

# Simple Shell

모바일 시스템 공학과

32201345 나웅철

## 목차

- 1. 셸의 개념
- 2. 셸과 커널
- 3. 셸 vs 터미널 vs 커맨드라인 인터페이스
- 4. 소스코드가 프로세스가 되기까지
- 5. 리눅스와 파일
- 6. 명령어 실행
  - 명령어 입력 시 실행되는 과정
  - 셸 자체 내장 명령어와 외부 명령어
- 7. 셸 실행 실습

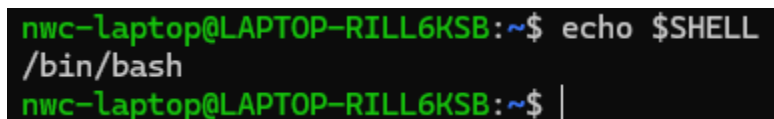
## Intro.

리눅스 운영체제에서 사용자는 커널을 직접 조작할 수 없기 때문에 사용자는 시스템 자원을 사용하거나 프로그램을 실행하기 위해 중개자 역할을 하는 소프트웨어인 셸을 통해 운영체제와 소통한다.

## 01. 셸의 기본 개념

### \_셸의 개념

셸(Shell)은 커맨드 라인 인터페이스(CLI)를 통해 사용자가 명령어를 입력하면, 이를 해석하고 시스템 명령어를 실행하는 역할을 한다.



```
nwc~laptop@LAPTOP-RILL6KSB:~$ echo $SHELL
/bin/bash
nwc~laptop@LAPTOP-RILL6KSB:~$ |
```

[그림 1] bash 셸을 사용하고 있는 것을 확인 할 수 있다.

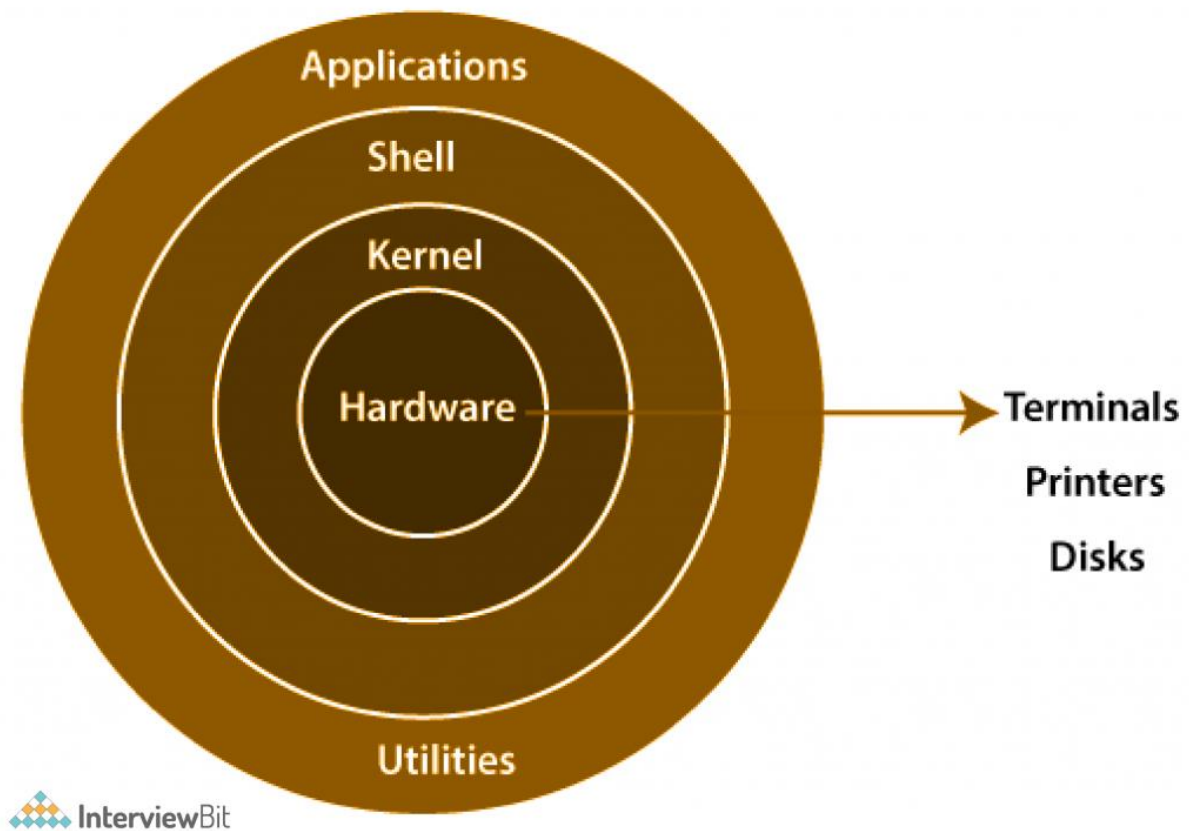
리눅스에 로그인 하면 제일 처음 나오는 셸을 로그인 셸이라고 한다. 꼭 bash 셸만 사용해야 하는 것은 아니며 리눅스의 경우 Bash셸 말고도 Zsh, Ksh, sh, csh, tsch 등이 있으며 바뀌어서 사용할 수 있다. 셸은 주로 다음과 같은 역할을 수행한다.

- 명령어 해석 : 사용자가 입력한 명령어를 해석하여 적절한 프로그램을 실행한다.
- 프로세스 제어 : 명령어 실행 시 생성되는 프로세스를 관리한다. (fg/bg, exit 등등)
- 입출력 제어 : 명령어 실행 시 입출력 리다이렉션, 파이프라인 처리 등의 기능을 제공한다.

### \_셸의 역할

셸은 운영체제의 여러 핵심적인 역할을 돕는다. 셸을 통해 사용자는 프로그램을 실행하거나 파일을 관리할 수 있으며, 네트워크를 제어하고 시스템 설정을 변경할 수 있다. 또한 셸은 사용자와 운영체제 간의 인터페이스를 제공하는 중요한 소프트웨어로, 사용자의 명령을 해석하고 시스템 호출(System Call)을 통해 커널과 상호작용한다.

## 02. 셸과 커널



[그림 2] Linux Architecture

[그림 2]는 리눅스 아키텍처를 간단하게 표현한 그림이다. 각 항목을 계층으로 표현하며 주요 3계층은 다음과 같다.

1. 하드웨어 계층 : CPU, 메인 메모리, 디스크, 드라이브를 포함하며 주변 디바이스 (키보드, 모니터 등등..)까지 모두 포함한다.
2. 커널 계층 : 아래에서 기술
3. 사용자 영역 계층(userland): 셸 같은 운영체제 구성요소, ps나 ssh같은 유틸리티를 비롯한 대부분의 앱들이 실행되는 곳. [그림 2]에서는 shell계층과 Applications 계층을 합한 영역을 의미한다.

셸과 커널은 운영체제의 중요한 두 구성요소로, 각각 사용자와 시스템 간의 인터페이스 역할을 담당하며, 시스템 자원 관리의 책임을 맡는다. 이 두 요소는 서로 다른 기능을 수행하지만, 서로 밀접하게 상호작용하면서 운영체제가 원활히 동작하도록 한다.

## \_커널(Kernel)의 역할

커널은 운영체제의 핵심 부분으로, **하드웨어와 소프트웨어 사이에서** 자원을 관리하고 **시스템의 모든 주요 기능을 제어한다**. 커널은 프로세스 관리, 메모리 관리, 장치 관리, 파일 시스템 관리와 같은 여러가지 주요 기능을 담당한다. 시스템 하드웨어 자원을 직접 제어하고 관리한다는 점이 핵심이다. 기존의 지식에서는 c언어의 포인터의 경우 메모리 주소를 이용하여 직접 하드웨어를 다룬다고 알고 있었지만 하드웨어라고 생각했던 메모리는 사실 가상 메모리였음을 확인하였다. 즉, 사용자는 가상메모리를 다루고 커널이 실제 하드웨어를 다룬다는 것이다.

### 커널의 주요 기능

- 프로세스 관리 : 커널은 시스템에서 실행되는 모든 프로세스를 제어하고, 프로세스 간의 자원 경쟁을 조정하며, 프로세스 스케줄링을 담당한다.
- 메모리 관리 : 커널은 프로그램들이 사용할 수 있는 가상 메모리를 관리하고, 실제 물리 메모리와의 매핑을 담당한다.
- 장치 관리 : 커널은 다양한 하드웨어 장치들과 상호작용하고, 이를 소프트웨어에서 사용할 수 있도록 제어한다.
- 파일 시스템 관리 : 커널은 파일의 저장, 검색, 읽기, 쓰기를 관리하며, 파일 시스템의 일관성과 보안을 유지한다.

## \_셸과 커널의 상호작용

커널은 사용자가 직접 접근할 수 없기 때문에, 셸과 같은 소프트웨어를 이용해서 간접적으로 접근하게 된다. 셸과 커널은 서로 다른 계층에서 동작하며, 상호 보완적인 역할을 수행한다. 사용자가 셸을 통해 명령을 입력하면, 셸은 이를 해석하여 시스템 호출을 통해 커널에 전달한다.

사용자 → 셸 → 커널 → 하드웨어

커널은 그 명령을 실행하고, 실행 결과를 다시 셸을 통해 사용자에게 반환한다.

사용자 ← 셸 ← 커널 ← 하드웨어

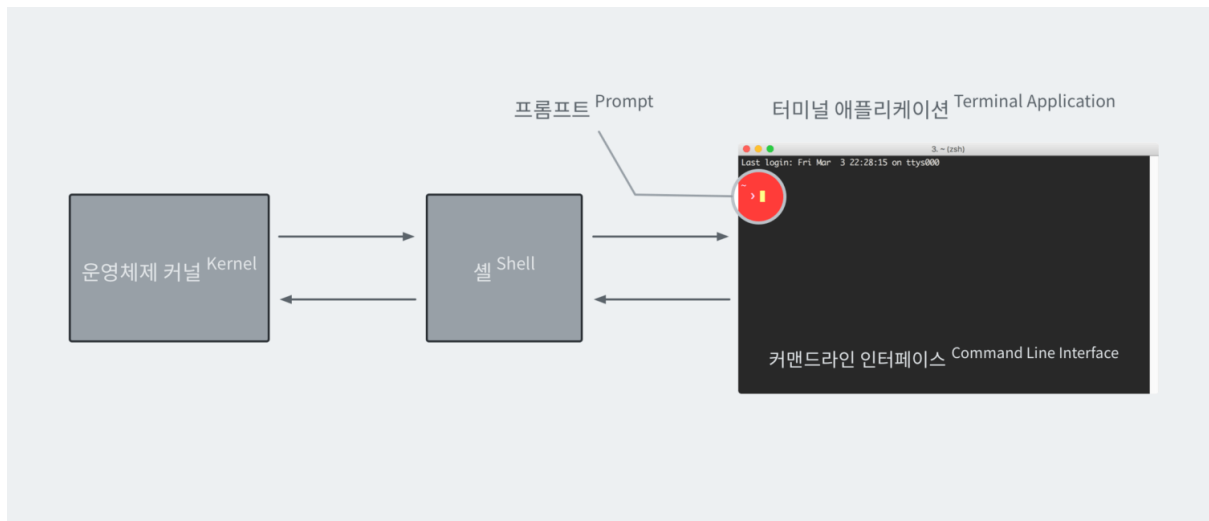
### 03. 셸 vs 터미널 vs CLI

필자는 처음에 '사용자의 입력을 받아들인다' 측면에서 셸과 터미널, 그리고 CLI를 별 생각 없이 동일시하여 바라보았다. 하지만 과제를 수행하면서 셸과 터미널 그리고 CLI는 완전히 다른 소프트웨어라는 것을 알게 되었다. 중요도를 따졌을 때 상대적으로 그리 높지는 않을 수 있지만 그래도 짚고 넘어가고자 한다.



```
nwc-laptop@LAPTOP-RILL6K5:~$ echo $SHELL
/bin/bash
nwc-laptop@LAPTOP-RILL6K5:~$
```

[그림 3] 터미널



[그림 4] 셸, 터미널, CLI 개념도

#### \_터미널

터미널은 셸을 실행할 수 있는 환경을 제공한다. 초기 컴퓨터 시스템에서 터미널은 모니터와 키보드가 있는 물리적인 장치였으나 현대에는 터미널 에뮬레이터를 통해 구현된다.

터미널은 사용자가 셸을 실행할 수 있는 창을 제공하며, 사용자는 터미널을 통해 커맨드라인 인터페이스에서 셸 명령어를 입력하고 결과를 출력 받는다. 터미널은 셸에서 발생하는 모든 작업을 시각적으로 확인할 수 있는 도구인 것이다.

## \_커맨드라인 인터페이스(CLI)

필자는 이전에 K8s CLI인 kubectl을 이용하는 과정에서 상당히 애를 먹고 도대체 CLI라는 것이 무엇이길래 나를 이렇게 괴롭히는 걸까 라는 생각을 한적이 있었다. 또 한 예로 아나콘다 CLI를 사용하면서 명령어 치는 것은 똑같은데 왜 사용해야 할까 라는 의문을 가진 적이 있었다. 이제 그 개념과 역할에 대해 작성하고자 한다.

커맨드라인 인터페이스(CLI)는 사용자가 명령어를 텍스트로 입력하여 시스템을 제어하는 방식이다. CLI는 그래픽 사용자 인터페이스(GUI)와는 다르게 텍스트 기반으로 동작하며, GUI보다 적은 자원을 사용하면서도 효율적이고 빠르게 시스템 작업을 수행할 수 있다. 또한, CLI는 복잡한 작업을 스크립트로 자동화할 수 있다. 리눅스 CLI의 경우에는 운영체제를 대상으로 하며, 쿠버네티스 CLI인 kubectl은 쿠버네티스 프로그램을, 아나콘다 CLI는 아나콘다 프로그램을 대상으로 하여 제어한다.

결론적으로 셸과 터미널, 그리고 CLI는 서로 밀접하게 연관되어 있다. 터미널은 셸을 실행할 수 있는 환경을 제공하고, 셸은 사용자가 입력한 명령어를 해석하여 커널에 전달하며, CLI는 이러한 명령어를 입력하는 방식인 것이다. 즉 사용자가 터미널을 열고 CLI 명령어를 입력하면, 셸이 이를 해석하고 실행하며, 결과는 다시 터미널을 통해 출력된다.

## 04. 소스코드가 프로세스가 되기까지

운영체제의 핵심 인터페이스인 셸은 프로세스를 생성하고 제어하는 중요한 역할을 한다. 셸을 통해 사용자가 명령어를 입력하면, 그 명령어를 실행하기 위해 새로운 프로세스가 생성된다. 이 과정에서 셸과 프로세스가 상호작용하게 된다. 그렇다면 프로세스란 무엇일까?

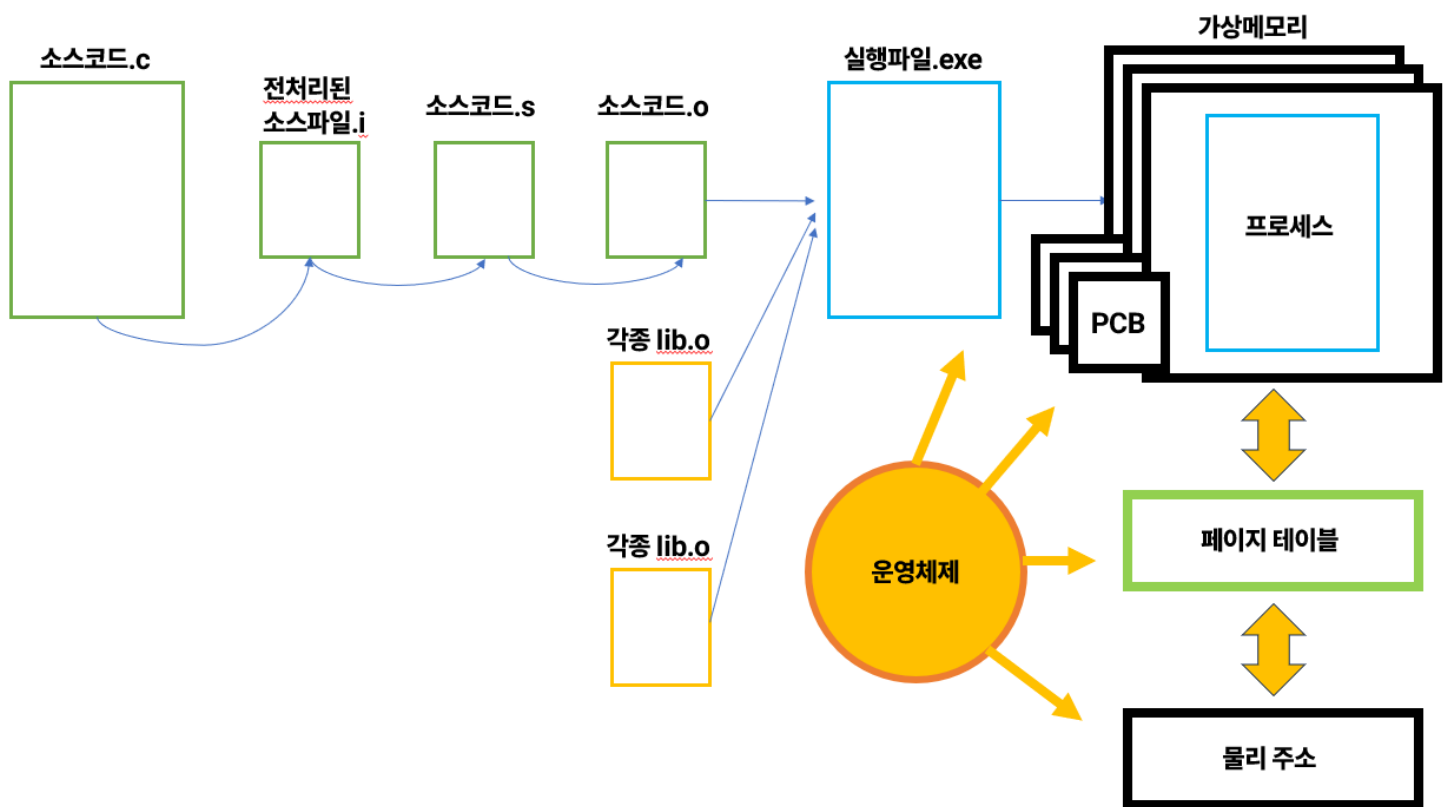
\_소스코드가 프로세스가 되는 과정

C언어로 된 소스코드가 프로세스가 되는 과정에 대해 간략히 표현하면 다음과 같다.

1. 소스코드 작성
2. 컴파일(Compile) – 전처리 → 컴파일 → 어셈블의 과정을 거쳐 기계어로 된 오브젝트 파일이 생성된다.
3. 링크(Linking) – 오브젝트 파일과 라이브러리를 결합하여 실행 가능한 바이너리 파일이 생성된다. 여기서 실행 가능한 바이너리 파일이란, CPU가 직접 실행할 수 있는 기계어 명령어를 포함하여 운영체제가 로드할 수 있는 구조로 되어있다는 의미이다.
4. 로딩>Loading) – 운영체제가 **실행 파일을 가상 메모리 공간에 할당**한다.
  - 프로그램의 코드 섹션, 데이터 섹션, 스택, 힙 등 메모리 공간이 가상 메모리에 배정된다.
  - 이때, 실행 파일의 기계어 코드가 메모리에 적재되고, 프로세스 제어 블록이 생성된다.
5. 실행>Execution) – 프로세스 생성 및 실행
  - 운영체제는 실행 파일을 기반으로 프로세스를 생성한다.
  - 프로세스는 고유한 프로세스 ID(PID)를 가지게 된다.
  - 독립된 가상 메모리 공간에서 프로그램의 main함수가 호출되며 프로세스의 실행이 시작된다.
6. 프로세스 실행 후 종료
  - 프로세스는 main()함수의 종료와 함께 return 0을 호출하며 종료된다.

- 프로세스가 종료되면 운영체제는 이 프로세스가 **사용했던 자원**을 반환한다.

여기서 중요한 내용은 운영체제는 실행파일(실행가능한 바이너리 파일)을 가상 메모리에 로드하여 프로세스를 만들고, 프로세스가 종료되면 해당 프로세스가 사용했던 자원을 반환한다는 점이다. 사용했던 자원이라 함은 실행파일을 로드한 가상 메모리에 해당하는 자원이라고 할 수 있다.



[그림 5] 소스코드가 프로세스가 되기까지의 과정 // 직접 제작

## 05. 리눅스와 파일

리눅스 운영체제의 핵심 철학 중 하나는 바로 “모든 것은 파일이다”라는 개념이다. 이 철학은 리눅스 시스템을 이해하는 데 중요한 개념으로, 리눅스에서는 다양한 하드웨어 장치, 네트워크, 시스템 자원 등 모든 것이 파일로 표현되며, 이러한 파일을 통해 시스템과 상호작용할 수 있다. 심지어는 리눅스 커널도 파일이고 시스템 설정도 파일에 기록된다. 필자가 가장 인상깊게 여긴 부분은 “명령어 또한 파일이다”라는 점이였다.



## 06. 명령어의 실행

셸에서 명령어를 입력했을 때 리눅스 내부에서 명령어가 실행되는 과정은 다음과 같다.

1. 사용자가 키보드로 문자열을 터미널의 커맨드라인 인터페이스에 입력한다.
2. 키보드로 입력한 문자열을 받아 셸에게 전달된다.
3. 문자열에 해당하는 명령어와 일치하는 파일을 PATH 환경 변수에 지정된 디렉토리에서 찾는다.
4. 발견한 명령어 실행 파일을 실행한다.
5. 실행파일은 가상 메모리에 로드되고, 프로세스가 생성된다.
6. 생성된 프로세스는 커널에 의해 관리되며 실행된다.

이 때 중요한 개념 두가지가 있다. 바로 fork()함수와 exec계열의 함수이다.

1. Fork() 함수 : 프로세스 복제
  - 리눅스는 멀티태스킹 운영체제이기 때문에 여러 프로세스를 동시에 실행할 수 있다. 새로운 프로세스를 생성하기 위해 fork()함수가 사용된다.

```

int main(int argc, char *argv[])
{
    pid_t pid;
    int i;

    for (i = 0 ; i < 10 ; i++) {
        pid = fork();
        if (pid == -1) {
            perror("fork error");
            return 0;
        }
        else if (pid == 0) {
            // child
            printf("child process with pid %d (i: %d) \n", getpid(), i);
            exit (0);
        } else {
            // parent
            wait(0);
        }
    }
    return 0;
}

```

[그림 6] : fork함수 코드

- fork()함수는 현재 실행 중인 프로세스(부모 프로세스)의 복사본을 만들어 새로운 프로세스(자식 프로세스)를 생성한다.
- 자식 프로세스는 부모 프로세스의 거의 모든 것을 복제하지만, 고유한 PID를 가지게 된다. 이때 자식 프로세스는 부모와 동일한 코드를 실행하게 되지만, 실행할 명령을 변경하기 위해 다음 단계에서 exec계열(execve(), execvp())의 함수가 사용된다.

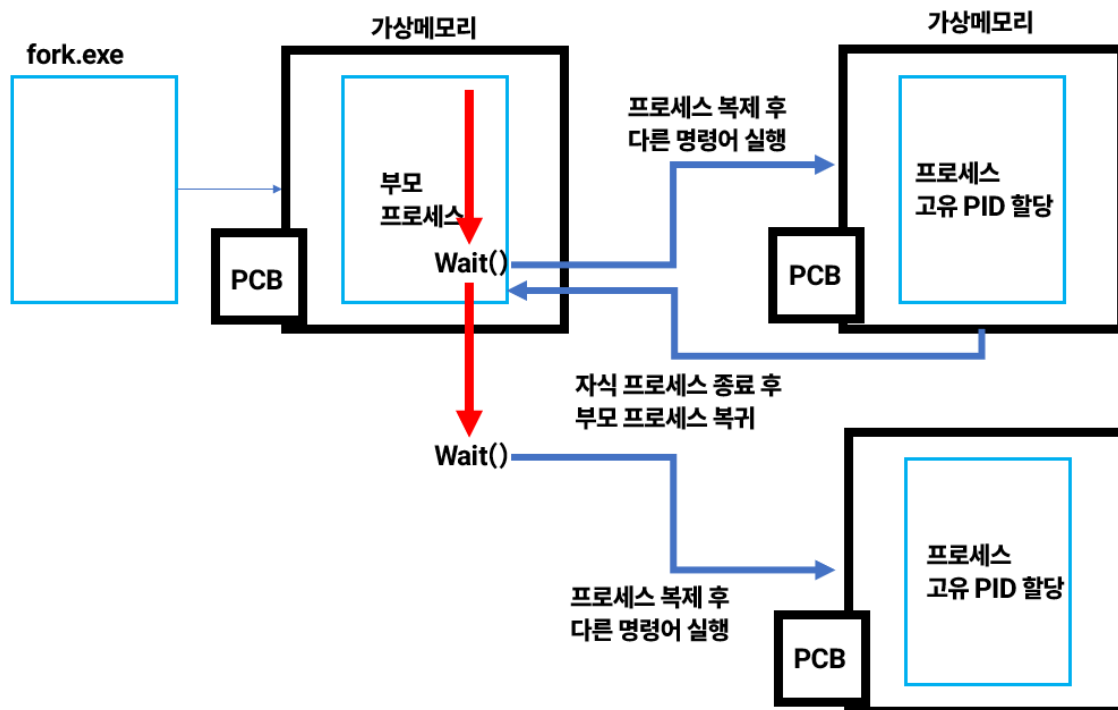
## 2. exec() 계열 함수 : 명령어 실행

- fork() 함수로 새로운 프로세스가 생성된 후, 자식 프로세스는 exec 계열의 함수를 사용하여 실제로 사용자가 입력한 명령을 실행한다.
- execvp()함수와 execve()함수

두 함수 모두 기존 프로세스를 새로운 프로그램으로 교체하며, 교체 후에는 이전 프로세스의 코드는 정지 상태가 된다. 두 함수 모두 실행파일을 실행 하지만, 인수 전달 방식과 환경 변수를 처리하는 방식에서 차이가 있다.

1. `execvp()` 함수의 경우 Execute program with vector and path 라는 뜻으로 파일 경로를 직접 지정하지 않아도 PATH에서 해당 프로그램을 찾아 실행한다.
  - 다음과 같이 선언한다.
  - `int execvp(const char *file, char *const argv[]);`
  - file에는 실행할 명령어가, argv[]에는 프로그램이 들어간다.
  
2. `execve()` 함수의 경우 Execute program with vector and environment 라는 뜻으로 파일 경로를 PATH에서 환경변수를 검색하지 않기 때문에 직접 입력해야 하는 함수이다.
  - 다음과 같이 선언한다.
  - `int execve(const char *pathname, char *const argv[], char *const envp[]);`
  - pathname에는 실행할 파일 즉 명령어의 절대 경로 또는 상대경로이다.
  - argv[] 에는 프로그램 이름이 들어간다
  - envp[] 에는 새로운 프로그램이 사용할 환경 변수 목록이 들어간다.

`execve`함수의 경우 파일의 경로를 명시해주어야 하므로 현재 프로세스에서 특정 환경 변수를 가져오는 명령어인 `getenv`명령어와 같이 사용할 수 있다.



[그림 7] `fork()`, `wait()`, 부모프로세스 → 자식프로세스 // 직접 제작

그림 8

```
nwc-laptop : /home/nwc-laptop/dev/os-assignment [23:29:14] : SiSH >>> getenv HOME PATH USER
HOME=/home/nwc-laptop
/home/nwc-laptop
***-----***
PATH=/home/nwc-laptop/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/bin/remote-cli
/home/nwc-laptop/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/bin/remote-cli
***-----***
USER=nwc-laptop
nwc-laptop
***-----***
nwc-laptop : /home/nwc-laptop/dev/os-assignment [23:29:47] : SiSH >>> getenv PATH
PATH=/home/nwc-laptop/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/bin/remote-cli
/home/nwc-laptop/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/bin/remote-cli
***-----***
```

[그림 9] : `getenv`처리 함수의 사용

```

416 // getenv 명령어 처리 함수
417 void handle_getenv_command(char *args[], int arg_count) {
418     int i, j;
419     char *env, *str;
420     char *tok[100], *saveptr;
421
422     // 인자가 없는 경우 사용법 출력
423     if (arg_count == 1) {
424         printf("usage: getenv env_vars ... \n");
425         return;
426     }
427
428     // 각 환경 변수를 처리한다
429     for (i = 1; i < arg_count; i++) {
430         env = getenv(args[i]); // 환경 변수를 가져온다
431         if (env == NULL) {
432             printf("%s is not set\n", args[i]);
433             continue;
434         }
435
436         printf("%s=%s\n", args[i], env); // 환경 변수를 출력한다
437
438         // 경로들을 출력한다 ( ':' 으로 구분됨 )
439         for (j = 0, str = env; ; str = NULL, j++) {
440             tok[j] = strtok_r(str, ":", &saveptr); // ':'로 구분된 토큰 처리한다
441             if (tok[j] == NULL) break;
442             printf("\t%s\n", tok[j]); // 각 경로를 출력한다.
443         }
444
445         printf("***-----***\n");
446     }
447 }

```

[그림 10] getenv 처리 함수

## 07. 셸 자체 내장 명령어와 외부 명령어

리눅스 셸에서는 사용자가 입력한 명령어를 처리하는 방법에 따라 내장 명령어와 외부 명령어로 나뉘며, 각 명령어는 처리 방식과 실행 속도, 사용 용도에서 차이가 있다.

셸 내장 명령어는 셸 자체에 포함된 명령어로, 셸이 직접 처리하고 실행한다. 이러한 명령어들은 별도의 실행 파일 없이 셸의 일부로 구현되어 있어, 실행 시 **시스템 콜 없이 빠르게 처리될 수 있다**. 내장 명령어는 주로 시스템 환경을 제어하거나 셸의 동작을 관리하는 데 사용된다. 따라서 셸을 직접 구현하는 이번 과제의 경우 내장 명령어는 [06. 명령어의 실행]에서의 3번 단계인 'PATH에서 명령어를 찾는다' 과정 대신 셸 자체에서

해결하도록 구현해야 한다.

```
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which find
/usr/bin/find
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which cd
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which jobs
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which fg
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which bg
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which history
```

[그림 11] 명령어 파일의 경로가 출력되는 제일 위의 find 명령어와는 달리 나머지 명령어들은 셸 자체에서 처리되기 때문에 경로가 반환되어 출력되지 않는다.

```
if (strcmp(args[0], "jobs") == 0) {
    list_jobs();
    return;
}

if (strcmp(args[0], "fg") == 0) {
    if (args[1] != NULL) {
        bring_to_foreground(atoi(args[1]));
    }
    return;
}

if (strcmp(args[0], "bg") == 0) {
    if (args[1] != NULL) {
        send_to_background(atoi(args[1]));
    }
    return;
}

if (strcmp(args[0], "history") == 0) {
```

[그림 12] SISH의 코드 : 입력된 문자열을 받아 내장 명령어일 경우 자체적으로 처리하는 코드

셸 외부 명령어는 셸에 내장되어 있지 않은 명령어로, 시스템의 파일 경로에 있는 실행 파일을 호출하여 실행한다. 셸은 외부 명령어를 실행할 때 프로세스를 생성하여 시스템 호출을 통해 해당 명령어를 실행하게 된다. 이 명령어들은 보통 /bin, /usr/bin 등의 경로에 위치한 실행 파일로 존재하며, 셸은 사용자가 입력한 명령어를 찾기 위해 PATH 환경 변수를 참조한다. 내장 명령어는 현재 진행중인 프로세스에서 그대로 처리되지만, 외부 명령어의 경우 자식 프로세스를 생성하여 실행한다는 특징이 있다.

```

nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which ls
/usr/bin/ls
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which echo
/usr/bin/echo
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ which cat
/usr/bin/cat

```

[그림 13] 외부 명령어의 경우 각각 명령어 실행파일의 경로를 반환하여 출력하는 모습

```

if (strcmp(args[0], "history") == 0) {
    for (int i = 0; i < history_count; i++) {
        printf("%d: %s\n", i + 1, command_history[i]);
    }
    return;
}

pid_t pid = fork();
if (pid < 0) {
    perror("fork failed");
} else if (pid == 0) {
    // Child process
    handle_redirection(args, &arg_count);

    char *path = find_command_in_path(args[0]);
    if (path != NULL) {
        execve(path, args, NULL);
        perror("execve failed");
        free(path);
    } else if (args[0][0] == '.' && args[0][1] == '/') {
        // Execute file in current directory
        execve(args[0], args, NULL);
        perror("execve failed");
    }
}

```

[그림 14] SSh의 코드 : 위쪽의 history 명령어까지는 자체적으로 처리하고 외부명령어인 경우 fork()함수를 이용해 자식 프로세스를 생성하여 실행한다.

## 08. 셸 실행 실습

### 1. Makefile 만들기

요구사항 : 라이브러리 필요 → `sudo apt-get install libreadline-dev`

```
makefile
1  CC = gcc
2  CFLAGS = -Wall -g
3  LDFLAGS = -lreadline
4  SRC = x.c
5  OBJ = x.o
6  TARGET = xyz
7
8
9  all: $(TARGET)
10
11  $(TARGET): $(OBJ)
12      $(CC) $(CFLAGS) -o $(TARGET) $(OBJ) $(LDFLAGS)
13
14  $(OBJ): $(SRC)
15      $(CC) $(CFLAGS) -c $(SRC) -o $(OBJ)
16
17  clean:
18      rm -f *.o $(TARGET)
19
```

```
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ make
gcc -Wall -g -o x x.o -lreadline
```

```
x
x.c
x.o
```

### 2. 셸 시작과 종료

```
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$ ./x
SiSH >>> echo hello
hello
SiSH >>> quit
nwc-laptop@LAPTOP-RILL6KSB:~/dev/os-assignment$
```

### 3. SiSH 작동



## \_기본 작동

```
SiSH >>> getenv PATH
PATH=/home/nwc-laptop/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc27a
li:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
sr/lib/wsl/lib:/mnt/c/Python312/Scripts:/mnt/c/Python312:/mnt/c/Program F
62.9-hotspot/bin:/mnt/c/Program Files/Java/openjdk-20.0.1_windows-x64_bin/j
c/WINDOWS/system32:/mnt/c/WINDOWS:/mnt/c/WINDOWS/System32/Wbem:/mnt/c/WINDO
PowerShell/v1.0:/mnt/c/WINDOWS/System32/OpenSSH:/mnt/c/Program Files/Inte
Program Files/Common Files/Intel/WirelessCommon:/mnt/c/Program Files (x86).
gram Files/Git/cmd:/mnt/c/Program Files (x86)/Windows Kits/10/Windows Perfo
t/c/Program Files/PuTTY:/mnt/c/Program Files/nodejs:/mnt/c/ProgramData/ch
/Users/woong/AppData/Local/Pub/Cache/bin:/mnt/c/MinGW/bin:/mnt/c/Users/woon
Users/woong/anaconda3/Library/bin:/mnt/c/Users/woong/anaconda3/Library:/mnt
onda3/Scripts:/mnt/c/Windows/Microsoft.NET/Framework64/v4.0.30319:/mnt/c/Us
/flutter_windows_3.13.3-stable/flutter/bin:/mnt/c/Users/woong/AppData/Local
thon311/Scripts:/mnt/c/Users/woong/AppData/Local/Programs/Python/Python311
stem32/config/systemprofile/AppData/Local/Microsoft/WindowsApps:/mnt/c/User
al/Programs/Microsoft VS Code/bin:/mnt/c/Users/woong/AppData/Local/afreeca.
Downloads/flutter_windows_3.13.3-stable/flutter/bin:/mnt/c/Program Files/Ja
indows-x64_bin/jdk-20.0.1/bin:/mnt/c/Users/woong/AppData/Roaming/npm:/mnt/
ser/32201345/AppDat/Local/Pub/Cache/bin:/mnt/d/woong/dev/Lens/resources/cli
ev/IntelliJ IDEA 2024.2.1/bin:/mnt/d/woong/dev/CLion 2024.2.1/bin:/mnt/d/wo
24.2.1/bin:/snap/bin
    /home/nwc-laptop/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc
e-cli
    /usr/local/sbin
    /usr/local/bin
    /usr/sbin
    /usr/bin
    /sbin
    /bin
    /usr/games
    /usr/local/games
    /usr/lib/wsl/lib
    /mnt/c/Python312/Scripts/
    /mnt/c/Python312/
```

[그림 15] PATH 가져오기

```
SiSH >>> sleep 10
```

```
□
```

```
SiSH >>> sleep 10
```

```
SiSH >>> □
```

[그림 16] 프로그램 실행 중 셸이 비활성화 되었다가 10초가 지난 후 다시 활성화되었다

```
nwc-laptop : /home/nwc-laptop/dev/os-assignment [20:31:58] : SiSH >>> ls
a.out  cla3    claude.c    new      output.txt  shellPipe.c  x.c
cla2   cla3.c  my_program  original pipe      test.c
cla2.c claude  my_program.c output    shell.c     x
nwc-laptop : /home/nwc-laptop/dev/os-assignment [20:32:00] : SiSH >>> history
1: ls
2: ./a.out
3: ls
4: history
```

[그림 17] 사용자 정의 프롬프트 사용

```
void display_prompt() {
    char cwd[BUFFER_SIZE];
    char *user = getenv("USER");
    char *pwd = getcwd(cwd, sizeof(cwd));
    time_t t = time(NULL);
    struct tm *tm_info = localtime(&t);
    char time_buf[9];

    strftime(time_buf, 9, "%H:%M:%S", tm_info);

    if (user != NULL && pwd != NULL) {
        printf("%s : %s [%s] : SiSH >>> ", user, pwd, time_buf);
    } else {
        printf("SiSH >>> ");
    }

    fflush(stdout);
}
```

[그림 18] 사용자 정의 프롬프트 함수

```
SiSH >>> ./my_program woongcheol 32201345 mobilesyst
Program name: ./my_program
Arguments passed to the program:
Argument 1: woongcheol
Argument 2: 32201345
Argument 3: mobilesyst
SiSH >>> []
```

[그림 19] 매개변수 입력이 있는 프로그램 실행

## \_각종 기능구현

### 1. 기본 명령어 실행

- 사용자 입력을 받아 시스템 명령어 실행
- PATH 환경변수를 이용한 명령어 검색

### 2. 내장 명령어

- cd : 디렉토리 변경
- getenv : 환경 변수 출력
- jobs : 백그라운드 작업 목록 출력
- fg : 작업을 포그라운드로 가져오기
- bg : 작업을 백그라운드로 보내기
- history : 명령어 히스토리 출력

### 3. 파이프라인

- 여러 명령어를 파이프로 연결하여 실행

### 4. 입출력 리다이렉션

- 출력 리다이렉션 : >, >>
- 입력 리다이렉션 : <
- 에러 출력 리다이렉션 : 2>, 2>&1

### 5. 백그라운드 실행

- 명령어 뒤에 &를 붙여 백그라운드에서 실행

### 6. 명령어 자동 완성

- readline 라이브러리를 이용한 탭 자동 완성 기능

## 7. 명령어 히스토리

- 이전에 실행한 명령어 저장 및 조회

### \_SISH 기능 실행

```
SiSH >>> ls -l
total 368
-rwxr-xr-x 1 nwc-laptop nwc-laptop 17256 Sep 28 17:53 a.out
-rwxr-xr-x 1 nwc-laptop nwc-laptop 22144 Oct  1 00:12 cla2
-rw-r--r-- 1 nwc-laptop nwc-laptop 12511 Oct  1 00:12 cla2.c
-rwxr-xr-x 1 nwc-laptop nwc-laptop 27000 Oct  1 01:54 cla3
-rw-r--r-- 1 nwc-laptop nwc-laptop 14478 Oct  1 01:54 cla3.c
-rwxr-xr-x 1 nwc-laptop nwc-laptop 21680 Sep 30 08:45 claude
-rw-r--r-- 1 nwc-laptop nwc-laptop  7804 Sep 30 08:45 claude.c
-rw-r--r-- 1 nwc-laptop nwc-laptop  273 Oct  1 22:24 makefile
-rwxr-xr-x 1 nwc-laptop nwc-laptop 16000 Sep 26 11:10 my_program
-rw-r--r-- 1 nwc-laptop nwc-laptop  425 Sep 26 11:05 my_program.c
drwxr-xr-x 2 nwc-laptop nwc-laptop 4096 Oct  1 03:02 new
drwxr-xr-x 4 nwc-laptop nwc-laptop 4096 Sep 24 17:49 original
-rw-r--r-- 1 nwc-laptop nwc-laptop  168 Sep 30 08:41 output
-rw-r--r-- 1 nwc-laptop nwc-laptop  133 Oct  1 01:48 output.txt
-rwxr-xr-x 1 nwc-laptop nwc-laptop 21664 Sep 30 08:31 pipe
-rw-r--r-- 1 nwc-laptop nwc-laptop  6187 Sep 28 23:22 shell.c
-rw-r--r-- 1 nwc-laptop nwc-laptop  8046 Sep 30 08:31 shellPipe.c
-rw-r--r-- 1 nwc-laptop nwc-laptop  5642 Sep 24 20:53 test.c
-rwxr-xr-x 1 nwc-laptop nwc-laptop 42160 Oct  1 22:24 x
-rw-r--r-- 1 nwc-laptop nwc-laptop 14714 Oct  1 22:23 x.c
-rw-r--r-- 1 nwc-laptop nwc-laptop 37048 Oct  1 22:24 x.o
-rwxr-xr-x 1 nwc-laptop nwc-laptop 42160 Oct  1 22:24 xyz
SiSH >>> pwd
/home/nwc-laptop/dev/os-assignment
```

### [그림 20] 기본 명령어 실행 :

- execute\_command 함수에서 fork()를 사용하여 자식 프로세스를 생성하고 execve()를 통해 명령어를 실행
- find\_command\_in\_path 함수에서 PATH 환경변수를 파싱하고 각 디렉토리에서 명령어 검색

```

0 // PATH에서 명령어의 절대 경로를 찾는 함수
1 char* find_command_in_path(char *command) {
2     char *path_env = getenv("PATH");
3     if (path_env == NULL) {
4         return NULL; // PATH 환경 변수가 없으면 명령어를 찾을 수 없음
5     }
6
7     char *path = strdup(path_env); // PATH 문자열 복사
8     char *dir = strtok(path, ":"); // ':'로 구분된 각 디렉토리 처리
9     struct stat sb;
10    char full_path[BUFFER_SIZE];
11
12    while (dir != NULL) {
13        // 디렉토리와 명령어 이름을 결합하여 전체 경로 생성
14        snprintf(full_path, sizeof(full_path), "%s/%s", dir, command);
15
16        // 해당 경로에 실행 파일이 있는지 확인
17        if (stat(full_path, &sb) == 0 && sb.st_mode & S_IXUSR) {
18            free(path); // 메모리 해제
19            return strdup(full_path); // 명령어 절대 경로 반환
20        }
21
22        dir = strtok(NULL, ":"); // 다음 디렉토리로 이동
23    }
24
25    free(path); // 메모리 해제
26    return NULL; // 명령어를 찾지 못함
27 }

```

[그림 21] find\_command\_in\_path 함수

```

SiSH >>> pwd
/home/nwc-laptop/dev/os-assignment
SiSH >>> cd new
SiSH >>> pwd
/home/nwc-laptop/dev/os-assignment/new
SiSH >>> history
1: ls -l
2: pwd
3: cd new
4: pwd
5: history
SiSH >>> 

```

[그림 22] 내장 명령어 실행

```

SiSH >>> ls -l | grep my | sort
-rw-r--r-- 1 nwc-laptop nwc-laptop 425 Sep 26 11:05 my_program.c
-rwxr-xr-x 1 nwc-laptop nwc-laptop 16000 Sep 26 11:10 my_program
SiSH >>> ls -l | grep my
-rwxr-xr-x 1 nwc-laptop nwc-laptop 16000 Sep 26 11:10 my_program
-rw-r--r-- 1 nwc-laptop nwc-laptop 425 Sep 26 11:05 my_program.c

```

[그림 23] 파이프라인

- execute\_pipeline 함수에서 pipe()시스템 콜을 사용하여 파이프를 생성하고, 여러 자식 프로세스를 fork()하여 각 명령어 실행
- dup2 함수를 사용하여 표준 입출력을 파이프로 리다이렉션함

```

8 void execute_pipeline(char *commands[], int num_commands) {
9     int pipes[MAX_PIPES - 1][2];
10    pid_t pids[MAX_PIPES];
11
12    for (int i = 0; i < num_commands - 1; i++) {
13        if (pipe(pipes[i]) == -1) {
14            perror("pipe failed");
15            return;
16        }
17    }
18
19    for (int i = 0; i < num_commands; i++) {
20        pids[i] = fork();
21        if (pids[i] < 0) {
22            perror("fork failed");
23            return;
24        } else if (pids[i] == 0) {
25            // Child process
26            if (i > 0) {
27                dup2(pipes[i - 1][0], STDIN_FILENO);
28            }
29            if (i < num_commands - 1) {
30                dup2(pipes[i][1], STDOUT_FILENO);
31            }
32
33            for (int j = 0; j < num_commands - 1; j++) {
34                close(pipes[j][0]);
35                close(pipes[j][1]);
36            }
37
38            char *args[MAX_ARGS];
39            int arg_count = 0;
40            parse_command(commands[i], args, &arg_count);
41            handle_redirection(args, &arg_count);
42
43            char *path = find_command_in_path(args[0]);
44            if (path != NULL) {
45                execve(path, args, NULL);
46                perror("execve failed");
47                free(path);
48            } else {
49                fprintf(stderr, "Command not found: %s\n", args[0]);
50            }
51            exit(EXIT_FAILURE);
52        }
53    }
54
55    // Parent process
56    for (int i = 0; i < num_commands - 1; i++) {
57        close(pipes[i][0]);
58        close(pipes[i][1]);
59    }
60
61    for (int i = 0; i < num_commands; i++) {
62        waitpid(pids[i], NULL, 0);
63    }
64 }

```

```
SiSH >>> ls | grep .c | sort > output.txt
```

→

```
output.txt
1  cla2.c
2  cla3.c
3  claude.c
4  my_program.c
5  shell.c
6  shellPipe.c
7  test.c
8  x.c
9
```

[그림 24] 리다이렉션 (덮어쓰기)

```
SiSH >>> cat < output.txt
cla2.c
cla3.c
claude.c
my_program.c
shell.c
shellPipe.c
test.c
x.c
```

```
output.txt
1  cla2.c
2  cla3.c
3  claude.c
4  my_program.c
5  shell.c
6  shellPipe.c
7  test.c
8  x.c
9  makefile
```

```
SiSH >>> ls | grep make >> output.txt
```

→

[그림 25] 리다이렉션 (이어쓰기)

- handle\_redirection 함수에서 open()시스템 콜로 파일을 열고, dup2()로 표준 입출력을 리다이렉션 함

```

void handle_redirection(char *args[], int *arg_count) {
    for (int i = 0; i < *arg_count; i++) {
        if (strcmp(args[i], ">") == 0 || strcmp(args[i], ">>") == 0) {
            if (i + 1 < *arg_count) {
                int flags = O_WRONLY | O_CREAT;
                if (strcmp(args[i], ">>") == 0) {
                    flags |= O_APPEND;
                } else {
                    flags |= O_TRUNC;
                }
                int fd = open(args[i + 1], flags, 0644);
                if (fd == -1) {
                    perror("open failed");
                    return;
                }
                dup2(fd, STDOUT_FILENO);
                close(fd);
                args[i] = NULL;
                *arg_count = i;
            }
            break;
        } else if (strcmp(args[i], "<") == 0) {
            if (i + 1 < *arg_count) {
                int fd = open(args[i + 1], O_RDONLY);
                if (fd == -1) {
                    perror("open failed");
                    return;
                }
                dup2(fd, STDIN_FILENO);
                close(fd);
                args[i] = NULL;
                *arg_count = i;
            }
            break;
        } else if (strcmp(args[i], "2>") == 0) {
            if (i + 1 < *arg_count) {
                int fd = open(args[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
                if (fd == -1) {
                    perror("open failed");
                    return;
                }
                dup2(fd, STDERR_FILENO);
                close(fd);
                args[i] = NULL;
                *arg_count = i;
            }
            break;
        } else if (strcmp(args[i], "2>&1") == 0) {
            dup2(STDOUT_FILENO, STDERR_FILENO);
            args[i] = NULL;
            *arg_count = i;
            break;
        }
    }
}

```



<pre>SiSH &gt;&gt;&gt; sleep 20 &amp; [1] 38347 SiSH &gt;&gt;&gt; jobs [1] 38347 Running sleep SiSH &gt;&gt;&gt; fg 1 []</pre>	<pre>SiSH &gt;&gt;&gt; sleep 10 &amp; [1] 38854 SiSH &gt;&gt;&gt; bg 1 [1] 38854 continued SiSH &gt;&gt;&gt; [1] Done sleep SiSH &gt;&gt;&gt; []</pre>
--	--

[그림 26] 백그라운드 실행 , 작업관리(잡스, fg,bg)

- job 구조체와 jobs배열을 사용하여 백그라운드 작업을 관리
- handle\_sigchld함수에서 SIGCHLD신호를 처리하여 종료된 자식 프로세스 정리

```

void add_job(pid_t pid, char *command, int is_background) {
    if (job_count < MAX_JOBS) {
        jobs[job_count].pid = pid;
        strncpy(jobs[job_count].command, command, BUFFER_SIZE);
        jobs[job_count].is_background = is_background;
        job_count++;
    } else {
        fprintf(stderr, "Maximum number of jobs reached\n");
    }
}

void list_jobs() {
    for (int i = 0; i < job_count; i++) {
        printf("[%d] %d %s %s\n", i + 1, jobs[i].pid,
            jobs[i].is_background ? "Running" : "Stopped",
            jobs[i].command);
    }
}

void bring_to_foreground(int job_id) {
    if (job_id > 0 && job_id <= job_count) {
        int status;
        pid_t pid = jobs[job_id - 1].pid;

        if (kill(pid, SIGCONT) < 0) {
            perror("kill (SIGCONT)");
        }

        waitpid(pid, &status, WUNTRACED);

        if (WIFSTOPPED(status)) {
            printf("Job [%d] stopped\n", job_id);
            jobs[job_id - 1].is_background = 0;
        } else {
            // Remove the job from the list
            for (int i = job_id - 1; i < job_count - 1; i++) {
                jobs[i] = jobs[i + 1];
            }
            job_count--;
        }
    } else {
        fprintf(stderr, "Invalid job ID\n");
    }
}

void send_to_background(int job_id) {
    if (job_id > 0 && job_id <= job_count) {
        pid_t pid = jobs[job_id - 1].pid;

        if (kill(pid, SIGCONT) < 0) {
            perror("kill (SIGCONT)");
        } else {
            jobs[job_id - 1].is_background = 1;
            printf("[%d] %d continued\n", job_id, pid);
        }
    } else {
        fprintf(stderr, "Invalid job ID\n");
    }
}

void handle_sigchld(int sig) {
    pid_t pid;
    int status;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        for (int i = 0; i < job_count; i++) {
            if (jobs[i].pid == pid) {
                if (WIFEXITED(status) || WIFSIGNALED(status)) {
                    printf("[%d] Done %s\n", i + 1, jobs[i].command);
                    // Remove the job from the list

```

```
SiSH >>> history
1: sleep 100 &
2: jobs
3: fg 1
4: jobs
5: sleep 10 &
6: jobs
```

[그림 27] 히스토리 기능

- readline 라이브러리의 `add_history()` 함수를 사용하여 입력된 명령어를 히스토리에 추가
- `command_history` 배열에 명령어 저장, `history` 명령어로 출력

```
void add_to_history(const char* command) {
    if (history_count < MAX_HISTORY) {
        command_history[history_count] = strdup(command);
        history_count++;
    } else {
        free(command_history[0]);
        for (int i = 1; i < MAX_HISTORY; i++) {
            command_history[i-1] = command_history[i];
        }
        command_history[MAX_HISTORY-1] = strdup(command);
    }
}
```