운영체제

HW #1

Simple Shell(SiSH)

모바일시스템공학과

32161620

박산희

1. project introduction

키보드로 명령어를 입력하고 운영체제가 구동하도록 하는 프로그램인 shell을 간단히 만든다(simple shell, SiSH).

[SiSH 실행]	shell코드를 작성하여 컴파일 후 실행을 시키면 shell이 실행된다. 실행 창은 SiSH 와 현재 경로가 출력된다.
[SiSH 종료]	입력창에 quit을 입력하면 shell이 종료된다.
[SiSH 작동]	입력창에 받는 command는 string으로 입력받는다. 명령어를 입력받아서 실행시킨다(ex. ls, rm, mkdir, cp 등). program을 실행시킬 때에는 full path를 입력하지 않아도 된다. full path를 입력해도 실행이 가능하다. PATH를 받아서 program file을 찾아서 실행시킨다. file을 찾지 못하면, 실행하지 않는다. user-input program이 실행되면 shell이 도중에 실행되지 않는다. quit을 입력하기 전까지 계속해서 입력을 받는다.
[환경 변수]	getenv 함수를 사용하여 ($PATH, $HOME, $USER, $PWD) 등을 실행시킨다.

2. motivation

간단한 shell을 구현함으로써 명령어가 어떻게 처리되는지, process가 어떻게 수행되는지를 알게 되었다. os의 주요기능이 프로그램 및 사용자 보호, 입출력 관리, 주기억 장치 관리, CPU 관리, 유저 인터페이스임을 알고 이를 더 자세히 알기 위한 과정 중에서 shell 구현을 하였다.

program 이 메모리에 위치해서 실행이 되면 process가 된다는 program 과 process의 차이를 알았다. 또한 프로세스가 실행되기 위해서 파일을 찾고, 실행을 system call을 통해서 실행을 시키는 과정을 알았다.

system call(시스템 호출) 함수가 커널이 담당 하는 기능을 user가 접근하여 사용할 수 있게 한다. 프로세스가 하드웨어에 접근하는 것이다. system call function으로는 익숙한 file open, read, write, malloc, free 부터 해서 이번에 사용한 exec family 함수, fork 함수, wait 함수 등이 있고 이를 적절히 사용하는 방법을 배운다.

git hub를 통해서 많은 사람들과 같이 작업을 하는 방법을 익힌다. 다른 사람의 코드를 쉽게 받을 수 있고, 나의 코드를 쉽게 올릴 수 있다. 또한 같은 코드를 서로 고쳐서 병합할 수도 있다. branch로 나누어서 구분을 쉽게 할 수 있고, 다른 branch 로부터 file을 받아올 수도 있다. 그동안은 혼자서 코드를 작성했기 때문에 코드를 주고받을 때 메일을 통해서 했지만, 코드가 길어지고 앞으로 많은 프로젝트를 할 때 유용하게 쓰일 수 있는 git hub를 익히고, 사용하였다.

3. Important concept, specific and unique considerations for implementation

-운영체제

운영체제(Operating System)란 하드웨어란 응용 프로그램사이에서 인터페이스 역할을 하는 system software이다. 운영체제는 커널(Kernel) 과 서비스 프로그램으로 구성되어 있다. 커널이란 OS의 핵심이다. 커널은 부팅할 때 주기억장치에 로드되며, CPU 스케쥴링, interrupt 수행, 입출력기능 수행, 시스템 자원 관리를 한다. 운영체제는 프로그램 및 사용자 보호, 입출력 관리, 주기억 장치 관리, CPU 관리, 유저 인터페이스를 수행한다. OS의 종류로는 DOS, MS Windows, UNIX, Mac OS, LINUX 등이 있다.

-shell

shell은 키보드로 명령어를 입력하고 운영체제가 구동하도록 하는 프로그램이다. shell에는 환경변수를 입력해서 환경변수에 대한 정보를 얻을 수 있고, 명령어를 입력해서 내가 원하는 명령을 수행할 수 있다.

-프로세스

프로그램이란 하드디스크(보조 기억장치)에 저장되어있다. 프로그램의 명령어와 코드가 메모리에 가서 실행하게 되면 프로세스가 된다. 프로세스는 프로그램이 실행되고 있는 것이다. 프로세스의 정보는 PCB(Process Control Block) 에 저장이 된다. OS는 PCB에 저장되어 있는 정보를 가지고 process 스케쥴링을 한다. PCB에는 PID, 프로세스 상태, PC, 권한 등을 가지고 있다.

-makefile

makefile을 만들고 make명령어를 수행한다. makefile은 여러 개의 코드를 하나의 파일 형식으로 모아서 compile을 시킬 수 있다. 원하는 source code에 어떤 code를 같이 compile시킬지 정하기에 쉽다. 코드하나에 일일이 넣을 필요 없이 필요 없는 코드는 makefile에서 같이 compile하지 않겠다하면 해당 코드는 compile이 되지 않는다.

-fork()

프로세스를 생성하기 위해서 사용하는 함수이다. fork()를 사용하면 프로세스가 복제되어서 parent(기존 프로세스) process 와 child (복사 프로세스) process로 나누어진다. 그 둘은 다른 PID(Process ID)를 가지고 있다.
parent process는 child process가 실행을 하는 동안에는 wait()함수를 사용하여 child process가 종료 될 때까지 기다린다.

-wait()

child process가 종료될 때까지 기다린다. child process가 종료되었다면 함수는 바로 return 되면서 child 가 사용한 시스템 자원을 모두 해제한다. child process보다 parent process가 먼저 종료되면 child process가 좀비 process가 되기 때문에 그것을 막기 위해서 wait()를 사용한다.

-getenv()

#include <stdlib.h>
char *getenv(const char *varname);
이 함수는 varname에 대한 환경 변수의 list를 검색한다. 일치하는 이름의 환경변수가 있으면 값을 return 하고 없으면 null을 return한다.

-stat()

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
파일의 상태를 알아올 수 있다. file의 크기, 권한, 생성일시 등을 알 수 있다. file의 상태를 알아서 buf에 넣는다. 비슷한 함수로는 lstat 와 fstat함수가 있다. 세 함수 모두 파일 정보를 읽어온다.

-execve() / exe family

exe 뒤에 붙는 알파벳에 따라서 하는 기능들이 조금씩 다르다.

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg ,..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int execve (const char *filename, char *const argv [], char *const envp[]);
```

execve()는 현재 실행중인 프로그램을 종료하고 filename 프로그램을 처음부터 실행한다. filename에는 정확한 path를 지정해주어야 하고, argv인자에는 마지막에 NULL이 꼭 들어가야 한다. 마지막 인자인 envp에는 환경변수 문자열로 이 인자역시 마지막에 NULL이 들어가야 한다.

exec family는 현재 실행되는 프로세스에서 다른 프로세스의 일을 하게 하는 것이다. exec를 호출 할 때 원래의 process가 없어지기 때문에 fork() 와 같이 사용하면 child process에서 새로운 process를 loading 하기 때문에 child process가 작업을 완료하고 종료되면 shell의 대기 상태로 돌아가게 된다. 따라서 fork() 함수와 execve() 함수를 같이 써야 한다.

4. program structure

SiSH에서 다음과 같은 명령어가 실행된다.

: ls, rm, mkdir, whereis, cp 등

: 추가적으로 cddir() 와 getcwd()를 사용하여 cd(cd ..)를 구현하였다.

: ls –alh, ls –l 같은 option을 주는 명령어도 처리가 가능하게 구현하였다.

: echo 일 경우는 loop를 만들어서 처리하였다.

: full filename을 써도 실행이 가능하게 하였다.

SiSH에서 다음과 같은 환경변수에 대한 정보를 불러올 수 있다.

: $PATH, $USER, $HOME 등

: $TIME (헤더 파일 : <time.h>)을 추가로 구현하였다.

[program structure]

- getenv(PATH)를 사용하여 PATH를 미리 받아와 string array에 저장하였다.
  :PATH를 각각의 경로마다 이차원 배열을 사용하여 token화하여 저장하였다.
  (stat함수에서 (filename)에 사용된다.)

- 무한루프(while(1))를 실행하여 입력 값이 "quit"일 때까지 실행되도록 하였다.
- getcwd()를 사용하여 prompt를 받아와서 SiSH 다음에 출력하였다.
- fgets를 통해 command를 입력받는다.
  : fgets 함수는 '₩n' 까지 읽기 때문에 마지막 new line을 NULL('₩0')로 바꾸어 주었다.
- 입력 값(command)을 token시켜서 str[]에 저장하였다.
  : command가 full filename인 경우
    : command를 stat에 넣어서 존재하는지 검색하고, 존재하면 바로 fork()실행해서 execve() 실행
  : command의 명령어가 echo인 경우 echo에서의 loop를 실행한다.
    loop(echo뒤의 문자열(token strings)들이 모두 끝날 때 까지)
    : echo뒤에 나온 문자열이 환경변수인 경우 -> getenv실행
    : echo뒤에 나온 문자열이 환경변수가 아닌 경우 -> print out
  : command의 입력 값이 space인 경우 1을 return.
  : space가 아니면 0을 return.
- token함수의 return value가 0인 경우(space를 입력하지 않은 경우)

(입력 값의 맨 앞의 문자(char)가 '$'인 경우이면 환경변수, 아니면 명령어 실행으로 나눈다.)

  : '$' 일 경우 : getenv 실행, 결과 값 출력
  (getenv안에 넣는 string값에 $를 빼서 넣어줌(ex. $PATH->getenv(PATH)))
    : 'time'인 경우 따로 처리해줌 (time.h 사용)
  : '$' 아닐 경우 : stat() -> 미리 저장해둔 path를 가지고 명령어 검색
    : 검색했을 때 존재하면 fork() 실행해서 execve()실행
    : 없으면 실행 안함
    : 'cd' 경우는 따로 처리해줌(chdir 함수 사용)

5. problems and solutions

새로운 함수들이 많이 사용되어서 사용법을 익히는데 많은 시간이 들었다. 인자에 넣을 때 string 마지막에 NULL 이 있어야 하는지, 없어야 하는지에 따라 올바른 결과가 나오는 것에 어려움을 많이 느꼈다. stat에 들어가는 filename에는 끝에 NULL이 있으면 안 되지만, execve 에 들어가는 argv에는 NULL이 들어가야만 했다. 구글링을 많이 해서 함수에 대한 정보를 정확히 아는 것이 중요하다는 것을 알았다. return value나 type이 무엇인지, 인자의 type이 무엇이고 함수가 무슨 역할을 하는지를 읽을 줄 알아야 한다는 것을 알았다.

또한 execve 함수의 두 번째 인자에 argv값이 들어가야 했다. main함수를 실행하면 input값을 자동으로 계산하여 argc 값과 argv 값을 정해주는데, 이 shell은 한 번 실행하면 loop안에서 돌기 때문에 계속해서 argc 값과 argv 값이 설정이 되지 않았다. 그래서 같은 기능을 하는 str_num 과 argv_cpy를 만들어서 입력 값을 token해서 초기화 시켜줬다.
동적 할당을 하는 부분에서 오류가 많이 생겼다. 이차원 배열을 동적 할당 시켜줘야 하는데 malloc을 사용하여 한 번에 행과 열의 size를 곱해서 할당해 주었더니 오류가 생겼다.
이차원 배열에서는 동적 할당을 한 번에 해주는 것이 아니라 for loop을 두 번 사용해서 할당을 해주어야 한다는 것을 알았다.

git hub를 사용하는데 있어서 많은 어려움을 겪었다. git push에 file을 commit을 하고 push를 해서 file을 update를 해야 하는데 'non-fast-forward' 에러가 뜨면서 되지 않았다. git pull을 하거나 git merge origin/master를 해서 새로 업데이트 된 부분들을 받아왔어야 했는데 그것도 되지 않았다. branch를 새로 만들고 삭제하기를 몇 번 반복하다가 git stash를 써서 working directory를 지우고 나서야 해결이 되었다.

space를 입력했을 때 segmentation fault 가 자꾸 뜨고 SiSH이 종료가 되었다. 입력 값이 space여도 SiSH이 계속 실행이 되었어야 했다. 그래서 문제를 보니 command를 입력 받고 나서 command token을 할 때 오류가 발생되었다. token한 값의 처음에 NULL이 들어가고 다음이 실행이 되지 않았기 때문이다. 그래서 token한 값의 첫 번째 string이 NULL일 경우에는 loop의 처음으로 돌아가서 다시 command를 받도록 하였다.

string 관련 함수들을 사용하면서 strcat, strtok, strcpy 등의 함수들을 쓰게 되었다. 이때 segmentation이 많이 떴는데 그 이유가 인자로 들어가는 string1, string2 의 크기 문제였다. strcat의 경우 string1이 string2까지 합쳐서 들어가기 때문에 string1의 크기가 string2보다 훨씬 커야 했던 것을 몰랐었다. 또한 char* str 과 char str[size]가 같다고 생각했는데 char*로 선언을 해주고 바로 초기화를 시켜버리면 그 이후에 값이 더 늘어날 수가 없어서 오류가 많이 생겼었다. 이런 경우 동적 할당으로 크기를 넉넉히 잡아준 후, 초기화를 시켜주고 나서 strcat을 사용해야 한다는 것을 알았다.

echo에서 뒤에 환경변수가 나오면 출력이 안 되고 $PATH 같은 그냥 문자열로 읽어서 문자열을 출력하게 되었다. 그래서 환경변수를 getenv로 받아서 출력을 하게 하였다.
하지만 이 과정에서 환경변수와 문자열을 같이 입력할 경우 (ex. echo $PATH abcd $HOME) $PATH를 출력, abcd 출력, $HOME을 출력해야 하는데 2개 이상은 받을 수가

없었다. 그래서 echo를 입력했을 경우에는 그 안에서 무한루프를 돌려서 입력 값이 다 끝날 때까지 token 한 것이 환경변수를 읽으면 getenv를 실행하고, 문자열을 받으면 문자열을 fork에서 execve를 실행하도록 했다. 하지만 이런 식으로 하면 문자열이 환경변수의 이후에 들어가면 먼저 실행이 되고, 환경변수가 차례대로 실행되었다(ex. abcd $PATH $HOME). execve를 실행하면 현재 프로세스를 종료하고 자신의 프로세스를 먼저 실행시키기 때문에 $PATH가 실행되어도 execve가 먼저 출력이 되는 것 같았다. 그래서 execve를 실행하지 못하고 문자열을 읽게 되면 그냥 printf()로 문자열을 출력하였다.

6. build environment

실행 파일로 들어가기 위해서

sanhee16@assam:~/2018_os_hw1/

로 들어간다.

컴파일 하기 위해서

gcc myshell.c

를 한 후 ./a.out을 통해 실행을 시킨다.

github

branch : 32161620_sanhee

- myshell.c
- makefile_shell에 들어가면 makefile과 myshell.c을 나누어서 make를 통해 test를 만들어서 실행 시킬 수 있다.

7. screen capture

환경 변수

```
sanhee16@assam:~/2018_os_hw1/hw1$ gcc shell3.c
sanhee16@assam:~/2018_os_hw1/hw1$ ./a.out
SiSH /home/sanhee16/2018_os_hw1/hw1 $PATH
PATH=/home/sanhee16/bin:/home/sanhee16/.local/bin:/home/sanhee16/arm/arm-linux-gnueab
i-gcc/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/us
r/local/games:/snap/bin

SiSH /home/sanhee16/2018_os_hw1/hw1 $HOME
HOME=/home/sanhee16

SiSH /home/sanhee16/2018_os_hw1/hw1 $TIME
[TIME] 22:57:23

SiSH /home/sanhee16/2018_os_hw1/hw1 $PWD
PWD=/home/sanhee16/2018_os_hw1/hw1

SiSH /home/sanhee16/2018_os_hw1/hw1 $USER
USER=sanhee16
```

명령어

```
SiSH /home/sanhee16/2018_os_hw1/hw1 ls
a.out  aa.c  co.c  copy.c  myshell.c  shell2.c  shell3.c  test  test.c

SiSH /home/sanhee16/2018_os_hw1/hw1 ls -l
total 52
-rwxrwxr-x 1 sanhee16 sanhee16 14648 Sep 27 22:53 a.out
-rw-rw-r-- 1 sanhee16 sanhee16  4312 Sep 27 18:10 aa.c
-rw-rw-r-- 1 sanhee16 sanhee16  3323 Sep 27 21:51 co.c
-rw-rw-r-- 1 sanhee16 sanhee16  3274 Sep 26 01:40 copy.c
-rw-rw-r-- 1 sanhee16 sanhee16  3334 Sep 26 01:55 myshell.c
-rw-rw-r-- 1 sanhee16 sanhee16  3964 Sep 27 21:27 shell2.c
-rw-rw-r-- 1 sanhee16 sanhee16  3339 Sep 27 22:53 shell3.c
drwxrwxr-x 2 sanhee16 sanhee16  4096 Sep 27 19:36 test
-rw-rw-r-- 1 sanhee16 sanhee16  4030 Sep 27 19:23 test.c

SiSH /home/sanhee16/2018_os_hw1/hw1 ls -s
total 52
16 a.out   8 aa.c   4 co.c   4 copy.c   4 myshell.c   4 shell2.c   4 shell3.c   4 test   4 test.c

SiSH /home/sanhee16/2018_os_hw1/hw1 ls -alh
total 60K
drwxrwxr-x 3 sanhee16 sanhee16 4.0K Sep 27 22:53 .
drwxrwxr-x 4 sanhee16 sanhee16 4.0K Sep 26 18:07 ..
-rwxrwxr-x 1 sanhee16 sanhee16  15K Sep 27 22:53 a.out
-rw-rw-r-- 1 sanhee16 sanhee16 4.3K Sep 27 18:10 aa.c
-rw-rw-r-- 1 sanhee16 sanhee16 3.3K Sep 27 21:51 co.c
-rw-rw-r-- 1 sanhee16 sanhee16 3.2K Sep 26 01:40 copy.c
-rw-rw-r-- 1 sanhee16 sanhee16 3.3K Sep 26 01:55 myshell.c
-rw-rw-r-- 1 sanhee16 sanhee16 3.9K Sep 27 21:27 shell2.c
-rw-rw-r-- 1 sanhee16 sanhee16 3.3K Sep 27 22:53 shell3.c
drwxrwxr-x 2 sanhee16 sanhee16 4.0K Sep 27 19:36 test
-rw-rw-r-- 1 sanhee16 sanhee16 4.0K Sep 27 19:23 test.c
```

```
SiSH /home/sanhee16/2018_os_hw1/hw1 ls -slh
total 52K
 16K -rwxrwxr-x 1 sanhee16 sanhee16  15K Sep 27 22:53 a.out
8.0K -rw-rw-r-- 1 sanhee16 sanhee16 4.3K Sep 27 18:10 aa.c
4.0K -rw-rw-r-- 1 sanhee16 sanhee16 3.3K Sep 27 21:51 co.c
4.0K -rw-rw-r-- 1 sanhee16 sanhee16 3.2K Sep 26 01:40 copy.c
4.0K -rw-rw-r-- 1 sanhee16 sanhee16 3.3K Sep 26 01:55 myshell.c
4.0K -rw-rw-r-- 1 sanhee16 sanhee16 3.9K Sep 27 21:27 shell2.c
4.0K -rw-rw-r-- 1 sanhee16 sanhee16 3.3K Sep 27 22:53 shell3.c
4.0K drwxrwxr-x 2 sanhee16 sanhee16 4.0K Sep 27 19:36 test
4.0K -rw-rw-r-- 1 sanhee16 sanhee16 4.0K Sep 27 19:23 test.c

SiSH /home/sanhee16/2018_os_hw1/hw1 whereis rm
rm: /bin/rm /usr/share/man/man1/rm.1.gz

SiSH /home/sanhee16/2018_os_hw1/hw1 cp aa.c test3.c

SiSH /home/sanhee16/2018_os_hw1/hw1 ls
a.out  aa.c  co.c  copy.c  myshell.c  shell2.c  shell3.c  test  test.c  test3.c

SiSH /home/sanhee16/2018_os_hw1/hw1 rm test3.c

SiSH /home/sanhee16/2018_os_hw1/hw1 ls
a.out  aa.c  co.c  copy.c  myshell.c  shell2.c  shell3.c  test  test.c

SiSH /home/sanhee16/2018_os_hw1/hw1 cd test

SiSH /home/sanhee16/2018_os_hw1/hw1/test ls
a.out  malloc.c  token.c

SiSH /home/sanhee16/2018_os_hw1/hw1/test cd ..

SiSH /home/sanhee16/2018_os_hw1/hw1 quit
sanhee16@assam:~/2018_os_hw1/hw1$
```

```
SiSH /home/sanhee16/2018_os_hw1/hw1 /bin/ls
a.out  aa.c  co.c  copy.c  myshell.c  shell2.c  shell3.c  test  test.c
SiSH /home/sanhee16/2018_os_hw1/hw1 ls
a.out  aa.c  co.c  copy.c  myshell.c  shell2.c  shell3.c  test  test.c
SiSH /home/sanhee16/2018_os_hw1/hw1 whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
SiSH /home/sanhee16/2018_os_hw1/hw1 whereis mkdir
mkdir: /bin/mkdir /usr/share/man/man2/mkdir.2.gz /usr/share/man/man1/mkdir.1.gz
```

8. personal feelings

shell을 간단히 만드는 과정에서도 생각할 부분이 많았다. 함수별로 인자에 들어갈 type 이 char * 로 같아도 마지막이 NULL인 것이 있고 아닌 것이 있듯이 모든 함수를 알 필요는 없지만 한번 볼 때 자세히 봐야겠다고 생각했다. 또한 fgets처럼 마지막에 new line이 들어가는지 NULL이 들어가는지, gets랑 어떤 차이가 있는지도 알고 나서 적절히 사용해야 한다는 것을 알았다.

또한 git hub를 사용하는 방법을 익히는 과정이 어려우면서도 신기하였다. 많은 명령어들이 있고, commit하는 법, push같은 것들을 신중하게 해야 한다는 것을 알았다. 기록이 다 남기 때문에 내가 못했던 것들이 기록 남는 것이 싫으면서도, 미리 그렇게 하면서 배워나간다는 생각이 들었다. 리눅스를 처음 접할 때 명령어들이 어려웠는데, git hub도 같은 느낌을 받았다. 하지만 명령어들을 검색해 나가면서 알아가고, 사용하면서 더 리눅스가 편해졌던 것처럼, git 도 그럴 수 있을 것 같다.

처음에 코드를 짤 때 다 완전히 이해하지 못하는 상황에서 코드를 짜기 시작했다. 막연하게 시작해서 많은 오류들을 거치고 어느 정도 틀이 나오고 나니까 그동안 코드를 짜면서 공부해왔던 것들이 맞춰지면서 어떻게 코드를 짜야할지 큰 틀이 잡혔다. 그래서 어떻게 코드를 짜고, 함수를 만들고, 변수를 정할지 다시 차근차근 생각하고, 글로 정리한 다음 처음부터 다시 짰다. 생각보다 다시 짜는데 짧게 걸렸고, 코드를 어떻게 짤지 먼저 정리하고 작성하는 것이 중요하다고 느꼈다.

일부만 간단히 구현을 해본 것이지만, 실행되지 않는 명령어와 환경변수, 방향키로 전에 입력했던 command들을 불러오는 기능 같은 것들을 구현 하지 못한 것이 아쉬웠다. 하지만 full filename을 입력해도 작동이 되는 것과 space를 입력해도 오류가 뜨지 않는 것들을 구현해 보면서 재미있었고, strtok같은 잘 안 쓰던 함수들도 기존보다 더 잘 다루게 되었다.

다른 사람들의 질문들을 받아주고 도움을 주는 과정에서 많이 부족함을 느꼈다. 코드를 다 짰다고 해서 좋은 코드도 아니었고, 개념을 잘 아는 것도 아니었다. 도움을 주면 줄수록 더 많이 공부하게 되고 다양한 구현 방법을 생각하게 되었다. 또한 다른 사람들의 의견들을 들으면서 같이 공부하면 더 효과적인 방법이 나오게 된다는 것을 알았다.