Linux

Muchang Bahng

January 2024

Contents

The following set of notes describes the everyday use of a Linux operating system. I refer to it for mainly my personal desktop, but it is also useful for working in computing clusters. Some of the commands are specific to the Arch Linux distribution (since that is what I work with), but I occasionally include those from Ubuntu and Red Hat, since I run into these distributions often in servers.

I try to organize this in a way so that one who wishes to get started in Linux can go through these notes chronologically. For now, we will assume that you have a Linux distribution installed. There are many resources beyond this book that helps you do that.

1 Introduction

1.1 Vim and Neovim

Vim is guaranteed to be on every Linux system, so there is no need to install it. However, you may have to install Neovim (which is just a command away). Vim can be a really big pain in the ass to learn, but I got into it when I was watching some video streams from a senior software engineer at Netflix called The Primeagen. He moved around the code like I've never seen, and I was pretty much at the limit of my typing speed, so I decided to give it a try during the 2023 fall semester. My productivity plummetted during the first 2 days (which was quite scary given that I had homework due), but within a few weeks I was faster than before, so if you have the patience, I would recommend learning it. Here is a summary of reasons why I would recommend learning Vim:

1. It pushes you to know the ins and outs of your editor. As a mechanic with his tools, a programmer should know exactly how to configure their editor.
2. The plugin ecosystem is much more diverse than other editors such as VSCode. You can find plugins/extensions for everything. Here is a summmary of them here.
3. You're faster. If you're going to be coding for the next 5 years, then why no t spend a month to master something that will make you faster? You'll increase total productivity.
4. Computing clusters and servers will be much easier to navigate since they all run Linux with Vim.
5. Vim is lightweight, and you don't have to open up VSCode every time you want to edit a configuration file.

1.1.1 Vim vs Neovim

Experience wise, Vim and Neovim are very similar, and if you configure things rihght, you may not even be able to tell the difference. But there are 3 differences that I want to mention:

1. Neovim can be configured in Lua, which is much cleaner than Vimscript.
2. Neovim provides mouse control right out of the box, which is convenient for me at times and can be easier to transition into, while Vim does not provide any mouse support.
3. There are some plugins that are provided in Neovim that are not in Vim.

Either way, the configuration is essentially the same. At startup, the text editor will parse some predetermined configuration file and load those settings.

1.1.2 Vim Configuration File

In Vim, your configuration files are located in `/.vimrc` and plugins are located in `/.vim/`. In here, you can put in whatever options, keymaps, and plugins you want. All the configuration is written in VimScript.

```
# options
filetype plugin indent on
syntax on
set background=dark
set expandtab ts=2 sw=2 ai
set nu
set linebreak
set relativenumber

# keymaps
inoremap <C-j> <esc>dvbi
inoremap jk <esc>
nnoremap <C-h> ge
nnoremap <C-l> w
```

1.1.3 Neovim Configuration File

In Neovim, I organize it using Lua. It essentially looks for the `/.config/nvim/init.lua` file and loads the options from there. We also have the option to import other Lua modules for better file structure with the `require` keyword. The tree structure of this configuration file should be the following below. The extra `user` director layer is necessary for isolating configuration files on multiple user environments.

```
... ftplugin
.   ... cpp.lua
.   ... html.lua
... init.lua
... LICENSE
... lua
.   ... user
.       ... options.lua
.       ... keymaps.lua
.       ... plugins.lua
.       ... telescope.lua
.       ... toggleterm.lua
... plugin
    ... packer_compiled.lua
```

The init file is the "main file" which is parsed first. I generally don't put any explicit options in this file and reserve it only for require statements. It points to the following (group of) files:

1. **options.lua**: This is where I store all my options.
2. **keymaps.lua**: All keymaps.
3. **plugins.lua**: First contains a script to automatically install packer if it is not there, and then contains a list of plugins to download.
4. **Plugin Files**: Individual configuration files for each plugin (e.g. if I install a colorscheme plugin, I should choose which specific colorscheme I want from that plugin).
5. **Filetype Configuration Files**: Options/keymaps/plugins to load for a specific filetype. This helps increase convenience and speed since I won't need plugins like VimTex if I am working in JavaScript.

Once you have your basic options and keymaps done, you'll be spending most of your time experimenting with plugins. It is worth to mention some good ones that I use.

1. **Packer** as the essential package manager.

2. **Plenary**
3. **Telescope** for quick search and retrieval of files.
4. **Indent-blankline** for folding.
5. **Neoformat** for automatic indent format.
6. **Autopairs** and **autotag** to automatically close quotation marks and parantheses.
7. **Undotree** to generate and navigate undo history.
8. **Vimtex** for compilation of LaTeX documents.
9. **Onedark** and **Oceanic Next** for color schemes.
10. **Vim-Startify** for nice looking neovim startup.
11. **Comment** for commenting visual blocks of code.

It is also worthwhile to see how they are actually loaded in the backend. Each plugin is simply a github repo that has been cloned into `/.local/share/nvim/site/pack/packer/`, which contains two directories. The packages in `start/` are loaded up every time Neovim starts, and those in `opt/` are packages that are loaded up when a command is called in a certain file (known as lazy loading). Therefore, if you have any problems with Neovim, you should probably look into these folders (and possibly delete them and reinstall them using Packer if needed).

1.1.4 Troubleshooting

A good test to run is `:checkhealth`, which checks for any errors or warnings in your Neovim configuration. You should aim to have every (non-optional) warning cleared, which usually involves having to install some package, making it executable and/or adding to `$PATH`.

If you are getting plugin errors, you can also manually delete the plugin directory in 'pack/packer' and run 'PackerInstall' to re-pull the repos. This may help.

1.2 Distributions and Package Managers

Linux comes in many flavors of distributions. Most beginners look at screenshots of these distributions on the internet and judge them based on their aesthetics (e.g. I like how Kali Linux looks so I'll go with that one). A common feature of all Linux distributions is that they provide the user the power to customize their system however they want, so you can essentially make every linuxdistribution look like any other. So what are some things you should consider when choosing a distribution?

1. First is the popularity and how well it is supported. This includes the number of people who use the distribution (e.g. the Ubuntu StackExchange is a very large community) and how good the documentation is overall (e.g. the ArchLinux wiki is very well documented).
2. Each linux distribution essentially consists of a kernel and package manager. The architecture, design, and the update scheme of the kernel may be an interest to many linux users.
3. Every distribution has its own native package manager, and the availability of certain necessary packages, the ease of installation, and the updating schemes is also something to consider.
4. The ideals of the respective communities. The community behind each distribution has a certain set of ideals that they lean more towards. For example, the Ubuntu community likes having programs that are right out of the box, with good GUI support and is more beginner-friendly while Arch has more of a minimal and extremely customizable nature to it with its software being much more CLI dependent.

Let's begin with the package managers. Every application on your system (Firefox, Spotify, pdf readers, VSCode, etc.) is a package, and manually downloading and managing each one is impossible to do. Therefore, each distribution has its own native package manager that automatically takes care of downloading, installing, removing, checking dependency requirements of each package. In order to download a package, a package manager should also know where it is downloading *from*. Essentially, a package manager itself can be downloaded with other package managers, so package managers are packages as well.

1. **apt** : The advanced packaging tool is the native manager for Ubuntu distributions.
2. **pacman** : Native package manager for Arch Linux.
3. **yay** : The package manager for software in the **Arch User Repository**.
4. **snap** :
5. **flatpak** :
6. **dpkg** : Package manager for Debian based distributions.

Chances are if you are using one distribution, you would only have to work with a small subset of these package managers. Each package manager has one or more files in the computer that specify a list of **repositories**.

1.2.1 Pacman

For example, the configuration file for pacman is located at `/etc/pacman.conf`. In the options section, I can configure stuff like text color, enabling/disabling parallel downloads, choosing specific packages to ignore upgrading, etc. Then, we can specify the servers that we should download from. In the text below, the server variable defines which server we should look at first, and then the Include variable stores the location of the file `mirrorlist` that defines a list of other servers that we should download from.

The mirrorlist file stores a list of URLs. Each URL is a **mirror**, which is a server that contains a physical replica of all the packages that are available to you via `pacman` (hence the name mirror). You can literally type in the links provided in Figure 2 (replacing `$repo` with `core` and `$arch` with `x86_64`). It contains a tarball of each package ready to be downloaded. Some repos might contain more packages than others, some might have packages that only they supply that others don't, but if you can install the piece of software via your package manager then one of your configured repos is declaring they have it available and therefore should have the file on hand to give to you if asked for it. A list of all available mirrors are available here (this only uses HTTPS, but HTTP mirrors are also available).

The mirrors that you download from should be trustworthy and fast. The speed is mainly related to how close you are to that mirror geographically, so if you are moving to another country you should probably update this mirrorlist for faster download speeds. There is a default mirrorlist file that is generated, but you can download and use the **reflector** package to update it.

Here are some common commands:

1. Install a package: `sudo pacman -S pkg1` (`-s` stands for synchronize)
2. Remove a package: `sudo pacman -R pkg`
   - remove dependencies also: `-s` (recursive)
   - also remove configuration files: `-n` (no save)
   - also removes children packages: `-c` (cascade)
3. Update all packages: `sudo pacman -Syu`
   - synchronize: `-S`
   - refresh package databases: `-y` (completely refresh: `-yy`)
   - system upgrade: `-u`

```
# The following paths are commented out with their default values listed.
# If you wish to use different paths, uncomment and update the paths.
#RootDir     = /
#DBPath      = /var/lib/pacman/
#CacheDir    = /var/cache/pacman/pkg/
#LogFile     = /var/log/pacman.log
#GPGDir      = /etc/pacman.d/gnupg/
#HookDir     = /etc/pacman.d/hooks/
HoldPkg     = pacman glibc
#XferCommand = /usr/bin/curl -L -C - -f -o %o %u
#XferCommand = /usr/bin/wget --passive-ftp -c -O %o %u
#CleanMethod = KeepInstalled
Architecture = auto

# Pacman won't upgrade packages listed in IgnorePkg and members of IgnoreGroup
#IgnorePkg   =
#IgnoreGroup =

#NoUpgrade   =
#NoExtract   =

# Misc options
#UseSyslog
#Color
#NoProgressBar
CheckSpace
#VerbosePkgLists
ParallelDownloads = 5
ILoveCandy
```

Figure 1: Subset of contents of the `/etc/pacman.conf` file

```
Server = https://archlinux.mailtunnel.eu/$repo/os/$arch
Server = https://mirror.cyberbits.eu/archlinux/$repo/os/$arch
Server = https://mirror.theo546.fr/archlinux/$repo/os/$arch
Server = https://mirror.sunred.org/archlinux/$repo/os/$arch
Server = https://mirror.f4st.host/archlinux/$repo/os/$arch
Server = https://md.mirrors.hacktegic.com/archlinux/$repo/os/$arch
Server = https://mirrors.neusoft.edu.cn/archlinux/$repo/os/$arch
Server = https://mirror.moson.org/arch/$repo/os/$arch
Server = https://archlinux.thaller.ws/$repo/os/$arch
```

Figure 2: Contents of the `/etc/pacman.d/mirrorlist` file

4. List installed packages: `pacman -Q`
    - List detailed info about a package: `pacman -Qi pkg`
    - List all files provided by a package: `pacman -Ql pkg`
    - List all orphaned packages: `pacman -Qdt`
    - List all packages that have updates available: `pacman -Qu`
    - List all explicitly installed packages: `pacman -Qet`
    - Display the dependency tree of a package: `pactree pkg` (from the `pacman-contrib` package)
    - List last 20 installed packages:

```
expac --timefmt='%Y-%m-%d %T' '%l\t%n' | sort | tail -n 20
```

5. To check size of current packages and dependencies, download `expac` and run `expac -H M '%m t%n' | sort -h`
6. The package cache stored in `/var/cache/pacman/pkg/` keeps old or uninstalled versions of packages automatically. This is helpful since it also keeps older versions of packages in the cache, and you can manually downgrade in case some packages break.
    - We can delete all cached versions of installed and uninstalled packages, except for the most recent 3, by running `paccache -r` (provided by the `pacman-contrib` package).
    - To remove all cached packages not currently installed, run `pacman -Sc`
    - To remove all cached aggressively, run `pacman -Scc`
    - To downgrade, you go into the package cache directory and say you want to see which versions of neovim you have installed. You can `ls` the directory to see the following.

```
neovim-0.9.5-1-x86_64.pkg.tar.zst
neovim-0.9.5-1-x86_64.pkg.tar.zst.sig
neovim-0.9.5-2-x86_64.pkg.tar.zst
neovim-0.9.5-2-x86_64.pkg.tar.zst.sig
```

    We have an older version of neovim installed, and to roll it back we can use

```
pacman -U neovim-0.9.5-1-x86_64.pkg.tar.zst
```

The pacman log (`/var/log/pacman.log`) is also useful since it logs all pacman outputs when you do anything with pacman. So if you are looking for the packages that have been installed in the latest `pacman -Syu`, then you can use this to individually see each package that has been upgraded.

1.2.2 Yay

Yay is used to install from the Arch User repository and must be updated separately. To run this, you can either run `yay -Syu` or you can just run `yay`. Since this is not officially maintained, these packages are more likely to break something. The yay logs are not stored separately can can be accessed in the pacman logs.

1.2.3 Apt

1.3 Display Servers

X11, Xorg, Wayland.

1.4 Windows Managers and Desktop Environments

These days, the terms window managers (WMs) and Desktop Environments (DEs) are used interchangeably, but they mean slightly different things. A window manager is the display software that determines how the pixels for each window overlaps with other and their movement. This is generally divided into two paradigms with the most familiar being **floating WMs** and the other being **tiling WMs**. Even before I knew about tiling WMs, I found myself manually tiling windows on floating WMs, so the move to tiling WMs was a no-brainer.

Some DEs and WMs are:

1. GNOME
2. KDE Plasma
3. Qtile

1.5 File Tree of Linux Systems

The root director contains the following directories. Let's briefly go over what each of them do:

1. **bin**: Contains all the binaries (i.e. executables) and is usually included in the PATH environment variable. Elementary commands like `ls` or `cd` reside in this directory. If you pay attention, you'll see that **/bin** is a symlink to **/usr/bin**. What about **/usr/local/bin**?
2. **sbin**: Also a symlink to **/usr/bin**.
3. **tmp**: A temporary folder for storing caches, local configurations, and a place to download packages to build and install.
4. **boot**: Booting files.
5. **home**: Contains all users.

1.6 Shells and Terminals

Beginners may think of the shell and the terminal to be the same thing, but they are different. The **shell** is a command line interpreter, a layer that sits on top of the kernel in which the user can interact with. It is essentially the only API to the kernel where the user can input commands and processes them. The **terminal emulator** is a wrapper program that runs a shell and allows us to access the API. It may be useful to think of the shell as like a programming language and the terminal as a text editor like VSCode.

The three most common shells are the following:

1. **Bash**:
2. **Zsh**:
3. **Fish**:

Some common terminal emulators (most of which comes as a part of the desktop environment) are the following:

1. **Kitty**:
2. **Alacritty**:
3. **Gnome-Terminal**:

1.7 Boot Configuration and Automatic Scripts

Find which file you need to put commands in to load stuff up on boot (and specify which part of the boot process). Also talk about crontab.

2 Booting

2.1 Hardware

2.2 BIOS and UEFI

2.3 FileSystems

2.4 Systemd

When the computer boots up, the init process is the first process (PID 1) that the kernel starts. It is used to get the system running and for starting other processes. It is responsible for killing zombie processes or parenting orphaned processes.

Historically, SysVinit was a series of plaintext files that ran as scripts to start processes, but due to some problems, Linux now uses systemd. **Systemd** is a collection of smaller programs, services, and libraries such as systemctl, journalctl, init, process management, network management, login management, logs, etc. Some processes may depend on other processes, and with hundreds of them, it's very hard to do manually, which is why systemd does it all for you. A **unit** is anything that is managed by systemd, and their files are all over the place:

1. `/lib/systemd/system` contains standard systemd unit files
2. `/usr/lib/systemd/system` are from locally installed packages, e.g. if I installed a pacman package that contained unit files, then those would go here.
3. `/etc/systemd/system` is where you put your custom files. etc also has the highest priority, so it overwrites the other files.

If we go into one of these unit files, which have the prefix `.service`, they are usually formatted as such:

```
# comments are just the same as in bash Scripts
# the headers are important!

[Unit]        #
Description=Description of the unit file
Documentation=man:something
After=network.target

[Service]
Type=forking  # tells that the process may exit and is not permanent
PIDFile=      #
ExecStartPre= # scripts to run before you start
ExecStart=    # scripts to run when starting
ExecReload=   # script to run when you try to reload the process
ExecStop=     # script to run to stop the process

[Install]   # Tells at what point should this be running
WantedBy=multi-user.target
```

The two main commands that you will use to interact with systemd is `systemctl` and `journalctl`.

1. `systemctl status pkg` checks the status, ouputting the description, whether it's enabled/disabled, and whether it's active/inactive.
2. `systemctl enable pkg` enables it, which means that it will start when booting the computer. It does this by creating a symlink to the unit file. This is different from start.

3. `systemctl disable pkg` disables it.
4. `systemctl start pkg` starts it now and runs it immediately.
5. `systemctl stop pkg` makes it inactive.
6. `systemctl reload` will run whatever is in the `ExecReload` in the unit file.
7. `systemctl restart` runs ExecStop and then ExecStart.
8. `systemctl kill pkg` kills the process.

Some of the statuses that you may see are inactive (deactivated, exited), active (activating, running), failed, static (not started, frozen by systemd), bad (broken, probably due to bad unit files), masked (ignored by systemd), indirect (disabled, but another unit file references it so it could be activated).

To troubleshoot, you should run `systemctl --failed` to see if there are any failed processes, which can be a problem, and then you can use `journalctl --since=today` to view your systemd logs. This log is important for diagnosing fundamental problems with your system. To view only entries logged at the error level or above, you can set the priorities with `-p err -b`.

3 LaTeX and VimTex

Latex is a great way to take notes. One can go to Overleaf and have everything preconfigured, but in here I set it up on my local desktop. I will already assume you have a PDF viewer installed. I use zathura, which is lightweight and also comes with vim motions for navigation.

First install the VimTex plugin in `plugins.lua` with `use lervag/vimtex`. Then, you want to install TexLive, which is needed to compile tex files and to manage packages. The directions for TexLive installation is available [here](https://tug.org/texlive/quickinstall.html). Once I downloaded the install files, I like to run `sudo perl ./install-tl --scheme=small`. Be careful with the server location (which can be set with the `--location` parameter), as I have gotten some errors. I set `--scheme=small`, which installs about 350 packages compared to the default scheme, which installs about 5000 packages (7GB). I also did not set `--no-interaction` since I want to slightly modify the `--texuserdir` to some other path rather than just my home directory.

Once you installed everything, make sure to add the binaries to PATH, which will allow you to access the **tlmgr** package manager, which pulls from the CTAN (Comprehensive TeX Archive Network) and gives VimTex access to these executables. Unfortunately, the small scheme installation does not also install the **latexmk** compiler, which is recommended by VimTex. We can simply install this by running "' sudo tlmgr install latexmk "' Now run ':checkhealth' in Neovim and make sure that everything is OK, and install whatever else is needed.

To install other Latex packages (and even document classes), we can use tlmgr. All the binaries and packages are located in `/usr/loca/texlive/202*/` and since we're modifying this, we should run it with root privileges. The binaries can also be found here. Let's go through some basic commands:

1. List all available packages: `tlmgr list`
2. List installed packages: `tlmgr list --only-installed` (the packages with the 'i' next to them are installed)
3. Install a package and dependencies: `sudo tlmgr install amsmath tikz`
4. Reinstall a package: `sudo tlmgr install amsmath --reinstall`
5. Remove a package: `sudo tlmgr remove amsmath` More commands can be found here for future reference.

After this, you can install Inkscape, which is free vector-based graphics editor (like Adobe Illustrator). It is great for drawing diagrams, and you can generate custom keymaps that automatically open Inkscape for drawing diagrams within LaTeX, allowing for an seamless note-taking experience.

4 Networking

Networking is a large field in itself, but in here I go over the most useful and practical applications of it in my everyday use. Some ways that I personally benefit from this is:

1. Connecting to WiFi and diagnosing problems.
2. Connecting to WiFi and diagnosing problems.
3. Connecting to other networks such as computing clusters or third-party blockchains.
4. Seeing how more abstract schemes such as APIs work.
5. Ethical hacking.

I introduce these concepts and how to do some basic implementation a Unix operating system.

I like to learn about networking as if I am designing it from scratch. Some big questions to ask when designing network schemes are:

1. How do we uniquely identify computers?
2. How should we establish a connection between them? Through hardware or signals?
3. What protocols should we use, like a common language, so that all computers understand what each other are saying?
4. Can we implement security measures to prevent unwanted visitors into our computer?

5 Network Topologies and IP Addresses

Talk about public vs private networks (IPv4 vs v6). How data is transmitted (bandwidth, latency, etc). Then talk about ports.

In here we address the first of the big questions. Every computer has an IP address. We can access our public IPv4 address with the command `curl ifconfig.me`. Since this is public, any device connected to the same network/router should have the same IP address. However, if we want to find our private IP address, we use the following command: `ip -c a`

```
~ ip -c a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1
    link/ether 64:bc:58:11:c0:24 brd ff:ff:ff:ff:ff:ff
    inet 172.30.1.61/24 brd 172.30.1.255 scope global dynamic noprefixroute wlan0
       valid_lft 3235sec preferred_lft 3235sec
    inet6 fe80::6ab9:70c3:f583:ff3e/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

There's a few things to look at:

1. The type of connection that I am on is important. The 'lo' (loopback) connection is default to every computer, representing the local connection. 'wlan*' refers to WiFi, 'eth*' represents ethernet.

Every device has a public/private IP if it is connected to the internet. To connect to the computer, you need to know the IP and then find a port to connect to. Computers usually have $2^{16} = 65,536$ ports, and whenever computer A is connecting to computer B, data is both coming in and out through certain ports. A port, combined with an IP address, results in a socket address that is used to establish a connection between a client and a server.

5.1 OSI and Internet Protocols

5.2 HTTP and HTTPS

HTTP stands for hypertext transfer protocol, implemented in Layer 7, which transfers data between your computer and the server over the internet through **clear text**. This may not be the most ideal way since any interceptors can read the transferred data. This isn't a problem for regular internet browsing, but if you are inputting sensitive data, then HTTP should not be used. This is why HTTPS (which stands for secure HTTP) was invented, which is implemented in Layer 4 and encrypts the data being transferred, and every website where you input sensitive data should be using HTTPS (indicated by the `https://` prefix in the URL and a padlock symbol for modern browsers). Due to the extra security measures, HTTPS is less lightweight than HTTP, and its respective default ports are HTTP (80) and HTTPS (443).

A natural question to ask would be: which encryption scheme does HTTPS use? Both Secure Sockets Layer (SSL) and Transport Layer Security (TLS) is used in the modern web.

SSL certificate.

5.3 UDP and TCP

TCP handshake can be seen with curl.

5.4 SSH

6 Driver and Hardware Configuration

6.1 Audio Drivers

6.2 Bluetooth

Blueman.

6.3 Synaptics

6.4 Video Drivers

6.5 Monitor

6.6 Nvidia GPU Drivers

7 Development

7.1 Git

7.2 Python and Conda

Make sure to add conda path to PATH.