**System Programming Project 2**

담당 교수 : 박성용

이름 : 이도안

학번 : 20201610

1. **개발 목표**

**RINUX의 shell을 모방하여 c언어와 내장 함수를 사용하여 최대한 유사하게 만든다.**

**Phase 1, 2, 3으로 나누어 간단한 입력문들부터 백그라운드 실행, 파이프 등 전체적인 맥락의 쉘을 구현하고 테스트하며 실제 쉘과 가깝게 수정한다.**

**Fork()를 사용해서 exec계열 함수를 사용해도 현재 돌아가고 있는 myshell이 종료되지 않으면서 새로운 child에서 명령을 실행할 수 있게 한다.**

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

Myshell을 실행했을 때 무한루프가 돌며 매회 입력을 받기 시작한다.

cd, ls, exit 등의 간단한 기초 쉘 입력문들을 받아 실제 쉘처럼 동작하며, 이 쉘에서 실행한 mkdir, touch과 같은 실제 디렉토리에 영향을 주는 입력문들 또한 실제 디렉토리에 직접 반영할 수 있다.

1. Phase 2

Redirection 과 pipe를 구현한다.

(명령문 문장) | (명령문 문장) …. 을 실행 시 앞의 명령문 문장의 출력 결과물이 뒤의 input으로 들어가게 된다. 예를 들어, cat filename | grep “abc” 를 실행했을 때, filename의 내용들이 cat으로 출력되는데, 이게 stdout으로 출력되지 않고 뒤의 명령문 문장의 input으로 들어가 abc가 포함된 내용을 찾아 출력해주게 된다.

여기서 “”의 유무 또한 걸러지게 돼서 모든 경우에 출력이 가능하다.

1. Phase 3
   1. **개발 내용**

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

builtin\_command 에 포함된 입력이 명령으로 들어올 경우, fork 하지 않고 myshell에서 자체적으로 수행하여 출력한다.

포함되지 않은 경우에 fork()를 수행함과 동시에 pid 변수에 pid값을 저장해주며, pid==0을 만족시킬 경우에 child process를 수행한다는 뜻이 되고, 반대의 경우에는 parent process가 수행되고 있다고 판단할 수 있다. Fork 하여 child process를 생성하면, 부모와 독립적인 새로운 프로그램이기 때문에 execvp를 수행하여 여러 명령을 처리한다. 부모는 execvp를 수행하는 child를 reaping하기 위해 wait한다.

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

connection을 종료할 때 child는 부모 프로세스에게 sigchld 시그널을 보낸다.

메인 함수에서 Signal(SIGCHLD,sigchld\_handler); 를 실행하여 signal handler를 통해 signal handling을 해 준다. sigchld\_handler는 “sigchld” Signal을 받았을 때 대기하고 있던 parent에게 waitpid 명령을 실행하도록 한다.

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

현재 명령문에서 ‘|’ 가 존재할 경우, eval 함수에서 |를 경계로 각각의 명령문 문장을 파싱한다. 파싱된 문장은 2차원 문자열 배열 형태로 저장되며, exec\_pipe 함수로 전달된다. 이 함수에서는 fork를 해 주고, pipe 함수를 통해 fd[2]에 파일 디스크립터를 배정해준다. 마지막 명령문을 제외한 명령문들은 pipe input 부분을 닫아주고, stdout 대신에 pipe out을 dup2 함수를 통해 할당해준다. 그리고 이 pipe out 부분은 닫아준다.

Pipe out으로 전달된 출력은 부모의 pipe in으로 전달되며, 이것은 부모의 실행 부분에서 in변수에 저장된다. 저장된 출력이 다음 생성된 child의 input으로 돌려주며 이를 반복하여 pipe 동작을 수행한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Pipeline 개수에 따라 별도로 handling하지 않았다. 0개인 경우 기본적인 실행을 해주었고, 1개 이상이면 pipe\_exec 함수를 실행했다. 파이프라인 명령문에서 마지막 실행문을 제외한 모든 경우에는 출력을 pipe의 fd에 넣어주었고, 마지막의 경우에는 stdout으로 출력할 수 있도록 설계했다.

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

명령문의 마지막 문자에 &이 입력으로 들어오면 그 부분을 공백으로 처리함과 동시에 bg라는 flag를 1로 설정하였다. 그 아래의 모든 처리문(eval, pipe\_exec)에서 bg==1이면 wait을 사용하지 않고 곧바로 다음 반복문을 수행한다(부모). Bg==0이면 포그라운드 실행으로, sigchld를 unblock하고 pause해주는 Sigsuspend를 사용해서 child를 wait해준다.

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

모든 구현은 myshell.c 에 추가 및 수정하였다.

**Phase 1 :**

main에서 sigint, sigtstp, sigchld에 대한 시그널 핸들링을 추가한다. Eval 함수 내에서 “cd” 명령어에 대한 수행문을 추가하고 이 경우에 eval은 리턴된다(void).

그리고 bulit-in 명령어가 아닌 경우, fork()를 수행해 fork==0인 자식 프로세스에서 execvp를 수행하여 명령을 수행한다. 자식이 종료될 때 sigchld handling을 통해 종료되었다는 시그널을 받으면 void sigchld\_handler 함수에서 부모의 wait을 처리해준다.

**Phase 2:**

‘|’ 문자가 입력으로 들어오면 ‘|’를 경계로 파싱하여 2차원 문자열 배열을 만들고, 이를 인자로 받아 exec\_pipe가 실행하도록 추가했다.

Void exec\_pipe() 함수는 앞의 명령문부터 순차적으로 처리하며 뒤로 전달하며 마지막 출력은 stdout으로 출력되는 함수이다. pipe 함수를 통해 fd[2]에 파일 디스크립터를 배정해준다. 마지막 명령문을 제외한 명령문들은 pipe input 부분을 닫아주고, stdout 대신에 pipe out을 dup2 함수를 통해 할당해준다. 그리고 이 pipe out 부분은 닫아준다.

Pipe out으로 전달된 출력은 부모의 pipe in으로 전달되며, 이것은 부모의 실행 부분에서 in변수에 저장된다. 저장된 출력이 다음 생성된 child의 input으로 돌려주며 이를 반복하여 pipe 동작을 수행한다.

**Phase 3:**

추가된 함수:

int get\_max(proc \*jobs);

현재 실행중인 process 중 가장 순번이 큰 process의 pid를 반환.

int get\_index(proc\* , pid\_t);

pid에 해당하는 process 의 순번을 반환.

int get\_fg\_pid(struct proc \*jobs);

현재 포그라운드에서 실행중인 process의 pid를 반환.

void sigint\_handler(int sig);

Ctrl+C 핸들링. 현재 fg에 돌아가고 있는 process를 kill 하고 jobs에서 삭제.

void sigtstp\_handler(int sig);

Ctrl+Z 핸들링. 현재 fg에 돌아가고 있는 process를 stop 하고 state를 변경.

void sigchld\_handler(int sig);

child 상태 변화 핸들링. Child에 상태 변화에 따른 행동 추가. 종료되면 해당 pid를 0으로 만들어 jobs 목록에서 삭제함.

추가된 구조체 :

typedef struct proc{

    int pid; 프로세스의 pid

    char cmd[50]; 프로세스의 명령문

    int state; 프로세스의 상태(running or suspended)

    int fb; 백그라운드에서 실행되고 있는지 flag

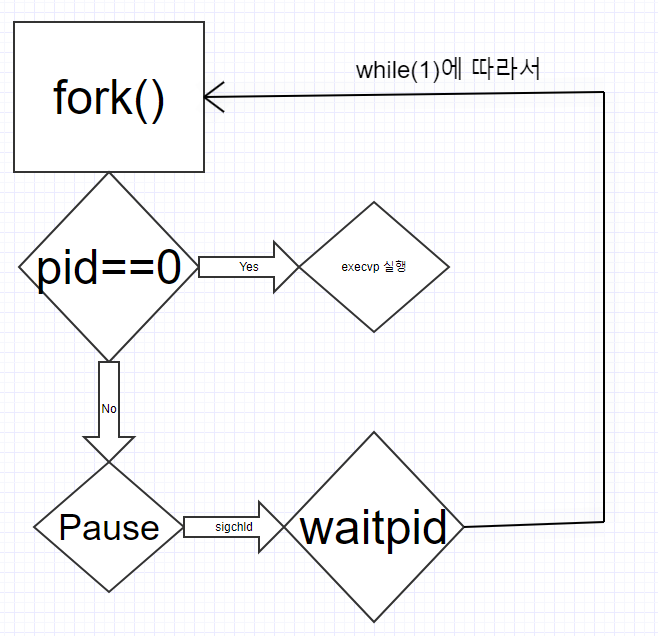
}proc;

이 구조체는 jobs 수행이나, bg, fg, kill 등 명령어에서 process list 관리를 위해 사용됨.

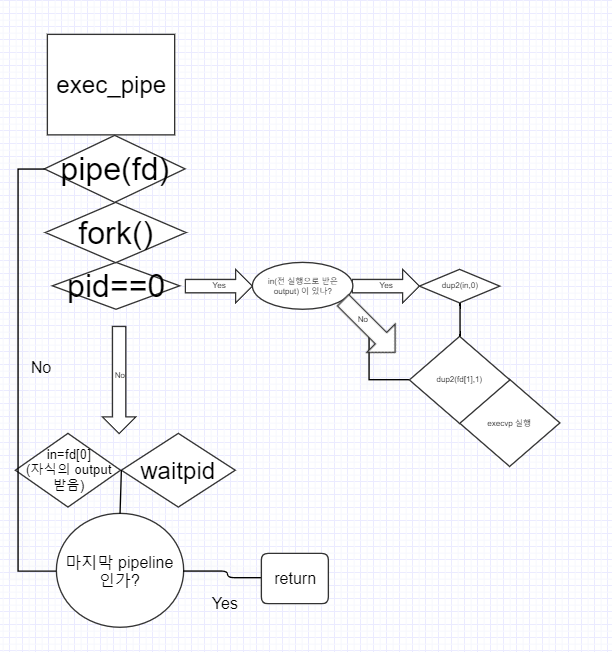
이 프로그램에서는 리스트로 구현하지 않고 정적 배열로 구현했음.

명령문 문장의 끝에 &가 추가되면 bg=1이 되고, 이를 통해 bg에서 실행하면 wait을 하지 않는 것으로 수행.

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



1. **Phase 2 (pipeline)**



1. **Phase 3 (background)**

