

Operating System

Homework 1:

Report of Simple Shell (SiSH)

< 5 free-days left >

Name : SeokBeom Lim (임석범)

Department : Mobile System Engineering

Student number : 32203743

Table of contents

I. 서론.....	2
II. 본론	
1) General Concept of Operating system	
i. Concept of Operating system.....	2
ii. Process Memory	6
iii. Parent Process and Child Process.....	7
iv. Process of states	9
2) Overall Concept of Code	
3) Detail structure of code	
i. SiSH.c	12
ii. explore.c	14
iii. fedit.c	16
iv. devproc.c.....	19
4) Linux Command of Code and Feature of Shell	22
5) Linux Shell Result	23
III. 결론.....	26

서론

현대의 Operating System은 Computer의 Hardware와 User간의 Interface 역할을 한다. Operating System(OS)은 Hardware(HW) 자원을 효율적으로 관리하며 Software와 Hardware 사이에서 Process 관리, Memory 관리 등 여러 기능을 제공한다. 본 SiSH(Simple Shell) 프로젝트는 본 프로젝트를 통해 운영체제의 일반적인 개념과 Multi Process 의 개념에 대해서 이해하고자 한다. Sish 프로젝트는 사용자와 System의 상호작용하는 OS의 인터페이스 역할에 대해서 이해하기 위해 Shell을 통해 프로그램을 실행하여 File과 Process를 관리 및 제어하는 역할을 구현한다. Shell은 SW와 HW의 사이에서 운영체제가 제공하는 System 호출을 통해 Process를 생성하고 Memory를 관리하여 입출력 하는 다리(Interface) 역할을 한다. 특히, Sish Shell에서 구현된 여러 Custom 명령어들을 통해 Process 간 통신 및 File 관리에 대해 어떤 연관이 되어 있는지 설명한다.

본론

1. General Concept of Operating System

1) Operating system의 개념

Operating System(OS)은 Computer System에서 중요한 역할을 담당하는 핵심 Software로 Hardware와 Application Software 간의 중재자 역할을 한다. OS는 HW 자원, 특히 CPU, Memory, File System, 그리고 Input/Output 장치의 관리를 책임지며, Hardware 자원을 효율적으로 사용할 수 있도록 다양한 기능을 제공한다. 위 과정에서 OS는 여러 Process의 실행을 관리하며 System의 안정성과 보안을 유지하는 핵심 역할을 한다.

OS의 가장 중요한 기능 중 하나는 Process 관리이다. OS는 여러 Process가 동시에 실행될 수 있도록 스케줄링(Scheduling)하고, 각 Process가 적절하게 자원을 사용할 수 있도록 관리한다. 위 Scheduling이 중요한 부분을 차지하는 Multitasking은 여러 Process가 동시에 실행되는 것을 가능하게 한다. 이때 Processor Scheduler는 각 Process의 우선순위와 실행 시간을 고려하여 CPU의 작업 순서를 결정한다.

Memory 관리도 OS의 중요한 역할 중 하나이다. OS는 각 Process가 자신의 메모리 영역을 효율적으로 사용할 수 있도록 관리하며, Process 간의 메모리 충돌을 방지하기 위해 메모리 보호 기능을 제공한다. 이때, OS는 Virtual Memory (가상 메모리)와 같은 기법을 사용하여, 실제 물리적 메모리보다 더 많은 메모리 공간을 사용할 수 있도록 지원하게 한다.

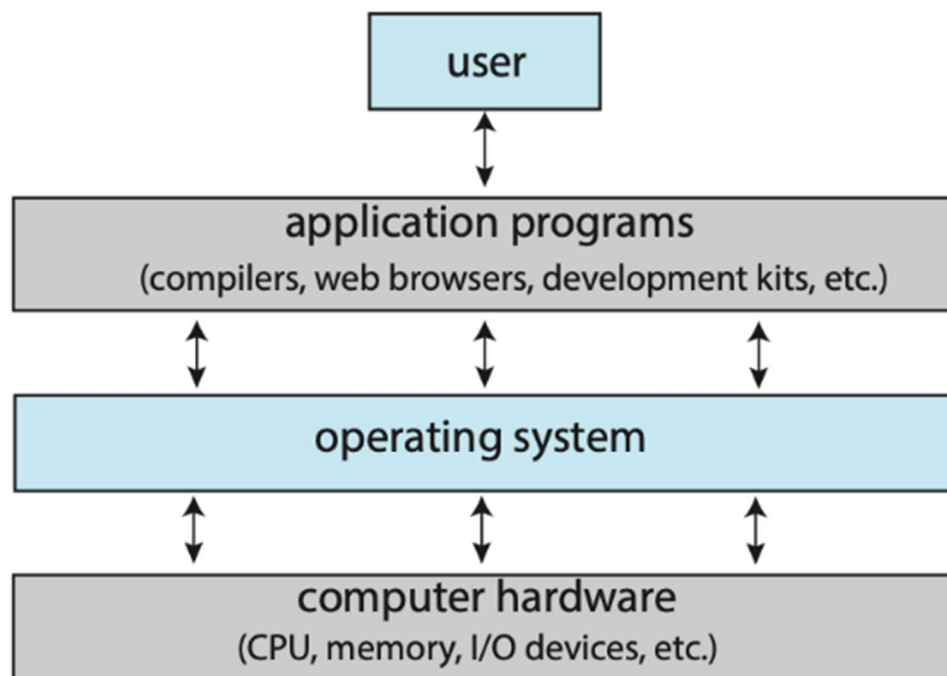
또한, OS는 File System을 관리하여, 파일의 생성, 삭제, 읽기, 쓰기 작업을 지원한다. OS는 파일의 위치를 체계적으로 관리하고, 파일에 대한 접근 권한을 설정함으로써 시스템 보안을 유지한다. 이러한 OS의 특징은 User가 OS를 통해 다양한 파일을 쉽게 관리하고, 데이터 저장 및 검색을 할 수 있다.

위와 같은 OS의 기능을 응용 프로그램이 사용하기 위해서는 System Call을 필요로 한다. System Call은 application program이 OS의 기능을 사용할 수 있도록 중개하는 Interface 이다. System Call은 커널(Kernel) 모드에서 실행되며, 이때 CPU는 User Mode에서 Kernel Mode로 전환된다. 커널 모드는 시스템 자원에 대한 최고 접근 권한을 가지고 있으며, OS의 핵심적인 작업이 이 모드를 통해 수행된다. OS는 Kernel Mode와 User Mode라는 두 가지 모드를 제공하는데, 이 두 모드는 시스템의 안정성과 보안을 보장하는 핵심 요소이다. Kernel Mode에서는 CPU가 모든 명령어와 자원에 접근할 수 있으며, 시스템의 중요한 기능들이 Kernel 모드에서 실행되는

반면, User Mode는 Application program이 실행되는 제한된 권한을 가지며, Application program은 HW 자원에 직접 접근할 수 없고, System Call을 통해서만 커널에 요청을 보낼 수 있다.

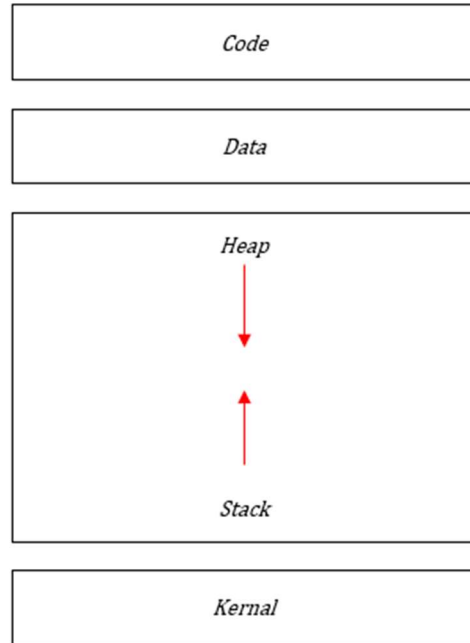
위 같은 Application program 특징은 직접 HW 자원에 접근하지 않도록 하여, System이 Application program의 오류나 악성 프로그램으로부터 보호 할 수 있다. Kernal Mode와 User Mode의 구분은 HW 자원을 안전하게 관리하고, 오류나 보안 위협으로부터 시스템을 보호하는 중요한 기법이다.

결론적으로, Operating System은 Computer System의 핵심적인 자원을 효율적으로 관리하고, System Call을 통해 Application Software와 HW 간의 상호작용을 안전하게 중재한다. Kernel Mode와 User Mode의 분리는 시스템의 안정성과 보안을 유지하는 데 중요한 역할을 하며, 이를 통해 OS는 사용자와 시스템 간의 상호작용을 안전하고 효율적으로 처리할 수 있다.



(출처 : <https://seonghun120614.tistory.com/m/208>)

2) Process Memory



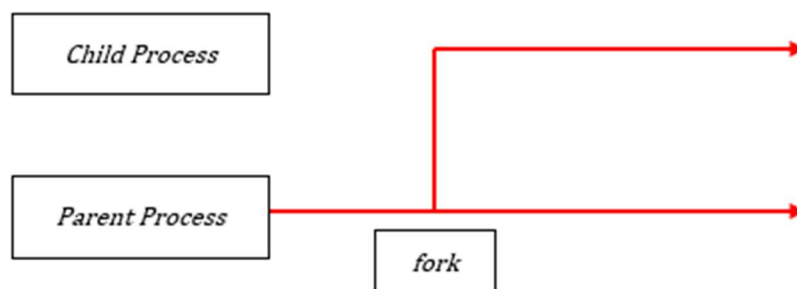
Process 는 실행 중인 Program 이다. Process는 OS가 작업을 수행하는 하나의 Unit 역할을 한다. Process는 Memory 를 사용하는데, Process Memory 구조는 HW 자원관리를 효율적으로 이루어 질 수 있도록 중요한 역할을 한다. Process Memory 구조는 크게 4가지 영역으로 나누어 지는데, 각각 Code 영역, Data 영역, St ack 영역과 Heap 영역으로 나뉜다.

1. **Code 영역** : Code 영역은 Process 가 실행할 명령어, 즉 프로그램의 실행되는 코드를 저장한다. 이 영역은 읽기 전용으로 설정되어 있으며 Program 실행 중 수정 되지 않게 하여 안전하게 명령어를 실행 할 수 있게 한다.
2. **Data 영역** : 이 영역에서는 Program 에서의 Global 변수와 Static 변수가 저장 된다. 이 Data는 Program의 시작부터 종료 시점까지 Data를 유지 하며 Data의 초기화의 유무에 따라 나뉘어 관리된다.

3. **Stack 영역** : Stack 영역에서는 Local 변수, Parameter 와 Return Address를 저장하는 영역이다. 함수 호출 시 새로운 Stack Frame이 생성되며, 함수가 종료되면 해당 Frame이 사라진다. 이에 대한 예시는, `addi sp sp -4`로 Stack 영역에 영역을 만들어 준 후 해당 영역에 관련 변수를 저장해준다.
4. **Heap 영역** : Heap 영역은 동적으로 할당 된 메모리를 저장하는 영역이다. 이 영역은 Program이 실행 중에 필요한 메모리를 할당하거나 해제하는데 사용되는데, Data 영역과 다르게 메모리의 크기를 동적으로 변화 시킬 수 있다.

3) Parent process and Child Process

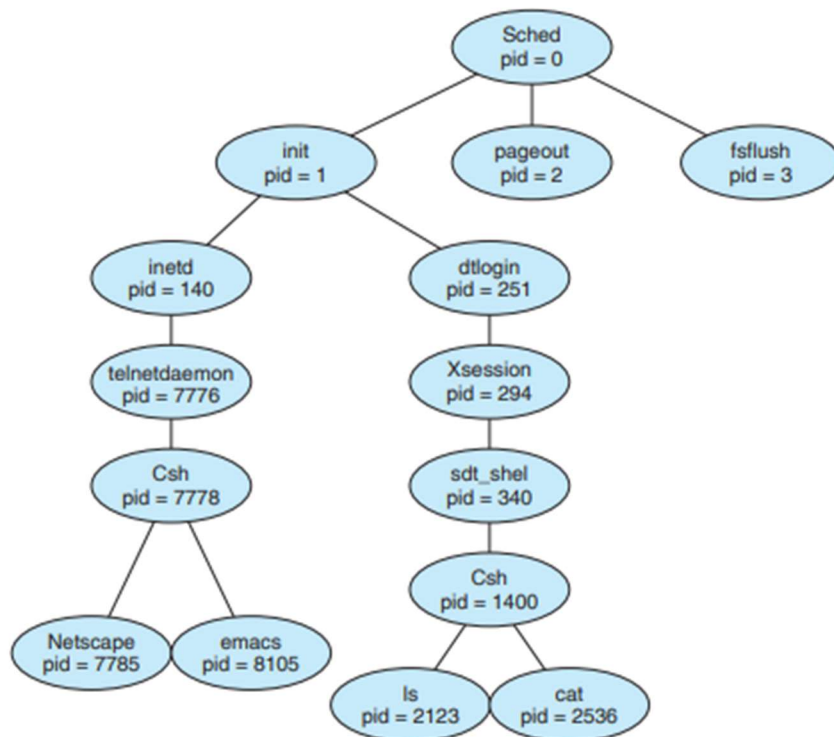
Process 를 실행 할 때, Process는 System call을 통해 새로운 Process를 생성 할 수 있다. 이때 Process를 생성하는 주체는 Parent process라 하고, Parent process 에 의해 생성된 Process는 Child Process라 부른다. 이러한 Process는 다음과 같이 표시할 수 있다.



이러한 새로운 Process 를 만들 때, fork 라는 System call을 통해 생성이 되는데, fork System call은 현재 실행 중인 Parent process의 복사본을 만들어 새로운 Process를 만든다. 이 경우, Parent process와 Child Process는 Memory 공간을 복사 하여 동일한 내용을 가지지만 서로 독립적으로 실행된다.

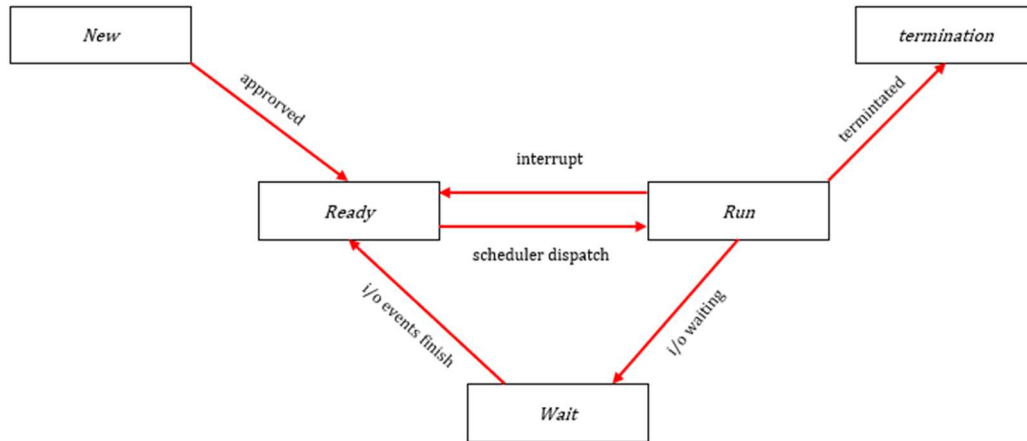
대부분의 OS에서는 Process들을 구별 하기 위해 정수를 통해 구별한다. 이 정수는 PID(Process Identifier) 라고 하며, PID는 고유한 번호 이므로 다른 Process와 구별되는 유일한 정수를 각각 사용한다.

Child Process 는 Parent process 로부터 자원을 상속 받아 진행한다. 즉, Parent Process의 자원의 일부를 한정적으로 사용한다. 같은 자원을 사용하지만, Parent Process와 Child Process는 독립적으로 실행되며 이에 대한 상태는 OS를 통해 관리된다. 이러한 독립적 실행은 병렬적 실행을 구현 할 수 있게 하였다. 아래는 PID 에 따른 Parent process와 Child process를 Tree 구조로 만든 예시이다.



(출처 : <https://4legs-study.tistory.com/37>)

4) Process of States



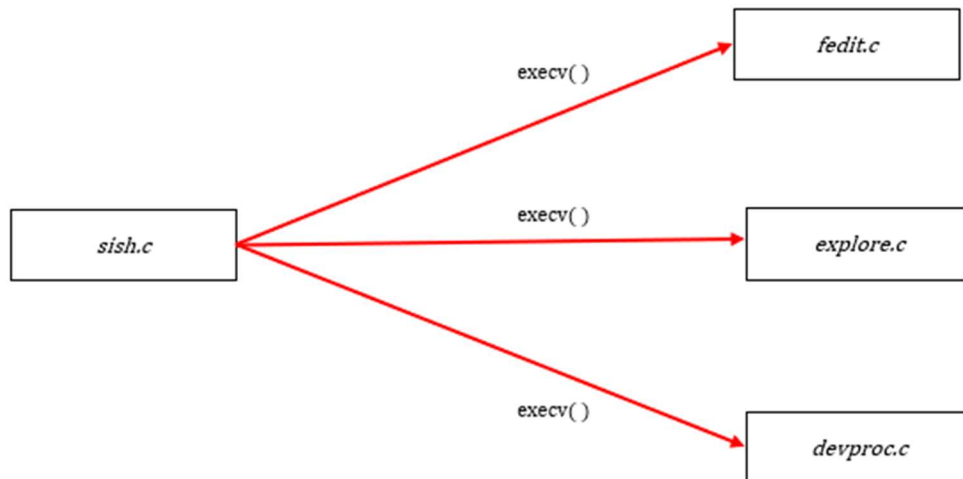
Process는 실행 되는 동안 여러 상태를 거친다. OS에서 Process가 어느 단계에 있는지 나타내는데, 위와 같은 Process는 여러 State로 전이 되는데 이는 Process Scheduler에 의해 제어된다. 이러한 State 변화를 기반으로 HW 자원을 효율적으로 관리한다.

1. New State : OS에 Process의 PCB(Process control block) 이 생성 되며, Memory에 Process가 추가되어 Process 준비를 마친 상태이다.
2. Ready State : Process가 CPU에 할당되기를 기다리는 상태이다. 이 State에서는 Process에 필요한 자원을 확보한 상태이며 Process Scheduler에 따라 이 상태의 Process중 어느 Process를 실행할지 기다린다.
3. Run State : CPU가 해당 Process에 할당 되어 실행 중인 State이다. 이 State에서는 Process가 명령어를 실행하며, HW 자원을 사용한다.
4. Wait state : Process가 I/O events와 같은 이벤트를 기다리는 상태이다. 위 Wait State에서는 I/O 작업이 완료되거나 이벤트가 완료되었다는 시그널이 발생하면 다시 Ready State로 바뀌게 된다.
5. Terminate state : Process의 실행이 OS에 의해 종료된 상태이다.

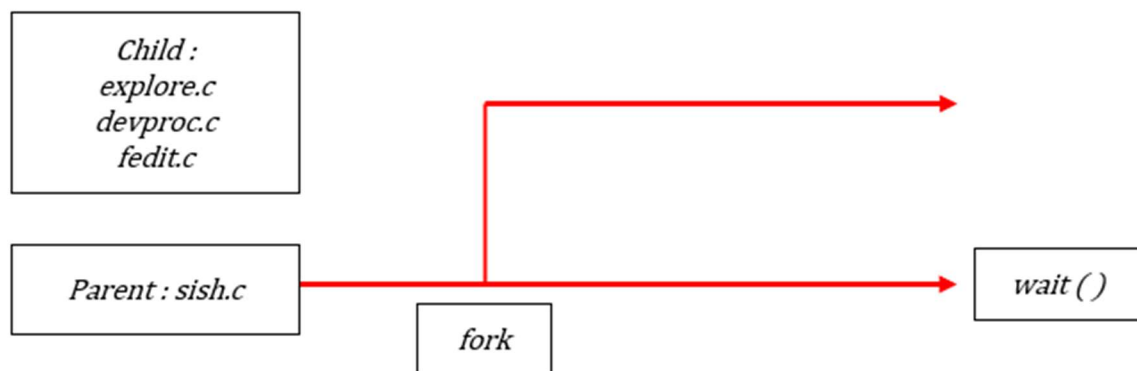
2. Overall Concept of Code

Sish (Simple Shell)은 Linux 환경에서 사용자와 System 간의 상호작용을 간소화 하여 사용자가 Custom 명령어와 Linux 명령어를 실행 할 수 있도록 하는 Shell Program을 목적으로 만들어 졌다. 이 Program은 기본적인 Shell 기능 뿐만 아니라 다양한 Custom 명령어를 지원하며, 각각의 기능은 분리된 모듈형 설계를 가진다. Custom 명령어의 특징은 환경 변수 관리, 파일 및 디렉토리 관리, 프로젝트 관리 및 컴파일 기능을 가진다.

Sish Shell Code의 특별한 특징은 fork와 execve 함수를 통해 각각의 C파일을 불러, 각각의 Process를 진행 할 수 있는 C코드 가진다. 즉, `sish.c` 코드는 `main` 함수의 역할을 하며 각각의 `fedit.c`, `explore.c`, `devproc.c` 는 `sish.c` 코드의 Child process 에서 `execv` 함수를 통해 각각 불러진다. 각각의 모듈화 된 코드들은 `sish.c` 파일 내에서 하나의 Process의 역할을 수행하게 된다.



SiSH shell의 Concept은 전반적으로 두가지가 있다. 첫번째는 C코드의 Project Templet 형성을 간소화된 명령어를 구현하는 것과 파일 디렉토리 관리를 수월하게 하는 Custom code를 구현하는 것이다. Project Templet을 형성하기 위해서, 디렉토리를 형성 할 수 있는 명령어와 C파일을 생성 및 수정 할 수 있는 능력이 요구된다. 위 요구사항을 하나의 명령어로 구현하기 위해서 explore.c 와 fedit.c의 Process가 devproc.c Process 에서 사용되어야 한다. 이는 Detail Structure of Code에서 자세히 다룬다. 두번째는 Sish 구현에 있어 Parent Process와 Child Process로 Process 분리하여 병렬처리를 시도하는 Concept 이다. 이 코드의 main 역할을 수행하는 sish.c의 경우, Child Process에서 execv 함수를 이용하여 입력된 명령어에 맞게, 모듈화 된 C코드중 하나를 부른다. 이때, Parent process에서는 child process가 끝나기 까지 wait함수를 통해 기다린다. devproc.c 의 경우 sish.c의 Child Process 이지만, 내부 3번의 fork를 통해 다른 C 코드(Process) 의 Parent process 역할을 하기도 한다.

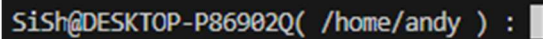


요약하자면, sish.c 코드는 Parent Process 의 역할을 수행하며, 나머지 C코드는 각각 sish.c의 Child Process로 수행된다. 즉, 이 SiSH에서 각각의 C코드는 하나의 Process의 역할과 유사하다.

3. Detail structure of Code

1) SiSH.c

sish.c 는 사용자가 입력한 명령어를 처리하는 Main Shell Program 이며, 4가지 기능을 수행한다. 첫번째 기능은 prompt 함수를 통해 현재 host 이름, 현재 Directory의 Path를 가져와 Shell의 기본 창을 띄운다. 기본 창은 사용자가 input을 받을 수 있게 띄워 놓은 문자열을 출력하는데, 기본 Shell의 형식은 “ SiSh@hostname (Current Working Dir) : “ 이다.



두번째 기능은 사용자로부터 받은 입력값을 나누어 주는 기능이다. 이 기능은 parse_input 함수를 통해 구현되며 각각의 input을 arg 배열로 넘겨주는 역할을 한다. 예를 들면, explore -s file_name 의 경우 한 문장으로 된 해당 문장을 explore, -s, file_name으로 parse 해준다.

세번째 기능은 필요한 프로세스를 구별하여 진행하는 역할을 한다. 만약, explore -s file_name 의 경우 explore Process가 요구 되므로, explore.c 파일을 execv 함수를 통해 실행하게 한다. 만약, 명령어의 첫 부분이 explore, fedit, devproc가 아니라면 네번째 기능 또는 리눅스 명령어인지 확인 한다. 위 두 경우에 대해 모두 해당사항이 없다면, 명령어가 발견되지 않았음을 Shell에 표시해준다.

네번째 기능은 **envir 명령어 모음**이다. 해당 명령어는 환경 변수를 조회하거나 설정하는 기능을 가진다. 현재 구현된 Custom 명령어들은 explore 명령어를 제외하면 모두 현재 Directory에서 진행되므로, **current working directory**를 필수적으로 인지하여야 한다. 아래는 envir 명령어 기능을 정리한 부분이다.

기능	arg[0]	arg[1]	arg[2]	arg[3]
cwd	envir	-c		
edit cwd	envir	-e	dir name	
list	envir	-l		
set	envir	-s	file name	content

1. `envir -c`

Current Working Directory를 조회하는 기능을 가진다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : envir -c
Current working directory: /home/andy
```

2. `envir -e PATH`

Current Working Directory를 수정하는 기능을 가진다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : envir -e /home/andy/Hw1
Directory changed to: /home/andy/Hw1
```

3. `envir -l`

현재 환경 변수들을 보여주는 기능을 가진다.

```
SiSh@DESKTOP-P86902Q( /home/andy/Hw1 ) : envir -l
SHELL=/bin/bash
COLORTERM=truecolor
WSL2_GUI_APPS_ENABLED=1
TERM_PROGRAM_VERSION=1.93.1
WSL_DISTRO_NAME=Ubuntu
NAME=DESKTOP-P86902Q
PWD=/home/andy
LOGNAME=andy
VSCODE_GIT_ASKPASS_NODE=/home/andy/.vscode-server/bin/38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/node
MOTD_SHOWN=update-motd
HOME=/home/andy
```

4. `envir -s content`

환경변수를 설정하는 기능을 가진다.

```
Sish@DESKTOP-P86902Q( /home/andy ) : envir -s TERM "xterm-256color"  
Environment variable TERM set to xterm-256color
```

2) `explore.c`

`explore.c`의 경우 파일시스템 탐색과 관련된 다양한 기능을 제공하는 Process이다. 이 프로그램은 파일 및 Directory 구조를 탐색하고, 파일을 출력하거나, 특정 파일의 이름과 확장자를 따라 검색하는 기능을 제공한다. 각각의 기능들은 `explore.c`의 경우 각각의 기능들이 함수화 되어 있으며, 파일과 디렉토리를 찾기위해 `dirent.h` library 함수인 `opendir`과 `readdir`과 같은 System 호출을 통해 Directory 내용을 읽어 온다. 또한, `dirent.h` library의 DIR 구조체를 이용하여 현재 디렉토리 및 찾는 디렉토리의 값을 받게 된다. Sish를 통해 불러온 Process(C 코드)들은 공통적으로 사용자가 Sish Prompt 에서 입력한 입력값(명령어)을 해당 c파일 `main`함수의 arguments인 `argv[]`에 받게 된다. `explore.c`의 기능은 다음과 같다.

기능	arg[0]	arg[1]	arg[2]	arg[3]
search	explore	-s	file name	PATH
type	explore	-t	extension type	
output	explore	-o	file name	PATH
show dir	explore	-sh	PATH	
help	explore	-h		

1. explore -s filename PATH

특정 경로에서 파일이름을 검색한다. PATH 부분을 생략 할 시 현재 경로를 기준으로 찾는다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : explore -s temp.c
File found: ./temp.c

SiSh@DESKTOP-P86902Q( /home/andy ) : explore -s hello.c /home/andy/Hancode
File found: /home/andy/Hancode/hello.c
```

2. explore -t 확장자 PATH

특정 경로에서 특정 확장자를 가진 파일을 검색한다. 1번과 마찬가지로 PATH 생략 시 현재 경로에서 찾기 진행.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : explore -t .txt
File:
love.txt
hihi.txt
```

3. explore -o filename PATH

특정 경로(생략 시 현재 경로) 에 찾는 파일을 읽을 수 있다면, 파일 내용을 출력한다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : explore -o hihi.txt
hihi what are you doing
```

4. explore -sh PATH

PATH Directory 안에 있는 모든 하위 파일을 트리 구조로 출력한다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : explore -sh /home/andy/2024
|-- temp_2
|-- Hancode
    |-- input_data_handler.c
    |-- input_data_handler.o
    |-- hello.c
    |-- main.out
    |-- main
    |-- main.c
```

5. explore -h

explore 내부 명령어들에 대한 설명을 알려주는 기능이다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : explore -h
Explore Command Help:
  explore -s filename [path] : Search for a file in the specified path (default: current path)
  explore -t extension [path] : Search for files with the specified extension in the path
  explore -o filename       : Output the contents of a file
  explore -sh [path]         : Display directory structure as a tree (default: current path)
  explore -h                 : Show help information
```

3) fedit.c

fedit.c 는 sish.c에서 받는 Child Process에서 `execv` 함수를 통해 불러지는 Process이다. 이 C 코드는 파일 및 Directory 생성, 편집, 삭제 등과 같이 파일과 Directory의 편집작업을 수행하는 프로그램이다. 이 기능에서 Linux 명령어 `echo`와 같이 간단한 메시지를 출력하는 기능도 제공된다.dj

기능	arg[0]	arg[1]	arg[2]	arg[3]
edit file	fedit	-e	file name	
make file	fedit	-m	file name	content
make dir	fedit	-md	dir name	
remove file	fedit	-rm	file name	
remove dir	fedit	-rmdir	dir name	
add file	fedit	-a	file name	content
help	fedit	-h		
echo	fedit	memo	content	

1. fedit -e filename

기존 파일을 수정하는 기능을 제공한다. 파일을 열고 현재 파일 내용 출력 후, 새롭게 입력된 내용의 파일을 덮어쓴다. 입력 받은 파일을 fopen 함수를 통해 열고 rewind 함수를 통해 파일의 첫 STREAM으로 돌아간 후 새로 입력 받은 Data를 덮어쓴다. (C파일 등 file io 가능하면 모든 파일 가능)

```
Sish@DESKTOP-P86902Q( /home/andy ) : fedit -e hihi.txt
Current content of hihi.txt:
hihi what are you doing

Enter new content for hihi.txt (Ctrl+D to save):
hihi what do you do?

Sish@DESKTOP-P86902Q( /home/andy ) : explore -o hihi.txt
hihi what do you do?
```

2. fedit -m filename

3. fedit -md dirname

파일/Directory를 생성하는 기능을 가진다. mkdir 함수를 사용 했으며 mkdir(dirname, 0755) 에서 0755의 의미는 소유자가 4+ 2+ 1 =7 R,W,실행의 권한을 가지며 그룹 사용자는 4+ 1 = 5 R,W의 권한을 가지며 이외의 사용자는 4 + 1 = 5 R,W의 권한을 가지게 된다. make_file의 경우 fopen 함수를 통해서 구현된다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : fedit -md mobile
Directory created: mobile

SiSh@DESKTOP-P86902Q( /home/andy ) : fedit -m mobile.c
File created: mobile.c
```

```
C mobile.c
```

```
> mobile
```

4. fedit -rmdir dirname

5. fedit -rm filename

위 명령어들은 제거 명령어들이며 -rmdir 의 경우 빈 Directory에 한해서만 삭제하며 rmdir 함수를 통해 Directory가 지워진다. -r 의 경우 unlink 함수를 통해서 삭제가 된다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : fedit -rmdir mobile
Directory removed: mobile

SiSh@DESKTOP-P86902Q( /home/andy ) : fedit -rm mobile.c
File removed: mobile.c
```

6. fedit memo "content"

memo는 echo기능과 유사한 기능을 한다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : fedit memo "hihi"
hihi
```

7. fedit -h

```

SiSh@DESKTOP-P86902Q( /home/andy ) : fedit -h
Fedit Command Help:
fedit -e filename          : Edit an existing file
fedit -m filename [content] : Create a new file with optional content
fedit -md dirname          : Create a new directory
fedit -rm filename         : Remove a file
fedit -rmdir dirname       : Remove a directory
fedit -a filename [content] : Append content to a file
memo "message"             : Display a message to the screen

```

4) devproc.c

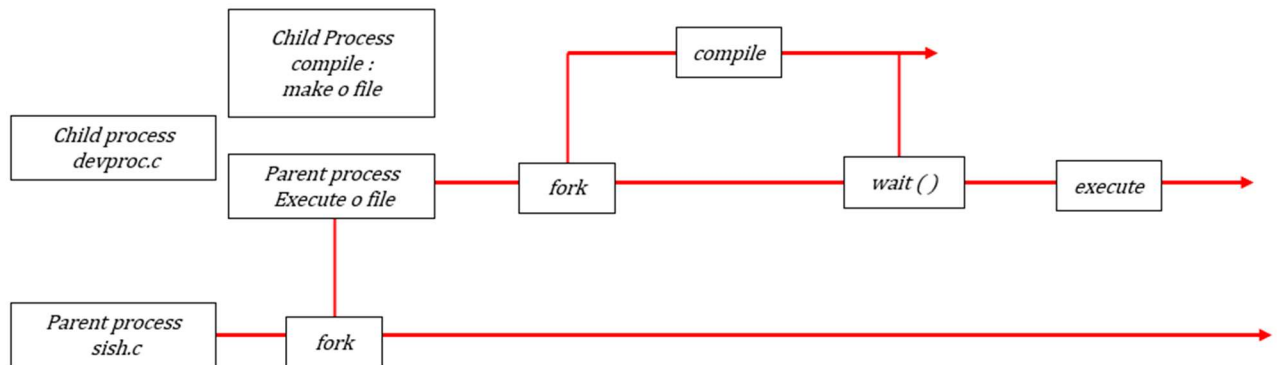
개발자를 위한 명령어 모음 C코드이다. Project templet을 자동으로 생성하고 관리하는 기능은 구현되었으며 Compile 명령어의 경우 Compile 명령어를 통해 해당파일의 .c파일을 .o 파일로 compile 해준 후, 실행까지 하는 명령어이다. 이 명령어 코드는 Parent process와 Child Process의 병렬적으로 코드가 진행되었다.

기능	arg[0]	arg[1]	arg[2]	arg[3]
compile	devproc	-c	file name	
makefile	devproc	-mf	file name	
templet	devproc	-t	dir name	
help	devproc	-h	file name	

1. devproc -c filename.c

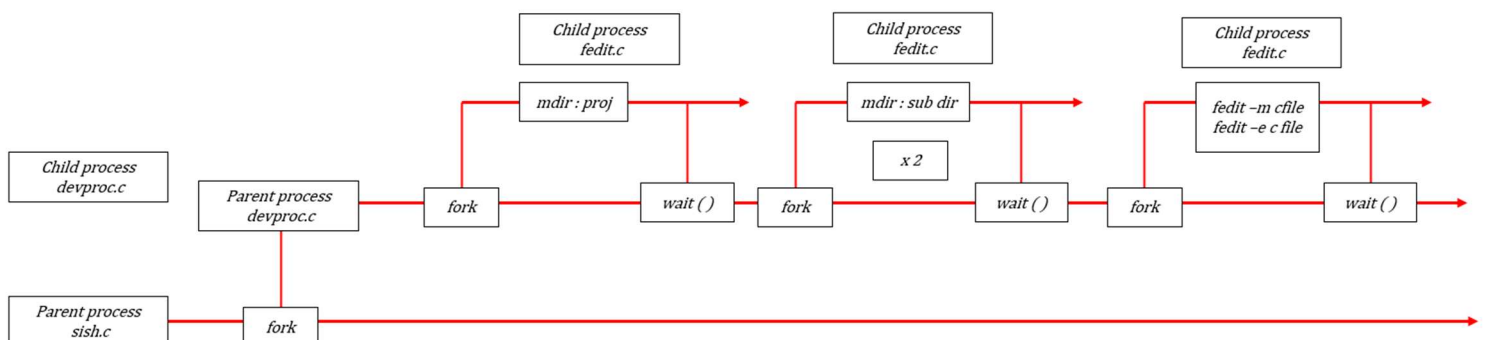
해당 기능은 파일을 컴파일하고 실행하는 명령어이다. 이 기능의 Concept은 사용자가 주어진 Source file을 compile 하고 생성된 실행파일을 실행하는 기능이다. 이 부분은 Parent process와 Child Process의 병렬적 Process를 통해 Compile을 구현하고자 했다. Child Process의 경우, C file을 Compile하고 Compile이 끝날 때 까지 기다린 Parent process는 O file을 실행하는 구조를 가진다. 해당 명령어의 compile이 구현되지 않아 ofile을 따로 만들고 실행해야 한다. o file이 만들어진 상태에서 해당 파일을 compile 하면 Child process는 진행되지 않지만 Parent process가

진행되어 실행된 파일의 결과를 볼 수 있다. 아래는 Compile 명령어의 구조이다.



2. devproc -t project name

해당 명령어는 Project templet을 만들어 주는 명령어이다. Project templet 구성은 Project 하위 Directory에 Source code directory와 Header file directory가 있으며 Sources code directory 내부에는 C파일을 생성한다. 해당 C파일 이름은 main.c 파일이며 마지막에 main.c를 fedit -e를 통해 해당 C파일을 수정하는 절차를 가진다. 즉, devproc.c 파일에서 fedit.c 파일을 fork와 execv 함수를 통해서 다시 불러들인다. 아래는 해당 명령어의 구조이다.



3. devproc -m ccode.c (부분 구현)

위 명령어는 ccode.c의 makefile을 만들어준다.

```
SiSh@DESKTOP-P86902Q( /home/andy ) : devproc -t mobile
Directory created: mobile
Project directory mobile created.
Directory created: mobile/header
Header directory created: mobile/header
Directory created: mobile/source
Source directory created: mobile/source
File created: mobile/source/main.c
main.c file created: mobile/source/main.c
Current content of mobile/source/main.c:

Enter new content for mobile/source/main.c (Ctrl+D to save):
#include <stdio.h>

int main(void)
{
    printf("hello Linux\n")

    return 0;
}

Content written to main.c in mobile/source/main.c
Directory created
Subdirectories created

SiSh@DESKTOP-P86902Q( /home/andy ) : explore -sh /home/andy/mobile
|-- header
|-- source
    |-- main.c

SiSh@DESKTOP-P86902Q( /home/andy ) : explore -o /home/andy/mobile/source/main.c
#include <stdio.h>

int main(void)
{
    printf("hello Linux\n")

    return 0;
}
```

```
Sish@DESKTOP-P86902Q( /home/andy ) : devproc -c temp.c
Running the program: ./temp
hello to my friends
```

4. Linux Command of Code

본 장에서는 Custom Command가 아닌 **Linux 명령어에 대한 Code 처리**에 대해서 설명하고 이에 대해 결과로 보여주고자 한다. fork를 통해 Child Process에서 진행 되며, Linux 기본 명령어의 경우 Custom 명령어를 조건문으로 먼저 확인한 이후, args 에 넣어진 값을 **execv 함수의 Parameter로 입력하여 실행 된다**. Linux Code와 함께 구현된 기능은 **Pipe 기능, Redirection 기능, 복합 Redirection 및 Pipe 기능이 있다**. **Pipe()** 의 경우, 두개의 명령어를 연결하며, 첫번째 출력을 두번째 명령의 입력으로 전달 할 수 있다. 해당 기능을 구현하기 위해, pipe 함수의 System call을 통해 형성한 후 dup2를 통해 Pipe의 입력과 출력을 연결하여 실행 된다. **Redirection** 의 경우 > < 를 사용하여 입력과 출력 Redirect 하는 기능이다. 위 기능을 구현하기 위해 open 함수와 dup2 함수를 통해 파일을 열었으며, Standard input 또는 Standard output을 **execv 함수가 진행 되기 전에 Redirection 처리된다**. 이 Sish의 기능의 한계 중 하나는 **Pipe와 Redirection을 복합적으로 사용 할 수 있지만 각 Pipe와 Redirection은 한번 씩만 사용된 명령어에 한해서만 가능하다**. 아래는 Linux 기본 명령어와 Pipe, Redirection, Pipe&Redirection 기능을 사용한 사진이다.

```
SiSh@DESKTOP-P86902Q( /home/andy/mobile ) : cat second.c
#include <stdio.h>

int main(void){

    printf("Hello Linux");

    return 0;

}

SiSh@DESKTOP-P86902Q( /home/andy/mobile ) : ls -l
total 32
-rw-r--r-- 1 andy andy 2471 Sep 26 12:27 first.c
drwxr-xr-x 2 andy andy 4096 Sep 30 10:24 header
-rw-r--r-- 1 andy andy   24 Sep 30 09:28 hihi.txt
-rw-r--r-- 1 andy andy    4 Sep 30 16:07 hiho.txt
-rw-r--r-- 1 andy andy   77 Sep 29 14:34 second.c
-rw-r--r-- 1 andy andy 8010 Sep 28 21:39 simple_shell.c
drwxr-xr-x 2 andy andy 4096 Sep 30 10:24 source

SiSh@DESKTOP-P86902Q( /home/andy/mobile ) : pwd
/home/andy/mobile

SiSh@DESKTOP-P86902Q( /home/andy/mobile ) : cd ..
Directory changed to: /home/andy

SiSh@DESKTOP-P86902Q( /home/andy ) : echo "hello world"
hello world

SiSh@DESKTOP-P86902Q( /home/andy ) : whoami
andy

SiSh@DESKTOP-P86902Q( /home/andy ) : █
```


[Redirection]

```
Sish@DESKTOP-P86902Q( /home/andy ) : echo "This is a test" > test.txt
Sish@DESKTOP-P86902Q( /home/andy ) : cat test.txt > output.txt
Sish@DESKTOP-P86902Q( /home/andy ) : cat output.txt
This is a test
Sish@DESKTOP-P86902Q( /home/andy ) : cat test.txt
This is a test
Sish@DESKTOP-P86902Q( /home/andy ) : grep "#include" sish.c > includes.txt
Sish@DESKTOP-P86902Q( /home/andy ) : cat includes.txt
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <fcntl.h>
Sish@DESKTOP-P86902Q( /home/andy ) : cat < test.txt
This is a test
Sish@DESKTOP-P86902Q( /home/andy ) : █
```

[Pipe]

```
Sish@DESKTOP-P86902Q( /home/andy ) : ls -l | grep ".c"
-rwxr-xr-x 1 andy andy 21288 Sep 29 20:37 devproc
-rw-r--r-- 1 andy andy 8261 Sep 30 16:20 devproc.c
-rw-r--r-- 1 andy andy 5297 Sep 30 16:20 explore.c
-rw-r--r-- 1 andy andy 5761 Sep 30 09:15 fedit.c
-rw-r--r-- 1 andy andy 120 Sep 30 16:37 includes.txt
-rw-r--r-- 1 andy andy 456 Sep 28 20:01 prog.c
-rw-r--r-- 1 andy andy 7992 Sep 30 16:43 sish.c
-rw-r--r-- 1 andy andy 148 Sep 29 19:41 temp.c

Sish@DESKTOP-P86902Q( /home/andy ) : cat temp.c | sort

    printf("hello to my friends\n");
    return 0;
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[]){
}

Sish@DESKTOP-P86902Q( /home/andy ) : who | wc -l
1

Sish@DESKTOP-P86902Q( /home/andy ) : ps aux | grep "bash"
root      230  0.0  0.0   4784  3368 ?        Ss   Sep29   0:00 /bin/bash
andy      396  0.0  0.0   6124  4792 pts/1    S+   Sep29   0:00 -bash
andy      716  0.0  0.0   6244  5216 pts/6    Ss   Sep29   0:00 /bin/bash
ash.sh
```


[Redirection & Pipe, 그리고 gcc 관련 명령어]

```
Sish@DESKTOP-P86902Q( /home/andy ) : grep "#include" sish.c | sort > include_sorted.txt

Sish@DESKTOP-P86902Q( /home/andy ) : cat include_sorted.txt
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
```

```
Sish@DESKTOP-P86902Q( /home/andy ) : ls -l | grep ".c" > c_files.txt

Sish@DESKTOP-P86902Q( /home/andy ) : cat c_files.txt
-rw-r--r-- 1 andy andy      0 Sep 30 16:49 c_files.txt
-rwxr-xr-x 1 andy andy 21288 Sep 29 20:37 devproc
-rw-r--r-- 1 andy andy  8261 Sep 30 16:20 devproc.c
-rw-r--r-- 1 andy andy  5297 Sep 30 16:20 explore.c
-rw-r--r-- 1 andy andy  5761 Sep 30 09:15 fedit.c
-rw-r--r-- 1 andy andy   120 Sep 30 16:48 include_sorted.txt
-rw-r--r-- 1 andy andy   120 Sep 30 16:37 includes.txt
-rw-r--r-- 1 andy andy   456 Sep 28 20:01 prog.c
-rw-r--r-- 1 andy andy  7992 Sep 30 16:43 sish.c
-rw-r--r-- 1 andy andy   148 Sep 29 19:41 temp.c
```

```
Sish@DESKTOP-P86902Q( /home/andy ) : grep "main" sish.c | wc -l > main_count.txt

Sish@DESKTOP-P86902Q( /home/andy ) : cat main_count.txt
1
```

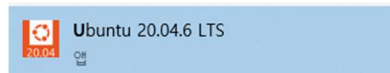
```
Sish@DESKTOP-P86902Q( /home/andy ) : gcc temp.c -o temp

Sish@DESKTOP-P86902Q( /home/andy ) : ./temp
hello to my friends
```

결론

SiSH 는 Linux 환경에서 기본 명령어들과 Custom 명령어들을 실행 할 수 있는 Shell Program 이다. 본 프로젝트는 Custom 명령어들의 기능들을 통해 개성 있는 Shell Program으로 설계 하였으며, Linux 기본 명령어에 한해서는, Redirection, Pipe, 복합 Redirection 및 Pipe 명령어 기능을 지원하는 다양한 기능을 가진 Shell Program으로 설계 할 수 있었다. 이러한 여러 명령어 및 기능을 구현하면서 Linux C programming 에 대해 쉽게 입문 할 수 있었으며, Parent Process와 Child Process 를 구현하며 fork, execve, wait 과 Process 에 대한 이해를 높일 수 있었다. 하지만, 현재 구현된 파이프는 두개의 명령어만 처리 할 수 있으며, Pipe와 Redirection 이 2개이상 초과 하는 기능을 제공하지 않아 향후 다중 파이프를 구현하는 숙제가 남아있다. 본 프로젝트를 마무리 하며, 다중 파이프와 여러 좋은 기능을 향후 프로젝트를 통해 구현하고자 한다.

[Ubuntu Linux 에서 작업]



- **makefile**이 작동하지 않는다면, 다음 Command를 입력해주세요.

```
gcc explore.c -o explore
```

```
gcc devproc.c -o devproc
```

```
gcc fedit.c -o fedit
```

```
gcc sish.c -o sish
```

```
./sish
```

- **makefile** 작동 시 : makefile extension 이 있어야 Makefile이 작동

```
make
```

```
./sish
```

