REPORT

보고서 작성 서약서

1. 나는 타학생의 보고서를 복사(Copy)하지 않았습니다.
2. 나는 타학생의 보고서를 인터넷에서 다운로드 하여 대체하지 않았습니다.
3. 나는 타인에게 보고서 제출 전에 보고서를 보여주지 않았습니다.
4. 보고서 제출 기한을 준수하였습니다.

나는 보고서 작성시 위법 행위를 하지 않고,
성.균.인으로서 나의 명예를 지킬 것을 약속합니다.

과　　목 : 시스템 SW 실습 2
담당교수: 정진규
학　　과 : 반도체시스템공학과
학　　번 : 2018314788
이　　름: 오해성
제 출 일 : 2020.11.20

서 론

본 리포트는 배쉬쉘의 일부 기능을 구현한 미니쉘 프로그램에 대해 다룹니다.

본론에서는 파일 설계도와 FLOW CHART 를 시작으로
구현 파일의 각 함수에 대해 ADT 를 설명하고
코드의 중요한 부분에 대해서만 간략히 설명합니다.

순서는 다음과 같습니다.

1. TABLE
2. CVECTOR
3. ERRHANDELR
4. HEAD
5. TAIL
6. CAT
7. CP
8. MV
9. RM
10. PWD
11. MINISHELL

본 론

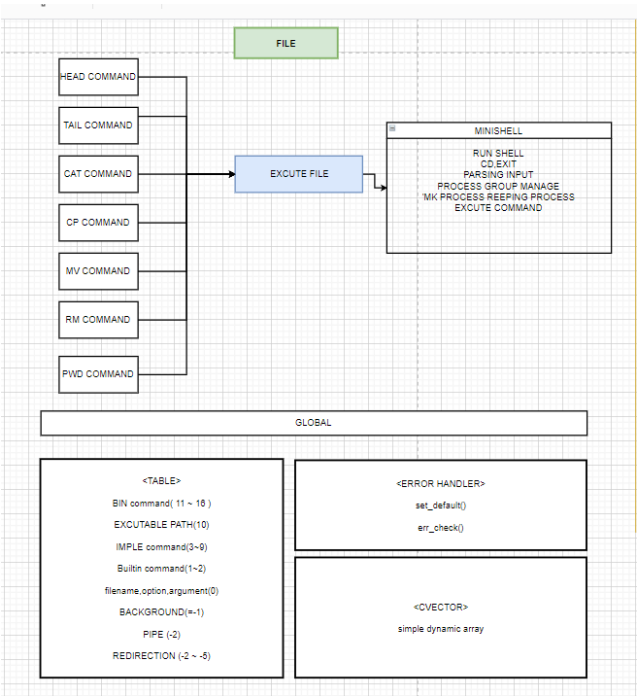
파일 설계도

우측 도면과 같이 구현한 소스파일은 총 11개 입니다. (헤더 파일 제외)

실제로 미니쉘은 구동하기 위한 미니쉘 파일과

binary 파일을 만들기 위한 커맨드 관련 파일 7개

위 파일의 동작을 돕기 위해 글로벌하게 쓰이는 파일 3개 입니다.



FLOW CHART

프로그램의 실행은 다음과 같이 간단합니다.

exit 커맨드가 들어오지 않으면 반복해서

명령어를 입력받고 유효한지 판단 후

유효하다면 foreground 인지 background 인지 판단하여

미니쉘의 생성된 자식(새로운 프로세스 그룹 리더)이 커맨드를 실행합니다.

이 리더 프로세스가 파이프라인이나 리다이렉션에 따라 자식을 생성하여

실행 커맨드를 관리합니다.

이때 foreground 라면 미니쉘의 자식 그룹은 세션의 foreground 그룹이 되고

미니쉘은 자식의 종료를 기다립니다.

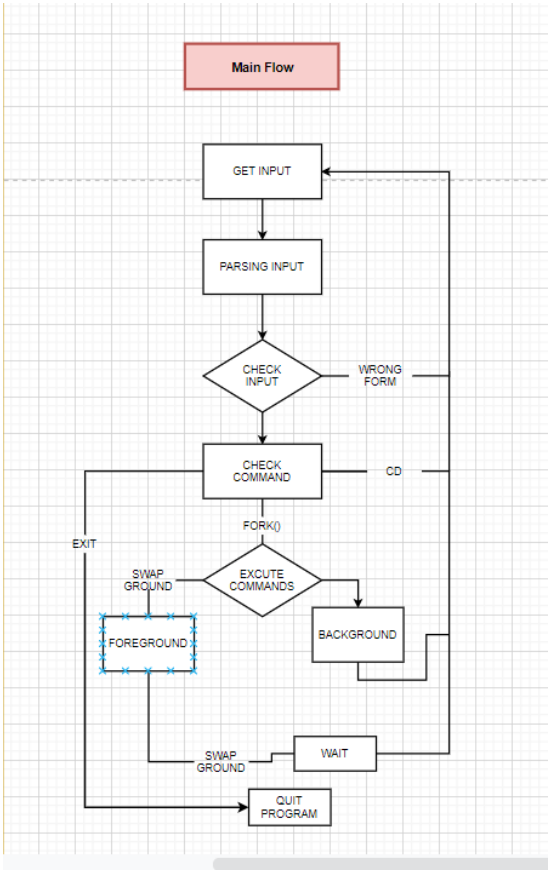
background 라면 미니쉘은 자식의 종료를 기다리지 않고

다음 커맨드를 계속해서 받습니다.

자식의 실행이 종료되거나 시그널을 받아 stop 상태가 된 경우

미니쉘 프로세스에서 이를 reaping 합니다.

reaping 시 그룹 단위로 signal 을 보내 좀비 프로세스가 생성되지 않도록 합니다.



TABLE

실행 인자를 빠르게 파악하기 위해 문자열을 숫자에 매칭 시켰습니다.

비교연산자를 이용하여 빠르게 문자열의 타입을 파악할 수 있고

숫자를 조합하여 어떤 종류의 리다이렉션인지 빠르게 확인할 수 있습니다.

C에서 기본적으로 제공되는 map 구조의 stl이 없어 strcmp 로 간단히만

구현했습니다.

빈번히 사용되는 table 이므로 트라이나 해싱을 이용하여

더 최적화된 map 으로 개선이 필요합니다.

```c
#include <table.h>

int cmap(char * str)
{
    /* keyword -1 ~ -5 */

    if(!strcmp(str,"&")) return -1;
    else if(!strcmp(str,"|")) return -2;
    else if(!strcmp(str,"<")) return -3;
    else if(!strcmp(str,">")) return -4;
    else if(!strcmp(str,">>")) return -5;


    /* bulitin command 1~2 */

    else if(!strcmp(str,"exit")) return 1;
    else if(!strcmp(str,"cd")) return 2;

    /* imple command 3 ~ 9*/

    else if(!strcmp(str,"head")) return 3;
    else if(!strcmp(str,"tail")) return 4;
    else if(!strcmp(str,"cat")) return 5;
    else if(!strcmp(str,"cp")) return 6;
    else if(!strcmp(str,"mv")) return 7;
    else if(!strcmp(str,"rm")) return 8;
    else if(!strcmp(str,"pwd")) return 9;

    else if(strlen(str) >= 2 && str[0] == '.' && str[1] == '/') return 10; // path executable

    /* bin command 11 ~ 16*/

    else if(!strcmp(str,"ls")) return 11;
    else if(!strcmp(str,"grep")) return 12;
    else if(!strcmp(str,"man")) return 13;
    else if(!strcmp(str,"sort")) return 14;
    else if(!strcmp(str,"awk")) return 15;
    else if(!strcmp(str,"bc")) return 16;

    else return 0; // filename || option || argument || invalid command
}
```

CVECTOR

객체 지향 언어가 아닌 C의 특성 상

정형화된 동적배열이 존재하지 않아 매번 동적 할당 후 해제해야 되는 불편함이 있어

자주 쓰이는 int 형으로만 동적배열 자료구조를 만들었습니다,

동작은 C++ vector 자료구조와 유사하며

push 연산 시 bucket의 끝에 다다르면 bucket 을 2배로 늘립니다.

메모리의 해제는 clear 함수를 이용하여 해제합니다.

직접적인 구현은 cvector.c 을 보면 알 수 있으나

크게 복잡한 부분이 없어 이 문서에서는 생략합니다.

```h
#ifndef _CVECTOR_H
#define _CVECTOR_H 1

#include <stdlib.h>
#include <stdio.h>

typedef struct _cvector
{
    int csz;
    int tsz;
    int * arr;
}cvector;

void cv_init(cvector * _cv,int _sz);
void cv_push(cvector * _cv,int _data);
int cv_pop(cvector * _cv);
int cv_size(cvector *_cv);
int cv_back(cvector * _cv);
void cv_clear(cvector *_cv);
void cv_print(cvector *_cv);

#endif /* cvector.h */
```

ERRORHANDLER

에러를 매번 소스코드에서 핸들하면

코드가 지저분해지기도 하고 비효율적이라 생각해

따로 에러를 처리해주는 파일을 만들었습니다.

이 파일을 이용하는 소스코드에서는 set_default 함수로 errno 를 기본값으로 설정 후

동작이 끝난 후 err_check 함수로 error 가 있다면 그에 해당하는 error 를 형식에 맞게

출력하도록 하였습니다.

```c
#include "definition.h"
#include "errhandler.h"

void set_default(void)
{
    errno = 0;
    return;
}

void err_check(char * str)
{
    if(!errno) return;

    switch (errno)
    {
    case EACCES:
        fprintf(stderr,"%s: Permission denied\n",str);
        break;

    case EISDIR:
        fprintf(stderr,"%s: Is a directory\n",str);
        break;

    case ENOENT:
        fprintf(stderr,"%s: No such file or directory\n",str);
        break;

    case ENOTDIR:
        fprintf(stderr,"%s: Not a directory\n",str);
        break;

    case EPERM:
        fprintf(stderr,"%s: Operation not permitted\n",str);
        break;

    default:
        printf("%s: Error occurred: <%d>\n",str,errno);
        break;
    }

    return;
}
```

HEAD

head 커맨드의 옵션이 사용된 경우와 사용되지 않은 경우

파일명이 인자로 들어온 경우와 그렇지 않은 경우를

라인수를 나타내는 변수 otk 와 파일 포인터인 inf 함수로 구분하여

29번 라인에서 otk 만큼의 라인을 읽어들이고 표준 출력으로 출력합니다.

```c
#include "definition.h"
#include "errhandler.h"

int main(int argc,char * argv[])
{
    set_default();

    int otk = 0;
    FILE * inf = NULL;

    if(argc > 2 && !strcmp(argv[1],"-n")) otk = atoi(argv[2]);

    if((otk && argc == 4) || (!otk && argc==2))
    {
        inf = fopen(argv[3-(!!otk)*2],"r");
        if(!inf)
        {
            err_check("head");
            return 0;
        }
    }

    if(!otk) otk = 10;

    char * line = NULL;
    size_t line_sz = 0;
    ssize_t read;

    while((read = getline(&line,&line_sz,(inf?inf:stdin)))!=-1 && otk--) printf("%s",line);

    if(inf) fclose(inf);
    err_check("head");


    return 0;
}
```

TAIL

파일의 앞 부분은 head 파일과 동일합니다.

head는 앞에서 읽어들이는 대로 바로 출력하면 되는 반면

tail은 그렇게 간단히 구현되지 않았습니다,

가장 단순하게 tail 을 구현하는 방법은

파일을 라인대로 읽어 전부 메모리에 올린 후

(전체 라인 수 - k) 라인부터 전체 라인 수 까지 출력하는 방식입니다,

이 방식은 메모리 상 효율적이지 않으므로

출력할 파일의 라인 크기 K 에 해당하는 배열에

가장 끝 K 라인의 offset만 저장하는 식으로 최적화 하였습니다.

배열은 원형 큐 형식으로 가장 뒤의 k 라인의 offset만 항상 저장되고

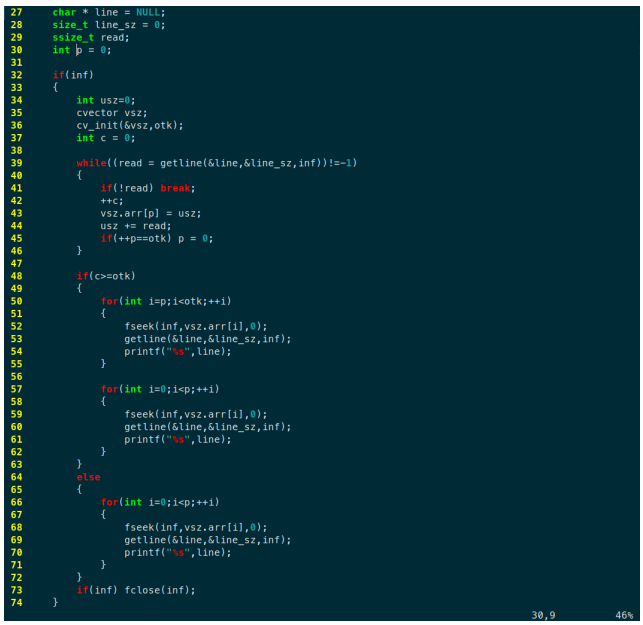
마지막 포인팅하는 위치부터 배열을 전부 한번씩 출력하면 됩니다.

이때 fseek 함수를 이용하여 해당하는 offset으로 이동한 후 다시 라인을 읽어

출력합니다.

STDIN 의 경우 offset 을 사용할 수 없기 때문에

char ** 배열을 K만큼 동적할당하여 위와 유사한 방식으로 처리하였습니다.



```c
    char * line = NULL;
    size_t line_sz = 0;
    ssize_t read;
    int p = 0;

    if(inf)
    {
        int usz=0;
        cvector vsz;
        cv_init(&vsz,otk);
        int c = 0;

        while((read = getline(&line,&line_sz,inf))!=-1)
        {
            if(!read) break;
            ++c;
            vsz.arr[p] = usz;
            usz += read;
            if(++p==otk) p = 0;
        }

        if(c>=otk)
        {
            for(int i=p;i<otk;++i)
            {
                fseek(inf,vsz.arr[i],0);
                getline(&line,&line_sz,inf);
                printf("%s",line);
            }

            for(int i=0;i<p;++i)
            {
                fseek(inf,vsz.arr[i],0);
                getline(&line,&line_sz,inf);
                printf("%s",line);
            }
        }
        else
        {
            for(int i=0;i<p;++i)
            {
                fseek(inf,vsz.arr[i],0);
                getline(&line,&line_sz,inf);
                printf("%s",line);
            }
        }
        if(inf) fclose(inf);
    }
                                                                30,9          46%
```

CAT

CAT 의 경우 head 와 구현이 크게 다르지 않습니다.
파일인지 STDIN 인지 판단 후
22 라인에서 해당하는 입력을 받으면서 입력의 끝까지
계속해서 출력합니다.

```c
#include "definition.h"
#include "errhandler.h"

int main(int argc,char * argv[])
{
    set_default();

    FILE * inf = NULL;

    if(argc == 2) inf = fopen(argv[1],"r");

    if(argc==2 && !inf)
    {
        err_check("cat");
        return 0;
    }

    char * line = NULL;
    size_t line_sz = 0;
    ssize_t read;

    while((read = getline(&line,&line_sz,(inf?inf:stdin)))!=-1) printf("%s",line);
    if(inf) fclose(inf);

    err_check("cat");

    return 0;
}
```

CP

CP 커맨드의 경우 인자 형식이 맞지 않으면 우선 오류를 처리합니다.

파일의 복사는 입력과 출력을 위한 두개의 파일 디스크립터를 열어서
한 라인씩 읽고 출력하는 식으로 합니다.

파일 디스크립터를 열 때 파일이 존재하지 않거나
권한이 없을 수 있으므로 23번 라인과 같이 에러가 발생한 경우
에러 메세지를 출력하고 프로그램을 종료 합니다.

30번 라인부터 56번 라인까지는 cp 의 dest 부분이
파일의 경로일수도 있고 바꿔야 할 파일일 수도 있으므로
이를 처리하기 위해
dest 의 마지막 문자가 '/' 인 경우 src의 파일명을 붙여주고

마지막 문자가 '/' 가 아닌 경우
해당 문자열이 파일인지 디렉토리 인지 판단하기 위해
opendir 함수의 반환값을 확인하여 디렉토리라면
dest 에 '/' 와 파일이름을 붙여줍니다.

```c
#include "definition.h"
#include "errhandler.h"
#include "dirent.h"

char src[512];

int main(int argc,char * argv[])
{
    set_default();
    if(argc==1) fprintf(stderr,"cp: missing file operand\n");
    else if(argc==2) fprintf(stderr,"cp: missing destination file operand after %s\n",argv[1]);
    else
    {

        int ifd,ofd,ril;
        ifd = ofd = -1;
        char buf[1024];

        set_default();

        ifd = open(argv[1],O_RDONLY);

        if(errno)
        {
            err_check("cp");
            close(ifd);
            return 0;
        }

        int d = -1;
        while(argv[1][++d]);
        while(d>0 && argv[1][--d]!='/');
        if(argv[1][d]=='/') ++d;
        for(int i=0;argv[1][d];++i) src[i] = argv[1][d++];

        int c = -1;
        while(argv[2][++c]);
        int ck = (argv[2][--c] == '/');

        if(ck) strcat(argv[2],src);
        else
        {
            DIR * dcheck = opendir(argv[2]);

```

```
44
45          if(errno == EACCES)
46          {
47              err_check("cp");
48              return 0;
49          }
50
51          if(dcheck)
52          {
53              strcat(argv[2],"/");strcat(argv[2],src);
54          }
55          if(dcheck) closedir(dcheck);
56      }
57
58      set_default();
59
60      if(ifd == -1) ;
61      else
62      {
63          ofd = open(argv[2],O_WRONLY|O_CREAT,0644);
64          while((ril = read(ifd,buf,1024)) > 0) write(ofd,buf,ril);
65      }
66
67      if(ifd != -1) close(ifd);
68      if(ofd !=-1) close(ofd);
69  }
70
71  err_check("cp");
72  return 0;
73 }
                                                          73,1          Bot
```

MV

mv 커맨드의 경우 파일 디스크립터를 지정하지 않고

rename 함수를 이용하여 cp 와 유사한 전처리 후 구현하였습니다.

RM

RM 의 경우 별다른 옵션이 없고

unlink 함수를 이용하면 파일을 삭제할 수 있어

간단히 구현하였습니다.

```
File  Edit  View  Search  Terminal  Help
 1 #include "definition.h"
 2 #include "errhandler.h"
 3
 4 int main(int argc,char * argv[])
 5 {
 6     set_default();
 7     if(argc == 2) remove(argv[1]);
 8     err_check("rm");
 9     return 0;
10 }
~
```

PWD

pwd 커맨드 또한

별다른 옵션 없이

getcwd 함수를 이용하여 배열에 현재 경로를 담은 후

출력하는 식으로 구현하였습니다,

```
 1 #include "definition.h"
 2 #include "errhandler.h"
 3
 4 char pwd[1024];
 5
 6 int main(void)
 7 {
 8     set_default();
 9     getcwd(pwd,sizeof(pwd));
10     printf("%s\n",pwd);
11     err_check("pwd");
12     return 0;
13 }
~
```

MINISHELL

설명에 앞서 C 언어의 특성 상 객체 단위로 사고하기가 힘들어

많은 함수들이 MINISHELL 에 치중된 경향이 있습니다.

지금 생각해보면 파일을 파싱하는 함수들은 따로 분리하는 것이 맞지 않았나 싶습니다,

그렇기에 MINISHELL 의 소스코드는 흐름대로 보기에 복잡한 측면이 있어

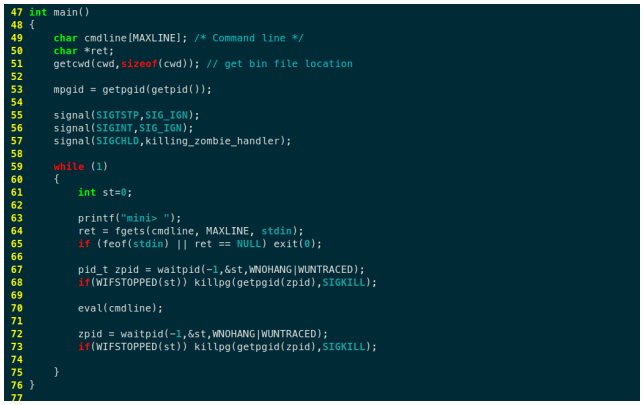
최대한 실행 흐름대로 따라가면서 주 흐름 외 사용되는 함수는 ADT만 간단히 설명하겠습니다.

MAIN

MINISHELL 함수를 구동하는 몸통입니다.

프로그램의 명령을 계속해서 받는 while 루프로 들어가기전

초기 세팅을 하게 됩니다.



```
int main()
{
    char cmdline[MAXLINE]; /* Command line */
    char *ret;
    getcwd(cwd,sizeof(cwd)); // get bin file location

    mpgid = getpgid(getpid());

    signal(SIGTSTP,SIG_IGN);
    signal(SIGINT,SIG_IGN);
    signal(SIGCHLD,killing_zombie_handler);

    while (1)
    {
        int st=0;

        printf("mini> ");
        ret = fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin) || ret == NULL) exit(0);

        pid_t zpid = waitpid(-1,&st,WNOHANG|WUNTRACED);
        if(WIFSTOPPED(st)) killpg(getpgid(zpid),SIGKILL);

        eval(cmdline);

        zpid = waitpid(-1,&st,WNOHANG|WUNTRACED);
        if(WIFSTOPPED(st)) killpg(getpgid(zpid),SIGKILL);

    }
}
```

- getcwd 함수를 이용하여 구현한 커맨드의 실행파일 위치 저장
- getpgid 함수를 이용하여 미니쉘의 그룹id 를 mpgid 변수에 저장
- 미니쉘 함수는 SIGINT,SIGTSP를 무시하도록 설정
- SIGCHLD 시그널에 대해 reaping 하는 함수로 설정

이러한 초기 설정 후 while 루프를 돌면서

터미널로 부터 입력을 받고

eval 함수를 사용하여 명령어를 분석 , 실행하게 됩니다.

이때 eval 함수 전후로 자식프로세스가 중단된 경우 reaping 을 해줍니다.

후에 서술하겠지만 eval 함수에서 명령어를 실행하게 된다면 미니쉘의 프로세스가

명령어를 실행하는 자식프로세스를 생성하고 그룹을 분리한 뒤 그 뒤로 생성되는

자식프로세스는 모두 명령실행프로세스에 의해 생성되고 같은 그룹으로 관리되기 때문에

killpg 함수를 이용하여 명령실행프로세스 그룹의 모든 구성원에게 SIGKILL 을 날려

놓치는 좀비프로세스가 발생하지 않도록 합니다.

EVAL

명령어를 평가하고 실행하는 함수입니다.

이 함수에 너무 많은 로직을 담은게 프로그램의 단점입니다.

우선 사용되는 변수들을 선언,초기화 한 뒤

90번 라인에서 기본적인 파싱을 거칩니다.

95번 라인에서 단축평가로

invalid_checker 와 parsecommand 함수를 사용합니다.

기본적으로 invaild_check 에서 커맨드가 유효한 커맨드 인지 검사하고

유효하다면 0을 반환하여 parsecommand 함수가 실행되게 됩니다,

parsecommand 함수는 parseline 함수에서 1차적으로 파싱된 배열을

파이프 단위로 다시 토큰화 합니다.

이때 86번 라인 cvector vsz,vtp 배열에

토큰화된 커맨드의 인자 개수와 리다이렉션 종류가 담기게 됩니다.

parsecommand 단계에서도 2차적으로 커맨드의 유효성을 검증하여

결과적으로 유효하지 않은 커맨드에 대해서는 97번 라인으로

유효한 커맨드의 경우 102 번 라인으로 가서 명령어를 수행하게 됩니다.

102번 라인에 도달하는 시점에 커맨드는 vsz,vtp 배열에 담긴 정보로

토큰화 되있다고 할 수 있습니다.

108라인에 백그라운드가 아니면서 cd,exit인 경우 실행 후 eval 함수를 종료하게 됩니다.

115번 라인부터 133번 라인까지 프로세스 관리의 핵심을 담고 있습니다.

미니쉘이 자식을 생성한 후 foreground 라면 명령이 전부 실행되고 종료되기 까지 기다리고

background 라면 main 함수 반복문으로 혼자 돌아가게 됩니다.

미니쉘에 생성된 자식프로세스를 편의상 명령프로세스라고 칭하겠습니다.

123라인에서 명령 프로세스는 곧바로 새로운 그룹을 자신의 id로 설정하게 됩니다.

이 때 백그라운드 옵션이 아니라면 tcsetpgrp 함수를 이용하여 세션의 터미널과 통하는

그룹을 자신으로 만듭니다.(자신의 그룹을 포어그라운드 그룹으로 설정합니다,)

```
78 void eval(char *cmdline)
79 {
80     char *argv[MAXARGS]; /* Argument list execve() */
81     char buf[MAXLINE];   /* Holds modified command line */
82     int bg;              /* Should the job run in bg or fg? */
83
84     pid_t pid;
85
86     cvector vsz,vtp;
87     cv_init(&vsz,4); cv_init(&vtp,4);
88
89     strcpy(buf, cmdline);
90     bg = parseline(buf, argv);
91
92     /* Ignore empty lines */
93     if (argv[0] == NULL) return;
94
95     if(invaild_checker(argv) || parsecommand(argv,&vsz,&vtp))
96     {
97         fprintf(stderr,"mini: command not found\n");
98         cv_clear(&vsz);
99         cv_clear(&vtp);
100        return;
101    }
102
103    int cn = cv_size(&vsz);
104    int c=0;
105    int fd[2];
106    int bfd=0;
107
108    if(!bg && builtin_command(argv))
109    {
110        cv_clear(&vsz);
111        cv_clear(&vtp);
112        return;
113    }
114
115    if((ppid=fork()))
116    {
117        cv_clear(&vsz);
118        cv_clear(&vtp);
119        if(!bg) waitpid(ppid,NULL,0);
120        return;
121    }
122
123    setpgrp();
124
125    if(!bg) // swap session ground
126    {
127        signal(SIGTTOU,SIG_IGN);
128        tcsetpgrp(STDOUT_FILENO,getpgid(getpid()));
129    }
130
131    signal(SIGTSTP,SIG_DFL);
132    signal(SIGINT,ign_handler);
133
134    if(cmap(argv[0]) < 3) // built-in background
135    {
136        cv_clear(&vsz);
137        cv_clear(&vtp);
138        exit(0);
139    }
140
142    for(int i=0;i<cn;++i,++c)
143    {
144        int saz = vsz.arr[i] - (vtp.arr[i]/3) * 2;
145        int pc = c;
146
147        char ** command = malloc(sizeof(char*) * (saz+1));
148
149        char * f1 = NULL;
150        char * f2 = NULL;
151
152        for(int j = 0; c-pc< vsz.arr[i]; ++j)
153        {
154            int x = cmap(argv[c]);
155            if(x >= 0) command[j] = argv[c++];
156            else
157            {
158                ++c;
159                if(f1) f2 = argv[c++];
160                else f1 = argv[c++];
161                --j;
162            }
163        }
164
165        command[saz] = NULL;
166
167        pipe(fd);
168
169        pid = fork();
170
171        if (!pid)
172        {
173            char path[512];
174
175            dup2(bfd,0);
176
177            if(i != cn-1) dup2(fd[1],STDOUT_FILENO);
178
179            close(fd[0]);
180
181            int tp = vtp.arr[i];
182
183            if(tp == 4 || tp == 5) // output redirection
184            {
185                tp -=4;
186                int ofd = open(f1,(tp*O_APPEND)|O_WRONLY| O_CREAT,0644);
187                dup2(ofd,STDOUT_FILENO);
188            }
                                                                188,5         44%
```

SIGTSTP,SIGINT를 무시하는 미니쉘 프로세스와는 다르게 명령프로세스는 이 시그널들을 받아야합니다.

다만 SIGINT 의 경우 명령프로세스가 SIGINT 를 받아 종료되는 경우 명령프로세스의 자식프로세스들은

좀비 프로세스가 되므로 이를 막기 위해 또다른 handler를 SIGINT에 설정합니다.

134번 라인은 background 이면서 built-in command 경우 아무런 동작도 하지않으므로 이를 맞춰주기 위해서

넣었습니다. 조금 억지스럽게 구현한 느낌이 있습니다.

152번 라인부터 251 라인까지 명령프로세스가 명령어를 실행하는 부분입니다.

for 문으로 파이프로 분리된 인자개수만큼 돌면서

144번 라인부터 165번 라인까지 parsecommand 함수에서 담은 정보를 가지고

실제적인 토큰화를 하게 됩니다.

이때 리다이렉션 뒤에 오는 파일이름은 차례대로 f1,f2 에 할당되게 되고

char ** command 에는 실행하는 커맨드 옵션 인자만 담기게 됩니다.

그 뒤로는 파이프라인과 리다이렉션에 대한 처리가 이루어 지게 되는데

명령프로세스가 커맨드마다 fork 하여서 자식은 실제적인 실행을 하고

명령프로세스는 자식의 종료를 기다리게 됩니다.

마지막 커맨드가 아니면 다음 파이프라인으로 dup2 를 사용하여 출력을 보내고

첫 커맨드가 아니면 dup2 로 전 파이프라인에서 입력을 받게 됩니다.

vtp 에 저장되어있는 커맨드의 리다이렉션 타입에 따라

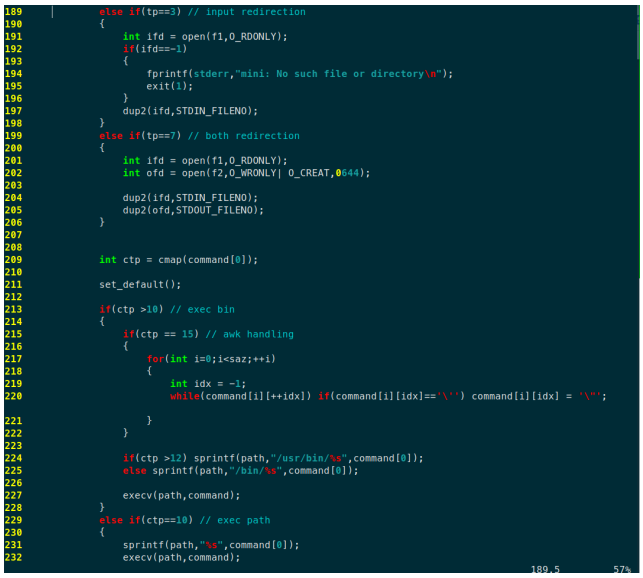
파일 디스크립터를 생성 후 dup2 함수로 적절히 연결한 후

213 번 라인부터 238 번 라인까지 execv 함수를 이용해 명령을 실행 후 종료하게 됩니다.

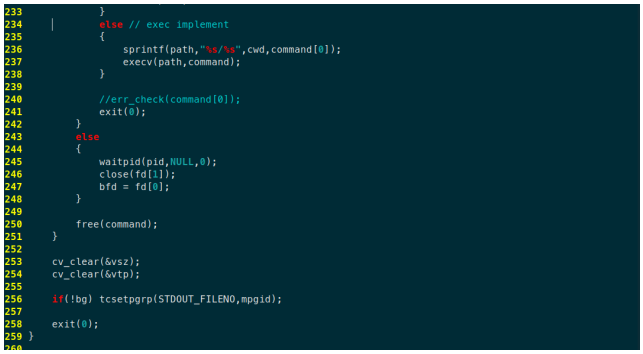
마지막 명령 프로세스의 종료 시

명령프로세스가 포어그라운드 였다면 다시 미니쉘 프로세스를

포어그라운드 그룹으로 만들고 종료하게 됩니다.



```
189         else if(tp==3) // input redirection
190         {
191             int ifd = open(f1,O_RDONLY);
192             if(ifd==-1)
193             {
194                 fprintf(stderr,"mini: No such file or directory\n");
195                 exit(1);
196             }
197             dup2(ifd,STDIN_FILENO);
198         }
199         else if(tp==7) // both redirection
200         {
201             int ifd = open(f1,O_RDONLY);
202             int ofd = open(f2,O_WRONLY| O_CREAT,0644);
203 
204             dup2(ifd,STDIN_FILENO);
205             dup2(ofd,STDOUT_FILENO);
206         }
207 
208 
209         int ctp = cmap(command[0]);
210 
211         set_default();
212 
213         if(ctp >10) // exec bin
214         {
215             if(ctp == 15) // awk handling
216             {
217                 for(int i=0;i<saz;++i)
218                 {
219                     int idx = -1;
220                     while(command[i][++idx]) if(command[i][idx]=='\'') command[i][idx] = '\"';
221                 }
222             }
223 
224             if(ctp >12) sprintf(path,"/usr/bin/%s",command[0]);
225             else sprintf(path,"/bin/%s",command[0]);
226 
227             execv(path,command);
228         }
229         else if(ctp==10) // exec path
230         {
231             sprintf(path,"%s",command[0]);
232             execv(path,command);
```



```
233         }
234         else // exec implement
235         {
236             sprintf(path,"%s/%s",cwd,command[0]);
237             execv(path,command);
238         }
239 
240         //err_check(command[0]);
241         exit(0);
242     }
243     else
244     {
245         waitpid(pid,NULL,0);
246         close(fd[1]);
247         bfd = fd[0];
248     }
249 
250     free(command);
251 }
252 
253 cv_clear(&vsz);
254 cv_clear(&vtp);
255 
256 if(!bg) tcsetpgrp(STDOUT_FILENO,mpgid);
257 
258 exit(0);
259 }
```

결 론

이 리포트에서는 직접 배쉬쉘을 모방한 미니쉘 프로그램을 구현하고
전체적인 흐름과 특정한 문제들에 대해서 간단히 다루었습니다.

내부 구현에 대한 정확한 이해는 소스코드를 보고도 이해할 수 있게 끔
단순하게 짰다고 생각합니다.

과제를 구현하면서 시그널에 대한 이해와 프로세스에 대한 이해가 많이 향상된 것 같습니다.
다만 위에서 계속 언급하였듯이 모듈화 측면에서 아쉬움이 많이 남습니다.
객체지향언어를 사용한다면 더욱 더 깔끔하게 쉘을 구현할 수 있었지 않았을까 싶습니다.