운영체제 실습과제

#나만의 쉘 구현

강좌 명: 운영체제

교수님: 정준호 교수님

학과: 컴퓨터공학과

학년: 3학년

학번: 2017112138

이름: 정여준

리눅스 Shell에 필요한 것

1. 디렉토리를 이동하는 명령어 cd

2. 디렉토리의 리스트를 출력하는 명령어 ls

3. 파일을 복사하는 명령어 cp

4. 파일 삭제를 명령하는 명령어 rm

5. 파일을 이동시키는 명령어 mv

6. 디렉토리를 생성하는 명령어 mkdir

7. 파일을 읽기 전용으로 여는 명령어 cat

8. 쉘을 종료하는 명령어 exit

위와 같은 명령어가 있는데 이러한 명령어들을 입력했을 때 입력 라인을 분석하는 함수가 필요하고 명령어를 실행하는 exec계열 시스템 콜을 사용하는 것도 필요하며 터미널의 제어를 받아오는 것 등등이 필요하다.

여기서 입력했을 때 입력 라인을 파싱하는 것을 구현하려면 각 문자들이 왔을 때 일단 그것들을 분리해야 합니다. 하나의 단어 씩 분석해서 일반 문자일 경우와 파이프를 위한 바인 경우(|) 종료 심벌인 경우(;) 리다이렉션인 경우(>) 그 반대인 경우 (<) 리다이렉션 APPEND를 위한 경우 (>>)를 switch문을 통해서 나누어 실행하도록 구현한다.

일반 문자일 경우에는 명령어를 복사한다. 심볼이 |라면 파이프로 현재 표준출력 디스크립터를 함께 넘겨 명령어를 분석한다. 파이프가 열려 있다면 파이프를 연결한다. 그리고 명령어를 수행한다. 그리고 인자로 입력된 값들의 메모리를 모두 해제해준다. 그리고 <일 경우 >일 경우를 나눠서 구현해준다.

리다이렉트를 위한 함수도 필요하다. 입력 받았을 때 소스 파일 디스크립터가 있는 경우와 없는 경우로 나눠야 한다. 있을 경우 소스파일을 연다. 출력 파일 디스크립터가 있다면 파일이 없을 경우에는 쓰기도 가능하도록 파일을 설정하고 생성한다. 그리고 파일을 연다. 이런 식으로 리다이렉트를 구현해준다.

이런 명령어들이 들어왔을 때 파싱을 통해서 무슨 명령어인지 파악하고 맞게 동작하도록 구현해주면 된다.

모든 함수들을 만들어주고 이를 Makefile을 통해 정의하고 make를 통해 실행가능한 파일을 생성하여 나만의 쉘을 구현한다.

<common.h>

```
#ifndef __MINISH_H__
#define __MINISH_H__

#define ERROR             (-1)
#define BADFD             (-2)

#define MAXFNAME  10
#define MAXARG            10
#define MAXWORD           20
#define MAXFD             20
#define MAXVAR            50
#define MAXNAME           20

#define TRUE              1
#define FALSE             0

typedef int BOOLEAN;

// 파일의 타입을 결정하는 열거형 변수
typedef enum
{
        S_WORD,
        S_BAR,
        S_AMP,
        S_SEMI,
        S_GT,
        S_GTGT,
        S_LT,
        S_NL,
        S_EOF
} SYMBOL;

#endif
```

<command.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include "common.h"

int check_arg(char *av[], const char *opt)//  해당인자가 존재하는지 확인하는 함수
{
        int count = 0;

        while(*av != '₩0')        // 모든 인자값 확인하면 종료
        {
                if(!strcmp(av[count], opt))// opt 인자가 존재할 때
                {
                        return TRUE;
                }
                av++;
        }

        return FALSE;
}

void cmd_cd(int ac, char *av[]) //디렉토리를 변경하는 함수
{
        char *path;

        if(ac > 1)        // 인자가 있을 경우 path를 설정
        {
                path = av[1];
        }

        else if((path = (char*)getenv("HOME")) == NULL)      // 인자가 없으면 home으로 이동
        {
                // 환경 변수가 없을 경우 현재 디렉토리로 설정
                path = ".";
        }

        // 디렉토리를 변경한다.
        if(chdir(path) == ERROR)
        {
                fprintf(stderr, "%s: bad directory.₩n", path);
        }
}

void cmd_exit()//  프로그램을 종료
{
        exit(1);
}

void cmd_ls(int ac, char *av[]) //디렉토리 리스트를 출력하는 함수
{
        DIR *dp;
        struct dirent *entry;
        char *path;
        int count;
        int opt_a;
```

```
int opt_l;

if(ac < 2)        // 인자가 없을 경우 자기자신의 디렉토리로 설정
{
        path = ".";
}
else // 인자가 있을 경우
{
        path = av[1];
}

if((dp = opendir(path)) == NULL)// 디렉토리를 연다.
{
        fprintf(stderr, "Can't open directory: %s", av[1]);
        return;
}

// 다음의 인자들이 존재하는지 확인
opt_a = check_arg(av, "-a");
opt_l = check_arg(av, "-l");

count = 0;

while((entry = readdir(dp)) != NULL)// 파일이나 디렉토리를 읽어들인다.
{
        // -a 옵션이 없을 경우
        if(!opt_a)
        {
                if(entry->d_name[0] == '.')
                {
                        continue; // 숨김 파일은 표시하지 않는다.
                }
        }

        // 출력
        printf("%s₩t", entry->d_name);

        if(opt_l)         // -l 옵션이 설정되어있을 경우
        {
                printf("₩n");//줄마다 한원소씩을 출력한다.
        }

        // 한줄에 3개씩 출력한다.
        else
        {
                if(count > 3)
                {
                        printf("₩n");
                        count = 0;
                }
                else
                {
                        count++;
                }
        }
}

closedir(dp);     // 디렉토리를 닫는다.
```

```
        printf("₩n");
}

void cmd_cp(int ac, char *av[]) // 파일을 복사하는 함수
{
        FILE *src;
        FILE *dst;
        char ch;


        if(ac < 3) // 인자가 2개 이하일 경우
        {
                fprintf(stderr, "Not enough arguments.₩n");
                return;
        }

        if((src = fopen(av[1], "r")) == NULL) // 복사할 소스 파일을 연다.
        {
                fprintf(stderr, "%s: Can't open file.₩n", av[1]);
                return;
        }

        if((dst = fopen(av[2], "w")) == NULL) // 쓰기를 할 파일을 연다.
        {
                fprintf(stderr, "%s: Can't open file.₩n", av[2]);
                return;
        }

        while(!feof(src)) // 복사
        {
                ch = (char) fgetc(src);

                if(ch != EOF)
                {
                        fputc((int)ch, dst);
                }
        }

        if(check_arg(av, "-v"))// -v 옵션이 있을 경우
        {
                printf("cp %s %s₩n", av[1], av[2]); //내용 출력
        }

        fclose(src);
        fclose(dst);
}

void cmd_rm(int ac, char *av[]) // 파일 삭제 명령 함수
{

        if(ac < 2)// 인자가 없을 경우
        {
                fprintf(stderr, "Not enough arguments.₩n");
                return;
        }


        unlink(av[1]); // 파일 삭제
```

```
        if(check_arg(av, "-v"))// -v 옵션이 있을 경우
        {
                printf("rm %s₩n", av[1]); //결과 출력
        }
}

void cmd_mv(int ac, char *av[]) // 파일을 이동시키는 함수
{
        FILE *src;
        FILE *dst;
        char ch;

        if(ac < 3) // 인자가 2개 이하일 경우
        {
                fprintf(stderr, "Not enough arguments.₩n");
                return;
        }

        if((src = fopen(av[1], "r")) == NULL) //복사할 파일을 연다.
        {
                fprintf(stderr, "%s: Can't open file.₩n", av[1]);
                return;
        }

        if((dst = fopen(av[2], "w")) == NULL) // 쓰기할 파일을 연다.
        {
                fprintf(stderr, "%s: Can't open file.₩n", av[2]);
                return;
        }

        while(!feof(src))// 복사
        {
                ch = (char) fgetc(src);

                if(ch != EOF)
                {
                        fputc((int)ch, dst);
                }
        }

        fclose(src);
        fclose(dst);

        unlink(av[1]); // 기존 파일 삭제

        if(check_arg(av, "-v")) // -v 옵션
        {
                printf("mv %s %s₩n", av[1], av[2]); //결과를 출력한다.
        }
}

void cmd_mkdir(int ac, char *av[]) // 디렉토리를 생성하는 함수
{
        if(ac < 2) // 인자가 없을 경우 에러
        {
                fprintf(stderr, "Not enough arguments.₩n");
```

```
                return;
        }

        if(mkdir(av[1], 0755)) // 디렉토리를 생성
        {
                fprintf(stderr, "Make directory failed.₩n");
        }
}

void cmd_rmdir(int ac, char *av[]) // 디렉토리를 삭제하는 함수
{

        if(ac < 2) // 인자가 없을 경우 에러
        {
                fprintf(stderr, "Not enough arguments.₩n");
                return;
        }

        if(rmdir(av[1])) // 디렉토리를 삭제한다.
        {
                fprintf(stderr, "Remove directory failed.₩n");
        }
}

void cmd_cat(int ac, char *av[]) // cat 명령어 함수
{
        int ch;
        FILE *fp;

        if(ac < 2) // 인자가 없을 경우 에러처리
        {
                fprintf(stderr, "Not enough arguments");
                return;
        }

        if((fp = fopen(av[1], "r")) == NULL) // 읽기전용으로 파일을 연다.
        {
                fprintf(stderr, "No such file on directory.₩n");
                return;
        }

        while((ch = getc(fp)) != EOF) // 내용 출력
        {
                putchar(ch);
        }

        fclose(fp);
}
```

<symbol.c>

```
#include <stdio.h>
#include "common.h"

typedef enum {NEUTRAL, GTGT, INQUOTE, INWORD} STATUS;

//  문자열을 분석하며 심볼을 찾는 함수
SYMBOL getsymbol(char *word)
{
        STATUS state;
        int c;
        char *w;

        state = NEUTRAL;
        w = word;

        while ((c = getchar()) != EOF)
        {
                switch (state)
                {
                        case NEUTRAL :
                                switch (c)
                                {
                                        case ';' :
                                                return S_SEMI;

                                        case '&' :
                                                return S_AMP;

                                        case '|' :
                                                return S_BAR;

                                        case '<' :
                                                return S_LT;

                                        case '₩n':
                                                return S_NL;

                                        case ' ' :
                                        case '₩t':
                                                continue;

                                        case '>' :
                                                state = GTGT;
                                                continue;

                                        case '"' :
                                                state = INQUOTE;
                                                continue;

                                        default  :
                                                state = INWORD;
                                                *w++ = c;
                                                continue;
                                }

                        case GTGT:
```

```
                        if (c == '>')
                        {
                                return S_GTGT;
                        }
                        ungetc(c, stdin);
                        return S_GT;

                case INQUOTE:
                        switch (c)
                        {
                                case '₩₩' :
                                        *w++ = getchar();
                                        continue;

                                case '"'  :
                                        *w = '₩0';
                                        return S_WORD;

                                default   :
                                        *w++ = c;
                                        continue;
                        }

                case INWORD:
                        switch (c )
                        {
                                case ';' :
                                case '&' :
                                case '|' :
                                case '<' :
                                case '>' :
                                case '₩n':
                                case ' ' :
                                case '₩t':
                                        ungetc(c, stdin);
                                        *w = '₩0';
                                        return S_WORD;

                                default  :
                                        *w++ = c;
                                        continue;
                        }
                }
        }

   return S_EOF;
}
```

<parser.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "common.h"

//  명령어를 파싱한다.
SYMBOL parse(int *waitpid, BOOLEAN makepipe, int *pipefdp)
{
        SYMBOL symbol, term;
        int argc, sourcefd, destfd;
        int pid, pipefd[2];
        char *argv[MAXARG+1], sourcefile[MAXFNAME];
        char destfile[MAXFNAME];
        char word[MAXWORD];
        BOOLEAN append;

        argc = 0;
        sourcefd = 0;
        destfd = 1;


        while (TRUE)
        {
                // 하나의 단어씩을 분석한다.
                switch (symbol = getsymbol(word))
                {
                        // 일반 문자일 경우
                        case S_WORD :
                                if(argc == MAXARG)
                                {
                                        fprintf(stderr, "Too many args.₩n");
                                        break;
                                }

                                // 새로운 인자 배열을 메모리 할당 한다.
                                argv[argc] = (char *) malloc(strlen(word)+1);

                                if(argv[argc] == NULL)
                                {
                                        fprintf(stderr, "Out of arg memory.₩n");
                                        break;
                                }

                                // 명령어 복사
                                strcpy(argv[argc], word);

                                // arg 카운터 증가
                                argc++;
                                continue;

                        // < 일 경우
                        case S_LT   :

                                // 파이프가 열려있다면 오류
                                if(makepipe)
                                {
```

```
                        fprintf(stderr, "Extra <.₩n");
                        break;
                }

                // 소스파일의 심볼값을 검사한다.
                if(getsymbol(sourcefile) != S_WORD)
                {
                        fprintf(stderr, "Illegal <.₩n");
                        break;
                }

                sourcefd = BADFD;
                continue;

        // > 혹은 >> 일 경우
        case S_GT   :
        case S_GTGT :

                // 목적 파일이 정의되어있지 않을 경우 에러
                if(destfd != 1)
                {
                        fprintf(stderr, "Extra > or >>.₩n");
                        break;
                }

                // 목적어 파일의 심볼타임이 문자열이 아니면 에러
                if(getsymbol(destfile) != S_WORD)
                {
                        fprintf(stderr, "Illegal > or >>.₩n");
                        break;
                }

                // GTGT일 경우 추가 모드로
                destfd = BADFD;
                append = (symbol == S_GTGT);
                continue;

        // |, &, ;, 줄바꿈 문자일 경우 - 하나의 명령어 단위
        case S_BAR  :
        case S_AMP  :
        case S_SEMI :
        case S_NL   :

                argv[argc] = NULL;
                // 심볼이 파이프(|) 일 경우
                if(symbol == S_BAR)
                {
                        if(destfd != 1)
                        {
                                fprintf(stderr, "> or >> conflicts with
|.₩n");
                                break;
                        }

                        // 현재의 표준출력 디스크립터를 함께 넘겨 명령어를
분석한다.
                        term = parse(waitpid, TRUE, &destfd);
                }
```

```
                                        // 종료 문자 세팅
                                        else
                                        {
                                                term = symbol;
                                        }

                                        // 파이프가 열려있을 경우 파이프를 연결한다.
                                        if (makepipe)
                                        {
                                                if (pipe(pipefd) == ERROR)
                                                {
                                                        syserr("pipe");
                                                }
                                                *pipefdp = pipefd[1];
                                                sourcefd = pipefd[0];
                                        }

                                        // 명령을 수행한다.
                                        pid = execute(argc, argv, sourcefd, sourcefile,
                                                                                destfd, destfile,append, term
== S_AMP);

                                        // 파이프가 아닐 경우 기다릴 PID를 설정
                                        if (symbol != S_BAR)
                                        {
                                                *waitpid = pid;
                                        }

                                        // 인자값을 없을 경우
                                        if (argc == 0 && (symbol != S_NL || sourcefd > 1))
                                        {
                                                fprintf(stderr, "Missing command.₩n");
                                        }

                                        // 인자로 입력된 값들의 메모리 해제
                                        while (--argc >= 0)
                                        {
                                                free(argv[argc]);
                                        }

                                        return term;

                                // 명령이 잘못되었을 경우 종료
                                case S_EOF :
                                        exit(0);
                        }
                }
}
```

<redirect.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include "common.h"

//  표준 입출력을 이용한 리다이렉트 함수
void redirect(int sourcefd, char *sourcefile, int destfd, char *destfile, BOOLEAN append, BOOLEAN
backgrnd)
{
        int flags, fd;

        // 열려있는 소스 파일디스크립터가 없고 백그라운드 실행일 경우
        if (sourcefd == 0 && backgrnd)
        {
                strcpy(sourcefile, "/dev/null");
                sourcefd = BADFD;
        }

        //소스 파일 디스크립터가 있을 경우
        if (sourcefd != 0)
        {
                //표준입력을 닫는다.
                if (close(0) == ERROR)
                {
                        syserr("close");
                }

                if (sourcefd > 0)
                {
                        //표준입력을 재정의한다.
                        if (dup(sourcefd) != 0)
                        {
                                fatal("dup");
                        }
                }

                //소스파일을 연다.
                else if (open(sourcefile, O_RDONLY, 0) == ERROR)
                {
                        fprintf(stderr, "Cannot open %s₩n", sourcefile);
                        exit(0);
                }
        }

        //출력 파일 디스크립터가 있다면
        if (destfd != 1)
        {
                //표준 출력을 닫는다.
                if (close(1) == ERROR)
                {
                        syserr("close");
                }

                //표준 출력을 재정의
```

```
                if (destfd > 1)
                {
                        if (dup(destfd) != 1)
                        {
                                fatal("dup");
                        }
                }

                else
                {
                        //파일이 없을 경우에 생성하고, 쓰기도 가능하게 파일을 설정
                        flags = O_WRONLY | O_CREAT;

                        //추가 모드가 아닐 경우 파일을 비우는 옵션도 추가
                        if (!append)
                        {
                                flags |= O_TRUNC;
                        }

                        //파일을 연다.
                        if (open(destfile, flags, 0666) == ERROR)
                        {
                                fprintf(stderr, "Cannot create %s₩n", destfile);
                                exit(0);
                        }

                        //추가 모드라면 가장 끝을 찾는다.
                        if (append)
                        {
                                if (lseek(1, 0L, 2) == ERROR) syserr("lseek");
                        }
                }
        }

        // 파일디스크립터를 닫는다.
        for (fd =3; fd < MAXFD; fd++)
        {
                close(fd);
        }

        return;
}
```

<util.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "common.h"

//  치명적인 오류 처리
void fatal(char *message)
{
        fprintf(stderr, "Error: %s₩n", message);
        exit(1);
}

//  일반 에러 처리
void syserr(char *message)
{
        fprintf(stderr, "Error: %s (%d", message, errno);
        exit(1);
}

//  부모가 자식의 실행이 끝나길 기다리는 함수
void waitfor(int pid)
{
        int wpid, status;

        // wait를 이용하여 프로세스를 기다린다.
        while ((wpid = wait(&status)) != pid && wpid != ERROR);
}

//  프롬프트를 출력한다.
void print_prompt()
{
        char *ps;
        char *index;

        // PS2 환경 변수를 읽어 사용한다.
        if((ps = (char*)getenv("PS2")) != NULL)
        {
                // 환경 변수의 값이 끝날때까지 한바이트씩 이동하며 확인
                while(*ps != '₩0')
                {
                        // 역슬래쉬(₩)가 있다면 다음의 문자를 확인한다.
                        if(*ps == '₩₩')
                        {
                                ps++;

                                // 사용자 정보를 출력
                                if(*ps == 'u')
                                {
                                        printf("%s", getenv("USER"));
                                }

                                // 호스트명을 출력
                                else if(*ps == 'h')
                                {
                                        printf("%s", getenv("HOSTNAME"));
```

```
                                }

                                // 현재 디렉토리를 출력한다.
                                else if(*ps == 'w')
                                {
                                        printf("%s", get_current_dir_name());
                                }
                        }

                        // 아무것도 아닐 경우 문자를 출력
                        else
                        {
                                printf("%c", *ps);
                                ps++;
                        }
                }
        }

        // PS2 환경 변수가 없다면 기본 프롬프트를 출력한다.
        else
        {
                printf(">> ");
        }
}

//  사용자 정의 명령을 처리하는 함수
BOOLEAN shellcmd(int ac, char *av[], int sourcefd, int destfd)
{
        char *path;

        // cd
        if(!strcmp(av[0], "cd"))
        {
                cmd_cd(ac, av);
        }
        // ls
        else if(!strcmp(av[0], "ls"))
        {
                cmd_ls(ac, av);
        }
        // cp
        else if(!strcmp(av[0], "cp"))
        {
                cmd_cp(ac, av);
        }
        // rm
        else if(!strcmp(av[0], "rm"))
        {
                cmd_rm(ac, av);
        }
        // mv
        else if(!strcmp(av[0], "mv"))
        {
                cmd_mv(ac, av);
        }
        // mkdir
        else if(!strcmp(av[0], "mkdir"))
        {
```

```
                cmd_mkdir(ac, av);
        }
        // rmdir
        else if(!strcmp(av[0], "rmdir"))
        {
                cmd_rmdir(ac, av);
        }
        // cat
        else if(!strcmp(av[0], "cat"))
        {
                cmd_cat(ac, av);
        }
        // exit
        else if(!strcmp(av[0], "exit"))
        {
                cmd_exit();
        }
        else
        {
                return FALSE;
        }

        if (sourcefd != 0 || destfd != 1)
        {
                fprintf(stderr, "Ilegal redirection or pipeline.₩n");
        }

        return TRUE;
}
```

<execute.c>

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "common.h"

//  명령어를 실행하는 함수
int execute(int ac, char *av[], int sourcefd, char *sourcefile, int destfd, char *destfile,
        BOOLEAN append, BOOLEAN backgrnd)
{
        int pid;

        // 인자가 없거나 사용자 정의 명령을 수행하여 실패하였을 경우
        if (ac == 0 || shellcmd(ac, av, sourcefd, destfd))
        {
                return 0;
        }

        pid = fork();// 프로세스를 fork한다.

        switch (pid)
        {
                // 에러가 났을 경우
                case ERROR :
                        fprintf(stderr, "Cannot create new process.₩n");
                        return 0;

                // 자식일때, 프로그램을 실행하는 실체
                case 0 :
                        redirect(sourcefd, sourcefile, destfd, destfile, append, backgrnd);
                        execvp(av[0], av);
                        fprintf(stderr, "Cannot execute %s₩n", av[0]);
                        exit(0);

                // 부모일 때
                default :
                        // 읽기 파일디스크립션을 닫는다.
                        if(sourcefd > 0 && close(sourcefd) == ERROR)
                        {
                                syserr("close sourcefd");
                        }

                        // 쓰기 파일디스크립션을 닫는다.
                        if(destfd > 1 && close(destfd) == ERROR)
                        {
                                syserr("close destfd");
                        }

                        // 백그라운드 명령일 경우 pic 출력
                        if(backgrnd)
                        {
                                printf("%d₩n", pid);
                        }

                        return pid;
        }
}
```

<minish.c>

```c
#include <stdio.h>
#include "common.h"

//  메인 함수
int main()
{
	int pid, fd;
	SYMBOL term;

	// 프롬프트를 출력
	print_prompt();

	// 무한 반복하며 사용자 입력을 받는다.
	while (TRUE)
	{
		// 사용자 명령을 입력받아 분석한다.
		term = parse(&pid, FALSE, NULL);

		// 명령의 마지막 심볼이 &일 경우 백그라운드로 명령을 수행한다.
		// 즉 wait를 하지 않는다.
		if (term != S_AMP && pid != 0)
		{
			// 백그라운드 명령이 아닌 경우 자식이 프로세스를 종료할때까지 기다린다.
			waitfor(pid);
		}

		// 마지막 문자가 줄바꿈 문자일 경우 새로운 프롬프트를 출력한다.
		if (term == S_NL)
		{
			print_prompt();
		}

		// 파일 디스크립터를 닫는다.
		for (fd=3; fd<MAXFD; fd++)
		{
			close(fd);
		}
	}
}
```

이렇게 만든 파일들을 리눅스에서 Makefile을 만들고 make를 통해 실행가능한 파일 myshell를 생성한다.

Makefile은 다음과 같이 작성한다.

<Makefile>

```
CC              =gcc

CFLAGS          = -g -O2

TARGET          = myshell

OBJECTS         = execute.o parser.o redirect.o symbol.o util.o command.o

MAIN_SOURSE     = minish.c


all: $(TARGET)


$(TARGET): $(OBJECTS)

        $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(MAIN_SOURCE)

.c.o:

        $(CC) $(CFLAGS) -c $<


clean:

        rm -f $(OBJECTS)

        rm -f $(TARGET)
```

이제 make명령어를 통해서 myshell을 생성한다.

```
pucoy33@yeo:~/myshell$ make
gcc -g -O2  -o myshell execute.o parser.o redirect.o symbol.o util.o command.o minish.c
minish.c: In function 'main':
minish.c:13:2: warning: implicit declaration of function 'print_prompt' [-Wimplicit-function-declaration]
  print_prompt();
  ^
minish.c:19:10: warning: implicit declaration of function 'parse' [-Wimplicit-function-declaration]
   term = parse(&pid, FALSE, NULL);
          ^
minish.c:26:4: warning: implicit declaration of function 'waitfor' [-Wimplicit-function-declaration]
    waitfor(pid);
    ^
minish.c:38:4: warning: implicit declaration of function 'close' [-Wimplicit-function-declaration]
    close(fd);
    ^
```

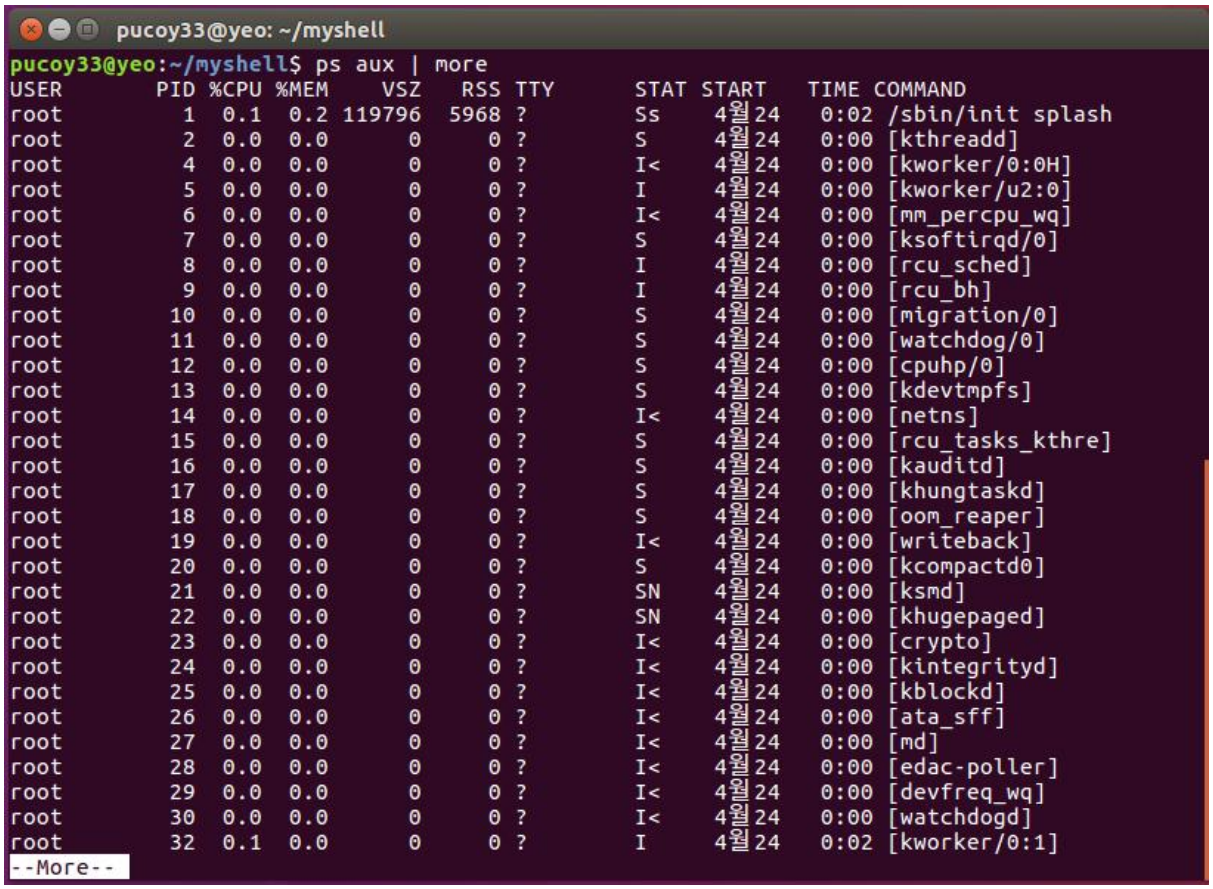
```
pucoy33@yeo:~/myshell$ ls
Makefile   command.o  execute.c  minish.c  parser.c  redirect.c  symbol.c  util.c
command.c  common.h   execute.o  myshell   parser.o  redirect.o  symbol.o  util.o
```

실행가능한 파일 myshell이 생성되었다. 이제 myshell을 실행해보자.

```
pucoy33@yeo: ~/myshell
pucoy33@yeo:~/myshell$ ./myshell
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        symbol.c        redirect.o
execute.o       Makefile        parser.o        redirect.c      myshell
command.c
>> cp execute.c cp1.c
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        symbol.c        redirect.o
execute.o       cp1.c   Makefile        parser.o        redirect.c
myshell command.c
>> rm cp1.c
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        symbol.c        redirect.o
execute.o       Makefile        parser.o        redirect.c      myshell
command.c
>> cp symbol.c cp2.c
>> mv cp2.c mv.c
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        mv.c    symbol.c
redirect.o      execute.o       Makefile        parser.o        redirect.c
myshell command.c
>> rm mv.c
>> mkdir yeo
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        symbol.c        redirect.o
execute.o       Makefile        parser.o        redirect.c      myshell
yeo     command.c
```

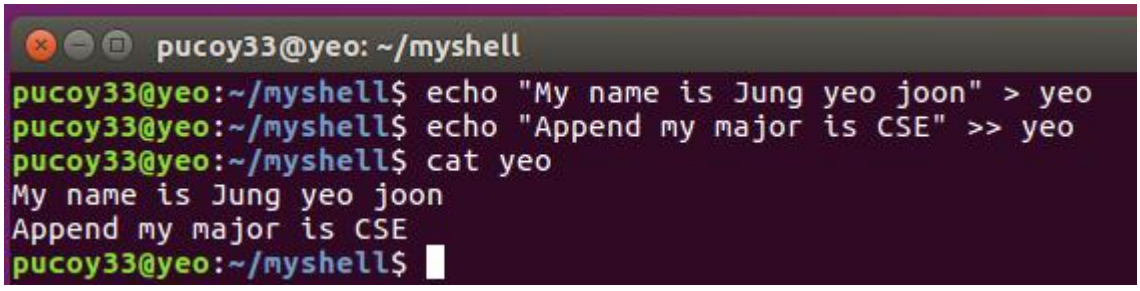
```
>> cd yeo
>> ls

>> cd ..
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        symbol.c        redirect.o
execute.o       Makefile        parser.o        redirect.c      myshell
yeo     command.c
>> rmdir yeo
>> ls
execute.c       symbol.o        common.h        util.c  util.o
command.o       minish.c        parser.c        symbol.c        redirect.o
execute.o       Makefile        parser.o        redirect.c      myshell
command.c
>> exit
pucoy33@yeo:~/myshell$
```

실행하고 ls명령어를 통해서 현재 해당하는 위치에 있는 파일들의 이름을 나열했다. 그리고 cp명령어를 통해서 execute.c 파일을 복사한 cp1.c 파일을 생성했다. 또한 rm 명령어를 통해서 생성한 cp1.c파일을 삭제했다. 다시 cp명령어를 통해서 symbol.c파일을 복사해서 cp2.c파일을 생성했다. mv 명령어를 통해 cp2.c파일의 이름을 mv.c파일로 바꿔줬다. 다시 rm명령어를 사용해서 mv.c파일을 지웠다. 이제 mkdir 명령어를 통해서 새로운 디렉토리를 생성한다. cd 명령어를 통해서 이동했다가 다시 cd명령어를 통해서 원래 있던 디렉토리로 복귀한다. 이후 rmdir 명령어를 통해 생성했던 디렉토리를 지운다. 그 이후 exit 명령어를 통해 나만의 쉘을 종료한다.



ps aux | more을 명령어를 통해서 파이프를 사용한다. 프로세스의 상태를 나타내는데 | more을 통해서 일정 부분까지 나오고 enter를 입력 시 한 줄 씩 정보가 더 나오는 것을 확인할 수 있다.

또한 q를 누르면 종료된다.



위는 리다이렉트의 생성과 추가를 해보는 작업이다.

<소감>

이번에 나만의 쉘 프로젝트를 진행하면서 인터넷으로 많은 것을 검색하고 많은 것을 배울 수 있었습니다. 일단 쉘을 만들기 위해서 어떤 것이 필요한지에 대해서 생각할 수 있었고 찾아보면서 배울 수 있었습니다. 그리고 필요한 것들을 구현하기 위해서 코드를 짜면서 내가 생각했던 부분을 어떻게 구현하는지에 대해서 배울 수 있었습니다. 과제의 난이도는 조금 높았던 것 같지만 많은 자료들이 인터넷에 있었기 때문에 인터넷에서 자료를 찾아 스스로 공부하면서 과제를 할 수 있었기 때문에 좋았던 것 같습니다. 지금까지의 과제들보다 더 제 실력에 도움을 많이 준 것 같아서 뿌듯했습니다.