**System Programming Project 4**

담당 교수 : 김영재

이름 : 최재원

학번 : 20171700

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

본 프로젝트는 리눅스 쉘(Linux shell)을 총 3개의 Phase로 구현한다. 쉘은 커맨드라인을 읽고, 입력된 라인을 해석해서 해당하는 명령을 실행시킨다. Phase1에서는 표준 입력으로부터 받은 명령을 child process에서 처리하고, child process는 fork함수를 사용하여 만든다. Built – in command 들은 parent process에서 처리하며, 이외의 명세서에 상기된 cd, cat, ls, mkdir, rmdir, touch, echo를 child process에서 처리한다.

Phase2에서는 리눅스 내의 파이프라인 기능을 구현한다. ‘|’ 문자로 구분된 한 개 이상의 연속된 명령어들을 pipe 함수를 사용하여 구현하였다. Parent 와 child process 사이에 pipe를 만들고, child process의 결과가 다음 child process의 input으로 들어가는 일련의 과정을 반복하여 구현한다.

Phase3에서는 background 기능을 구현한다. Background에서는 여러개의 child process가 돌아갈 수 있으며, foreground에서는 background process들의 종료를 기다리지 않고 다른 명령어들을 수행할 수 있다. ‘&’문자가 command의 마지막에 붙으면 background 에서 작동한다. 또한 ctrl + z, ctrl + c 를 SIGTSTP handler, SIGINT handler에서 처리하고, SIGCHLD 에 대한 handler도 작성함으로써 background process들의 수행이 종료되었을 때 parent process로 signal을 보내, 정상적으로 process들이 reap되어 defunct process가 생기지 않도록 하였다. 추가적으로 kill, bg, fg, jobs 명령어의 구현을 통해 상기된 내용들을 수행하도록 하였다.

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

**< ‘ls’ command >**

Makefile 을 통해 maker 하여 컴파일한 후, 실행파일 shellex를 실행시킨 후 ls 를 입력하면, 해당 directory 내에 있는 파일 및 디렉토리가 출력된다. 이외에도 ls -al와 같은 옵션이 추가된 ls 명령어도 정상 작동한다.

**< ’mkdir’ & ‘rmdir’ command >**

‘Mkdir 이름’을 입력하게 되면 해당 이름의 디렉토리가 생성되고, ls를 통해서 확인할 수있다. ‘rmdir 이름’을 입력하게 되면 해당 이름의 디렉토리가 삭제되고, 이또한 ls를 통해서 확인할 수 있다.

**< ‘touch’ command >**

‘touch 이름’을 입력하게 되면 해당 이름의 파일이 생성된다. 예시로 ‘touch dummy2.c’을 입력하면 해당 파일이 생성하되고, comment가 없는 더미 파일이 생성됨을 확인할 수 있다.

**< ‘echo’ command >**

‘echo string’을 입력하게 되면 해당 string이 그대로 출력된다. “ “(큰 따옴표) 또는 ‘ ‘(작은 따옴표)안에 들어간 string은 따옴표를 제외하고 위와 같이 출력된다. 추가적으로 ‘echo 환경변수’ 입력 시에는, 환경변수 안에 들어있는 값이 출력된다.

**< ‘cat’ commend >**

‘cat 파일이름’ 입력을 하면, 해당 파일 안의 내용이 출력된다. Cat만 단독으로 입력 시에는 프롬포트에 표준입력을 받고, 해당 string을 복제하여 출력해준다. Ctrl + d를 입력하여 빠져나올 수 있다.

**< ‘cd’ & ‘exit’ command >**

Cd를 입력하면 home directory로 돌아가며, ‘cd 이름’을 입력하면 해당 이름의 디렉토리로 들어간다. ‘cd ..’을 입력하면 상위 디렉토리로 들어감을 볼 수 있다. Exit명령어는 shell 프로그램을 종료시킨다.



1. **Phase 2**

**< pipe commands1 >**

“cat dum1 | grep -v “abc” | sort -r”

Phase2에서는 파이프 커맨드를 구현한다. cat dum1 의 결과를 pipe를 통해서 다음 명령어의 input으로 들어가고, 해당 dum1파일에서 “abc”가 들어간 line을 제외하는 grep -v 명령어를 수행한다. 해당 명령어의 결과는 또 다음 명령어의 input으로 들어가, sort -r (오름차순) 으로 출력하는 명령어를 수행하게 되고, 최종적으로 abc가 들어간 line을 제외하고 출력됨을 확인할 수 있다.

**< pipe commands2 >**

“ls -al | grep dum1” & “cat dum1 | less”

‘ls -al | grep dum1’ 명령어 입력을 하면, ls -al의 결과가 다음 명령어인 grep dum1으로 들어가며 dum1이 포함된 line이 출력된다.

‘cat dum1 | less’ 명령어를 하면, dum1 파일 내부의 내용을 새로운 화면에서 띄우게 되며, q를 입력하고 빠져나올 수 있다.

위에 언급한 명령어들 외에도 파이프라인을 이용한 command들이 정상작동함을 확인하였다.

1. **Phase 3**

**< pipe commands in background >**

Phase2에서 구현했던 명령어들에 &를 붙이면 background에서 돌아감을 확인하였다. 포그라운드에서의 수행과 다르게 parent process가 background의 child process가 수행이 끝날 때까지 기다리지 않으므로 바로 다음 명령어를 받을 수 있는 상태가 되고, 백그라운드에서 수행중이였던 process가 끝나면 출력된다. 프로세스의 결과 출력이 다음 프롬포트의 입력을 받는 “CSE20171700-SP..>”과 겹쳐서 출력이 됨을 확인할 수 있다.

< ‘jobs’ & ‘fg’ & ‘bg’ & ‘kill’ commands >

test file ‘while.c’를 무한루프 파일로 만들어 테스트하였다. 해당 파일을 포그라운드에서 실행하고, ‘Ctrl + Z’를 입력하면 stopped 상태가 되며, ‘jobs’명령어를 통해 확인할 수 있다.

‘bg %#’ 명령어를 입력하면 해당 job number를 가진 백그라운드 process가 stopped 상태에서 재게 됨을 확인하였다.

‘fg %#’ 명령어를 입력하면 해당 job number를 가진 백그라운드 process가 stopped, 또는 running 상태로 백그라운드에 있다가 포그라운드로 넘어가 수행됨을 볼 수 있다. 이 때 ‘Ctrl + Z’를 입력하면 다시 stopped 상태가 된다.

‘Ctrl + C’를 입력하여 포그라운드 프로세스를 종료 시킬수 있고. ‘kill %#’명령어는 해당 프로세스를 종료시킨다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명
* phase1의 명세서에 설명된 명령어들을 bulit – in 을 제외하고 전부 child process에서 처리하기 위해서, eval 함수에서 fork 함수의 return 값을 pid\_t 자료형의 pid 변수로 받았고, 이 값에 따라서 0일 때는 child process인 경우로, 0이 아닐 때는 ( child process의 process id 값) parent process인 경우로 하여 작성하였다. Child process인 경우에서 명령어를 수행하고 끝낼 때까지 parent process는 waitpid 함수를 통해서 해당 child의 pid 프로세스를 기다리게 된다. 또한 fork함수를 호출하기 이전에 따옴표 또는 ‘|’(파이프 라인) 등의 tokenizing 과정을 완료하였고, 이과정에서 argv 배열에 저장된 명령어들을 통해서 명령어를 처리한다.
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
* main 함수에서 특정 signal에 대한 처리를 signal handler에서 할 수 있도록 signal()함수를 호출하였다. SIGINT, SIGCHLD, SIGTSTP, SIGCONT 신호들을 각각의 handler에서 처리하며, 포그라운드와 별개로 수행된다. SIGCHLD에서는 다른 signal들을 일괄적으로 waitpid() 함수를 통해서 따로 수거(reap)해 주고, while문을 통해 반복한다. Eval 함수 내부의 fork() 함수를 호출한 이후에, child process를 처리하는 부분에서는 signal(SIGINT, SIG\_IGN) 및 signal(SIGINT, SIG\_DFL) 함수를 사용하여 background에서 수행되는 child인 경우에는 SIGINT (Ctrl + C)를 무시할 수 있도록 하였다. 또한, backgorund에서 수행되는 경우가 아닐 때는 해당 자식 프로세스를 부모 프로세스에서 waitpid를 통해서 수거(reap)해 주므로, 이때는 SIGCHLD 신호를 무시해 주었다. 다음 명령어를 받을 때는 다시 main으로 돌아가서 signal함수로 정의했던 대로 돌아간다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
* 파일 디스크립터 역할을 하는 Fd[MAXARGS][2] 배열과 pipe() 함수를 통해서 파이프를 생성하고, 최대 MAXARGS개의 명령어들이 파이프라인으로 연결된 경우를 구현하였다. Unnamed pipeline을 사용하였으며, 따라서 다음 명령어로 넘어갈 때마다 fork함수를 호출하였다. Tokenizing된 명령어들이 들어가 있는 argv 배열을 사용하여, 가장 왼쪽(처음)에 적힌 명령어를 먼저 수행한다. STDOUT\_FILENO에 fd[0][1]을 dup2를 통해 복사해 주고, execvp()함수를 통해서 수행시키며, 이는 다음 명령어의 input으로 들어간다. STDOUT\_FILENO 와 STDIN\_FILENO에 대한 복사 과정이 반복된다.
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
* 파이프 라인이 하나일 경우, 처음 명령어의 결과(stdout)이 다음 명령어의 입력(input)으로 들어가므로, 각각에 대해서 dup2를 사용하여 file descriptor를 복사하였다. 1개 이상인 경우부터 처음 명령어와 마지막 명령어의 사이의 명령어들을 for문을 통해서 결과(stdout)를 다음 명령어의 입력(stdin)으로 넘어갈 수있게 하였고, 이때 파이프라인의 개수에 상관없이 매번 fork를 함으로써 중간에 parent를 거쳐서 다시 child process로 이동하도록 하였다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

명령어의 마지막에 &이 붙으면 background에서 수행하도록 하였다. Background 명령어에 대해선 부모 프로세스가 기다리지 않으므로 Waitpid를 건너뛰었고, 해당 자식 프로세스에 대한 종료는 sigchild handler를 이용하여 출력하는 방식으로 부모 프로세스에 알려주었다. 포그라운드 명령어에 대한 process id 배열을 따로 만들고, 자식 프로세스 처리 블락 안에서 setpgid함수를 호출함으로써 SIGTSTP 및 SIGINT가 포그라운드 프로세스에만 적용되도록 하였다. 또한 백그라운드에서 수행중인 프로세스에 대한 정보를 array안에 저장하였고, 이를 통해서 jobs 명령어 입력시에 출력할 수 있도록 하였다. Fg 및 bg 명령어를 수행할때는 직접 kill함수를 사용하여 각 signal을 넘겨주었다.

* 1. **개발 방법**

**<Phase1>**

**Function :**

1. Void eval(char\*cmdlline)

이 함수 내에서 fork 함수를 호출하고, return 값을 pid로 받았다. 또한 pid 값에 따라 0인경우 execute() 함수를 호출하여 그 내에서 명령어에 따라서 execvp() 함수를 사용하여 실행하거나, 추가적인 예외처리가 필요한 경우 따로 함수를 만들어 호출하였다. Pid가 0이 아닌경우 parent process이므로, Waitpid 함수를 통해서 child process가 끝날때까지 기다렸다가, 정상적으로 끝나지 않은 경우에 대해서 에러 메시지를 출력하도록 하였다. 또한 따옴표에 대한 무시를 하기 위해서 tokenizing하는 과정을 수행하였다.

1. Pid\_t Waitpid(pid\_t pid, int \*iptr)

Waitpid()를 수행해주며, 함수 내에 선언된 retpid값으로 해당 리턴 값을 받아, 0보다 작을 경우 에러 메시지를 출력한다. 이 외에 child process가 비정상적으로 종료된 경우에 대한 메시지도 출력해 준다.

1. Void execute(const char \*filename, char \*argv[])

Main에서 받은 명령어에따라서 case를 나누어서, 기본 명령어들에 대한 수행을 해준다. 일반적인 경우 execvp() 함수를 사용하여 처리해 주었고, cat, echo에 대해서는 따로 함수를 호출해 준다.

1. Command\_echo(char \* argv[])

Echo 명령어가 들어왔을 때 환경변수를 출력해야 하는 상황 등에 대한 예외처리와 실행을 함께 해준다.

1. Command\_cat(char \*const argv[])

Cat 명령어가 들어왔을 때 cat만 들어온 경우와, cat filename이 들어온 경우에 따라서 나누었고, redirection도 임시로 구현하였다.

1. Command\_cd(Char\* const argv[])

Cd 명령어가 들어왔을 때 built. In 함수에서 해당 함수를 호출하여 cd를 처리한다. Chdir() 함수를 사용하였고, 인자는 argv[1]이며, cd 만 입력된 경우, getenv() 함수를 통해서 $HOME 환경변수 값을 가져와, 이를 chdir의 인자로 사용한다.

1. Builtin\_command(char \*\*argv, char\*cmdline)

해당 함수에서 cd와 exit에 대한 경우를 추가하여 이에 대한 명령어 처리를 해주었다.

**<Phase2>**

**Function:**

1. Eval(char \*cmdline)

Eval 함수 내에서 ‘|’ 에대한 tokenizing을 추가로 해주었고, pipeline이 있는 경우, pid == 0인 경우에서 따로 pipe\_cmd 함수를 호출해서 해당 함수에서 명령어에 대한 수행을 해주었다.

1. Pipe\_cmd(char \*const argv[], int \*const pindex, int pcount)

파이프라인으로 연결된 n개의 명령어 들에 대해서, 가장 처음 명령어와 가장 마지막 명령어, 그리고 나머지 명령어들에 대한 수행을 나누어 크게 3개의 블록으로 파이프라인을 구현하였다. 이차원 배열 fd가 파이프를 가르키는 File descriptor 역할을 하며, pipe 함수를 이용하였다. 가장 먼저 받은 명령어에 대해서는 결과를 다음 결과로 넘기므로, stdout을 가르키는 fd에 fd[0][1]을 복사해주고 명령어 실행은 execvp를 사용하였다.

다음 명령어 부터는 Fd[i][0]을 STDIN\_FILENO로 복사하고, Fd[i+1][1]을 STDOUT\_FILENO로 복사해서 1~ n-1개의 명령어들에 대해서 이를 수행할 수 있도록 구현하였다. 마지막 명령어는 input을 받고 끝나므로, STDIN\_FILENO만 이용하여 앞의 모든 명령어들의 파이프라인 결과에 대해서 수행하도록 하였다.

**<Phase3>**

**Varable:**

* + 1. Struct Jobs joblist[MAXJOBS]

백그라운드에서 실행되는 프로세스에 대한 정보를 저장하는 구조체 배열이다. Process id 를 저장하는 pid, job number를 저장하는 number, 멈춘 상태인가를 나타내는 stop, 상태를 나타내는 state[30], 명령어를 저장하는 cmd[100] 등을 저장하고 있으며, jobs 명령어를 구현하기 위한 구조체이다. 해당 프로세스가 종료되거나 포그라운드로 넘어갈 때 삭제된다.

* + 1. Pid\_t cpids[MAXJOBS]

포그라운드에서 실행되는 프로세스에 대한 process id를 저장하는 배열이다. 해당 배열에 들어있는 값은 프로세스가 종료될 때 삭제된다.

* + 1. Char fcommands[MAXJOBS][MAXARGS]

포그라운드에서 실행되는 프로세스에 대한 명령어들을 저장하는 배열이다. 포그라운드에서 수행되다가 SIGTSTP에 의해서 jobs 때 출력되어야 하는 경우 이 배열을 사용한다.

**Function:**

1. Void addjob(Jobs\* job,pid\_t id, char state, char\*const command, int stp)

새로운 백그라운드 명령어가 joblist 명령어에 추가된다. Stopped인 상태로 들어오는 경우 예외처리를 해준다.

1. Jobs deletejob(Jobs\* job,pid\_t id)

백그라운드 명령어가 완료되거나 kill되었을 때 joblist에서 삭제해주는 역할을 수행한다.

1. Void listjob(Jobs\* job)

‘Jobs’ 명령어 입력시 joblist에 들어있는 값들을 출력한다.

1. Jobs Getjobpid(Jobs\* job,pid\_t id,int st)

Kill, fg, bg 명령어들을 수행할 때 해당되는 pid에 대한 정보를 가져오기 위해 정의된 함수이다. Job number로 해당되는 정보를 찾아 반환하는 경우와 pid로 찾는 경우로 나누어 구현하였다.

1. Void Changetarget(Jobs\* job,int jnum)

Joblist를 출력할 때 현재 대상인 프로세스를 나타내는 ‘+’ , ‘-‘, 등을 업데이트 해주는 함수이다.

1. Void waitfg(pid\_t pid)

‘fg’ 명령어를 입력받았을 때 해당되는 process를 포그라운드에서 수행되고 완료되기까지 기다렸다가 수거(reap)해주는 함수이다. While문과 sleep 함수를 사용하여, TSTP를 입력 받아 중지되거나, 정상종료 되는 경우 등에 따라서 수거할 수 있도록 구현하였다.

1. void sigchild\_handler(int sig)

SIGCHLD 시그널이 발생하였을 때 이 함수를 통하여 처리되도록 한다. 해당 함수는 백그라운드에서 수행되던 프로세스들이 종료되었을때 좀비 프로세스가 되지 않도록 하는 역할을 하며( 종료된 프로세스를 waitpid를 통해서 수거함), 특정 시그널에 따라서 메시지를 출력한다.

1. void sigint\_handler(int sig)

ctrl + c 가 들어왔을 때 sigint 시그널이 발생하고 해당 함수가 호출된다. Kill함수를 사용하여 cpids 배열에 들어가 있는 모든 포그라운드 프로세스들을 sigkill 시그널로 종료시킨다.

1. void sigstp\_handler(int sig)

ctrl + z 가 들어왔을 때 sigtstp 시그널이 발생하고 해당 함수가 호출된다. Kill함수를 사용하여 cpids에 있는 값들을 모두 joblist 배열에 addjob함수를 사용하여 추가해준다.

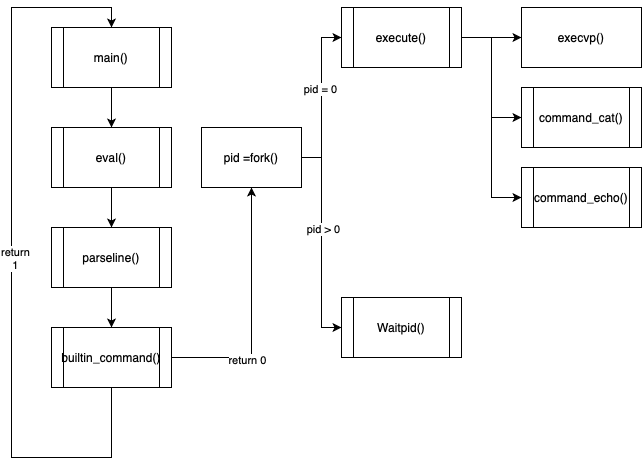
1. void sigcont\_handler(int sig)

fg 나 bg 명령어에 의해서 멈췄던 프로세스가 다시 시작되면 해당 함수가 호출된다. State를 running으로 바꿔주고, kill함수를 사용하여 SIGCONT를 보낸다.

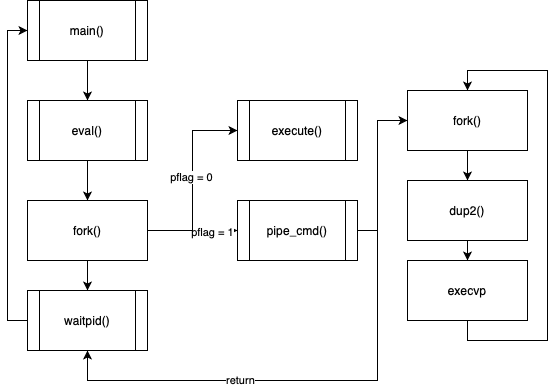
1. Int Builtin\_command(char \*\*argv, char\* cmdline)

Fg와 bg 및 kill 명령어를 builtin에서 구현하였다. 각각의 명령어에 대해서 해당되는 메시지를 출력하며, kill인 경우 SIGKILL로 종료시키고, fg 와 bg는 sigcont 시그널을 kill 함수로 보낸다. 특히 fg의 경우 waitfg 함수를 통해서 해당 프로세스가 종료되거나 중단될때까지 기다린다.

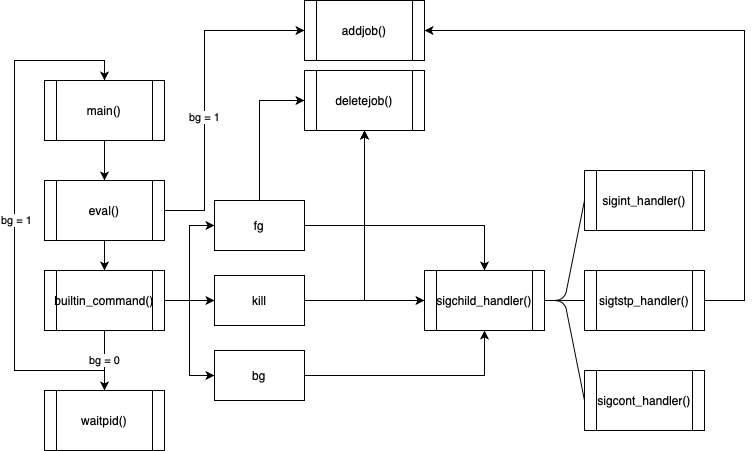
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****