**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 김승원

학번 : 20182186

**1. 개발 목표**

본 프로젝트의 이름은 Myshell이다. Myshell(Project 1)은 Linux shell program을 구현하는 것을 목표로 한다. Linux shell은 사용자가 입력하는 명령들을 실행하며, 프로그램을 실행한다. 반복적으로 프롬프트를 출력하여 명령을 기다리고, 입력된 명령어를 확인한 후 작업한다. concepts of system-level process control, process signaling, interprocess communication, and running processes and jobs in the background에 대하여 학습하고, 직접 구현한다.

3개의 Phase로 진행한다. 각각 하단의 Phase에서 구현한 내용을 바탕으로 shell의 기능을 확장할 것이므로, 목표 내용을 모두 구현하고 다음 phase로 진행한다.   
 Phase1에서는 프로세스에 대한 이해를 요구한다. exec(), wait() 등의 syscall을 사용해서 입력 받은 명령어를 child process에서 실행할 수 있도록 구현한다. ls, cd, mkdir, rmdir, cat, exit, touch, echo, history 기능이 수행될 수 있도록 작성한다.

Phase2에서는 Redirection과 pipe(‘|’) 기능을 추가한다. Pipeline에 있는 각 명령어들을 위한 New process를 생성하고, parent process가 마지막 command를 기다리도록 한다. 이때 dup()와 dup2() syscall을 이용해서 한 프로세스의 출력을 다른 프로세스 입력으로 전달되는 기능을 이용한다.

Phase3에서는 프로세스를 백그라운드에서도 실행 되도록 구현한다. 시그널을 이해하고, jobs, bg, fg, kill 명령어들이 실행될 수 있도록 한다. job들을 background와 foreground간의 이동이 가능하고, 프로세스의 상태를 변경하고, 또한 CTRL + Z(SIGTSTP)를 사용해 stop시키거나 kill을 이용하여 종료 시킬 수 있도록 한다.

**2. 개발 범위 및 내용**

**A. 개발 범위**

**1) Phase 1**

• cd: 원하는 디렉토리로 이동하도록 구현

• ls: 현 위치의 파일, 디렉토리 목록을 보여주도록 구현

• mkdir: 디렉토리를 생성하도록 구현

• rmdir: 빈 디렉토리를 삭제하도록 구현

• touch: 빈 파일을 생성하도록 구현

• cat: 파일 내용을 출력하도록 구현

• echo: 문자열을 출력하도록 구현

• history: shell이 시작되고 난 뒤 실행됐던 모든 명령어를 추적하도록 했다.   
 모든 명령어들이 기본적으로 추가되도록 했으나, !로 시작하는 !#과 !!명령어에 대해서는 history 목록에 넣지 않는다. 또한 중복된 명령어가 입력될 경우 history에 포함시키지 않는다.

• exit: 모든 자식프로세스를 종료하고, shell을 종료하도록 구현

fork()를 통해서 process를 생성하고, exec()를 사용하여 process를 새로운 program으로 대체하도록 한다.

**2) Phase 2**

‘|’ pipe 명령어를 사용하여 ‘|’의 앞부분에 해당하는 명령어를 ‘|’의 뒷부분에 해당하는 명령어의 input으로 입력을 전달하고, 명령이 실행된다. Pipe가 2개 이상이 나오면, 계속해서 그 다음으로 입력을 전달하도록 한다. Process를 fork()를 통해 생성하고, exec()를 사용해서 프로그램을 대체한다.  
• ls | grep filename: ls 명령어의 output을 grep filename의 input으로 전달하여 명령을 실행한다. 현재 디렉토리 output들을 확인하여 grep filename에서 input으로 받고, filename에 해당하는 부분만 출력한다.

• cat filename | less: cat filename으로 해당 파일 내용을 less명령어의 input으로 전달하여 출력된 결과가 페이지 단위로 스크롤 할 수 있도록 한다.

• cat filename | grep -i "abc" | sort -r: Cat filename을 통해 file의 내용을 grep -I “abc”의 input으로 전달하고, grep -I “abc”를 실행한 뒤의 output을 또 다시 sort -r의 input으로 전달한다. 명령어의 결과를 다음 명령어로 2번 전달한다.

**3) Phase 3**

• jobs: 현재 background에서 실행이거나 중지된 작업 목록을 표시함

• bg ⟨job⟩: stopped background 작업을 Running상태로 변경함, 명령어 뒤에 &를 붙여서 실행하도록 한다.

• fg ⟨job⟩: 중지됐거나 실행중인 background 작업을 foreground에서 실행함

• kill ⟨job⟩: background job을 종료한다. <job>은 작업번호로 실행되도록 함.

bg, fg, kill은 모두 bg %num과 같이 %num을 붙여서 실행되도록 한다.

• ls -al filename | grep filename &: &앞의 해당명령어가 background작업에서 실행되도록 한다. 명령어를 입력 후 jobs를 사용하여 목록을 확인하면, [Num] Running/stopped (입력한 명령어)로 출력되도록 한다.

• cat filename | grep -i "abc" &: 위와 동일하다.  
  
 또한 SIGTSTP(Ctrl + Z)를 구현하여 Foreground에서 실행중인 작업을 일시 중지시켜서 background로 전환하도록 한다. SIGINT(Ctrl + C)를 통해 process를 강제 종료하는 signal을 구현한다. 또한 SIGCONT로 일시 중지된 process를 다시 시작하도록 한다. SIGTSTP의 경우 fg명령어로 SIGCONT signal을 통하여 다시 시작할 수 있도록 구현한다.

**B. 개발 내용**

**1. Phase1 (fork & signal)**

**1) fork를 통해서 child process를 생성하는 부분에 대해서 설명**  
 fork()를 호출하여 child process를 생성한다. fork를 진행할 시 성공적으로 child process가 생성됐을 경우 parent process는 child process의 PID값을 받고, child process는 ‘0’값을 반환한다. 만약 제대로 생성되지 않았을 경우 parent process는 ‘-1’값을 반환 받고, perror 또는 unix\_error function을 통해 메시지를 오류 종류 출력하도록 했다.

Child process가 ‘0’을 반환할 경우, 제대로 생성된 것이다. 따라서 fork 이후 라인부터 execve()를 통해 입력 받은 명령을 수행하도록 한다. 이때 execve() system call은 실행할 프로그램의 파일 경로나, argument, 환경 변수 등을 인자로 받아서 성공적으로 호출될 경우 현재 실행중인 process가 새로운 program으로 대체 된다. 이때 parent process는 child process가 새로운 program을 실행하고 종료될 때까지 waitpid() system call을 사용해서 기다리도록 한다.

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

child process가 종료된 경우 parent process에게 signal을 보내는 방법은 waitpid() 함수를 활용한다. Child process가 종료되기까지 parent process를 대기 상태로 놓고, child process가 종료되면, 그 signal에 대한 정보를 반환한다. 이를 통하여 parent process는 child process가 종료된 것을 확인한 후 이후 작업을 진행하도록 했다. 이번 개발은 이 방법을 사용했다.

또한 이외의 방법으로 kill system call을 활용해서 부모 프로세스에게 signal을 보낼 수 있다. Child process에게 getppid() system call을 사용하여 parent process의 PID를 얻고, kill() system call을 사용해서 parent process에게 signal을 보내는 방법이 있다.

**2. Phase2 (pipelining)**

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

Pipeline은 pipe, fork, dup2, execvp system call을 활용했다. Pipe 함수를 통해서 각 명령어 사이에 파이프를 생성했다. 이후 fork를 사용해서 child process를 생성하였고, 각 자식 process는 해당 명령을 실행하도록 했다. 이 때 dup2를 사용해서 child process의 stdin과 stdout이 알맞은 pipe의 read 또는 write 끝에 연결했다. 이후 execvp를 사용해서 명령을 실행하고, child process는 새로운 프로그램으로 대체되도록 했다.

**2) Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명**

우선 argv 문자열 배열을 검색하면서 명령어를 분석한 다음 ‘|’문자를 기준으로 파이프 라인을 분할 시켰다. 이후 배열(commands)을 생성하여 저장시켰다. 이렇게 해서 각 명령을 분리하고, pipe() 함수를 사용해서 분리한 명령 사이에 pipe를 생성했다. Pipefiledescriptor(pipefds) 배열을 선언하고, 각 pipe의 file discriptor를 저장했다. 이때, 전체 명령어 개수보다 하나 적은 수의 pipe를 생성하도록 했다. 즉, pipe가 2개인 경우 첫 명령어와 마지막 명령어는 한쪽 끝만 연결되기 때문에 하나 적은 수의 pipe를 생성했다. 따라서 분석 후 명령어가 2개인 경우 경우 pipe를 1개 생성했고, 3개인 경우 pipe가 2개가 생성되도록 했다.  
 파이프를 생성한 뒤에는 child process를 생성하고, 각 child process의 표준 입출력을 알맞은 pipe의 읽기와 쓰기 끝에 연결했다. 첫 명령어는 표준 출력만 pipe의 쓰기 끝에 연결했고, 마지막 명령어는 표준 입력만 pipe의 읽기 끝에 연결하였다. 개수는 MAX\_COMMANDS를 통해서 4개까지 처리할 수 있도록 했다. 중간 명령어의 경우 표준 입력과 출력이 연속된 파이프에 연결되도록 했다.

**3. Phase3 (background process)**

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

먼저 명령어를 파싱했다. 그리고 ‘&’문자의 유무를 확인한 후 bg값을 설정했다. bg값이 1이면 background process, 0이면 foreground process이다. Background process 또한 fork를 통해 child process를 생성한다. 이후 bg 값을 확인한 다음 다른 처리를 주었다. Background process의 경우, parent process는 명령어를 처리하면서 process를 background에서 실행되도록 했다. 이때, add\_job 함수(직접 구현한 함수)를 호출해서 background process를 jobs 목록에 추가하고, 작업ID와 실행한 명령어를 출력하여 확인할 수 있도록 했다.

**C. 개발 방법**

우선 Base code로 csapp.c, csapp.h와 shellex.c(skeleton code)를 사용했다. csapp에 있는 기본 함수들이 있고, skeleton code 소스코드에 본 구현 내용을 추가하면서 myshell.c를 작성했다.

**1) phase1:** 프로젝트 내용에 맞는 기본적인 틀(Prompt,)을 main함수안에 완성했다. 그리고 eval함수에 ls, rmdir, mkdir, touch, echo, cat 명령어들을 인식하고 실행할 수 있도록 기능을 추가했다. 또한 eval함수에서 builtin\_coimmand(argv, cmdline)을 호출해서 내장 명령어가 아닌 경우에만 fork()를 사용하여 child process를 생성하고, 이후, child process에서 execve()함수를 사용해서 명령어를 실행하도록 구현했다. 또한 waitpid()를 사용해서 parent process가 child process가 종료될 때 까지 기다리도록 했다.

또한 history기능을 구현했다. Save\_history함수를 사용해서 cmdline으로 전달된 명령어를 사용했다. History\_count를 통해서 0이거나 마지막으로 저장된 이력과 현재 명령어가 다를 경우에 명령어가 저장되도록 하였다. 이렇게 하면, 동일한 명령어가 연속으로 저장되는 것을 방지할 수 있다. 만약 history 개수가 최대가 된다면, history 배열에서 첫 배열을 삭제 시키고, 이력들을 이동시키고, 배열의 마지막 위치에 현재 명령어를 저장하도록 했다. 이렇게 하여 배열이 무한히 저장될 수 있도록 구현했다. 마지막으로 history 목록은 history.txt파일에 저장해서 open, write, read 함수를 통해서 shell 종료후에도 기록이 유지될 수 있도록 구현했다.  
<사용한 함수 및 구조체>  
- void save\_history(char\* cmdline) // Save history function

- Typedef struct{

Char cmdline[MAXLINE]

} History; // history를 위한 구조체 선언 Cmdline을 받아서 history 정보를 저장

**2) Phase 2:** 우선 execute\_piped\_commands() 함수를 구현했다. 이는 pipe로 연결된 여러 명령어를 처리하도록 구현했다. Pipe로 연결된 명령어들을 저장하기 위한 2차원 문자열 배열 commands를 생성했고, pipe의 fd를 저장하기 위하여 pipefds 배열을 생성했다. 주어진 문자열 배열을 pipe를 기준으로 나누어서 commands배열에 저장하고, 필요한 만큼 pipe를 생성하도록 했다.

이후 child process를 생성해서 명령어를 실행하고, pipe를 통해서 연결했다. Dup2() 함수를 사용해서 fd를 복사하고, 첫 명령어가 아닌 경우 이전 파이프의 출력을 표준입력으로 받도록 했다. 마지막 명령어가 아니면 현재 pipe의 입력을 표준 출력으로 설정했다. 마지막에는 모든 pipe의 fd를 닫고 자식 process들이 종료될 때 까지 wait했다.

<구현한 함수 및 구조체>

- void execute\_piped\_commands(char\*\* argv) // pipe 실행을 위한 함수

**3) phase 3:** eval() 함수에서 먼저 background(&) 실행 여부를 판별하도록했다. Background 실행인 경우 실행결과를 출력하는 것이 아니고, 즉시 prompt를 반환하도록 했다. Foreground 실행인 경우, child process를 생성해서 명령어를 실행하고, 마찬가지로 parent process는 대기하도록했다. 이때 추가적으로 SIGINT(Ctrl + C)를 추가해 시그널 핸들러가 동작할 수 있도록 SIGINT\_Handler 함수를 추가하고 코드에 추가했다. Background process가 제대로 제어될 수 있도록 이외에도 SIGTSTP, SIGCHLD Handler 함수를 추가해서 Ctrl + Z키를 눌렀을 때, 프로세스를 정지시킬 수 있도록 코드를 추가했고, 자식 프로세스가 종료됐을 때, 멈춰진 상태를 parent process가 알도록 코드를 추가했다.

Add\_job 함수를 추가해 background에서 실행중인 process를 추적하기 위해서 jobs 목록을 유지할 수 있는 함수를 구현했다. 입력으로 받은 pid, status, cmdline 등의 정보를 바탕으로 하여 새로운 작업을 추가했다. Jobs 배열에서 비어 있는 작업을 찾아서 정보를 저장하고, 해당 작업의 번호를 반환했다.

Bg는 background에서 일시중지된 작업을 다시 시작시키는 함수로 구현했다. 만약 일시중지 되어있다면 SIGCONT signal을 해당 작업에 보내서 다시 시작할 수 있도록 구현했다.

Fg는 background에서 실행중인 작업을 foreground로 가져오는 함수로 구현했다. 주어진 job\_id가 유효한지 확인 후 해당 작업이 stop 상태인지 확인한다. 정지 상태면 SIGCONT signal을 보내서 다시 실행상태를 Running으로 변경될 수 있도록 했다. WUNTRACED flag를 사용해서 작업이 중지된 경우(SIGTSTP) 상태를 반환했다. 그렇지 않은 경우 해당 작업 pid를 0으로 설정해서 해당 작업이 완료된 것으로 표시했다.

Terminate\_job 함수를 구현해서 background에서 실행중인 작업을 강제로 종료시키는 함수를 구현했다. Jobs로 해당 작업을 확인하고, 강제로 종료하기 위해 SIgKILL signal을 해당 작업으로 보내서 강제로 종료될 수 있도록 했다.

List\_jobs 함수는 현재 background에서 실행중인 작업 목록을 출력하도록 구현했다. 현재 상태를 표현했고, [Num] Status “command” 로 출력되도록 했다.

FG, BG, KILL 명령어의 경우 bG %NUM로 실행할 수 있도록 구현했다.

그리고 fgpid, pid\_to\_jid 함수를 구현했다. Fgpid는 현재 실행 중인 foreground 작업의 PID를 반환하는 함수다. 이 함수는 jobs배열에서 state가 foreground인 작업을 찾아서 pid를 반환하도록 했다. 현재 foreground작업이 없으면 0을 반환하도록 했다. Pid\_to\_jid 함수는 PID를 JID로 mapping하도록 하는 함수다. jobs배열에서 해당 pid를 가지는 작업을 찾아서 그 작업의 id를 반환했다. 마찬가지로 해당 작업이 없으면 0을 반환했다.

또한 Shell에서 처리한 작업에 대한 정보를 저장할 수 있는 구조체를 설정했다. pid는 프로세스 ID, 작업 상태를 나타내주는 status, state, 작업 id를 표현하기 위한 id 변수를 설정했다.

<사용한 함수 및 구조체>

- void terminate\_job(int job\_id);

- void bg(int job\_id);

- void fg(int job\_id);

- void list\_jobs();

- Int add\_job(pid\_t pid, int status, char\* cmdline);

- void remove\_job(pid\_t pid);

- void sigchld\_handler(int sig);

- void sigtstp\_handler(int sig);

- void sigint\_handler(int sig);

- pid\_t fgpid(job\_t\* jobs);

- int pid\_to\_jid(pid\_t pid);

- /\* struct for job management \*/

typedef struct {

pid\_t pid; /\* Process ID \*/

int status; /\* JOB STATUS! 0: running, 1: stopped \*/

int state; /\* Job state \*/

int id; /\* Job ID \*/

char cmdline[MAXLINE]; /\* Command line \*/

} job\_t;

**3. 구현 결과**

**A. Flow Chart**

**1) Phase 1 (fork)**

텍스트, 빛이(가) 표시된 사진

자동 생성된 설명

**2) Phase 2 (pipeline)**

도표이(가) 표시된 사진

자동 생성된 설명

**3) Phase 3 (background)**

도표이(가) 표시된 사진

자동 생성된 설명