

임베디드 리눅스 시스템 프로그래밍

Chapter3-1

리눅스 원격접속

챕터의 포인트

- 터미널의 이해
- 원격접속의 중요성
- 원격접속 준비
- 원격접속 프로토콜
- 포트 포워딩
- 원격접속 실습

터미널의 이해

목표

가상 머신에 설치한 리눅스에 원격 접속을 하기 전에 터미널에 대해 학습한다.

터미널이란?

Terminal

컴퓨터에 접속하기 위한

Text 기반 **장치**

사실 터미널은 H/W 였다.

컴퓨터

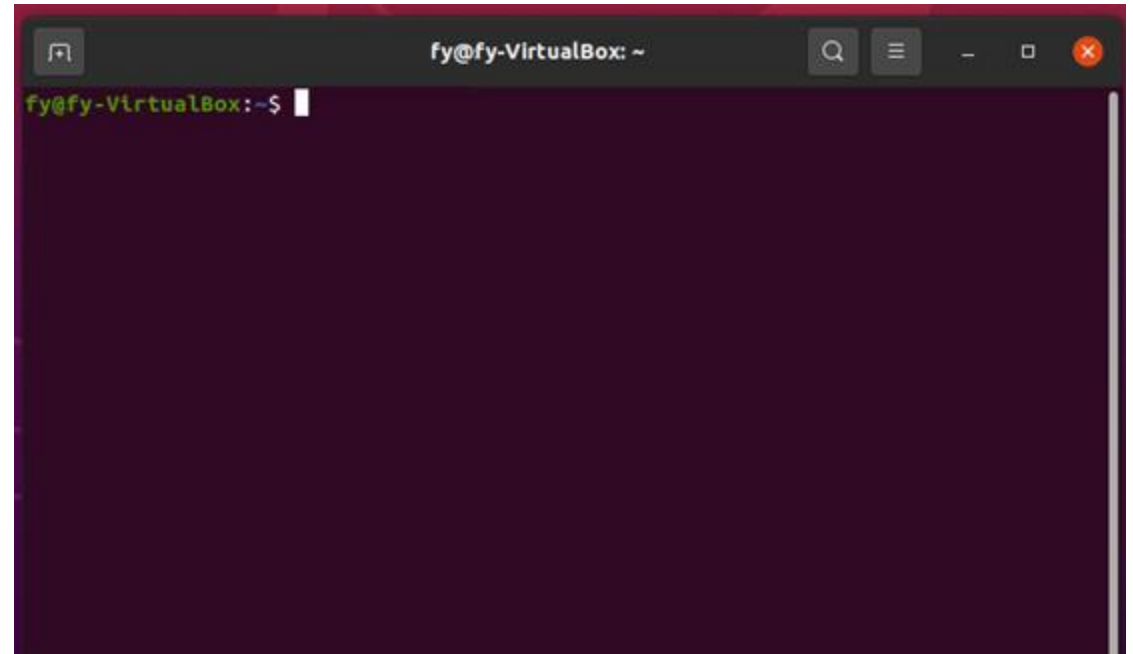
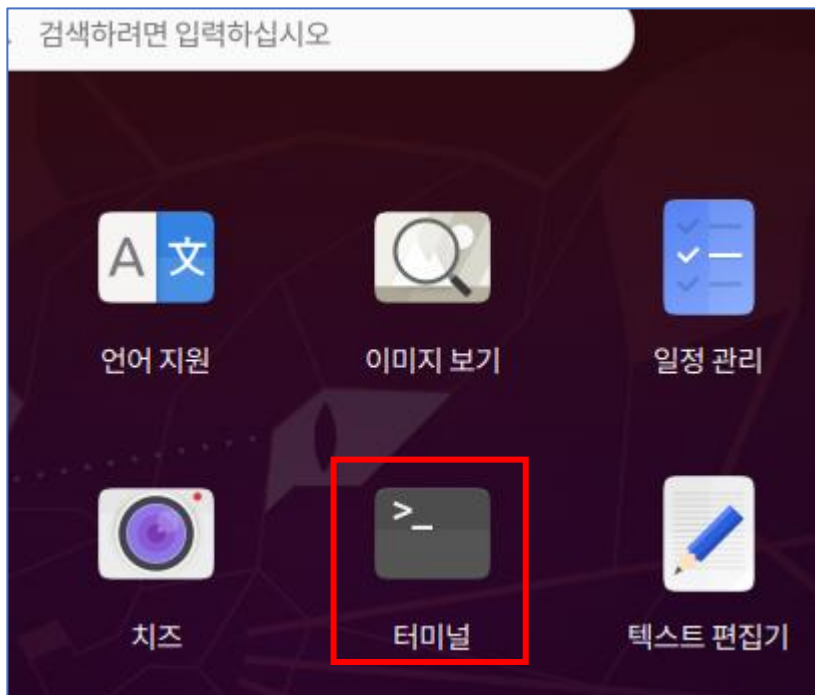


1973 NOVA 2 컴퓨터

터미널 창

H/W 였던 터미널이 남아, S/W 로 구현된 것

H/W 였던 그 시절과 마찬가지로 사용자는 터미널을 이용해 명령을 내린다.
터미널 or 콘솔이라고 부른다.

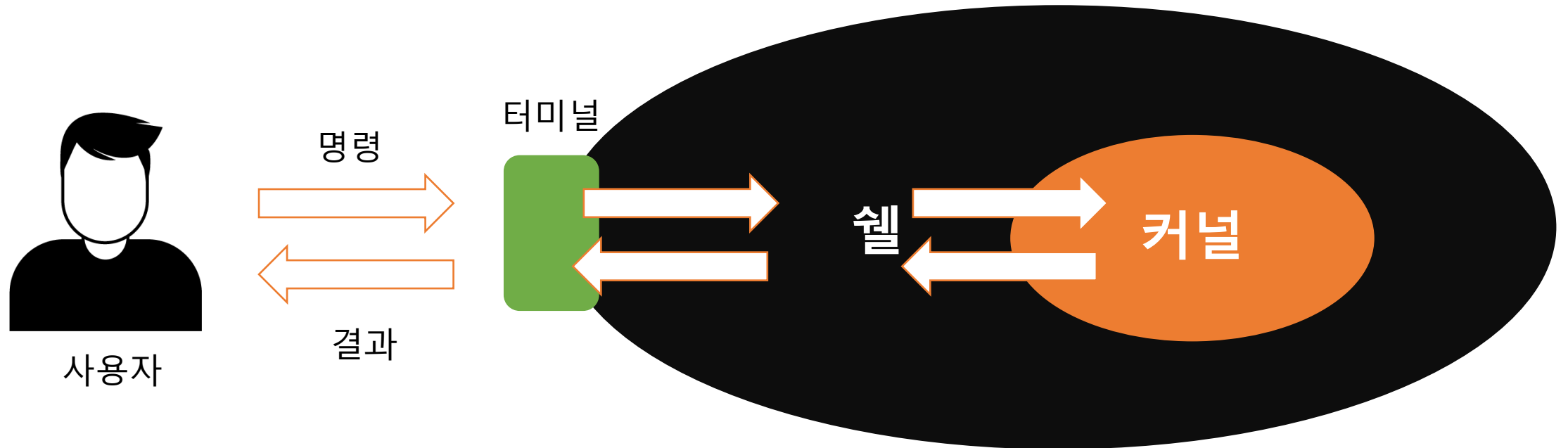


터미널 창을 통해 명령을 내린다.

터미널을 이용한 명령의 동작 과정이다.

사용자는 터미널 창을 열어 명령어를 입력한다.

입력된 명령어는 터미널을 통해 Shell 로 전달된다.

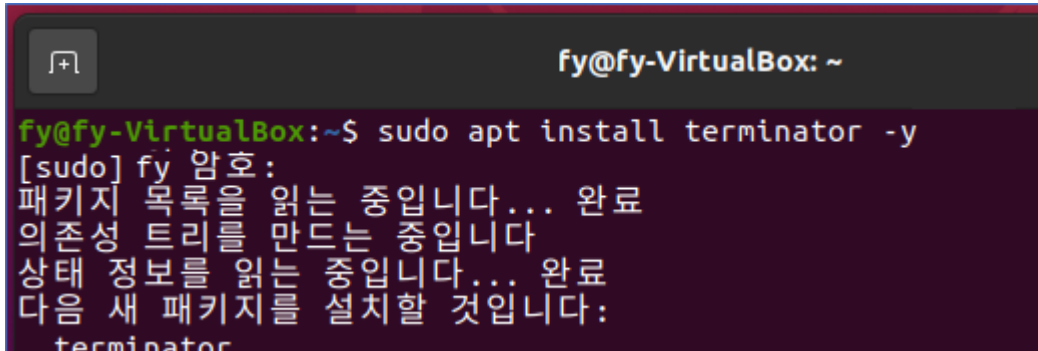


터미널 에뮬레이터 설치하기

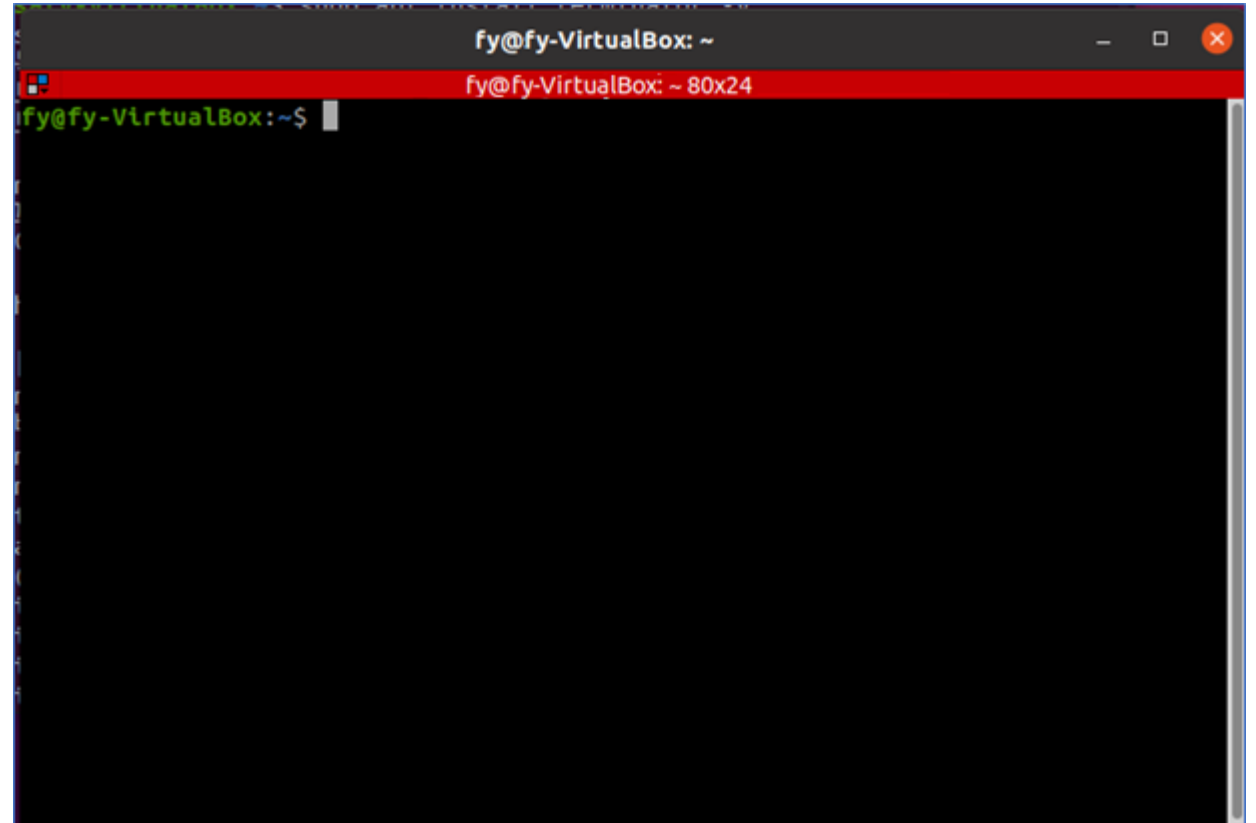
GUI 환경에서 훨씬 사용하기 편한 Terminator 에뮬레이터가 있다.

```
sudo apt install terminator -y
```

리눅스 개발자들이 흔히 사용하는 툴이다.

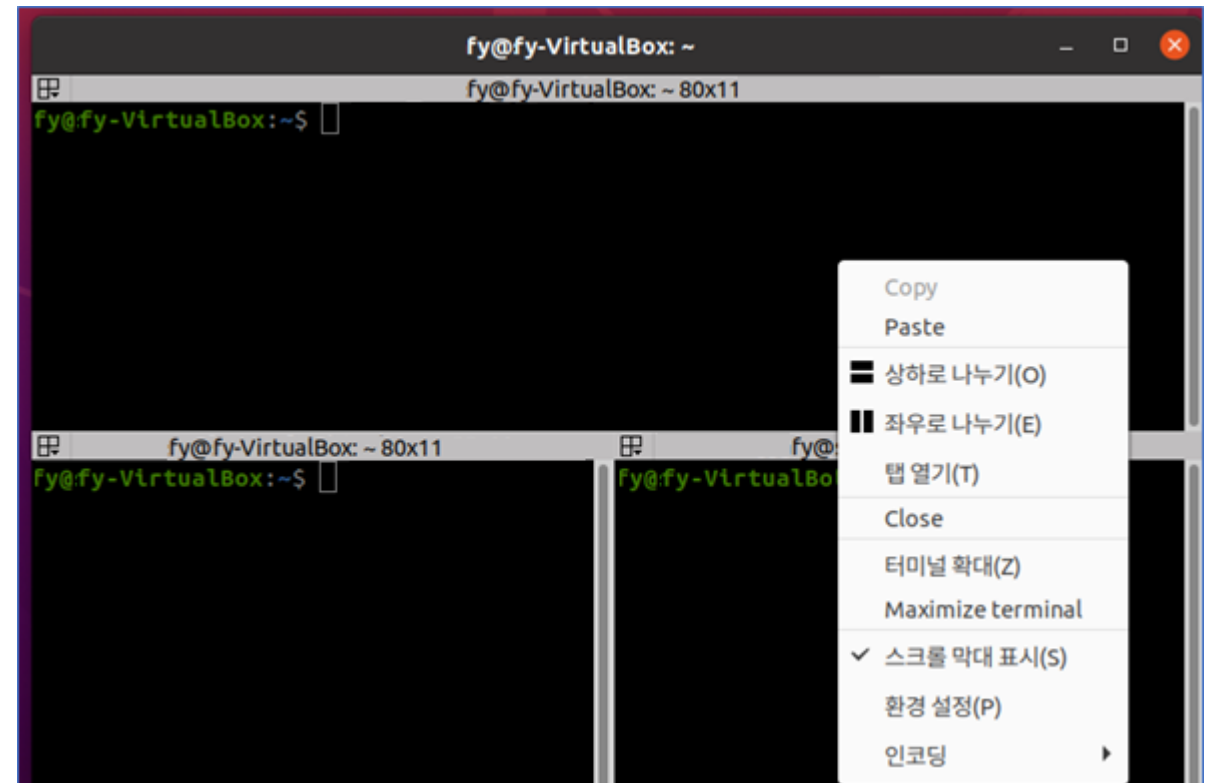


```
fy@fy-VirtualBox: ~  
fy@fy-VirtualBox:~$ sudo apt install terminator -y  
[sudo] fy 암호 :  
패키지 목록을 읽는 중입니다... 완료  
의존성 트리를 만드는 중입니다  
상태 정보를 읽는 중입니다... 완료  
다음 새 패키지를 설치할 것입니다:  
terminator
```



Terminator 활용 - 1

터미네이터는 창 분할 기능이 있다.
마우스 우클릭



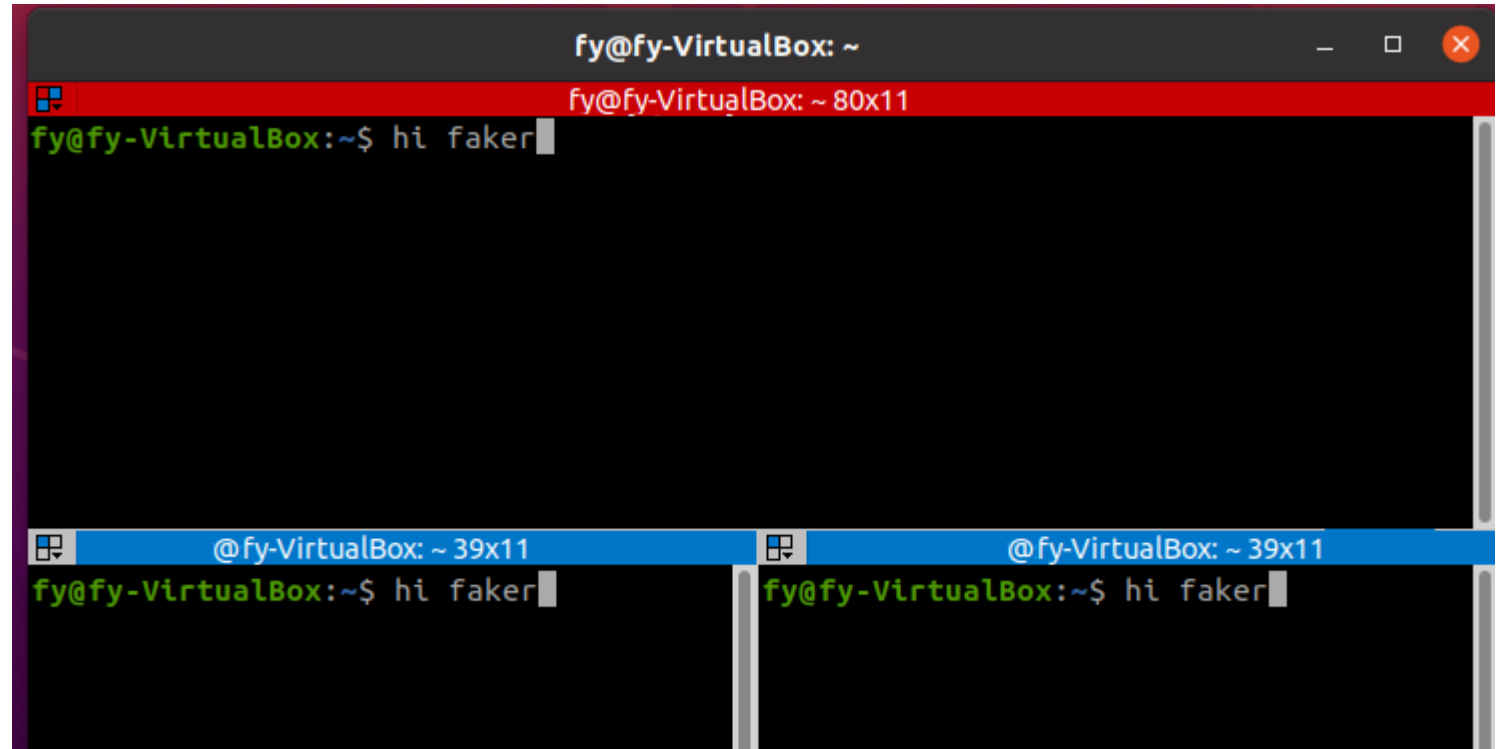
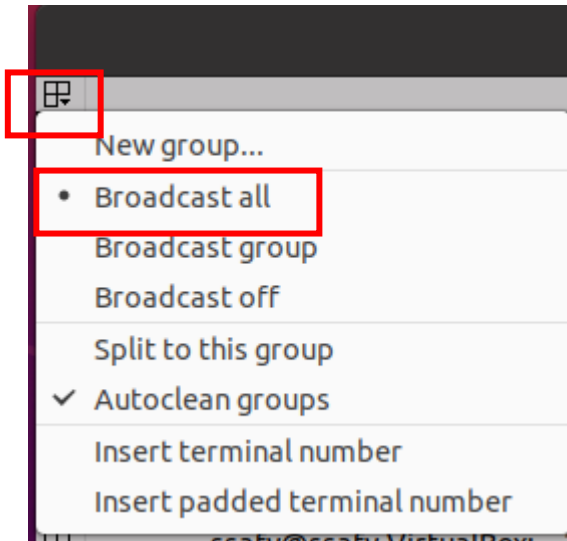
Terminator 활용 - 2

터미네이터는 동시 입력 기능도 있다!

창 분할 후 창 좌측 상단 클릭

Broadcast all 클릭

여러 서버 또는 시스템 관리 시 유용



원격접속의 중요성

목표

임베디드 SW 개발자의 업무 환경에 대해 이해하고 원격 접속의 중요성을 이해한다.

원격접속을 왜 해야 할까?

회사에서 집에 있는 PC에 원격 접속을 해서
집에서 작업한 결과물이 필요할 수도 있다.

반대의 경우도 마찬가지이다.

항상 보안에 유의하자.

임베디드 SW 개발자의 업무 환경

SW 개발자인 여러분은 사무실에서 개발을 하고 있다.

따뜻한 햇살

향긋한 커피

~~변함없는 출퇴근 지옥철~~

~~모든 것을 알고 있는 회사 복사기~~



임베디드 시스템의 위치

임베디드 시스템은 정말 어디에 있을 지 알 수 없다.

또한, 임베디드 시스템은 직접 키보드를 꺾을 환경이 안될 수 있다.

그렇기 때문에 임베디드 SW 개발자로서 원격 접속은 매우 중요하다.

원격접속 준비

목표

원격 접속을 하기 위한 준비를 한다.
mobaXterm 을 이용해 원격 접속을 한다.
포트 포워딩을 진행한다.

원격지에 있는 PC에 접속한다.

접속하고자 하는 PC 는 항상 켜져 있어야 한다.

네트워크에 연결되어 있어야 한다.

IP 주소를 알아야 한다.

- 네트워크로 접속

id 와 password 를 알아야 한다.

- 원격지 PC에 설정된 사용자 계정으로 접속
- 원격 접속용 터미널 프로그램이 필요하다.

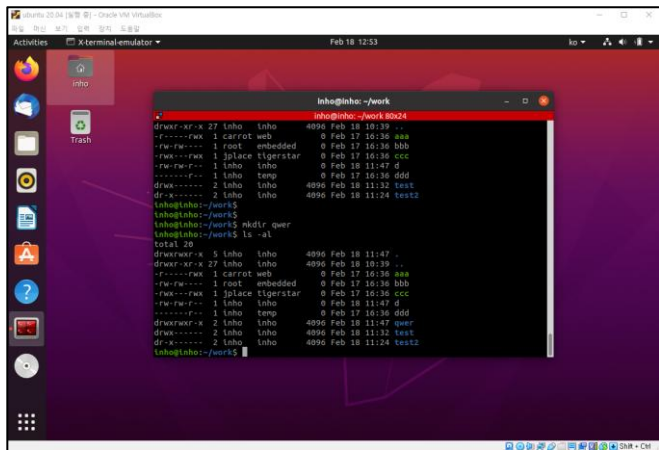
원격 접속하는 방법은 서버 접속하는 방법과 유사하다

서버와 유사한 원격 접속

우리가 사용하고 있는 PC → 개발 PC, Client 가 된다.

원격지에 있는 임베디드 시스템 장치 → Server 가 된다.

client 와 server 각각 준비가 필요하다.



제주도에 있는 임베디드 시스템
(Server)



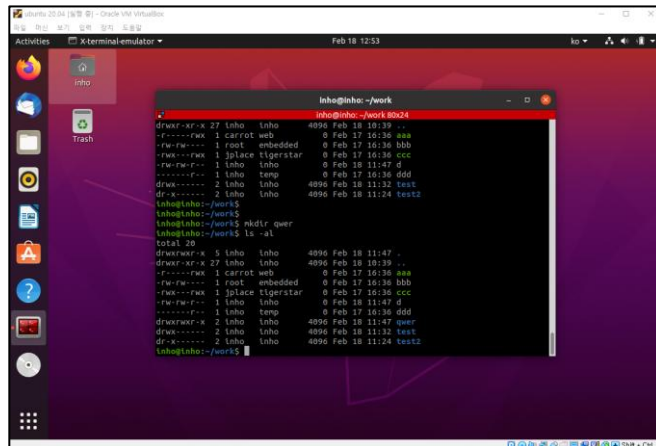
개발 PC
Client

원격 접속용 터미널 프로그램 (CLI)

Client PC는 원격지 PC에 접속하기 위해 터미널 프로그램이 필요하다.

CLI 기반 원격 접속용 터미널 프로그램 : MobaXterm
다양한 CLI 기반 터미널 프로그램이 있다.

우리는 CLI 기반 터미널 프로그램을 사용한다!



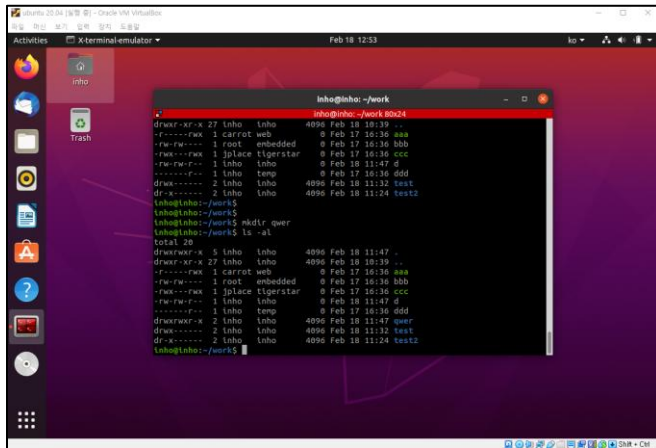
제주도에 있는 임베디드 시스템
(Server)



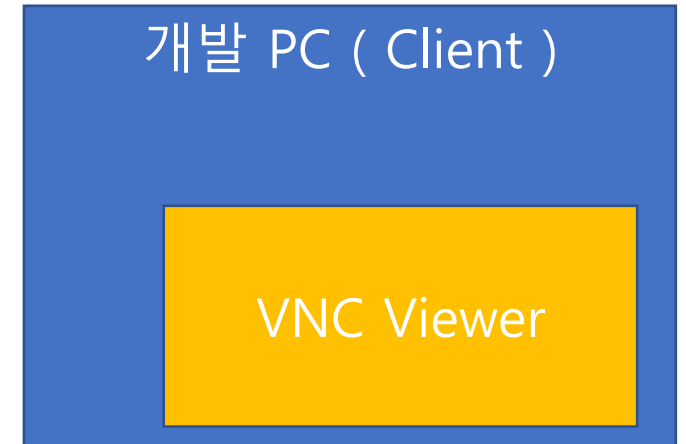
원격 접속용 터미널 프로그램 (GUI)

Client PC는 원격지 PC에 접속하기 위해 터미널 프로그램이 필요하다.

GUI 기반 원격 접속용 터미널 프로그램 : VNC Viewer
라즈베리파이 수업 때 실습한다.



제주도에 있는 임베디드 시스템
(Server)



CLI 기반 터미널 프로그램으로 사용하는 이유

임베디드 리눅스에서 GUI 원격 접속이 안되는 경우가 많다.

임베디드 리눅스 : PC보다 저사양 장치에서 동작하는 경량 리눅스

다양한 CLI 기반 터미널 프로그램이 있다.

Putty : 가장 널리 사용됨

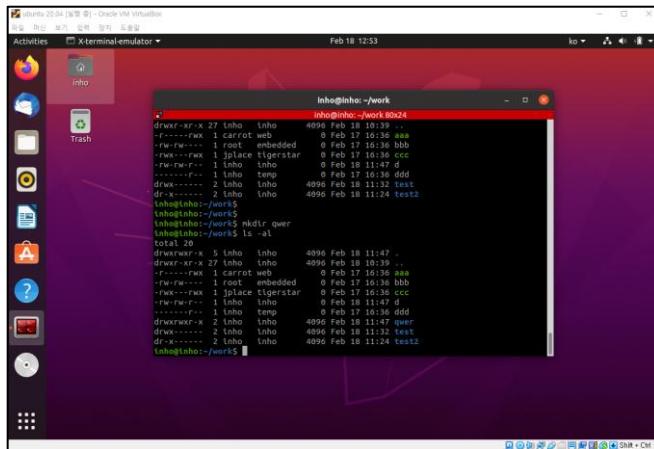
MobaXterm : 편리한 오픈소스 터미널

우리 수업에서는 MobaXterm 으로 접속한다.

원격지 PC는 서버로써 동작해야 한다.

서버 프로그램이 필요하다.

서버 프로그램이 되어 원격 접속 시 파일 관리, 보안 강화 등의 역할을 한다.



제주도에 있는 임베디드 시스템
(Server 프로그램 필요)

원격접속

개발 PC (Client)

MobaXterm

원격 접속 환경

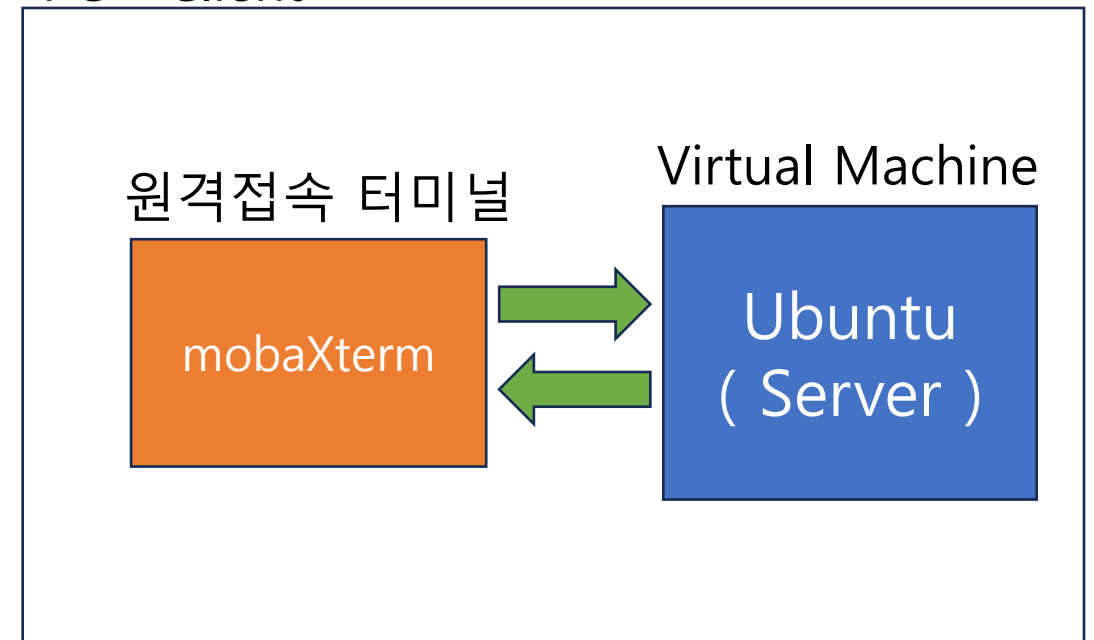
VM에 설치된 Ubuntu : 원격 접속할 PC, Server

원격 접속을 위한 server 프로그램 설치가 필요하다

Windodw : 개발 PC, Client

원격 접속용 터미널 프로그램 설치가 필요하다.

PC - Client



Client 준비

PC에 mobaXterm을 설치한다.

설치는 간단하다.

<https://mobaxterm.mobatek.net/download.html>

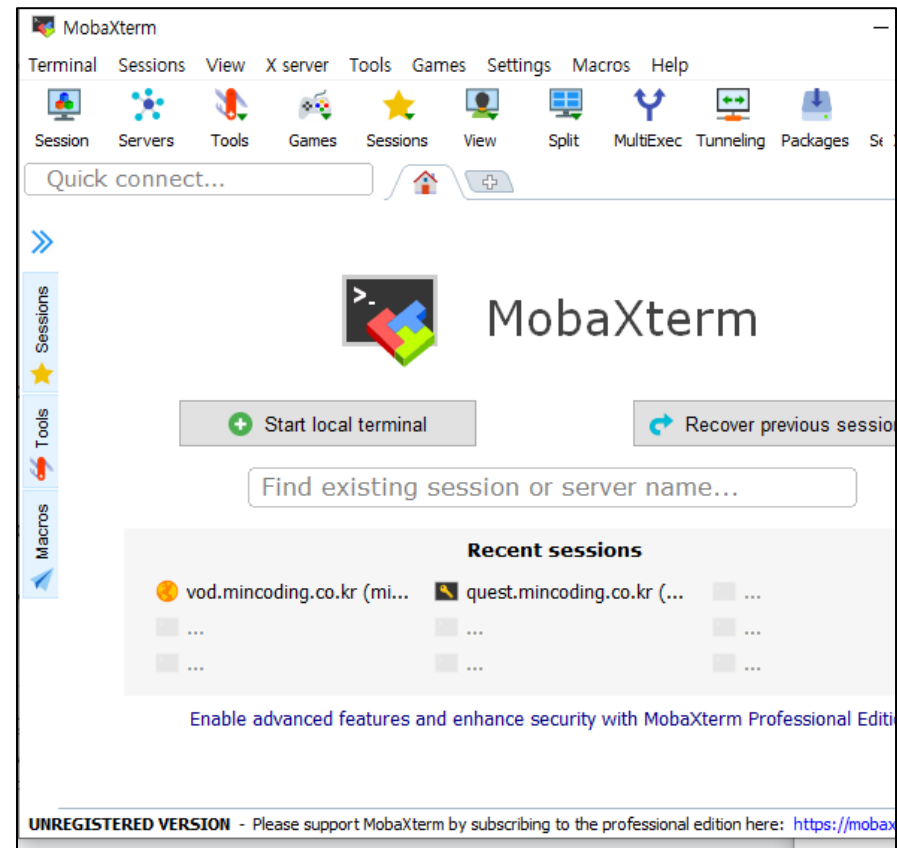
다운 받아서 설치한다.

설치가 완료 되면 실행한다.

Portable edition : 설치 없이 사용가능

Installer edition : 설치 후 사용 가능

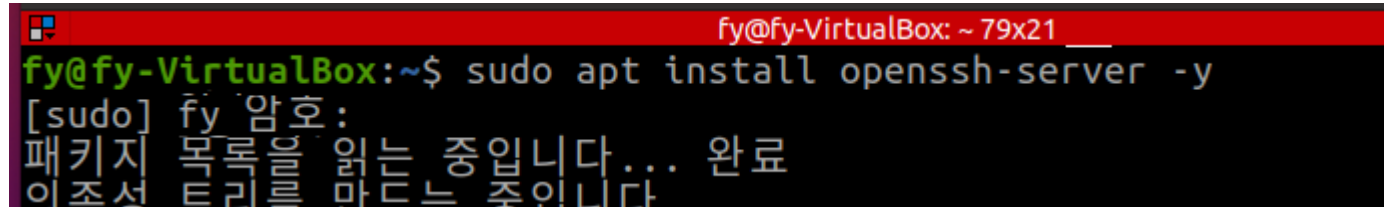
*둘 다 같은 기능을 한다



Server 준비

Ubuntu 에 서버 프로그램을 설치한다.

```
sudo apt install openssh-server -y
```

A terminal window with a red title bar showing the command 'sudo apt install openssh-server -y' being executed. The output shows the password prompt, confirmation of package installation, and completion of the process.

```
fy@fy-VirtualBox: ~ 79x21
fy@fy-VirtualBox:~$ sudo apt install openssh-server -y
[sudo] fy 암호:
패키지 목록을 읽는 중입니다... 완료
이 조서 트리를 만드는 중입니다
```

설치하자마자 자동으로 실행한다.

윈도우의 서비스 개념과 같다.

백그라운드에서 실행되고 있다.

- 예시 : 보안프로그램, 방화벽

리눅스에서는 백그라운드에서 실행되는 프로그램을 "시스템 데몬" 이라 한다.

[참고] 시스템 데몬 상태 방법

눈으로 봐야 믿을 수 있다.

`sudo systemctl status ssh`

systemctl : 시스템컨트롤

눈에 보이지 않고, 뒤에서 동작하는 프로그램이라, 시스템컨트롤 명령어로 동작을 확인한다.

q 누르면 종료

```
fy@fy-VirtualBox: ~  
fy@fy-VirtualBox: ~ 99x2  
● ssh.service - OpenBSD Secure Shell server  
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)  
   Active: active (running) since Sun 2024-03-17 15:07:16 KST; 1min 45s ago  
     Docs: man:sshd(8)  
           man:sshd_config(5)  
   Main PID: 734 (sshd)  
     Tasks: 1 (limit: 4579)  
    Memory: 2.3M  
    CGroup: /system.slice/ssh.service  
            └─734 sshd: /usr/sbin/sshd -D [listener] 0 of 1024...
```

원격접속 프로토콜

목표

원격 접속을 위한 다양한 프로토콜이 존재한다.
프로토콜에 대한 간략한 설명을 듣고 원격 접속을 한다.

서로 다른 두 장치와 시스템이 통신을 하기 위한 규칙

앞으로 프로토콜이라는 말을 듣게 되면, 통신 규칙이라고 보면 된다.

당연히, 같은 프로토콜을 사용해야 통신이 가능하다.

- 다른 프로토콜을 사용하는 경우는 말하지 않겠다.

1루수

Server 와 Client는 어떤 프로토콜을 사용할 지 정해야 한다.

셸 접속용 프로토콜 : 원격 접속용 프로토콜

1. telnet : 원격 접속용 표준 프로토콜 중 하나, 암호화 안됨, 요즘은 권장하지 않음
2. ssh : 원격 접속 및 안전한 파일 전송용 프로토콜, 데이터 암호화, 보안 강화
 - 파일 전송도 가능하다. → SSH File Transfer Protocol SFTP 라고 한다.

파일 전송용 프로토콜 : 파일 전송을 위한 프로토콜

1. FTP (File Transfer Protocol) : 파일 전송용 표준 프로토콜, 암호화 안됨
2. SFTP (Secure File Transfer Protocol) : 보안 강화용 파일 전송 프로토콜

ssh를 이용해 원격 접속도 가능하고, 파일 전송도 가능하다!

아까 설치한 서버 프로그램이 바로 ssh 서버 프로그램이다!

포트 포워딩

목표

포트에 대해 학습하고 가상 머신에 설치된 우분투를 포트 포워딩한다.

오늘도 우리는 PC에서 네트워크를 이용해 다양한 작업을 한다.

PC에서 youtube 라이브를 본다.

SNS로 채팅을 한다.

음악을 듣는다.

네트워크로 들어온 다양한 정보가 각각의 프로그램에 전달될 수 있도록 하는 것이 바로 포트다!

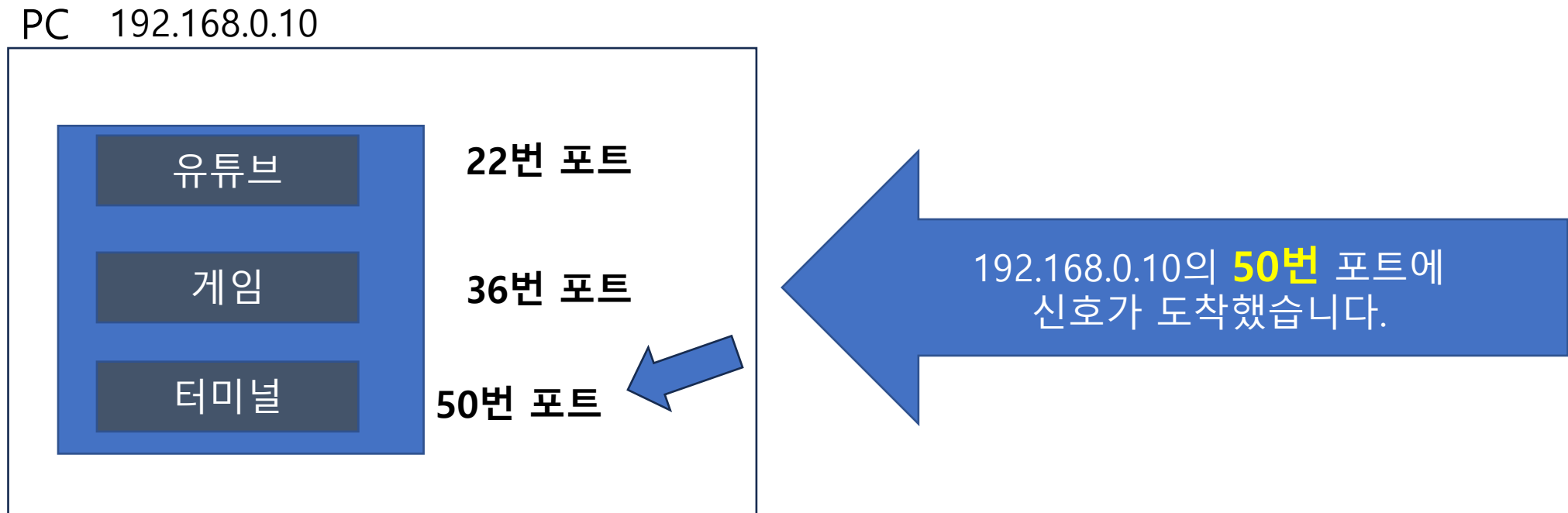
그림으로 이해하기

ip주소를 이용해 데이터가 도착했다

데이터를 각각의 프로그램에 전달해야 한다.

그때 사용하는 것이 바로 포트!

ip뿐 아니라 포트 번호도 알아야 통신이 가능하다.



자주 사용되는 포트 번호

22번 포트

ssh, sftp 프로토콜을 쓸 때 자주 사용되는 포트 번호

23번 포트

telnet 프로토콜을 쓸 때 자주 사용되는 포트 번호

80번 포트

http 프로토콜을 쓸 때 자주 사용되는 포트 번호

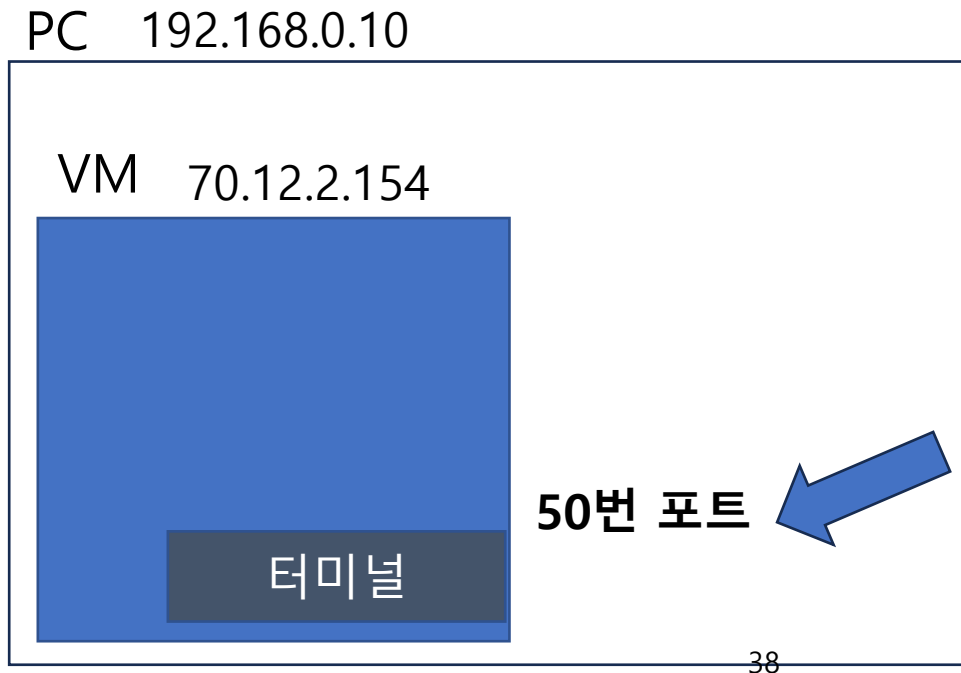
국제관리기구에서 정한 포트 번호이다.
well-known port number 라고 하며,
권장사항이다.

포트 포워딩 Port Forwarding

포트를 일치 시키는 것을 포트 포워딩이라 한다.

ex) ip 주소 192.168.0.10 을 가진 PC의 50번 포트에 들어오는 데이터를
ip 주소 70.12.2.154 을 가진 VM의 22번 포트에 보내!

외부에서 접속을 허용하기 위해 포트포워딩이 필요하다!



192.168.0.10의 **50번** 포트에
신호가 도착했습니다.

포트 포워딩 준비 - 1

윈도우에서 가상 머신 ip 확인하자.

cmd 창을 열고 (단축키 : 윈 + r)

ipconfig 명령어를 입력해서 가상머신의 ip 주소를 확인한다.

대부분의 경우 가상머신의 ip 주소는 192.168.56.1 로 고정이다.

```
이더넷 어댑터 이더넷 2:

   연결별 DNS 접미사 . . . . . : 
   링크-로컬 IPv6 주소 . . . . . : fe80::92cb:61ef:cc63:9089%9
   IPv4 주소 . . . . . : 192.168.56.1
   서브넷 마스크 . . . . . : 255.255.255.0
   기본 게이트웨이 . . . . . :
```

포트 포워딩 준비 - 2

Ubuntu 에서 ip 주소 확인하기

sudo apt install net-tools -y

ifconfig

10.0.2.15

```
fy@fy-VirtualBox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::d575:c1e1:5ec9:ae20 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:cf:80:7a txqueuelen 1000 (Ethernet)
    RX packets 702182 bytes 1011613423 (1.0 GB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 115407 bytes 8452163 (8.4 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

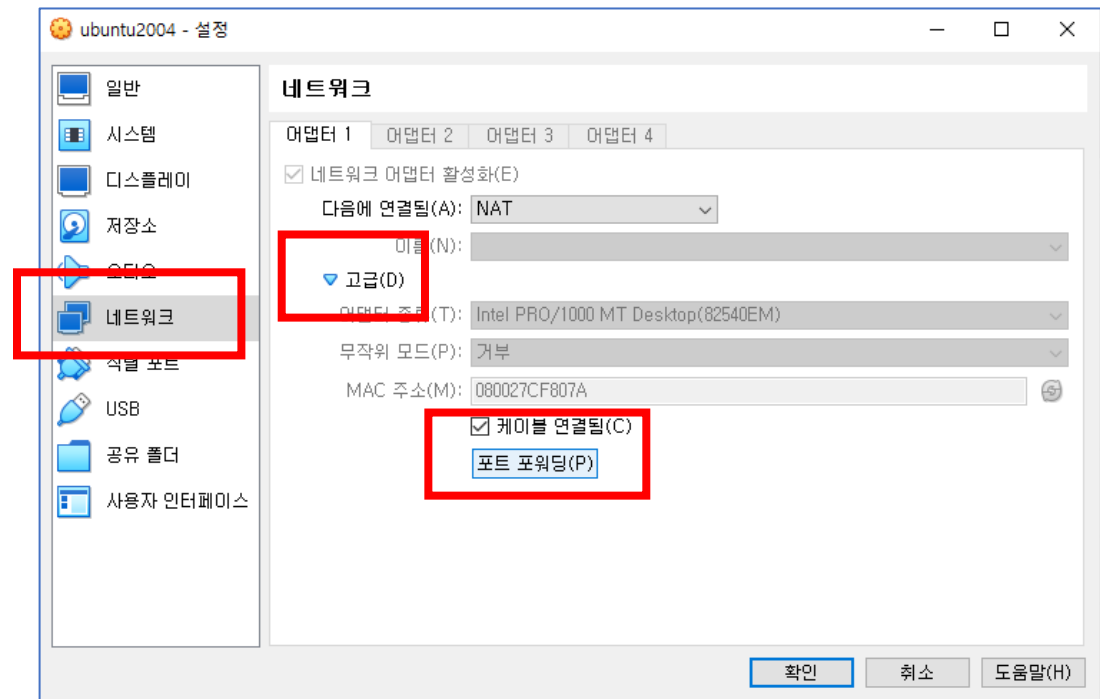
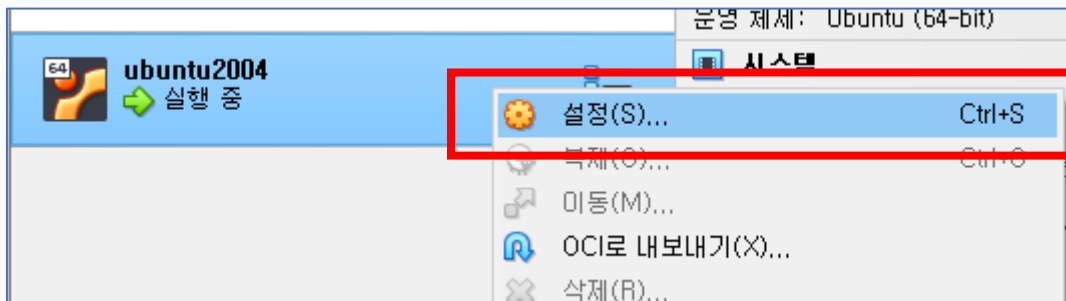

포트 포워딩 설정하기 - 1

Virtual Box 에서 설정한다.

실행 중인 우분투는 종료할 필요 없다.

마우스 우클릭 → 설정 클릭

네트워크 → 고급 → 포트 포워딩 클릭



포트 포워딩 설정하기 - 2

규칙 추가

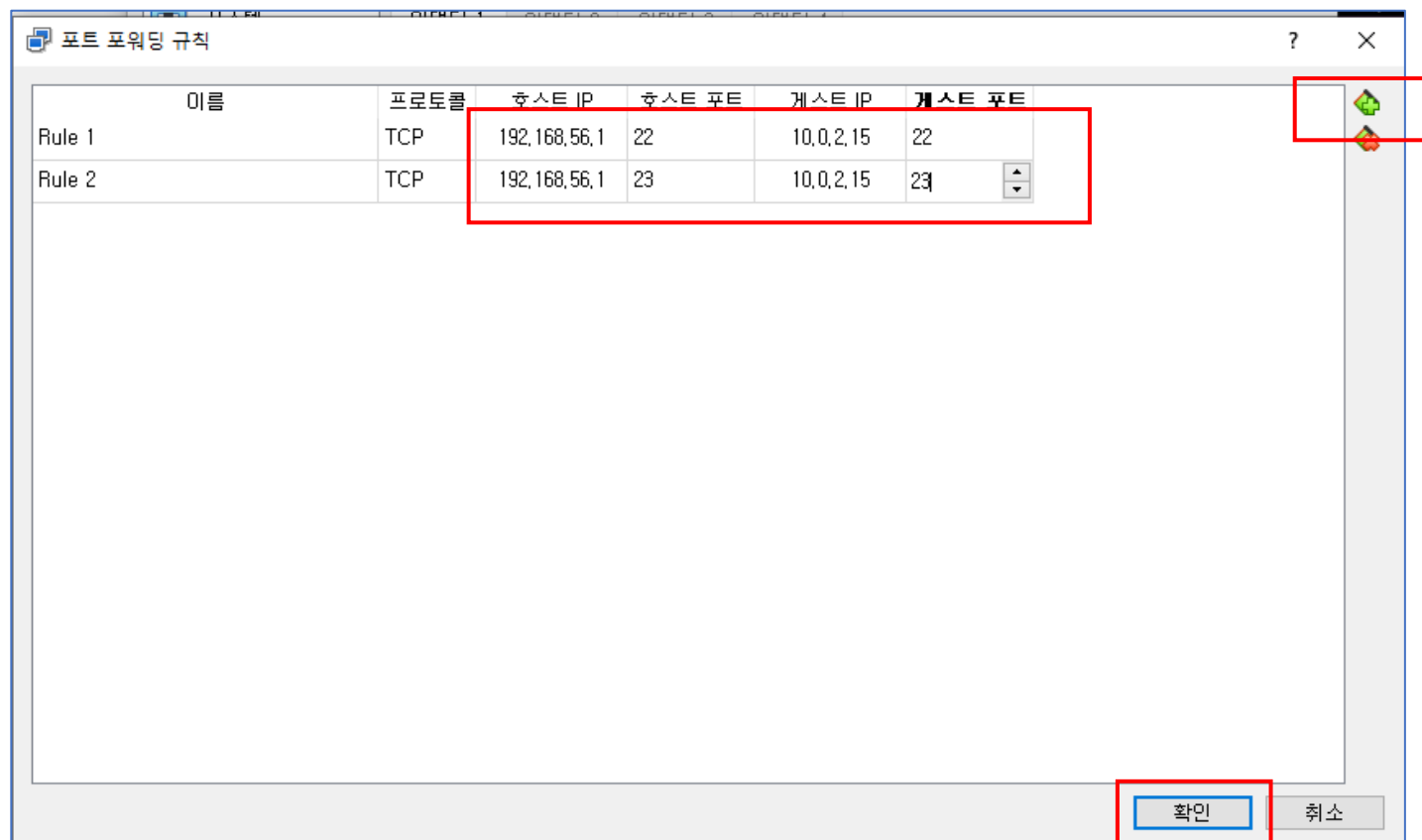
+ 클릭

22 / 23 포트 두 개의 규칙을 설정한다.

호스트 ip : 192.168.56.1

게스트 ip : 10.0.2.15

확인 클릭



원격접속 실습

목표

moshXterm 을 이용한 원격 접속을 완료한다.

moshXterm 을 이용해 파일 전송 및 클립 보드 공유를 테스트한다.

원격 접속하기 - 1

mobaXterm 에서 접속한다.

화면 좌측 상단 Session 클릭

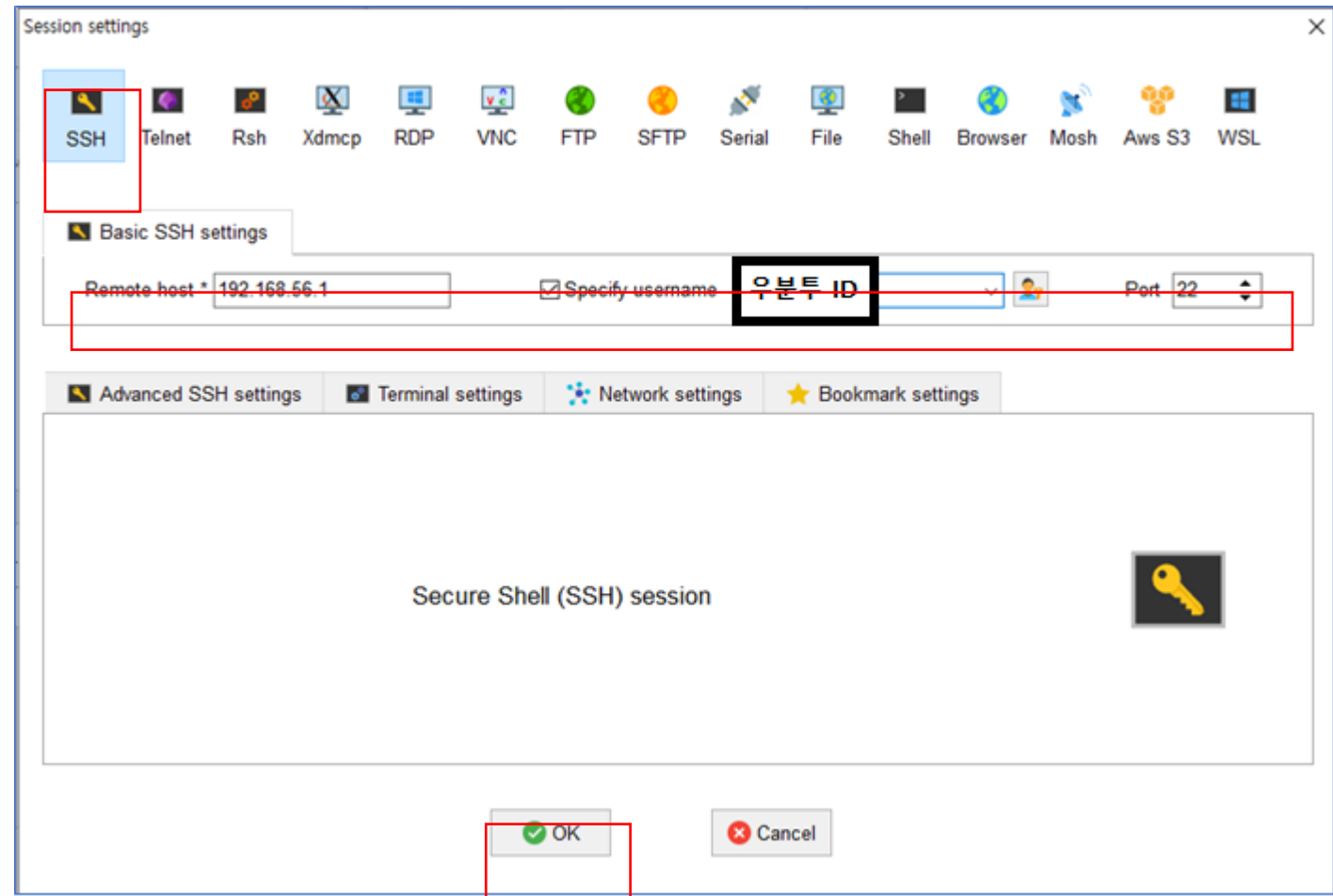
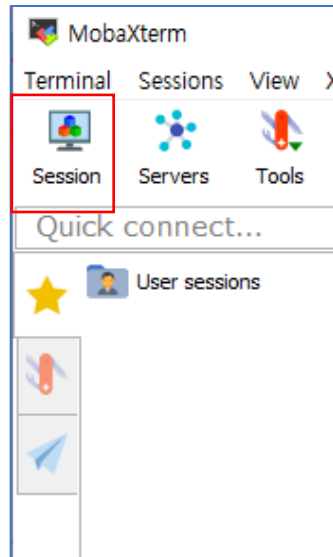
SSH 클릭

Remote host : 192.168.56.1

Specify user name : 아이디

Port : 22 (SSH)

OK 클릭

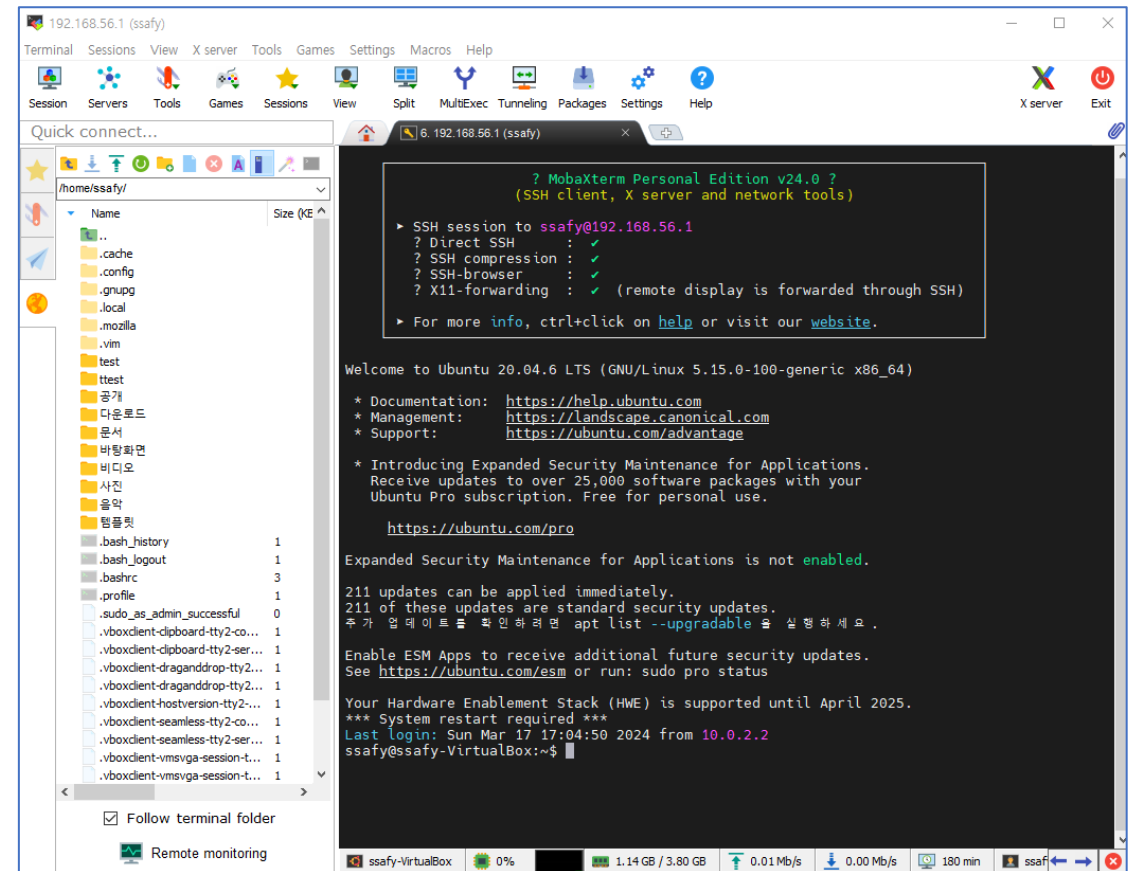
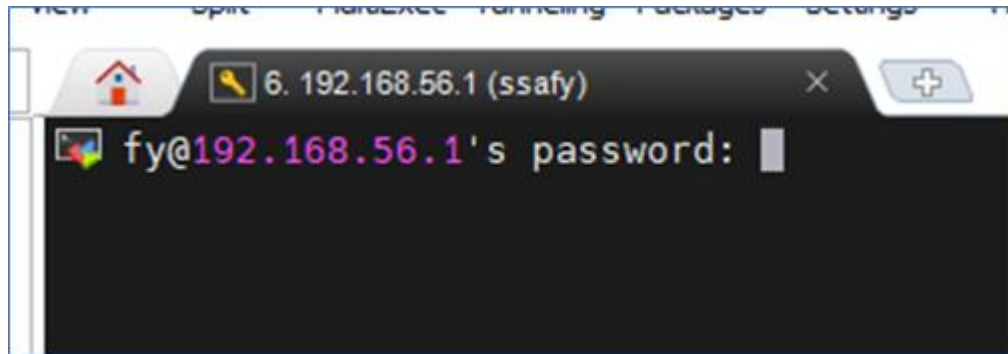


원격 접속하기 - 2

비밀번호 입력 : 1 엔터

리눅스는 보안 상의 이유로 몇 글자를 입력 했는 지조차 표시 하지 않는다.

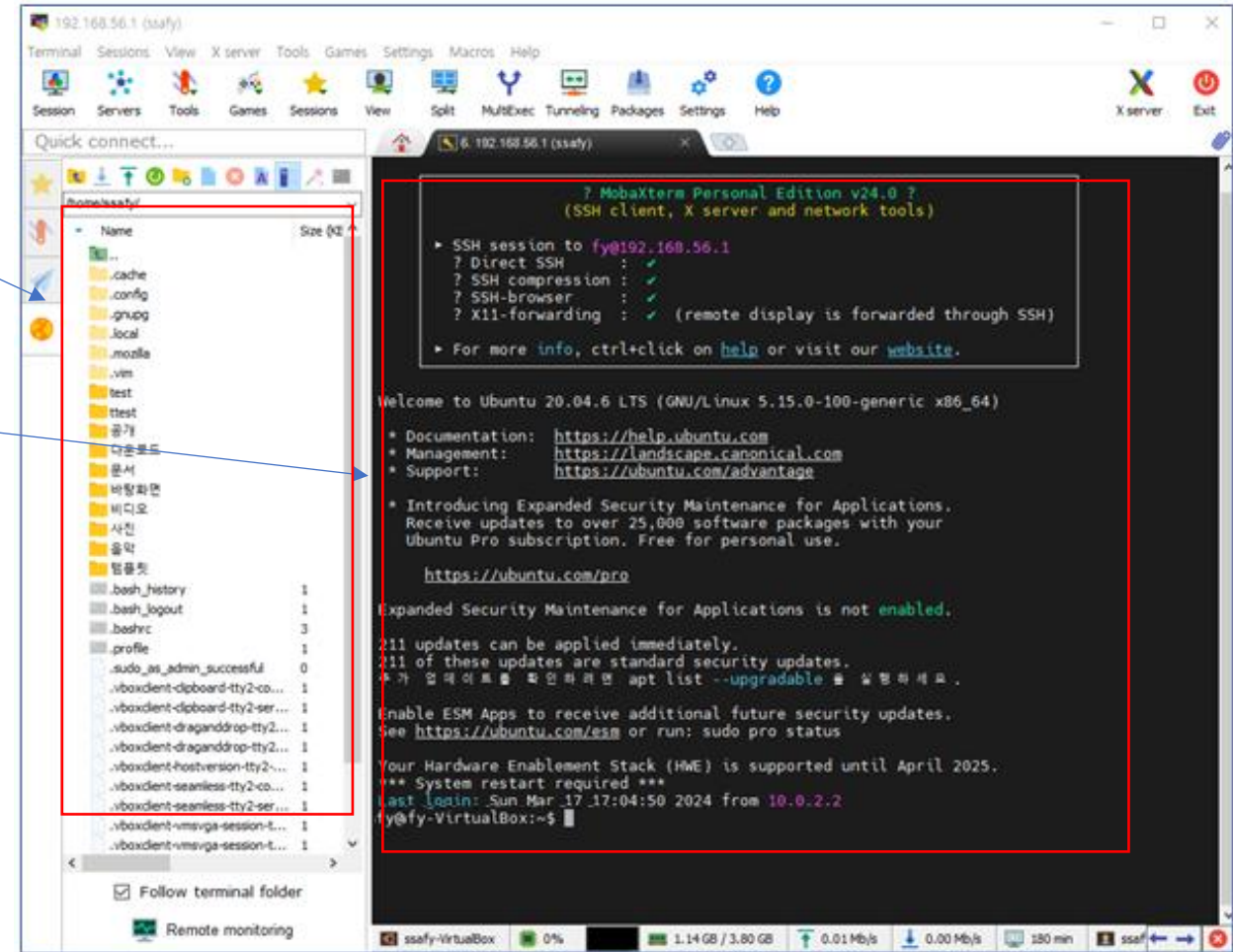
접속 성공!



Mobaxterm 소개

파일 시스템

터미널 창

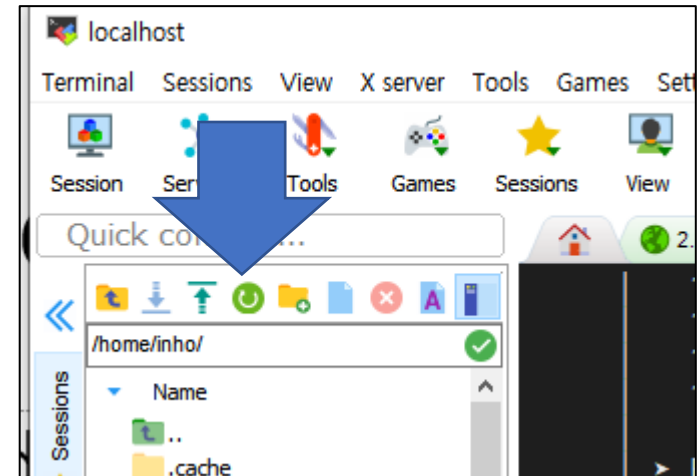


Mobaxterm 소개

새로고침

자동 업데이트가 느려

갱신을 직접 해줘야 하는 경우가 많다.

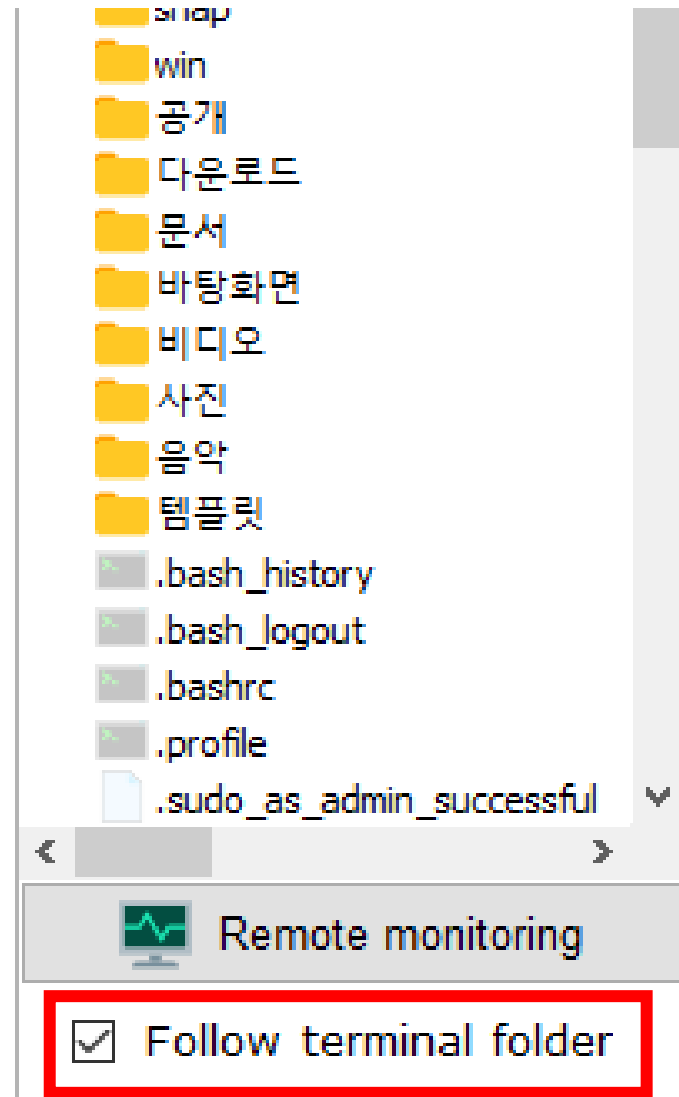


Mobaxterm 소개

Follow 옵션 체크 시

cd로 폴더 이동하면, 디렉토리 트리가 자동으로 이동된다
디렉토리 내 파일 쉽게 확인 가능 (ls 대체)

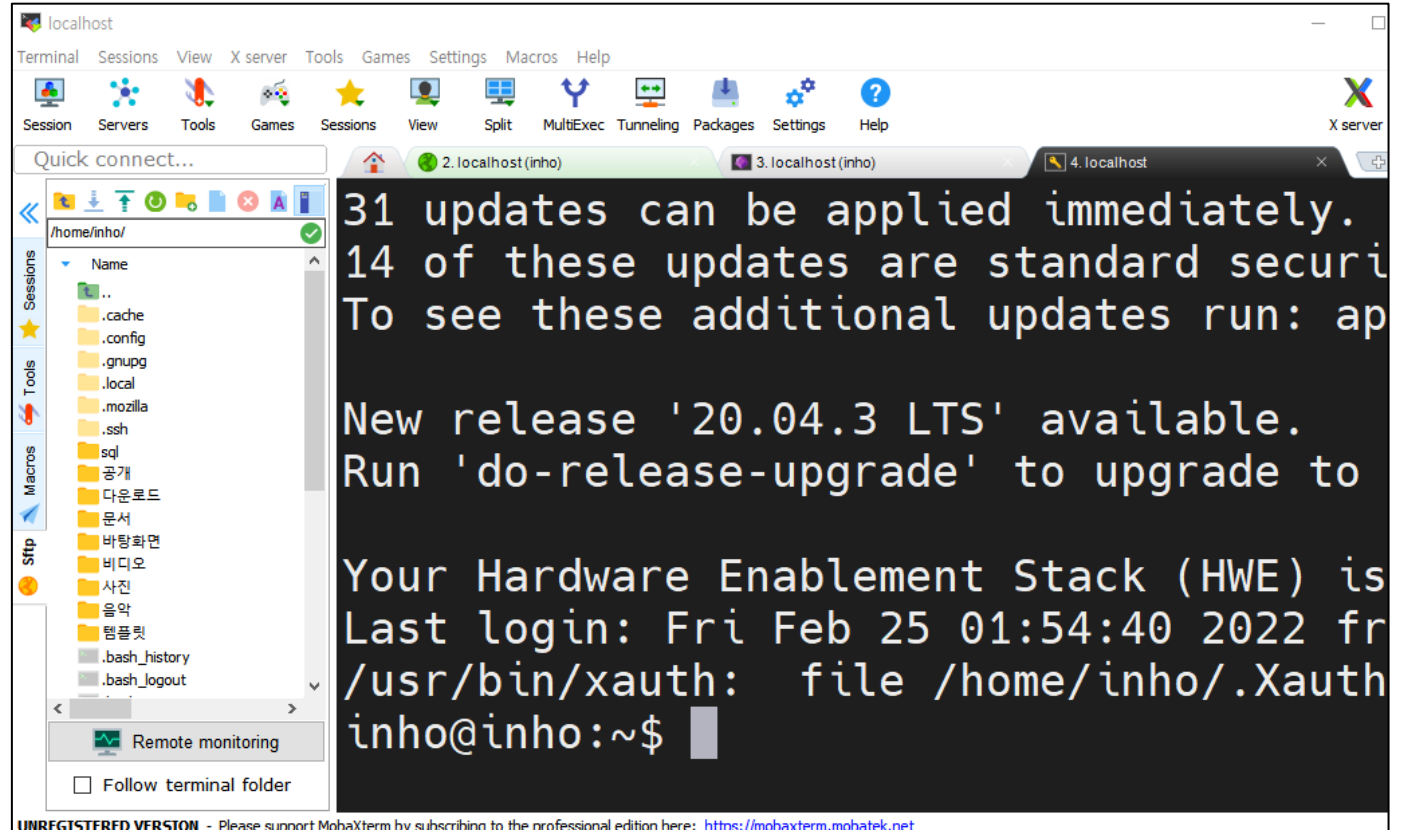
* 이동이 안 되면, 탭 종료 후 재 접속



Mobaxterm 소개

글씨 크기 변경

Ctrl + 마우스 Wheel

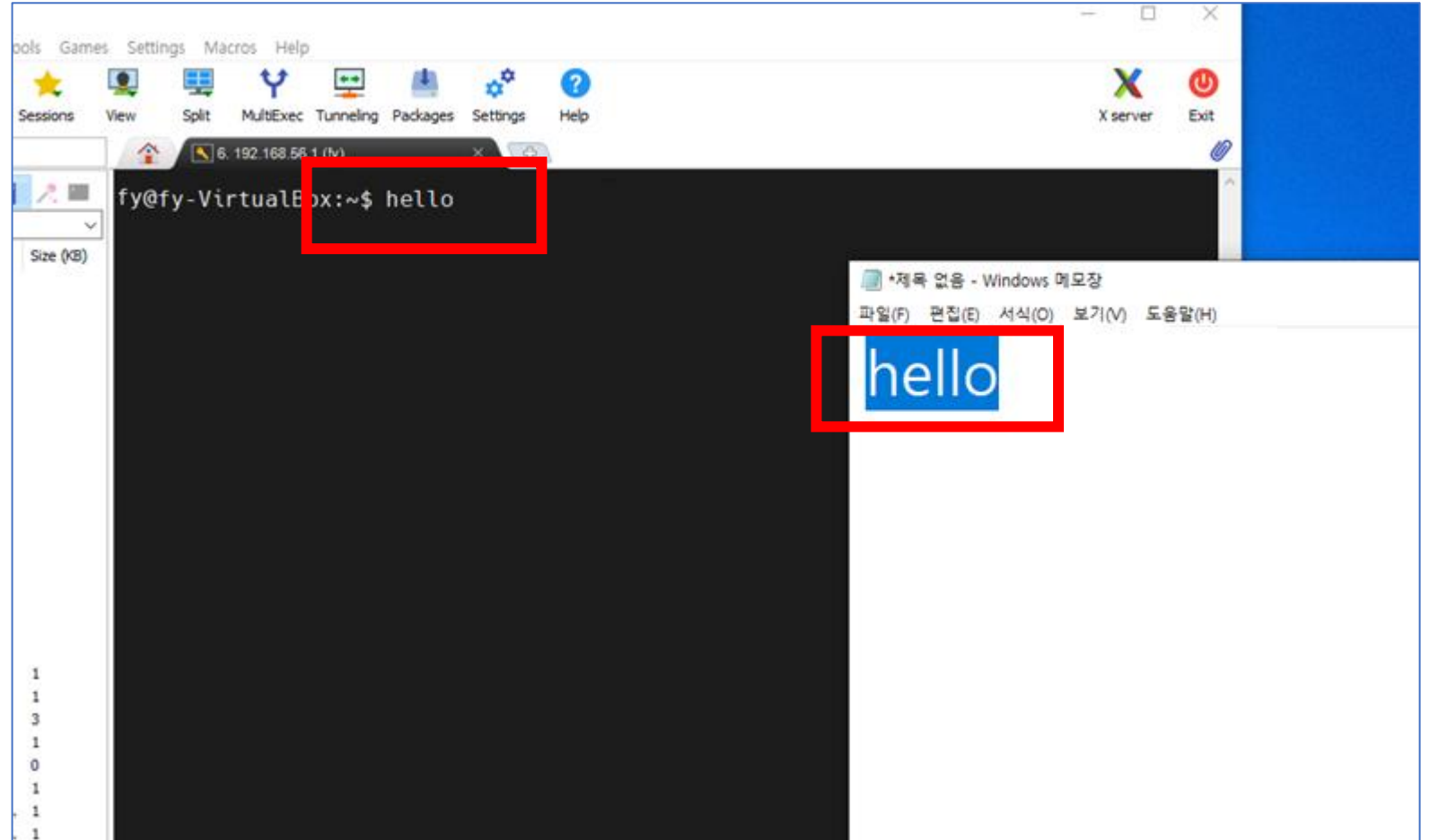


Mobaxterm 소개

클립보드 양방향 공유

당연히, 터미널에서 제공한다.

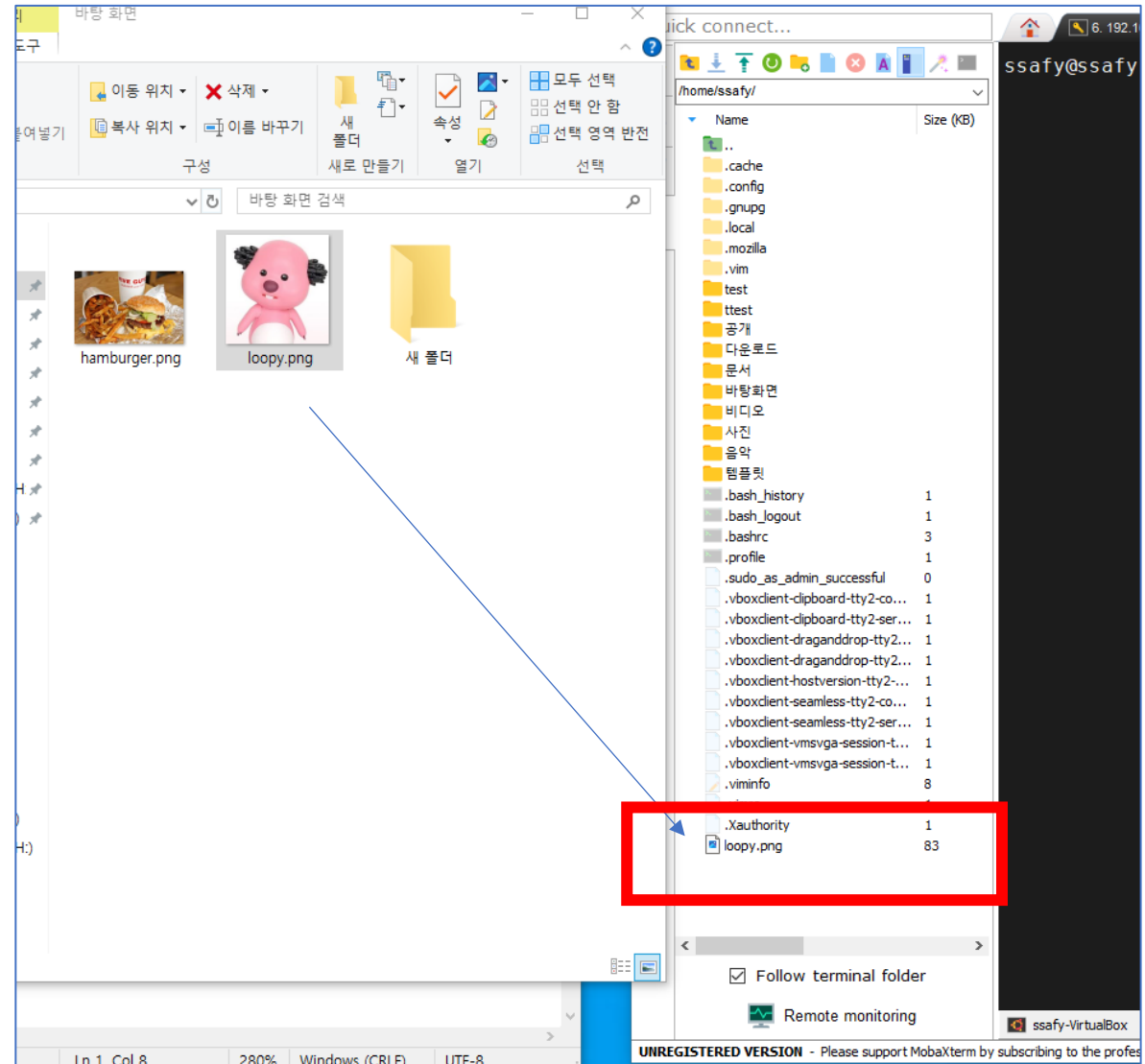
ctrl + Insert / shift + Insert



Mobaxterm 소개

파일 공유 (양방향)

드래그 & 드롭으로 공유가 편리하다.
사진 파일을 열어 볼 수 도 있다.



Chapter3-2

리눅스 사용자 및 권한 관리

챕터의 포인트

- Root 의 이해
- User 와 Group
- 리눅스 파일의 종류
- 파일의 권한
- 파일 권한 바꾸기

Root의 이해

챕터 목표

목표

리눅스 시스템에서 최고 권한을 갖는 root에 대해 학습한다.
root 권한에 대해 학습한다.

상식 QUIZ

네트워크에서

Host는 컴퓨터 주인을 뜻한다.

O / X ????

정답은 X

네트워크에 연결되어 있는 장치,
컴퓨터를 Host 라고 한다.

리눅스에서 Host 는?

리눅스는

다중 사용자 시스템으로 설계 되었다.

한 컴퓨터를 여러 명에서 사용함
서버용 OS로 많이 사용되는 이유

네트워크에 연결된 한 컴퓨터 개념으로

리눅스가 설치된 컴퓨터 한 대를 Host 라고 부른다.

여러분의 컴퓨터가 Host이다.

다중 사용자 시스템

여러 사용자가 하나의 PC를 사용한다.

우리 가족이 전체 쓰는 컴퓨터
각자 계정으로 로그인 가능

주인이 없으면 안된다.

→ 관리자 지정 필요

공용 프로그램들
설치 / 제거를 모두에게 허용할 순 없다.

user1 : fy



user2 : kim

user3 = lee



리눅스 시스템에서 관리자

최고 권한을 갖는다.

프로그램의 설치/삭제 관리

user 관리

다양한 권한을 갖는다.

user1 : fy



user2 : kim

user3 = lee



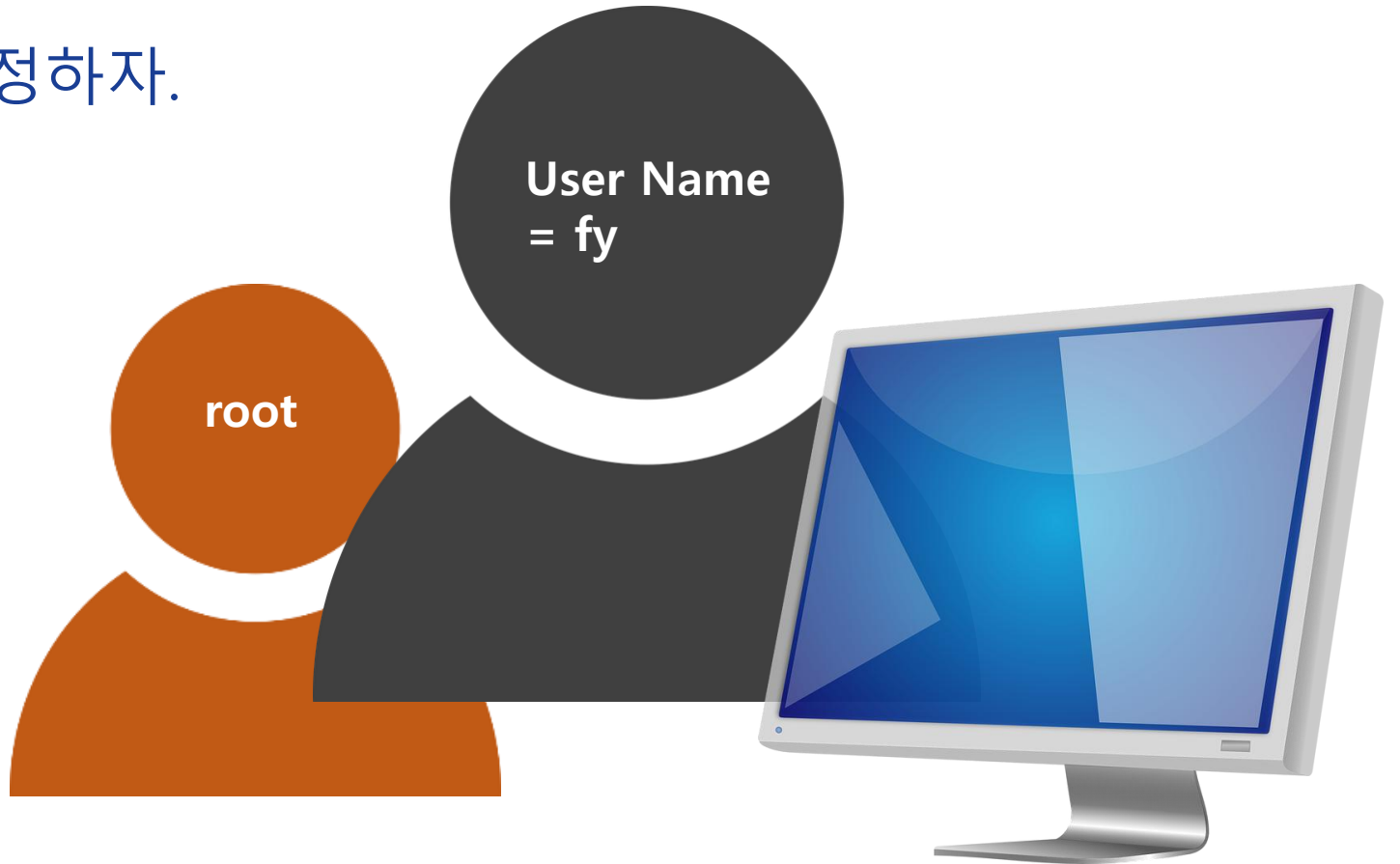
root



Root의 생성 시점

우리가 처음 리눅스를 설치할 때 사용자 계정을 만들었다.
교재에서는 "**fy**" 라는 이름으로
처음 리눅스를 설치했다고 가정하자.

리눅스 설치가 되면
사용자 계정(**fy**)과 동시에
root 계정도 같이 생성된다.



사용자 계정으로 어떻게 프로그램을 설치했을까?

사용자 계정은 프로그램 설치 / 삭제를 할 수 없다.

sudo 라는 키워드로 root의 권한을 빌려서 사용한다.

sudo 라는 키워드는 아무 사용자나 사용할 수 없다.

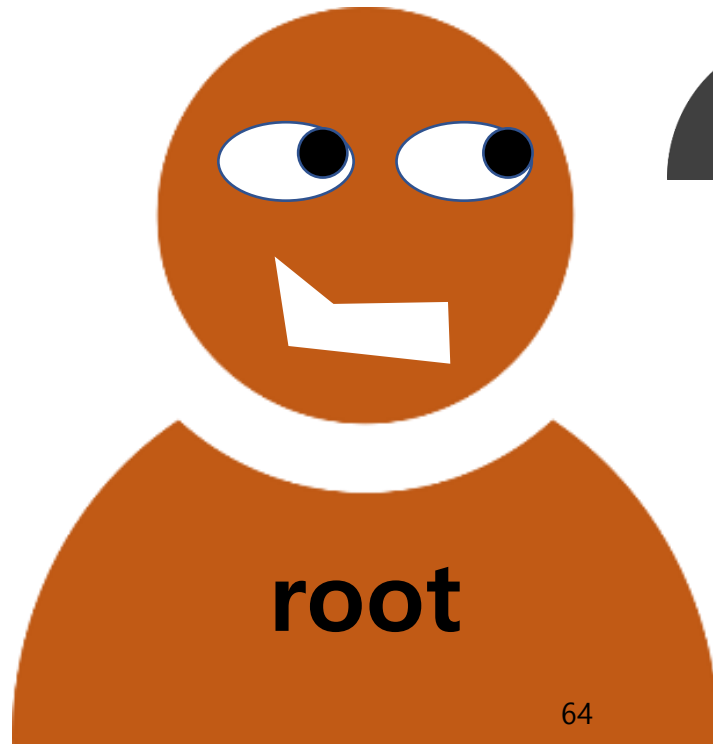
처음 생성한 사용자 계정(fy) 은

root 의 권한을 빌려 쓸 수 있도록 설정이 되어 있다!

상식 QUIZ

root 사용자는 모든 파일을 다 뒤적거릴 수 있는가?

○ / X ????



user1 : fy



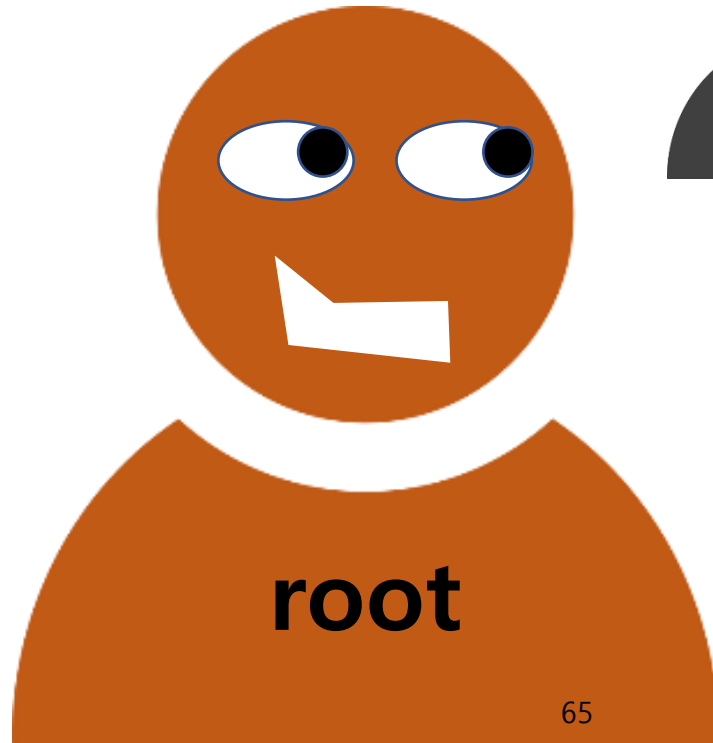
user2 : kim

user3 = lee



root의 권한은 막강하다.

root 로 아무나 접속을 못하게
보안에 신경 써야 한다.



user1 : fy



user2 : kim

user3 = lee



[참고] 임베디드 시스템에서 root

일반적인 경우, 임베디드 시스템에서 다른 user 가 필요하진 않다.

임베디드 리눅스에서는

다음과 같이 동작되도록 한다.

1. 켜자마자 바로 root 로 자동 로그인 되도록 한다.
2. 자동으로 세탁기 App을 실행하도록 만든다.
→ 즉, 켜자마자 root 권한으로 자동 App 실행



User 와 Group

목표

리눅스에서 사용자와 그룹의 개념에 대해 학습한다.
user와 group 을 생성 / 삭제하는 방법에 대해 학습한다.

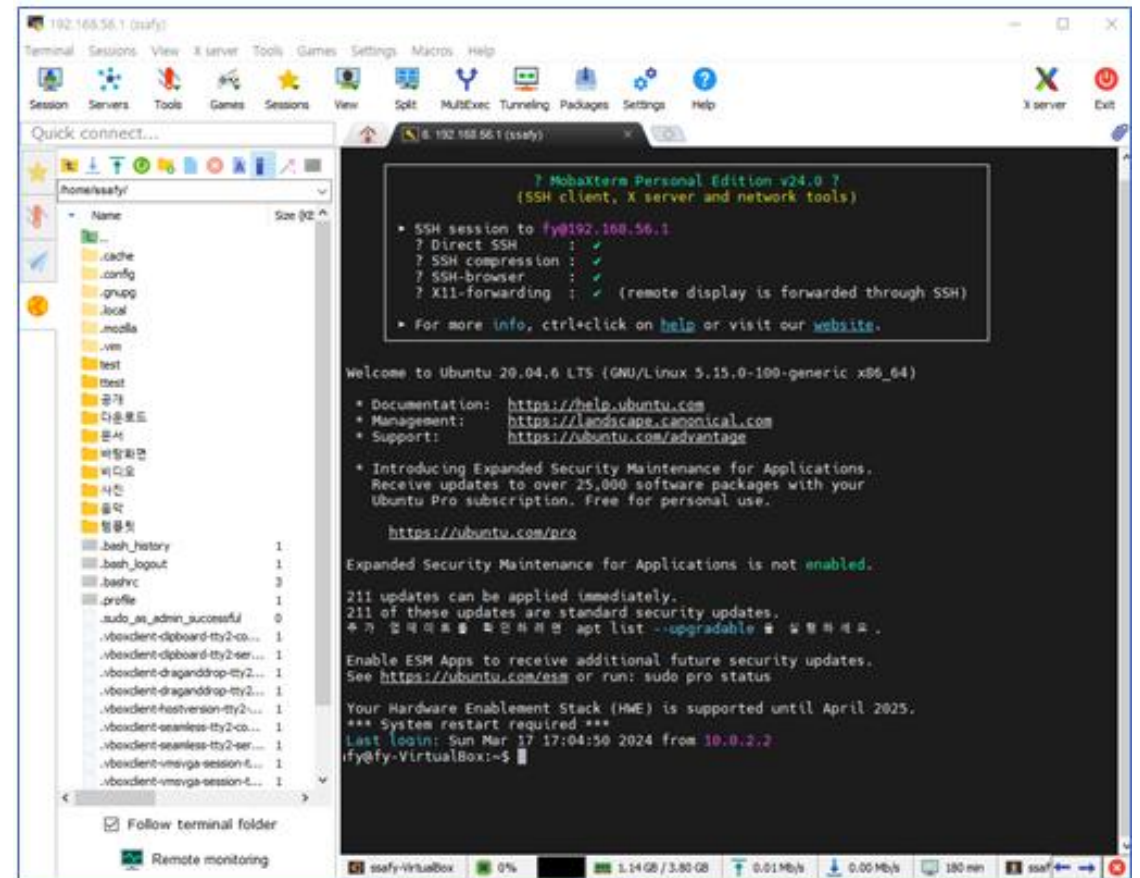
다중 사용자 시스템인 리눅스

리눅스는 다중 사용자 시스템이다.

이제 여러 user를 생성하고 관리하는 방법에 대해 학습한다.

Linux 직접 실행이 아닌, 원격 터미널 연결로 권한 테스트를 하자.
우분투를 종료하는 게 아니다. → 종료하면, 원격 접속을 할 수 없다^^

MobaXterm 에서 실습한다!



user 생성하기

adduser [계정명]

계정을 추가한다는 것은 아무나 할 수 없다. **root의 권한**이다. (따라서 sudo를 써줘야한다.)

sudo adduser kfc

비밀번호 : **1**

여러 묻는 질문에

엔터로 대답해주자.

엔터 = Default 값으로 세팅

```
fy@fy-VirtualBox:~$ sudo adduser kfc
'kfc' 사용자를 추가 중 ...
새 그룹 'kfc' (1001) 추가 ...
새 사용자 'kfc' (1001) 을 (를 ) 그룹 'kfc' (으 )로 추가 ...
'/home/kfc' 홈 디렉토리를 생성하는 중 ...
'/etc/skel'에서 파일들을 복사하는 중 ...
새 암호 :
새 암호 재 입력 :
passwd: 암호를 성공적으로 업데이트했습니다
kfc의 사용자의 정보를 바꿉니다
새로운 값을 넣거나, 기본값을 원하시면 엔터를 치세요
이름 []:
방 번호 []:
직장 전화번호 []:
집 전화번호 []:
기타 []:
정보가 올바른지 확인하십시오 [Y/n]
```

[참고] adduser 와 useradd 의 차이

새로운 계정을 생성하는 명령어

adduser : 편리한 자동 설정

- 필요한 모든 설정들을 Default 값으로 생성



이것을 쓰자

useradd : 전문가용, 세부 옵션을 해야한다.

- 홈 디렉토리, 계정설정, 사용자 UID 설정, 시작 셸 환경 등 세부 설정 가능

user 생성 확인하기

사용자 계정을 만들면, 홈 디렉토리에 자동 생성 된다.

```
cd /home
```

```
ls
```

```
inho@inho:~$ cd /home/  
inho@inho:/home$ ls  
inho  kfc  
inho@inho:/home$
```

user 를 만들면 생기는 일

새로운 user를 만들면

1. 해당 user 이름으로 로그인 가능하다.
 - **root 권한이 없는**, 새로운 계정이 만들어진다.
2. 해당 경로에, user 전용 디렉토리가 만들어진다.
 - /home/[user이름]

새로운 user 로그인하기

su [계정명]

kfc 계정으로 접속하자.

su kfc

암호 : 1

kfc 의 사용자 홈 디렉토리 확인

cd ~

ls

```
fy@fy-VirtualBox:~$ su kfc
암호 :
kfc@fy-VirtualBox:/home/fy$ ls
loopy.png  test  ttest  공개  다운로드  문서
kfc@fy-VirtualBox:/home/fy$ cd ~
kfc@fy-VirtualBox:~$ ls
kfc@fy-VirtualBox:~$
```

apt 설치하기

sudo apt install sl -y
안된다.

```
kfc@fy-VirtualBox:~$ sudo apt install sl -y
[sudo] kfc 암호 :
kfc은 (는 ) sudoers 설정 파일에 없습니다. 이 시도를 보고합니다.
kfc@fy-VirtualBox:~$
```

kfc 는 일반 사용자 계정이다.

kfc 는 root의 권한을 사용할 수 없다.

adduser로 사용자 계정을 만들면, root 권한 없는 계정이 만들어지기 때문
당연하게도 kfc 한테 root의 권한을 줄 수 있는 방법은 있지만, 우리 수업에서는 다루지 않는다.

kfc 로그아웃

exit

다시 fy 계정으로 복귀

```
kfc@fy-VirtualBox:~$ sudo apt install sl -y
[sudo] kfc 암호 :
kfc은 (는 ) sudoers 설정 파일에 없습니다 . 이 시도를 보고합니다 .
kfc@fy-VirtualBox:~$
```

```
kfc@fy-VirtualBox:~$ exit
exit
fy@fy-VirtualBox:~$
```

user 삭제하기

deluser [계정명]

계정을 삭제한다는 것은 아무나 할 수 없다. **root의 권한**이다.

`sudo deluser kfc`

```
fy@fy-VirtualBox:~$ sudo deluser kfc
'kfc' 사용자 제거 중 ...
경고 : 'kfc' 그룹이 회원 목록에 더 이상 없음 .
완료 .
fy@fy-VirtualBox:~$
```

새로운 사용자 만들기

이어서 Group 실습을 위해 계정 두 개를 추가한다.

```
sudo adduser jdragon  
sudo adduser onnew
```

```
fy@fy-VirtualBox:/home$ ls  
jdragon  onnew  fy  
fy@fy-VirtualBox:/home$ █
```

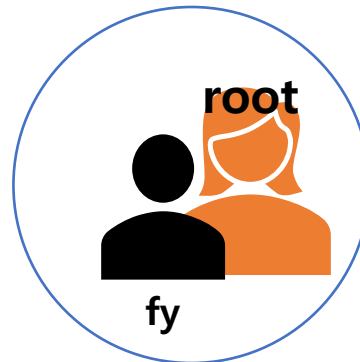
리눅스 Group 규칙

User를 생성하면 그룹이 함께 생성된다.
나는 나 스스로 그룹이다!

jdragon group



fy group



onnew group



group 확인하기

groups [계정명]

특정 user가 소속된 그룹 확인하기

계정 이름 : 소속된 그룹 이름

groups fy

groups jdragon

```
fy@fy-VirtualBox:/home$ groups fy
fy : fy adm cdrom sudo dip plugdev lpadmin lxd sambashare
fy@fy-VirtualBox:/home$ groups jdragon
jdragon : jdragon
fy@fy-VirtualBox:/home$
```

fy 는 처음 생성한 사용자여서 더 많은 그룹에 속해 있다.

Group을 쓰는 이유

리눅스에서는 Group 단위로, 권한을 단체 설정하려고

특정 Group만 읽고 쓸 수 있는 파일 설정 가능

특정 Group만 쓸 수 있는 디렉토리 설정 가능

특정 Group만 실행 시킬 수 있는 프로그램 설정 가능

실제 회사에서도 이런 방식으로 부서별 그룹을 지정한다.

회사 부서별 (그룹별) 파일 접속,
디렉토리 접속 권한을 달리 부여함

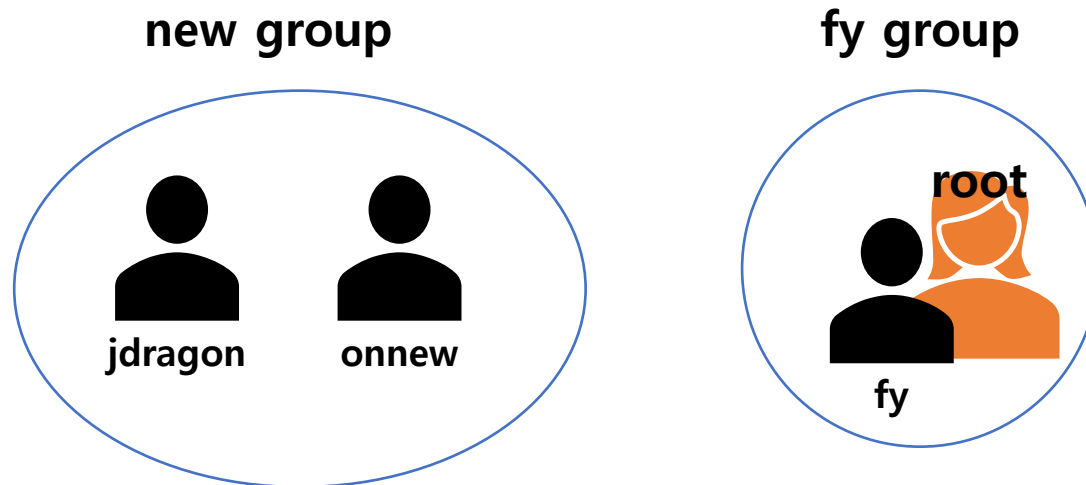
타 부서 파일을 읽지 못하도록
그룹 별 권한 설정 가능

리눅스 Group 제어

그룹을 더 생성할 수도 있으며,
특정 User를 그룹 안에 포함시킬 수 있다.

우리 수업 범위에 포함되지 않는 내용이므로, 그룹 제어 실습은 생략한다.

- 리눅스를 서버 용도로 사용하는 경우 Group의 중요도는 매우 높다.
- 리눅스를 임베디드용 OS 용도로 사용하는 경우 Group의 중요도는 상대적으로 낮다.



리눅스 파일의 종류

챕터 목표

목표

리눅스에서 파일의 종류에 대해 학습한다.

QUIZ

리눅스는 모든 것을 파일로 관리할까?

O / X

정답은 0

특이한 리눅스 시스템이다.

리눅스는 모든 것을 파일로 관리한다!

매우 중요한 철학

리눅스 파일의 종류

Reguler File

일반 파일

Directory File

리눅스 커널 내부에서는, 디렉토리도 파일로 취급

Link File

윈도우의 **바로가기**와 비슷한 파일

Device File

새로운 장치 (마우스 등) 연결하면, **상징물**과 같은 파일이 하나 생긴다.
이 파일을 건드리면, 장치를 제어할 수 있다.

권한을 공부하기 전,
암기하셔야 하는 내용
파일의 네 가지 종류!!



파일 종류 확인하기

ls -al 명령어로 확인할 수 있다

cd /

ls -al

가장 앞 한 글자

- - : regular file
- d : directory file
- l : link file
- c, b : device file, /dev 로 가서 ls -al

```
drwxr-xr-x 31 root root  
lrwxrwxrwx 1 root root  
drwxr-xr-x 11 root root 40  
drwxr-xr-x 2 root root 40  
-rw----- 1 root root 2147483  
dr-xr-xr-x 13 root root
```


파일의 권한

목표

리눅스의 파일 관리에 대해 학습하고, 파일에 권한을 부여한다.
user / group 마다 다르게 권한을 부여하는 방법에 대해 학습한다.

파일 권한 관리란?

파일 권한 관리란?

특정 파일에 대한 read/write/execute 권한을 부여하는 것을 의미한다.

다중 사용자 시스템에서 굉장히 중요한 내용

ex) 인사팀의 인사고과 파일을 개발팀, 설계팀의 장지효는 존재까지는 알 수 있어도 볼 수 없다.

ex) 인사팀은 모든 부서 인원의 인적 명세를 볼 수 있어도, 개발팀은 볼 수 없다.

ex) 인사팀의 고과 파일은 인사계원은 볼 수 없지만, 인사 팀장은 볼 수 있다.

ls -al 로 나오는 정보 - 1

파일의 종류 다음으로 나오는 9개의 글자는 권한을 의미한다.

owner 권한 : 3글자

owner group 권한 : 3글자

other 권한 : 3글자

r : read 읽기

w : write 쓰기

x : execute 실행

- : 없음

```
drwxr-xr-x  31 root root  
lrwxrwxrwx   1 root root  
drwxr-xr-x  11 root root  
drwxr-xr-x   2 root root  
-rw-----   1 root root 2147483  
dr-xr-xr-x  13 root root
```

ls -al 로 나오는 정보 - 2

그 다음 나오는 정보는

파일의 소유자와 소유 그룹에 대한 정보이다.

해당 파일의 owner 와 owner group을 의미한다.

앞서 설명한 권한이 owner 와 owner group을 의미한다.

파일의 권한은 owner 와 owner group이 다를 수 있다.

```
drwxr-xr-x  31 root root  
lrwxrwxrwx   1 root root  
drwxr-xr-x  11 root root  
drwxr-xr-x   2 root root  
-rw-----   1 root root 2147483  
dr-xr-xr-x  13 root root
```

other 란?

other 란,
owner 도 owner group도 아닌 user 들이다.

```
drwxr-xr-x  31 root root  
lrwxrwxrwx   1 root root  
drwxr-xr-x  11 root root 4  
drwxr-xr-x   2 root root 4  
-rw-----   1 root root 2147483  
dr-xr-xr-x  13 root root
```

파일 권한 바꾸기

파일 권한 실습을 위해 파일을 준비한다.

~ 에서 진행한다.

```
mkdir work
```

```
cd ./work
```

```
touch abc
```

```
ls -al
```

```
fy@fy-VirtualBox:~$ mkdir work
fy@fy-VirtualBox:~$ cd ./work
fy@fy-VirtualBox:~/work$ ls
fy@fy-VirtualBox:~/work$ touch abc
fy@fy-VirtualBox:~/work$ ls -al
합 계 8
drwxrwxr-x  2 fy fy 4096  3월 17 22:25 .
drwxr-xr-x 19 fy fy 4096  3월 17 22:24 ..
-rw-rw-r--  1 fy fy    0  3월 17 22:25 abc
fy@fy-VirtualBox:~/work$
```


파일 권한 변경하기

chmod [mode] [파일명]

파일 권한을 file mode 라고 한다.

당연히, 관리자의 권한이 필요하므로 앞에 "sudo" 키워드를 붙인다.

sudo chmod u=rwx abc

- u : user
- r,w,x : 읽기/쓰기/실행

ls -al 로 결과를 확인하자.

```
fy@fy-VirtualBox:~/work$ ls -al
합계 8
drwxrwxr-x  2 fy fy  4096  3월  17 22:25 .
drwxr-xr-x 19 fy fy  4096  3월  17 22:24 ..
-rw-rw-r--  1 jdragon embedded  0  3월  17 22:25 abc
fy@fy-VirtualBox:~/work$ sudo chmod u=rwx abc
fy@fy-VirtualBox:~/work$ ls -al
합계 8
drwxrwxr-x  2 fy fy  4096  3월  17 22:25 .
drwxr-xr-x 19 fy fy  4096  3월  17 22:24 ..
-rwxrw-r--  1 jdragon embedded  0  3월  17 22:25 abc
```



파일 권한 변경하기 - 2

user / **g**roup / **o**ther / **a**ll 에 권한을 부여할 수 있다.

```
sudo chmod u=rx abc  
sudo chmod g=wx abc  
sudo chmod o=r abc  
sudo chmod a=x abc
```

부여할 때 기존에 있던 권한에 +,- 하거나 = 지정할 수 있다.

```
sudo chmod u+r abc  
sudo chmod g+r abc  
sudo chmod o+r abc  
sudo chmod a+w abc
```

파일 권한 변경하기 - 3

2진수로 권한을 부여할 수 있다.

user / group / other 의 각각 3자리를 0-7로 표현한다.

r-- -w- --x = 421

rwX rwX rwX = 777

rw- rw- r-- = 664

```
fy@fy-VirtualBox:~/work$ sudo chmod 421 abc
fy@fy-VirtualBox:~/work$ ls -al
합 계 8
drwxrwxr-x  2 fy  fy   4096  3월  17  22:25 .
drwxr-xr-x 19 fy  fy   4096  3월  17  22:24 ..
-r---w---x  1 jdragon embedded  0  3월  17  22:25 abc
```

당연하게도 모두 알아야 한다.

숫자로 권한을 지정하는 방식

"u+x " 와 같이 기존 권한에 권한을 추가 부여하는 방식

이와 같은 방법으로,

각 파일마다 사용자에게 다른 실행권한 / 읽기권한 / 쓰기 권한을 줄수있다.

디렉토리 또한 파일로 취급된다.

디렉토리도 파일로 간주된다. 따라서 디렉토리에도 권한을 부여할 수 있다.

```
cd /  
ls -al
```

l : link 파일
(바로가기 파일)

d : 디렉토리

```
fy@fy-VirtualBox:/$ ls -al  
합 계 2097240  
drwxr-xr-x 20 root root 4096 3월 13 19:40 .  
drwxr-xr-x 20 root root 4096 3월 13 19:40 ..  
lrwxrwxrwx 1 root root 7 3월 13 19:38 bin -> usr/bin  
drwxr-xr-x 4 root root 4096 3월 17 15:42 boot  
drwxrwxr-x 2 root root 4096 3월 13 19:40 cdrom  
drwxr-xr-x 19 root root 4140 3월 17 20:30 dev  
drwxr-xr-x 131 root root 12288 3월 17 22:24 etc  
drwxr-xr-x 5 root root 4096 3월 17 21:53 home
```

디렉토리의 권한

디렉토리 read 권한

ls 로 파일 목록을 읽을 수 있는 권한

디렉토리 write 권한

디렉토리 내 파일 생성 / 삭제 권한

디렉토리 execution

cd 로 디렉토리에 접근할 수 있는 권한

디렉토리 권한 테스트

디렉토리를 생성하고 권한을 준다.

mkdir test

sudo chmod u=x test

ls -al ./test

- 거부 된다.

cd ./test

- 가능

touch abc

- 거부 된다.

```
fy@fy-VirtualBox:~$ mkdir test
fy@fy-VirtualBox:~$ sudo chmod u=x test
fy@fy-VirtualBox:~$ ls -al ./test
ls: './test' 디렉터리를 열 수 없음 : 허가 거부
```

```
fy@fy-VirtualBox:~$ cd ./test
fy@fy-VirtualBox:~/test$ touch abc
touch: 'abc'를 touch할 수 없음 : 허가 거부
fy@fy-VirtualBox:~/test$
```

[참고] 파일의 실행 권한

텍스트 파일을 실행하면 쉘 스크립트 로 동작한다.

파일 생성하고 오른쪽처럼 추가한다.

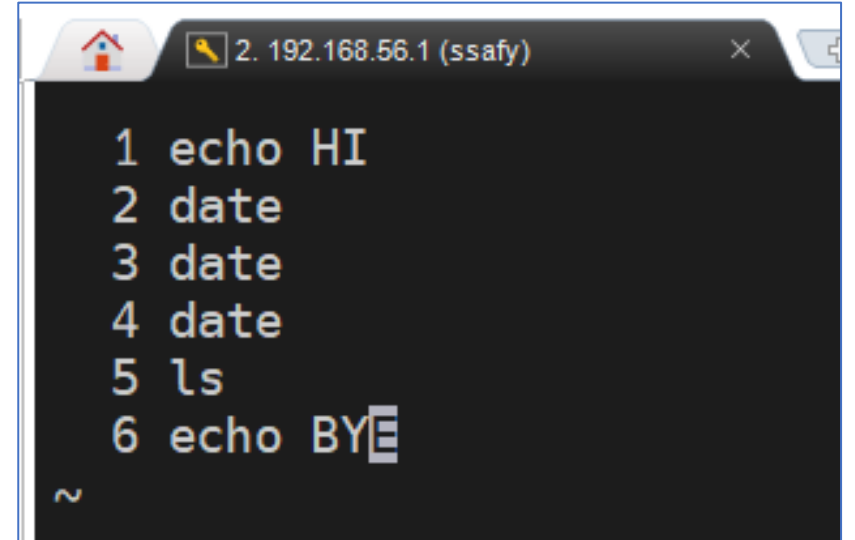
- vi ~/bts

파일에 권한을 설정한다.

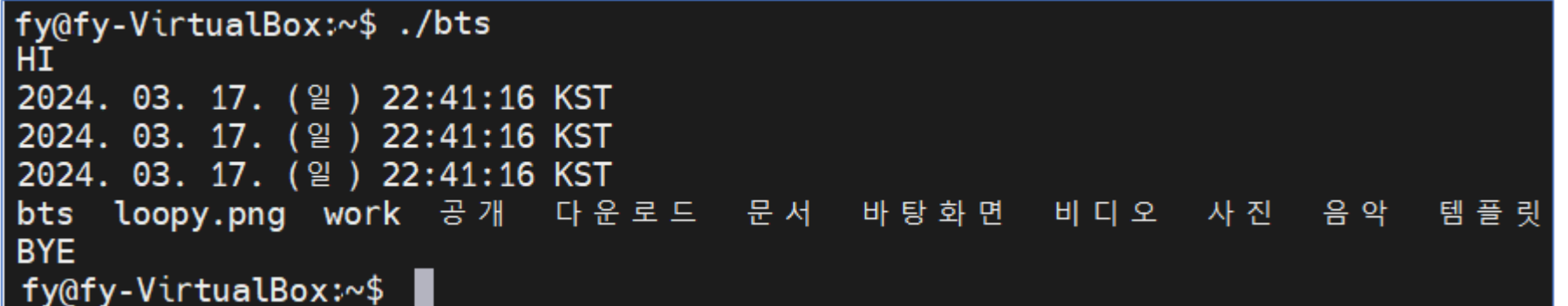
- sudo chmod u=rx bts

파일을 실행한다.

- ./bts



```
1 echo HI
2 date
3 date
4 date
5 ls
6 echo BYE
```



```
fy@fy-VirtualBox:~$ ./bts
HI
2024. 03. 17. (일) 22:41:16 KST
2024. 03. 17. (일) 22:41:16 KST
2024. 03. 17. (일) 22:41:16 KST
bts  loopy.png  work  공개  다운로드  문서  바탕화면  비디오  사진  음악  템플릿
BYE
fy@fy-VirtualBox:~$
```

Chapter4-1

Build System

챕터의 포인트

- gcc Build Process
- 빌드 자동화 스크립트
- build system 체험
- 간단한 make 문법
- 단계별 Makefile 제작

gcc Build Process

목표

gcc 를 이용해 build 과정을 나눠서 확인한다.

compile & assemble 그리고 linking 을 통해 실행 파일이 만들어 지는 것을 실습한다.

Build System 이란?

Build 할 때 필요한 여러 작업을 도와주는 프로그램들

임베디드 리눅스에서 자주 사용되는 Build System 종류

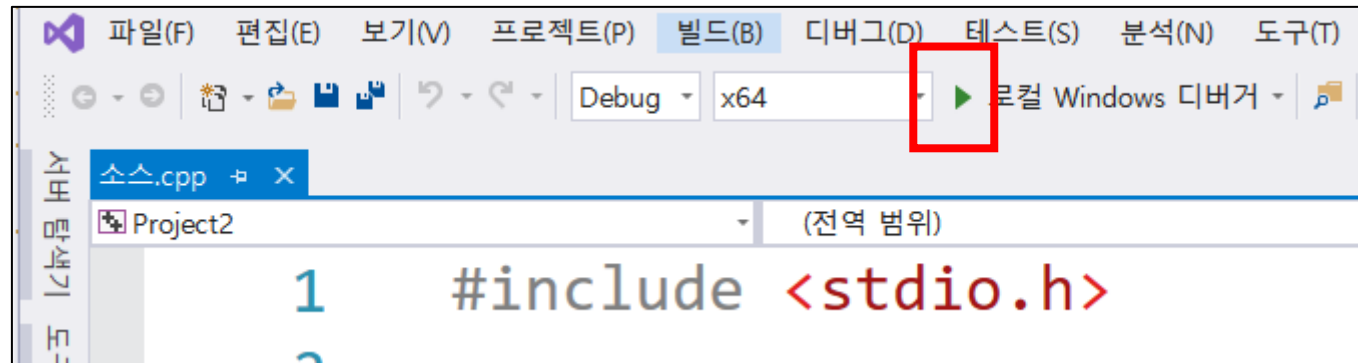
1. **make** (우리 수업에서는 **make**만 다룬다.)
2. cmake

make에 대해 살펴보기 전

gcc를 이용해 빌드의 과정을 잠깐 살펴보자.

빌드란?

소스코드(.c, .cpp)에서 실행 가능한 Software(.elf, .exe)로 변환하는 **과정 (Process)** 또는 **결과물**
저 작은 버튼 하나에 굉장한 과정들이 함축되어 있다.



코드를 작성한다.

~/test1 디렉토리 생성

mkdir test1

cd ./test1

main.c 생성

yellow.c 생성


yellow.h 생성

```
fy@fy-VirtualBox:~/test1$ ls
main.c  yellow.c  yellow.h
fy@fy-VirtualBox:~/test1$
```


<https://gist.github.com/hoconoco/e91f094c0b11626f555ff367e15e2e99>

 shell_day4_test1_main.c

```
1  #include <stdio.h>
2  #include "yellow.h"
3
4  int main(){
5      printf("I'm Green!\n");
6      yellow();
7
8      return 0;
9  }
```

 shell_day4_test1_yellow.c

```
1  #include <stdio.h>
2
3  void yellow(){
4      printf("I'm yellow\n");
5  }
```

 shell_day4_test1_yellow.h

```
1  void yellow();
```

gcc 기준 빌드 과정은 크게 둘로 나뉜다.

Compile & Assemble

- 하나의 소스코드 파일이 0과 1로 구성된 Object 파일이 만들어짐

Linking

- 만들어진 Object 파일들 + Library 들을 모아 하나로 합침

Compile & Assemble

.c 파일을 각각 Compile & Assemble 하자.

```
gcc -c ./main.c  
gcc -c ./yellow.c  
ls
```

.o 파일이 생성된다! (Object 파일)

```
fy@fy-VirtualBox:~/test1$ gcc -c ./yellow.c  
fy@fy-VirtualBox:~/test1$ ls  
main.c  main.o  yellow.c  yellow.h  yellow.o  
fy@fy-VirtualBox:~/test1$
```


Linking

만들어진 Object 파일들과 라이브러리 함수들을 하나로 합친다.

```
gcc ./main.o ./yellow.o -o ./go
```

- -o 옵션 : output 파일 지정

```
ls
```

```
./go
```

```
fy@fy-VirtualBox:~/test1$ gcc ./main.o ./yellow.o -o ./go
fy@fy-VirtualBox:~/test1$ ls
go  main.c  main.o  yellow.c  yellow.h  yellow.o
fy@fy-VirtualBox:~/test1$ ./go
I'm Green!
I'm yellow
fy@fy-VirtualBox:~/test1$
```

[참고] 똑똑한 GCC

GCC 는 똑똑해서 굳이 나눠서 작업하지 않아도 동작한다.

```
sudo rm -r *.o
sudo rm -r ./go
gcc ./*.c
ls
./a.out
```

```
fy@fy-VirtualBox:~/test1$ sudo rm -r *.o
fy@fy-VirtualBox:~/test1$ sudo rm -r ./go
fy@fy-VirtualBox:~/test1$ ls
main.c  yellow.c  yellow.h
fy@fy-VirtualBox:~/test1$ gcc ./*.c
fy@fy-VirtualBox:~/test1$ ls
a.out  main.c  yellow.c  yellow.h
fy@fy-VirtualBox:~/test1$ ./a.out
I'm Green!
I'm yellow
fy@fy-VirtualBox:~/test1$
```

빌드 자동화 스크립트

목표

bash shell script 를 이용한 build 방식을 체험한다.
bash shell script 빌드의 단점을 확인한다.

자동화 프로그램 개발에 특화된 Script 언어가 있다. (.sh)

이번 챕터에서는 bash Script 로 build 스크립트를 작성하고 실행한다.

특징 : 기술된 script 내용대로 차례대로 작업이 일어난다.

코드를 작성하고 빌드한다.

bash shell 명령어로 스크립트를 제작하고 실행한다.

~/test1/ 를 복사해서 ~/test2 생성

cp -r ~/test1 ~/test2

vi ~/test2/build.sh

source build.sh

./gogo

```
fy@fy-VirtualBox:~/test2$ source build.sh
fy@fy-VirtualBox:~/test2$ ls
build.sh  gogo  main.c  yellow.c  yellow.h
fy@fy-VirtualBox:~/test2$ ./gogo
I'm Green!
I'm yellow
fy@fy-VirtualBox:~/test2$
```

 shell_day4_build.sh

```
1  #! /bin/bash
2  gcc -c ./main.c
3  gcc -c ./yellow.c
4  gcc ./main.o ./yellow.o -o ./gogo
5  rm -r ./*.o
```

<https://gist.github.com/hoconoco/c8023cdbe76f159caa279ad35ea3d7bd>

yellow.c 를 수정한다면?

yellow.c 하나만 수정해도, 다시 build 를 진행하게 된다

main.c 는 다시 build 할 필요가 없지만, 다시 build 가 된다. → 작성된 순서대로 작업을 할 뿐이다!

만약, 이런 파일이 1000개가 넘는다면?

build만 하다가 시간이 다 간다.

야근이다!

이럴 때 필요한 게 바로 make build system이다!



build system 체험

소프트웨어 빌드 자동화 도구

소스 코드 파일로부터 실행 파일이나 라이브러리 등을 생성하는 데 사용된다.

주로 C, C++ 그리고 다른 컴파일 가능한 언어의 프로젝트에서 사용된다.

Makefile 이라는 특별한 형식의 파일을 사용한다.

- 별도의 문법을 사용한다.

make 설치

```
sudo apt install make -y
```

```
fy@fy-VirtualBox:~$ sudo apt install make -y
[sudo] fy 암호 :
패키지 목록을 읽는 중입니다 ... 완료
의존성 트리를 만드는 중입니다
상태 정보를 읽는 중입니다 ... 완료
패키지 make는 이미 최신 버전입니다 (4.2.1-1.2).
```

make 사용방법

1. **Makefile**이라는 스크립트 파일을 만든다.
make 문법에 맞추어서 작성해야 한다.
2. **Makefile**을 작성한 뒤 **make**를 실행한다.
명령어 : make

코드를 작성하고 빌드한다.

Makefile 을 작성하고 빌드한다.

~/test2 를 복사한다.

cp -r ~/test2 ~/test3

cd ./test3

rm -r ./gogo build.sh

vi Makefile

make

./gogo

```
fy@fy-VirtualBox:~/test3$ ls
Makefile main.c yellow.c yellow.h
```

```
1  gogo : main.o yellow.o
2          gcc main.o yellow.o -o gogo
3
4  main.o : main.c
5          gcc -c main.c
6
7  yellow.o : yellow.c
8          gcc -c yellow.c
9
10 clean :
11        rm -r ./*.o ./gogo
```

<https://gist.github.com/hoconoco/54dd4461a5858acab6957e11b514e694>

make 실습하기

수정 없이 make 하면, build가 진행되지 않는다.

make

```
fy@fy-VirtualBox:~/test3$ ls
Makefile gogo main.c main.o yellow.c yellow.h yellow.o
fy@fy-VirtualBox:~/test3$ make
make: 'gogo'은(는) 이미 업데이트되었습니다.
fy@fy-VirtualBox:~/test3$
```

소스코드 하나 수정하고 make 하면, 수정된 파일만 build 된다.

vi yellow.c

make

```
fy@fy-VirtualBox:~/test3$ make
gcc -c yellow.c
gcc main.o yellow.o -o gogo
fy@fy-VirtualBox:~/test3$
```

특정 키워드로 동작 시킬 수 있다.

make clean

.o 파일 , 실행 파일 삭제

```
fy@fy-VirtualBox:~/test3$ make clean
rm -r ./*.o ./gogo
fy@fy-VirtualBox:~/test3$ ls
Makefile main.c yellow.c yellow.h
```

Make Build System의 장점 두 가지

Build 자동화

- 기술된 순서대로 Build 작업을 수행하는 자동화 스크립트 지원

Build 속도 최적화

- 불필요한 Compile & Assemble 피하기
- 파일 간의 의존성을 추적하여, 파일이 변경된 경우에만 컴파일한다!

make 문법 소개

목표

makefile 실습을 하기 전 간단하게 make 문법에 대해 소개한다.

make는 표준화된 문법을 사용한다.

새 디렉토리 생성

```
mkdir ~/test5
```

```
cd ./test5
```

```
vi Makefile
```

<https://gist.github.com/hoconoco/17bac3516c4543c9142ee2f800cc4e53>

```
1 HI:
2     echo "HI"
3
4 HELLO :
5     echo "HELLO"
6
```


make 문법 - 1

Target 타겟

목표 파일 이름, 빌드하려는 최종 결과물
1개 이상의 Target 이 있어야 한다.
comment 를 실행한다.
(반드시 Tab 들여쓰기 해야 한다.)

make → 첫번째 Target HI 실행
make HI
make HELLO

띄어쓰기가
아닌 탭!

```
1 HI:
2 echo "HI"
3
4 HELLO :
5 echo "HELLO"
6
```

```
fy@fy-VirtualBox:~/test5$ make
echo "HI"
HI
fy@fy-VirtualBox:~/test5$ make HI
echo "HI"
HI
fy@fy-VirtualBox:~/test5$ make HELLO
echo "HELLO"
HELLO
fy@fy-VirtualBox:~/test5$
```

Dependency 의존성

Target을 생성하기 위한 파일 목록
의존성 Target을 먼저 수행하게 된다.

make HI

- Target HI를 하려고 보니 HELLO가 의존성
- HELLO를 우선 실행하게 된다.

<https://gist.github.com/hoconoco/e37269effd03b118bf47ad4a95b6ffd6>

```
1 HI : HELLO
2     echo "HI"
3
4 HELLO :
5     echo "HELLO"
6
```

```
fy@fy-VirtualBox:~/test5$ make HI
echo "HELLO"
HELLO
echo "HI"
HI
```

Variable 변수

소괄호 () or 중괄호 { } 를 붙여서 사용한다.

\$ 를 앞에 붙여서 사용한다.

가독성을 위해 script 최상단에 작성한다.

<https://gist.github.com/hoconoco/81866dab5dabf477abc8bbfbf09e9afa>

```
1 MSG1 = "HI"
2 MSG2 = "HELLO"
3
4 HI : HELLO
5     echo ${MSG1}
6
7 HELLO :
8     echo $(MSG2)
```

Comment 주석

#을 이용해서 주석을 표시한다.
공식적으로 한 줄 주석만 지원된다.

<https://gist.github.com/hoconoco/42bbe4b98febb4db3c8d061113b369e7>

```
1 MSG1 = "HI"
2 MSG2 = "HELLO"
3
4 #comment
5
6 HI : HELLO
7     echo ${MSG1}
8
9 HELLO :
10     echo $(MSG2)
```

특수 변수

자동 변수라고도 한다.

자주 사용 되는 값들을 참조한다.

`$@` : Target 이름

`$$` : Dependency 목록 전체

`$<` : Dependency 목록 중 첫 번째

등등이 있다.

```
1 MSG1 = "HI"
2 MSG2 = "HELLO"
3
4 HI : HELLO
5     echo $@
6
7 HELLO :
8     echo $(MSG2)
9
```

<https://gist.github.com/hoconoco/4202c99965faa2dd6c7cc10f6e2e1aaf>

[참고] = Variation - 1

= 연산자는 다양한 variation을 갖는다.

+=

- 덧 붙이기
- hi faker

```
1 MSG = "hi"  
2 MSG += "faker"  
3  
4 who :  
5     @echo $(MSG)
```

```
fy@fy-VirtualBox:~/test5$ make  
hi faker
```

<https://gist.github.com/hoconoco/a556b915fca6581caa71049903f9a73d>

[참고] = Variation - 2

= 연산자는 다양한 variation을 갖는다.

:= (Simple Equi)

- Script 순서대로 현재 기준에서 값을 넣는다.
- hi faker

= (Recursive Equi)

- 최종 변수 결과를 집어 넣는다.
- hi faker yojongdo

```
fy@fy-VirtualBox:~/test5$ make SIMPLE
hi faker
fy@fy-VirtualBox:~/test5$ make RECUL
hi faker yojongdo
fy@fy-VirtualBox:~/test5$
```

```
1 MSG = "hi"
2 MSG += "faker"
3
4 SIMPLE := $(MSG)
5 RECUL = $(MSG)
6
7 MSG += "yojongdo"
8
9 SIMPLE :
10     @echo $(SIMPLE)
11
12 RECUL :
13     @echo $(RECUL)
```

단계별 Makefile 제작

목표

Makefile 을 단계별로 작성하며 작성 법을 배운다.

코드를 작성하고 빌드한다.

~/test6 디렉토리 생성

main.c common.h

func1.c func1.h

func2.c func2.h

Makefile

```
fy@fy-VirtualBox:~/test6$ ls
Makefile common.h func1.c func1.h func2.c func2.h main.c
fy@fy-VirtualBox:~/test6$
```

```
result: main.o func1.o func2.o
        gcc main.o func1.o func2.o -o result

main.o: main.c common.h func1.h func2.h
        gcc -c main.c

func1.o: func1.c common.h func1.h
        gcc -c func1.c

func2.o: func2.c common.h func2.h
        gcc -c func2.c

clean:
        rm -r ./*.o result
```

<https://gist.github.com/hoconoco/8333cd7a9cd8721285a08ce280fe3c81>

make 테스트한다.

make 한다.

make
./result

이제 Makefile 을 단계별로
바꿔가며 작성한다.

```
fy@fy-VirtualBox:~/test5$ make
gcc -c main.c
gcc -c func1.c
gcc -c func2.c
gcc main.o func1.o func2.o -o result
fy@fy-VirtualBox:~/test5$ ./result
TEST START!
Func1 TEST START
=====
Func1 TEST END!
FUNC 2 TEST START!!
*****
FUNC 2 TEST END!!
TEST END!

fy@fy-VirtualBox:~/test5$
```

2단계 - 변수 추가

변수 추가

CC : compiler 가 바뀔 때마다 변경

OBJS : 목적 파일 목록

Test

make

make clean

func1.c 수정 한 뒤 다시 make
make clean

```
1 CC = gcc
2 OBJS = main.o func1.o func2.o
3
4 result : $(OBJS)
5     $(CC) -o result $(OBJS)
6
7 main.o : main.c common.h func1.h func2.h
8     $(CC) -c main.c
9
10 func1.o : func1.c common.h func1.h
11     $(CC) -c func1.c
12
13 func2.o : func2.c common.h func2.h
14     $(CC) -c func2.c
15
16 clean :
17     rm -r $(OBJS) result
```

3단계 - 특수 변수 사용

자주 사용되는 키워드를 변수로 표현

`$@` : Target을 나타냄

`$^` : 의존성 타겟들을 나타냄

`$<` : 의존성 타겟 중 첫 번째 파일을 나타냄

Test

make clean

make

./result

```
1 CC = gcc
2 OBJS = main.o func1.o func2.o
3
4 result : $(OBJS)
5     $(CC) -o $@ $^
6
7 main.o : main.c common.h func1.h func2.h
8     $(CC) -c $<
9
10 func1.o : func1.c common.h func1.h
11     $(CC) -c $<
12
13 func2.o : func2.c common.h func2.h
14     $(CC) -c $<
15
16 clean :
17     rm -r $(OBJS) result
```

4단계 – 컴파일러 옵션

컴파일 옵션 지정 \$(CFLAGS)

- g : 디버깅 (Trace) 가능하도록 설정
- Wall : Warning 이 뜨면 Error처럼 멈추도록 함
- O2 : 최적화 2단계 옵션

```
1 CC = gcc
2 CFLAGS = -g -Wall -O2
3 OBJS = main.o func1.o func2.o
4
5 result : $(OBJS)
6         $(CC) -o $@ $^
7
8 main.o : main.c common.h func1.h func2.h
9         $(CC) $(CFLAGS) -c $<
10
11 func1.o : func1.c common.h func1.h
12         $(CC) $(CFLAGS) -c $<
13
14 func2.o : func2.c common.h func2.h
15         $(CC) $(CFLAGS) -c $<
16
17 clean :
18         rm -r $(OBJS) result
```

5단계 – wildcard, 확장자 치환 사용

wildcard 함수

지정된 패턴에 해당하는 파일 목록 갖고오기

*.c → 현재 디렉토리 내 모든 .c 파일 가져오기

확장자 치환 사용

변수에 할당된 파일 목록에서 확장자 치환

변수명:pattern=replacement

SRCS의 .c 를 .o 변경해서 OBJS 에 저장

```
1 CC = gcc
2 CFLAGS = -g -Wall -O2
3 SRCS = $(wildcard *.c)
4 OBJS = $(SRCS:.c=.o)
5
6 result: $(OBJS)
7     $(CC) -o $@ $^
8
```

makedepend 유틸리티 - 1

입력한 .c 파일을 분석해서
의존성 헤더파일을 등록해주는 make 도우미 유틸리티

makedepend 설치

```
sudo apt install xutils-dev -y
```

makedepend 실행

```
makedepend main.c func1.c func2.c -Y
```

```
$ makedepend main.c func1.c func2.c -Y
```


makedepend 유틸리티 - 2

makedepend 사용 시

Makefile.bak 백업 파일 생성

Makefile 에 의존성 파일 작성 됨

```
fy@fy-VirtualBox:~/test7$ ls
Makefile      common.h  func1.h  func2.h
Makefile.bak  func1.c  func2.c  main.c
```

```
15 # DO NOT DELETE
16
17 main.o: common.h func1.h func2.h
18 func1.o: common.h
19 func2.o: common.h
```

6단계 – make depend, SUFFIXES 사용

make depend 유틸리티

depend Target 추가

SUFFIXES

파일 확장자와 관련된 규칙을 지정할 때 사용
.c 파일을 .o 파일로 컴파일 하라는 의미.
(사실 default 설정과 동일하므로 생략해도 됨)

Test

make depend
make

```
1 CC = gcc
2 CFLAGS = -g -Wall -O2
3 SRCS = $(wildcard *.c)
4 OBJS = $(SRCS:.c=.o)
5 SUFFIXES = .c .o
6
7 result : $(OBJS)
8     $(CC) -o $@ $^
9
10 .c .o:
11     $(CC) $(CFLAGS) -c $<
12
13 clean :
14     rm -r $(OBJS) result
15
16 depend:
17     make depend $(OBJS) -Y
```

7단계 - 파일명 매크로 추가

최종 결과물로 나올 파일명 (result)을
변수를 만들어두자.

all

Target 파일이 여러 개일 때
사용하는 이름, 큰 의미는 없다.

```
1 CC = gcc
2 CFLAGS = -g -Wall -O2
3 SRCS = $(wildcard *.c)
4 OBJS = $(SRCS:.c=.o)
5 SUFFIXES = .c .o
6 TARGET = result
7
8 all : $(OBJS)
9     $(CC) -o $(TARGET) $^
10
11 .c .o:
12     $(CC) $(CFLAGS) -c $<
13
14 clean :
15     rm -r $(OBJS) $(TARGET)
16
17 depend:
18     makedepend $(OBJS) -Y
```

[참고] SUFFIXES 는 default 다.

.c 파일을 .o 로 컴파일하라는 내용은 Default 이다.

그래서 생략해도 된다.

→ SUFFIXES 변수를 삭제했다.

<https://gist.github.com/hoconoco/86e2934b042b2ccec48f3c020d2c6773>

```
1 CC = gcc
2 CFLAGS = -g -Wall -O2
3 SRCS = $(wildcard *.c)
4 OBJS = $(SRCS:.c=.o)
5 TARGET = result
6
7 all : $(OBJS)
8     $(CC) -o $(TARGET) $^
9
10 clean :
11     rm -r $(OBJS) $(TARGET)
12
13 depend:
14     makedepend $(OBJS) -Y
```

감사합니다.