Assignment 4
"mysh 구현"

학과 : 공과대학 소프트웨어학과
학번 : 32164120
이름 : 정성구

1. mysh

shell은 리눅스 인터페이스상의 명령어 해석기이다. 사용자로부터 input을 받으면 그 명령어를 parsing하고, 외부 명령어일 경우에는 fork()후 자식 프로세스가 명령어를 실행하게 하고, 내부 명령어일 경우에는 shell 내부적으로 처리하는 구조를 가지고 있다. 과제의 요구 조건은 *1. 쉘의 기본 로직 구현(ls, ps, gcc 등이 동작하도록), 2. background processing 구현* 이다. 또한 보너스 조건으로 *redirection 구현*이 있었다.

이 기능들을 구현하기 위해 서술한 요구 조건의 순서대로 구현을 해보았다.

1-1) shell의 기본 로직 구현

쉘의 기본 로직은 input을 parsing하고 외부 명령어를 실행하기 위해 fork()를 실행하고 이로 만들어진 자식 프로세스가 execve()를 통해 parsing한 명령어를 수행하는 구조이다. 이를 구현하기 위해 강의노트 5의 예시의 구조를 참고해서 만들게 되었다.

1-1-1) main함수

```
int main(int argc, char* argv[])
{
        char str[MAX_BUF];
        while(1)
        {
                printf("%s$ ",get_current_dir_name());
                fgets(str,sizeof(str)-1,stdin);
                str[strlen(str)-1] = '\0';
                if(!shell_run(str))
                {
                        break;
                }
        }
        return 0;
}
```

main 함수는 prompt($)를 출력하고 사용자로부터 input을 입력받아 그 input을 shell_run함수에 전달한다.

구현)

- 우선 input을 입력받기 위한 배열 str을 선언한다. MAX_BUF는 512이다.
- while문을 무한루프로 돌리면서 prompt와 현재 디렉토리의 위치를 출력한다.
- fgets()함수를 이용하여 명령어를 입력받는다.

  -> 이때 fgets()함수는 개행문자까지 입력받으므로 개행문자를 '\0'로 지워준다.
- shell_run()함수에 input이 저장된 배열 str를 인자로 주고 그 return값에 따라 루프를 지속할지, 탈출할지를 정한다.

1-1-2) shell_run()함수

```
//shell을 구동하는 함수//
int shell_run(char str[])
{
        char* token[MAX_BUF];
        int flag;
        tokenize(str,token);
        flag = advanced_func(token);
        //명령어를 입력하지 않았거나 &를 맨 앞에 입력한 경우//
        //바로 prompt 출력//
        if(token[0] == NULL)
        {
                //&의 경우는 실제 shell에서 출력되는 메세지 출력//
                if(flag){printf("syntax error near unexpected token '&'\n");}
                return 1;
        }
        //exit일 경우 mysh 종료//
        else if(!strcmp(token[0],"exit"))
        {
                return 0;
        }
        else if(!strcmp(token[0],"help") || !strcmp(token[0],"?"))
        {
                cmd_help(token);
                return 1;
        }

        switch(flag){
                //background와 일반 명령어 처리//
                case 0: case 1:
                        normal_cmd(token,flag);
                        break;
                //redirection 처리//
                case 2: case 3: case 4:
                        redirection_cmd(token,flag);
                        break;
                //cd 구현//
                case 5:
                        if(chdir(token[1]) < 0)
                        {
                                printf("chdir error\n");
                                break;
                        }
                        break;
        }
        return 1;
}
```

shell_run()함수는 mysh구동의 중심이 되는 함수이다. 이 함수에서 input을 parsing하는 tokenize()함수, flag를 설정하는 advanced_func()함수, 이후의 명령어를 실행할 normal_cmd(), redirection_cmd()등을 호출한다.

구현)

- 명령어를 parsing한 token들을 저장할 문자열 배열 **token**을 선언한다.
- 내, 외부 명령어와 redirection, background processing등을 구분할 변수 **flag**를 선언한다.
- parsing을 위해 token과 str를 인자로 하여 **tokenize()**함수를 호출한다.
- flag설정을 위해 token을 인자로 하여 **advanced_func()**함수를 호출한다.
- 다음은 예외처리이다.

-> token[0]이 NULL일 경우는 1. 명령어를 아무것도 입력하지 않았거나 2. 맨 앞에 &를 입력했을 경우이다. 이 경우에는 아래의 명령어 실행 과정이 필요없으므로 1을 return하여 main()함수로 돌아가 루프를 계속 수행한다.

* &로 인한 에러는 이 경우 하나밖에 없기 때문에 기본 shell에서 출력되는 에러 메시지를 출력

-> token[0]이 "exit"일 경우는 mysh를 종료하므로 0을 return하여 루프를 탈출하고 프로그램을 종료한다.

-> token[0]이 "help"나 "?"일 경우는 **cmd_help()**함수를 호출하여 안내문을 출력한다. 이후에는 1을 return하고 main()함수로 돌아가 다시 루프를 수행한다.

- 이후 switch문을 사용하여 flag의 값에 따라 명령어를 실행한다.

-> flag가 0,1일 경우는 token과 flag를 인자로 해서 **normal_cmd()**함수를 호출한다.

-> flag가 2,3,4일 경우는 token과 flag를 인자로 해서 **redirection_cmd()**함수를 호출한다.

-> flag가 5일 경우는 내부 명령어 "**cd**"를 실행한다.

- 이후 1를 return하며 main()으로 돌아가 다시 루프를 실행한다.

1-1-3) tokenize()함수

```
//parsing 작 업 하 는  함 수 //
int tokenize(char str[],char* tok[])
{
        int i = 0;
        tok[i] = strtok(str," ");

        for(;tok[i] != NULL && i < MAX_BUF;)
        {
                i++;
                tok[i] = strtok(NULL," ");
        }

        return i;
}
```

tokenize()함수는 input을 parsing하는 함수이다.

구현)

- 반복문을 조절할 변수 i를 선언한다.
- strtok()함수를 이용하여 명령어를 token으로 분리한다.

-> strtok()함수는 첫 번째 인자의 문자열을 두 번째 문자열로 나누는 함수이다.

-> strtok()함수는 첫 번째 토큰을 얻을 때만 분리할 문자열을 입력받고 그 후부터는 구분 문자열만을 입력받는다.

-> strtok()함수는 문자열의 현재 위치를 가리키는 내부 포인터를 유지한다고 한다.

- 아래 for문의 결과로 i는 token의 개수가 된다.
- i를 return한다.

1-1-4) advanced_func() 함수

```
//parsing한 명령어를 분석해 flag를 세팅하는 함수//
int advanced_func(char *token[])
{
        int i, flag = 0;

        for(i=0;token[i] != NULL && i < MAX_BUF;i++)
        {
                if(!strcmp(token[i],"&"))
                {
                        token[i--] = '\0';
                        flag = 1;
                }
                else if(!strcmp(token[i],">"))
                {
                        flag = 2;
                }
                else if(!strcmp(token[i],"<"))
                {
                        flag = 3;
                }
                else if(!strcmp(token[i],">>"))
                {
                        flag = 4;
                }
                else if(!strcmp(token[i],"cd"))
                {
                        flag = 5;
                }
        }
        return flag;
}
```

advanced_func()함수는 shell의 advanced function들과 내부 명령어들을 구분하는 flag를 설정하는 함수이다.

구현)

- flag를 저장할 변수 flag를 설정한다. default값은 0
- token[i]이 NULL이 아니고 i가 MAX_BUF보다 작을 때까지 반복문을 돌린다.
- token 중에 "&"가 있을 경우 background processing을 위해 flag를 1로 설정

-> 이후에 execve()시 "&"는 인식이 되지 않고, 예외처리를 위해서도 '\0'으로 지운다.

- token 중에 ">"가 있을 경우 redirection을 위해 flag를 2로 설정
- token 중에 "<"가 있을 경우 redirection을 위해 flag를 3로 설정
- token 중에 ">>"가 있을 경우 redirection을 위해 flag를 4로 설정
- token 중에 "cd"가 있을 경우 명령어 실행을 위해 flag를 5로 설정
- flag를 return한다.

1-1-5) cmd_help() 함수

```
//help나 ?시 출력하는 안내문//
void cmd_help(char *tok[])
{
        printf("--------------------mysh--------------------\n");
        printf("|                                           |\n");
        printf("|    쉘의 기본적인 로직을 구현했습니다.    |\n");
        printf("|          *추가로 구현한 기능들*           |\n");
        printf("| cd : change directory                     |\n");
        printf("| exit : exit mysh                          |\n");
        printf("| & : background processing                 |\n");
        printf("| >,<,>> : redirection                      |\n");
        printf("| help,? : show this message                |\n");
        printf("---------------------------------------------\n");
}
```

cmd_help()함수는 mysh이 “help”나 “?”를 입력받았을 때 호출되며 입력한 안내문을 출력한다.

1-1-6) normal_cmd() 함수

```
//일반적인 명령어들과 background processing 실행//
void normal_cmd(char *tok[], int flag)
{
        pid_t fork_return;
        if((fork_return = fork()) < 0)
        {
                printf("fork error\n");
                exit(-1);
        }
        else if(fork_return == 0)
        {
                if(execvp(tok[0],tok) < 0)
                {

                        printf("%s : command not found!\n",tok[0]);
                        exit(-1);
                }
        }
        else
        {
                //flag가 1이 아니라면 자식 프로세스의 종료를 기다림//
                if(!flag)
                {
                        wait();
                }
        }
}
```

normal_cmd() 함수는 flag가 0,1일 경우, 즉 일반적인 외부 명령어와 background processing을 실행하는 함수이다.

구현)

- fork()의 return값을 받을 변수 fork_return을 선언한다.
- fork()를 시도한다.

-> fork_return이 음수라면 fork()가 실패한 것이므로 에러 메시지와 함께 프로세스를 종료한다.

-> fork_return이 0이라면 자식 프로세스이다. 자식 프로세스에서 execvp()함수로 token을 인자로 하여 명령어를 실행한다.

-> fork_return이 양수라면 부모 프로세스이다. 부모 프로세스는 flag가 0일 시 wait()를 통해 자식 프로세스가 종료되는 것을 기다린다.

Snapshot)

```
sys32164120@embedded: ~
sys32164120@embedded:~$ vim mysh.c
sys32164120@embedded:~$ gcc -g mysh.c -o mysh
sys32164120@embedded:~$ ./mysh
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test.c
/home/2-class/sys32164120$ clear
/home/2-class/sys32164120$ ls -l
total 52
-rw-rw-r-- 1 sys32164120 sys32164120   424 10월  29 02:15 3061.c
-rw-rw-r-- 1 sys32164120 sys32164120     6 10월  29 10:11 arg
drwxrwxr-x 2 sys32164120 sys32164120  4096  9월  23 21:10 assignment
drwxrwxr-x 3 sys32164120 sys32164120  4096 10월  19 19:14 assignment3
drwxrwxr-x 2 sys32164120 sys32164120  4096 10월  29 10:14 datastructure
-rw-r--r-- 1 sys32164120 sys32164120  8980  4월  20  2016 examples.desktop
-rwxrwxr-x 1 sys32164120 sys32164120 11996 10월  29 17:32 mysh
-rw-rw-r-- 1 sys32164120 sys32164120  4278 10월  29 17:32 mysh.c
-rw-rw-r-- 1 sys32164120 sys32164120     0 10월  29 10:48 test.c
/home/2-class/sys32164120$ ps
  PID TTY          TIME CMD
19486 pts/28   00:00:00 bash
30442 pts/28   00:00:00 mysh
31215 pts/28   00:00:00 ps
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test.c
/home/2-class/sys32164120$ mkdir test
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test  test.c
/home/2-class/sys32164120$ rmdir test
/home/2-class/sys32164120$ cat test.c
/home/2-class/sys32164120$ vim test.c
/home/2-class/sys32164120$ vim test.c
/home/2-class/sys32164120$ cat test.c
#include<stdio.h>

int main()
{
        printf("hi! this is mysh!\n");
        return 0;
}
/home/2-class/sys32164120$ gcc -o test.out test.c
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test.c  test.out
/home/2-class/sys32164120$ ./test.out
hi! this is mysh!
/home/2-class/sys32164120$ whoami
sys32164120
/home/2-class/sys32164120$ date
2020. 10. 29. (목 ) 17:36:10 KST
/home/2-class/sys32164120$
```

1-2) Background Processing 구현

background processing은 실행할 명령어의 마지막에 &를 붙이면 실행된다. 이 방식을 통해 하나의 쉘에서 여러 개의 프로세스를 실행할 수 있다. background processing을 실행하면 명령어를 실행한 직후 shell의 prompt가 출력되며 실행한 명령은 background에서 돌아가고, shell은 또 다른 명령어를 받아서 실행할 수 있다.

구현)

```
//flag가 1이 아니라면 자식 프로세스의 종료를 기다림//
if(!flag)
{
        wait();
}
```

background processing은 따로 구현한다기 보다는 wait()의 유무 차이이다. flag가 0일 경우는 foreground processing이므로 부모 프로세스는 자식 프로세스가 명령어를 종료되고 실행할 때 까지 기다려야 한다. 그러므로 wait()함수를 이용해 자식 프로세스가 종료된 후 main()으로 돌아와 prompt를 출력한다.

flag가 1일 경우는 wait()함수를 거치지 않는다. 고로 부모 프로세스는 즉시 main()함수로 돌아와서 prompt를 출력하고 다시 명령어를 받을 수 있다. 자식 프로세스는 background에서 실행되고 종료된다.

snapshot)

```
sys32164120@embedded: ~
sys32164120@embedded:~$ ./mysh
/home/2-class/sys32164120$ sleep 100 &
/home/2-class/sys32164120$ ps
  PID TTY          TIME CMD
 5796 pts/28   00:00:00 mysh
 5982 pts/28   00:00:00 sleep
 6029 pts/28   00:00:00 ps
19486 pts/28   00:00:00 bash
/home/2-class/sys32164120$ ./test.c
./test.c : command not found!
/home/2-class/sys32164120$ ./test.out
hi! this is mysh!
/home/2-class/sys32164120$ sleep 50 &
/home/2-class/sys32164120$ ps
  PID TTY          TIME CMD
 5796 pts/28   00:00:00 mysh
 5982 pts/28   00:00:00 sleep
 6552 pts/28   00:00:00 sleep
 6577 pts/28   00:00:00 ps
19486 pts/28   00:00:00 bash
/home/2-class/sys32164120$ whoami
sys32164120
/home/2-class/sys32164120$ date
2020. 10. 29. (목 ) 17:38:18 KST
/home/2-class/sys32164120$
```

1-3) cd 구현

cd는 디렉토리를 이동하는 명령어이다. 이 명령어는 매우 기본적인 명령어이지만 내부 명령어이기 때문에 따로 구현을 해 주어야 작동한다. 굳이 구현할 필요는 없었지만 뭔가 shell에 이 명령어가 없다면 어색한 느낌이 들어서 구현해보았다.

구현)

```
//cd 구현 //
case 6:
        if(chdir(token[1]) < 0)
        {
                printf("chdir error\n");
                break;
        }
        break;
```

chdir()은 인자로 들어온 directory path로 현재 작업 디렉토리를 변경하는 system call이다. 위의 advanced_func()함수에서 token을 검사하다 cd를 발견하면 flag에 6를 세팅한다. 이후 shell_run()의 switch문에서 flag가 6일 경우 실행한다.

chdir()함수에 token[1]을 인자로 준 이유는 cd명령어가 "cd foldername"같은 식으로 쓰이기 때문이다. 인자로 directory path를 주는 것이다. 그 다음 chdir()함수는 정상적으로 작동되지 않았을 시 음수값을 반환하므로 그에 따른 에러처리를 해준다.

snapshot)

```
sys32164120@embedded: ~
/home/2-class/sys32164120$ exit
sys32164120@embedded:~$ ./mysh
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test.c  test.out
/home/2-class/sys32164120$ cd assignment
/home/2-class/sys32164120/assignment$ ls
assignment2  copythemall
/home/2-class/sys32164120/assignment$ cd ..
/home/2-class/sys32164120$ cd ..
/home/2-class$ ls
sys32143180  sys32164682  sys32171207  sys32173047  sys32174435  sys32180805  sys32187505  sys32190789
sys32152446  sys32167548  sys32171536  sys32173058  sys32174534  sys32181070  sys32187545  sys32191865
sys32160827  sys32170078  sys32171603  sys32173243  sys32174727  sys32182861  sys32187588  sys32192881
sys32161337  sys32170140  sys32171607  sys32173437  sys32174774  sys32183118  sys32187755  sys32193078
sys32162956  sys32170260  sys32171723  sys32173441  sys32174919  sys32183897  sys32190473  sys32193120
sys32163540  sys32170402  sys32172183  sys32173698  sys32174937  sys32187225  sys32190590  sys32193386
sys32164120  sys32171204  sys32172479  sys32173717  sys32177457  sys32187268  sys32190661  sys32193406
/home/2-class$ cd sys32164120
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test.c  test.out
/home/2-class/sys32164120$ whoami
sys32164120
/home/2-class/sys32164120$ date
2020. 10. 29. (목 ) 17:52:43 KST
/home/2-class/sys32164120$ exit
sys32164120@embedded:~$
```

1-4) redirection 구현

이 과제를 시작하기 전 알고 있는 redirection의 종류는 '>', '<' 두 가지뿐이었는데 과제를 수행하기 위해 조사를 하면서 '>>'의 존재를 알게 되었다. '>>'는 redirection시 파일의 끝에서부터 이어서 출력한다는 뜻이라고 한다. '>'은 출력 재지정으로써 '>'앞의 실행결과를 '>'뒤의 실행결과에 써준다. 원래대로면 terminal에 출력되어야 하는 내용을 원하는 파일에 저장하도록 하는 것이다. '<'는 입력 재지정으로써 terminal로부터 입력값을 받는 대신 '<' 뒤의 파일의 내용을 입력값으로 받게 만든다. '<<'는 존재하지 않는 것 같다.

1-4-1) redirection_cmd() 함수

```
//redirection 구 현 //
void redirection_cmd(char *tok[],int flag)
{
        pid_t fork_return;
        int i, j, fd;
        char *part1[MAX_BUF], *part2[MAX_BUF];

        for(i=0;i<MAX_BUF;i++)
        {
                if(!strcmp(tok[i],">") || !strcmp(tok[i],"<") || !strcmp(tok[i],">>"))
                break;

                part1[i] = tok[i];
        }
        part1[i] = '\0';

        for(i=i+1,j=0;tok[i] != NULL && i < MAX_BUF;i++,j++)
        {
                part2[j] = tok[i];
        }
        part2[j] = '\0';
```

```
        //'>', '>>' redirection 구현 //
        if(flag == 2 || flag == 4)
        {
                if((fork_return = fork()) < 0)
                {
                        printf("fork error\n");
                        exit(-1);
                }
                else if(fork_return == 0)
                {
                        if(flag == 2)
                        {
                                fd = open(part2[0],O_RDWR | O_CREAT, 0664);
                        }
                        else if(flag == 4)
                        {
                                fd = open(part2[0], O_RDWR | O_CREAT | O_APPEND, 0664);
                        }
                        dup2(fd,STDOUT_FILENO);
                        close(fd);
                        if(execvp(part1[0],part1) < 0)
                        {
                                printf("%s : command not found\n");
                                exit(-1);
                        }
                }
                else
                {
                        wait();
                }
        }
```

```
        //'<' redirection 구현 //
        else if(flag == 3)
        {
                if((fork_return = fork()) < 0)
                {
                        printf("fork error\n");
                        exit(-1);
                }
                else if(fork_return == 0)
                {
                        fd = open(part2[0],O_RDWR | O_CREAT, 0664);
                        dup2(fd,STDIN_FILENO);
                        close(fd);
                        if(execvp(part1[0],part1) < 0)
                        {
                                printf("%s : command not found\n");
                                exit(-1);
                        }
                }
                else
                {
                        wait();
                }
        }
}
```

구현)

- fork()의 return값을 저장할 fork_return, open시 file descripter를 저장할 변수 fd를 선언한다.
- >,<,>>를 기준으로 앞 뒤로 나눈 문자열을 저장할 배열 part1, part2를 선언한다.
- 반복문을 이용해서 >,<,>> 앞의 토큰들을 part1에 저장한다.

-> 반복문이 종료된 후 part1[i]는 초기화되지 않은 쓰레기값이 들어있다. 그러므로 '\0'를 삽입해서 초기화함과 동시에 인덱스 i-1를 마지막 성분으로 만들어준다.

- 그 다음 token의 >,<,>> 다음 문자열 토큰들을 part2에 저장한다.

-> 반복문이 종료된 후 part2[j]는 초기화되지 않은 쓰레기값이 들어있다. 그러므로 '\0'를 삽입해서 초기화함과 동시에 인덱스 j-1를 마지막 성분으로 만들어준다.

- part2는 [0]부터 데이터를 저장해야 하지만 tok는 >,<,>>이후의 문자열들을 주어야 하므로 인덱스의 변수를 다르게 사용한다.

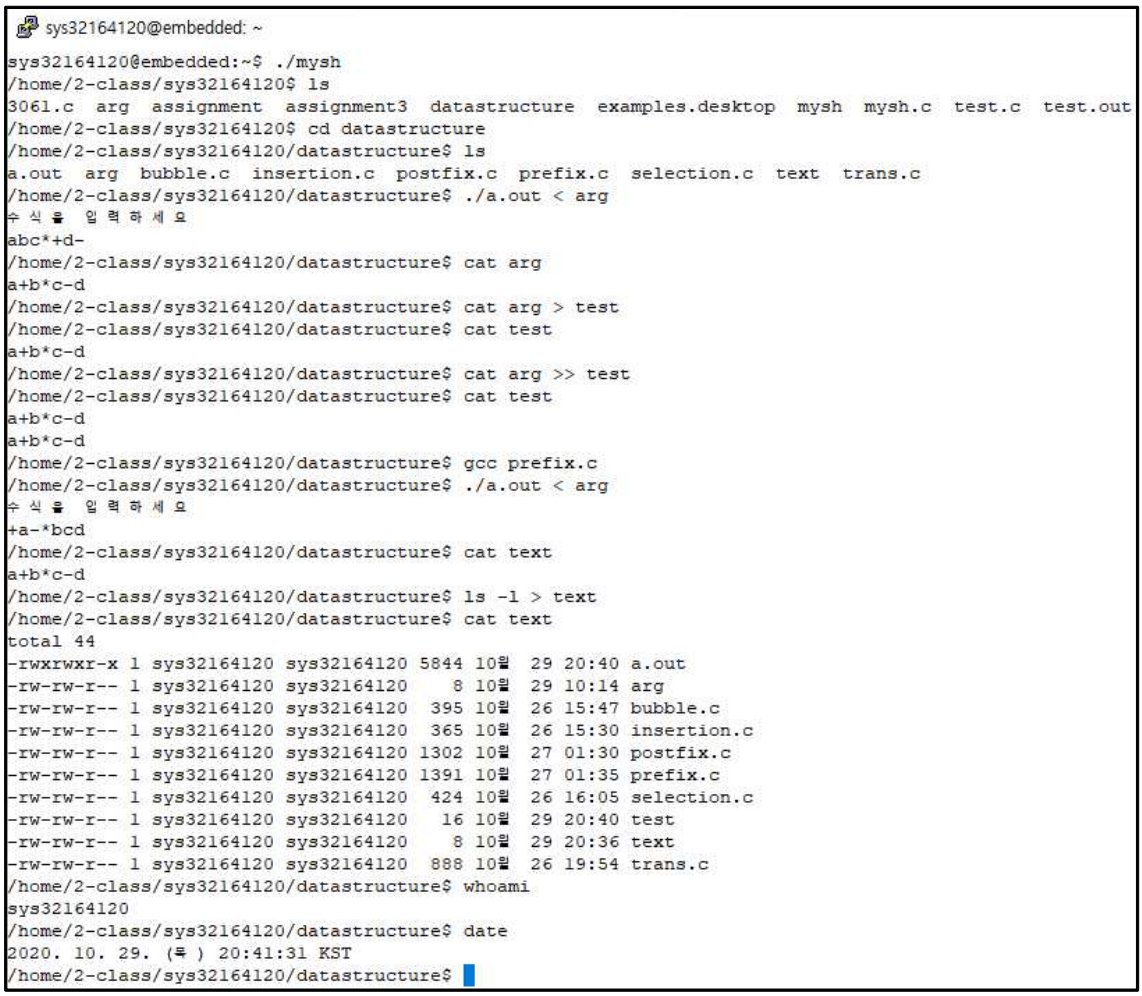
- flag가 2또는 4일 경우 '>', '>>'의 redirection을 실행한다.
- fork()를 실행한다. fork_return == 0이면 자식 프로세스, 양수이면 부모 프로세스이다.
- 부모 프로세스는 wait()함수로 자식 프로세스가 끝나길 기다린다.
- 자식 프로세스는 flag가 2일 경우 '>'를, 4일 경우 '>>'를 실행한다.

-> 우선 open()을 통해 대상 파일(part2[0])을 open한다.
-> 대상 파일이 존재하지 않을 경우 생성해야 하므로 O_CREAT flag를 추가한다.
-> '>'는 출력 재지정이고 대상 파일의 내용이 있으면 그 위에 덮어씌운다.
-> '>>'는 파일의 끝에 이어서 출력을 저장하므로 O_APPEND flag를 추가한다.
-> 파일을 open한 뒤 dup2를 통해 STDOUT_FILENO를 fd와 연결한다.
-> 이후 execve()를 통해 part1의 명령어를 실행하면 그 실행 결과가 대상 파일에 쓰여진다.

- flag가 3일 경우는 '<'의 redirection을 실행한다.
- fork()를 실행한다. fork_return == 0이면 자식 프로세스, 양수이면 부모 프로세스이다.
- 부모 프로세스는 wait()함수로 자식 프로세스가 끝나길 기다린다.
- 자식 프로세스는 '<'를 실행한다.

-> 우선 open()을 통해 대상 파일(part2[0])을 open한다.
-> '<'는 입력 재지정이고 대상 파일의 내용을 '<'앞 파일의 입력값으로 준다.
-> 파일을 open한 뒤 dup2를 통해 STDIN_FILENO를 fd와 연결한다.
-> 이후 execve()를 통해 part1의 명령어를 실행하면 대상 파일의 내용이 input으로서 주어진다.

snapshot)


sys32164120@embedded: ~
sys32164120@embedded:~$ ./mysh
/home/2-class/sys32164120$ ls
3061.c arg assignment assignment3 datastructure examples.desktop mysh mysh.c test.c test.out
/home/2-class/sys32164120$ cd datastructure
/home/2-class/sys32164120/datastructure$ ls
a.out arg bubble.c insertion.c postfix.c prefix.c selection.c text trans.c
/home/2-class/sys32164120/datastructure$ ./a.out < arg
수 식 을 입 력 하 세 요
abc*+d-
/home/2-class/sys32164120/datastructure$ cat arg
a+b*c-d
/home/2-class/sys32164120/datastructure$ cat arg > test
/home/2-class/sys32164120/datastructure$ cat test
a+b*c-d
/home/2-class/sys32164120/datastructure$ cat arg >> test
/home/2-class/sys32164120/datastructure$ cat test
a+b*c-d
a+b*c-d
/home/2-class/sys32164120/datastructure$ gcc prefix.c
/home/2-class/sys32164120/datastructure$ ./a.out < arg
수 식 을 입 력 하 세 요
+a-*bcd
/home/2-class/sys32164120/datastructure$ cat text
a+b*c-d
/home/2-class/sys32164120/datastructure$ ls -l > text
/home/2-class/sys32164120/datastructure$ cat text
total 44
-rwxrwxr-x 1 sys32164120 sys32164120 5844 10월 29 20:40 a.out
-rw-rw-r-- 1 sys32164120 sys32164120 8 10월 29 10:14 arg
-rw-rw-r-- 1 sys32164120 sys32164120 395 10월 26 15:47 bubble.c
-rw-rw-r-- 1 sys32164120 sys32164120 365 10월 26 15:30 insertion.c
-rw-rw-r-- 1 sys32164120 sys32164120 1302 10월 27 01:30 postfix.c
-rw-rw-r-- 1 sys32164120 sys32164120 1391 10월 27 01:35 prefix.c
-rw-rw-r-- 1 sys32164120 sys32164120 424 10월 26 16:05 selection.c
-rw-rw-r-- 1 sys32164120 sys32164120 16 10월 29 20:40 test
-rw-rw-r-- 1 sys32164120 sys32164120 8 10월 29 20:36 text
-rw-rw-r-- 1 sys32164120 sys32164120 888 10월 26 19:54 trans.c
/home/2-class/sys32164120/datastructure$ whoami
sys32164120
/home/2-class/sys32164120/datastructure$ date
2020. 10. 29. (목) 20:41:31 KST
/home/2-class/sys32164120/datastructure$

1-5) GDB 사용하기

mysh의 기본 로직과 background processing을 구현하고 redirection을 구현할 때 출력 재지정을 구현하는 중 출력값이 대상 파일로 써지지 않고 계속 터미널에 써지는 현상이 있었다. 이 현상을 해결하기 위해 gdb를 사용해 보았다. 분명 redirection과정에서 문제가 발생하는 것이라 생각하여 redirection_cmd()함수에 break point를 걸고 한 줄씩 실행해보기로 했다.

```
(gdb) l
warning: Source file is more recent than executable.
216                     }
217                     else if(!strcmp(token[i],"cd"))
218                     {
219                             flag = 5;
220                     }
221             }
222             return flag;
223     }
224
225     int main(int argc, char* argv[])
(gdb) b redirection_cmd
Breakpoint 1 at 0x8048793: file mysh.c, line 59.
(gdb) r
Starting program: /home/2-class/sys32164120/mysh
/home/2-class/sys32164120$ ls
3061.c  arg  assignment  assignment3  datastructure  examples.desktop  mysh  mysh.c  test.c  test.out
/home/2-class/sys32164120$ cat arg > txt
```

```
Breakpoint 1, redirection_cmd (tok=0xffffca70, flag=2) at mysh.c:59
59              for(i=0;i<MAX_BUF;i++)
(gdb) n
61                      if(!strcmp(tok[i],">") || !strcmp(tok[i],"<") || !strcmp(tok[i],">>"))
(gdb) p i
$1 = 0
(gdb) n
64                      partl[i] = tok[i];
(gdb) p partl[i]
$2 = 0x0
(gdb) n
59              for(i=0;i<MAX_BUF;i++)
(gdb) p partl[i]
$3 = 0xffffd2a0 "cat"
(gdb) n
61                      if(!strcmp(tok[i],">") || !strcmp(tok[i],"<") || !strcmp(tok[i],">>"))
(gdb) n
64                      partl[i] = tok[i];
(gdb) n
59              for(i=0;i<MAX_BUF;i++)
(gdb) p partl[i]
$4 = 0xffffd2a4 "arg"
(gdb) n
61                      if(!strcmp(tok[i],">") || !strcmp(tok[i],"<") || !strcmp(tok[i],">>"))
(gdb) n
66              partl[i] = '\0';
(gdb) n
68              for(i=i+1;tok[i] != NULL && i < MAX_BUF;i++)
(gdb) p token[i]
No symbol "token" in current context.
(gdb) p tok[i]
$5 = 0xffffd2a8 ">"
```

<,>,>> 문자 이전의 token들을 배열에 저장하는 첫 번째 for문은 문제없이 작동되었다. 이제 그 다음 <,>,>>이후의 token들을 저장하는 두 번째 반복문을 한 줄씩 실행해보았다.

```
(gdb) n
70                      part2[i] = tok[i];
(gdb) p tok[i]
$6 = 0xffffd2aa "txt"
(gdb) n
68              for(i=i+1;tok[i] != NULL && i < MAX_BUF;i++)
(gdb) p part2[i]
$7 = 0xffffd2aa "txt"
(gdb) n
72              part2[i] = '\0';
(gdb) n
75              if(flag == 2 || flag == 4)
(gdb) n
77                      if((fork_return = fork()) < 0)
(gdb) n
82                      else if(fork_return == 0)
(gdb) a+b*c
n
102                             wait();
(gdb) p i
$8 = 4
(gdb) set i = 0
Ambiguous set command "i = 0": inferior-tty, input-radix, interactive-mode.
(gdb) setQuit
(gdb) set variable i = 0
(gdb) p part2[i]
$9 = 0x0
```

의도한 대로 대상 파일인 "txt"가 part2[i]에 입력되었는데 계속 한 줄씩 실행해보니까 또 다시 terminal에 출력하고 redirection은 이루어지지 않았다. 혹시나해서 i값을 확인해 보았는데 원래 의도한 대로면 part2[0]에 저장된 txt가 open되면서 txt파일을 생성하고 그 곳에 출력 데이터를 쓰는 것이었는데 i의 값이 0이 아닌 4였다. part2[0]의 값을 확인해보니 NULL이었다. txt는 part2[3]에 저장되어 있었고 open()함수는 filename으로 NULL값을 받아서 open하지 못한 것이었다.

이 문제를 해결하기 위해서는 두 번째 for문이

'part2[0] = tok[i == 3]; -> tok[4] == NULL이므로 반복문 종료'

의 형식으로 진행되어야 했다. 그래서 두 번째 for문에 새로운 변수 j를 추가하여 코드를 수정했다.

```
for(i=i+1,j=0;tok[i] != NULL && i < MAX_BUF;i++,j++)
{
        part2[j] = tok[i];
}
part2[j] = '\0';
```

위와 같이 part2의 인덱스와 tok의 인덱스가 독립적으로 돌아가게 설정하니 드디어 redirection이 이상없이 구동되는 것을 확인할 수 있었다.

```
/home/2-class/sys32164120$ cat arg > txt
/home/2-class/sys32164120$ txt
txt : command not found!
/home/2-class/sys32164120$ cat txt
a+b*c
/home/2-class/sys32164120$ cat arg
a+b*c
```

2. 토의

대학교에 입학한 뒤 여러 가지 많은 과제들이 있었지만 과제를 하면서 요구사항이 아니지만 스스로가 '해보고 싶다'라고 느껴서 구현하고 공부해본 것은 처음이 아니었나 싶다. 이 이야기는 cd에 관한 이야기인데 처음 리눅스를 접하고 간단한 명령어들을 가지고 놀 때 가장 많이 사용했던 명령어 중 하나가 cd였다. 그래서 mysh를 구현할 때 cd는 내부 명령어이기에 따로 구현해주어야 하고, 그래서 만들 필요도 없다는 것이 아쉬웠다. 다행히도 시험 기간 중에도 틈틈이 작업하면서 기틀을 잡아 뒀기에 시험이 끝나고 빠르게 기본 요구사항들을 구현할 수 있었고, 내가 해보고 싶은 구현을 해 볼 수 있었다. cd를 구현 가능하다고 생각하게 된 것도 사실 재미있다. 시험에서 사용한 system call들을 추적하는 리눅스 명령어를 쓰라는 문제가 있었는데 그때 ptrace라는 system call을 쓴 것이다. 나중에 동기에게 설명을 들으니 ptrace는 system call이고 strace가 리눅스 명령어라는 것을 알 수 있었다. 그래서 "cd라는 디렉토리를 변경하는 명령어가 있다면 디렉토리를 변경하는 system call도 있지 않을까?"란 생각으로부터 chdir() system call을 찾을 수 있었고 생각보다 간단하게 구현할 수 있었다.

열심히 과제를 수행했고, 어느 정도는 의도한 대로 구현에 성공했으나 아쉬운 부분도 있다. 딱 하나를 꼽자면 background processing 관련이다. background processing을 확인하기 위해 sleep을 background로 돌리고 이후에 sleep이 종료되면 terminal에서 출력의 정렬이 이상하게 변했었다.

```
sys32164120@embedded:~$ ./mysh
/home/2-class/sys32164120$ sleep 10 &
/home/2-class/sys32164120$ ps
  PID TTY          TIME CMD
23680 pts/28   00:00:00 bash
26865 pts/28   00:00:00 mysh
26891 pts/28   00:00:00 sleep
26952 pts/28   00:00:00 ps
/home/2-class/sys32164120$ ls
3061.c  assignment   datastructure     mysh    test.c
arg     assignment3  examples.desktop  mysh.c  test.out
/home/2-class/sys32164120$ cat arg
a+b*c
/home/2-class/sys32164120$ ps
/home/2-class/sys32164120$   PID TTY          TIME CMD
23680 pts/28   00:00:00 bash
26865 pts/28   00:00:00 mysh
27310 pts/28   00:00:00 ps

/home/2-class/sys32164120$ ls
/home/2-class/sys32164120$ 3061.c       assignment   datastructure     mysh
est.c
arg     assignment3  examples.desktop  mysh.c  test.out
```

이 문제를 해결하고 싶어서 열심히 구글링 해보았지만 내 부족함인지 관련 정보들을 찾지 못했다. 부모 프로세스가 자식의 종료를 알지 못해서 이러는건가 싶어서 waitpid()를 사용해보기도 했지만 (waitpid함수는 세 번째 인자로 WNOHANG을 주면 자식 프로세스가 종료되지 않았더라도 기다리지 않고 0을 반환하는 함수였고, 그냥 wait()를 쓰지 않는 것과 결과적으로 차이도 없었다.) 이유를 찾지 못한 것이 아쉽다. 프로세스가 백그라운드에서 실행되는 것은 확인하였지만 이후에 터미널이 이상해지니 뒷맛이 찝찝한 느낌이랄까

교수님이 이 과제를 내주시면서 말씀하셨듯이 정말 많은 것들을 공부할 수 있는 기회가 되었고, 동기부여도 정말 많이 되었다.