**System Programming Project 4**

담당 교수 : 김영재

이름 : 전민수

학번 : 201716867

1. **개발 목표**

* **Phase1,2,3에 걸쳐 더 많은 기능을 구현할 수 있도록 한다.**
* **Phase1에서는 가장 기본적인 명령어들만 수행할 수 있는 쉘을 만든다.**
* **Phase2 에서는 파이프라이닝을 지원하는 쉘을 만든다.**
* **Phase3에서는 백그라운드 작업을 수행하게 하는 명령어(bg, &), 백그라운드 작업을 포그라운드 작업을 바꾸는 명령어(fg), 시그널들을 핸들링(sigint, sigtstp, sigchld), 수행되고 있는 명령어를 죽이는 명령어(kill)를 활용할 수 있는 쉘을 만든다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

기본적인 명령어(cd, ls, mkdir, exit, touch, cat, echo 등)을 입력하였을 때, 그 명령어에 맞춰 명령어를 실행한다.

1. Phase 2

파이프 라이닝을 통해 이전 파이프의 출력을 다음 파이프의 입력으로 넘긴다.

기본명령어들을 조합하여 좀 더 많은 명령들을 수행할 수 있게 된다.

1. Phase 3

백그라운드 작업을 수행할 수 있다.(bg, &)

백그라운드 작업이 끝나면 foreground로 작업이 끝났음을 알려준다. (sigint handler)

현재 foreground 및 background에서 돌아가고 있는 작업들의 리스트를 관리한다. (fg, bg, jobs, kill)

백그라운드 또는 멈춰있는 작업을 foreground로 올릴 수 있다. (fg)

컨트롤 c를 입력받았을 때, childprocess를 강제종료하고 컨트롤 z를 입력받았을 때 프로세스를 suspend시킨다.

* 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

child process는 생성될 때 부모 프로세스의 가상 메모리 공간을 모두 동일하게 복제한다.

child process내에서는 fork의 return value가 0이고 부모프로세스에게는 return value가 0이 아니기 때문에, 구분해서 child process의 부분을 처리해줄 수 있다.

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

child process가 죽을 때 부모는 child process가 죽을 때까지 wait함수를 통해 기다리고 있는다. 이후, child process가 죽었을 때 child process의 여러 system resource들을 reaping해준다.

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

dup2를 통해

* 0번 프로세스 : 출력을 pipe 0번의 한쪽 끝(0+1)으로 해준다.
* 1번 프로세스 : 입력을 pipe 0번의 다른쪽 끝(0+0)으로 해준다.

출력을 pipe 1번의 한쪽 끝(2\*1 + 1)으로 해준다.

* 2번 프로세스 : 입력을 pipe의 1번의 다른쪽 끝(2\*1 + 0)으로 해준다.

출력을 pipe의 2번의 한쪽 끝(2\*2 +1)으로 해준다.

…

* 마지막 프로세스 : 입력의 pipe n-1번의 다른쪽 끝(2\*(n-1))로 해준다.

파이프를 이렇게 다 구성을 해줬다면, 모든 파이프를 다 닫고 execvp를 수행해준다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

일반화 하여 파이프라인을 연결해줬기 때문에, 파이프라인 개수에 크게 영향을 받지 않는다. 다만, 파이프라인을 만드는 배열의 크기가 한정적이기 때문에, 파이프라인은 15개까지로 한정해두었다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
* 먼저 &로 입력이 들어온 경우.

부모 프로세스에서 wait를 안하고 프로세스를 진행하다가 sig child 시그널이 들어왔을 때 처리를 해주면 된다.

* sigstop으로 인해 child process가 멈춘경우

sigchild 핸들러로 자식 프로세스가 sigstop에 의해 죽었다는 사실이 들어온다.

이 때 프로세스를 멈췄다는 처리를 하면 된다.

* 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 (fork & signal)**
* void eval(char \*cmdline);
* 실제로 커멘드 라인을 수행하는 함수이다.
* int parseline(char \*buf, char \*\*argv);
* buffer에 입력받은 커맨드 라인을 띄어쓰기 단위로 argv로 파싱을 해준다.
* int builtin\_command(char \*\*argv);
* 빌트인 커맨드를 처리해주는 함수이다.
* int cd(char\* path);
* 빌트인 cd를 처리하기 위한 함수이다.

fork를 하나 띄우고 fork를 처리하는(자식 프로세스를 처리하는)부분은 마지막 부분에 exit 0를 넣어서 부모와 자식의 프로세스가 겹치지 않도록 한다.

부모 프로세스에서는 wait를 하여 자식 프로세스가 foreground에서 돌고, 다 끝났으면 reaping될 수 있도록 한다.

* **Phase2 (pipelining)**
* int parse\_pipe(char\*\* argv, char\* argv\_for\_pipe[30][128])
* 띄어쓰기 기준으로 파싱 된 argv를 파이프까지 파싱해주기 위해 선언한 함수이다.
* void pipe\_line(char \*argv\_for\_pipe[30][128], int argv\_idx, int bg, char\* cmdline){
* 파이프를 처리하기 위한 함수이다. argv내에 pipe가 있다면 파이프를 처리해주기 위해 선언하였다.

pipe line에서는 int pipes[30];을 선언함으로써 파이프 총 15개를 사용하였다.

* **Phase3 (background process)**
* typedef struct {
* pid\_t pid; //job의 process id
* int jid; // job의 job id
* int status; // job이 bg인가 fg인가 stop인가
* char args[100]; // job을 실행할 때 들어온 인자
* int used; //지금 사용되고 있는 job인가
* int job\_num; // 파이프라인일때 이 job을 언제 delete할 수 있을까
* } job\_type;
* job을 관리하기 위해 job을 관리하는 struct를 만들었다. 각각은 주석의 내용의 역할을 담당한다.
* int JID = 1;//현재 값이 들어가있는 가장 큰 JID를 설정한다
* job\_type jobs[MAXJOBS];
* int joblen = 0;
* void init\_job();
* void addjob(pid\_t pid, int bg, char\* cmdline, int jobnum);
* int deletejob(pid\_t pid);
* void print\_jobs();
* //job에 관련된 연산을 수행한다.
* void sigchild\_handler();
* void sigint\_handler();
* void sigstop\_handler();
* 시그널들을 관리하기 위한 시그널 핸들러이다
* int main\_pid; //입력을 다시 메인으로 돌리기 위해 메인 그룹 pid를 가지고 있는다
* void bg(int); //bg command를 처리하기 위함이다
* void fg(int); //fg comand를 처리하기 위함이다
* sigjmp\_buf sigjmp;//시그널 점프를 하여 main의 while loop가장 위로 돌아오기 위함이다. 메인프로세스에서 sigint가 들어왔을 때 수행한다.
* struct termios main\_terminal\_attr; //메인 터미널 어트리뷰트를 저장해둔다.

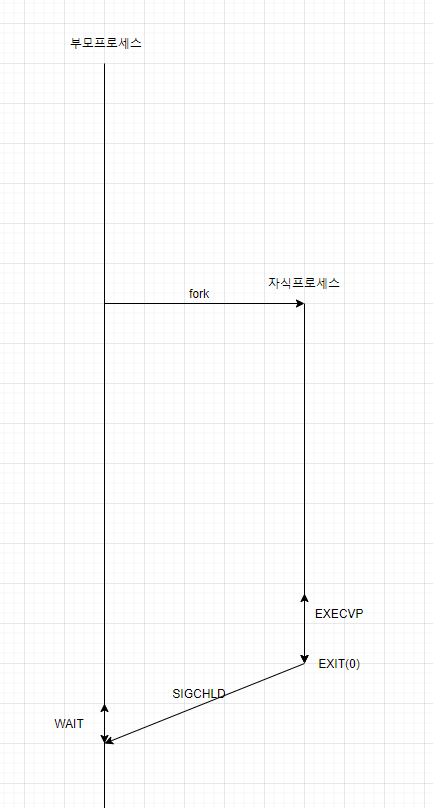
main pid, main\_terminal\_attr은 보통 때에는 필요가 없지만, python이나 less와 같은 stdin을 자신의 프로세스로 가져오는 명령어들의 경우 stdin을 다시 메인프로세스로 돌려놔야 진행이 되기 때문에 필요하다.

보통의 시그널들은 sigchld핸들러로 모두 구현이 된다(차일드 핸들러는 default handler를 통해 구현하였음).

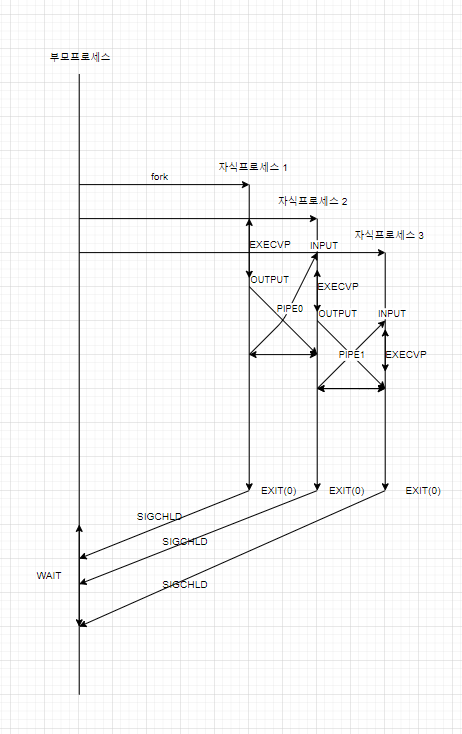
1. **구현 결과**
   1. **Flow Chart**

* **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)**



1. **Phase 2 (pipeline)**



1. **Phase 3 (background)**

