다.

Main 함수에서는 가장 처음에 job을 모두 초기화를 시켜준다. 이때 초기 status 값은 zero, pid=0, job\_id=0, commandline=’\0’ 이다.

그리고 signal을 받을 수 있도록 signal(SIGCHILD,sigchld\_handler), signal(SIGTSTP, sigtstp\_handler), signal(SIGINT, sigint\_handler) 를 적었다.

나머지 signal에 대한 처리는 eval 함수에서 대부분 처리 했다.

다음은 Eval 함수에서 추가된 내용이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

background 에서 일어나는 일인지 아닌지 처리하는 과정을 추가했다. Commandline의 뒤에서부터 &가 있는지 없는지 파악을 해서 있으면 bg=1 로 바꿔 background 에서 실행해야 함을 알렸다. 만약 child process 에서는 bg=0 이라면 foreground 에서 시행되는 Job 이기 때문에 signal(SIGTSTP, sigtstp\_handler), signal(SIGINT, sigint\_handler) 를 추가해서 ctrl+z로 일시정지, ctrl+c 로 interrupt 하기 에 대한 기능을 사용 할 수 있도록 했다. Background job 이라면 해당 기능은 SIG\_IGN handler로 block 해야 한다. Parent process 에서는 지금 까지 일어난 일들에 대해 job 구조체에 기록해야 한다. 그래서 해당 내용들을 모두 기록해주고 waitpid에서 WUNTRACED 옵션을 통해 ctrl+z suspended 에 대해서도 wait 함수가 catch 할 수 있도록 해야 한다. 만약 정상적으로 wait 해서 reaping 이 되었다면 해당 Job을 다시 초기화 해야 한다.

만약 background에서 시행되었다면 foreground에서 실행되지 않기 때문에 stdout에 보이지 않는다. 그래서 그냥 waitpid 없이 바로 Job에 해당 정보만 기록 하고 return 해준다.

Pipe가 있을 경우도 비슷하기 때문에 생략하도록 하겠다.

마지막으로는 builtin\_command 함수에 대한 내용이다.

텍스트, 모니터, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

우선 jobs 명령어가 들어온다면 지금까지 struct 구조체에 기록한 job status 가 zero 인 경우를 제외하고 job에 대한 정보를 모두 출력해야 한다. 텍스트이(가) 표시된 사진

자동 생성된 설명

Kill 명령어가 들어온다면 % 기호가 있는지 없는지 부터 확인한다. % 다음에 숫자가 오게 되는데 해당 번호에 대한 job을 삭제해야 한다. 만약 해당 번호가 없다면 ‘no such job’ 이라는 오류 메세지를 stdout 하고 return 한다. Status를 k 로 바꾸고 kill 함수에서 SIGKILL 시그널을 호출해서 해당 job을 죽인다.텍스트이(가) 표시된 사진

자동 생성된 설명

Bg 명령어가 들어온다면 % 기호가 있는지 없는지 부터 확인한다. % 다음에 숫자가 오게 되는데 해당 번호에 대한 job을 background에서 돌리게 해야 한다. 만약 해당 번호가 없다면 ‘no such job’ 이라는 오류 메세지를 stdout 하고 return 한다. Status를 r 로 바꾸고 kill 함수에서 SIGCONT 시그널을 호출해서 해당 job이 background에서 시행됨을 알린다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Fg 명령어가 들어온다면 % 기호가 있는지 없는지 부터 확인한다. % 다음에 숫자가 오게 되는데 해당 번호에 대한 job을 foreground에서 돌리게 해야 한다. 만약 해당 번호가 없다면 ‘no such job’ 이라는 오류 메세지를 stdout 하고 return 한다. Status를 f 로 바꾸고 kill 함수에서 SIGCONT 시그널을 호출해서 해당 job이 foreground에서 되도록 하고 또한 무한 반복문을 통해 해당 job이 끝날 때 까지 기다렸다가 job status가 f 가 아닐 때 Return 한다.

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)**

**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**