

Course	고급시스템 프로그래밍
Instructor	황선태
report ID	Project 3
Due date	2014.10.29

Department	Computer Engineering
Student id.	20093284
Student name	나홍철
Submission date	2014.10.29

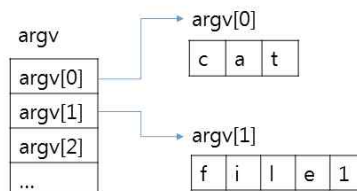
1. 자신의 접근 방법, 구현 방법 등을 기술, UNIX System Call (fork, exec, wait)의 Study 결과

* 접근 & 구현 방법

- **명령어 처리** : Mini Shell 프로그램은 실제 리눅스 Shell 프로그램의 일부 기능을 수행한다. 명령어 스트링을 무한히 입력받으며 입력받은 스트링은 이를 분해하고 분석하는 함수를 수행한다. 실제 Shell과 같이 프로그램들을 하나의 스트링으로 입력을 받아 token단위로 구분하여 명령을 실행하고 관리한다. 입력된 스트링들은 Space Bar 기준으로 분해가 되도록 하였다. 명령어를 실행하고 관리하기에 좀 더 편리하고 가독성이 좋아지기 때문이다.

- **프로세스 정보** : 각각의 명령어를 수행하기 위한 프로세스의 정보를 구조체로 따로 선언해 두었으며 해당 프로세스 구조체는 File Descriptor[Standard_IN, Standard OUT, Standard Error] 배열과 명령어를 담은 char형 2차 배열과 입력받은 인수의 개수를 알려주는 argc변수와 명령어를 실행하는 프로세스의 Process ID(PID)와 Process Group ID(PGID) 변수를 포함하고 있다. 이 변수들은 새로운 명령어를 수행하게 될 때마다 초기화 된다.

[Process Structure]



fd	
fd[0]	STD_IN
fd[1]	STD_OUT
fd[2]	STD_ERROR

- **프로세스 복제 & 수행** : 명령어를 실행하기 위해서는 프로세스를 복제하는 **fork**함수를 사용하여 생성된 자식 프로세스에서 명령어가 수행이 가능하도록 하였다. 이 기능을 하는 함수를 따로 구현을 하여 호출이 가능하면서 자식프로세스의 PID값을 반환 할 수 있도록 구현하였다. fork함수는 PID를 제외한 프로세스의 모든 것을 똑같이 복제하며 복제된 이후의 코드를 수행한다. 이때 fork함수를 수행 한 후 반환되는 값은 부모 프로세스는 자식의 PID를 자식 프로세스는 '0'을 반환 받는다. 이 반환값을 활용하여 부모 프로세스는 자식프로세스의 PID를 반환하는 코드를 수행한다. 반면 자식 프로세스는 **execv**함수를 수행하며 입력된 명령어를 수행하는 프로세스로 변경이 된다. 이 execv함수는 명령어의 위치를 함수의 첫 번째 인자가 되면 이후 명령어를 포함한 인자들의 2차 배열 포인터를 두 번째 인자로 넘겨준다. 그러면 명령어가 수행된다. 이 후 프로그램이 완료되면 부모 프로세스에게 반환된다. 이 때 부모 프로세스는 **waitpid**함수를 수행하여 자식 프로세스들의 수행 상태를 반환 받는다. 자식프로세스를 wait하지 않는 경우 종료를 해도 커널 상에 반환값이 떠돌게 되는 좀비 프로세스가 된다. 이를 방지하기 위해 PID가 0번인 프로세스가 자동으로 부모 프로세스가 되고 이 좀비 프로세스들을 wait함수로 받아들인다.

- **명령어 보조** : 명령어 수행의 보조적인 역할로서 Shell만의 특별한 기능들이 있다. 이 기능들은 대표적으로 Back Ground, Pipe, Redirection Standard In, Redirection Standard Out, Redirection Standard Error 가 있다. 기능들을 살펴보면 아래와 같다.

① BackGround ('&')

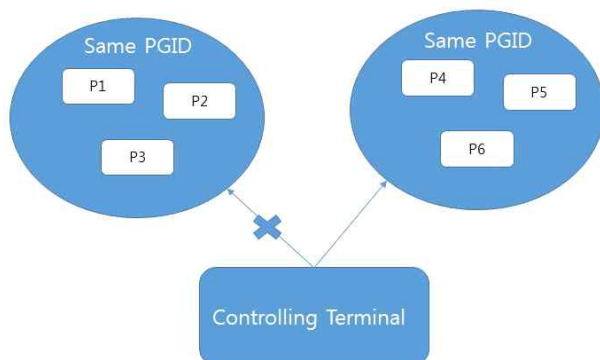
- 명령어를 실행하는 자식 프로세스들이 부모 프로세스인 Shell로부터 Controlling Terminal 명령을 받지 않도록 하는 것이다. 즉, 그들만의 그룹을 이루는 것이다.

- 그들만의 그룹을 이루기 위해서는 부모 프로세스로부터 복제된 처음 자식 프로세스의 PGID로 한 이후 파이프를 연결된 프로세스들은 처음 프로세스의 PGID값을 PGID로 갖는다. 이렇게 백그라운드 수행이 되는 프로세스들은 처음 프로세스를 그룹리더로 하는 하나의 그룹을 이루게 된다. 또한 백그라운드 실행은 Shell로부터 Controlling Terminal 명령을 받지 않고 독립되어 수행이 되어야 한다. 그러하기 위해서는 부모프로세스인 Shell은 waitpid함수를 수행 하지 않으면서 자식 프로세스의 죽음을 기다리지 않고 자신의 업무를 수행한다. 이렇게 하면 자식 프로세스는 좀비 프로세스로 남게 되며 init프로세스의 자식으로 exit를 하게 되면 init의 wait에 반납이 된다. 예를 들면

```
$ P1 | P2 | P3 &
```

```
$ P4 | P5 | P6
```

일 때 P1, P2, P3은 백그라운드 실행을 하면서 같은 그룹ID를 갖는다. 반면 P4, P5, P6은 포그라운드 실행을 수행한다.



② Pipe ('|')

- 다수의 명령어들이 서로 파이프를 통해서 연결이 된다. 이 때 연결되는 File Descriptor의 번호는 출력하는 프로그램의 경우 Standard Out이 파이프의 write와 연결되며 입력받는 프로그램의 경우 Standard In이 파이프의 read와 연결된다. 이렇게 되면 프로세스간 단방향 통신이 가능하게 된다. 이 때 연결된 파이프는 두 프로세스 간 한방향으로만 연결이 되어야 한다. 그러므로 다른 프로세스들 혹은 양방향으로 연결이 되어있다면 정리를 해주어야 한다.

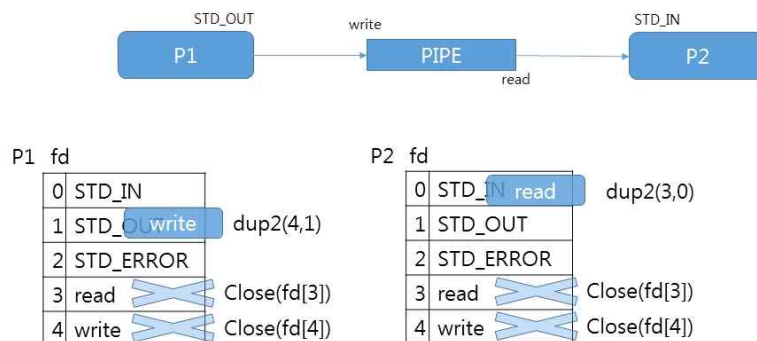
- 여러 프로세스간 통신이 가능하기 위해서는 여러개의 파이프를 필요로 한다. 가능한 프로

세스의 개수에서 한 개 적은 수만큼 파이프를 미리 만들어 놓는다. 파이프는 pipe함수로 생성이 가능하며 정수형 배열을 인자로 전달하면 그 인자에 open된 파이프의 File Descriptor 번호가 저장된다. 그 후 파이프를 수행해야 한다면 각 프로세스의 File Descriptor를 파이프의 File Descriptor 번호로 교체를 해준다. Mini Shell에서는 프로세스 구조체에 있는 File Descriptor 배열에 교체를 한다. 이 후 교체가 된 번호만 dup2함수를 통해서 실제 프로세스 File Descriptor 테이블에 교체를 한다.

더욱 자세히 보면 파이프를 수행하기 전 프로세스는 Standard Out이 파이프에 연결되며 수행 후 프로세스는 Standard In이 파이프에 연결된다. 예를 들면

\$ P1 | P2

일 때 P1의 Standard Out은 파이프의 write에 연결되고 P2의 Standard In은 파이프의 read에 연결이 된다. 이는 P1프로세스는 File Descriptor 테이블에서 파이프의 write 번호는 dup2함수를 사용해서 복제하여 Standard Out에 할당하고 모든 파이프의 Descriptor번호를 close함수를 사용하여 닫아준다. P2프로세스는 File Descriptor 테이블에서 파이프의 read 번호는 dup2함수를 사용해서 복제하여 Standard In에 할당하고 모든 파이프의 Descriptor번호를 close함수를 사용하여 닫아준다. 이렇게 하면 프로세스 간 연결이 가능해진다. 이 후 자식프로세스와 부모 프로세스는 생성된 모든 파이프들을 닫아주어 두 프로세스만이 통신이 가능케 한다.



③ Redirection Standard In ('<')

- 명령어 프로세스의 Standard In File Descriptor가 파일로 Redirection된다.
- 명령어 프로세스에서 입력을 받는 곳이 터미널의 키보드가 아닌 파일로부터 입력을 받게 된다. 이때 System Call인 read함수나 write함수는 Unbuffered I/O이므로 버퍼를 지정해 주고 사용하기 때문에 상관이 없다. 반면 Standard I/O인 fopen, scanf, getc...등의 함수들은 Buffered I/O이므로 버퍼를 기본적으로 제공받는다. 하지만 터미널에 연결 되어 있을 때는 line buffered방식으로 '\n'을 만나면 함수의 수행을 멈추지만 파일에 연결 된다면 fully buffered방식이 되므로 버퍼가 가득차기 전까지는 함수가 멈추지 않는다 그러므로 수동적으로 버퍼를 flush 해주어야 한다. 이는 Redirection Standard Out, Redirection Standard Error를 사용 할 때와 파이프를 사용할 때 마찬가지로 적용된다.

④ Redirection Standard Out (' > ')

- 명령어 프로세스의 Standard Out File Descriptor가 파일로 Redirection된다.
- 명령어 프로세스에서 출력을 하는 곳이 터미널의 모니터가 아닌 파일이 된다. 명령어의 수
행 출력내용이 지정한 파일에 저장된다.

⑤ Redirection Standard Error (' 2> ')

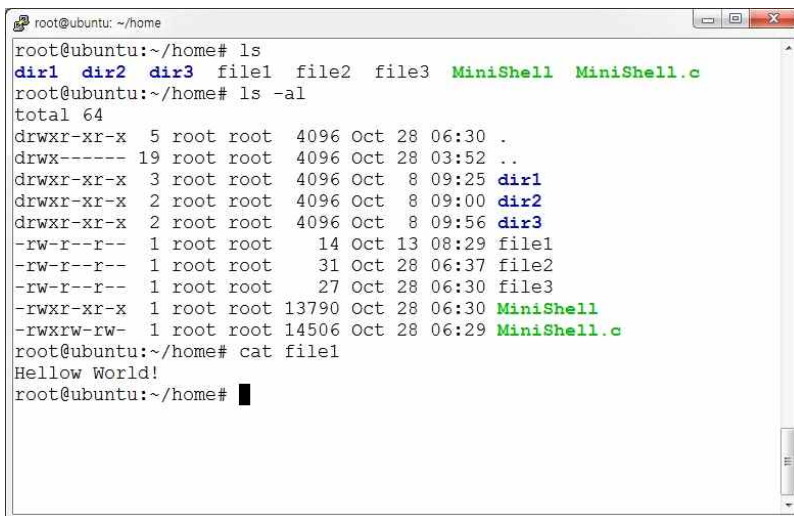
- 명령어 프로세스의 Standard Error File Descriptor가 파일로 Redirection된다.
- Redirection Standard Out과 같은 기능이지만 출력되는 내용은 명령어 프로세스에서 오
류로 출력되는 내용들이 파일에 출력된다.

2. TESTCASE

명령어 실행

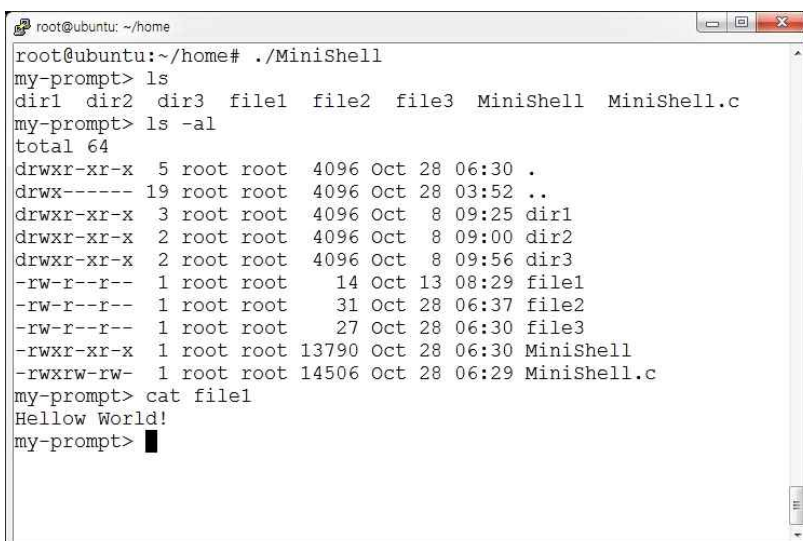
- 명령어 'ls' 와 'cat'을 사용하여 실제 리눅스의 쉘처럼 기능을 수행하는지 검사하였다.
- ls 명령어를 통해서 디렉토리 내의 파일들의 출력물이 리눅스의 쉘과 MiniShell과 결과물이 같음을 확인 하였다.
- cat 명령어를 통해서 파일 내부의 내용이 리눅스의 쉘과 MiniShell과 결과물이 같음을 확인 하였다.

< 리눅스 Shell 수행 장면 >



```
root@ubuntu: ~/home
root@ubuntu:~/home# ls
dir1 dir2 dir3 file1 file2 file3 MiniShell MiniShell.c
root@ubuntu:~/home# ls -al
total 64
drwxr-xr-x  5 root root  4096 Oct 28 06:30 .
drwx----- 19 root root  4096 Oct 28 03:52 ..
drwxr-xr-x  3 root root  4096 Oct  8 09:25 dir1
drwxr-xr-x  2 root root  4096 Oct  8 09:00 dir2
drwxr-xr-x  2 root root  4096 Oct  8 09:56 dir3
-rw-r--r--  1 root root    14 Oct 13 08:29 file1
-rw-r--r--  1 root root   31 Oct 28 06:37 file2
-rw-r--r--  1 root root   27 Oct 28 06:30 file3
-rwxr-xr-x  1 root root 13790 Oct 28 06:30 MiniShell
-rwxrw-rw-  1 root root 14506 Oct 28 06:29 MiniShell.c
root@ubuntu:~/home# cat file1
Hellow World!
root@ubuntu:~/home#
```

< MiniShell 수행 장면 >

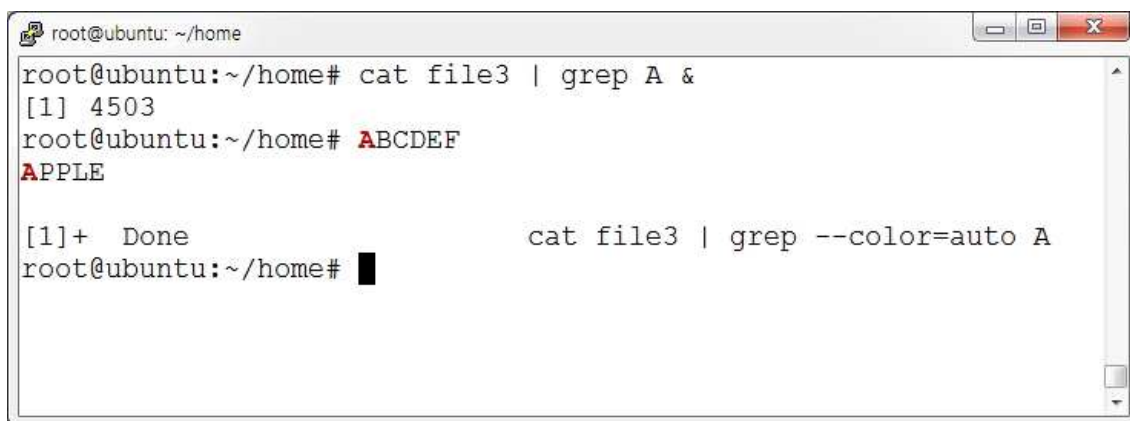


```
root@ubuntu: ~/home
root@ubuntu:~/home# ./MiniShell
my-prompt> ls
dir1 dir2 dir3 file1 file2 file3 MiniShell MiniShell.c
my-prompt> ls -al
total 64
drwxr-xr-x  5 root root  4096 Oct 28 06:30 .
drwx----- 19 root root  4096 Oct 28 03:52 ..
drwxr-xr-x  3 root root  4096 Oct  8 09:25 dir1
drwxr-xr-x  2 root root  4096 Oct  8 09:00 dir2
drwxr-xr-x  2 root root  4096 Oct  8 09:56 dir3
-rw-r--r--  1 root root    14 Oct 13 08:29 file1
-rw-r--r--  1 root root   31 Oct 28 06:37 file2
-rw-r--r--  1 root root   27 Oct 28 06:30 file3
-rwxr-xr-x  1 root root 13790 Oct 28 06:30 MiniShell
-rwxrw-rw-  1 root root 14506 Oct 28 06:29 MiniShell.c
my-prompt> cat file1
Hellow World!
my-prompt>
```

BACKGROUND (“&”) 실행

- 명령어 ‘cat’, ‘grep’의 백그라운드 실행이 가능한지 알아보았다.
- 실제 리눅스 셸처럼 백그라운드 프로세스를 기다리지 않고 셸이 실행되어 실행된 명령어의 출력물들이 불규칙적으로 출력되었음을 알 수 있었다.

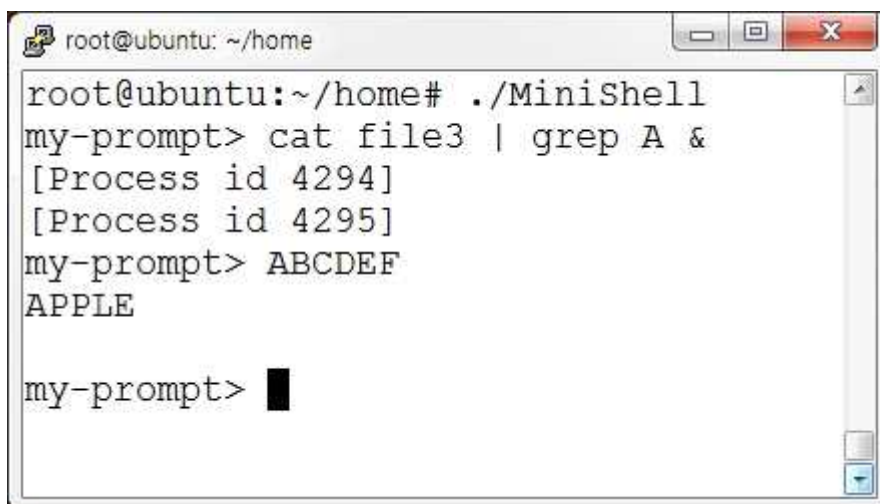
< 리눅스 Shell 수행 장면 >



```
root@ubuntu: ~/home
root@ubuntu:~/home# cat file3 | grep A &
[1] 4503
root@ubuntu:~/home# ABCDEF
APPLE

[1]+  Done                  cat file3 | grep --color=auto A
root@ubuntu:~/home#
```

< MiniShell 수행 장면 >



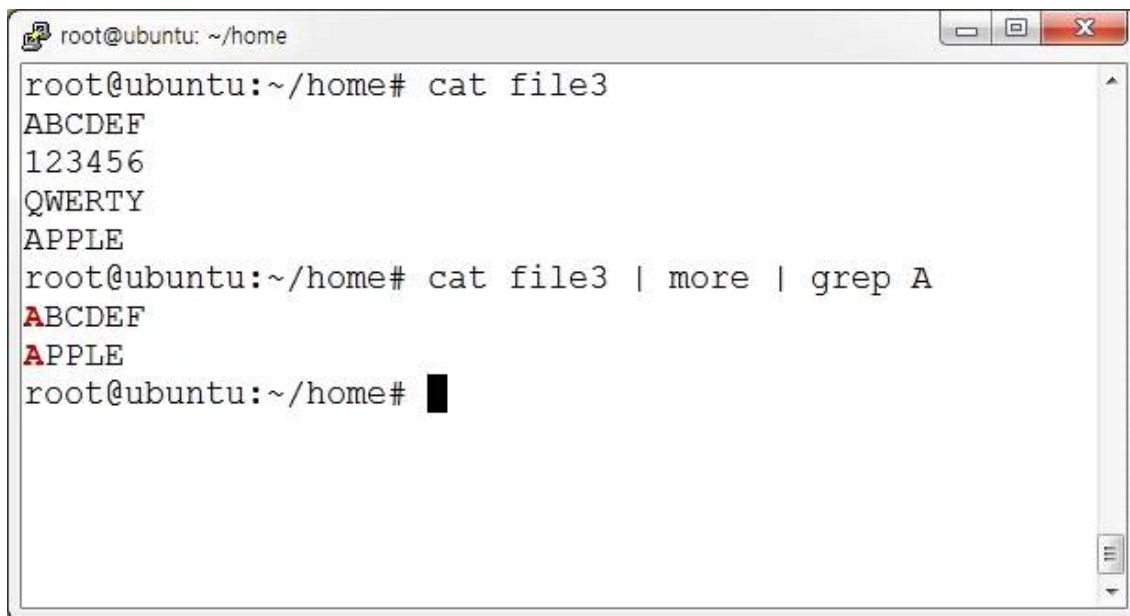
```
root@ubuntu: ~/home
root@ubuntu:~/home# ./MiniShell
my-prompt> cat file3 | grep A &
[Process id 4294]
[Process id 4295]
my-prompt> ABCDEF
APPLE

my-prompt>
```

PIPE (“|”) 실행

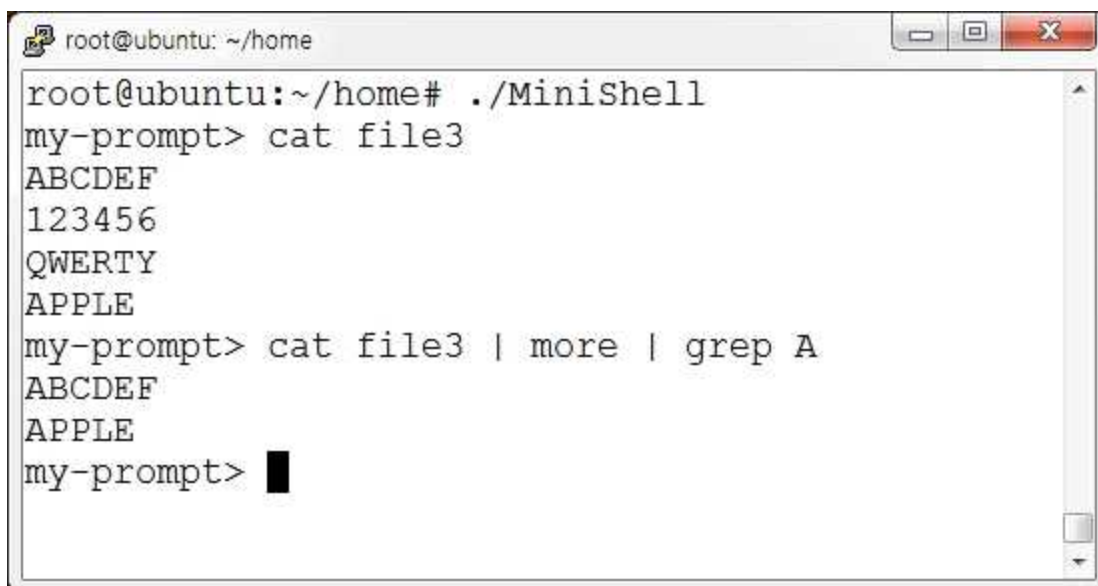
- 명령어 ‘cat’을 통해서 읽어온 파일의 내용의 출력이 ‘more’명령어를 통해서 ‘grep’ 명령어를 수행 할 수 있는지 확인 하였다.
- 실제 리눅스 셸과 같은 출력물의 결과를 얻을 수 있었다.

< 리눅스 Shell 수행 장면 >



```
root@ubuntu: ~/home
root@ubuntu:~/home# cat file3
ABCDEF
123456
QWERTY
APPLE
root@ubuntu:~/home# cat file3 | more | grep A
ABCDEF
APPLE
root@ubuntu:~/home#
```

< MiniShell 수행 장면 >

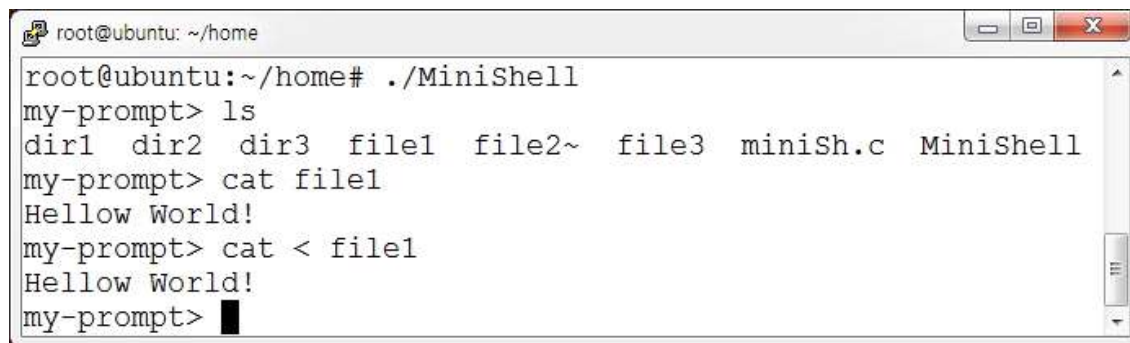


```
root@ubuntu: ~/home
root@ubuntu:~/home# ./MiniShell
my-prompt> cat file3
ABCDEF
123456
QWERTY
APPLE
my-prompt> cat file3 | more | grep A
ABCDEF
APPLE
my-prompt>
```


REDIRECTION_IN (“<”) 실행

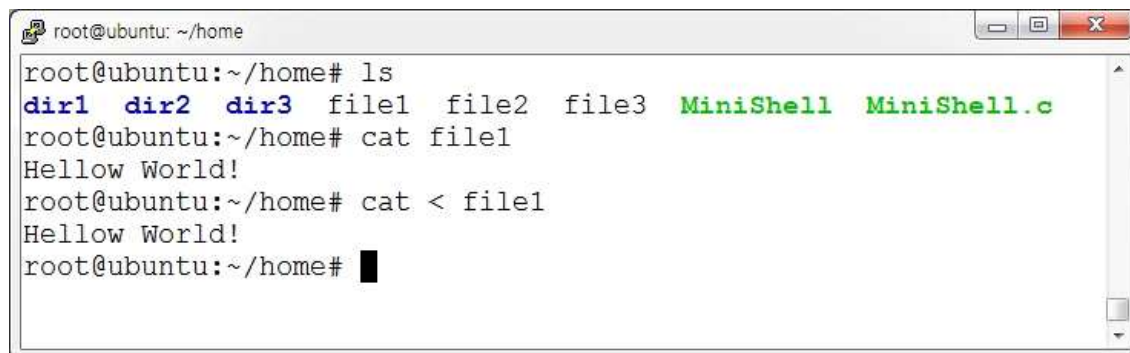
- 명령어 ‘cat’을 통해서 파일의 Standard In을 파일로 변경해 보았다.
- 실제 리눅스 셸과 같은 출력 결과물을 수행함을 알 수 있다.

< 리눅스 Shell 수행 장면 >

A terminal window titled 'root@ubuntu: ~/home' showing the execution of a custom shell named 'MiniShell'. The user runs './MiniShell' and enters 'my-prompt>'. The prompt changes to 'my-prompt>'. The user runs 'ls' and sees a list of files and directories. Then, the user runs 'cat file1' and 'cat < file1', both of which output 'Hellow World!'.

```
root@ubuntu: ~/home
root@ubuntu:~/home# ./MiniShell
my-prompt> ls
dir1 dir2 dir3 file1 file2~ file3 miniSh.c MiniShell
my-prompt> cat file1
Hellow World!
my-prompt> cat < file1
Hellow World!
my-prompt> █
```

< MiniShell 수행 장면 >

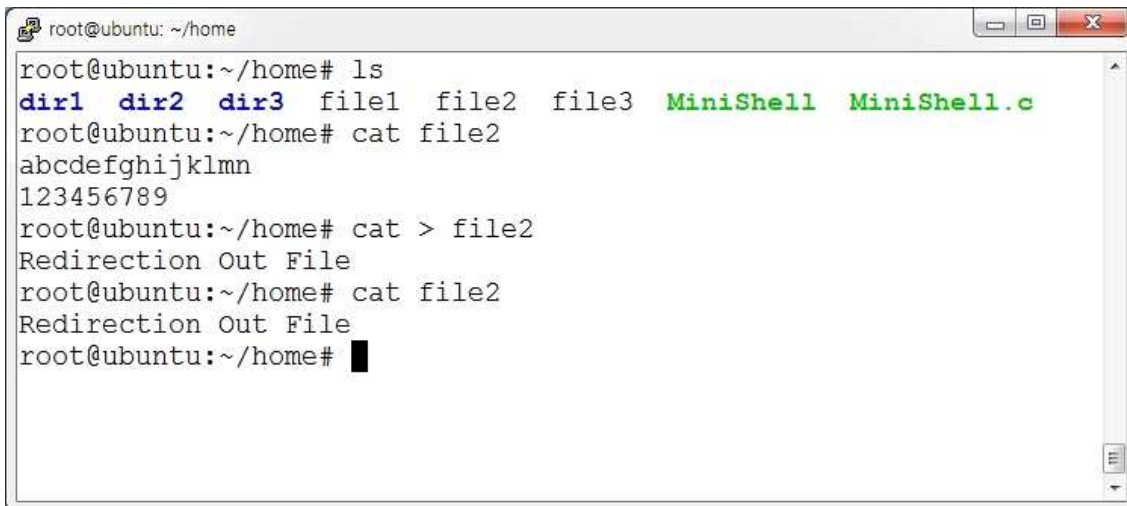
A terminal window titled 'root@ubuntu: ~/home' showing the execution of the 'ls' command. The output is color-coded: directories are blue, files are green, and executables are red. The user then runs 'cat file1' and 'cat < file1', both of which output 'Hellow World!'.

```
root@ubuntu: ~/home
root@ubuntu:~/home# ls
dir1 dir2 dir3 file1 file2 file3 MiniShell MiniShell.c
root@ubuntu:~/home# cat file1
Hellow World!
root@ubuntu:~/home# cat < file1
Hellow World!
root@ubuntu:~/home# █
```

REDIRECTION_OUT (“>”) 실행

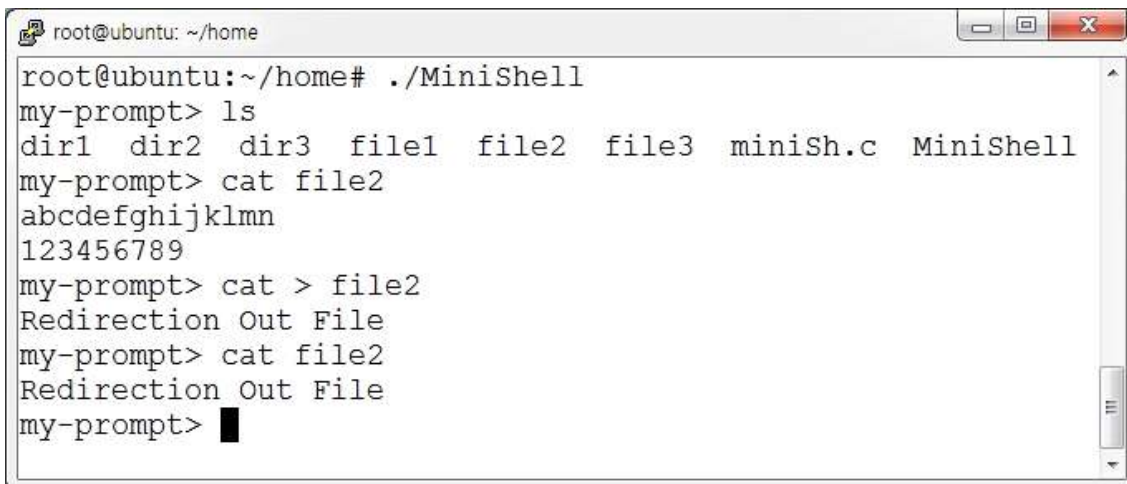
- 명령어 ‘cat’의 Standard Out을 file로 변경하여 파일에 출력이 가능하도록 했다.
- 실제 리눅스 셸과 같은 파일 출력이 되었음을 알 수 있다.

< 리눅스 Shell 수행 장면 >



```
root@ubuntu: ~/home
root@ubuntu:~/home# ls
dir1 dir2 dir3 file1 file2 file3 MiniShell MiniShell.c
root@ubuntu:~/home# cat file2
abcdefghijklmn
123456789
root@ubuntu:~/home# cat > file2
Redirection Out File
root@ubuntu:~/home# cat file2
Redirection Out File
root@ubuntu:~/home#
```

< MiniShell 수행 장면 >



```
root@ubuntu: ~/home
root@ubuntu:~/home# ./MiniShell
my-prompt> ls
dir1 dir2 dir3 file1 file2 file3 miniSh.c MiniShell
my-prompt> cat file2
abcdefghijklmn
123456789
my-prompt> cat > file2
Redirection Out File
my-prompt> cat file2
Redirection Out File
my-prompt>
```

REDIRECTION_ERROR_OUT (“2>”) 실행

- 명령어 ‘cat’으로 디렉토리를 열었을 때 출력되는 에러 메시지를 파일에 출력하도록 기능하는 Standard Error 변경을 해보았다.
- 실제 리눅스 셸처럼 에러를 출력 받은 파일이 같은 출력내용이 입력 됐음을 알 수 있었다.

< 리눅스 Shell 수행 장면 >



```
root@ubuntu: ~/home
root@ubuntu:~/home# ls
dir1  dir2  dir3  file1  file2  file3  MiniShell  MiniShell.c
root@ubuntu:~/home# cat file2
Redirection Out File
root@ubuntu:~/home# cat dir1 2> file2
root@ubuntu:~/home# cat file2
cat: dir1: Is a directory
root@ubuntu:~/home#
```

< MiniShell 수행 장면 >

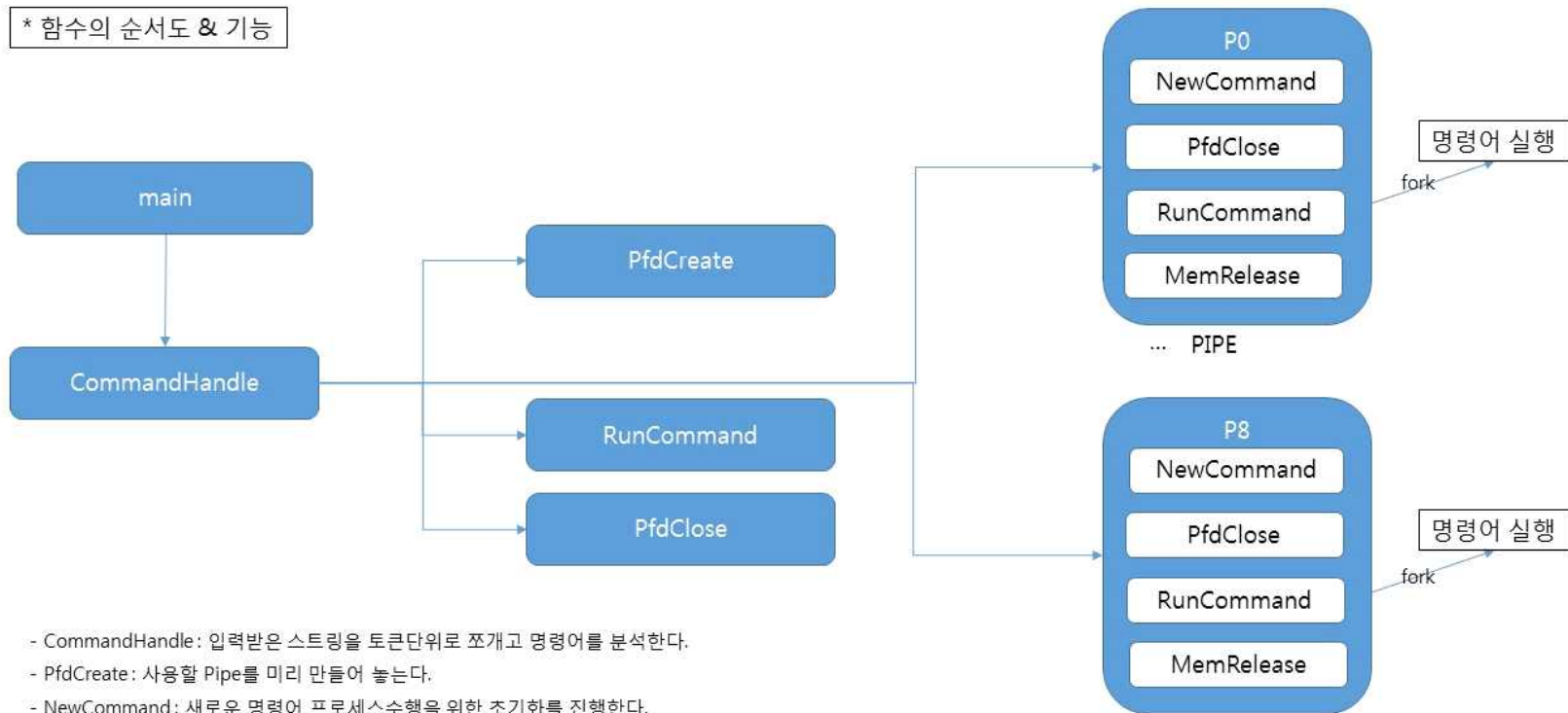


```
root@ubuntu: ~/home
root@ubuntu:~/home# ./MiniShell
my-prompt> ls
dir1  dir3  file2  MiniShell
dir2  file1  file3  MiniShell.c
my-prompt> cat file2
Redirection Out File
my-prompt> cat dir1 2> file2
my-prompt> cat file2
/bin/cat: dir1: Is a directory
my-prompt>
```

3. 부록: 소스 코드

[함수 순서도]

* 함수의 순서도 & 기능



- CommandHandle: 입력받은 스트링을 토큰단위로 쪼개고 명령어를 분석한다.
- PfdCreate: 사용할 Pipe를 미리 만들어 놓는다.
- NewCommand: 새로운 명령어 프로세스수행을 위한 초기화를 진행한다.
- RunCommand: 부모 & 자식 프로세스가 생성되는 함수다. 부모프로세스는 자식프로세스를 생성하고 자식프로세스는 명령어 기능을 수행한다.
- PfdClose: 생성되어 연결되었던 Pipe들을 모두 닫아준다.
- MemRelease: 동적으로 할당된 명령어 인자들의 메모리를 회수한다.

[MiniShell.c]

```
/*-----  
고급시스템 프로그래밍 : Mini Shell  
  
국민대학교 컴퓨터공학과  
20093284 나홍철  
-----*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/wait.h>  
#include <fcntl.h>  
  
#define TRUE 1  
#define FALSE 0  
  
#define AMPERSAND "&" /* Background */  
#define PIPE "|" /* Pipe */  
#define REDIR_IN "<" /* Redirection Input */  
#define REDIR_OUT ">" /* Redirection Output */  
#define REDIR_EOUT "2>" /* Redirection Error Output */  
  
#define PATH "/bin/" /* Command prgoram path */  
#define END_CMD "goodbye" /* Exit shell command */  
#define PROMPT "my-prompt>" /* Prompt string */  
#define SPACE_BAR " "  
#define NEW_LINE "\n"  
  
#define MAX_ARG 8 /* Maximum number of arguments */  
#define MAX_PROCESS 8 /* Maximum number of process */  
#define MAX_LENGTH 32 /* Maximum string length of one argument */  
#define MAX_LINE 256 /* Maximum length of letters at single line */  
#define MAX_FDTABLE 3 /* Maximum size of descriptor table  
0=Standard In, 1=Standard Out, 2=Standard Error Out */  
#define MAX_PFDTABLE 2 /* Maximum size of Pipe descriptor table */
```

```

/**
 * [ Process ]
 * 이 구조체는 하나의 명령어를 실행하는데 필요한 프로세스의 정보를
 * 담고 있는 구조체다. 하나의 명령어가 실행 될 때 사용되는 정보로서
 * 현재 MiniShell에서는 총 8개의 명령어 프로세스를 사용 할 수 있다.
 */
struct process
{
    int fd[MAX_FDTABLE]; /* 한 명령어 프로세스 Standard In, Standard Out,
                          Standard error out 디스크립터 번호 테이블 배열이다.
                          이 테이블 값의 변동이 있을 때 프로세스의
                          파일 디스크립터도 변경시켜준다.*/

    char **argv; /* 한 명령어 프로세스의 입력받은 명령어 배열 */
    int argc; /* 한 명령어 프로세스의 입력받은 인수의 총 개수다. */
    int pid; /* 한 명령어 프로세스의 pid */
    int pgid; /* 한 명령어 프로세스의 pgid => 백그라운드 실행 시 백그라운되는
              처음 프로세스를 그룹리더로 다른 프로세스들을 같은 그룹으로 묶어준다 */
};

/**
 * [ Memory Release ]
 * 2차배열로 할당된 메모리를 회수하는 함수다.
 */
int MemRelease(struct process *p)
{
    int i;

    /* 각 1차 배열 메모리 회수 */
    for (i = 0; i < p->argc; i++)
    {
        free(p->argv[i]);
        if (p->argv[i] != NULL)
            return 1;
    }
    /* 2차 배열 메모리 회수 */
    free(p->argv);
    return 0;
}

```

```

/**
 * [ Pipe file descriptor Create ]
 * MiniShell에서 사용 될 pipe를 미리 입력가능한 명령어 개수 만큼 생성한다.
 * 프로세서에 생성된 파이프의 파일 디스크립터 번호를 사용하기 위해
 * 이미 생성한 파이프 전용 테이블배열에 저장하는 함수다.
 */
int PfdCreate(int pfd[][MAX_PFDTABLE])
{
    int i;
    int fd[MAX_PFDTABLE]; /* 생성된 pipe의 파일 디스크립터
                           번호가 임시로 저장되는 정수배열 변수다. */

    /* 생성된 pipe의 파일 디스크립터 번호를 테이블배열 에 저장한다. */
    for (i = 0; i<MAX_ARG; i++)
    {
        if (pipe(fd) < 0)
            perror("ERROR pipe :");
        pfd[i][0] = fd[0];
        pfd[i][1] = fd[1];
    }
    return 0;
}

/**
 * [ Pipe file descriptor Close ]
 * 모든 pipe들을 프로세서의
 * 파일 디스크립터 테이블에서 제거한다.
 */
int PfdClose(int pfd[][MAX_PFDTABLE])
{
    int i;

    /* 생성된 pipe들을 모두 닫아 준다. */
    for (i = 0; i<MAX_ARG; i++)
    {
        close(pfd[i][0]);
        close(pfd[i][1]);
    }
    return 0;
}

```

```

/**
 * [ NewCommand ]
 * 새로운 명령어를 입력받을 때 사용하는 함수다. 새로운 구조체를 초기화하는
 * 함수역할을 한다. 입력받은 스트링을 " "(space bar)기준으로 잘라낸 토큰값이
 * 새로운 명령어일 경우 새로 할당해주며 명령어의 위치를 덧 붙여준다.
 */
int NewCommand(struct process *p, char *str)
{
    char *ptr;
    int i;

    p->argc = 0;    /* 입력받은 인자의 개수를 초기화 한다. */
    p->pid = 0;     /* PID번호를 초기화 한다. */
    p->pgid = 0;    /* PGID번호를 초기화 한다. */

    /* 명령어와 인자를 저장할 배열을 초기화 한다.*/
    if ((p->argv = (char**)malloc(sizeof(char*) * MAX_ARG)) == NULL)
    {
        perror("mem faild\n");
        exit(1);
    }

    /* 프로세스 구조체의 파일 디스크립터를 초기화한다. */
    for (i = 0; i < MAX_FDTABLE; i++)
        p->fd[i] = i;

    ptr = strtok(str, SPACE_BAR); /* 스트링을 space bar 기준으로 잘라낸다. */

    /* 명령어와 인자를 저장할 배열을 초기화 한다.*/
    if ((p->argv[p->argc] = (char *)malloc(MAX_LENGTH * sizeof(char)) ) == NULL)
    {
        perror("mem faild\n");
        exit(1);
    }

    sprintf(p->argv[p->argc], "%s%s", PATH, ptr); /* 명령어를 명령어의 위치에 덧 붙여준다. */

    return 0;
}

```



```

/**
 * [ Running Command ]
 * 명령어 실행을 위해 fork()를 한 후 해당되는 명령어 프로세스를
 * 실행 시켜주는 함수다. 혹시 pipe는 redirection이 있다면
 * Standard In이나 Standard Out을 해당 파일 디스크립터 번호로
 * 변경(dup2)시켜준다.
 */
int RunCommand(struct process *p, int pfd[][MAX_PFDTABLE])
{
    pid_t pid;
    int i;

    /* fork() 함수를 통해 프로세서를 복제한다.
    반환된 pid값이 0이라면 복제된 child 프로세스로서
    실행하고자 하는 명령을 실행한다. */
    switch (pid = fork()) {
    case -1:
        perror("miniSh");
        return (-1);
    case 0:
        /* 자식프로세스 */
        setpgid(0, p->pgid); /* 명령어 프로세스의 PGID를 미리 지정해놓은
                               Group ID로 변경해 준다 */

        /* 변경된 Standard In, Standard Out, Standard error out이 있다면
        적용시켜준다. */
        for (i = 0; i < MAX_FDTABLE; i++)
        {
            if (p->fd[i] != i)
                dup2(p->fd[i], i);
        }
        PfdClose(pfd); /* 프로세스의 파일 디스크립터 테이블에 등록된
                        pipe들 간 충돌이 일어나지 않도록 닫아주어 정리한다.*/
        execv(p->argv[0], p->argv); /* 명령을 실행한다. */
        perror("exec failed");
        return 1;
    default:
        /* 부모프로세스 */
        return pid;
    }
}

```

```

/**
 * [ Command Handle ]
 * MiniShell에서 가장 중요한 함수이다. 입력받은 스트링을
 * Spacebar기준으로 잘라낸다. 잘라내어진 토큰은 6가지 경우로 나누어진다.
 *
 *  AMPESAND : "&"인 경우 프로세스를 백그라운드 실행을 하기 위해 Background flag를 TRUE로
 *  변경시켜준다. 나중에 프로세스를 실행 한 후 부모 프로세스는
 *  자식프로세스를 wait하지 않고 좀비 프로세스로 남긴다.
 *
 *  PIPE : "|"인 경우 파이프 카운터를 1증가 시켜주어 명령어가 하나 반드시 온다는 것을
 *  알려주고 미리 초기화시켜 두었던 파이프의 파일디스크립터 번호테이블 배열값을
 *  각 프로세스의 파일디스크립터 테이블 배열 값에 입력한다.
 *  나중에 프로세스 실행 시 기존 Standard In, Standard Out 디스크립터 번호가 아니라면
 *  프로세스의 파일디스크립터 테이블에 할당(dup함수) 될 것이다.
 *  ex) $ cat file1 | grep a
 *  fd[0][0] => cat명령어의 standard in으로 원래 번호인 '0'
 *  fd[0][1] => cat명령어의 standard out으로 원래 번호가 아닌
 *  파이프의 write 디스크립터 번호를 저장한다.
 *  fd[1][0] => cat명령어의 standard in으로 원래 번호가 아닌
 *  파이프의 read 디스크립터 번호를 저장한다.
 *  fd[1][1] => cat명령어의 standard out으로 원래 번호인 '1'
 *
 *  REDIR_IN : "<"인 경우 프로세스의 Standard In은 "<" 이후에 오는 파일을
 *  read open한 디스크립터번호가 된다.
 *
 *  REDIR_OUT : ">"인 경우 프로세스의 Standard In은 ">" 이후에 오는 파일을
 *  write open한 디스크립터번호가 된다.
 *
 *  REDIR_EOUT : "2>"인 경우 프로세스의 Standard error out은 "<" 이후에 오는 파일을
 *  write open한 디스크립터번호가 된다.
 *
 *  그 외 인자인 경우 3차원 배열에 저장하여 추후 명령어 실행 시 인자값 리스트로 넘겨준다.
 */
int CommandHandle(char *str)
{
    struct process p[MAX_PROCESS]; /* 명령어 프로세스를 실행할 구조체로서
                                     총 8개의 프로세스 실행이 가능하다. */
    char *ptr; /* 나누어진 스트링 토큰주소의 처음을 가리키는 포인터다. */
    int pfd[MAX_PROCESS-1][MAX_PFDTABLE]; /* 생성된 파이프 디스크립터
                                             번호 테이블 배열이다.
                                             파이프의 총 개수는
                                             '실행가능한 프로세스의 총개수-1' 이다 */

    int pCnt; /*프로세스 카운트*/
    /* 스트링에 입력된 파이프의 개수로서 명령어의 개수를 판단 할 수 있는 정보로 사용된다. */
    int bFlag; /* 스트링에 백그라운드 실행이 존재하는가를 알려주는 백그라운드 flag다. */

```

```

int status;      /* 부모프로세스가 자식프로세스를 wait하여 반환된 상태를 저장 할 변수다. */
int i;

PfdCreate(pfd); /* 사용할 pipe를 가능한 명령어만큼 초기에 생성 해준다.*/

pCnt = 0;      /* 실행되는 명령어 프로세스의 개수다. */
bFlag = FALSE;

/* 새로운 명령어를 입력받아 명령어 2차배열의 처음주소에 할당한다. */
NewCommand(&pCnt, str);

/* 입력받은 스트링을 Spacebar(" ") 단위로 잘라 토큰(단어)을 생성한다. */
while (ptr = strtok(NULL, SPACE_BAR))
{
    /* 나누어진 토큰이 AMPESAND(&) */
    if (strcmp(ptr, AMPERSAND) == 0)
    {
        bFlag = TRUE;
        break;
    }

    /* 나누어진 토큰이 PIPE(|)
    PIPE를 기준은 왼쪽 명령어의 Standard Out이
    Pipe의 Write 디스크립터의 번호를 할당받고
    PIPE를 기준으로 오른쪽 명령어의 Standard In이
    Pipe의 read 디스크립터의 번호를 할당받는다.*/
    else if (strcmp(ptr, PIPE) == 0)
    {
        pCnt++;
        /* 새로운 명령어를 입력받아 명령어 2차배열의 처음주소에 할당한다. */
        NewCommand(&pCnt, NULL);

        p[pCnt - 1].fd[1] = pfd[pCnt - 1][1];
        p[pCnt].fd[0] = pfd[pCnt - 1][0];
    }

    /* 나누어진 토큰이 REDIR_IN(<) 일 때, 명령어의 Standard In을
    입력받은 파일을 Open한 디스크립터 번호를 할당한다. */
    else if (strcmp(ptr, REDIR_IN) == 0)
    {
        ptr = strtok(NULL, SPACE_BAR);
        p[pCnt].fd[0] = open(ptr, O_RDONLY);
    }

    /* 나누어진 토큰이 REDIR_OUT(>) 일 때, 명령어의 Standard Out을
    입력받은 파일을 Open한 디스크립터 번호를 할당한다. */
    else if (strcmp(ptr, REDIR_OUT) == 0)

```

```

    {
        ptr = strtok(NULL, SPACE_BAR);
        p[pCnt].fd[1] = open(ptr, O_WRONLY | O_CREAT | O_TRUNC, 0755);
    }

    /* 나누어진 토큰이 REDIR_EOUT(2>) 일 때, 명령어의 Standard Out을
    입력받은 파일을 Open한 디스크립터 번호를 할당한다. */
    else if (strcmp(ptr, REDIR_EOUT) == 0)
    {
        ptr = strtok(NULL, SPACE_BAR);
        p[pCnt].fd[2] = open(ptr, O_WRONLY | O_CREAT | O_TRUNC, 0755);
    }

    /* 나누어진 토큰이 그외 일때, 인자 개수를 하나 추가시키고
    새로이 2차배열에 메모리를 할당 시켜주어 입력받는다. */
    else
    {
        p[pCnt].argc++;

        if ( (p[pCnt].argv[p[pCnt].argc] = (char *)malloc(MAX_LENGTH *
sizeof(char))) == NULL)
        {
            perror("mem faild\n");
            exit(1);
        }
        strcpy(p[pCnt].argv[p[pCnt].argc], ptr); /* 나누어진 토큰을 새로 할당 받은
        인자배열 메모리에 복사 */
    }
}

/* 백그라운드 실행이 아니라면 커널의 신호를 받을 수 있도록 커널 그룹을 갖는다.
백그라운드 실행이 아닐때는 초기값인 '0'이 PGID로 셋팅되어 자기자신 PID로 하는
그룹이되는 동시에 그룹리더가 되어 다음 프로세스들의 그룹리더가 된다. */
if (!bFlag)
    p[0].pgid = getpid();

p[0].pid = RunCommand(&p[0], pfd); /* 첫번째 명령어 프로세스를 실행하고
PID를 반환받는다. */

/* 백그라운드 실행이라면 그룹리더인 처음 프로세스의 PID를
다음 프로세스들의 그룹을 PGID로 한다. */
if (bFlag)
    p[0].pgid = p[0].pid;

/* 처음 실행되는 명령어 프로세스 이후 실행되는 프로세스들이다. */
for (i = 1; i <= pCnt; i++)
{

```

```

        p[i].pgid = p[0].pgid;
        p[i].pid = RunCommand(&p[i], pfd);          /* 명령어 프로세스를 실행하고
                                                    PID를 반환한다. */
    }

    PfdClose(pfd); /* 생성된 모든 Pipe들을 파일디스크립터 테이블에서 제거 한다. */

    /* 자식프로세스를 백그라운드 실행이 아니라면 waitpid 함수를 통해서
    종료시 수신한다. 백그라운드 실행이라면 수신하지 않고 좀비프로세스로 만든다. */
    for (i = 0; i <= pCnt; i++)
    {
        MemRelease(&p[i]);          /* 프로세스 구조체에서 사용한 명령어
                                    배열변수의 메모리를 반환한다 */

        if (bFlag == TRUE)
            printf("[Process id %d]\n", p[i].pid);
        else
            if (waitpid(p[i].pid, NULL, 0) == -1)
                perror("Error:waitpid ");
    }
    return 0;
}

```

```

/**
 * [ Shell Program ]
 * 스트링을 입력받아 그 스트링사이의 Spacebar단위로 나누어 토큰으로 만든다.
 * 만들어진 각 토큰들은 판독 프로세스를 통해서 판독을 하게 되고,
 * 각 역할에 맞추어진 함수들을 실행한다.
 */
int main(void)
{
    char *str; /* 입력받는 스트링 저장 변수*/
    char *ptr; /* 나누어진 토큰의 포인터 */

    while (1)
    {
        /* Initialize */
        str = (char *)malloc(MAX_LINE * sizeof(char));

        printf("%s ", PROMPT); /* 프롬프트를 출력한다. */
        fgets(str, MAX_LINE, stdin); /* 스트링을 입력 받는다. */

        ptr = strtok(str, NEW_LINE); /* 입력받은 스트링의 '\n'을 제외한다 */

        /* "goodbye"를 입력받으면 셸을 종료하며, 단순히 '\n'만 입력 받은 경우
        위에서 제거되었기 때문에 NULL값을 갖는다. 그러한 경우를 제외하고
        스트링을 분석 실행하는 함수를 수행한다. */
        if (strcmp(str, END_CMD) == 0)
            return 0;
        else if (ptr != NULL)
        {
            CommandHandle(str); /* 스트링 분석, 수행 함수*/
        }
        free(str);
    }

    return 0;
}

```