# Komplex Server Manual for User

Complex Network Research Center

최호연[1] 2021.07.19 ~

[1]hoyun1009@snu.ac.kr

# Contents

# Chapter 1

# Introduction

이 문서는 *서울대학교 통계물리그룹의 서버 사용자를 위해 쓰여졌다.* 지금까지 리눅스를 사용해 본 경험이 없는 사람이라도 읽을 수 있도록 기초적인 내용을 많이 수록하였다. 본 매뉴얼의 내용을 확실히 알고 있다면, 서버를 사용하는데 큰 어려움이 없을것을 목표로 작성한다.

첫장에서는 리눅스에 대한 전반적인 소개와 기본적인 명령어들을 다룰 것이다. 터미널을 사용해 보지 않은 사용자를 대상으로 작성하였기 때문에, 이미 어느정도 경험이 있는 독자라면 이 부분은 건너뛰어도 무방하다. 필요할 경우에만 잠깐 돌아와 확인하는 식으로 읽어도 충분할 것이다.

두번째 장에서는 일반적으로 SSH 라는 기능을 통해 서버에 접속하는 방법을 설명할 것이다. 이를 위해 첫장에서 설명한 명령어들을 포함하여 원격 접속에 사용되는 추가적인 명령어들을 소개할 것이다. 위 내용은 대부분의 서버에서 통용되는 방식으로, 정확히 알고 있다면 꼭 연구실의 서버뿐만 아니라 다른 경우에도 도움이 많이 될것이라 기대한다.

세번째 장에서는 연구실에서 운영하는 "komplex server"(이하 komplex)의 구조와 서버를 사용하는데 있어 알아야 할 기본적인 방법을 설명할 것이다. 두번째 장에서 설명한 내용이 보편적인 설명이라면, 세번재 장에서는 조금 더 komplex에서만 적용되는 내용일 것이다. komplex를 처음 사용하는 사용자라면 반드시 이 장 만큼을 정독을 하고 넘어가길 바란다. 만약 komplex에 대한 조금 더 자세한 설명이 필요하다면, 관리자를 위한 komplex manual을 참고할 수 있을 것이다.

마지막으로 네번째 장에서, 꼭 알아야 하진 않지만, 알면 편하게 서버를 사용할 수 있는 내용들을 부록의 형식으로 첨부해 놓았다. 해당 내용 역시 가벼운 마음으로 한번씩 읽고 넘어가길 권장한다.

본 글에서는 사용자가 알아야 하는 *최소한의* 정보를 담고 있다. 작성된 내용만을 따라 서버를 사용한다면, 큰 문제없이 사용할 수 있지만, 조금 더 편리하게 사용하기 위해서는 리눅스와 쉘등에 대한 많은 공부를 해야 할 것이다. 또한 사람마다 편하게 쓸 수 있는 방식이 다르기 때문에 본 글에서 추천하는 설정은 모두 저자의 개인적인 기호에 따른 것으로 반드시 따라해야 할 필요는 없을 것이다.

매뉴얼에서 서술된 내용을 잘 이해하기 어렵다면, 언제든지 그룹원들이나 관리자에게 물어보자. komplex 서버의 구조가 일반 서버보다 사용자에게 꽤나 많은 권한을 부여하기 때문에, 뭔지 모르는 상황에서 서버의 설정을 잘못 건드릴 경우 상당히 큰 문제로 이어질 수 있으니 주의하자. 대부분

맞닥드릴 수 있는 문제 상황은 일반 리눅스 pc에서의 상황과 비슷하기 때문에, 구글링 하여 해결해 보는 것도 빠른 실력 향상의 지름길이다.

2021년 최호연

## 1.1 Linux

컴퓨터 관련 수업을 들어본 사람이라면, linux(리눅스)라는 용어를 들어본 적이 있을 것이다. 이는 Windows, Mac과 같이 운영체제(os)의 하나라고 이해하면 좋을 것이다. Windows home, Windows pro, Windows server 등과 같이 윈도우에 다양한 버전이 있는 것 처럼 리눅스 역시 다양한 버전(배포판)이 존재한다. 특히 마이크로소프트가 개발한 윈도우 os와 애플이 개발한 맥 os에 비하여, 리눅스를 만들어 오픈 소스로 공개한 리누즈 토발즈[1] 덕분에 모든 os를 구성한 코드들이 공개되어 있다. 따라서 전세계에 다양한 개발자들에 의해 유지, 보수되며 이에 따라 엄청나게 많은 배포판이 존재한다.

komplex를 비롯한 대부분의 서버에서는 이렇게 다양한 종류의 linux 배포판들 중 하나를 선택하여 사용한다. 이 뿐만 아니라 코딩을 하는 사람들에게 주 운영체제로 linux를 추천하기도 한다. 이는 linux가 다른 운영체제들에 비해 컴퓨터의 각 구성 요소(CPU, RAM 등)에 접근하기 쉽고, 다중 사용자가 동시에 다양한 작업을 돌리기 용이하기 때문이다. 특히 komplex에서는 arch linux라는 배포판을 사용한다. 이후 설명하는 모든 linux 관련 설명은 모두 arch linux에 기반한 설명이 될 것이다.

### Desktop Environment

지금까지 컴퓨터를 다루어본 사람들에게 아이콘을 더블클릭하여 프로그램을 실행하고 실행된 프로그램의 화면을 보며 작업을 하는 것이 매우 당연할 것이다. 하지만 이는 컴퓨터와 상호작용 하는 방식 중 하나인 GUI(Graphic User Interface)를 통해 컴퓨터를 사용하는 방법일 뿐이다. 이렇게 마우스를 사용하고 프로그램 화면을 띄워주는 또다른 프로그램들의 집합을 desktop environment라고 부르며, 윈도우와 맥을 설치할때 기본적으로 깔리게 된다.

반면 리눅스의 경우 desktop environment가 사용하는 자원마저[2] 아끼기 위해 GUI를 사용하지 않는 경우가 많다. 코딩, 혹은 해킹을 다루는 많은 미디어에서 검은 화면에 흰 글씨를 프로그래머를 본 적 있을 것이다. 이것을 CLI(Command Line Interface)라 부른다. 이러한 경우에는 오직 명령창을 통해 컴퓨터와 상호작용하며, 그 결과 역시 명령창을 통해 확인할 수 있다.

윈도우와 맥에서 역시 각각 powershell, terminal이라는 기본 프로그램을 통해 CLI를 제공하며, 이 외에도 CLI를 사용할 수 있는 다양한 프로그램들도 존재한다. 앞으로 이러한 프로그램을 터미널이라 부를것이며, 서버를 사용함에 있어 이러한 방식을 지속적으로 사용할 것이기 때문에, 많이 써보며 익숙해져 보도록 하자.

[1] GNU project를 개발한 리처드 스톨만의 주도로 만들어진 자유 소프트웨어 재단이 기여한 바 역시 매우 크다.

[2] 화면을 띄워야 하기 때문에 일반적으로 GPU 연산을 필요로 하며, 추가적으로 CPU작업 및 RAM이 필요하다.

## 1.2 Basic Linux Commands

상술했듯이 komplex에서는 CLI를 기반으로, 다양한 종류의 명령어를 사용해야 할 것이다. 우선 개인 pc에서의 터미널에서 이러한 명령어를 사용해 보며 연습해 보자. 개인 pc를 맥 혹은 linux를 사용한다면 후술될 명령어들을 잘 사용할 수 있겠지만, 윈도우의 경우 지원하지 않는 명령어들이 많을 것이다. 이 경우, 4.1.1절을 참고하여 WSL(Windows Subsystem Linux)를 사용하여 윈도우 상에서 리눅스 운영체제를 사용하는 것을 적극 권장한다[3]. 이후, 특별한 언급이 없는 경우 모든 상황은 리눅스 혹은 맥(유닉스)를 기반으로 한다.

이 절에서는 가장 기본적인 명령어만 서술할 것이다. 일반적으로 기본 명령어에 다양한 옵션을 주는 형식으로 작동하게 되는데, 본 매뉴얼에서는 가장 기본적인 옵션들만을 다룰 것이다. 만약 이에 대해 더 자세하게 알고 싶다면 터미널에 `$ man` *command* 혹은 구글에 man *command* 를 입력해 보자. 이외의 다른 명령어들의 경우 나올 때 마다 하나씩 알아보도록 할 것이다.

만약 리눅스 명령어에 익숙하다면 이 절의 내용을 넘어가도 무방하다.

### 1.2.1 표기 방식

앞으로 다양한 종류의 명령어들을 서술할 것이다. 이를 위해 본 문서에서는 다음과 같은 표기를 따를 것이다.

[*user*@*hostname path*]$ *command*

앞부분의 [...]의 경우 현재 터미널창에 대한 정보를 나타낸다.

- *user*: 현재 터미널 창을 운영하는 사용자 계정의 이름을 나타낸다. 본 문서에서는 개인 pc의 계정을 *me*로, 서버에서의 계정을 *newbie*라고 표기하겠다.
- @: 일반적으로 사용자 이름과 컴퓨터 이름을 이어주는 symbol이다.
- *hostname*: 현재 접속되어 있는 컴퓨터의 이름을 나타낸다. 본 문서에서는 개인 pc를 *local*로 표기하며, 서버의 경우 특별한 경우가 아니라면 터미널에서의 작업을 기본으로 하기 때문에, *terminal*로 표기할 것이다.
- *path*: 현재 명령어가 실행될 경로를 나타낸다. 경로에 대한 자세한 설명은 1.2.2를 참고하자.

이후에 나타난 $의 경우, *command*가 어떠한 권한으로 실행되어야 하는지를 나타낸다. 일반 사용자의 권한으로 실행되는 경우 $를 사용하며, 관리자 권한으로 실행해야 하는 경우 #을 쓴다. 본 글에서는 항상 $를 사용할 것이다.

마지막으로 나타나는 *command*는 실제로 컴퓨터에 명령을 내리는 내용으로, 일반적으로 배쉬 명령어를 **마젠타**로 표기한다. 이러한 기본 명령어 이외에 프로그램을 추가적으로 설치하여 사용되는 명령어의 경우 **파랑**으로 표기한다.

[3] 필자의 경우 이를 필수로 하고 싶지만 강제하긴 어려워 적극 권장으로 표현한다

### 1.2.2 Path

리눅스에서 나타내는 경로는 Absolute path(절대경로)와 Relative path(상대경로) 두가지로 나타난다. 각각의 장점이 있기 때문에, 두가지 방식 모두 익숙해 지길 바란다. 디렉토리들의 구분은 항상 /로 나타난다.

#### Absolute path

절대 경로는 이름 그대로 현재 터미널의 위치(1.2.1에서의 *path*)와 무관하게 결정되는 경로이며 항상 /로 시작된다. 예를 들어 다음과 같은 경로가 있을 것이다.

- `/` : 가장 상위 디렉토리. 일반적으로 root라고도 부른다.
- `/home/`*me* : 사용자 *me*의 홈 디렉토리. 사용자에 대한 모든 내용이 있는 디렉토리이며, 해당 디렉토리 및 그 하위 디렉토리 이외에 접근할 일은 거의 없을 것이다.

절대 경로를 사용하는 경우는, 현재 어떤 경로에 위치해 있는지와 상관없이 항상 같은 결과를 얻고 싶을때 많이 사용한다. 따라서 어떠한 상황에서도 사용할 수 있는 코드등에 많이 사용된다. 현재의 위치를 나타내는 **pwd**의 결과물 역시 절대 경로로서 표기된다.

#### Relative path

상대 경로는 현재 터미널의 위치에 따라 가리키는 위치가 바뀌는 형태의 경로이다. 이는 다음과 같이 표기될 수 있다.

- `.` : 현재 디렉토리를 가리킨다. 대부분의 경우 생략하고 사용할 수 있다.
- `..` : 현재 디렉토리의 상위 디렉토리를 가리킨다.
- `~` : 현재 사용자의 홈 디렉토리를 가리킨다. 일반적으로 `/home/`*me* 의 경로가 된다. komplex의 경우 `/pds/pdsXY/`*newbie* 가 홈 디렉토리가 된다. 홈 디렉토리의 경로에 대한 자세한 내용은 3.1절을 참조하자.

### ls

**ls**(list)는 디렉토리 [4]에 있는 항목들을 나열한다. 리눅스에서는 모든 것들이 파일 혹은 디렉토리의 형태를 띄고 있기 때문에, 모든 내용을 다 보여주는 셈이다.

**ls**는 이후 어떠한 경로도 입력하지 않으면 현재 경로에 대한 정보를 보여주며, 1.2.2절에서 서술한 경로를 특정지으면, 해당 경로의 디렉토리에 있는 내용을 모두 나열한다. 알고 있으면 좋을 옵션은 다음과 같다.

[4]폴더라는 용어에 익숙한 독자도 있을 것이다. 리눅스에서는 디렉토리라는 용어를 대신 쓴다고 이해하면 될 것이다.

- `-l` : 항목의 상세정보를 보여준다. 해당 항목이 파일인지 디렉토리인지, 권한은 어떠한지, 만들어진 날짜와 그 용량은 얼마나 되는지 등에 대한 자세한 정보가 나타난다.
- `-a` : 숨겨진 항목을 포함하여 보여준다. `.` 으로[5] 시작하는 이름을 가진 항목들은 이 옵션을 주어야 확인할 수 있다. 이들은 대부분 설정 파일 혹은 캐시 파일들이다.
- `-h` : 항목의 용량을 사람이 읽기 편한 KB, MB 등의 단위로 표시해 준다.

예를 들어 `/home/`*me* 에 속해있는 모든 파일들의 상세정보를 보고 싶다면 다음과 같은 명령어들을 사용할 수 있을 것이다.

- [*me*@*local* ~]$ `ls -a -l .` : 현재 위치( `.` )에서 `-a` , `-l` 옵션으로 항목을 표기한다.
- [*me*@*local* ~]$ `ls -al` : 현재 위치를 생략하고, `-a` , `-l` 옵션을 한번에 쓸 수 있다.
- [*me*@*local anywhere*]$ `ls ~ -a -l` : 어떤 장소에서든, 홈 디렉토리( `~` )의 항목을 `-a` , `-l` 옵션으로 표기한다.
- [*me*@*local anywhere*]$ `ls -a /home/`*me* `-l` : 어떤 장소에서든, 홈 디렉토리(절대 경로)의 항목을 표기한다. 옵션의 위치는 자유롭게 쓸 수 있다.

위와 같이 동일한 옵션 ( `-a` , `-l` )을 적용하는데 다양한 표기를 사용할 수 있다. 이는 앞으로 사용될 대부분의 명령어에도 통용되는 방식이다.

### cd

**cd**(change directory)는 현재의 위치를 옮기는 명령어이다. `cd` 는 하나의 인자를 받으며, 해당 인자가 가르키는 경로로 현재 위치를 이동한다. 경로는 상대경로와 절대 경로 모두 사용할 수 있다. 처음에는 한단계 위 아래의 디렉토리로 움직이며 **ls**를 통해 그 결과를 확인하고, 익숙해 진다면 다양하게 활용해보자.

- [*me*@*local anywhere*]$ `cd` : `cd` 이후 어떠한 경로도 쓰지 않으면 *me*의 홈 디렉토리로 이동한다.
- [*me*@*local anywhere*]$ `cd ..` : 상위 디렉토리로 이동한다.
- [*me*@*local anywhere*]$ `cd test` : `test` 라는 이름의 하위 디렉토리로 이동한다. 만약 해당 디렉토리가 없다면, `cd: no such file or directory: test` 라는 오류가 나타날 것이다. 현재 디렉토리를 생략하지 않은 `cd ./test` 도 같은 명령어이다.
- [*me*@*local anywhere*]$ `cd /home/`*me* : 절대 경로로 표현된 홈 디렉토리로 이동한다.

[5] 상대 경로를 뜻하는 `.` 와 헷갈리지 않도록 하자. `./test.txt` 의 경우 현재 디렉토리에 있는 test.txt라는 파일이며 `.test.txt` 는 현재 디렉토리에(앞에 경로를 특정짓기 않았기 때문에) 숨겨져 있는 .test.txt 라는 파일이다.

### mkdir & touch

**mkdir**(make directory)와 **touch**(touch)는 각각 빈 디렉토리와 빈 디렉토리와 빈 파일을 만드는 명령어이다. 인자로는 대상의 경로를 받는다. 두 명령어의 사용 방법은 동일하기 때문에 **mkdir**를 통한 예시를 살펴보자.

- `[me@local ~]$ mkdir test` : 현재 디렉토리(홈 디렉토리)에 test라는 이름의 디렉토리를 만든다. 이미 동일한 이름의 디렉토리가 있다면, 오류가 나타날 것이다. 역시나 현재 디렉토리를 생략하지 않은 경로인 `mkdir ./test` 역시 동일한 결과를 나타낸다.
- `[me@local ~]$ mkdir test/test2` : `~/test` 디렉토리 안에 `test2`라는 디렉토리를 만든다. 만약 `~/test`가 존재하지 않는다면, 에러가 나타날 것이다.
- `[me@local anywhere]$ mkdir /home/me/test` : 절대 경로 `/home/me`에 `test`라는 디렉토리를 만든다.

위의 예시에서 본 것 처럼 **mkdir**의 경우 가장 마지막 디렉토리 하나만을 만들 수 있다.[6] `touch`의 경우 파일을 만들기 때문에 이러한 것을 신경쓸 필요 없다. 만들어진 파일의 경우 터미널 창에서 **vim** 혹은 **nano**등의 텍스트 에디터 프로그램으로 작성할 수 있다. 자세한 내용은 후술될 text editor 부분을 참고하자.

### rm & rmdir

파일과 디렉토리를 삭제하기 위해서 **rm**(remove)와 **rmdir**(remove directory)명령어를 사용한다. 일반적으로 **rmdir**는 오직 비어있는 디렉토리 [7]를 삭제하는 데만 사용되기 때문에 **rm**를 사용하는 것이 일반적이다.

- `[me@local anywhere]$ rmdir test` : 현재 디렉토리에 있는 `test`라는 이름의 디렉토리를 삭제한다. 만약 해당 디렉토리가 없거나 비어있지 않다면 에러가 나타난다.
- `[me@local anywhere]$ rm test.txt` : 현재 디렉토리에 있는 `test.txt`라는 이름의 파일을 삭제한다.
- `[me@local anywhere]$ rm -r test` : 현재 디렉토리에 있는 `test`라는 이름의 파일 혹은 디렉토리를 삭제한다. `-r` 옵션은 recursive의 약자로, 만약 `test`가 디렉토리일 경우, 그 내부에 있는 모든 항목까지 삭제하며 해당 디렉토리까지 지운다. 매우 강력한 명령어로 조심해서 사용하여야 한다.
- `[me@local anywhere]$ rm -rf test` : 이전 명령어에서 force를 뜻하는 `-f` 옵션이 추가되었다. 이 경우, read-only등의 권한을 가진 파일까지 강제로 삭제한다. 파일 권한에 대한 것은 이곳에서 다루지 않는다.

[6] `-p` 옵션을 통해 상위 디렉토리가 없다면 한번에 만들수도 있다.
[7] `ls -a`로 확인하였을 때 항목이 나오지 않는 경우

### cp & mv

파일 및 디렉토리의 복사와 잘라내기를 해주는 **cp**(copy)와 **mv**(move)를 살펴보자. 두개의 명령어는 *source*와 *target*이라는 경로의 형태의 인자를 받는다. **cp**는 *source*를 *target*으로 복사해 주며, **mv**는 *source*를 *target*으로 이동해준다. 두 명령어의 사용 방법은 완전히 동일하기 때문에 **cp**를 통한 예시를 살펴보자.

- `[me@local anywhere]$ cp ../test.txt test2.txt` : 상위 디렉토리에 있는 `test.txt`라는 파일을 현재 디렉토리의 `test.txt`라는 파일로 복사한다.
- `[me@local anywhere]$ cp -r ../test .` : 상위 디렉토리에 있는 `test`라는 파일 혹은 디렉토리를 현재 디렉토리로 복사한다. `-r` 옵션은 **rm**와 마찬가지로 대상이 디렉토리라면 해당 디렉토리에 속해 있는 모든 항목을 복사해 온다. *target*에 항목의 이름을 특정짓지 않는다면, 동일한 이름으로 항목이 복사된다.

**mv** 역시 위와 같은 규칙으로 작동하며, 이를 활용하면 파일 혹은 디렉토리의 이름을 바꿀 수 있다.

```
[me@local anywhere]$ mv test test2
```

위의 명령어는 현재 위치에 있는 `test`라는 항목을 현재 위치에 `test2`라는 항목으로 옮긴 것이다. 즉, 항목이 존재하는 위치는 현재 디렉토리로 일정하고 그 이름만을 `test2`로 바꾼것이라 이해할 수 있다.

### htop

자신이 작업을 돌렸을 때, 그 상태를 보고 필요하다면 작업을 중지하는데 사용될 수 있는 것은 **htop** 명령어를 통해 이루어 진다. `$ htop`을 치면, 현재 컴퓨터의 코어 사용량과 메모리 사용량이 가장 위에 나오고, 아래쪽에 프로세스들이 쭉 나열되어 있는 것을 볼 수 있을 것이다. 각 프로세스들을 실행한 사용자가 USER에 적혀있고, 자신의 계정 이름이 하얗게 highlight 되어있을 것이다. 이를 보고 현재 돌아가는 작업을 확인하고, 필요하다면 해당 프로세스를 선택하고 F9-enter를 눌러 그 프로세스를 종료할 수 있다.

### cat

파일의 내용을 확인하는 명령어는 **cat**이다. 예를들어 `cat test.txt`은 터미널 창에서 `test.txt` 파일의 내용을 보여주는 명령어이다. 또한, 파일의 내용을 pipe, redirection등으로 다른 명령어에 인자로 받거나 다른 파일에 내용을 넣을 수도 있을 것이다.

현재 komplex 서버에서는 같은 기능을 하지만 syntex highlighting까지 지원해주는 **bat**도 같이 사용되고 있다.

### asterisk

asterisk는 *을 뜻하며, 일종의 wildcard와 같은 역할을 한다. 즉, 1.2.2에서 사용한 path중 일부분을 *으로 치환한다면, 그 패턴에 맞는 모든 종류의 경로를 뜻한다. 예를 들어 다음과 같다.

```
$ rm *.txt
```

위의 명령 같은 경우에는 현재 디렉토리에 있는 파일 중 .txt로 끝나는 이름을 가진 모든 파일을 삭제하라는 명령어이다.

### text editor

텍스트 에디터는 말 그대로, 텍스트로 이루어진 파일을 수정할 수 있는 프로그램을 뜻한다. 일반적으로 윈도우에서의 메모장등을 생각하면 편할 것이다. 하지만 서버에서는 GUI없이, 터미널창에서 텍스트 파일을 편집해야 할 경우가 있을 수 있다. 이를 위해 간단하게라도 터미널에서 텍스트 파일을 수정하는 방법을 알고 있으면 편리할 것이다. 대표적인 두가지만 소개하자면 다음과 같다. 사용방법은 인터넷에서 찾아보자.

1. **nano**: 가장 기본적인 텍스트 에디터이다. 터미널에서의 작업이 익숙하지 않은 사람이라면 가장 쉽게 (그나마) 직관적으로 사용할 수 있다. 다만 별다른 설정을 하지 않는다면 syntex highlighting등을 지원하지 않기 때문에 정말 간단한 파일의 수정할때만 사용하는 것을 추천한다.

2. **vi**, **vim**: 터미널 텍스트 에디터라고 하면 가장 대표적으로 떠오르는 프로그램이다. 복잡한 단축키들을 사용하는데, 이에 익숙해지기 위해서는 조금의 시간이 필요할 것이다. 다양한 종류의 플러그인등을 설치하여 개인별 맞춤 설정을 하여 사용할 수 있어 상당히 인기가 좋다.

**vim**이외에도 **emacs**라는 프로그램 역시 매우 유명하며, 이를 통해 코딩을 하는 사람들도 많이 있다. 하지만 필자를 비롯하여 많은 연구실 분들은 2.2.4절에서 소개하는 VSCode를 사용하는 것을 훨씬 추천한다.

### environment

서버에서 작업을 하다보면 반복적인 일을 하게 될 경우가 생긴다. 이런 경우를 도움이 될 만한 몇 가지는 방법이 있다.

- bash autocompletion: 자동완성기능(auto-completetion) 사용하는 것이다. 주로 'Tab' 키를 이용해서 자동완성기능을 사용한다. autocompletion를 제공하는 패키지를 설치할 수도 있고 수동으로 설정하는 방법도 있다.

- bash alias: bash alias는 단축키 지정과 같은 효과를 준다. `~/.bashrc`에서 현 계정에서 사용하는 bash alias들을 확인할 수 있다. bash alias를 이용해서 **ls**나 **grep**의 coloring option

을 항상 켜지도록 할 수 있다. **source**와 같이 사용하면 alias를 불러올 수도 있는데 여러모로 유용하니 기억해두자.

- bash history: 배쉬는 기존에 받은 명령어들은 기록해두는데 이를 bash alias를 이용하면 기존의 사용했던 명령어들을 쉽게 불러올 수 있게 할 수 있다. 물론 history를 불러오는 다른 방법도 있지만 이 방법이 가장 간단한 것 같다.

대체로 각각의 기능들을 설정하는 방법과 사용하는 방법은 아치리눅스 위키를 참고하도록 하자.

## scripting

배쉬에서 작업하다보면 명령어가 길어지게 되어 있고 명령어줄에서 작업하는게 불편해지는 시점이 온다. 특히 반복적인 일을 하거나 더 추상화된 일을 하기 위해서는 `for` 문, `if` 문등을 사용한 배쉬 스크립팅은 필수적인 작업이 될 것이다. 배쉬 스크립팅과 관련해서 다음 사이트를 참고하자.

- Greg's BashGuide (`https://guide.bash.academy/`)

참고로 보기 좋게 배쉬 스크립팅하는 건 정말 어렵다. 그럼에도 불구하고 awk와 gnuplot 그리고 배쉬를 이용한 스크립트만으로 시뮬레이션을 제외한 연구에 대부분의 일이 가능하고 배쉬 스크립팅을 배워두면 서버에서 작업도 더욱 용이하고 효율적으로 할 수 있으므로 배워서 나쁠게 없다.

### Shebang and POSIX

대체로 배쉬 스크립트의 첫머리가 `#!/bin/bash`로 시작하는 것을 볼 수 있다. 이것을 shebang line 혹은 bang line이라고 부른다. shebang line은 쉘이 배쉬 스크립트임을 알게된다. sawk나 gnuplot의 스크립트 역시 shebang line을 이용해서 스크립트임을 알려준다. 간혹 shebang line이 없거나 `#!/bin/sh`으로 시작하는 경우에 다른 운영체제들과 호환성을 높인 포직스(POSIX)를 이용하는 스크립트들이다.

# Chapter 2

# Connect to Server

일반적으로 서버에 접속하기 위해 매우 다양한 방법들이 존재한다. 이번 챕터에서는 필자 스스로 여러 시행착오를 겪고, 가장 최적이라고 판단한 방법을 소개하겠다. 이외에도 이미 본인이 알고 있거나 사용하고 있는 방법이 있다면 그대로 사용해도 큰 문제가 없을 것이다. 그렇지 않다면, 우선은 필자를 믿고 서술된 방법을 따라가자. 시간이 지남에 따라 이 방법이 최선이 아닐 수도 있으며, 혹시 본인이 찾은 방법이 조금 더 괜찮다고 생각한다면, 내용을 수정해도 될 것이다.

## 2.1 Account

서버에 접속하기 위해서는 우선 서버에 본인의 계정이 있어야 한다. 서버 관리자에 문의하여 계정을 생성하도록 하자. 계정을 생성하는데는 이름과 비밀번호가 필요하며, 일반적으로 이름은 15자 이내의 소문자 영문 혹은 숫자[1]를 포함할 수 있다. 비밀번호는 본인만 알 수 있는 번호로 각각 한개 이상의 영문과 숫자, 그리고 특수문자를 조합한 12자리 이상로 이루어져야 한다. 이때 비밀번호에는 계정의 이름이 들어가지 않도록 주의하자.

관리자로부터 계정 생성안내를 받으면 이제 서버에 접속할 준비가 모두 된 것이다.

## 2.2 SSH

komplex 서버는 서울대학교 22동 414호에 있기 때문에, 서버를 사용하는 거의 모든 작업은 원격접속을 통해 이루어진다.[2] 이 때문에, 원격접속을 위한 환경을 세팅하는 것이 필수적이다.

이제부터는 개인 pc에서 하는 작업과 서버에서 하는 작업을 잘 구분해서 확인해야 한다. 이를 위해 개인 pc를 *local*, 본인 컴퓨터에 로그인 된 계정의 이름을 *me*라 하자. 또한 서버(터미널) 컴퓨터를 *terminal*, 서버에서 생성된 계정의 이름을 *newbie*라 표기하겠다. 1.2.1절의 내용을 다시 한번

[1] 사용자의 실명을 기반으로 정하는 것을 추천한다.

[2] 실제로 서버를 보수할 때를 제외하고 서버실에는 거의 들어가지 않는다.

확인하고 돌아오자.

서버는 SSH(Secure Shell)이라는 방식으로 접속하는 것을 기본으로 한다. *local*의 터미널에서 다음과 같은 명령어를 쳐보자.

```
[me@local ~]$ ssh newbie@komplex.terminal.snu.ac.kr -p 56
```

위의 명령어는 *newbi*라는 user 이름으로 `komplex.terminal.snu.ac.kr` 이라는 서버에 56번 포트로 접속하라는 의미이다.

처음으로 서버에 접속할 경우 fingerprint를 등록하라 라는 메세지가 나올텐데, yes라고 해주면 된다. 이는 *local*에 서버의 fingerprint 정보를 저장하는 것으로, 만약 동일한 주소로 들어갔지만 이전과 다른 fingerprint를 확인하게 된다면 서버와의 접속 혹은 서버 자체에 보안상 문제가 있다고 보고 접속을 차단하게 된다.

Windows의 경우 powershell에서 위와 같은 명령어를 칠 것인데, 만약 ssh 명령어를 찾을 수 없다는 에러가 나온다면, 선택적 기능 설치에서 openssh와 관련된 기능들을 모두 설치해 주자. 1.2절의 초반부에 소개하였던 WSL의 ubuntu 배포판의 경우 `# sudo apt install openssh-clinet` 등으로 설치해 주면 된다.

Fingerprint 메세지를 지나치면, 계정의 비밀번호를 물을 것이다. 서버에 등록한 비밀번호를 성공적으로 쳤다면 이제 서버에 접속하였을 것이다. 다음과 같은 형태의 창이 뜬다면 접속에 성공한 것이다.

```
[newbi@terminal ~]$
```

이제 1.2절에서 배웠던 기본적인 리눅스 명령어들을 사용하며 서버를 돌아다녀 보자.[3] 서버에서 나오기 위해서는 `exit` 명령어를 치거나 Ctrl-D를 누르면 나올 수 있다.

매우 한정적인 상황에서 터미널이 아닌 다른 서버로 들어가야 할 경우도 있는데, 이때는 `ssh` *disk* 와 같이 원하는 서버 노드의 이름으로 들어갈 수 있다. 다른 노드는 모두 접속해도 괜찮지만, 마스터와 자신의 홈 디렉토리가 위치해 있지 않은 디스크 서버로는 들어가지 말자. 서버 구조에 대한 자세한 내용은 3.1절에서 소개할 것이다.

### 2.2.1 ssh key

서버에 접속할 때 마다 비밀번호를 치는것이 번거로울 수 있을 것이다. 특히, 보안상의 이유로 komplex 서버에서는 비밀번호의 조건을 까다롭게 설정해 두었기 때문에, 특히 귀찮을 수 있다. 만약 실수로 비밀번호를 과도하게 많이 틀린다면, 24시간동안 해당 계정의 로그인이 제한될 수 있으며 이 경우 관리자에게 문의하여 로그인 제한을 풀어달라고 요청해야 한다.

이러한 문제를 막기 위해, ssh key 라는 것을 사용할 수 있다. 이는 자물쇠와 열쇠에 해당하는 한 쌍의 key를 만들고, 자물쇠를 서버에 등록하여 그에 맞는 열쇠를 가진 컴퓨터의 경우 비밀번호 인증

[3] 다른 계정의 홈디렉토리는 들어가지 못한다. 다만 관리자는 다른 사람의 홈 디렉토리에 들어갈 수 있다는 점을 명심해야 한다.

없이 로그인을 허용해 주는 것이다. 일반적으로 비밀번호보다 관리가 쉽고, 유출의 위험이 덜하기 때문에 권장하는 방법이다.

ssh key를 활용한 서버로의 접속방법은 다음과 같다. 다시 한번 강조하지만, 개인 pc에서 하는 작업과 서버에서 하는 작업을 잘 구분해서 확인하자.

1. 개인 pc에서 자물쇠와 열쇠쌍을 만든다. 만약 이전에 만들어 놓은 ssh key가 있다면, 다시 만들지 않아도 괜찮다. 동일한 자물쇠 열쇠 쌍을 여러개의 서버에 사용하는 것은 문제가 되지 않는다.[4]

   [*me*@*local* ~]$ ssh-keygen -t rsa -b 4096

   `-t rsa` 옵션은 ssh key의 종류중 RSA 방식의 암호화를 사용한 키를 뜻한다. `-b 4096` 은 암호 복잡도를 4096 bit를 사용한다는 뜻이다. default 값인 2048보다 길이가 긴 키가 만들어진다. 이후 키를 저장할 경로와 paraphrase를 입력하라는 메세지가 뜨는데, 이곳에 아무것도 치지 않고 넘어가도 무방하다.

2. 한쌍의 key는 `~/.ssh` 디렉토리에 `id_rsa`, `id_rsa.pub` 로 만들어 질 것이다. 이것이 있는지 확인하자. `id_rsa` 는 private key라고 부르며, 열쇠에 해당한다. 이는 결코 외부(인터넷)에 유출되어서는 안될 것이다. 반면 `id_rsa.pub` 은 public key라고 부르며, 자물쇠에 해당한다.

3. public key를 인터넷을 통해 서버에 복사하자. 자물쇠는 외부에 공개되어도 되기 때문에, 인터넷을 통해 자유롭게 왔다갔다 할 수 있다.

   [*me*@*local* ~]$ scp -P 56 ~/.ssh/id_rsa.pub *newbi*@komplex.terminal.snu.ac.kr:~

   이때 사용하는 명령어는 **scp**(secure copy)로, 1.2절에서 소개한 **cp**의 서버 버전이라고 이해하면 된다. 또한 서버 도메인 뒤의 : 문자의 경우, 도메인이 끝났으며, 이후에 나오는 경로는 해당 인터넷 주소 내부에서의 경로를 지칭한다. **ssh**와 다르게 포트를 특정할 때 소문자 p가 아닌 대문자 P를 쓴다는 것을 유의하자.

4. 이제 서버에 접속하여 자신의 홈 디렉토리에 public key가 잘 도착했는지 확인하자.

   [*newbi*@*terminal* ~]$ ls id_rsa.pub

   위의 명령어에서 항목이 나온다면 잘 도착한 것이다.

5. public key를 서버에 등록하는 방식은 다음과 같다.

   [*newbi*@*terminal* ~]$ cat id_rsa.pub >> ~/.ssh/authorized_keys

   이 작업은 `id_rsa.pub` 파일의 내용을 `authorized_keys` 에 append하는 작업으로, 명령어가 이해가지 않더라도 괜찮다.

[4] 물론 한번 열쇠가 유출된다면 동일한 자물쇠로 잠겨있는 모든 서버들이 위험해 질수 있다.

6. 이제 모든 작업이 완료되었다. 새로운 터미널을 열어 비밀번호 없이 서버에 접속되는지 확인하자.

### 2.2.2 ssh config

위의 작업을 하며 **ssh**를 통해 몇번씩 서버에 접속했다 나오고, **scp**를 통해 서버에 파일을 복사하는 작업을 하다 보면, 서버에 접속하는 명령어가 상당히 길다는 것을 느꼈을 것이다.[5] 이러한 사람들을 위해 미리 **ssh** 정보를 config 파일 안에 저장하고, 이를 사용하는 방식을 소개한다.

이 작업은 개인 pc에서 진행하여야 한다. 개인 pc의 `~/.ssh/config` 파일에 다음과 같이 써보자.

[*me*@*local*]~/**.ssh/config**

```
Host komplex
    HostName komplex.terminal.snu.ac.kr
    User newbie
    Port 56
```

파일을 저장한 다음, `ssh komplex` 라고 치면, komplex 호스트 항목에 있는 configuration들이 적용되어 **ssh**가 실행된다. 만약 ssh key까지 등록을 해 놓았다면 이 명령어로 바로 서버에 접속까지 될 것이다. 이러한 configuration은 **ssh** 뿐만 아니라 상술한 **scp**, 후술될 **sshfs**등의 명령어에서도 모두 공유되기 때문에 2.2.4절, 3.3.2절 등에서도 지속적으로 사용될 것이다.

현재 설명할 것 이외에도 다양한 configuration option들을 특정할 수 있는데, 이 중 ProxyJump에 대해 알아보자. 이는 이후 disk나 worker 노드에 곧바로 접속하기 위해 매우 유용할 것이다.

[*me*@*local*]~/**.ssh/config**

```
Host disk
    HostName disk
    User newbie
    ProxyJump komplex
```

이때 *disk*는 자신의 홈 디렉토리가 위치한 디스크 서버를 뜻한다. 위와 같이 configuration을 적용한 다음 `$ ssh disk` 를 한다면, 곧바로 디스크 서버로 접속되는 것을 볼 수 있을 것이다. 이는 반드시 이전 komplex 호스트 항목 내용을 proxy로 하여 접속을 하기 때문에 가능한 것이다.[6]

### 2.2.3 sshfs

기본적으로 서버에 개인 pc의 파일을 업로드 하거나 다운로드 하는 과정은 2.2.1절에서 다룬 **scp**를 통해 진행한다. 하지만 명령어를 통해 파일을 옮기는 작업은 꽤나 번거로울 수 있다. 이에,

[5] 심지어 비밀번호 치기도 귀찮은데 모든 명령어를 다 쓰기는 더 귀찮다.

[6] ProxyJump 옵션이 적용되지 않는 경우, 현재 사용하는 **ssh**의 버전이 최신이 아닌 것이다. 보안상의 이유 때문에라도 **ssh**는 최신 버전으로 유지하는 것이 좋기 때문에 가능하다면 업데이트를 하자. 만약 업데이트를 할 수 있는 상황이 아니라면, 다음과 같은 옵션으로 ProxyJump를 대신할 수 있다. `ProxyCommand ssh komplex -W %h:%p`

직접 서버에 접속하지 않고도, 서버의 디스크를 개인 pc에 마운트 하여 코드를 작성하거나 데이터 및 그림등을 확인 할 수 있는 **sshfs**(ssh file system)를 소개하고자 한다. 이 프로그램은 ssh로 접속 가능한 서버의 파일 시스템을 현재 로컬 컴퓨터에 마운트 할 수 있게 해주는 프로그램이다.

윈도우의 경우와 리눅스/맥의 경우에 따라 하는 방식이 다르기 때문에 각자 해당하는 방식을 따르자.

### Windows

윈도우에서는 기본적으로 **sshfs** 기능을 지원하지 않는다. 따라서, 다음과 같은 세개의 프로그램을 순서대로 설치해야 한다. 모두 github 링크에서 설치할 수 있을 것이다.

1. winfsp: `https://github.com/billziss-gh/winfsp`
2. sshfs-win: `https://github.com/billziss-gh/sshfs-win`
3. sshfs-win-manager: `https://github.com/evsar3/sshfs-win-manager`

위의 프로그램을 모두 순서대로 설치하였다면, SSHFS-Win-Manager라는 프로그램이 설치되었을 것이다. 해당 프로그램을 열고 add connection을 통해 komplex 서버의 configuration을 하자. 이때 중요한 점은 Remote path를 자신의 홈 디렉토리 절대 경로로 해야 한다. Configuration을 save 하고 connect를 한다면 윈도우의 파일 탐색기에서 자유롭게 확인할 수 있을 것이다.

한가지 주의할 점은, 윈도우에서는 리눅스와 다르게 파일의 권한에 대한 설정이 매우 엉망이다. 따라서 위의 방법으로 연결된 서버에 바로 파일 혹은 디렉토리를 드래그&드랍 혹은 ctrl+c,v 로 복사한다면, 파일의 권한에 대한 문제가 생길 가능성이 매우 높다.[7] 이 매뉴얼에서는 파일의 권한에 대해 따로 다루지 않기 때문에, 필요할 경우 리눅스 파일 권한 혹은 **chmod**등의 명령어를 검색하여 찾아보도록 하자.

### Linux, Mac

리눅스와 맥(유닉스)에서는 별다른 설치 없이 **sshfs**를 사용할 수 있다. 개인 pc에 **sshfs** 패키지가 설치되어 있지 않다면 우선 설치해주자. 그리고, 2.2.2절에서 다룬것처럼 ssh config 파일에 디스크에 해당하는 부분을 적어두자.

그리고, 서버가 마운트 될 디렉토리를 만들어 주어야 한다. 일반적으로 비어있는 디렉토리에 마운트를 하며, 서버에서의 절대경로와 동일한 경로로 만드는 것을 적극 권장한다. 이는 다양한 코드들에서 절대 경로를 사용한 경우 발생할 수 있는 문제를 최소화 함에 있다. 예를들어 서버에서 자신의 홈 디렉토리가 `/pds/pds`$XY$`/`*newbie* 일 경우[8], 본인의 로컬 컴퓨터에서 다음과 같이 디렉토리를 만들고 마운트 할 수 있다.

[7] 특히 디렉토리(폴더)를 통째로 복사 붙여넣기 할 경우에 문제가 될 수 있다.

[8] 자신의 홈 디렉토리의 경로는 `[`*me*`@`*local* `~]$ pwd` 를 통해 알수 있다.

```
$ mkdir -p /pds/pdsXY/newbie
$ sshfs disk:/pds/pdsXY/newbie /pds/pdsXY/newbie
```

첫번째 명령을 통해 빈 디렉토리를 만들며, 두번째 명령어를 통해 `~/.ssh/config`에 적혀있는 disk 호스트 항목의 configuration을 사용하여 **sshfs**를 한다. **sshfs**의 경우 개인 pc의 비밀번호를 요구할 수도 있다.

명령어가 모두 성공적으로 수행되었다면, 이제 개인 pc에서 서버의 디렉토리를 탐색할 수 있는지 확인해 보자.

### 2.2.4 VSCode

Visual Studio Code는 Microsoft사에서 만든 텍스트 에디터로, 다양한 종류의 extension를 통한 커스터마이징을 지원한다. 이는 연구에 사용되는 대다수의 프로그래밍 언어들(C++, Python 등) 뿐만 아니라 논문 작성을 위한 latex, 코드 관리를 위한 git 지원등 매우 다양한 기능들이 존재하기 때문에, 실제로 연구와 관련된 일을 할 때 VSCode로 안되는 작업을 찾기가 어려울 정도이다. 이 챕터에서는 VSCode에서 지원하는 다양한 기능 중, 서버 접속에 대한 기능만에 한정하여 다룰 것이다. 이외에 추가적인 팁이 필요할 경우, 4.3를 참고하자

VSCode를 통해 서버에 접속하기 위해서는, 우선 Remote-SSH라고 하는 extension을 설치하여야 한다. 설치를 완료하면 가장 왼쪽에 위치해 있는 active bar에서 remote 접속에 대한 메뉴가 생길 것이다. Remote explorer에서 SSH target을 찾자. 2.2.2절의 내용을 잘 따라했다면[9], 한두개의 ssh Host들이 보일 것이다. 해당 *Host*을 선택하면, 서버에서 작동하고 있는 새로운 VSCode 창이 열리게 된다.

처음 서버에 VSCode를 통해 접속하게 된다면, 관련 설정파일들을 설치하는데 시간이 조금 걸릴 수 있기 때문에 기다려보자. 창의 가장 왼쪽 아래에, SSH:*Host*가 보이는지를 확인한다면, 지금 서버에 잘 접속해 있는 것이다.

마지막으로, 이미 VSCode에 몇가지 extension을 설치하였더라도 서버에 접속할때 다시 한번 서버측에 설치해 주어야 하는 extension들이 있다. 대표적으로 프로그래밍 언어 분야의 C/C++, Python 등의 extension들이다. 이러한 extension을 설치할 때, 3.3절에서 언급하는대로, 디스크 서버에 바로 접속하여 설치하는 것이 훨씬 빠를 것이다. 2.2.2절에서 이미 디스크에 대한 configuaration을 해두었다면, VSCode의 ssh Host에도 디스크 서버가 보일 것이다. 우선 디스크 서버에서 인터넷이 되는지 확인하고[10], 디스크로 들어가 extension을 설치하자.

[9] 윈도우의 경우 WSL에서의 ssh config가 아닌, windows에서의 ssh config를 기본으로 사용한다. WSL에서의 설정만 하였다면, 다시 돌아가 윈도우에서의 config 파일도 동일하게 써주자.

[10] `ping www.google.com` 등으로 확인할 수 있다. 인터넷이 되지 않을 경우 관리자에게 문의하자. 디스크에서 외부 인터넷을 써야 하는 작업이 모두 끝난다면 반드시 관리자에게 다시 말하여야 할것이다.

# Chapter 3

# Using Komplex

지금까지 알아보았던 내용들은 대부분 꼭 komplex 서버가 아니더라도 일반적으로 SSH 접속을 지원하는 서버들에 모두 통용되는 방법이었다. 연구실에서 사용하는 komplex 서버도 다른 서버들과 거의 다를 바 없지만, 몇몇 komplex에 특화된 내용들이 있기에 이를 알아두어야 할 것이다.

## 3.1 Structure

komplex 서버는 수백개의 컴퓨터로 이루어져 있는데, 컴퓨터들은 각각의 역할에 따라 다음과 같이 나뉘어져 있다.

- Master: 마스터 서버는 komplex에서 1대만 존재하는 컴퓨터로 계산 노드들에 os를 제공해 주는 역할을 하는 '뇌'의 역할을 한다. 이곳에서 문제가 생긴다면 서버 전체에 영향을 끼칠 수 있기 때문에, 일반적인 경우에는 관리자만이 접근하여야 하며 사용자는 접속하지 않아야 할 것이다.
- Terminal: 사용자들이 외부에서 접속할 수 있게 해주는 컴퓨터이다. 연구실에서 사용할 대부분의 작업은 터미널에서 진행하게 될 것이다.
- Disk: 사용자의 저장공간이 있는 컴퓨터이다. 2.2.4, 3.3.1절과 같이 극히 일부분의 경우를 제외하고 사용자가 접근할 일은 없다. 특히나 자신의 홈 디렉토리가 있는 디스크가 아닌 경우는 가능하면 들어가지 말자.
- CPU worker: CPU를 활용하는 계산을 하는 컴퓨터이다.
  - tenet: 데스크탑 cpu가 장착된 컴퓨터들이다. 4코어에서 8코어의 cpu가 장착되며, 8-64G의 시스템 메모리(RAM)이 장착되어 있다. 일반적인 cpu 계산을 돌릴때 사용한다.
  - xenet: 서버용 Xeon cpu가 장착된 컴퓨터들이다. 대부분 10코어 이상을 가지고 있는 cpu가 장착되며 최소 64G-1.5T의 시스템 메모리(RAM)이 장착되어 있다. 다만 싱글 코어의

성능이 tenet보다 (매우) 떨어지기 때문에, 많은 메모리를 사용해야 하는 시뮬레이션이나 병렬 컴퓨팅을 사용한 코드에 적합하다.

- GPU worker: GPU를 활용하는 계산을 하는 컴퓨터이다. CPU 계산도 가능하지만, 일반적으로 거의 수행하지 않는다. 현재 kuda라는 이름을 가지고 있으며, NVIDIA의 그래픽카드들을 장착하고 있다.

서버에 어떤 종류의 컴퓨터들이 연결되어 있는지를 확인하기 위해서는 3.2절에서 소개될 `spg`를 사용하여 다음과 같이 확인할 수 있다.

```
[newbi@terminal ~]$ spg list
```

주의할 점으로, 현재 xenet9에는 1TB SSD와 RAID5로 묶인 16TB의 하드 디스크가 있으며, 각각 `/xds/xds1`, `/xds/xds2`의 경로로 mount 되어 있다. 일반적인 disk server를 사용할 때 네트워크 IO 병목을 느낀다면, 이 저장장치를 사용하도록 하자. 해당 pc에 local하게 설치된 저장 장치이기 때문에, Disk server와 다르게 다른 노드들에서는 접근할 수 없으며, 홈 디렉토리로도 사용할 수 없다.

## 3.2 SPG

SPG(Statistical Physics Group)는 작업 스케줄러로 언제부터 이렇게 이름이 붙여져 쓰여졌는지는 모른다. 현재 komplex서버에서 사용되는 SPG는 이전 최철호 박사님께서 파이썬으로 작성한 코드를 2021년 8월경에 필자가 refactoring하여 사용하고 있다. 현재 오픈소스로 공개하고 있으며, 소스코드는 깃허브[1]에서 확인해 볼 수 있다.

`help` 문서들이 모두 작성되어 있기 때문에 `spg -h` 혹은 `spg` *option* `-h`등을 친다면 관련 documentation을 모두 볼 수 있을 것이다. 본 매뉴얼의 내용과 각 옵션들의 help는 **반드시** 숙지하고 사용해 보자.

`list`, `run` 옵션을 제외하고 모든 명령이 모두 네트워크를 통해 현재 서버의 상태를 확인하는 작업을 한다. 따라서, 자주 `spg`명령어를 사용한다면 서버의 전반적인 네트워크 과부하를 줄 수 있기 때문에 필요한 경우에만 한번씩 사용하도록 하자.[2]

### spg list

`spg list`는 현재 spg에 등록되어 있는 컴퓨터들을 나열해 준다. 3.1절에서 언급한 노드들 중, worker 노드들에 대한 정보가 그 종류에 따라 나뉘어 보여진다. 구체적으로는 노드의 이름과 cpu 종류, 그리고 cpu 코어 개수와 장착된 메모리의 크기를 보여준다. 가장 아래쪽에 전체 summary를 확인할 수 있다.

[1] `https://github.com/hoyunchoi/SPG`

[2] 예를 들어 10초에 한번씩 `spg free`를 사용하여 현재 남아있는 서버 자원을 확인한다면 서버 전체에 매우 심각한 네트워크 부하를 줄 수 있다.

• `-g`, `--groupList` 혹은 `-m`, `--machineList` 옵션을 통해 worker 그룹 혹은 worker 노드를 특정하여 프린트 할 수 있다.

### spg free

`spg free`는 spg에 등록되어 있는 노드들중, 사용 가능한 컴퓨터들을 나열해 준다. `list` 옵션과 마찬가지 정보들을 보여주지만, 이번에는 현재 가용 가능한 자원들만을 보여주게 되어 있다.
• `-g`, `--groupList` 혹은 `-m`, `--machineList` 옵션을 통해 worker 그룹 혹은 worker 노드를 특정하여 프린트 할 수 있다.

### spg job

`spg job`은 현재 서버에서 돌아가고 있는 작업들을 보여준다. 아무런 옵션을 주지 않았을 경우 자신의 job 내용을 볼 수 있다. 구체적으로 보여지는 내용은 다음과 같다.

- Machine: 해당 job이 돌아가고 있는 노드의 이름
- User: 해당 job을 돌린 계정 이름. `spg job`을 사용하였다면 본인의 이름만이 나올 것이다.
- ST: 해당 job의 state. 만약 이 값이 D일 경우, 디스크 혹은 네트워크등의 IO 문제가 있는 것으로, 자신의 코드를 확인해 보고 필요할 경우 관리자에게 말해야 한다.
- PID: 해당 job의 PID(Process ID) 번호. 후술될 `spg KILL -p` *pid* 에서 사용할 수 있다.
- CPU(%): 해당 job의 cpu 사용량. 처음 시작할때 부터 지금까지 평균 cpu 사용량을 표기해 준다.
- MEM(%): 해당 job의 메모리 사용량. 해당 노드에 장착된 메모리 대비 얼마의 메모리를 사용하고 있는지를 알려준다.
- Memory: 해당 job의 메모리 사용량. 절대적인 메모리 사용량을 알려준다.
- Time: 해당 job이 돌아가고 있는 시간. HH:MM:SS의 형식으로 표시되며, 하루가 넘어갈 경우 DD-HH:MM:SS의 형태로 표기된다.
- Start: 해당 job이 시작된 시각. HH:MM의 형식으로 표시되며, 하루가 넘어갈 경우 시작된 날짜를 표기한다.
- Command: 해당 job의 command를 보여준다. 이를 통해 구체적으로 어떤 job인지를 확인할 수 있을 것이다.

• `-a` 옵션을 통해 모든 사용자가 돌리고 있는 job의 내용을 볼 수 있다. 자신 이외의 특정 사용자가 돌리는 job을 내용을 보고 싶다면 `-u` *user* 옵션을 사용할 수 있다
• `-g`, `--groupList` 혹은 `-m`, `--machineList` 옵션을 통해 worker 그룹 혹은 worker 노드를 특정하여 프린트 할 수도 있을 것이다.

### spg user

`spg user`는 현재 서버에서 작업을 돌리고 있는 사용자의 통계를 보여준다. `spg job -a`과 비슷하지만 사용자 별로 돌리고 있는 job의 개수를 보여주게 된다.

- `-g, --groupList` 옵션을 통해 특정 worker 그룹을 특정하여 프린트 할 수 있다.

### spg run

`spg run`은 서버 노드에 작업을 돌리는 명령어이다. 다음과 같은 순서로 옵션을 받는다.

```
$ spg run [machine name] [program] (arguments)
```

내가 계산을 돌리고 싶은 서버 노드를 선택하고, 명령어를 적는다. 코딩 방식에 따라 해당 명령어에 추가로 인자를 주어야 하는 경우 그대로 이어서 쓰면 된다. 몇가지 주의사항은 다음과 같다.

- 현재 자신이 위치해 있는 디렉토리의 기준으로 입력된 명령어를 실행한다. 만약 절대경로로 코딩을 하였다면 상관 없겠지만, 상대경로로 코딩을 하였다면 정확한 위치에서 `spg run`을 해야할 것이다.
- 백그라운드에서 작업을 돌리게 해 주는 & suffix를 붙이지 말자. 기본적으로 `spg`에서 이를 해주기 때문이다.
- redirection을 할 경우 '<'와 같이 quotation 사이에 표기를 넣어야 한다.

### spg runs

`spg runs`는 `run`의 변형으로, 여러개의 작업을 동시에 돌려준다. 돌리고자 하는 명령어를 `commands.txt`와 같은 파일에 line seperate하게 미리 작성해 두자. 그리고 머신 그룹을 특정하여 작업을 돌리면 된다.

```
$ spg runs [command file] [group name] (start end)
```

이를 사용하기 위해 알아야 할 점은 다음과 같다.

- `[group name]` 뒤에 start와 end 번호를 같이 넣을 수 있는데, 이 경우 해당 번호를 포함하여 그 사이에 있는 계산 노드들에 작업을 할당시켜준다.
- `commands.txt`에 들어가는 명령어는 모두 `spg run`에 적힌 명령어 제한을 동일하게 받는다.
- 남아있는 코어의 개수가 없거나 `commands.txt`에 적힌 명령을 모두 처리하면 작업이 종료된다. 컴퓨팅 자원이 남아있더라도 한번에 돌리는 명령의 최대 개수가 50개로 제한되어 있다는 점에 주의하자.
- 수행된 명령들은 모두 `commands.txt`에서 지워지며, 수행되지 않은 명령어들만이 남아있게 된다.

### spg KILL

`spg KILL`은 현재 돌아가고 있는 작업을 중지할 때 사용되는 명령어이다. 어떤 작업들을 특정지을 것이냐에 따라 매우 많은 옵션들이 존재하며, 이는 다음과 같다.

- `-g`, `--groupList` 혹은 `-m`, `--machineList` 을 통해 특정 worker 그룹 혹은 worker 노드들의 작업을 중지할 수 있다.
- `-p` *pid list* `-m` *machine* 를 통해 단일 worker 노드의 작업을 중지할 수 있다. `spg job` 등을 통해 원하는 job(들)의 PID를 확인하고 그 job이 돌아가고 있는 한개의 machine을 특정지어야 한다.
- `-c` *command* 를 통해 특정 command가 들어간 작업들을 중지할 수 있다. 띄어쓰기를 인식하며, `spg job` 에서 나타나는 command 기준으로 job을 찾는다.
- `-t` *time* 을 통해 작업이 돌아간 시간이 설정한 시간 간격보다 짧을 경우 중지한다. `spg job` 에서 나타나는 Time을 기준으로 찾는다.
- `-s` *start* 를 통해 작업이 시작된 시점이 설정된 시점일 경우 중지한다. `spg job` 에서 나타나는 Start를 기준으로 찾는다.

만약 여러개의 옵션이 주어진다면, 해당 옵션들에 모두 해당하는 job들만을 찾아 종료하게 된다. 예를 들면 다음과 같다.

```
$ spg KILL -g kuda -t 90m 30s -c conda/envs/
```

위의 명령어는 kuda라는 이름의 머신 그룹에 속한 작업 들 중, 실행된지 90분 30초 이내이며 command에 `conda/envs/` 라는 문구가 들어가는 작업을 모두 중지하라는 뜻이다.

### Deprecated options

이외에도 이전 `spg` 버전과의 하위호환을 유지하기 위해 다음과 같은 옵션이 있지만 모두 상술된 옵션들로 통·폐합 되었다.

- `spg machine` : `spg list` 로 변경
- `spg all` : `spg job -a` 로 변경
- `spg me` : `spg job` 으로 변경
- `spg kill` : `spg KILL -m` *machine* `-p` *pidList* 로 통합
- `spg killall` : `spg KILL` 로 변경
- `spg killmachine` : `spg KILL -m` *machine* 으로 통합

- `spg killthis` : `spg KILL -c` *command* 로 통합
- `spg killbefore` : `spg KILL -t` *time* 으로 통합

## 3.3 Python

최근들어 다양한 패키지가 있다는 이점으로 python 언어를 많이 사용하고 있다. komplex 서버에는 기본적으로 python의 최신 버전 인터프리터가 설치되어 있으며 `/usr/bin/python` 의 경로로 존재한다. 하지만, 이 경로의 파이썬은 3.2절의 `spg` 등 서버에서 필요한 가장 기초적인 코드를 위해 사용되기에 일반 사용자는 사용하지 않아야 한다.

### 3.3.1 Virtual Environment

상술했듯 파이썬은 다양한 패키지들이 존재하며, 때때로 패키지들간의 호환성 문제로 부득이하게 현재 사용하고 있는 패키지의 이전, 혹은 이후 버전을 사용해야 할 때가 종종 있다. 예를들면 다음과 같다. A라는 프로젝트에 사용되는 a라는 패키지는 파이썬 버전 3.8까지만을 지원하지만, B라는 프로젝트에 사용되는 b라는 패키지는 파이썬 3.9 이상의 버전에만 호환된다. 이 경우, A와 B 프로젝트를 동시에 진행하기 어려운 상황이 있다.

이를 위해 파이썬에서는 가상환경이라는 기능을 지원하여 각 환경마다 설치되는 파이썬의 버전, 패키지의 종류 및 버전을 독립적으로 관리하도록 한다. 가상환경 기능은 다양한 방식으로 여러가지 방법이 있는데, 본 매뉴얼에서는 conda를 사용한 방법을 소개한다. 이미 다른 종류의 가상환경(venv, pyenv 등)을 사용하고 있다면, 굳이 conda를 사용하지 않아도 무방하다.

#### Conda

콘다, 혹은 conda는 파이썬의 가상환경 관리 프로그램이자 패키지 설치 프로그램이기도 하다. 가장 기본적인 패키지만 동봉하는 miniconda와 다양한 data science 관련된 패키지를 한번에 설치하는 anaconda가 있는데, komplex에서는 가벼운 miniconda를 선호한다. 설치 방법은 공식 documentation을 따라간다.[3]

conda의 설치는 자신의 홈 디렉토리가 있는 디스크 서버에서 진행해야 할 것이다. 터미널에서 진행하여도 결과는 동일하지만, 설치 작업이 많은 양의 디스크 IO를 요구하기 때문에, 상당히 오랜 시간이 걸려서 설치될 것이다. `ssh` *disk* 를 통해 자신의 디스크 서버로 넘어가자. 그리고 디스크 서버가 현재 인터넷 연결이 되어 있는지를 확인해야 한다.

```
[newbie@disk ~]$ ping www.google.com
```

만약 패킷이 보내지 않고 문제가 생긴다면, 관리자에게 문의하여 본인의 디스크 서버의 인터넷 연결을 허용해 달라고 이야기 하자. 그리고 conda와 가상환경의 설치가 모두 완료된다면 반드시 관리자에게 인터넷 사용을 다 했다고 이야기 해 주어야 한다.

[3] `https://conda.io/projects/conda/en/latest/user-guide/install/linux.html`

인터넷 연결을 완료하였으면 miniconda를 설치해 보자. 우선, 개인 pc에서 공식 홈페이지의[4] 최신 버전의 miniconda 설치 파일을 받아온다. Linux 버전의 64-bit 파일을 받으면 된다.

받은 파일의 이름은 `Miniconda3-`*version*`-Linux-x86_64.sh` 의 형식을 따를 것이다. 해당 파일을 2.2.1절에서 사용한 **scp**를 사용해 서버상의 홈 디렉토리로 옮겨준다. 그리고 다음과 같은 명령어를 통해 설치해 주면 된다.

```
[newbie@disk ~]$ bash Miniconda3-version-Linux-x86_64.sh
```

이후에는 나오는 내용대로 잘 따라가다가 마지막으로 `conda init` 을 하겠냐는 질문에 yes로 답해주면 모두 완료 될 것이다.

conda를 사용하는 방법은 인터넷에 잘 정리된 내용들이 많으니 하나씩 따라가며 익숙해지자. 설치된 이후 anaconda와 miniconda는 모두 같은 명령어를 따르기 때문에, 구분 없이 찾아보면 될 것이다. 가장 기본적인 패키지 `numpy`, `matplotlib` 등과 자신의 필요한 패키지들을 `conda install` 을 통해 설치해 보자.

모든 설치가 완료되면, 디스크 서버에서 로그아웃을 하고 관리자에게 반드시 말해야 한다. 비록 디스크 서버에서 conda를 설치했지만, 추가적인 작업 없이 서버에 존재하는 모든 노드들에서 사용할 수 있다.

### 3.3.2 Jupyter

파이썬을 사용하다보면, jupyter가 필요한 상황이 있을 것이다. 이는 interactive python을 사용할 수 있게 해주는 프로그램으로, 파이썬 뿐만 아니라 julia, R, scala등 다양한 언어를 지원하는 플랫폼이기도 하다. 먼저 디스크 서버에서 다음과 같이 conda로 jupyterlab을 설치해 주자.

```
[newbie@disk ~]$ conda install jupyterlab -c conda-forge
```

뒤에 붙은 `-c` 옵션은 패키지를 불러올 채널을 특정짓는 것으로, jupyterlab의 경우 conda-forge 채널을 사용하는 것을 권장한다. 이렇게 서버에 설치된 jupyter를 사용하는 방법은 크게 브라우저를 사용하는 방법과 2.2.4절에서 소개한 VSCode를 사용하는 방법이 있다. 두가지 모두 장단점이 있으므로, 모두 시도해 보고 편한 방식을 사용하도록 하자.

#### Browser

일반적으로 jupyter를 사용할때(jupyter notebook 혹은 jupyterlab 모든 경우) 생각하는 기본적인 방식이다. 개인 pc에서 jupyter 커널을 실행한다면, 브라우저에서 `http://localhost:8888` 로 접속하는 방식으로 실행한다. 이는 자신의 컴퓨터의 8888번 포트를 보여주는 것으로 이해할 수 있다.

우선 사용하는 서버의 가상환경에서 `$ jupyter notebook password` 를 통해 jupyter로 접근할 때 필요한 비밀번호를 설정해 주어야 한다. 그리고 jupyter 커널을 특정 포트로 실행한 다음 개인 pc의 포트로 해당 내용을 포워딩 하여 실행할 수 있다. 구체적인 방법은 다음과 같다.

[4] `https://docs.conda.io/en/latest/miniconda.html#linux-installers`

[*me*@*local* ~]$ ssh komplex -L *client-port*:localhost:*server-port*

2.2.2절에서 사용했던 komplex 호스트를 그대로 사용하였으며, 새로운 옵션인 `-L` 옵션을 주었다. 위의 명령어는 터미널의 *server-port*를 개인 pc의 *client-port*로 포워딩 해주는 명령어이다. 각 포트 번호는 1~65535 사이의 적당한 숫자를 정하면 되지만, 22번 포트와 같이 이미 역할이 정해져 있는 번호는 피하도록 하자. 많은 경우 *client-port*와 *server-port*의 순서를 헷갈려 하기 때문에, 자신이 기억하기 쉬운 번호를 하나 정해 동일한 포트 번호로 사용할 수도 있다.[5]

포트 포워딩을 동반한 ssh 접속이 완료되었다면, 이제 터미널에서 jupyter lab을 다음과 같이 실행시킬 수 있다.

[*newbie*@*terminal* ~]$ jupyter lab −−port=*server-port* −−no-browser

마지막으로 개인 pc의 브라우저에서 `http://localhost:`*client-port* 로 jupyter에 접속할 수 있을 것이다.

문서 작업과 같이 간단한 jupyter 작업을 할때는 괜찮겠지만, 머신 러닝 혹은 데이터 분석등을 위해 jupyter를 사용할 경우에는 터미널에서 jupyter 커널이 실행되면 안될 것이다. 우선 `spg free` 등을 통해 사용 가능한 노드를 확인하고 해당 노드에서 jupyter 커널을 실행해야 할 것이다. 이를 위해 이전에 해 주었던 포트 포워딩을 다시 한번 해주어야 한다. 예를 들어 GPU worker인 kuda1에서 작업을 하고자 한다고 하면 다음과 같다.

[*newbie*@*terminal* ~]$ ssh kuda1 -L *server-port*:localhost:*server-port2*

이를 통해 kuda1의 *server-port2*를 터미널의 *server-port*에 포워드 하였고, 이는 개인 pc의 *client-port*에 포워드 되었으므로, kuda1에서 jupyter lab을 실행시킨다면 개인 pc의 브라우저에서 접속 할 수 있다.

[*newbie*@*kuda1* ~]$ jupyter lab −−port=*server-port2* −−no-browser

이 방식은 이미 jupyter를 사용해 보았고, 필요한 플러그인이나 단축키등을 개인의 스타일에 맞춰 미리 설정해 놓았을 경우에 추천한다.

#### VS Code

2.2.4절에서 소개한 VSCode 역시 Jupyter라는 이름의 extension을 통해 jupyter를 지원한다. 단순하게 VSCode의 파일 탐색기에서 IPython의 확장자인 .ipynb 파일을 연다면 해당 extension이 활성화 되며 마치 브라우저에서 작동하는 jupyter와 같이 움직인다.

서버로의 접속은 이전 Browser를 이용하는 방법보다 훨씬 간단하다. 자신이 접속하고자 하는 노드를 지정하고, 2.2.2절의 disk 호스트와 마찬가지로 ProxyJump를 사용한 config 파일을 작성하자.

[5]하나는 개인 pc의 localhost의 포트 번호이고, 다른것은 터미널의 localhost의 포트 번호이기 때문에, 같은 번호를 사용하더라도 무방하다.

그리고 2.2.4에서 소개한 Remote-SSH를 사용해 해당 노드로 들어가면 된다. 들어간 이후 .ipynb 파일을 찾아 열면, 그대로 현재 있는 계산 노드에서 jupyter 커널이 실행된다.

이 방식은 ssh config 파일에만 익숙하다면 훨씬 편한 방법이 될 것이다. Browser를 이용한 방식처럼 jupyter의 다양한 plugin들을 사용하지 못하지만 대신 그에 상응하거나 더 많은 기능을 포함하고 있는 VSCode의 extension들을 그대로 사용할 수 있다는 장점이 있다. 또한 VSCode에서 설정된 단축키들 역시 그대로 사용할 수 있다는 점이 가장 편리하여 현재 필자는 이 방법을 사용하고 있다.

다만 2021년 9월 현재, 아직까지 VSCode에서 jupyter를 지원하는 부분은 지속적으로 개발 중에 있다. 최근 몇달간 많은 발전이 있어 이제 추천할 수 있을 정도로 안정화가 되었지만, 아직까지 자잘한 버그들이 있을 수 있다. 이는 어느정도 개발과정이 정상궤도에 오르면 해결 될 수 있을 것이라 기대한다.

# Chapter 4

# Appendix

마지막 장에서는, 사용자가 굳이 알지 않아도 괜찮은 내용들을 담고 있다. 다만 알고 있다면, 조금은 더 편한 개발 환경을 구축하고 서버를 사용할 수 있는 내용들을 담았으니, 가볍게 읽어보자. 무엇보다 이번 장은 필자의 개인적인 선호가 담긴 내용들이 많기 때문에, 무작정 따라하는 것 보다 관련 정보를 찾아보며 다른 옵션으로는 어떤 것이 있나 등을 알아보자.

## 4.1 Tips for Windows

필자는 지금까지 개인 pc의 운영체제로 windows를 사용해 왔으며, 맥에 대한 경험은 거의 전무하다. 따라서 이번 절은 개발 환경을 세팅할 때 윈도우에서 어떻게 해야 편한지를 담고 있으며, 맥을 사용하는 사용자들에게는 거의 해당되는 내용이 없을 것이다. 하지만 이전 매뉴얼의 내용을 읽으면서 느꼈겠지만 맥의 경우 유닉스를 기반으로 하기 때문에 따로 개발 환경이라는 것을 구축하지 않아도, 대부분 문제 없이 잘 사용할 수 있기도 하다.

### 4.1.1 WSL

Windows Subsystem Linux, 줄여서 WSL은 윈도우에서 지원하는 리눅스 가상 머신 기능이다. 2015년 말에 처음 발표되어, 2019년 중순에서 WSL2로 업데이트 되며 상당히 많은 개선이 이루어졌다. 통상적인 가상 머신과 다르게 실제 리눅스 커널을 탑재하여 작동하며, 상당히 가볍고 빠른 리눅스 가상 머신이라고 이해하면 충분하다.

윈도우는 다양한 환경 변수의 설정, 지원하지 않는 프로그램 패키지, 매우 불친절한 에러 메세지 등으로 개발을 진행하기 까다로운 운영체제이다. 하지만 WSL의 기능이 도입됨으로 인해 통상적인 문서 작업이나 웹 브라우징 등은 윈도우에서 하며 개발 및 코딩 관련 작업들은 모두 WSL의 리눅스에서 진행할 수 있는 매우 이상적인 환경이 구축되었다.

2021년 9월 현재까지 WSL은 CLI 환경이 기본이며, 베타 버전의 GUI를 지원하고 있다. 만약

본인이 GUI를 써보고 싶다 할 경우, 공식 깃허브[1]의 가이드를 따라가자. 그리고 CUDA 가속등을 WSL에서 사용하고 싶을 경우 윈도우 참가자 프로그램 신청 후 가이드[2]를 따라 설치를 할수도 있다.

**Installing**

WSL의 설치는 마이크로소프트의 공식 문서에[3] 매우 잘 정리되어 있다. 어떤 리눅스 배포판을 설치할까 고민될 경우에는 가장 대중적인 Ubuntu를 설치하자. komplex 서버의 운영체제인 Arch 리눅스의 경우, 공식적인 지원은 아직 없으며 깃허브[4]의 문서를 따라가야 한다.

WSL에서 사용될 계정 이름과 암호까지 만들고 나면, 설치를 완료한 것이다. 계정의 이름은 komplex에서와 같이 가능하다면 대문자 없이 소문자로만 정하도록 하자. 나머지는 대부분 일반적인 리눅스를 사용하는 것과 동일하다.

**File Explorer**

WSL을 사용하면서 가장 먼저 알아야 하는 것은 윈도우와 WSL에서 각자 서로의 파일을 확인하는 방식이다.

- Windows에서 WSL의 파일 확인하기
  윈도우 파일 탐색기의 주소창에 `\\wsl$`를 입력하자. 자신이 설치한 리눅스 배포판이 보일 것이고, 그곳에 들어가 보면 리눅스의 가장 하위 경로 `/`인 것을 확인할 수 있다. WSL에서 만든 계정의 홈 디렉토리는 `/home/`*wsl-username*의 경로에 있다. 매번 이렇게 들어가기 귀찮을 수 있기 때문에, 해당 홈 폴더를 즐겨찾기에 고정 해 놓으면 다음부터 편하게 들어갈 수 있을 것이다.

- WSL에서 Windows 파일 확인하기
  WSL의 기준으로, 윈도우의 파일 시스템은 `/mnt` 디렉토리에 있다. 윈도우의 C 드라이브는 `/mnt/c`, D 드라이브는 `/mnt/d`에 있는 식이다. WSL에서 윈도우의 디렉토리를 돌아다녀 보면 윈도우 시스템 디렉토리등에 permission deny 에러가 뜨는 경우가 많을 텐데, 실제로 접근이 필요한 디렉토리 및 파일들은 확인 가능하므로 크게 신경쓰진 않아도 된다. 윈도우 홈 디렉토리는 `/mnt/c/Users/`*windows-username*에 있다.

**VSCode**

또 다시 VSCode 관련 설정이다. 많은 부분은 4.3절에서 다룰 것이고, 지금은 WSL에 관련된 내용만 확인하자. 아마 WSL을 설치하고 VSCode를 실행한다면, 자동으로 WSL에 대한 extension을 추천해 줄 것이다. Remote-WSL이라는 extension으로, 2.2.4절에서 설치한 Remote-SSH와 같은 방식으로 작동한다. 이미 Remote-SSH를 사용해 보았다면 문제 없이 사용할 수 있을 것이다.

[1] `https://github.com/microsoft/wslg`
[2] `https://docs.microsoft.com/ko-kr/windows/ai/directml/gpu-accelerated-training`
[3] `https://docs.microsoft.com/ko-kr/windows/wsl/install-win10`
[4] `https://github.com/yuk7/ArchWSL`

**sshfs**

WSL에서도 **sshfs**를 사용하여 서버 시스템을 마운트 할 수 있다. 2.2.3절을 참고하여 따라가 보자. 선호에 따라서 VSCode의 Remote-WSL으로 마운트 된 서버 파일 시스템을 사용하는 것이 편할 수 있다.

### 4.1.2 Windows Terminal

WSL 설치 가이드를 보면, 그 아래쪽에 optional하게 Windows Terminal의 설치에 대한 사항을 확인할 수 있을 것이다. Windows Terminal은 윈도우의 기본 쉘인 cmd, powershell 뿐만 아니라 WSL에서 작동하는 쉘등 대부분의 것들을 한번에 사용할 수 있다. 쉘과 터미널에 대한 내용은 4.4절을 참고하자.

무엇보다, 터미널을 사용하다 보면 창을 분할해서 사용하거나 탭을 띄워 하고 싶은 경우가 있는데, 기본 powershell 혹은 wsl의 창은 이러한 기능을 지원하지 않는다.

## 4.2 Font

이번 절에서는 폰트(font)에 대한 소개를 해보고자 한다. 완전히 개인 선호에 따른 영역이라고 생각할 수도 있겠지만, 의외로 코딩을 할 때 상당히 도움되는 내용일 수 있을 것이다.

### Monospaced Font

일반적인 영문 문서 작업을 할때와 코딩을 할 때 내용의 가장 큰 차이점은 특수문자의 여부이다. 코드의 경우 일반적인 문서보다 특수문자의 사용이 월등히 많은 것을 알고 있을 것이다. 기본적인 bracket과 사칙연산 기호에서 시작하여 period, comma, quotation등 키보드에서 바로 입력할 수 있는 대부분의 특수문자를 사용한다.

일반적인 폰트를 사용한다면, 각 영문자간, 그리고 특수 문자의 폭이 서로 달라 코드가 잘 정렬되지 않을 수 있다. 이런 문제를 해결 하기 위해 등장한 폰트가 고정 폭 글꼴, monospaced font이다. 이렇게 분류된 글꼴들로는 모든 영문자 뿐만 아니라 숫자, 특수문자들 하나하나의 폭이 동일하기 때문에, 코드가 매우 깔끔하게 정렬되는 것을 볼 수 있을 것이다.

가능하다면 최소한 개인이 자주 사용하는 터미널과 텍스트 에디터에는 monospaced font를 설정해 보도록 하자.

### Liagature Font

상술된 monospaced font에 더 나아가, 코딩에 자주 사용되는 몇몇 문구들을 합자(ligature)하여 만든 폰트들이 존재한다. 예를 들자면, 작거나 같다를 뜻하는 문구 `<=`의 경우 수학기호인 ≤로 바꾸어 준다. 특히 위의 예시를 포함한 비교연산자(`==`, `!=`)들이나 화살표(`->`) 등에서 매우 유용하게 사용된다.

### PL Font

이제부터는 조금 더 많은 기능을 지원하는 폰트를 소개하고자 한다. Powerline font의 경우, Powerline symbol이라 불리는 삼각형등의 형태를 포함한 폰트로, 4.4절에서 소개할 zsh의 기본 테마들은 Powerline font가 아닌 경우 깨짐 현상이 일어날 가능성이 매우 크다. 만약 **vim**등을 자신의 기본 에디터로 사용하고 있다면 statusline 등을 커스터마이징 하는데 필수적으로 사용되는 폰트이기도 하다.

### Nerd Font

가장 마지막으로, 이름부터 심상치 않은 nerd font이다. 이 폰트는 이전에 소개한 Powerline symbol를 포함하여 3000개가 넘는 다양한 종류의 아이콘들을 포함한 폰트군이다. `https://www.nerdfonts.com/`에 접속하여 구경해보자.

### In the end..

결론적으로, 위에서 소개한 monospace 특성을 가지고 있으며 ligature 문자를 구현하였으며 nerd symbol들을 포함하는 폰트를 사용하면 된다. 많이 사용되는 두개의 폰트는 다음과 같다.

- Caskaydia Cove Nerd Font: 마이크로소프트에서 발표한 Cascadia Code에 nerd symbol들을 추가한 버전이다. 저작권상의 문제로 본래의 이름에서 조금 변형을 하였다.
- FiraCode Nerd Font: 유명한 프로그래밍 폰트인 FiraCode에 nerd symbol이 추가된 버전이다.

이외에도 많은 종류의 폰트들이 있으니, 만약 두가지 다 마음에 들지 않는다면 위의 nerdfont 페이지의 다운로드 링크로 들어가자. 다운로드 가능한 대부분의 폰트들을 직접 쳐 볼수도 있으니 개인 선호에 맞춰 선택하자.

## 4.3 VSCode

본 매뉴얼에서 (아마도) 가장 많이 등장한 VSCode에 대한 추가적인 내용을 소개하는 절이다. 필자는 연구 생활에 있어서의 대부분의 작업을 VSCode로 할 정도로, 상당히 많은 기능을 지원하기에 더욱 자주 등장하기도 하는 것 같다. 해당 프로그램 이외에도 텍스트 에디터로 유명한 프로그램은 Notepad++, Sublime Text, Atom등이 있으며 C/C++의 IDE로는 Visual Studio,CLion 등이 대표적이며 Python의 IDE로는 pycharm, spyder등 정말 다양한 종류의 대체 프로그램들이 있다.

하지만 다양한 종류의 프로그램을 사용해 본 결과, 결국 VSCode로 정착하였다. 필자가 사용하면서 추천할 만한 설정들을 소개해 보도록 하겠다.

### Extension

VSCode의 가장 큰 장점은 다양한 종류의 확장 프로그램(extension)들이 있다는 것이다. 많은 수의 개발자들이 extension을 개발하고 있으며, VSCode의 개발사인 마이크로소프트 역시 많은 종류의 extension들을 지원하고 있다. 추천할 수 있는 목록은 다음과 같다.

- C/C++
  마이크로소프트에서 공식적으로 지원해주는 C언어와 C++ 언어에 대한 extension. 기본적인 syntex highlighting을 비롯하여 auto completion등을 지원해 준다.

- Python, Pylance, Jupyter
  마이크로소프트에서 공식적으로 지원해주는 python과 jupyter에 대한 extension. Pylance의 경우는 다양한 종류의 python 관련 편의 기능을 제공해 주므로 같이 설치해 주는 것을 추천한다.

- Latex Workshop
  논문을 작성할때 사용되는 latex를 지원해 준다. Latex를 컴파일 하기 위한 Tex Live등은 기본적으로 설치해져 있어야 하지만, 이외의 모든 기능을 지원해 준다.

- Markdown Preview Enhanced
  Git의 README를 작성할 때나, 간단한 문서 작성을 할 때 사용되는 markdown 언어를 preview 해줄 수 있는 extension이다. 기본적으로 VSCode에서도 markdown의 preview 기능을 지원해 주지만, pdf로 export등을 할 수 없기 때문에, 추천할 만 하다.

- Bracket Pair Colorizer2[5]
  코딩을 하다 보면 많은 수의 괄호(bracket)들을 중첩해서 사용하게 된다. 이 extension은 서로 다른 괄호를 색을 통해 구분해 준다.

- Better Comments
  코딩에 주석(comment)을 다는 것은 필수적인 일이다. 하지만 때때로 comment의 종류를 구분하고 싶을 때가 있으며 무엇보다 임시로 실제 코드를 comment out 해 두어 실행하지 않는 경우와 진짜 주석을 구분하는 것은 필요하다. 이 extension은 comment symbol(일반적으로 # 혹은 //)뒤에 특수문자(* ? @ 등)을 추가해 해당 comment의 색을 바꾸어 종류에 따라 구분할 수 있게 해준다.

- Path Intellisense
  코드에서 경로에 관련된 내용을 인식하여 자동완성을 지원해준다.

- Trailing Spaces
  코드에서 의미 없는 white space를 인식하여 표시해 주며, 필요하다면 지워준다.

[5]VSCode의 2021년 8월 업데이트로, native 설정으로 병합되었다.

- CodeSnap
  자신의 코드를 캡쳐하여 다른 곳에 올려야 할 때, 현재 사용하는 테마와 syntex highlighting에 맞게 캡쳐해 준다. 다만 상술된 Better Comments의 색이 다른 주석을 인식하지 못한다.

소개한 extension 이외에도 생각할 수 있는 거의 모든 종류의 프로그래밍 언어에 대한 지원 extension도 있다. 만약 docker 환경을 사용한다면 Remote-Container을 사용할 수도 있다.

### Shortcut

VSCode의 또 다른 장점중으로 매우 다양한 키보드 단축키가 있다. 모든 종류의 단축키를 커스터마이징 할 수 있으며, 만약 vim, emacs등에서 사용하던 단축키를 사용하고 싶다면 역시 extension으로 지원된다. 사용할때 알아두면 정말 좋은 단축키들 몇개를 소개한다. 모두 윈도우 기준이며, 대부분의 경우 맥은 Ctrl 대신 Cmd를 사용하면 된다.

- `Ctrl+/`
  이것만큼은 반드시 알고 있자. 현재 커서가 있는 줄의 comment 여부를 toggle한다.

- `Ctrl+c/x/v` without block
  텍스트를 복사,잘라내기 및 붙여넣기를 하는 단축키는 매우 익숙할 것이다. 하지만 일반적인 경우에서는 block으로 선택된 영역에 한하여 작동한다. VSCode에서는 block 선택을 하지 않고 위의 단축키를 사용한다면, 해당 줄 전체가 영향을 받게 된다. 줄 단위로 이동을 할 때 매우 유용하게 사용할 수 있을 것이다. 특히 잘라내기를 반복하여 필요없는 코드 라인을 지우기만 할 수도 있다.

- `Alt+up/down`
  줄 단위를 잘라내고 붙여넣는 대신에, 그대로 한 줄 위/아래로 움직이는 단축키이다. 생각외로 정말 많이 사용된다.

- `Ctrl+Alt+up/down`
  현재 커서가 위치해 있는 곳 위/아래로 멀티 커서를 만들어준다. 같은 열에 위치해 있는 내용을 삭제하거나 추가할때 매우 유용하게 사용된다. 특히 이 경우에 4.2절에서 소개한 monospaced font가 큰 도움이 될 것이다. 이외에도 `Alt`를 누르고 멀티 커서가 필요한 곳에 마우스 클릭을 하면, 멀티 커서를 쓸 수 있다.

- `Ctrl+f`
  꼭 VSCode가 아니더라도 대부분의 프로그램에서 특정 텍스트를 찾는 명령어로 익숙할 것이다. VSCode에서는 추가적으로 대/소문자 구별 여부, 전체 단어의 일치 여부, 정규표현식 사용 여부등을 설정할 수 있다.

- `Ctrl+Shift+p`
  VSCode의 명령 팔레트를 연다. 열려진 팔레트에서 VSCode에 존재하는 모든 종류의 명령어를 사용할 수 있다.

### Git

Git은 코드의 버전을 관리하거나 공유할 때 많이 사용된다.[6] 일반적으로 터미널에서 명령어를 통해 git과 상호작용 하지만, 마이크로소프트가 깃허브를 인수하면서 VSCode에 git과 관련된 기능들이 급격히 많이 들어오기 시작했다. 그러면서 명령어를 전혀 사용하지 않고도 git을 initialize하고 commit을 관리하며, branch를 생성하는 등 대부분의 일들을 할 수 있게 되었다.

또한 이미 github 계정으로 VSCode에 로그인한 상태라면, 본인의 github repository에 쉽게 publish 할수도 있다. 아직 git을 사용해 보지 않았을 경우 진입장벽이 많이 낮기 때문에, 한번쯤 공부해 보는것도 좋을것 같다.

## 4.4 ZSH

komplex 서버, 혹은 리눅스를 사용한다면 자연스럽게 터미널창 혹은 쉘(shell)을 사용하는 시간이 많아질 것이다. 지금까지 본 매뉴얼에서는 이를 구분하지 않아 서술하였지만, 실제로 터미널과 쉘은 명확히 구분되는 개념이다.

터미널의 경우, 쉘의 입출력을 화면에 표시해 주는 프로그램이다. 윈도우의 경우 4.1.2절에서 다룬 windows terminal등이 대표적이며, 맥에서는 terminal, iTerm2 리눅스에서의 xterm, gnome-terminal 등 매우 다양한 종류가 있다. 각 터미널들의 종류에 따라 탭의 사용 여부, 분할창의 사용 여부, 폰트 등을 설정할 수도 있을 것이다.

반면 쉘의 경우, 사용자가 입력한 텍스트 기반 명령어를 이해하고, 실제로 작업을 수행하는 프로그램이며, 그 종류로는 대표적인 bash(Bourne Again Shell)에서 시작하여 zsh(Z Shell), Fish(Fish shell)등 다양하게 존재하며, 대부분 sh(Bourne Shell)의 문법을 따른다. 하지만 그 종류가 나뉜 만큼 호환되지 않는 명령어들 역시 존재한다.

대체로 리눅스에서는 bash 쉘을 기본으로 사용하며, komplex 서버 역시 사용자의 기본 쉘[7]을 bash로 정의한다. 현재 자신이 사용하고 있는 쉘은 `$ echo $SHELL`을 통해 확인 할 수 있다. 기본이 되는 쉘인 만큼 일반적으로 쉘 스크립트라고 불리는 코드들은 대부분 bash에서 작동되는 배쉬 스크립트인 경우가 많다. 연구실에서 사용되는 모든 스크립트들 역시 bash상에서 수행되며, bash만을 사용해도 전혀 무방하다.

하지만 zsh라는 쉘은 조금 더 커스터마이징이 많이 가능하여 만약 조금 더 편한 터미널 창을 사용하고 싶다면 아래에서 소개한 방법을 한번쯤 따라가보아도 좋을 것이다. 만약 bash보다 마음에 든다면, 서버 관리자에게 요청하여 zsh를 komplex의 기본 쉘로 설정해보자.

[6] 평범한 연구실 생활에서는 거의 사용할 일이 없을 것이다. 하지만 만약 이후 코딩과 관련된 일을 하고 싶다면, 알아야 할 필수 지식이 될 것이다.

[7] 공식 명칭으로는 login shell이라 한다

## Theme

zsh가 커스터마이징에 장점을 보이는 가장 큰 이유는 Oh My Zsh 라는 프로그램 덕분이다. 아무런 커스터마이징도 하지 않은 zsh는 기본 bash보다 훨씬 열악한데, 해당 프로그램을 통해 매우 간단하게 보기 좋은 터미널 창을 만들 수 있다. 해당 프로그램을 설치하기 위해 우선 공식 깃허브[8]에서 installation guide를 따라가자.

zsh의 기본 설정은 `~/.zshrc`에 저장되어 있으며, ohmyzsh를 설치하면 자동으로 해당 파일을 수정해 줄 것이다. 그리고 공식 깃허브에 올라와 있는 것 처럼 ZSH_THEME의 내용을 설치된 테마 중 하나로 지정하고, `$ source ~/.zshrc`를 입력한다면, 바로 해당 테마로 터미널이 바뀔 것이다. 이때, 4.2절에서 언급한 power line 계열의 폰트를 터미널의 폰트를 설정해 두어야 글자들이 깨지지 않을 것이다.

### Powerlevel10k

ohmyzsh와 함께 설치되는 기본 테마들도 충분히 멋지지만, 서드파티 테마를 사용하여 조금 더 터미널을 신경써서 커스터마이징 할 수도 있다. 다양한 종류의 테마 중 대표적으로 powerlevel10k 라는 테마를 소개하고자 한다.

역시나 공식 깃허브[9]에서 installation guide를 따라가자. 해당 테마의 경우 4.2절에서의 nerd font를 설치해야 정상적으로 작동한다. 그리고 `~/.zshrc` 파일을 다음과 같이 수정하자.

**~/.zshrc**

```
...
ZSH_THEME="powerlevel10k/powerlevel10k"
...
```

그리고 상술한 바와 같이 `$ source ~/.zshrc`를 통해 powerlevel10k 테마를 적용할 수 있다. 처음으로 적용한다면, 우선 해당 테마의 설정을 진행해야 한다. 모두 자신의 취향에 맞게 설정해 보자. 조금 더 자세한 설정은 `~/.p10k.zsh`에서 할 수 있다.

## Plugins

ohmyzsh의 또 다른 강력한 장점은 다양한 plugin 기능이다. 기본적으로 git에 대한 plugin이 설정되어 있으며, 매우 다양한 종류가 기본으로 설치되기 때문에, 관심있다면 하나씩 찾아보자. 다만, 너무 많은 plugin을 동시에 실행하면 zsh을 실행할 때 느려질 수 있다고 경고한다.

### zsh-syntex-highlighting

테마와 마찬가지로 서드파티 플러그인들도 존재하는데, 이 중 `zsh-syntax-highlighting`이라는 플러그인을 추천한다. 자신이 입력한 명령어가 현재 쉘에서 수행할 수 있는지를 쉽게 알려준다.

[8] `https://github.com/ohmyzsh/ohmyzsh`
[9] `https://github.com/romkatv/powerlevel10k`

설치 방법은 공식 깃허브[10]을 따라가자.

설치를 완료했다면 테마의 경우와 마찬가지로 `~/.zshrc`을 수정하여 적용할 수 있다.

**~/.zshrc**

```
...
plugins=(git zsh-syntax-highlighting)
...
```

[10] `https://github.com/zsh-users/zsh-syntax-highlighting`