**System Programming Project 1**

1. **개발 목표**

이 프로젝트는 간단한 셸 프로그램을 구현하는 것으로, 사용자로부터 입력된 명령을 해석하고 실행하는 기능을 제공한다. 이를 위해 3개의 단계로 나누어 개발하며, 각 단계에서는 이전 단계에서 구현한 기능을 기반으로 더욱 확장된 기능을 구현해야 한다.

첫 번째 단계에서는 fork(), exec(), wait() 등의 시스템 콜을 사용하여 입력된 명령을 자식 프로세스에서 실행하는 기본적인 셸을 작성해야 한다. ‘cd’, ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’, ‘cat’, ‘echo’, ‘history’, ‘exit’와 같은 basic command가 작동할 수 있어야한다.

두 번째 단계에서는 redirection과 piping 기능을 추가해야 한다. 이를 위해, pipeline에 있는 각 명령을 위한 새로운 프로세스를 생성하고, 한 프로세스의 출력을 다른 프로세스의 입력으로 전달하는 기능을 구현해야 한다. 이를 위해 dup(), dup2() 등의 시스템 콜을 사용한다.

세 번째 단계에서는 백그라운드에서 프로세스를 실행할 수 있는 기능을 추가해야 한다. 이를 위해 job control 개념을 사용하여, 백그라운드와 포그라운드 간에 job 이동이 가능하도록 하고, job 내에 있는 프로세스의 상태를 변경할 수 있는 다양한 내장 명령을 구현해야 한다. 이를 위해 시그널에 대한 개념을 이해해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

Shell Program은 사용자의 명령을 받고, 입력 받은 command에 해당하는 작업을 수행한다. 사용자가 quit이나 exit를 입력할 때까지 프롬프트를 출력하며 무한루프 안에서 반복적인 입력을 받는다.

Phase1에서는 cd, ls, mkdir, rmdir, touch, cat, echo, history, exit 명령어를 수행할 수 있다. 이 명령어들은 기본적으로 원래 기능과 동일한 작업을 수행한다.

‘cd’ 명령어는 현재 작업 디렉토리를 변경할 수 있고, ‘ls’ 명령어는 현재 디렉토리에 있는 파일과 디렉토리의 목록을 출력할 수 있다. ‘mkdir’와 ‘rmdir’ 명령어는 새 디렉토리를 생성하거나, 빈 디렉토리를 삭제할 수 있다. ‘touch’ 명령어는 새로운 빈 파일을 생성하거나, 파일의 최종 수정 시간을 업데이트할 수 있고, ‘cat’ 명령어는 파일의 내용을 출력한다. ‘echo’ 명령어는 주어진 입력을 터미널에 출력하고, ‘history’ 명령어는 실행된 명령어들의 리스트를 출력한다.

Parent process가 사용자의 명령어를 해석하고, 명령은 child process로 실행된다. fork()를 통해 process를 생성하고, execve를 사용해 process를 새 program으로 실행된다.

1. Phase 2

Phase2에서는 이전에 구현한 phase1에 파이프라인(pipe) 기능이 추가되었다. Shell Program에서는 파이프 라인에 연결된 각각의 명령어를 처리하기 위해 새로운 자식 프로세스와 파이프를 생성한다. 파이프를 통해 앞쪽 명령의 출력을 뒤쪽 명령의 입력으로 전달한다.

명세서에 나와있는 예제를 보면, "ls -al | grep filename" 명령은 "ls -al"의 결과 중 "filename"이라는 이름을 가진 파일의 정보만을 출력한다. "cat filename | less" 명령은 "filename"이라는 파일의 내용을 "less"를 이용하여 보여준다. 마지막으로, "cat filename | grep -v “abc” | sort -r" 명령은 "filename"이라는 파일에서 "abc" 패턴이 없는 행만을 출력하고, 그 결과를 역순으로 정렬하여 출력할 수 있다.

1. Phase 3

Phase3의 Shell Program에서는 job control 기능을 제공하여 백그라운드에서 실행되는 프로세스를 제어할 수 있다. 이를 통해 메모리 누수를 방지하고 Foreground와 Background에서 모두 프로세스를 실행할 수 있게 된다.

Background에서 실행하고자 하는 명령어 끝에 ‘&’ 기호를 붙이면 해당 명령어가 백그라운드에서 실행되며, ‘jobs’ 명령어를 사용하여 백그라운드에서 실행 중인 job들의 상태를 확인할 수 있다.

‘bg’ 명령어를 사용하여 중지된 백그라운드 job을 실행 상태로 변경하고, ‘fg’ 명령어를 사용하여 백그라운드 job을 Foreground로 변경할 수 있다.

또한 ‘kill’ 명령어를 사용하여 job을 종료시킬 수 있다.

Ctrl-c를 입력하면 Foreground job의 각 프로세스에 SIGINT 시그널이 전달되어 해당 프로세스가 종료된다. Ctrl-z를 입력하면 Foreground job의 각 프로세스에 SIGTSTP 시그널이 전달되어 해당 프로세스가 중지 상태가 되며, SIGCONT 시그널을 받으면 다시 실행된다.

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

phase1에서는 shell program에서 입력 받은 명령어를 실행하며, 빌트인 명령어가 아니라면, fork()함수를 통해 자식 프로세스에서 명령어를 실행한다. 여기서 사용되는 fork() 함수는 부모 프로세스에서 현재 실행중인 프로세스와 똑 같은 프로세스를 하나 더 생성한다. 따라서 자식 프로세스에서 실행되는 명령어는 부모 프로세스와 독립적으로 실행되며, 명령어 실행이 끝난 후에는 자식 프로세스는 종료된다.

자식 프로세스와 부모 프로세스의 구분은 pid값을 통하여 구분하고, 반환된 pid값이 0이면 자식 프로세스로, 자식 프로세스에서 실행되는 명령어는 execve()함수를 통해 실행된다. Execve() 함수를 호출하기 전에, “/bin/” 디렉토리에 해당 명령어가 있는지 먼저 확인하고, 없는 경우 “/usr/bin/”디렉토리에서 다시 한 번 확인하는 과정을 거쳤다. 이 과정에서 명령어가 없을 경우 에러 메시지를 출력하고, exit(0)을 호출하여 자식 프로세스를 종료한다.

자식 프로세스가 실행되는 동안, 부모 프로세스는 foreground job이라면 자식 프로세스가 종료될 때까지 기다린다. Background job일 경우에는, 자식 프로세스를 생성하고 출력으로 해당 자식 프로세스의 pid와 명령어를 출력하도록 하였다.

이렇게 생성된 자식 프로세스는 명령어 실행을 끝내면 종료되며, 부모 프로세스는 다음 명령어를 입력받을 준비를 하게된다.

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

프로세스가 자식 프로세스를 생성할 때 sigprocmask를 통해 SIGCHLD 시그널을 block하고, sigprocmask를 통해 다시 unblock하게 된다. 이렇게 되면 자식 프로세스가 종료되면 커널이 부모 프로세스에 SIGCHLD 시그널을 보내게 되고, 이 시그널은 ‘sig\_child\_handler’ 함수에서 처리된다.

‘sig\_child\_handler’ 함수는 while문을 통해 ‘waitpid’ 함수를 이용하여 자식 프로세스가 종료되었는지 확인하고, 종료된 경우에는 ‘WIFEXITED’, ‘WIFSTOPPED’, ‘WIFSIGNALED’를 통해 각각 정상적으로 종료되었는지, 중단되었는지, signal에 의해 종료되었는지 확인할 수 있도록 하였다. 이후 ‘fg\_end\_flag’ 변수를 통해 부모 프로세스가 처리할 수 있도록 flag를 설정하고, 부모 프로세스가 이를 처리할 수 있도록 하였다.

따라서, 자식 프로세스가 종료되면 커널이 SIGCHLD 시그널을 보내고, 이를 ‘sig\_child\_handler’함수가 처리하여 부모 프로세스가 종료된 자식 프로세스에 대한 정보를 받아올 수 있다.

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

‘pipe\_execution’ 함수를 이용해, 파이프를 사용하는 명령어가 입력되면 ‘pipe\_execution’ 함수를 호출해 파이프 기능을 구현한다.

기본적인 구조는 eval 함수에서 사용자 입력을 받아서 파싱하여 명령어를 구분한 뒤 builtin\_command 함수로 내장 명령어를 처리한다. 만약 내장 명령어가 아닌 일반 명령어가 입력되면 pipe\_execution 함수를 호출하여 파이프 기능을 실행한다.

파이프 기능을 구현하는 방법은 pipefd 배열에 파이프를 생성하고, 자식 프로세스를 생성하여 pipefd 배열의 파일 디스크립터를 이용하여 입력과 출력을 연결하였다. 이때 자식 프로세스에서 실행되는 명령어들은 이전 명령어의 출력이 다음 명령어의 입력으로 들어가도록 설정하여 파이프 기능을 구현한다.

pipefd라는 2D 배열을 사용하여 파이프에 대한 파일 디스크립터를 저장하고 있다. pipe() 함수를 사용하여 파이프를 생성하고, 생성된 파이프의 읽기/쓰기 파일 디스크립터를 pipefd[i][0]과 pipefd[i][1]에 각각 저장하고 있다.

이후 dup2() 함수를 사용하여 파이프에 대한 파일 디스크립터를 복제하고 있다. 예를 들어, i가 0일 때 dup2(pipefd[0][1], STDOUT\_FILENO)를 호출하면, 파이프의 쓰기 파일 디스크립터인 pipefd[0][1]을 STDOUT\_FILENO에 복제하게 된다. 이러한 작업을 통해, execvp() 함수가 실행되는 시점에서 파이프로 출력을 전달하거나 입력을 받을 수 있게 된다.

파이프 기능이 실행되는 동안 자식 프로세스의 상태 변화를 감지하기 위해 sig\_child\_in\_pipe\_handler 함수가 등록되어 있고, 이 함수는 waitpid 함수를 이용하여 자식 프로세스가 종료되거나 멈춘 경우 상태 정보를 얻어내고, 이 정보를 이용하여 파이프 기능이 실행 중인지를 판단할 수 있다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

파이프 라인에 포함된 명령어의 개수(p\_command\_cnt)에 따라 다음과 같이 처리하였다. 먼저, 각 명령어를 파싱하여 실행에 필요한 인자를 배열에 저장하고, "/bin/" 경로를 argv[0]에 추가하여 명령어를 실행할 수 있는 경로를 지정하였다. 그리고 파이프 라인에 사용할 파일 디스크립터를 생성하고, 파이프 라인이 몇 개의 명령어로 이루어져 있는지에 따라 2차원 배열 pipefd를 적절하게 설정하였다.

첫 번째 명령어인 경우, STDOUT\_FILENO을 이전 파이프 라인의 write end로 변경하고, 이전 파이프 라인의 read end는 닫고, 현재 파이프 라인의 write end는 닫는다.

마지막 명령어인 경우, STDIN\_FILENO을 이전 파이프 라인의 read end로 변경하고, 이전 파이프 라인의 write end는 닫고, 현재 파이프 라인의 read end는 닫는다.

중간에 위치한 명령어인 경우, 이전 파이프 라인의 read end를 STDIN\_FILENO로 변경하고, 현재 파이프 라인의 write end를 STDOUT\_FILENO로 변경한다. 이전 파이프 라인의 read end와 현재 파이프 라인의 write end는 모두 닫는다.

마지막으로, execve()를 이용하여 명령어를 실행하고, 만약 명령어를 찾을 수 없는 경우, "/usr/bin/" 경로를 추가하여 다시 시도하고, 여전히 실패하는 경우에는 오류 메시지를 출력하고 자식 프로세스를 종료한다. 이 과정을 반복하여 모든 명령어를 실행하면서 파이프 라인을 연결하였다.

모든 명령어가 실행될 때까지 대기하기 위해, sig\_child\_in\_pipe\_handler()에서 pipe\_end\_flag를 1로 설정한 이후, while 루프를 사용하여 pipe\_end\_flag가 0일 동안 대기한다. pipe\_end\_flag가 1이 되면 while 루프를 탈출하고, pipe\_end\_flag를 0으로 초기화하고 자식 프로세스를 종료한다.

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

Phase1에서는 리눅스 기반의 커맨드 라인 쉘을 구현한다. 이 쉘은 사용자 입력을 받아, 명령어를 인식하고 해당 명령어를 실행시킨다.

먼저 사용자 입력을 받아 명령어와 인자를 분리하고, 이를 execve() 함수를 이용하여 실행시킨다. 이 코드는 몇 가지 다른 함수를 사용하여 작동한다. 이러한 함수는 아래와 같다.

void eval(char \*cmdline) : 사용자가 입력한 명령어를 평가하여 실행시킨다.

int parseline(char \*buf, char \*\*argv) : 사용자 입력 문자열을 명령어와 인자로 분리한다.

int builtin\_command(char \*\*argv) : 입력 명령어가 내부 명령어인지 확인하고, 내부 명령어라면 실행시킨다.

void add\_to\_history(char\* cmdline) : 사용자가 입력한 명령어를 history 배열에 저장한다.

void print\_history() : history 배열에 저장된 명령어를 화면에 출력한다.

void ncmdline(char\* cmdline) : 사용자 입력 문자열을 분석하여 명령어와 인자 사이의 공백을 처리한다.

void init\_sig() : 시그널 핸들러를 초기화한다.

void sig\_int\_handler(int sig) : SIGINT 시그널을 처리하는 핸들러이다. foreground 프로세스가 실행 중인 경우 SIGINT를 전달한다.

Void sig\_stop\_handler(int sig) : SIGTSTP 시그널을 처리하는 핸들러이다. foreground 프로세스가 실행 중인 경우 SIGTSTP를 전달한다.

void sig\_child\_handler(int sig) : SIGCHLD 시그널을 처리하는 핸들러이다. child 프로세스가 종료되었을 때, 해당 child 프로세스를 대기 상태에서 반환하고 status 정보를 출력한다.

Phase1의 주요 기능은 사용자가 입력한 명령어를 인식하여 실행시키도록 하는 것이다. 명령어는 내부 명령어와 외부 명령어로 구분되고, 내부 명령어는 해당 쉘의 기능을 수행하고, 외부 명령어는 fork() 함수를 이용하여 새로운 프로세스를 생성한 후, execve() 함수를 이용하여 명령어를 실행시킨다. 여기서 fork()함수의 반환값을 pid 변수에 저장하도록 하여, pid가 0인 경우에는 자식 프로세스를 의미하기 때문에 execve() 함수를 호출해 프로세스를 실행할 수 있도록 하였고, pid가 0보다 큰 경우는 부모 프로세스이므로, waitpid 함수를 이용해 자식 프로세스의 종료를 기다리고 reaping할 수 있도록 하였다. 또한, 명령어 실행 후, foreground와 background 모드를 구분하여 실행시키며, 프로세스의 상태 변화를 signal handler를 이용하여 처리한다.

* **Phase2 (pipelining)**

Phase2에서는 여러 개의 명령어가 파이프로 연결되어있을 때, 각각의 명령어를 자식 프로세스로 실행하고 파이프를 통해 연결하는 piepelining를 구현해야 한다.

먼저, 파이프라인을 구현하기 위해서 eval()함수를 수정하였다.

pipe\_flag: 파이프 라인이 있는지 여부를 저장하는 변수를 추가하였다.

p\_command: 파이프(|)를 기준으로 나눈 각각의 명령어를 저장하는 배열을 추가하였다.

p\_command\_cnt: 파이프(|)로 구분된 명령어의 개수를 저장하는 변수를 추가하였다.

p\_start: 각각의 명령어의 시작 지점을 저장하는 변수를 추가하였다.

buf2: 파이프 라인을 처리하기 위해 cmdline을 복사한 임시 변수를 추가하였다.

pipe\_execution 함수: 파이프 라인을 실행하는 함수가 추가하였다.

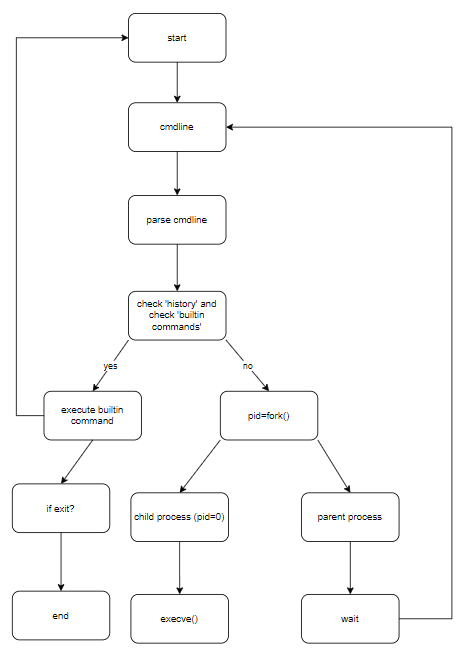
또한, Fork 함수를 사용하여 자식 프로세스를 생성하고, 파이프 라인이 있는 경우와 없는 경우를 나누어 처리하였다. 파이프 라인이 있는 경우, 파이프 라인을 기준으로 각각의 명령어를 나누어 p\_command 배열에 저장하고, pipe\_execution 함수를 호출하여 파이프 라인을 실행한다. 파이프 라인이 없는 경우에는 execve 함수를 호출하여 명령어를 실행한다.

Pipe\_execution 함수는 파이프를 이용한 명령어 실행을 수행하는 함수이다. 함수는 명령어 문자열 배열과 명령어 개수를 인자로 받는다. 함수는 먼저 필요한 변수들을 초기화하고, SIGCHLD 신호를 처리하기 위한 핸들러를 등록한다. 그 다음, 입력받은 명령어 문자열을 파이프(|)를 기준으로 나누고, 파이프를 사용하여 명령어를 실행한다.

각 명령어를 실행하기 전에, "/bin/" 디렉토리에서 해당 명령어를 찾을 것으로 예상하고 argv[0]을 "/bin/"으로 초기화한다. 그리고 파이프를 생성하고, 현재 명령어가 첫 번째인지, 마지막인지, 아니면 중간에 있는 명령어인지를 확인한다. 각 경우에 맞게, stdin과 stdout을 파이프의 read end와 write end로 리디렉션하고, 필요한 파일 디스크립터들을 닫는다.

마지막으로, execve를 이용하여 명령어를 실행하고, 만약 명령어가 실행되지 않는다면, "/usr/bin/" 디렉토리에서 해당 명령어를 찾을 것으로 예상하고, argv[0]을 "/usr/bin/"으로 초기화한다. 그리고 다시 execve를 호출하여 명령어를 실행합니다. 만약 명령어가 아직 실행되지 않는다면, "Command not found"라는 에러 메시지를 출력하고, 모든 명령어가 실행되면, 함수는 exit(0)을 호출하여 종료된다.

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



1. **Phase 2 (pipeline)**

