

Shell 구현

프로젝트 과제 최종보고서



12181667 이지율

요구사항 1. cd 명령이 제대로 먹히지 않는 버그를 수정하시오.

1) Pseudocode

```
void cmd_cd(){
    input: int cmdNum (command line vector의 배열 길이),
           char **cmdvector(Command line vector)

    IF (cmdNum is 0)
    THEN{
        //cd 뒤에 디렉토리가 따로 입력되지 않았을때, 경로 변수에 HOME 환경변수 값을 대입해준다.
        path<-Home Environment variable
    }
    ELSE{
        //이동할 디렉토리 이름을 경로 변수에 대입해준다.
        path<-cmdvector
    }

    IF(chdir(path) returns 0)
    THEN{
        //입력받은 디렉토리가 존재하지 않으면 오류메세지 출력
        print("there is no such directory!");
    }
}
```

2) 구현 방법

: 셸은 프로그램을 실행하기 위한 새로운 자식 프로세스를 생성하고, 해당 프로그램은 `execvp`를 호출하여 프로세스로 프로그램을 로드한 후, 명령을 실행하게 됩니다. 따라서 프롬프트에 커맨드를 입력하여 실행되는 프로그램들은, 모두 셸의 자식 프로세스에서 실행되고 있는 상태입니다. 하지만 현재 우리가 원하는 것은 셸 자체, 즉 부모 프로세스에서 현재 디렉토리를 이동하는 것입니다. 따라서 `cd` 명령어를 제대로 실행하기 위해서는 자식 프로세스에서 디렉토리를 이동하는게 아닌, 따로 부모 프로세스에서 디렉토리를 이동할 수 있도록 자식 프로세스를 `fork`하기 이전에 `cd` 명령 프로그램을 따로 구현해 주어야 합니다.

<main 함수 부분>

```
cmdline[strlen(cmdline) -1] = '\0';
//delim 공백을 기준으로 단어를 잘라낸다.

int cmdNum= makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG);
```

```

if(strcmp(cmdvector[0],"cd")==0){
    cmd_cd(cmdNum,cmdvector);
}

```

우선 자식 프로세스를 생성하기 이전에 부모 프로세스 실행 단계에서 커맨드를 처리해야하기 때문에, fork() 이후에 실행되던 makelist() 함수를 위부분으로 끌어올려 실행하였고, command line vector의 개수를 cmdNum이라는 변수로 리턴받았습니다.

그리고, cmdvector 배열의 첫번째 문자열을 "cd" 명령어와 비교하여, 만약 cd를 입력받았다면, cmd_cd() 함수를 실행합니다. 매개변수로는 cmdvector 배열의 요소 개수, 그리고 cmdvector 문자열 배열을 전달합니다.

<cd 함수 구현부>

```

void cmd_cd(int argc, char **argv){
    char *path;

    if(argc==1){
        //cd만 입력되고 디렉토리는 입력 안되었을 때
        path=(char*)getenv("HOME");
    }
    else{
        //path값에 디렉토리 이름을 대입한다.
        path=argv[1];
    }

    if(chdir(path)){
        //chdir 함수가 해당 path가 존재하지 않으면 0을 리턴함.
        printf("No such directory!");
    }
}

```

매개변수로 int argc와 char **argv를 생성하였습니다. 이 변수들은 각각 cmdvector의 배열 length와 cmdvector를 전달받습니다. 이제 이동할 directory를 결정할 경로 값을 담는 path 변수에 값을 대입해주어야 합니다. 만약 argc 값이 1, 즉 cd 명령어 하나만 입력되었다면 path값은 home 환경변수 값을 대입하게 되고, 디렉토리 값이 입력되었다면 path값에 해당 디렉토리 이름을 대입합니다

그 이후에, 현재 작업 디렉토리를 이동해주는 chdir함수를 통해서 path값을 전달하게 됩니다.

Chdir 함수는 입력받은 디렉토리가 존재하지 않아 에러가 발생하면 0을 리턴하게 되는데, 따라서 path 디렉토리가 존재하지 않는 에러가 발생하면, "No such directory!" 라는 경고 문구를 출력하도록 하였습니다.

3) 테스트 결과

1) 존재하는 디렉토리를 입력한 경우

```
myshell> pwd
/home/jiyullee/문서/Shell Project
myshell> cd dirA
myshell> pwd
/home/jiyullee/문서/Shell Project/dirA
myshell> cd ..
myshell> pwd
/home/jiyullee/문서/Shell Project
```

⇒ 디렉토리가 존재할 경우, 이동이 잘되는 것을 확인할 수 있습니다.

2) Cd 명령어만 입력하고 디렉토리를 입력하지 않은 경우

```
myshell> cd
myshell> pwd
/home/jiyullee
```

⇒ Home 환경변수로 지정된 값인 /home/jiyullee로 바로 이동하는 것을 확인할 수 있습니다.

3) 존재하지 않는 디렉토리 값을 입력했을 때

```
myshell> cd dirC
No such directory!
```

⇒ No such directory!라는 오류 메시지가 뜨는 것을 확인할 수 있습니다.

요구사항 2. Exit 명령을 구현하시오

1) Pseudocode

```
IF((strcmp(cmdvector),"exit")==0)
THEN{
    //입력받은 명령어가 exit과 일치하면 exit 함수 호출
    exit;
}
```

2) 구현 방식

```
int cmdNum= makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG);
```

```
else if (strcmp(cmdvector[0],"exit")==0){
    exit(1);
}
```

c언어에서 `exit()` 함수는 현재 실행되고 있는 프로그램 자체를 완전히 종료하는 기능을 갖고 있습니다.

따라서 프롬프트로 입력받은 값이 `exit`라면, `exit` 문자열과 비교하여 실행중인 모든 프로세스를 종료하고 현재 실행중인 셸 프로그램을 종료하게 됩니다.

3) 테스트 결과

```
myshell> exit
jiyullee@jiyullee-VirtualBox:~/문서/Shell Project$
```

셸 프로그램 실행중에 프롬프트에 `exit` 명령어를 입력하면 바로 셸이 종료됩니다

요구사항 3) 백그라운드 실행을 구현하시오.

1. Psuedocode

```
//background에서 작업을 실행하는 문자 &이 입력되었는지를 체크하기 위한 변수
int backgroundcheck=0;
//cmdvector에 입력된 명령어 벡터 길이
cmdNum <- cmdvector.length

//&는 보통 마지막 명령어로 입력되기 때문에, cmdvector의 마지막 문자열을 &와 비교한다.
IF(strcmp(cmdvector[cmdNum-1],"&")==0)
THEN{
    | //&이 존재하면 backgroundcheck 변수를 1로 세팅한다.
    backgroundcheck=1;
    cmdvector[cmdNum-1]<-NULL;
}

switch(pid=fork()){
    case 0:
        execvp();
    case -1:
        fatal();
    Default:
        //backgroundcheck값이 1로 세팅되어있다면,
        //부모 프로세스는 기다리지 않고 바로 다음 prompt를 입력받는다.
        if(backgroundcheck==1){
            backgroundcheck<-0;
            continue;
        }
        else{
            //backgroundcheck값이 세팅되지 않았다면
            //부모 프로세스는 wait함수를 실행하여 기다린다.
            wait();
        }
}
```

2. 구현 방법

사용자가 터미널에서 작업할 시에는 일반적으로는 하나의 프로세스만 실행할 수 있습니다. 사용자가 명령을 입력하면 셸은 사용자가 입력한 명령을 해석하고 곧바로 그 결과를 출력하게 됩니다. 이렇게 결과가 나올 때 까지 기다려야 하는 프로세스 작업 방식을 포그라운드 프로세스라고 합니다. 원래 기본적인 셸 역시 포그라운드 방식으로 작업이 수행됩니다.

그에 반대로 백그라운드 프로세스는 앞에서 프로세스가 실행되는 동안 뒤에서 다른 프로

세스가 실행될 수 있어 터미널에서 한번에 여러 가지 프로세스를 실행할 수 있습니다.

백그라운드 방식으로 명령을 실행하면 명령의 처리는 뒤에서 진행되고, 곧바로 프롬프트가 띄워져서 다음 명령어를 입력받을 수 있습니다. 백그라운드 작업은 명령의 실행 시간이 많이 걸릴걸로 예상되거나, 명령을 실행한 후에 다음 명령을 입력받아야 할 때 사용합니다. 명령을 백그라운드로 실행하려면 커맨드의 마지막에 & 기호를 입력하면 됩니다.

기본적으로 주어진 코드에서는 포그라운드 방식의 프로세스 실행만 가능했지만, 이번 과제를 통해서 백그라운드 실행을 구현해 보았습니다.

일반적으로 부모 프로세스 단계에서 프롬프트를 통해 명령어를 입력받고, 자식 프로세스가 명령을 실행하는 동안 부모 프로세스는 wait()함수를 통해 자식 프로세스가 끝나기를 기다립니다. 그렇지만 백그라운드 실행을 구현해야 하기 때문에 입력한 커맨드의 마지막에 &이 입력되었다면 백그라운드 플래그를 1로 세팅한 후, 부모 프로세스는 자식 프로세스가 끝나길 기다리지 않고 바로 다음 프로세스를 입력받습니다. &이 입력되지 않았다면 자식 프로세스가 끝날 때까지 기다리고, 프로세스 실행이 끝난 이후에 다음 커맨드를 입력받도록 해야 합니다.

<background 프로세스인지 확인하는 과정>

```
int cmdNum= makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG);

    if(strcmp(cmdvector[cmdNum-1], "&")==0){
        //vector의 마지막 문자열이 &라면, 백그라운드 확인 변수를 1로 세팅한다.
        backgroundCheck=1;
        cmdvector[cmdNum-1]='\0';
    }
```

프롬프트로 입력받은 명령어의 가장 마지막 문자열이 &라면 해당 프로세스가 백그라운드에서 실행되어야 한다는 뜻입니다. 따라서 cmdvector의 맨 마지막 문자열인 cmdNum-1 문자열과 &를 비교하여, 0을 리턴한다면 background 확인 플래그인 backgroundCheck 플래그를 1로 세팅합니다. 그리고 다른 명령어들을 처리하기 위해서 & 문자열은 제거합니다.

```

if(strcmp(cmdvector[0],"cd")==0){
    cmd_cd(cmdNum,cmdvector);
}
else if (strcmp(cmdvector[0],"exit")==0){
    exit(1);
}
else{
    switch(pid=fork()){
    case 0:
        //프롬프트로 입력받은 첫번째 단어를 실행
        execvp(cmdvector[0], cmdvector);
        fatal("main()");
    case -1:
        fatal("main()");
    default:
        //backgroundCheck 변수가 1로 세팅되어있다면 부모 프로세스는
        //자식 프로세스가 끝나는 것을 기다리지 않고 바로 다음 프롬프트 입력받음
        if(backgroundCheck==1){
            backgroundCheck=0;
            continue;
        }
        else{
            //background 변수가 세팅되지 않았다면 자식 프로세스가 끝나길 기다린다.
            wait(NULL);
        }
    }
}
}

```

- ⇒ 자식 프로세스가 생성되고 난 뒤에 백그라운드 실행을 위해서는 부모 프로세스가 기다리지 않고 바로 다음 명령을 입력받아야 됩니다. 따라서 switch 문의 default 케이스에서 만약 앞에서 background flag가 1로 세팅되었다면 다음 명령문을 위해서 background flag 세팅을 제거하고 , 기다리는 대신에 다음 명령어를 입력받기 위해서 continue 문을 실행합니다.
- ⇒ 만약 & 이 입력되지 않고, background 변수가 세팅되지 않았다면 wait()함수를 통해서 자식 프로세스가 끝나기 기다립니다.

3. 테스트 사항

- 1) Sleep 10 &, Sleep 20 & 을 입력하고 지연 없이 바로 다음 명령어를 입력받는지를 확인하세요.

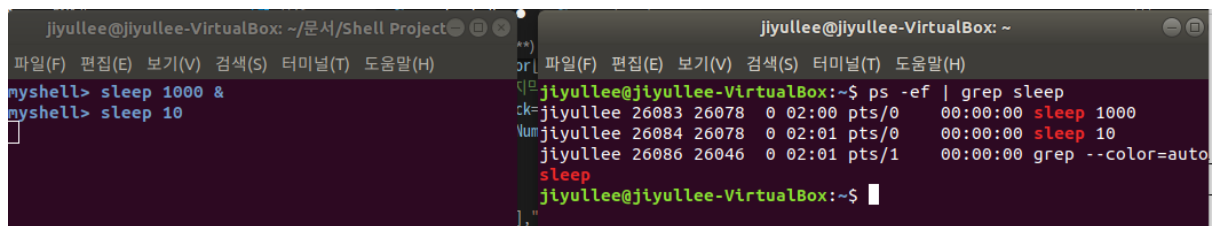
```

jiyullee@jiyullee-VirtualBox:~/문서/Shell Project$ ./a.out
myshell> sleep 10 &
myshell> sleep 20 &
myshell> ps
  PID TTY          TIME CMD
 1827 pts/0        00:00 bash
 25885 pts/0        00:00 a.out
 25906 pts/0        00:00 sleep
 25907 pts/0        00:00 sleep
 25908 pts/0        00:00 ps

```


Sleep 10 &을 입력하면 바로 지연 없이 다음 명령어를 입력받는 프롬프트가 실행되는 것을 알 수 있고, 곧바로 sleep 20 &을 실행하는 것을 알 수 있습니다. 그 이후에 ps 명령어를 통해 프로세스 상태를 확인하면 sleep 명령이 백그라운드에서 잘 실행되는 것을 확인할 수 있습니다.

2) sleep 1000 &와 sleep 10을 실행하고, 다른 터미널에서 ps -ef | grep sleep 명령어를 실행합니다.



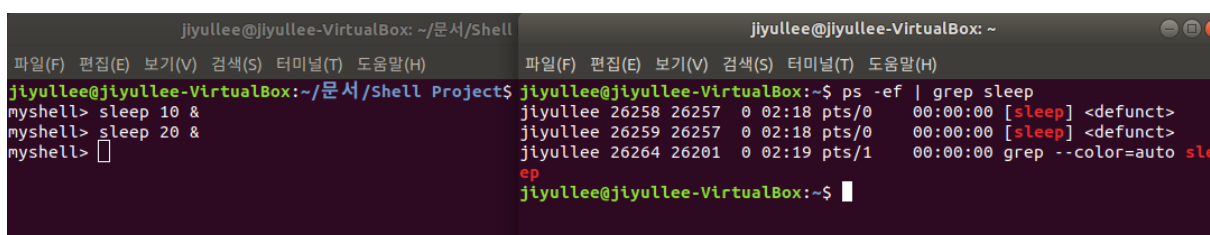
```
jiyullee@jiyullee-VirtualBox: ~/문서/Shell Project
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
myshell> sleep 1000 &
myshell> sleep 10

jiyullee@jiyullee-VirtualBox: ~
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
jiyullee@jiyullee-VirtualBox:~$ ps -ef | grep sleep
jiyullee 26083 26078 0 02:00 pts/0 00:00:00 sleep 1000
jiyullee 26084 26078 0 02:01 pts/0 00:00:00 sleep 10
jiyullee 26086 26046 0 02:01 pts/1 00:00:00 grep --color=auto sleep
jiyullee@jiyullee-VirtualBox:~$
```

Sleep 1000 & 명령어를 입력하고 나면 바로 지연 없이 다음 프롬프트를 출력해 다음 명령을 입력받습니다. 또 sleep 10을 &없이 입력하면 그대로 지연돼서 실행됩니다. 그리고 다른 창에서 ps -ef | grep sleep 을 입력하면 두 프로세스가 모두 잘 실행되고 있는 것을 확인할 수 있습니다.

4.테스트 사항 1의 문제점 분석

해당 테스트 1번을 실행하고 나서 다시 ps -ef | grep sleep 명령어를 다른 터미널에서 실행해 보았습니다.



```
jiyullee@jiyullee-VirtualBox: ~/문서/Shell Project
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
jiyullee@jiyullee-VirtualBox:~/문서/Shell Project$ ps -ef | grep sleep
jiyullee 26258 26257 0 02:18 pts/0 00:00:00 [sleep] <defunct>
jiyullee 26259 26257 0 02:18 pts/0 00:00:00 [sleep] <defunct>
jiyullee 26264 26201 0 02:19 pts/1 00:00:00 grep --color=auto sleep
jiyullee@jiyullee-VirtualBox:~$
```

그 결과 sleep 10, sleep 20 두 프로세스 모두 <defunct> 되었다는 결과를 확인할 수 있었습니다.

여기서 defunct된 프로세스는 일명 Zombie process로, 어떤 자식 프로세스가 자신의 일을 종료하였지만, 그 종료된 결과나 상태를 자신의 부모 프로세스에게 전달한 후에 종료하여야 하지만, 이를 전달받을 때까지 부모 프로세스가 wait 하여 자식 프로세스의 메모리를 수거하지 않고 다음 line을 계속 수행했기 때문에 발생하는 문제입니다. 이를 해결하기 위해서는 자식 프로세스가 종료되었을 때 다음 명령을 계속 수행하고 있는 부모 프로세스에게 자식 프로세스의 수행이 끝났다는 신호를 줘야 합니다. 이를 위해서는 부모 프로세스에 signal handler를 등록해야 합니다. 이 signal handler를 등록하게 되면 자식 프로세스의 종료 이후에 SIGCHLD 라는 이벤트가 발생하게

되는데, 이후에 SIGCHLD 이벤트를 처리하는 시그널 핸들러가 부모 프로세스에게 자식 프로세스가 끝났다는 시그널을 전달하게 되고, 부모 프로세스는 종료된 자식 프로세스의 상태를 전달받고 메모리를 수거할 수 있게 됩니다. 그 이후에는 자식 프로세스는 좀비 프로세스로 남아있지 않고 제대로 종료될 수 있을 것으로 보입니다.

Project #2

요구사항 1) SIGCHLD로 자식 프로세스가 wait() 시에 프로세스가 온전하게 수행되도록 구현해야 한다.

1) 수도코드

```
int main(){
    sigaction act; //sigaction 구조체
    act->handler=child_terminate; //핸들러에 함수 포인터 값을 지정해야 한다.
    sigemptyset(&act->sa_mask); //sa_mask의 모든 비트를 0으로 초기화
    act->sa_flags=0; //signal 함수를 위해 필요한 멤버가 아니므로 0으로 초기화
    sigaction(SIGCHLD,&act,0); //sigchld에 대한 handler를 지정하고 있다.
}

child_terminate(){
    pid p_id;
    int status;
    p_id->waitpid();
    //waitpid함수호출로 인해 자식프로세스는 좀비가 되지 않고 소멸한다.
}
```

2) 구현 방법

우선 앞서서 백그라운드 실행 시에 자식프로세스가 종료되어도 이를 처리해줄 부모 프로세스가 없기 때문에 좀비 프로세스가 발생하였습니다. 따라서 이를 해결하기 위해서 자식 프로세스가 종료되면 다음 명령을 수행하고 있는 부모 프로세스에게 이 사실을 전달해주어야 합니다. 따라서 SIGCHLD 라는 시그널을 전달해 문제를 해결할 수 있습니다.

따라서 signal함수보단 좀더 안정적으로 동작하지만, 같은 역할을 하는 sigaction 함수를 이용하여

시그널 핸들러를 등록하여 SIGCHLD가 발생하면 부모 프로세스에게 이를 전달하여 자식 프로세스를 처리할 수 있어야 합니다. 백그라운드 실행에서는 부모 프로세스가 자식 프로세스가 끝나기까지 기다리지 않습니다. 따라서 wait함수를 이용해 block되어있으면 안됩니다. 따라서 이를 방지하기 위해 핸들링 함수에서는 waitpid 함수를 이용해서 자식 프로세스가 끝날때까지 block되어있지 않고 프로세스가 할일을 실행하게 됩니다. 그러다가 자식 프로세스가 끝났다는 signal을 받으면 0을 반환하고, 자식 프로세스를 종료하게 됩니다.

3) 소스코드

<메인 함수 구현부>

```
int main(int argc, char**argv){
    pid_t pid;
    int i=0;

    struct sigaction act;

    act.sa_handler=child_terminate;
    sigemptyset(&act.sa_mask);
    act.sa_flags=0;
    sigaction(SIGCHLD,&act,0);
```

=> 우선 sigaction이라는 구조체 변수를 선언 및 초기화해주어야 합니다.

1. sa_handler => 멤버에는 시그널 핸들러의 함수 포인터 값을 저장해주면 됩니다. 따라서 시그널이 전달된 후에 호출할 함수의 이름을 저장하게 됩니다.

2. sa_mask, sa_flag는 signal함수를 대신하기 위해 필요한 함수가 아니므로 0으로 초기화해줍니다. 이를 위해 sigemptyset함수가 쓰이고, sa_flag값은 0으로 세팅해주면 됩니다.

마지막으로 sigaction 함수에 sigchld가 발생하면 호출될 함수가 저장된 sigaction 구조체를 변수로 전달하게 됩니다. 이를 통해 자식 프로세스가 종료된 후에 child_terminate 함수가 전달되게 됩니다.

<시그널 핸들러 함수>

```
void child_terminate(int sig){
    pid_t p_id;
    int status;
    p_id=waitpid(-1,&status,WNOHANG);
}
```

child_terminate함수에서는 sigchld를 전달받아서 waitpid함수를 호출합니다. 해당 함수를 통해 부모 프로세스는 본인 할 일을 하다가 자식 프로세스가 종료되었다는 사실을 알고 자식 프로세스를 처리하게 됩니다. 따라서 좀비 프로세스가 발생하지 않고, 자식프로세스는 소멸이 되게 됩니다.

4. 테스트 사항

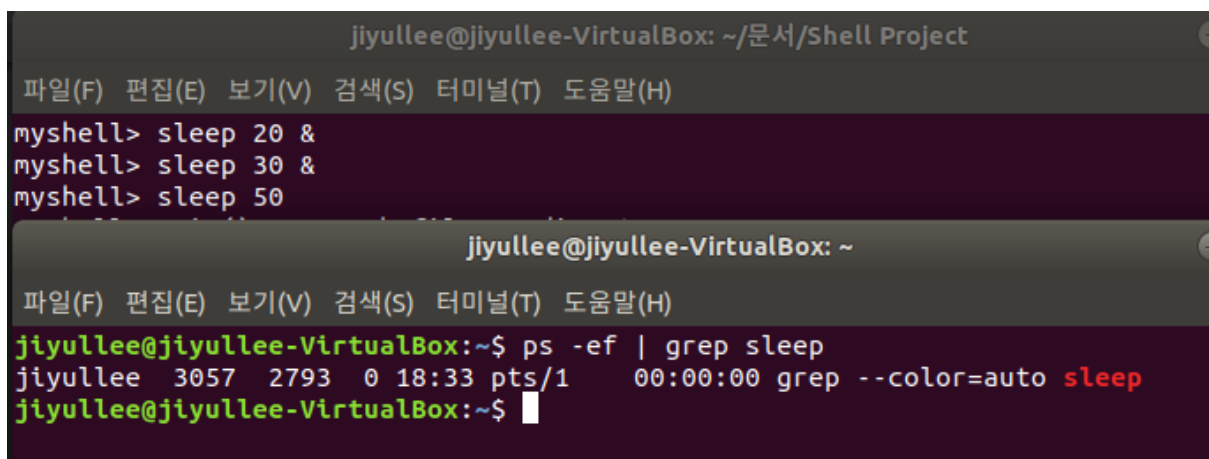
1) sleep 50이 sleep 20과 sleep 30에 방해되지 않고 잘 실행되는가?

=> 잘 실행이 되었다.

```
myshell> sleep 20 &
myshell> sleep 30 &
myshell> sleep 50
```

2) 좀비 프로세스가 발생했는가?

=> 자식 프로세스가 소멸되었고, defunct가 나타나지 않고 좀비 프로세스는 발생하지 않았다.



```
jiyullee@jiyullee-VirtualBox: ~/문서/Shell Project
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
myshell> sleep 20 &
myshell> sleep 30 &
myshell> sleep 50

jiyullee@jiyullee-VirtualBox: ~
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
jiyullee@jiyullee-VirtualBox:~$ ps -ef | grep sleep
jiyullee 3057 2793 0 18:33 pts/1 00:00:00 grep --color=auto sleep
jiyullee@jiyullee-VirtualBox:~$
```

요구사항 2) ^C(SIGINT), ^W(SIGQUIT) 사용시 셸이 종료되지 않도록 하고, Foreground 프로세스 실행 시 SIGINT를 받으면 프로세스가 끝나는 것을 구현

1. 수도 코드

```
int sig=1; //전역변수로, 시그널 발생시 다음 프롬프트를 입력하기 위해
void sigint_handler(){
    signal(sigint,sig_ign);
    sig=1;
}
void sigquit_handler(){
    signal(sigquit,sig_ign);
    sig=1;
}

while(1){
    signal(sigint,sigint_handler);
    signal(sigquit,sigquit_handler);

    fputs();
    fgets();

    if(sig==1){
        //셸 자체에서 signal이 발생했다면
        //signal함수를 통해 ctrl+c 명령은 무시되어서 실행되지 않았으니,
        //아예 fork를 하지 않기 위해 아래 코드는 무시하고, 다음 while루프를 실행해 다음 프롬프트를 입력받는다.
        sig=0;
        continue;
    }

    switch(){
        case 0:
            if(backgroundCheck!=1){
                signal(sigquit,sig_DFL);
                signal(sigint,SIG_DFL);}
            //원래 셸에서는 sigquit와 sigint를 무시하지만,
            //중간에 포그라운드 프로세스 실행시 자식 프로세스에서는 프로세스 중간에 ctrl+c와 ctrl+\를 입력할 수 있게 된다.
    }
}
```

2. 구현 방법

우선 while문 처음에 signal함수를 이용해 signal handler를 등록합니다. 여기서 바로 SIG_IGN를 사용하지 않는 이유는, 원래의 기본 셸 코드에 엔터를 바로 입력하면 세그멘테이션 오류가 발생하는데, SIG_IGN을 통해서 셸에서 ctrl+c를 무시했기 때문에 ctrl+c이후에 엔터키를 입력하면 아무것도 입력하지 않은 다음에 바로 엔터 키를 입력한 거나 다름없기 때문에 세그멘테이션 오류가 발생한다. 따라서 시그널 핸들러를 따로 등록해 하나의 프로세스를 더 진행해줘야 합니다.

셸에서 ctrl+c나 ctrl+/를 입력받으면 시그널 핸들러에서 signal함수를 통해 SIGINT와 SIGQUIT를 SIG_IGN을 통해서 무시해줘야 합니다. 그렇지만 무시만 되었을 뿐 엔터키를 입력해줘야 다음 프롬프트를 입력받을 수 있기 때문에 전역 변수 sig를 이용해서 만약 시그널이 발생하였으면, 해당 플래그를 1로 세팅하여 부모/자식 프로세스를 fork하지 않고 다음 while loop로 넘어가게 됩니다. 따라서 ctrl+c, ctrl+w 뒤에 무엇이 입력되든 이미 다음 while 루프로 넘어간 상태이기 때문에 무시가 되고, 엔터키를 입력하면 셸은 바로 다음 프롬프트를 출력하게 됩니다.

이제, 포그라운드 프로세스에서 `sigint`, `sigquit` 시그널이 발생하였어도 무시하지 않고 원래의 작업을 수행하여 프로세스를 중간에 종료시켜야 합니다. 따라서 `SIGNAL` 함수에 `SIG_DFL` 멤버변수를 등록하여 `SIGNAL`을 무시하지 않고 원래 역할대로 수행되어 포그라운드 프로세스를 종료시키게 됩니다. 이때, 포그라운드 프로세스인지를 확인하기 위해서 `backgroundCheck` 변수가 1이 아닐 때만 실행시킵니다.

3. 소스 코드

<메인 함수부>

```
while (1) {
    signal(SIGINT,(void*)sigint_handler);
    signal(SIGQUIT,(void*)sigquit_handler);

    //프롬프트 출력
    fputs(prompt, stdout);
    //cmdline으로부터 문자열 가져옴
    fgets(cmdline, BUFSIZ, stdin);

    if(sig==1){
        sig=0;
        continue;
    }

    cmdline[strlen(cmdline) -1] = '\0';
    //delim 공백을 기준으로 단어를 잘라낸다.

    int cmdNum= makelist(cmdline, " \t", cmdvector, MAX_CMD_ARG);

    if(strcmp(cmdvector[cmdNum-1],"&")==0){
        //vector의 마지막 문자열이 &라면, 백그라운드 확인 변수를 1로 세팅한다.
        backgroundCheck=1;
        cmdvector[cmdNum-1]='\0';
    }

    if(strcmp(cmdvector[0],"cd")==0){
        cmd_cd(cmdNum,cmdvector);
    }
    else if (strcmp(cmdvector[0],"exit")==0){
        exit(1);
    }
    else{
        switch(pid=fork()){
            case 0:
                //프롬프트로 입력받은 첫번째 단어를 실행
                if(backgroundCheck!=1){
                    signal(SIGQUIT,SIG_DFL);
                    signal(SIGINT,SIG_DFL);
                }
                execvp(cmdvector[0], cmdvector),
        }
    }
}
```

빨간색 박스 부분이 기존의 코드에서 추가된 부분입니다.

1) `ctrl+c` (`SIGINT`), `ctrl+₩` (`SIGQUIT`) 를 무시하고, 시그널이 발생한 것을 `while` loop에서도 알 수 있도록 하는 함수를 호출하는 시그널 핸들러를 등록합니다.

2) 만약 시그널이 발생하였다면, 부모+ 자식 프로세스를 생성하기 이전에 다음 코드를 무시하여

다음 while loop로 넘어갈 수 있도록 continue 코드를 추가합니다.

3) 마지막으로 foreground 프로세스일 때는 SIGINT와 SIGQUIT가 기존의 작업을 그대로 진행하여 프로세스를 종료시킬수 있도록 SIG_DFL를 등록합니다.

< 시그널 핸들러>

```
int sig=0;

void sigint_handler(){
    signal(SIGINT,SIG_IGN);
    sig=1;
    return;
}

void sigquit_handler(){
    signal(SIGQUIT,SIG_IGN);
    sig=1;
    return;
}
```

만약 ctrl+c와 ctrl+w가 입력되었으면 해당 시그널을 무시하고, signal이 발생하였는지 여부를 체크하는 sig 변수를 1로 세팅해줍니다.

4. 테스트 사항

1) 쉘이 ctrl+c와 ctrl+w를 무시하되, 종료되지 않는다.

```
myshell> ^C
myshell> ^C
myshell> ^\
myshell> ^\
myshell>
```

=> ctrl+c와 ctrl+w를 무시하며, 여러 번 입력해도 오류가 발생하지 않고 잘 종료된다.

2) 포그라운드 프로세스 중간에 ctrl+c, ctrl+w를 입력하면 프로세스가 종료된다.

```
myshell> sleep 50
^Cmyshell>
```

```
myshell> sleep 30  
^\\myshell> █
```

ctrl+c, ctrl+W 모두 입력하면 포그라운드 프로세스가 중간에 종료됩니다.

+) 테스트 사항에는 따로 없었지만, 중간에 포그라운드 프로세스에서 SIG_DFL을 이용해 시그널 발생시 무시하지 않고 디폴트 작업으로 바꿔 SIGINT/ SIGQUIT가 프로세스를 종료시키게 한 이후로도 while 문 처음 부분마다 셸에서는 SIGNAL 발생시 이를 무시할 수 있도록 핸들러를 등록하여 다음 프롬프트가 출력된 다음에도 ctrl+c와 ctrl+W를 잘 무시하게 됩니다.

```
myshell> sleep 30  
^\\myshell> ^C  
myshell> ^\\  
myshell>
```


Project #3 – Redirection & Pipe 구현

요구사항 1) Redirection 구현

1) 수도코드

-입력 리다이렉션 (cmd_redir_in)

```
int cmd_redir_in(char **cmdvector)
{
    input: fd, pos;
    //fd는 파일 디스크립터이고, pos는 < 기호의 위치이다.

    for(pos=0;cmdvector[pos]!=NULL;pos++){
        if(cmdvector[pos]=="<") break;
    }

    if(cmdvector[pos] exists){
        if(cmdvector[pos+1]) return -1; //입력할 파일 값이 없으면 -1을 리턴한다.
        else{
            if(fd=open(cmdvector[pos],O_RDONLY,0644)==-1){
                //파일을 열어야 하므로 open함수를 이용하고, 입력옵션을 지정해주고, 권한을 부여해준다.
                perror();
                //입력할 파일을 여는데 오류가 발생하면 에러 메시지를 띄운다.
            }
            //표준 입력(0번 file descriptor)을 닫아준다.
            close(STDIN_FILENO);
            //dup2함수를 이용해 모든 표준입력을 fd방향으로 바꿔준다. 이제 키보드가 아닌 파일에서 입력받게 된다.
            dup2(fd,STDIN_FILENO);
            //이제 모두 사용한 file descriptor는 닫아준다.
            close(fd);

            //그 이후로 <와 출력파일 cmdvector는 제거하고, 커맨드를 앞으로 당겨준다.
            for (pos=pos;cmdvector[pos]!=NULL;pos++){
                cmdvector[pos]=cmdvector[pos+2];
            }
            cmdvector[pos]=NULL;
        }
    }
    return 0;
}
```

-출력 리다이렉션 (cmd_redir_out)

```

int cmd_redir_out(char **cmdvector)
{
    input: fd, pos;
    //fd는 파일 디스크립터이고, pos는 > 기호의 위치이다.

    for(pos=0;cmdvector[pos]!=NULL;pos++){
        if(cmdvector[pos]==">") break;
    }

    if(cmdvector[pos] exists){
        if(cmdvector[pos+1])
            perror();
        return -1; //입력할 파일 값이 없으면 에러 메시지를 출력하고 -1을 리턴한다.
        else{
            if(fd=open(cmdvector[pos],O_RDWR|O_CREAT|O_TRUNC,0644)==-1){
                //파일을 열어야 하므로 open함수를 이용하고, 출력옵션을 지정해주고, 권한을 부여해준다.
                perror();
                //출력 파일을 여는데 오류가 발생하면 에러 메시지를 띄운다.
            }
            //표준 출력(1번 파일 디스크립터)을 닫아준다.
            close(STDOUT_FILENO);
            //dup2함수를 이용해 모든 출력을 fd방향으로 바꿔주므로,
            //터미널에 출력되어야 할 데이터는 파일에 출력되게 된다.
            dup2(fd,STDOUT_FILENO);
            //이제 모두 사용한 file descriptor는 닫아준다.
            close(fd);

            //그 이후로 <와 출력파일 cmdvector는 제거하고, 커맨드를 앞으로 당겨준다.
            for (pos=pos;cmdvector[pos]!=NULL;pos++){
                cmdvector[pos]=cmdvector[pos+2];
            }
            cmdvector[pos]=NULL;
        }
    }
}

```

2. 구현 방식

우선 리다이렉션이란 표준 입출력 에서 입출력 방향을 입력한 파일로 재지정하는 것을 의미합니다. 셸에서 키보드를 통해 명령을 입력받는 것을 표준입력, 즉 Standard Input이라고 하며, 키보드에서 입력받은 실행 결과를 모니터를 통해 출력하는 것을 Standard Output이라고 합니다. 원래 대로라면 사용자가 입력/출력할 내용은 각각 키보드에서 입력받고, 이를 스크린으로 출력하지만, 리다이렉션을 통해서 파일의 내용을 입력받거나 사용자가 입력한 내용을 파일로 출력할 수 있습니다.

유닉스 시스템에서는 표준입력과 표준출력을 file descriptor로 기본적으로 할당해놓았는데, 표준입력은 0, 표준출력은 1입니다. 이를 STDIN_FILENO, STDOUT_FILENO로 참조할 수 있습니다.

따라서 입 출력 리다이렉션을 구현하는 직접적인 틀은 비슷합니다.

- 1) 입,출력을 진행할 파일의 디스크립터를 지정하고, "<",">"기호의 위치를 파악할 pos변수를 선

연합니다.

2) 만약 파일 이름이 명시되지 않았다면 에러메세지를 출력하고 리턴합니다.

3) 파일 이름이 제대로 입력되었다면, open 함수를 통해 파일을 열어주고, 입/출력에 따라 O_FLAG 옵션을 다르게 지정합니다. 입력이라면 O_RDONLY값을 지정해주고, 출력이라면 O_RDWR,O_CREAT,O_TRUNC 옵션을 통해 파일이 없으면 생성하고, 있으면 파일의 길이를 0으로 변경하고 덮어씌워서 출력합니다.

4) 그 이후엔 표준입력/ 출력의 파일 디스크립터를 닫아줍니다.

5) dup2(fd,STDIN or STDOUT)함수를 통해서 file descriptor를 복제해 모든 입/출력을 fd로 향하게 해줍니다. 따라서 dup2가 호출된 이후에는 모든 입력은 해당 fd가 가르키는 파일로부터 입력받고, 모든 출력은 fd가 가르키는 파일로 출력되게 됩니다.

6) close함수를 통해 사용한 file descriptor는 닫아줍니다.

7) cmdvector에서 더 이상 "<", ">"기호와 파일 이름은 쓸모가 없으니 커맨드를 삭제하고 커맨드를 두 인덱스씩 앞으로 당겨 줍니다.

8) 그 이후에 메인 함수에서 execvp를 통해서 입력한 커맨드를 실행합니다.

이 과정을 통해서 입출력 방향을 표준 입출력에서 파일로 바꿔주는 과정을 리다이렉션이라고 합니다.

3. 구현 코드

1) 입력 리다이렉션

```
int cmd_redir_in(char ** cmdvector){
    int fd;
    int pos;
    for(pos=0;cmdvector[pos]!=NULL;pos++){
        //">"기호를 찾으면 해당 위치에서 멈추고, 위치를 저장해둔다.
        if(!strcmp(cmdvector[pos],"<")){
            break;
        }
    }
}
```

->우선 리다이렉션을 의미하는 ">"기호를 찾으면 해당 위치를 저장하고, for문을 빠져나옵니다.

->구현 방법에서 언급했듯 파일을 open해주고, 파일 디스크립터 값을 리턴받은 다음에 해당 표준 입력에서 파일로 입력 방향을 바꿔준 다음에 표준 입력과 파일 디스크립터는 모두 close해준다. 그 이후에 필요 없는 커맨드인 < 와 파일 이름을 제거해줍니다. 이를 통해 셸은 입력을 파일로부터 받게 됩니다.

```
if(cmdvector[pos]){
    if(!cmdvector[pos+1]){
        //파일 이름 입력하지 않았을 시 오류 메시지 출력
        perror("Enter file name!");
        return -1;
    }
    else{
        if((fd=open(cmdvector[pos+1],O_RDONLY,0644))== -1){
            //파일 오픈해주고, 오류 발생시 오류 메시지 출력
            perror("open error!");
            return -1;
        }

        //표준 입력을 닫아주고, 입력 방향을 파일로 바꿔준 다음에 파일 디스크립터도 닫아준다.
        close(STDIN_FILENO);
        dup2(fd,STDIN_FILENO);
        close(fd);

        //필요 없는 커맨드는 제거한다.
        for (pos=pos;cmdvector[pos]!=NULL;pos++){
            cmdvector[pos]=cmdvector[pos+2];
        }
        cmdvector[pos]=NULL;
    }
}
return 0;
}
```

2) 출력 리다이렉션

```
int cmd_redir_out(char **cmdvector){

    int fd;
    int pos;
    for(pos=0;cmdvector[pos]!=NULL;pos++){
        if(!strcmp(cmdvector[pos],">")){
            //">"기호를 찾으면 해당 위치에서 멈추고, 위치를 저장해둔다.
            break;
        }
    }

    if(cmdvector[pos]){
        if(!cmdvector[pos+1]){
            //파일 이름 입력하지 않았을 시 오류 메시지 출력
            perror("Enter file name!");
            return -1;
        }
        else{
            if((fd=open(cmdvector[pos+1],O_RDWR|O_CREAT|O_TRUNC,0644))== -1){
                //파일 오픈해주고, 오류 발생시 오류 메시지 출력
                perror("open error!");
                return -1;
            }
        }
    }
}
```

```

    }
    //표준 출력 닫아주고, 출력 방향을 파일로 바꿔준 다음에 파일 디스크립터도 닫아준다.
    close(STDOUT_FILENO);
    dup2(fd,STDOUT_FILENO);
    close(fd);

    //필요없는 커맨드는 제거한다.
    for (pos=pos;cmdvector[pos]!=NULL;pos++){
        cmdvector[pos]=cmdvector[pos+2];
    }
    cmdvector[pos]=NULL;
}
return 0;
}

```

->입력 리다이렉션과 동일하게 진행해줍니다. 반대로 출력 방향을 파일로 바꿔줌으로써 사용자가 입력한 내용을 파일로 출력하는 기능을 수행하게 됩니다.

-Main 함수부분

```

cmd_redir_in(cmdvector);
cmd_redir_out(cmdvector);
execvp(cmdvector[0], cmdvector);

```

->execvp를 이용해 커맨드를 실행하기 이전에, cmd_redir_in과 cmd_redir_out함수를 통해 리다이렉션 기호가 포함되어있는지 확인하고, 포함되어있다면 입출력 방향을 재지정해주게됩니다.

4) 테스트 내용

1) Redirection test

```

myshell> cat > test.txt
Hi my name is cat
what is yours?
myshell> cat < test.txt
Hi my name is cat
what is yours?
myshell> cat < test.txt > test1.txt
myshell> cat test1.txt
Hi my name is cat
what is yours?

```

입,출력 리다이렉션이 모두 잘 진행되는 것을 확인할 수 있습니다. 입,출력 리다이렉션을 동시에 진행했을 때에도 test파일로부터 입력받고, test1.txt파일을 새로 생성해 출력이 잘 되어있는 것

을 cat 을 통해 확인할 수 있습니다.

2) 백그라운드에서 리다이렉션 테스트

```
myshell> cat > test2.txt &  
myshell>
```

- ➔ 입력을 사용자로부터 받아야 하지만, 우선 백그라운드로 실행이 되어 하는 것이 조건에 있기 때문에 myshell 프롬프트가 바로 출력되게 함으로써 리다이렉션이 백그라운드에서도 잘 실행되는 것을 확인할 수 있습니다.

요구사항 #2) Pipe 구현

1) 수도코드

- 파이프 기호가 포함되어있는지 확인하는 함수 cmd_pipe_check()

```
//|를 기준으로 나뉜 각 명령어를 pipevec에 저장해줍니다.  
char* pipevec1[10];  
char* pipevec2[10];  
char* pipevec3[10];  
  
//cmd_pipe_check함수는 우선 파이프 기호인 "|"가 있는지 확인해줍니다.  
int cmd_pipe_check(){  
    int i;  
    int pipenum=0;  
    int j=0;  
    //|를 기준으로 나뉜 각 파이프 벡터의 명령어 개수  
    int c1,c2,c3=0;  
  
    for(i=0;cmdvector[i]!=NULL;i++){  
        if(!strcmp(cmdvector[i],"|")){  
            //|를 기준으로 파이프 커맨드를 잘라줍니다.  
            j=0;  
            pipenum++;  
        }  
        // 파이프 개수를 세주면서 파이프전후로 나뉜 각 명령어를 구분하여 pipevec에 입력해줍니다.  
        elif(pipenum==0){  
            pipevec1[j]=cmdvector[i];  
            j++; c1++;  
        }  
        elif(pipenum==1){  
            pipevec2[j]=cmdvector[i];  
            j++; c2++;  
        }  
        elif(pipenum==2){  
            pipevec3[j]=cmdvector[i];  
            j++; c3++;  
        }  
    }  
    //각 파이프 명령어를 담은 벡터의 끝부분은 execvp함수를 위해 NULL로 지정해줍니다.  
    pipevec1[c1]=NULL;  
    pipevec2[c2]=NULL;  
    pipevec3[c3]=NULL;  
    //파이프 개수를 리턴해줍니다.  
    return pipenum;  
}
```

➔ 파이프가 한 개 포함되어있을 때 사용하는 함수: cmd_pipe_proc2

```
//단일 파이프 구현
int cmd_pipe_proc2(char* vec1[],char* vec2[]) {
    int fd[2];
    if(pipe(fd)==-1){
        //하나의 파이프에 대한 두개의 파일 디스크립터가 생성됩니다.
        // 각각 파이프의 입구/출구를 말합니다.
        perror("pipe failed!");
        exit(1);
    }

    switch(fork())
    {case -1 : perror(); break; //fork가 잘 되지 않았을 때 에러 메시지를 출력합니다.
      case 0 : //이를 기준으로 앞쪽 명령을 자식 프로세스에서 실행
        //명령어에 리다이렉션이 포함되어있을 수 있으므로 한번 체크를 해준다.
        cmd_redir_in(vec1);
        cmd_redir_out(vec1);
        //
        close(STDOUT_FILENO);
        dup2(fd[1],STDOUT_FILENO); //표준출력을 파이프의 입구에 연결
        //이미 각 프로세스에서 fd의 복제가 끝났으니 둘다 닫아준다.
        if(close(fd[1])== -1 || close(fd[0]) == -1) perror("close error!");
        //앞쪽 명령을 수행해준다.
        execvp(vec1[0], vec1);
        //exec이 잘 되지 않았을 때 에러 메시지를 출력합니다.
        perror();
        return 0;
      default: //이를 기준으로 뒤쪽 명령은 부모 프로세스에서 실행
        cmd_redir_in(vec2);
        cmd_redir_out(vec2);
        close(STDOUT_FILENO);
        dup2(fd[0],STDIN_FILENO); //표준입력을 파이프의 출구에 연결
        if(close(fd[1]) == -1 || close(fd[0]) == -1) perror("close error!");
        //뒤쪽 명령을 수행해준다.
        execvp(vec2[0], vec2);
        perror();
        wait(0);
    }

    //사용한 파일 디스크립터를 모두 닫아준다.
}
```

- 다중 파이프 구현 : cmd_pipe_proc3

```
//다중 파이프 구현|
int cmd_pipe_proc3(char* vec1[],char* vec2[],char* vec3[]){
    int fd[2],fd2[2];
    if(pipe(fd)==-1){
        perror("pipe failed!");
        exit(1);
    }

    switch(fork()){
        case 0:
            //파이프를 하나 더 생성해준다.
            if(pipe(fd2)==-1){
                perror("pipe failed!");
                exit(1);
            }
            switch (fork()){
                case 0:
                    //리다이렉션이 있는지를 확인.
                    cmd_redir_in(vec1);
                    cmd_redir_out(vec1);
                    //표준 출력을 2번 파이프의 입구로 연결한다.
                    dup2(fd2[1],STDOUT_FILENO);
                    //파이프 두개가 복제되었으므로 모두 닫아준다.
                    if( close(fd[0])==-1 || close(fd[1])==-1 || close(fd2[0])==-1 || close(fd2[1])==-1)
                        perror("close error!");
                    execvp(vec1[0],vec1);
                    exit(0);

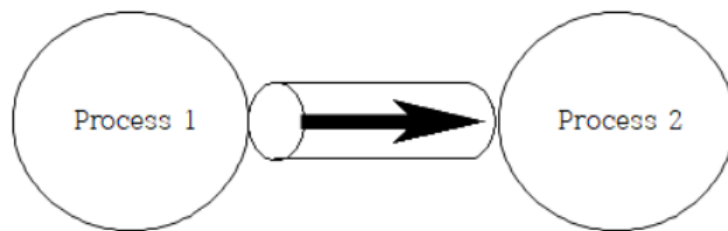
                default:
                    cmd_redir_in(vec2);
                    cmd_redir_out(vec2);
                    close(STDOUT_FILENO);
                    //표준 입력을 2번 파이프의 출구에 연결한다.
                    dup2(fd2[0],STDIN_FILENO);
                    //표준 출력을 1번 파이프의 입구에 연결한다.
                    dup2(fd[1],STDOUT_FILENO);
                    if( close(fd[0])==-1 || close(fd[1])==-1 || close(fd2[0])==-1 || close(fd2[1])==-1)
                        perror("close error!");
                    execvp(vec2[0],vec2);
                    wait(NULL);
            }
        }
    }
    exit(0);
}
```

```
        default:
            //표준 입력을 1번 파이프의 입구에 연결한다.
            cmd_redir_in(vec3);
            cmd_redir_out(vec3);
            close(STDIN_FILENO);
            dup2(fd[0],STDIN_FILENO);
            if( close(fd[0])==-1 || close(fd[1])==-1) perror("close error!");
            execvp(vec3[0],vec3);
            wait(NULL);
        }
    }
}
```


2. 구현방식

: 원래 멀티 프로세스에서는 각각의 메모리가 독립적이기 때문에 각 프로세스간에 데이터를 주고받는 게 불가능합니다. 이를 해결하기 위해 파이프가 등장하였습니다.

=> 파이프란 데이터가 어떤 프로세스에서 다른 프로세스로 단방향으로 전달되도록 합니다. 각 프로세스는 파이프에 의해서 연결되며, 파이프의 한쪽 입구에서 반대쪽 출구로 이동하면서 데이터를 공유할 수 있도록 합니다. 파이프의 한쪽 끝은 readable end라고 하여 읽기만 수행하고, 파이프의 한 쪽 끝은 writable end라고 하여 쓰기만 수행합니다.



따라서 파이프를 통해 연결된 프로세스의 데이터 흐름도는 위의 그림과 같습니다. 각 프로세스는 "|"파이프 기호로 구분되며, 첫 번째 프로세스에서 출력된 데이터는 파이프의 출구로 나와 두 번째 프로세스의 데이터 입력이 됩니다. 따라서 각 프로세스들의 입력과 출력 방향을 표준 입출력에서부터 바꿔주는 것은 리다이렉션을 구현하는 방식과 매우 비슷하다고 이해할 수 있습니다.

이 개념을 가지고 제가 우선 단일 파이프 개념을 구현한 방식을 설명해보도록 하겠습니다.

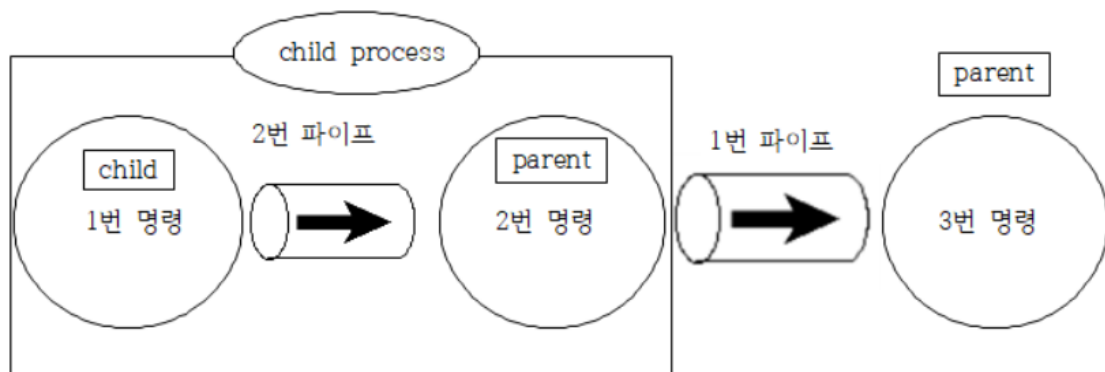
- 1) 우선 파이프기호가 입력되었는지 확인하고, 그리고 파이프 기호를 기준으로 각 명령어를 실행하는 프로세스를 생성하기 위해 명령어들을 파이프 벡터에 각각 저장해주었습니다.
- 2) 그리고 pipe()함수를 호출하여 부모 프로세스가 자식 프로세스와 데이터를 주고받을 수 있도록 파일 디스크립터를 파이프의 읽기 부분과 쓰기 부분에 연결시켜 줍니다.
- 3) 그 이후에 | 을 기준으로 fork함수를 통해 자식 프로세스는 첫번째 명령을 수행하도록, 부모프로세스는 두 번째 명령을 수행하도록 프로세스를 생성해 주었습니다.
- 4) 각 프로세스에서 cmd_redir_in, cmd_redir_out 함수를 각각 호출하여 리다이렉션이 있는지를 먼저 확인해줍니다.
- 5) 그리고 부모 프로세스의 출력 부분을 표준 입력의 파이프의 출구 부분인 fd[0]과 연결시켜 주기 위해서 dup2함수를 통해 복제해주고, 표준 입력과 파일 디스크립터 모두를 닫아 줍니다.
- 6) 자식 프로세스에서는 입력받는 부분을 표준 출력에서 파이프의 입구 부분인 fd[1]와 연결시

켜 줍니다. 그리고 표준 입력과 파일 디스크립터를 모두 닫아줍니다.

7) 그 이후에 각 프로세스에서 `execvp`을 통해 명령어를 수행해줍니다.

8) 그렇게 자식 프로세스에서 첫번째 명령이 수행된 결과가 파이프를 통해 송신되어 부모 프로세스가 데이터를 받아 두번째 명령을 수행할 수 있게 되는 것을 확인할 수 있습니다..

해당 방식으로 단일 파이프를 구현하였는데, 이를 바탕으로 제가 이중 파이프를 구현한 방법은 이렇습니다.



단일 파이프를 구현할 때는 파이프를 하나만 생성하였다면, 원래의 방식에서 파이프를 하나 더 생성하고, `fork()`를 프로세스 안에서 한번 더 진행하여 자식 프로세스에서 1번명령을 수행하고, 부모 프로세스에서 2번 명령을 수행하고, 조상 프로세스에서 3번 명령을 수행하는 식으로 구현하려 하였습니다. 1번 명령의 출력 부분을 2번 파이프의 읽기 끝에 연결하고, 2번 명령의 입력 부분을 2번 파이프의 출력 끝으로 연결한 후에, 1번 파이프의 입구에 2번 명령의 출력 부분을 연결하고, 3번 명령의 입구에 1번 파이프의 출구를 연결하는 식으로 구현하였습니다.

3) 구현 코드

1) 파이프 기호가 있는지를 확인하는 함수: cmd_pipe_check()

```
char* pipevec1[10];
char* pipevec2[10];
char* pipevec3[10];

int cmd_pipe_check(){
    int i;
    int pipenum=0;
    int j=0;
    int c1,c2,c3=0;

    for(i=0;cmdvector[i]!=NULL;i++){

        if(!strcmp(cmdvector[i],"|")){
            //|를 기준으로 파이프 커맨드를 잘라줍니다.
            j=0;
            pipenum++;
        }
        // |를 기준으로 각 명령어들을 각각의 벡터에 따로 담아줍니다.
        else if(pipenum==0){
            pipevec1[j]=cmdvector[i];
            j++;
            c1++;
        }
        else if(pipenum==1){
            pipevec2[j]=cmdvector[i];
            j++;
            c2++;
        }
        else if(pipenum==2){
            pipevec3[j]=cmdvector[i];
            j++;
            c3++;
        }
    }

    //execvp함수에서 EOF를 전달하기 위해 맨 끝부분을 NULL로 지정합니다.
    pipevec1[c1]=NULL;
    pipevec2[c2]=NULL;
    pipevec3[c3]=NULL;
    return pipenum;
}
```

2) 단일 파이프 코드

```
int cmd_pipe_proc2(char* vec1[],char* vec2[])  
{  
    int fd[2];  
    if(pipe(fd)==-1){  
        //하나의 파이프가 pipe함수를 통해 생성.  
        // 각 파일 디스크립터가 파이프의 읽기 끝/쓰기 끝 부분에 할당된다.  
        perror("pipe failed!");  
        exit(1);  
    }  
  
    switch(fork()){  
        case -1 : perror("fork error"); break;  
        case 0 :  
            //||을 기준으로 자식 프로세스에선 앞쪽 명령 프로세스 생성  
            cmd_redir_in(vec1);  
            cmd_redir_out(vec1);  
            close(STDOUT_FILENO);  
            dup2(fd[1],STDOUT_FILENO); //표준출력을 파이프의 입구 부분과 연결  
            if(close(fd[1])==-1 || close(fd[0])==-1) perror("close error!"); //파일 디스크립터  
            execvp(vec1[0], vec1);  
            perror("exec error!");  
            exit(0);  
        default:  
            //리다이렉션이 있는지를 확인.  
            cmd_redir_in(vec2);  
            cmd_redir_out(vec2);  
            close(STDIN_FILENO);  
            dup2(fd[0],STDIN_FILENO); //표준입력을 파이프의 출구 끝과 연결  
            if(close(fd[1])==-1 || close(fd[0])==-1) perror("close error!");  
            execvp(vec2[0], vec2);  
            perror("exec error!");  
            exit(0);  
    }  
    return 0;  
}
```

- ➔ 파이프를 생성하여, 부모 자식 프로세스간에 파이프를 파일 디스크립터를 통해 연결해 두 프로세스가 데이터를 주고받을 수 있도록 구현하였습니다.

3) 이중 파이프 구현 함수

```
int cmd_pipe_proc3(char* vec1[],char* vec2[],char* vec3[]){
    int fd[2],fd2[2];
    if(pipe(fd)==-1){
        perror("pipe failed!");
        exit(1);
    }
    //파이프를 하나 더 생성해준다.

    switch(fork()){
        case 0:
            if(pipe(fd2)==-1){
                perror("pipe failed!");
                exit(1);
            }
            switch (fork()){
                case 0:
                    //리다이렉션이 있는지를 확인.
                    cmd_redir_in(vec1);
                    cmd_redir_out(vec1);
                    //표준 출력을 2번 파이프의 입구로 연결한다.
                    dup2(fd2[1],STDOUT_FILENO);
                    //파이프 두개가 복제되었으므로 모두 닫아준다.
                    if( close(fd[0])==-1 || close(fd[1])==-1 || close(fd2[0])==-1 || close(fd2[1])==-1)
                        perror("close error!");
                    execvp(vec1[0],vec1);
                    exit(0);

                    default:
                        cmd_redir_in(vec2);
                        cmd_redir_out(vec2);
                        close(STDOUT_FILENO);
                        //표준 입력을 2번 파이프의 출구에 연결한다.
                        dup2(fd2[0],STDIN_FILENO);
                        //표준 출력을 1번 파이프의 입구에 연결한다.
                        dup2(fd[1],STDOUT_FILENO);
                        if( close(fd[0])==-1 || close(fd[1])==-1 || close(fd2[0])==-1 || close(fd2[1])==-1)
                            perror("close error!");
                        execvp(vec2[0],vec2);
                        wait(NULL);
            }
        }
    }
    exit(0);
}
```

```
        default:
            //표준 입력을 1번 파이프의 입구에 연결한다.
            cmd_redir_in(vec3);
            cmd_redir_out(vec3);
            close(STDIN_FILENO);
            dup2(fd[0],STDIN_FILENO);
            if( close(fd[0])==-1 || close(fd[1])==-1) perror("close error!");
            execvp(vec3[0],vec3);
            wait(NULL);
    }
}
```

4) 메인 함수 부

```
int pipeNum=cmd_pipe_check();

if(strcmp(cmdvector[0],"cd")==0){
    cmd_cd(cmdNum,cmdvector);
}
else if (strcmp(cmdvector[0],"exit")==0){
    exit(1);
}
else{
    switch(pid=fork()){
        case 0:
            //프롬프트로 입력받은 첫번째 단어를 실행
            if(backgroundCheck!=1){
                signal(SIGQUIT,SIG_DFL);
                signal(SIGINT,SIG_DFL);}
            else{
                signal(SIGINT,SIG_IGN);
                signal(SIGQUIT,SIG_IGN);}
            if(pipeNum==1){
                cmd_pipe_proc2(pipevec1,pipevec2);
                continue;
            }
            if(pipeNum==2){
                cmd_pipe_proc3(pipevec1,pipevec2,pipevec3);
                continue;}
            cmd_redir_in(cmdvector);
            cmd_redir_out(cmdvector);
            execvp(cmdvector[0], cmdvector);
            fatal("main()");
            return 0;
    }
}
```

-> 우선 파이프가 있는지 확인하고, 파이프 개수가 1개라면 단일 파이프를 구현하는 함수인 cmd_pipe_proc2를 실행하고, 파이프 개수가 2개라면 cmd_pipe_proc3를 실행합니다. 파이프를 수행했다면 continue로 밑의 코드는 생략하고 다음 루프로 넘어갑니다.

4. 테스트 사항

1) 단일 파이프

- ls > ls.txt | grep txt

```
myshell> ls > ls.txt | grep txt
b.txt
b1.txt
bbb.txt
ls.txt
ls2.txt
t.txt
test.txt
test1.txt
test2.txt
test_1.txt
u.txt
wow.txt
y.txt
```

-cat ls.txt | wc > test4.txt

```
myshell> cat ls.txt | wc > test4.txt
myshell> cat test4.txt
    19    19   158
```

```
jyullee@jyullee-VirtualBox:~/문서/Shell Project$ cat ls.txt | wc > test5.txt
jyullee@jyullee-VirtualBox:~/문서/Shell Project$ cat test5.txt
    19    19   158
```

➔ 단일 파이프는 모든 테스트 사항이 잘 실행되었습니다.

2) 다중 파이프는 조교님이 주신 테스트 사항이 완벽하게 실행되지는 않아 다른 테스트를 몇가지 수행해 보았습니다.

1) ls | grep t | grep -c 1

```
myshell> ls | grep t | grep -c 1
3
```

```
jyullee@jyullee-VirtualBox:~/문서/Shell Project$ ls | grep t | grep -c 1
3
```

➔ 원래 터미널과 동일한 결과값이 잘 나옵니다.

2) ls | wc | wc

```
myshell> ls | wc | wc
      1      3     24
```

```
jiyullee@jiyullee-VirtualBox:~/문서/Shell Project$ ls | wc | wc
      1      3     24
```

3) cat ls.txt | grep t | wc > t_num.txt

```
myshell> cat ls.txt
합계 152
-rwxr-xr-x 1 jiyullee jiyullee 18336 11월 29 18:16 a.out
-rw-r--r-- 1 jiyullee jiyullee    0 11월 29 04:43 b.txt
-rw-r--r-- 1 jiyullee jiyullee   29 11월 26 05:06 b1.txt
-rw-r--r-- 1 jiyullee jiyullee  342 11월 28 03:03 bbb.txt
-rw-r--r-- 1 jiyullee jiyullee    0 11월 29 18:27 dirA
-rw-r--r-- 1 jiyullee jiyullee   24 11월 29 18:27 dir_num.txt
-rw-r--r-- 1 jiyullee jiyullee   24 11월 29 18:28 dir_num2.txt
-rw-r--r-- 1 jiyullee jiyullee    0 11월 29 18:29 ls.txt
-rw-r--r-- 1 jiyullee jiyullee    0 11월 29 03:24 ls2.txt
-rw-rw-r-- 1 jiyullee jiyullee 10672 11월 29 08:07 shell.c
-rwxr-xr-x 1 jiyullee jiyullee 18328 11월 29 07:53 shell.out
-rw-rw-r-- 1 jiyullee jiyullee 10366 11월 29 18:21 simple_shell.c
-rw-r--r-- 1 jiyullee jiyullee  4384 10월 16 19:50 simple_shell.o
-rw-r--r-- 1 jiyullee jiyullee   16 11월 25 22:36 t.txt
-rw-r--r-- 1 jiyullee jiyullee   33 11월 29 02:31 test.txt
-rw-r--r-- 1 jiyullee jiyullee   33 11월 29 02:31 test1.txt
-rw-r--r-- 1 jiyullee jiyullee    3 11월 29 03:16 test2.txt
-rw-r--r-- 1 jiyullee jiyullee    2 11월 29 04:45 test3.txt
-rw-r--r-- 1 jiyullee jiyullee   24 11월 29 04:46 test4.txt
-rw-r--r-- 1 jiyullee jiyullee   24 11월 29 04:46 test5.txt
-rw-r--r-- 1 jiyullee jiyullee   33 11월 29 02:30 test_1.txt
-rw-r--r-- 1 jiyullee jiyullee   15 11월 25 21:14 u.txt
-rw-r--r-- 1 jiyullee jiyullee   55 11월 29 00:03 wow.txt
-rw-rw-r-- 1 jiyullee jiyullee  4033 11월 28 23:49 y.c
-rwxr-xr-x 1 jiyullee jiyullee 13640 11월 28 02:54 y.out
-rw-r--r-- 1 jiyullee jiyullee    85 11월 25 01:35 y.txt
```

```
myshell> cat ls.txt | grep t | wc > t_num.txt
myshell> cat t_num.txt
      21      189     1271
```

```
jiyullee@jiyullee-VirtualBox:~/문서/Shell Project$ cat ls.txt | grep t | wc > t
num2.txt
jiyullee@jiyullee-VirtualBox:~/문서/Shell Project$ cat tnum2.txt
      21      189     1271
```

➔ 기존 터미널과 동일한 결과값이 잘 나왔습니다.

➔ 테스트 사항이 모두 완벽하게 구현되지 못한 점이 아쉽다고 생각합니다. 이후에는 좀더 보완하여 모든 테스트 사항을 수행할 수 있는 셸을 만들어 보고 싶습니다.