SHELL

SiSH

HW1

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Fall / Operating Systems

Left freedays : 5

서론

SHELL 이란?

PC 를 비롯한 현대적 컴퓨터 시스템에서 Operating System(OS)는 중요한 역할을 수행한다. OS 는 application Software 가 hardware resource 를 사용하도록 도와주는 기반이 되어주는 layer 이며 software 이다. OS 는 virtualization, concurrency, 그리고 persistence 라는 아이디어를 바탕으로 자원을 관리 및 할당하고, 실행 모드를 구별하여 OS 와 application program 을 보호하고, 메모리, 파일 시스템, I/O 등을 관리하는 등의 다양한 기능을 수행 및 제공한다.

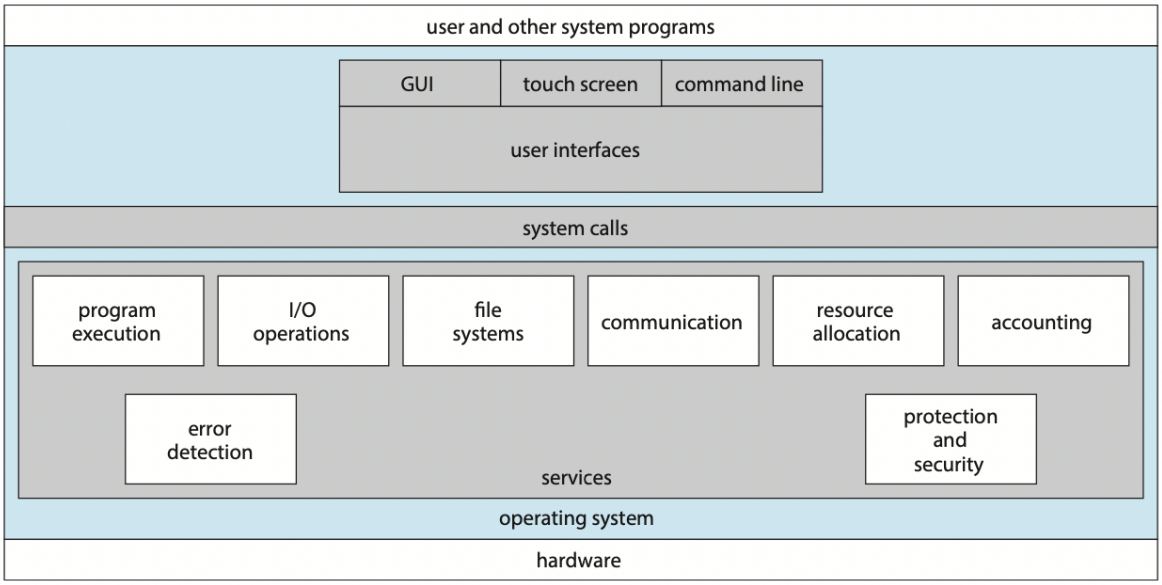


Figure 2.1 A view of operating system services.

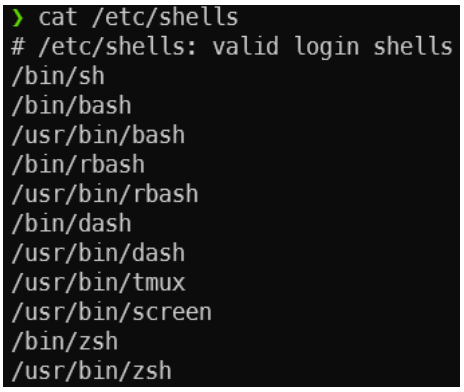
'Operating System Concepts'(Tenth Edition) - Abraham Silberschatz / Copyright © 2020 John Wiley & Sons, Inc.

이러한 OS 의 서비스 중, 일부는 사용자가 사용하기 쉬운 컴퓨터 환경을 구성하는 데에 그 목적이 있다. 이러한 목적을 띠는 대표적인 서비스는 바로 user interface 이다. User interface 는 사용자와 상호작용하며, OS 와 application software 를 사용하고 접근하기 위한 도구로써 작용한다. Shell 은 이러한 목적을 갖고 작동하는 user interface 프로그램이다. Shell 은 운영체제의 가장 바깥,

껍데기(shell)에서 사용자와 상호작용한다는 의미에서 비롯된 이름이다. GUI 등 다양한 방식의 User interface 에 따라 Shell 역시 다양한 방식으로 사용자와 만난다. Windows Shell 은 Graphic User Interface 로 사용자에게 shell 을 제공하고, Linux/UNIX 에서는 GUI 못지 않게 Command Line Interface(CLI)가 많이 사용된다. CLI 는 사용자가 한 줄 씩 명령어를 입력하면 이를 interpret 하여 명령을 실행하는 방식으로 동작한다.

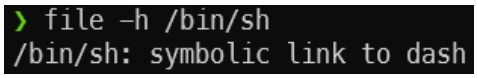
SHELLS in Linux

Linux / Unix 계열의 OS 에서는 다양한 shell 이 개발되어 사용되고 있다. Ubuntu 와 같은 일반적인 Linux 환경에서도 기본적으로 다양한 Shell 이 설치되어 있음을 확인할 수 있다.



(위의 사진에서 zsh 외의 shell 들은 별도의 설치 없이 초기 상태에서 설치된 shell 들이다)

위의 Bourne Shell(sh), BASH, DASH 중 sh 는 Bourne Shell 을 의미한다. Bourne Shell 은 Unix 버전 7 의 기본 shell 로 시작하여, 오랜 기간 많은 사용자에게 사랑받은 shell 이다. 그러나 현재는 Bourne Shell 의 후계인 BASH 로 대부분 대체되었다. 또한 심지어 현재 주로 사용되는 Ubuntu 배포판의 버전들에서 /etc/shells 의 /bin/sh 는 bourne shell 이 아니며 dash 의 symbolic link 이다. 이는 Ubuntu 6.10 이후로 적용된 변화이다. (https://wiki.ubuntu.com/DashAsBinSh)



BASH

BASH 는 현재 가장 많이 사용되는 대표적인 shell 중 하나이다. BASH 는 Bourne Again Shell 의 약자로, Bourne Shell 을 대체하기 위해 탄생하였다. Mac OS X 와 다수의 리눅스 배포판은 BASH 를 기본 로그인 셸로 채택하였으며 다양한 기능과 범용성으로 여전히 많은 사용자들이 BASH 를 사용하고 있다. 본 과제의 구현을 위해 BASH 를 통해 Shell(CLI)의 기본적인 작동 방식과 추가적으로 지원하는 기능 중 몇 가지를 알아보도록 하겠다.

Command Interpreter

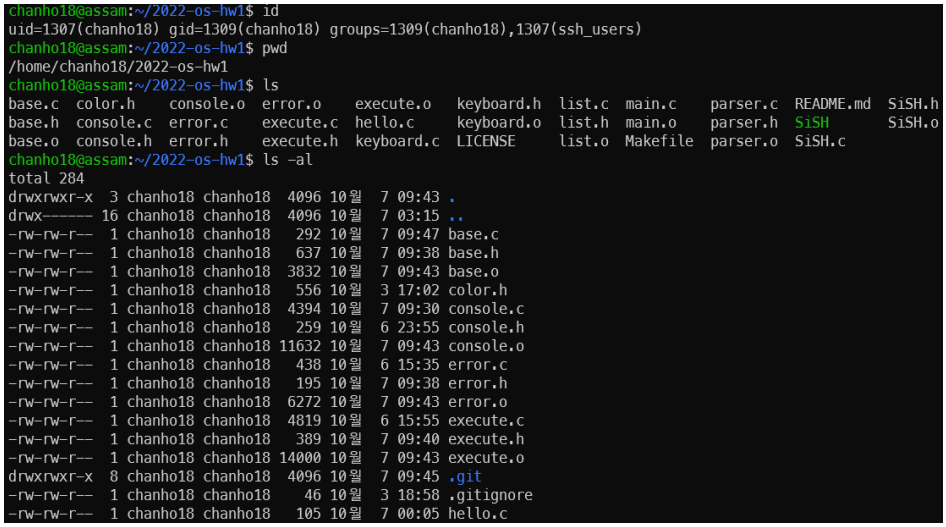
Shell 은 사용자로부터 한 줄을 입력 받아 parsing 하고, 명령어 또는 실행할 프로그램과 argument 를 분리하여, 명령어 또는 프로그램에 argument 를 전달하여 해당 작업을 수행하는 것이 기본 동작이다. 예컨대, 'ls -al'을 입력하면 ls 라는 프로그램에 -al 이라는 argument 를 전달하여 실행하고, 그동안 shell 은 inactive 한 상태로 대기한다. ls 의 실행이 종료되면, shell 은 active 한 상태로 돌아와, 사용자의 입력을 대기한다. 이러한 과정은 사용자가 quit 과 같은 명령을 입력하여 shell 을 명시적으로 종료하기 전까지 반복된다. 에러가 발생할 경우에도 shell 이 종료될 수 있으나, 이는 shell 의 옵션에 따라 종료되거나 다시 시작하도록 설정할 수 있는 부분이므로 기본 동작이 정해져 있지 않다.

이를 정리하면 shell 은

1. 사용자의 입력을 대기한다
2. 입력된 명령어와 argument 를 분리하고, 명령어에 따라 프로그램을 실행한다. 이때 프로그램에 argument 를 전달한다.
3. 프로그램의 실행 동안 대기한다.
4. 프로그램의 실행이 끝나면 1~3 의 과정을 반복한다. 단, 사용자가 종료 명령을 입력하면 shell 을 종료한다.

이와 같은 순서로 동작한다.

이러한 동작을 assam 의 BASH 에서 예시로 확인해보도록 하겠다.

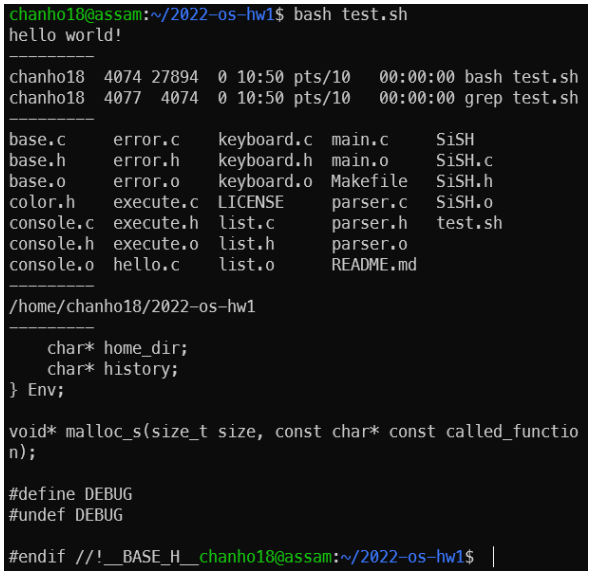


id, pwd, ls 등의 명령을 실행하였고, ls -al 을 입력했을 때 argument 를 전달하여 실행하였다.

Shell Script

BASH 등의 shell 은 shell script 를 입력으로 받아 실행할 수도 있다. Bash 의 binary 를 실행할 때 shell script 파일을 인수로 전달하면 된다. 간단한 예시를 아래에서 확인할 수 있다.

```
1 echo "hello world!"
2 echo "_________"
3 ps -ef | grep chanho18 | grep test.sh
4 echo "_________"
5 ls
6 echo "_________"
7 pwd
8 echo "_________"
9 tail base.h
```

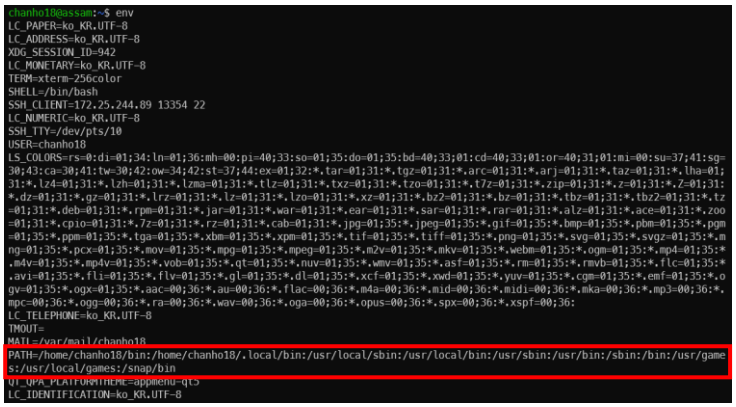


간단한 shell script(좌측)와 이를 실행한 결과(우측)이다.

Environment Variable (환경 변수)

Environment Variable 은 Bourne shell 부터 도입된 컨셉으로 특정 연산자 혹은 기호 뒤에 환경 변수명을 입력하면, 해당 환경 변수의 값을 출력할 수 있도록 한 Key-Value 의 집합이다. 이러한 환경 변수는 shell 의 동작 방식을 결정하는 동적인 값으로, BASH 의 경우에는 $ 기호를 사용하여 환경 변수의 값을 읽을 수 있다.

```
chanho18@assam:~$ echo $SHELL
/bin/bash
```



또한 env 명령을 통해 전체 환경변수를 출력할 수 있다

빨간 박스에서 확인할 수 있듯이, 환경 변수는 ls 와 같은 command 가 해당하는 binary 를 찾아 실행하도록 하는 PATH 라는 변수도 포함하고 있다.

Operator

앞서 환경변수에서 $를 이용하여 환경변수의 key 를 입력하면 value 를 출력할 수 있다고 했던 것처럼, BASH 를 비롯한 다양한 shell 은 operator 를 지원한다. Command line 의 입력을 parsing 한 결과에서, > >> < << || && | 등의 token 을 확인하면 해당 operator 에 정의된 기능을 수행한다. 예를 들어, | 는 pipe 로 앞의 명령을 수행한 결과를 그 다음 명령의 인풋으로 입력한다.

```
chanho18@assam:~/2022-os-hw1$ ls | grep base
base.c
base.h
base.o
chanho18@assam:~/2022-os-hw1$
```

Dot file

BASH 는 .bash 으로 시작하는 dot file 들을 가지고 있고, 이 파일들은 .bash_history, .bash_logout, .bashrc 등이 있다. .bash_history 는 command line 에서 위 아래 방향키로 command history 를 불러올 수 있도록 command history 를 저장하는 파일이다. 이외에 .bash_login, .bash_profile, .bashrc 등은 startup file 으로, BASH 가 시작될 때 특정 shell 스크립트를 실행하거나 BASH 의 옵션을 설정하기 위해 사용된다. .bash_logout 은 logout 시에 BASH 가 실행할 스크립트를 정의한다.

```
chanho18@assam:~$ ls -al | grep .bash
-rw-------   1 chanho18 chanho18    28339 10월  7 02:46 .bash_history
-rw-r--r--   1 chanho18 chanho18      220  7월  2  2021 .bash_logout
-rw-r--r--   1 chanho18 chanho18     3771  7월  2  2021 .bashrc
```

Auto Completion

BASH 를 포함한 여러 shell 에서 command 또는 argument 의 일부를 입력하고 tab 키를 누르면 그 일부를 포함하는 선택지들을 검색한다. 만약 그 선택지들이 공통적으로 갖는 문자열이 있다면, 해당 지점까지 자동완성을 수행한다. 선택지가 하나뿐이라면 전체 자동 완성을 수행한다. 선택지가 여럿인 경우 Tab 을 한 번 더 누르면 선택지들을 출력한다.

```
chanho18@assam:~$ cat .ba
```

```
chanho18@assam:~$ cat .bash
```

```
chanho18@assam:~$ cat .bash
.bash_history  .bash_logout   .bashrc
chanho18@assam:~$ cat .bash
```

Display Shell Information

Bash 에서는 User@Host:~{CurrentWorkingDirectory} 와 비슷한 꼴로 현재 환경의 정보를 보여준다.

Zsh 와 Oh-my-zsh 를 함께 사용하는 환경에서는 더 다양하고 화려한 화면을 구성할 수 있다.

Ignoring Signal & Enter

BASH 를 포함한 여러 shell 에서는 Ctrl+C, Ctrl+Z, Ctrl+₩ 등의 signal 을 무시한다. 더 정확히는 Ctrl+₩로 전달되는 SIGQUIT 와 Ctrl+Z 로 전달되는 SIGTSTP 는 무시되고, 상황에 따라 Ctrl+C 로 전달되는 SIGINT 는 실행 중인 loop 를 종료하도록 handling 되기도 한다. Ctrl+D 는 signal 이 아니라 EOF 로 동작한다. 따라서, command line 이 비어 있을 때, Ctrl+D 를 입력하면 경우에는 shell 이 종료된다. 또한 빈 줄에서 Enter 를 입력 시 새로운 빈 줄을 출력한다.

구현 목표

위의 기능들과 프로젝트의 요구사항을 바탕으로, 구현 목표를 정해보자

1. 기본적인 Shell 의 동작을 수행해야 한다
    A. 명령을 실행할 때까지 입력을 기다리다가, 명령을 실행하는 동안에는 명령 실행이 끝나기를 대기하고, 명령 실행이 끝나면 입력을 기다린다.
    B. 사용자가 종료 명령을 입력하면 종료한다.
    C. 명령 실행 시에 한 줄의 입력을 token 으로 parsing 하여 사용한다.
2. Shell Information (user, host, current directory)를 헤더로 출력한다.
3. Environment Variable 을 읽어서 PATH 경로 하의 프로그램을 명령어로 실행할 수 있다.
4. Signal 을 무시하도록 설정한다.
5. Tab 키를 눌러 자동완성을 수행한다.
6. Dot file 을 사용하여 Command History 를 저장하고, 위, 아래 방향키로 읽는다.
7. Shell Script 를 실행할 수 있다.
8. $, | operator 를 사용할 수 있도록 설정한다.
9. Makefile 을 구성하여 make 로 컴파일할 수 있도록 한다.

실행 방법

1. https://github.com/mobile-os-dku-cis-mse/2022-os-hw1 를 clone 받는다.
2. 32181928 branch 로 checkout 한다
3. make all 을 입력하여 build 한다.
4. ./SiSH 를 통해 interpreter mode 로 실행하거나, ./SiSH {script_file_name} 과 같은 식으로 script file mode 로 실행한다.

본론

소스파일 구성

먼저 repository 의 file tree 는 다음과 같다.

Repository

- ㄴ .gitignore
- ㄴ LICENSE
- ㄴ Makefile
- ㄴ README.md
- ㄴ SiSH.c
- ㄴ SiSH.h
- ㄴ base.c
- ㄴ base.h
- ㄴ color.h
- ㄴ console.c
- ㄴ console.h
- ㄴ error.c
- ㄴ error.h
- ㄴ execute.c
- ㄴ execute.h
- ㄴ hello.c
- ㄴ keyboard.c
- ㄴ keyboard.h
- ㄴ list.c
- ㄴ list.h
- ㄴ parser.c
- ㄴ parser.h
- ㄴ test.sh

각 파일에서 몇 가지를 골라 간단히 설명하면

.gitignore : compile 시 생기는 objects 와 executables 를 git 이 무시하도록 설정하였다.

SiSH.c / SiSH.h : SiSH 가 동작하는 메인 함수와 loop, file mode 함수가 존재한다.

base.c / base.h : 공통적으로 포함되는 헤더들과 매크로, env 구조체, malloc_s 를 포함한다.

color.h : ANSI 기반의 색깔 출력을 매크로로 정의한다.

console.c / console.h : 로고, 기본 포맷 등 shell 이 출력하는 기본 형식들을 정의한다.

error.c / error.h : 에러 출력 함수를 정의한다.

execute.c / execute.h : 명령 토큰들을 받아 실행하는 함수, cd, $, 명령 히스토리 저장을 정의한다.

keyboard.c / keyboard.h : terminal raw mode 를 모사하기 위해 getch()를 정의한다.

list.c / list.h : 파싱 과정 등에서 연결 리스트를 사용하기 위해 연결 리스트를 정의한다.

parser.c / parser.h : 명령 줄 입력을 인자로 받아 파싱 후 토큰들을 반환하는 함수를 정의한다.

구현 과정 및 결과

구현 과정 및 결과는 미리 정의한 각 요구 사항 및 기능을 기준으로 살펴보도록 하겠다.

1-A. A. 명령을 실행할 때까지 입력을 기다리다가, 명령을 실행하는 동안에는 명령 실행이 끝나기를 대기하고, 명령 실행이 끝나면 입력을 기다린다.

```
static int sish_loop(Env* env) {
    signal(SIGINT, handler);
    signal(SIGTSTP, handler);
    signal(SIGQUIT, handler);

    // 중략 : command history 불러와서 memory 에 로드

    while(1) {

    // 중략 : 버퍼 초기화, 변수 초기화, standard I/O 복사

        while (1) {
              // 중략 : 사용자 입력
        }

       // 중략 : 파싱

       // 중략 : | 와 $ 확인 및 처리

       // 중략 : quit 및 exit 확인, cd 명령어 처리


       // 중략 : command 실행 및 validity 확인

        // 중략 : standard I/O 복원

        // 중략 : buffer clear
    }
}
```

전체 코드의 로직을 주석으로 표시했을 때, shell 프로그램은 sish_loop 함수를 통해 사용자가 종료를 입력할 때까지 무한 반복하여 돌도록 설계되었다.

이때, 명령 실행을 하는 execute 함수 내에서, 명령이 실행되는 동안에 부모 프로세스는 자식 프로세스가 끝나기를 대기한다.

```
int execute(char* command, char** arr, const int pipe_index) {

// 중략

        pid_t pid;
        pid = fork();

#ifdef DEBUG
        fprintf(stdout, "%d", pid);
#endif
        switch (pid) {
            case -1:
                perror("ERROR: [Execute] Failed to fork");
                return -1;
            case 0:
                execvp(command, arr);
                *execute_checker = -1;
                fprintf(stderr, "ERROR: [Execute] Failed to execute '%s'",
command);
                perror(" ");
                exit(1);
                break;
            default:
                wait(0);
                int ret = *execute_checker;
                munmap(execute_checker, sizeof(*execute_checker));
                return ret;
        }
// 후략
```

execute 는 fork 로 자식 프로세스를 생성하고, 자식 프로세스는 execvp 로 command 를 실행하고, 부모 프로세스는 실행 완료를 기다린다. 이때 execute_checker 는 mmap 으로 메모리에 직접 할당하고 사용 후 해제하는 변수이다. 자식 프로세스가 fork 된 후 execvp 로 context 가 넘어가기 때문에, 자식 프로세스의 실행 결과를 확인하기 위해서 execute_checker 변수를 사용하였다.

1-B. 사용자가 종료 명령을 입력하면 종료한다.

```
static int sish_loop(Env* env) {

    // 중략

    while(1) {

        // 중략

        if (!strcmp(command, "quit") || !strcmp(command, "exit")) {
            //fprintf(stdout, "\n");
            exit(EXIT_SUCCESS);
        }


        // 중략
    }
}
```

sish_loop 함수에서 command 가 quit 또는 exit 이면 프로그램을 종료한다.

1-C. 명령 실행 시에 한 줄의 입력을 token 으로 parsing 하여 사용한다.

sish_loop 에서 parser 함수를 호출하는데, parser 함수는 아래와 같이 동작한다.

```
char** parser(const char* const input_string, int* size, const Env env) {
    char* delimiter = " \\\t";

    // 중략 : copied 로 input_string 복사, 연결 리스트 생성

    char* ptr = strtok_r(copied, delimiter, &save_ptr);

    while (ptr != NULL) {
        l_insert(list, ptr);
        ptr = strtok_r(NULL, delimiter, &save_ptr);
    }

    // 연결 리스트를 char** arr 로 복사, arr 마지막에 NULL 추가

    return arr;
}
```

Parser 는 파싱 후 토큰을 arr 로 반환한다. 마지막에 NULL 을 추가하여 execvp 실행을 준비한다.

이외에 1 번 항목 구현에서 중요했던 부분은 cd 명령이다. execute 함수는 binary 또는 PATH 아래의 프로그램을 command 로 실행할 수 있어야 하는데, BASH 등의 shell 에서 cd 명령은 별도의 프로그램이 존재하는 command 가 아니라, chdir 함수를 shell 프로그램에서 실행하는 방식으로 동작한다. 따라서, execute 함수에서 execvp 를 통해서 실행하는 방식으로 구현할 수 없고, sish_loop 함수 안에 별도의 로직을 구성하였다.

sish_loop 함수

```
        if (!strcmp(command, "cd")) {
            if (arr_size > 3) {
                // 중략 : cd {path} 꼴이 아닌 경우로 에러 처리
                continue;
            }
            char* path;
            if (arr_size == 2) { // cd 만 입력한 경우로 /home/{유저} 로 이동
                path = ""; // 빈 경로를 넣어 cd 함수에서 처리하도록 함
            } else {
                path = arr[1];
            }
            cd(path, env);
            continue;
        }
```

cd 함수

```
Validity cd(const char* const target_dir, Env* const env) {
    char* copied = malloc_s(sizeof(*copied) * env->path_max, __func__);

    if (!strcmp(target_dir, "")) { // 빈 경로의 경우
        strcpy(copied, "~"); // ~로 대체
    } else {
        strcpy(copied, target_dir);
    }
    if (!strcmp(copied, "~")) { // ~는 /home/{유저}로 대체
        strcpy(copied, env->home_dir);
    }

    if(chdir(copied) == -1) { // 경로 이동
       // 중략 : 에러처리
       return INVALID;
    }
    // 중략 : 경로 이동 후 env 의 현재 경로를 업데이트
    return VALID;
}
```

이를 통해 실제 명령을 실행한 결과를 캡쳐하였다.

```
○ chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ pwd
/home/chanho18/2022-os-hw1
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls
base.c  color.h    console.o  error.o    execute.o   keyboard.h  list.c  Makefile  parser.o   SiSH.c  test.sh
base.h  console.c  error.c    execute.c  hello.c     keyboard.o  list.h  parser.c  README.md  SiSH.h
base.o  console.h  error.h    execute.h  keyboard.c  LICENSE     list.o  parser.h  SiSH       SiSH.o
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ cd
● chanho18@assam.dankook.ac.kr:~$ pwd
/home/chanho18
● chanho18@assam.dankook.ac.kr:~$ cd 2022-os-hw1
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ pwd
/home/chanho18/2022-os-hw1
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$
```

```
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ cat hello.c
// to make test program binary

#include <stdio.h>

int main() {
    printf("HELLO");
    return 0;
}● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ echo hello
hello
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ id
uid=1307(chanho18) gid=1309(chanho18) groups=1309(chanho18),1307(ssh_users)
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ touch new_file
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls
base.c  color.h    console.o  error.o    execute.o   keyboard.h  list.c  Makefile  parser.h   SiSH    SiSH.o
base.h  console.c  error.c    execute.c  hello.c     keyboard.o  list.h  new_file  parser.o   SiSH.c  test.sh
base.o  console.h  error.h    execute.h  keyboard.c  LICENSE     list.o  parser.c  README.md  SiSH.h
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ rm new_file
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls
base.c  color.h    console.o  error.o    execute.o   keyboard.h  list.c  Makefile  parser.o   SiSH.c  test.sh
base.h  console.c  error.c    execute.c  hello.c     keyboard.o  list.h  parser.c  README.md  SiSH.h
base.o  console.h  error.h    execute.h  keyboard.c  LICENSE     list.o  parser.h  SiSH       SiSH.o
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ mkdir build
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls
base.c  build      console.h  error.h    execute.h  keyboard.c  LICENSE  list.o    parser.h  SiSH    SiSH.o
base.h  color.h    console.o  error.o    execute.o  keyboard.h  list.c   Makefile  parser.o  SiSH.c  test.sh
base.o  console.c  error.c    execute.c  hello.c    keyboard.o  list.h   parser.c  README.md  SiSH.h
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ rm -rf build
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls
base.c  color.h    console.o  error.o    execute.o   keyboard.h  list.c  Makefile  parser.o   SiSH.c  test.sh
base.h  console.c  error.c    execute.c  hello.c     keyboard.o  list.h  parser.c  README.md  SiSH.h
base.o  console.h  error.h    execute.h  keyboard.c  LICENSE     list.o  parser.h  SiSH       SiSH.o
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$
```

```
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls -al
total 288
drwxrwxr-x  3 chanho18 chanho18  4096 10월  7 22:33 .
drwx------ 16 chanho18 chanho18  4096 10월  7 20:38 ..
-rw-rw-r--  1 chanho18 chanho18   292 10월  7 13:32 base.c
-rw-rw-r--  1 chanho18 chanho18   637 10월  7 13:32 base.h
-rw-rw-r--  1 chanho18 chanho18  3832 10월  7 22:18 base.o
-rw-rw-r--  1 chanho18 chanho18   556 10월  7 13:32 color.h
-rw-rw-r--  1 chanho18 chanho18  4394 10월  7 13:32 console.c
-rw-rw-r--  1 chanho18 chanho18   259 10월  7 13:32 console.h
-rw-rw-r--  1 chanho18 chanho18 11632 10월  7 22:18 console.o
-rw-rw-r--  1 chanho18 chanho18   438 10월  7 13:32 error.c
-rw-rw-r--  1 chanho18 chanho18   195 10월  7 13:32 error.h
-rw-rw-r--  1 chanho18 chanho18  6272 10월  7 22:18 error.o
-rw-rw-r--  1 chanho18 chanho18  4819 10월  7 13:32 execute.c
-rw-rw-r--  1 chanho18 chanho18   389 10월  7 13:32 execute.h
-rw-rw-r--  1 chanho18 chanho18 14000 10월  7 22:18 execute.o
drwxrwxr-x  8 chanho18 chanho18  4096 10월  7 22:18 .git
-rw-rw-r--  1 chanho18 chanho18    46 10월  7 13:32 .gitignore
-rw-rw-r--  1 chanho18 chanho18   105 10월  7 13:32 hello.c
-rw-rw-r--  1 chanho18 chanho18   558 10월  7 13:32 keyboard.c
-rw-rw-r--  1 chanho18 chanho18   217 10월  7 13:32 keyboard.h
-rw-rw-r--  1 chanho18 chanho18  5416 10월  7 22:18 keyboard.o
-rw-rw-r--  1 chanho18 chanho18  1078 10월  7 13:32 LICENSE
-rw-rw-r--  1 chanho18 chanho18  1683 10월  7 13:32 list.c
-rw-rw-r--  1 chanho18 chanho18   452 10월  7 13:32 list.h
-rw-rw-r--  1 chanho18 chanho18  9000 10월  7 22:18 list.o
-rw-rw-r--  1 chanho18 chanho18   460 10월  7 20:38 Makefile
-rw-rw-r--  1 chanho18 chanho18   721 10월  7 13:32 parser.c
-rw-rw-r--  1 chanho18 chanho18   160 10월  7 13:32 parser.h
-rw-rw-r--  1 chanho18 chanho18  6008 10월  7 22:18 parser.o
-rw-rw-r--  1 chanho18 chanho18  4495 10월  7 13:32 README.md
-rwxrwxr-x  1 chanho18 chanho18 51720 10월  7 22:18 SiSH
-rw-rw-r--  1 chanho18 chanho18 14704 10월  7 20:39 SiSH.c
-rw-rw-r--  1 chanho18 chanho18   333 10월  7 13:32 SiSH.h
-rw-rw-r--  1 chanho18 chanho18 38616 10월  7 22:18 SiSH.o
-rw-rw-r--  1 chanho18 chanho18   152 10월  7 20:15 test.sh
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$
```

2. Shell Information (user, host, current directory)를 헤더로 출력한다.

Shell 의 현재 상태를 나타내는 정보는 print_line_format 함수에서 출력한다

```
void print_line_format(const Env env, const Validity valid) {
    print_valid_output(valid);
    fprintf(stdout, ANSI_BOLD_ON ANSI_COLOR_GREEN_LIGHT "%s@%s"
ANSI_COLOR_RESET ":" ANSI_BOLD_ON ANSI_COLOR_BLUE_LIGHT "%s" ANSI_COLOR_RESET
"$ " ANSI_COLOR_RESET, env.login_name, env.host_name, env.current_path);
}
```

미리 지정한 색깔에 따라 출력하도록 함수를 구성하였다.

출력 시에 아래처럼 미리 정의된 Validity 를 인자로 받는다.

```
#define Validity char
#define VALID 1
#define INVALID 0
#define INIT -1
```

Validity 에 따라, 포맷에 맞게 출력하기 전에 이전 명령 실행의 결과를 원으로 표시한다. 초기 상태 또는 이전 명령이 없다면 빈 원(◯)을 출력하고, 이전 실행이 valid 하다면 파란색, 이전 실행이 invalid 하다면 빨간색 꽉 찬 원(●)을 출력한다.

또한, shell 의 정보는 매 loop 마다 업데이트한다. 따라서, 경로 이동이나 다른 유저 로그인 시 다른 정보를 나타낸다.

3. Environment Variable 을 읽어서 PATH 경로 하의 프로그램을 명령어로 실행할 수 있다.

이는 별도의 PATH 확인이 필요하지 않고, execvp 로 command 를 실행할 경우 해결된다.

코드와 캡쳐는 앞선 내용과 동일하기 때문에 생략한다.

4. Signal 을 무시하도록 설정한다.

sish_loop 함수의 시작과 함께 Ctrl+C 의 SIGINT, Ctrl+Z 의 SIGTSTP, Ctrl+₩의 SIGQUIT 을 signal 함수로 handler 가 처리하도록 설정한다. handler 는 signal 에 맞는 문자를 출력한다.

```c
void handler(int sig) {
    switch (sig) {
        case SIGINT:
            fprintf(stdout, "^C\n");
            break;
        case SIGTSTP:
            fprintf(stdout, "^Z\n");
            break;
        case SIGQUIT:
            fprintf(stdout, "^\\\n");
            break;
        default:
            break;
    }
}
```

```
~/2022-os-hw1 / 32181928
./SiSH --nologo --nowelcome
charlieppark@DESKTOP-V3VRVE1:~/2022-os-hw1$ ^C
^Z
^\
ls
LICENSE    SiSH.h  color.h    error.h    hello.c     list.h    test.sh
Makefile   SiSH.o  console.c  error.o    keyboard.c  list.o
README.md  base.c  console.h  execute.c  keyboard.h  parser.c
SiSH       base.h  console.o  execute.h  keyboard.o  parser.h
SiSH.c     base.o  error.c    execute.o  list.c      parser.o
charlieppark@DESKTOP-V3VRVE1:~/2022-os-hw1$
```

5. Tab 키를 눌러 자동완성을 수행한다.

이 부분을 구현하기 위해 BASH 와 같은 shell 에서는 terminal raw mode 를 사용한다. 이는 tty 드라이버가 키보드가 입력되는 즉시 해당하는 문자를 넘기게 한다. 이는 fgets 와 같은 일반적인 입력 함수에서 한 줄을 통째로 입력 받는 cooked mode 와 다른 방식으로 동작한다. 따라서, 본 프로젝트에서는 keyboard.c 와 keyboard.h 에서 getch() 함수를 만들어 raw mode 를 모사하였다. Raw mode 를 직접 사용하는 것도 고려했으나, raw mode 를 모사하는 것이 구현 상의 난이도가 더 낮다고 판단하였다.

```
int getch() {
    int c;
    struct termios oldattr, newattr;

    tcgetattr(STDIN_FILENO, &oldattr);
    newattr = oldattr;
    newattr.c_lflag &= ~(ICANON | ECHO);
    newattr.c_cc[VMIN] = 1;
    newattr.c_cc[VTIME] = 0;
    tcsetattr(STDIN_FILENO, TCSANOW, &newattr);
    c = getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldattr);
    return c;
}
```

getch() 함수는 camp 프로젝트에서도 사용한 것과 동일한 것으로, Windows 에서의 getch 를 linux 에서 모사하기 위한 목적으로 만들어진 가장 유명한 버전을 그대로 사용하였다.

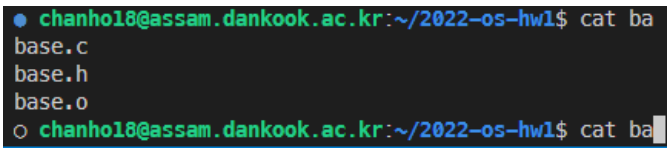
getch() 함수를 사용하여 raw mode 를 모사하기 시작하면, 기존의 fgets 등을 이용하여 입력을 받는 것보다 더 많은 처리를 필요로 한다. 일단, ascii 코드로 32 에서 126 까지의 문자만 화면에 출력할 수 있는 문자로 설정하였고, 이는 입력과 동시에 출력하도록 구현하였다. 또한, 엔터키는 빈 출을 출력하도록 구현하였다.

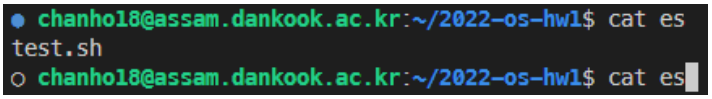
그리고 버퍼에 일부를 입력한 후, tab 키를 두 번 누르면 자동완성과 유사하게 뒤따라올 토큰을 추천하는 기능을 구현하였다. 'cat ba'를 입력 후 tab 을 두 번 누르면, ba 토큰을 미완성 토큰으로 생각하여 'ls | grep {미완성 토큰}' 명령을 실행한다. 그리고, 그 output 을 출력하고, 원래의 buffer 를 다시 출력하여 입력을 이어가도록 한다. 이 방식의 문제점은, grep {미완성 토큰} 형태를 사용하므로, 미완성 토큰의 문자열이 꼭 추천 문자열의 시작에 있지 않을 수도 있다.

```
            if ((before == 9) && (c == 9)) { // tab이 두 번 눌린 경우
                buffer[idx] = '\0';
                int arr_size;
                char** arr = parser("ls | grep blank", &arr_size, *env);
                pipe_index = 1;

                char* incomplete = malloc_s(sizeof(*incomplete) * 100,
__func__);

                strcpy(incomplete, buffer + blank_checker + 1);
                arr[3] = incomplete;

                empty_line();

                char* command = arr[0];

                execute(command, arr, pipe_index);

                dup2(stdin_copy, 0);
                dup2(stdout_copy, 1);
                close(stdin_copy);
                close(stdout_copy);

                print_line_format(*env, INIT);
                fprintf(stdout, "%s", buffer);
                pipe_index = -1;
                continue;
            }
```

이때 추후 다시 설명하겠지만, pipe 를 사용할 경우 stdin 과 stdout 을 따로 복사해두지 않으면 shell 에서 다른 프로그램을 실행 시 standard I/O 가 제대로 연결되지 않을 수 있다. 따라서 sish_loop 의 while loop 시작 시 복사해 둔 standard I/O 를 복원한다.

실행 결과는 아래와 같다. cat ba 를 입력할 경우 알맞게 추천해주지만, cat es 를 입력할 경우 es 로 시작하지 않는 내용도 추천하는 문제점이 있다.





6. Dot file 을 사용하여 Command History 를 저장하고, 위, 아래 방향키로 읽는다.

getch()를 사용하여 raw mode 를 모사한 덕분에, 위 아래 방향키를 사용하여 command history 를 읽고 실행할 수 있다. SiSH 는 sish_loop 의 초기화 단계에서 /home/{USER} 의 .SiSH_history 파일을 읽고 history_block 에 최대 500 개까지 load 한다. 만약 .SiSH_history 파일이 존재하지 않는다면 이를 생성하고, shell 을 재시작하기를 안내하며 종료한다.

```
    FILE* fp = fopen(env->history, "r");
    if (fp == NULL) {
        print_error(__func__, "Failed to open file : History not exists");
        FILE* fp = fopen(env->history, "w");
        if (fp == NULL) {
            print_error(__func__, "SERIOUS--Failed to open file twice");
            exit(1);
        }
        fclose(fp);
        fprintf(stdout, "CREATED HISTORY FILE. RUN SHELL AGAIN\n");
        exit(0);
    }
```

이후 방향키 입력과 함께 history 를 하나씩 불러 출력하고, 실행할 수 있게 buffer 에 채운다.

```
            if (before == '\33' && c == '[') {
                char arrow = getch();
                int buf_len;

                if ((arrow != 'A') && (arrow != 'B')) continue;

                if (strcmp(buffer, history)) {
                    buf_len = strlen(buffer);

                    erase_console(buf_len);
                    clear_buffer(buffer, sizeof(buffer));
                }

                switch (arrow) {
                    case 'A': // \33[A 위 방향키
                        if ((history_cursor < 500) && (history_cursor <
history_block->size - 1)) history_cursor++;
                        else continue;
```

```
                    break;
                case 'B': // \33[B 아래 방향키
                    if (history_cursor > 0) history_cursor--;
                    else {
                        history_len = strlen(history);
                        history = "";
                        clear_buffer(buffer, sizeof(buffer));
                        erase_console(history_len);
                        idx = 0;
                        continue;
                    }
                    break;
                default:
                    break;
            }

            history_len = strlen(history);
            history = l_get(history_block, history_block->size - 1 -
history_cursor);
            erase_console(history_len);
            fprintf(stdout, "%s", history);
            strcpy(buffer, history);
            idx = strlen(buffer);
            continue;
        }
```

History 를 출력하고, 방향키로 history 를 불러들이고 실행한 결과는 아래와 같다.

```
○ chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ cat ../.SiSH_history
ls
cat test.h
cat test.sh
ls
pwd
cat ../.SiSH_history
pwd
ls
cat ../.SiSH_history
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls
base.c     console.h  execute.c   keyboard.h  list.o    README.md  test.sh
base.h     console.o  execute.h   keyboard.o  Makefile  SiSH
base.o     error.c    execute.o   LICENSE     parser.c  SiSH.c
color.h    error.h    hello.c     list.c      parser.h  SiSH.h
console.c  error.o    keyboard.c  list.h      parser.o  SiSH.o
● chanho18@assam.dankook.ac.kr:~/2022-os-hw1$
```

7. Shell Script 를 실행할 수 있다.

./SiSH 를 실행할 때, shell script 파일을 인수로 추가하면 sish_file_mode 함수가 호출되어 해당 스크립트 파일을 실행한다. sish_file_mode 함수는 sish_loop 와 거의 동일한 로직을 가지지만, 입력을 받지 않고 shell script 를 한 줄 씩 읽어 입력으로 사용한다.

코드는 생략한다. 예시 shell script 와 실행 결과는 아래와 같다.

```
chanho18@assam:~/2022-os-hw1$ cat test.sh
echo hello world!
echo ----------
ps -ef | grep bash
echo ----------
ls
echo ----------
pwd
echo ----------
tail base.h
echo
echo ----------
echo $USER
echo
```

```
chanho18@assam:~/2022-os-hw1$ ./SiSH test.sh
hello world!
---------
chanho18  1834  1833  0 22:18 pts/13   00:00:00 -bash
junwoo18  5347  5113  0 20:30 pts/19   00:00:00 /bin/bash --init-file /home/junwoo1
8/.vscode-server/bin/64bbfbf67ada9953918d72e1df2f4d8e537d340e/out/vs/workbench/cont
rib/terminal/browser/media/shellIntegration-bash.sh
dabin17   5516  5515  0 23:14 pts/10   00:00:00 -bash
chanho18  6343 31909  0 23:26 pts/8    00:00:00 /bin/bash --init-file /home/chanho1
8/.vscode-server/bin/64bbfbf67ada9953918d72e1df2f4d8e537d340e/out/vs/workbench/cont
rib/terminal/browser/media/shellIntegration-bash.sh
chanho18 31779 31778  0 19:30 ?        00:00:00 bash
chanho18 32104 31909  0 19:30 pts/15   00:00:00 /bin/bash --init-file /home/chanho1
8/.vscode-server/bin/64bbfbf67ada9953918d72e1df2f4d8e537d340e/out/vs/workbench/cont
rib/terminal/browser/media/shellIntegration-bash.sh
---------
base.c     console.h  execute.c   keyboard.h  list.o    README.md  test.sh
base.h     console.o  execute.h   keyboard.o  Makefile  SiSH
base.o     error.c    execute.o   LICENSE     parser.c  SiSH.c
color.h    error.h    hello.c     list.c      parser.h  SiSH.h
console.c  error.o    keyboard.c  list.h      parser.o  SiSH.o
---------
/home/chanho18/2022-os-hw1
---------
    char* home_dir;
    char* history;
} Env;

void* malloc_s(size_t size, const char* const called_function);

#define DEBUG
#undef DEBUG

#endif //!__BASE_H__
---------
chanho18
```

8. $, | operator 를 사용할 수 있도록 설정한다.

$ 연산자는 getenv() 함수를 사용하여 환경 변수의 내용을 불러온다.

```
char* dollar(const char* const str) {
    if (str[0] == '$') {
        char* ret = getenv(str + 1);
        if (ret == NULL) {
            return "";
        } else {
            return ret;
        }
    }

    return NULL;
```

| 연산자는 조금 복잡한데, 만약에 | 연산자가 입력되었을 경우, | 앞의 프로그램을 실행하고 그 결과를 | 뒤의 프로그램으로 넘겨야 한다. 이를 구현하기 위해, file descriptor 인 int fd[2] 를 선언하고, 이를 pipe(fd)로 연결하였다. 이후 fd 를 standard input / output 로 복사하여 | 앞의 프로그램이 실행하는 출력을 | 뒤의 프로그램으로 전달하였다.

```
    }  else { // pipe 가 입력된 경우

        // 중략 : | 앞 뒤의 명령 분리

        int fd[2];

        if (pipe(fd) == -1) {
            perror("ERROR: [Execute-pipe] Failed to create pipe");
            exit(EXIT_FAILURE);
        }

        pid_t pid;
        pid = fork();

        switch (pid) {
            case -1:
                perror("ERROR: [Execute] Failed to fork");
                return -1;
            case 0:
                dup2(fd[1], 1);
                close(fd[0]);

                execvp(left_command, left_arr);
```

```
                *execute_checker = -1;

                fprintf(stderr, "ERROR: [Execute] Failed to execute '%s'",
command);
                perror(" ");
                exit(1);
                break;
            default:
                wait(0);
                break;
        }

        dup2(fd[0], 0);
        close(fd[1]);

        char line[1024];

        char* file_name = ".pipe_temp";

        FILE *fp;
        fp = fopen(file_name, "w");
        if (fp == NULL) {
            perror("ERROR: [Execute-fp] Failed to open file");
        }

        while (fgets(line, sizeof(line), stdin) != 0) {
            fprintf(fp, "%s", line);
        }

        fclose(fp);

        char** new_arr = malloc_s(sizeof(*new_arr) * (length + 1), __func__);

        for (int i = 0; i < length - 1; i++) {
            new_arr[i] = right_arr[i];
        }

        new_arr[length - 1] = file_name;

        new_arr[length] = NULL;

        pid_t npid;
        npid = fork();
```

```
        switch (npid) {
            case -1:
                perror("ERROR: [Execute] Failed to fork");
                return -1;
            case 0:
                execvp(right_command, new_arr);
                *execute_checker = -1;
                fprintf(stderr, "ERROR: [Execute] Failed to execute '%s'",
command);
                perror(" ");
                exit(1);
                break;
            default:
                wait(0);
                remove(file_name);
                free(new_arr);
                close(fd[0]);
                int ret = *execute_checker;
                munmap(execute_checker, sizeof(*execute_checker));
                return ret;
        }
    }
```

실행결과는 아래와 같다.

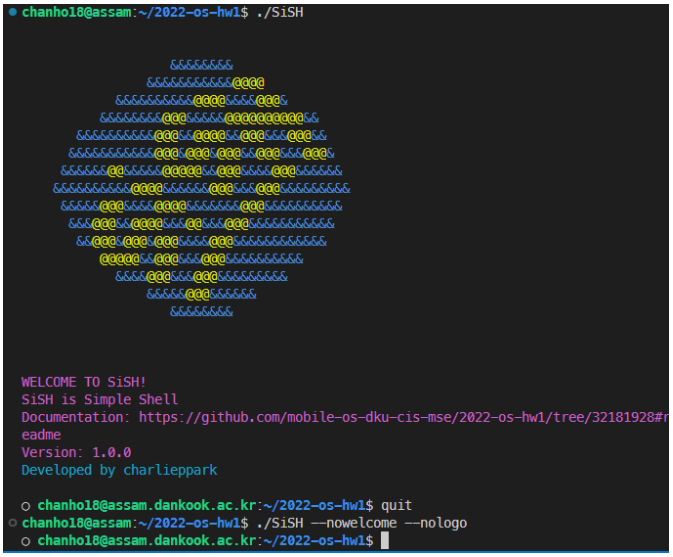
```
chanho18@assam:~/2022-os-hw1$ ./SiSH --nowelcome --nologo
chanho18@assam.dankook.ac.kr:~/2022-os-hw1$ ls | grep base
base.c
base.h
base.o
chanho18@assam.dankook.ac.kr:~/2022-os-hw1$
```

9. Makefile 을 구성하여 make 로 컴파일할 수 있도록 한다.

Makefile 은 아래와 같이 구성하였으며 별도의 설명은 생략한다.

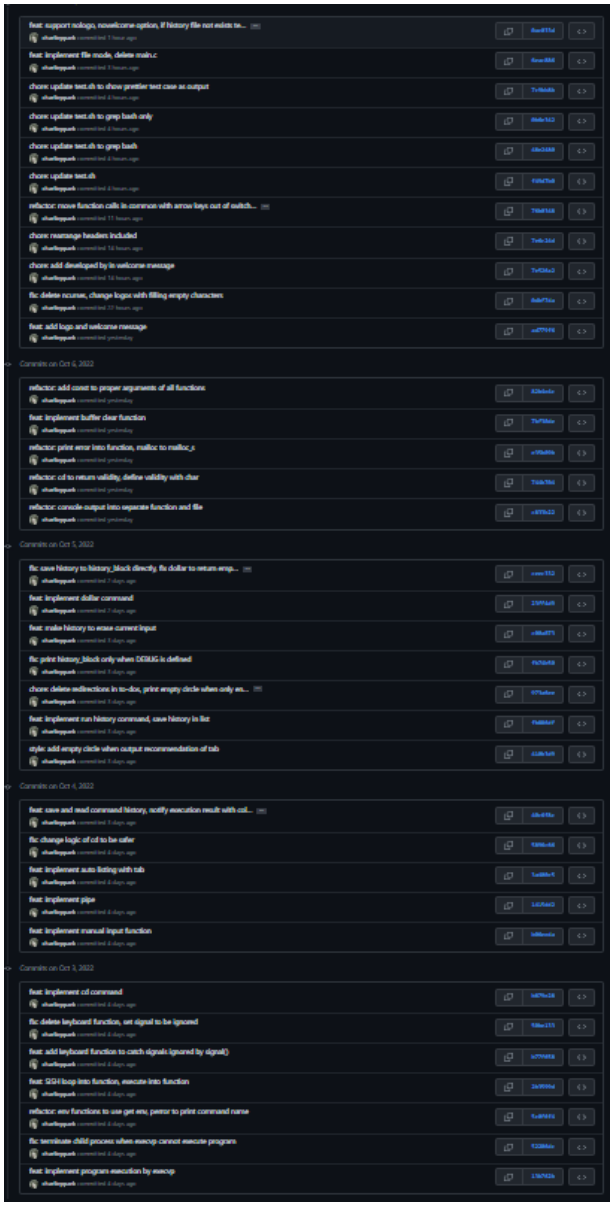
```
CC=gcc
CFLAGS=-g -Wall
OBJS=SiSH.o parser.o list.o execute.o keyboard.o console.o base.o error.o
TARGET=SiSH

all : $(TARGET)

$(TARGET): $(OBJS)
    $(CC) -o $@ $(OBJS)

clean :
    rm *.o SiSH

SiSH.o: color.h execute.h base.h SiSH.h SiSH.c
parser.o: parser.h base.h list.h parser.c
list.o: list.h list.c
execute.o : execute.h base.h parser.h execute.c
keyboard.o: keyboard.h keyboard.c
console.o: console.h console.c
base.o: base.h base.c
error.o: error.h error.c
```

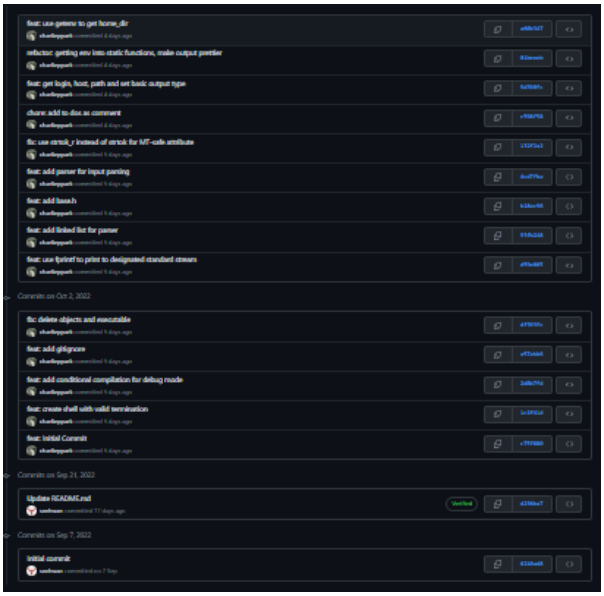
위의 구현에 추가로, SiSH 실행 시 인수를 받아 shell 을 설정할 수 있게 하였다. 설정할 수 있는 항목은 arg_max, user_name_max 등의 설정 변수이거나, 또는 --nologo 와 같이 shell 의 초기동작과 관련된 항목이다.



Commit log

이번 프로젝트를 수행하며 기록된 커밋 로그이다. 커밋은 정상적으로 실행이 가능한 코드가 최소 단위의 변화를 가진다고 생각했을 때 마다 수행하려고 노력했다.





결론

회고

본 프로젝트는 상당히 재미있었다. 기존에 아무 생각 없이 사용하던 shell 을 직접 구현하는 것은 많은 부분에서 까다로운 처리를 요했다. 이를 위해 fork, execvp, pipe, file descriptor 등을 깊게 배우고 사용하는 경험을 할 수 있었다.

참고자료

Operating Systems : Three easy pieces – Remzi H. Arpaci-Dusseau

Operating System Concepts – Abraham

OS 강의 자료

https://www.gnu.org/software/bash/manual

감사합니다