CS 470: Unix/Linux Sysadmin
Spring 2025 Lab 4
Ubuntu, containers with Docker, and web server with NGINX

Things from prior labs that are required to begin this lab:

- lab 3: Rocky Linux up, online, and sharing files via NFS

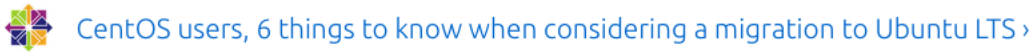
Things in this lab that are on the critical path for upcoming labs:

- getting Ubuntu installed, running, and connected to NFS
- getting Docker and NGINX operational

co-authoring credits:

- 2024 TA Bryan Zublin replaced BitWarden with VaultWarden and fleshed out the Linux boot process

This lab gets our hands dirty with the other super-major Linux distribution, and by far the most popular, Ubuntu. With the tactical missteps recently made by Red Hat as discussed in lab three, expect that gap to continue to grow. Ubuntu was clearly ready to capitalize; the screenshot below is of a link on the main page of the website at ubuntu.com, the day after Red Hat announced it was killing off CentOS as we knew it.



The link was there for almost three years, on the main page at ubuntu.com, without even having to scroll to see it. That says something.

We often call Linux operating systems "distributions" because, like we've discussed in lecture … Linux isn't an operating system. Linux is a kernel, and a collection of tools directly around, and related to that kernel. Everything else that a vendor puts around it, that makes it a whole operating system … uses Linux, but is not Linux. So we end up with these distributions, these piles of software, built around the Linux kernel, organized by vendors, that provide the value we expect of an operating system. Red Hat Linux, Ubuntu Linux … you get the idea.

Ubuntu Linux is based, in turn, upon Debian Linux, which is named for the project's founders, Deb and Ian. Debian-based Linux distributions tend to use the `.deb` package format and the `apt` package manager, very analogous to the situation with `.rpm` packages and the `yum` or `dnf` package managers on Rocky Linux and Red Hat-derived distributions, of which there are several, as discussed at the beginning of lab 3.

Other Debian-based distributions include Kali Linux, which some of you may have already used in CS 574/596, and ParrotOS, another security distribution I'm starting to like better than Kali. Ubuntu is so popular that many popular Debian derivatives are based directly on Ubuntu, rather than on Debian, for instance Linux Mint (used in prior iterations of this class), Kubuntu (Ubuntu with the KDE desktop instead of GNOME), Lubuntu (LXDE/LXQT desktop), Xubuntu (with Xfce), and Ubuntu Budgie (with the Budgie desktop).

Many who use Ubuntu came across it because it used to have the best-looking desktop … it's always been the one that was closest to something you could give your parents, or grandparents, to use. Ubuntu used to develop its own desktop environment, called "Unity," but has recently dropped it in favor of making its own additions to the community-developed GNOME desktop environment. I wholeheartedly suggest you *try*

Ubuntu Desktop on your own time later … but for this lab, we'll be using Ubuntu Server, with no graphical desktop like our minimal Rocky Linux install, to conserve resources.

Unlike Red Hat, there is no free similar distribution (like CentOS was and Rocky and Alma now are for Red Hat) for free patches … because patches for Ubuntu are just provided for free; Canonical, the company behind Ubuntu simply chooses not to tie downloads to you signing in, or to tie patches and updates to paid support ... Red Hat wants you to pay for that. If you want Ubuntu, you just download it. If you want support for Ubuntu, you just pay for it. It's that easy … and that's why people have been flocking to Ubuntu, in droves.

Your Ubuntu VM is going to be your lab network's web server. We're not going to design any websites here, but HTTP has long since grown past merely serving up websites, and is arguably the most-used protocol on the internet, for websites, for applications, the file service, for APIs. In order to spare you the mundane ease with which one can typically set up a web server today, we're going to take this opportunity to both introduce a layer of abstraction **and** to introduce a new technology, Docker. We'll run our web server inside Docker.

The idea behind Docker is a simple but powerful one: virtual machines are great for a lot of things, especially for creating a logical layer of separation between applications. They can also be very wasteful, though. For each virtualized server instance, we typically install a whole copy of an operating system each time, at a cost of anywhere from 1 to 10 GB of storage for each operating system instance. We also consume the memory (RAM) involved with running that additional operating system instance for each virtual machine. On top of that, additional storage, memory, and CPU are consumed emulating virtual hardware for each virtual instance.

Docker calls its instances "containers," and uses a very minimalist approach, bordering on extreme para-virtualization. Containers share services and resources and network stacks with a host operating system wherever possible, and this reduces the footprint inside each container's filesystems to **just** the base libraries and files required by the service running in each container. Wherever it makes sense, files or data to be served by containers are mounted into each container from the host computer's filesystems.

The result is profound; though we lose a lot of the logical separation provided by full-blown virtual machines, we get tremendous resource savings, because all of our service units are just the size of the service, with a much thinner logical layer of separation. If we need to add in common data files, we can "map" folders with common service data, into multiple containers if needed, and either ramp up a fleet to scale, or just provide a thin logical layer of separation and abstractions between separate services in our server fleet.

Docker gives you a great, economical way to cloud-host a lean-as-possible service instance, and to separate the services within that system from one another, by more than just filesystem access controls.

Since we showed you how to build a BSD kernel in lab 1, we'll also be building a Linux kernel from source in this lab … there are lots of reasons you might end up having to do this in field, so might as well get some practice in now. Compiling the Linux kernel requires a lot of RAM in this VM, so much so that this side quest is what put the class past supporting hosts with 8 GB of RAM. Your lab 4 VM will initially be configured with 8 GB of RAM … the Linux kernel will fail to build with less. We will peel back those resources

part zero: get it

Go to www.ubuntu.com and click the "Download Ubuntu" menu along the top bar, then click "Server," then "Get Ubuntu Server." Those with Intel CPUs and VMware hypervisors can click on "Get Ubuntu Server," then "Download 24.04.2 LTS." Those with ARM Macs should click on "alternative architectures," then "Download 24.04.2 LTS."

The download for Intel CPUs is 3.0 GB; the download for ARM CPUs is 2.7 GB. LTS stands for "long-term support." Amongst major releases of the operating system, those releases tagged "LTS" target stability, and Ubuntu will sell support for these releases for up to ten years, depending upon how you pay for it. Ubuntu versions not tagged with LTS at the end are development releases with shorter life spans.

part one: install it

Create a new custom VM on your shared VM network, using the following specifications:

- guest OS:
  - Intel/VMware: Ubuntu 64-bit
  - ARM/UTM: virtualize/Linux
- CPU: two virtual cores or processors
  one processor, two cores if you're on VMware Workstation
- boot firmware: UEFI
- RAM: 8 GB (8192 MB)
- hard disk: 16 GB

As before, those of you in UTM will want to make sure you're not using a GPU-supported virtual graphics card. Whereas the graphics card used to give us little difficulties in any operating system, it was just recently flaky in Rocky Linux, and even more so here with Ubuntu while we were updating this lab 4. Since we don't care about graphics anyways, those of you in UTM can side-step a virtual display entirely and use a virtual serial console for this VM. If you edit your VM, under devices, select the display and remove it. Then hit "new" under devices, and select to create a new serial device. All the defaults will be fine, and you'll have a much less flaky way of interacting with your VM.

Those of you in VMware, please de-select "Easy Install." The installation of Linux, and Ubuntu especially, are easy enough, and we don't want to hide from the details here ... especially when VMware's "Easy Install" just seems to make things hard. When the installer window comes up, if you've done Debian Linux before, you might recognize similarities.

The text-based version of Ubuntu's installer, included with their server ISOs, is similar in design and spirit to FreeBSD's text-drawn menus installer. Just like there, you'll be using the space bar and return keys to select things, and the arrow keys and tab to navigate between text-based UI elements.

1. As always, select your language and keyboard layout.

After language selection, the Ubuntu installer will sometimes offer to update itself. While Ubuntu's distributions are generally pretty solid, their installer has been hideously unstable at times ... I suspect this is why they developed a post-boot online installer update feature. This installer update feature often used to crash on me, but it hasn't failed on me in a couple of years now. If it fails for you, just reboot from the installation media (the ISO) and try the installation again, without letting it update.

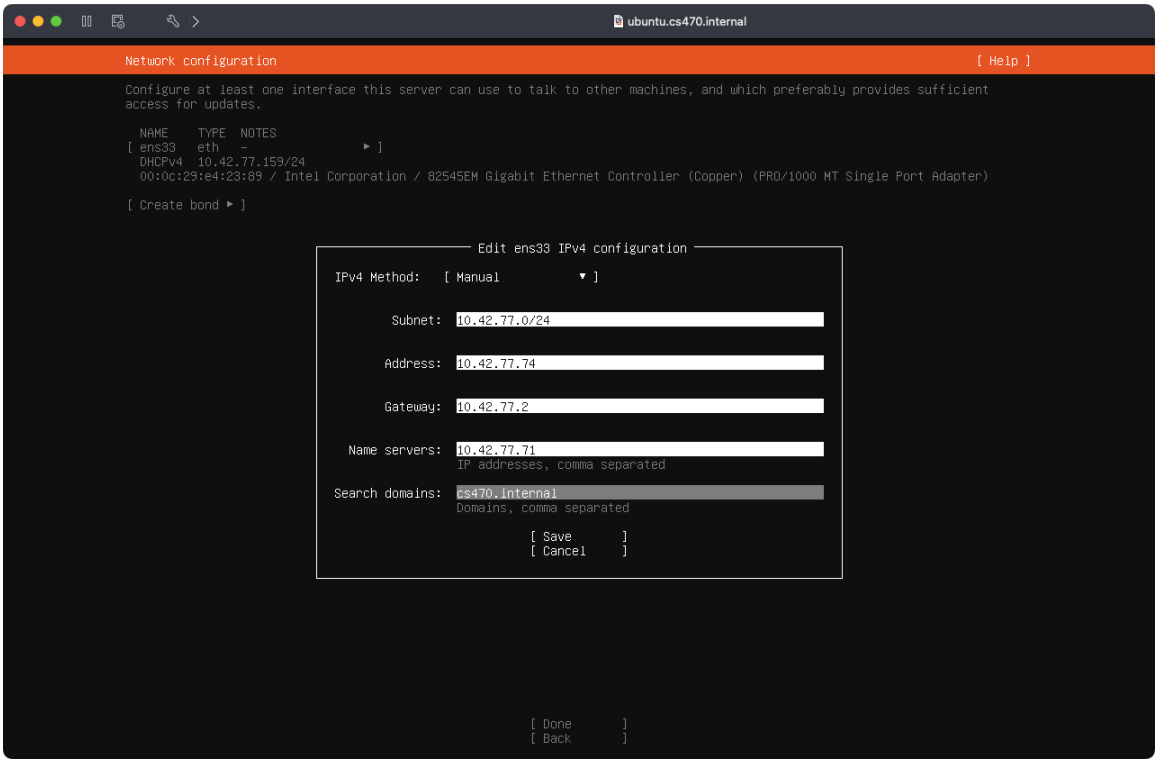
If you don't get the offer to update your installer, don't worry. You can just skip this step.

2. The next screen prompts us to "choose type of install" … let's do an Ubuntu Server, **not** minimized. It still has a fairly small footprint.

3. Network setup is next; just like with the last two systems we set up, we know our network details and we're going to set a static IP right here in the installer. Choose your VM's only network interface with the return or space key, then choose to edit IPv4, and change from automatic configuration to manual.

   In the subnet field, the installer expects the network address of your VMware NAT subnet, in CIDR notation. This is the .0 reserved "network" address on that subnet, followed by "/24," relating the netmask and thus the size of the subnet. Mine is 10.42.77.0/24.

   For address, remember your Ubuntu system is .74 on your VMware subnet. The gateway is whatever you discovered way back in lab 1, as usual. For the name server, provide the IP address of your OpenBSD VM, and cs470.internal as the only search domain.



4. Proxy?! We don't need no stinkin' proxy! Proxy servers are generally only required on the most security-conscious networks, where network operators care to inspect all inbound and outbound

traffic. That doesn't apply to us … we're directly connected to the internet, so leave it blank.

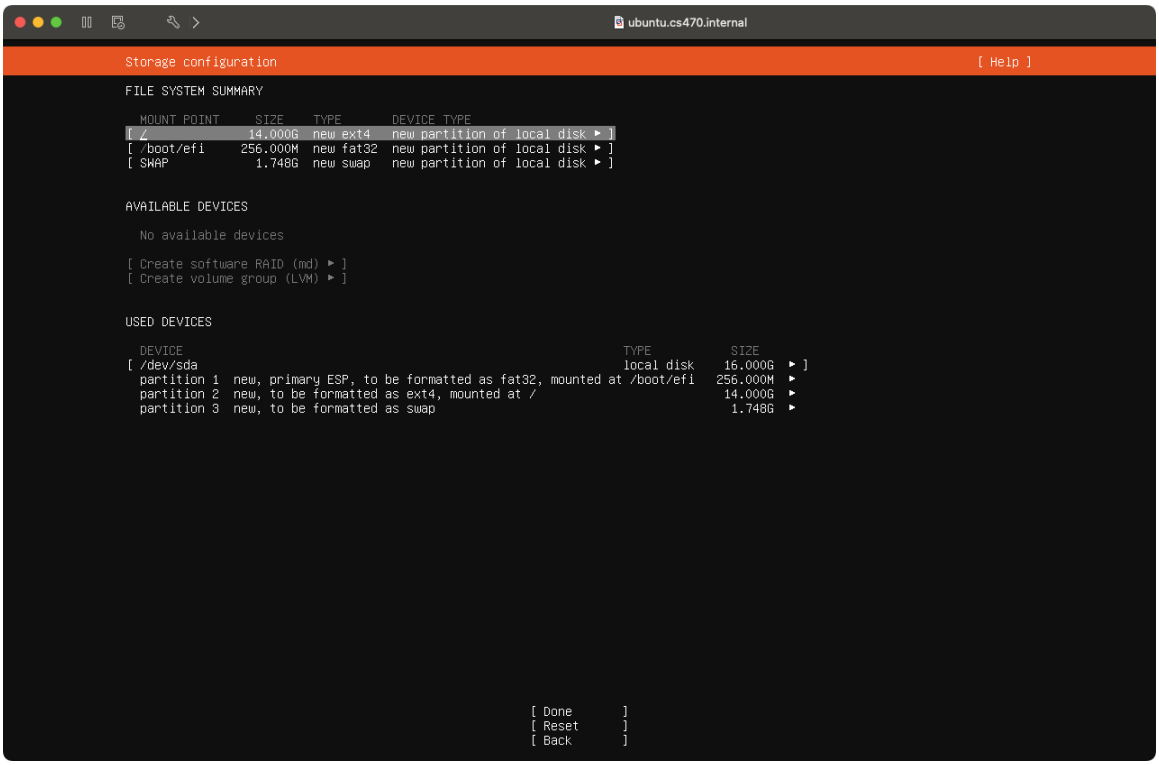
5. On the mirror selection screen, the default archive mirror is fine, or whichever mirror it provides.

6. On the storage configuration screens, start by selecting "custom storage layout." As before with Rocky Linux, I recommend you avoid LVM like the plague unless you're on physical hardware where you can actually make good use of it. Inside a VM, LVM is an unnecessary layer of abstraction.

   You should only have one disk option under "available devices," probably `/dev/sda` on Intel and `/dev/vda` on ARM. Just like in BSD, the first letters `sd` (SCSI disk) or `vd` (VirtIO disk) is the device type and type. Linux uses letters for the device instances … `a` means the first disk, `b` the second and so on, and partitions, as in Rocky, will be numbered. Select that disk device, whichever it is, and first, choose to use it as the boot device. You should see it create an EFI system partition ("ESP"). By default, it created a 768 MB ESP partition for me … if it does the same for you, edit it and reduce it to 256 MB.

   Select the disk again, its free space area, and choose to add a GPT partition. Make a 14 GB root filesystem, with the ext4 filesystem. Completing the setup of the root filesystem by selecting "create" will return you to the screen where you selected to create the root filesystem on `/dev/sda`. It should show that right under 2 GB remain available (mine said 1.748G).

   Select that free space area again, and create a swap petition with all remaining space.

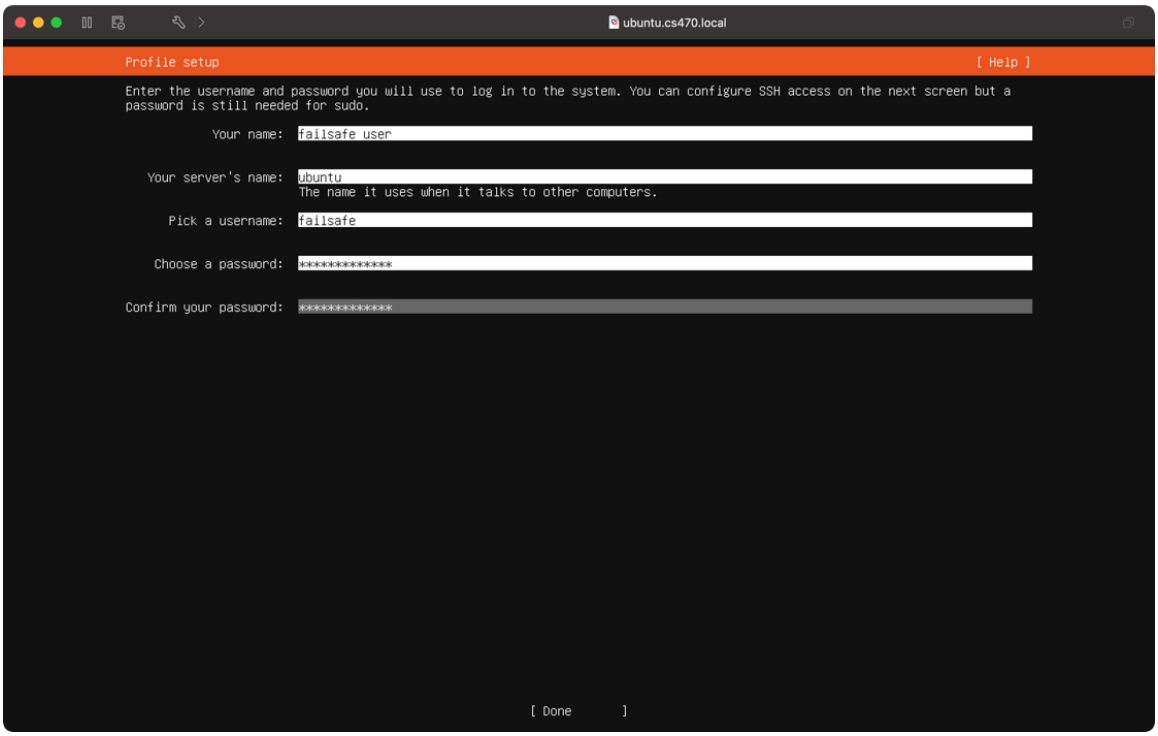


Your disk setup should look similar to the screenshot above. Once you're content with your disk setup, choose "done." It's going to ask you to confirm whether you want to take destructive actions …

remember that your VM's hard disk is just a virtual disk in a file on your computer, and you can "continue" without hesitation.

7. The Ubuntu installer will then go to the "profile setup" screen, where it'll ask you to set up a user. Create a user named `failsafe`, as always, and use `ubuntu` as your server's name.

   The below screenshot is from the installer from 22.04.3, before somebody removed the apostrophe, correctly indicating ownership, from "your server's name." Both the ARM64 and AMD64 installers for 22.04.4 and 24.04 have the apostrophe incorrectly removed.



   After you hit "done" on this screen, take a pass on Ubuntu Pro if offered it … please, however, feel free to look into it and opt in later if you please. On the following screen, please **DO** select to install OpenSSH server on the "SSH Setup" screen. Do NOT select to import an SSH identity. After you hit "done" on the SSH setup screen, you get "featured server snaps." Take a pass, go straight to "done."

   Hitting "done" starts the installation, and the top of the screen soon said "install complete!" but the installer was still runing "curtin" operations for a few minutes to install updates … hmmmm. I do not think complete means what they think it means! They haven't learned their lesson about when to correctly use an apostrophe, but hopefully they've learned their lesson about fighting a land war in Asia, or going in against Sicilians when death is on the line.

   After a couple minutes, I got the option to "reboot now," and I took it … you should too.

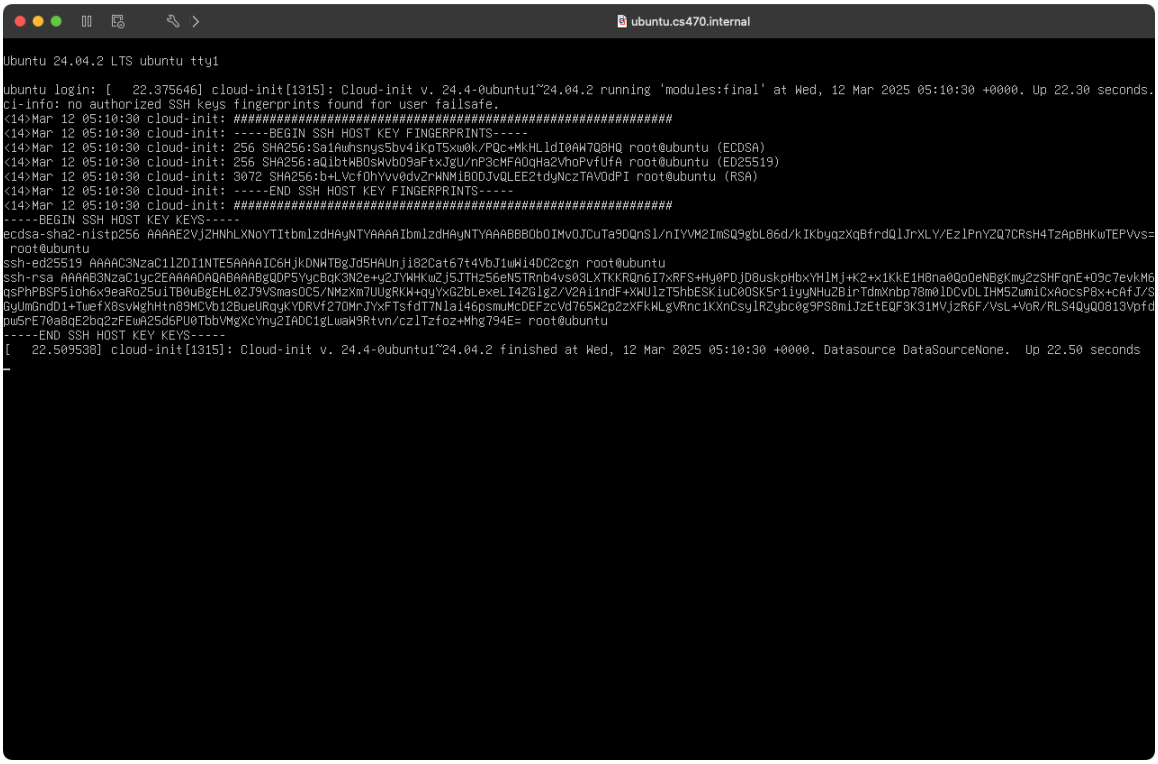
   The installer, as a parting gift, asked me to remove the installation medium (again, this is the `.iso` file in the VM's virtual optical drive that it is referring to) and press `enter` or `return` to reboot. Make sure

the `.iso` file is "disconnected" from the virtual optical drive in your VM, and hit `return`. This is one of the few changes that you can make to the VM while it's turned on … because we're simply ejecting a CD-ROM from the computer.

Hang on to your Ubuntu ISO. We'll be using it again in lab 8.

8. After the reboot, you'll get a login prompt.

   If the login prompt is buried on screen under a bunch of final, first-boot setup stuff, like generating a host key pair for the VM's SSH service, don't worry. If you hit the carriage return key after it stops doing things – my VM stopped after it reached the `cloud-init` target – you'll get a fresh login prompt.



   Note that you set no root password … on Ubuntu, like on macOS, and a lot of OSs these days, the root account is typically locked as a consquence of not setting a password ... no password will work to log you in. You are expected to use `sudo` whenever you want to flex your admin rights here.

   Also note, Ubuntu uses the same, standard GNU/GPL userland implementations of `shutdown`, `reboot`, and `poweroff` … and at this point in time, I shouldn't have to remind you when and how to use them.

9. Remember what I said about Ubuntu being descended from Debian Linux? Take a moment after you log in to check out the contents of the file `/etc/os-release`, especially the variable `ID_LIKE`, and see how you would identify Ubuntu if you were building something on top of it. Please also take a look at `man os-release`, especially if you're wondering what any of this stuff is actually intended for.

part two: additional configuration and installations

10. Ubuntu historically used the file `/etc/network/interfaces` to configure its network(s) … but that file, and its backing subsystems, have been deprecated since Ubuntu 18.04 in favor of the super-duper-mega-overengineered `netplan`. `netplan` initially stored its configurations in the folder `/etc/netplan`, but now it even use other network configuration "renderers" like NetworkManager for an additional layer of abstraction when setting up the network. Because the Ubuntu installer detected we were running under VMware, it bundled the network configuration along with – and named it after – a common suite of cloud instance initialization scripts, `cloud-init`. Which you might have seen littering up your Ubuntu VM's console after the first boot, or could see in my screenshots from Ubuntu 22.04.

    Like a lot of things in life, `cloud-init` and `netplan` are really intrusive, under the guise of trying to be helpful and prevent "harm."

    It's for this reason I want you to see how the network is statically configured on your Ubuntu VM.

    ```
    $ sudo more /etc/netplan/50-cloud-init.yaml
    ```

    Mine has the following contents.

    ```
    network:
      version: 2
      ethernets:
        ens33:
          addresses:
          - "10.42.77.74/24"
          nameservers:
            addresses:
            - 10.42.77.71
            search:
            - cs470.internal
          routes:
          - to: "default"
            via: "10.42.77.2"
    ```

    Unless your network is broken, you don't have to take any action the rest of this step.

    If you ever have to change it, save it out, and tell `netplan` to generate the necessary configuration …

    ```
    $ sudo netplan generate
    ```

    … and then apply it.

    ```
    $ sudo netplan apply
    ```

    Linux distributions, increasingly, are moving away from `ifconfig` towards `ip`. To check out your network configuration, try the command `ip a` … as in lab 1, if you can't ping your gateway, you got one of the IP addresses wrong. If you can't ping outside your gateway (say 4.2.2.2), you got your gateway IP wrong, probably. If you can't look up names, or ping you probably got your nameserver

wrong, or it's just not working correctly.

11. Ordinarily, we'd set up SSH key-based authentication here. Instead, we're going to set up the NFS client on this VM instead, and get our SSH directory – and the entirety of `/home` from the NFS server on the Rocky VM – and we won't have to copy over our SSH public key. In order to do *that*, we need to initialize Ubuntu's package management system, `apt`:

    ```
    $ sudo apt update
    ```

    You should see `apt` go out and grab the lists of the latest packages from Ubuntu's repository, probably the same "repo" as set up during the installation, and finish with something like …

    ```
    50 packages can be upgraded. Run 'apt list --upgradable' to see them.
    ```

    … only we don't care about that yet. We want NFS first.

    ```
    $ sudo apt install nfs-common
    ```

    `apt` will tell you that it's going to download and install precisely which packages, once it calculates all of the dependencies for `nfs-common`, and ask for your confirmation to proceed. After it's done installing, you should be able to mount the NFS shares of `/home` and `/srv/nfs` from your Rocky VM. First, `cd` out of `/home` …

    ```
    $ cd /
    ```

    … and then manually mount it …

    ```
    $ sudo mount rocky:/home /home
    ```

    … make the mount point for `/srv/nfs` and mount it too.

    ```
    $ sudo mkdir /srv/nfs
    $ sudo mount rocky:/srv/nfs /srv/nfs
    ```

    … running the `mount` and/or `df` commands should confirm that it's properly mounted. If it's not, you goofed something up above this – there's not much of it yet – and should backtrack.

12. You should now be able to SSH into your Ubuntu VM without using a password, using key-based authentication, as your public key is in place (`~/.ssh/authorized_keys`) thanks to the NFS mount. Test it. Log into your Ubuntu VM over SSH ... you may have to accept its SSH host key, as it is a new SSH server to your client, but it should not ask for a password. If it does, you missed something.

13. Having the same home directory everywhere is the way home directories should be; let's make our NFS mounts permanent. Use `vi` to add the following lines to `/etc/fstab` on your Ubuntu VM:

    ```
    rocky:/home /home nfs defaults 0 0
    rocky:/srv/nfs /srv/nfs nfs defaults 0 0
    ```

Check twice for typos – remember, a problem mounting any filesystem listed in `/etc/fstab` will cause your system to fail to fully boot up. Test it out by rebooting your VM; if you're going to have to go into recovery mode, better now when expected over later and as a surprise.

14. Once you get your VM back from the reboot, let's take a couple minutes to install packages used later by your instructor for grading purposes: `csh` (via `tcsh`), GNU `binutils`, and `net-tools`.

    ```
    $ sudo apt -y install tcsh binutils net-tools
    ```

15. It's time for network time! Ordinarily we'd go for `ntpd` and `ntpdate` but `systemd` comes with a new NTP client, and we don't need to do anything to turn it on.

    ```
    $ systemctl status systemd-timesyncd
    ```

    If it doesn't report that it's running and active, you should know what to do, given that it's integrated with `systemd` … and be able to check that it's running another way:

    ```
    $ ps auxww | grep timesyncd | grep -v grep
    ```

    The `-v` switch with `grep` excludes, instead of matches lines, so we're making sure we only see a line if there's an `timesyncd` process, not a line for our `grep` process trying to find it.

16. Next, mail and mail forwarding … Ubuntu appears to come with no built-in mail subsystem. This command returned no output.

    ```
    $ which mail
    ```

    This command also returned no output …

    ```
    $ which sendmail
    ```

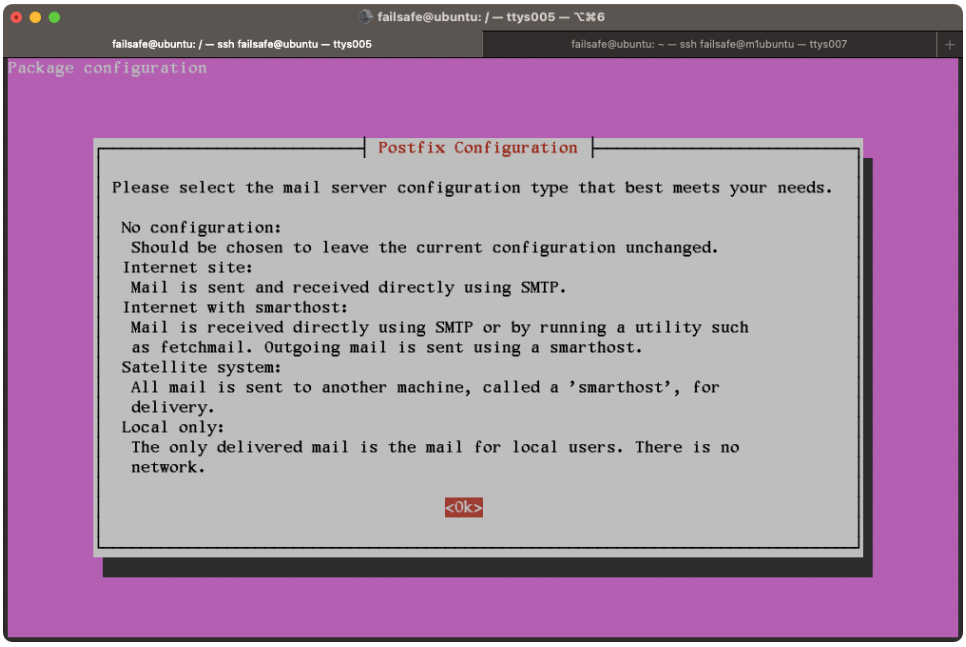
    … and most mail servers offer a `sendmail` command, for historical reasons.

    ```
    $ apt list --installed | grep -i mail
    ```

    This command also returned no useful output, just a warning about how the command line interface (CLI) for `apt` is in flux, and to be careful using it in scripts. So, let's install `postfix`; it's **way** easier to configure than `sendmail`.

    ```
    $ sudo apt install postfix
    ```

    After confirming you want to install `postfix`, you'll be greeted with a FreeBSD-looking text-based menu dialog asking which of configuration template you want. It used to look like the screenshot below, with descriptions of each configuration.



Choose "internet site" and specify `ubuntu.cs470.internal` as the system mail name.

In the middle of the output that follows, you should see something like this ...

```
Setting up postfix (3.8.6-1build2) ...
info: Selecting GID from range 100 to 999 ...
info: Adding group `postfix' (GID 111) ...
info: Selecting UID from range 100 to 999 ...

info: Adding system user `postfix' (UID 112) ...
info: Adding new user `postfix' (UID 112) with group `postfix' ...
info: Not creating home directory `/var/spool/postfix'.
Creating /etc/postfix/dynamicmaps.cf
info: Selecting GID from range 100 to 999 ...
info: Adding group `postdrop' (GID 112) ...
setting myhostname: ubuntu.cs470.internal
setting alias maps
setting alias database
changing /etc/mailname to ubuntu.cs470.internal
setting myorigin
setting destinations: $myhostname, ubuntu.cs470.internal, ubuntu,
localhost.localdomain, localhost
setting relayhost:
setting mynetworks: 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128
setting mailbox_size_limit: 0
setting recipient_delimiter: +
setting inet_interfaces: all
setting inet_protocols: all
/etc/aliases does not exist, creating it.
WARNING: /etc/aliases exists, but does not have a root alias.

Postfix (main.cf) is now set up with a default configuration.  If you need to
make changes, edit /etc/postfix/main.cf (and others) as needed.  To view
Postfix configuration values, see postconf(1).

After modifying main.cf, be sure to run 'systemctl reload postfix'.

Running newaliases
```

```
Created symlink /etc/systemd/system/multi-user.target.wants/postfix.service →
/usr/lib/systemd/system/postfix.service.
```

As you can see, the package post-installation script automates a lot of the postfix configuration we did manually in lab 3. It makes a new separate user, called `postfix`, to use when running the `postfix` mail server. If you use `ps auxww` and filter output to match `postfix`, you should see that it's running already. Now, as the installation messages suggest, we need to edit `/etc/aliases`. Add the following line to forward root's mail …

```
root: peter@cs470.internal
```

… and of course, replace my username with yours, and run `newaliases` to tell the mail subsystem to re-process the aliases database.

```
$ sudo newaliases
```

17. Now let's test our mail configuration. To do that, we need a command-line mailer, but …

```
$ which mail
```

… returns no output. So let's install `apt-file` to search the `apt` database for files included with each package …

```
$ sudo apt -y install apt-file
```

… `apt-file`, like `apt`, needs to be told to build its database …

```
$ sudo apt-file update
```

… and finally, search for `mail` ...

```
$ apt-file search mail
```

Wow, that command returns a lot of output! Unsurprising that a lot of packages deal with mail in some fashion, though … let's filter that output a bit by piping it through `grep` …

```
$ apt-file search mail | grep -w mail
```

… `grep` with the `-w` switch only matches output where "mail" is the whole word in each match, not a part of a larger word. Still not helpful. Maybe matching mail with a space after it? It ended up being helpful to remember that some implementations of `/bin/mail` were called `mailx` … because going straight for `mailx` helped a ton at reducing noise.

```
$ apt-file search mailx
bsd-mailx: /usr/bin/bsd-mailx
bsd-mailx: /usr/share/bsd-mailx/mail.help
bsd-mailx: /usr/share/bsd-mailx/mail.tildehelp
bsd-mailx: /usr/share/doc/bsd-mailx/README.Debian.gz
bsd-mailx: /usr/share/doc/bsd-mailx/changelog.Debian.gz
bsd-mailx: /usr/share/doc/bsd-mailx/copyright
```

```
bsd-mailx: /usr/share/man/man1/bsd-mailx.1.gz
mailutils-doc: /usr/share/doc/mailutils/mailutils.html/mailx-mail-variable.html
mailutils-mh: /usr/share/mailutils/mh/scan.mailx
manpages-pl: /usr/share/man/pl/man1/bsd-mailx.1.gz
manpages-posix: /usr/share/man/man1/mailx.1posix.gz
mmh: /etc/mmh/scan.mailx
mon: /usr/lib/mon/alert.d/mailxmpp.alert
nmh: /etc/nmh/scan.mailx
```

Bingo, much better. Looks like `bsd-mailx` is the droid we're looking for.

```
$ sudo apt -y install bsd-mailx
```

18. Finally, we have something to test with …

```
$ echo 'test' | mail -s test root
```

… and checking the mail log …

```
$ sudo tail /var/log/mail.log
```

… I saw this …

```
2025-03-23T20:56:15.688174+00:00 ubuntu postfix/master[6467]: daemon started --
version 3.8.6, configuration /etc/postfix
2025-03-23T21:16:40.094759+00:00 ubuntu postfix/pickup[6468]: 16FA741ADD: uid=1000
from=<failsafe>
2025-03-23T21:16:40.107839+00:00 ubuntu postfix/cleanup[7205]: 16FA741ADD: message-
id=<20250323211640.16FA741ADD@ubuntu.cs470.internal>
2025-03-23T21:16:40.109818+00:00 ubuntu postfix/qmgr[6469]: 16FA741ADD:
from=<failsafe@ubuntu.cs470.internal>, size=431, nrcpt=1 (queue active)
2025-03-23T21:16:40.125152+00:00 ubuntu postfix/cleanup[7205]: 1E6EF41ADE: message-
id=<20250323211640.16FA741ADD@ubuntu.cs470.internal>
2025-03-23T21:16:40.127730+00:00 ubuntu postfix/local[7207]: 16FA741ADD:
to=<root@ubuntu.cs470.internal>, orig_to=<root>, relay=local, delay=0.05,
delays=0.03/0.01/0/0, dsn=2.0.0, status=sent (forwarded as 1E6EF41ADE)
2025-03-23T21:16:40.127894+00:00 ubuntu postfix/qmgr[6469]: 1E6EF41ADE:
from=<failsafe@ubuntu.cs470.internal>, size=574, nrcpt=1 (queue active)
2025-03-23T21:16:40.127969+00:00 ubuntu postfix/qmgr[6469]: 16FA741ADD: removed
2025-03-23T21:16:40.283466+00:00 ubuntu postfix/smtp[7208]: 1E6EF41ADE:
to=<peter@cs470.internal>, orig_to=<root>,
relay=freebsd.cs470.internal[10.42.77.72]:25, delay=0.16, delays=0/0.04/0.07/0.04,
dsn=2.0.0, status=sent (250 2.0.0 52NLGe37007152 Message accepted for delivery)
2025-03-23T21:16:40.283652+00:00 ubuntu postfix/qmgr[6469]: 1E6EF41ADE: removed
```

Note the second-to-last line above, which mentions that the mail server on my FreeBSD system accepted the mail for delivery. This is the desired outcome.

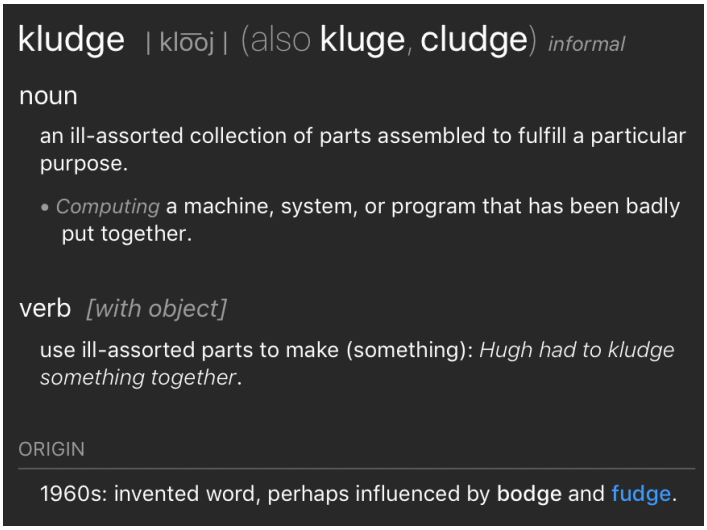
19. Read this step … but do not do this step.

!! AGAIN, READ BUT DO NOT DO THIS STEP, UNTIL THE START OF THE NEXT STEP. This step here is only here for your reading pleasure, to share past lessons learned in case you truly get stuck, to talk about SMTP smart hosts, and to illustrate a basic sysadmin concept and vocabulary word: "kludge."

In prior years, I was completely unable to get `postfix` to properly resolve the MX record to cleanly deliver diagnostic e-mails between VMs, and we had to resort to a "kludge."

Sometimes you need to send e-mail to the rest of the internet via another system, because network policy simply won't let you. Although I've thought about using this feature to get mail delivered from our little .internal domains, no such policy was in place in between our lab VMs. I just couldn't get `postfix` to behave.

In other words, often times, using an SMTP "smart host" isn't a kludge. It's necessary. However, in my case, it was definitely a kludge.



Just in case you hadn't been exposed to the word "kludge" before … please be introduced. Not the solution we want to roll out, but if holds things together until the end of the business day on Friday and for the weekend, we can fix it later, The Right Way™.

We're going to set up our FreeBSD VM as a "smart host." This means that our local host (in this case, our Ubuntu VM) doesn't know how to deliver mail, so it's going to use a system smarter than it … in this case, our FreeBSD VM, where we want the mail to go, anyways.

```
$ sudo vi /etc/postfix/main.cf
```

Under the section covering the option "relayhost," add the line …

```
relayhost = [freebsd.cs470.internal]
```

… and, after saving `main.cf`, tell `postfix` to reload its configuration …

```
$ sudo postfix reload
```

20. GnuPG. Fortunately, this is going to go way quicker than mail …

```
$ which gpg
```

```
/usr/bin/gpg
```

… because `gpg` is already installed. The package subsystem uses it to digitally sign packages. Wahoo, shortest step in a lab, EVAR.

21. Updates. Let's set up root's crontab to check for updates every night at midnight …

```
$ sudo crontab -e
```

… and don't even ***think*** of using `nano` as your editor when you are prompted. Add the following line …

```
0 0 * * * apt update && apt list --upgradable
```

… I'm hoping you can tell what this does, at this point. If not, look it up!

part three: configuring LDAP login

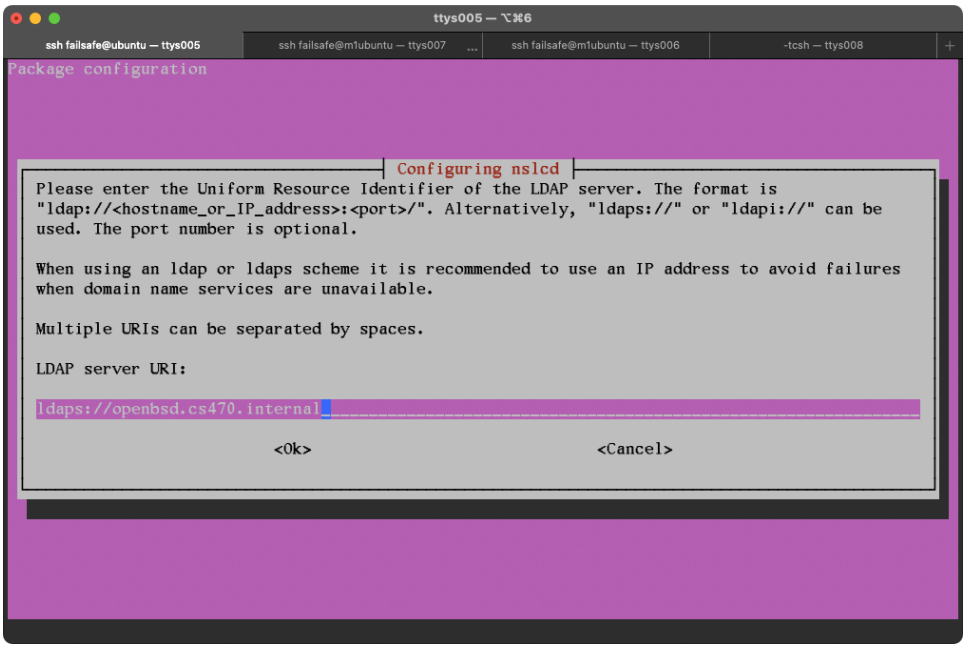
Now let's get our LDAP directory plugged into our new instance.

!! THE WARNINGS FROM LABS 1B AND 2 ABOUT LOCKING YOURSELF OUT OF YOUR VM APPLY AGAIN HERE.

22. Let's install the software required for LDAP integration.

```
$ sudo apt -y install libnss-ldapd libpam-ldap ldap-utils
```

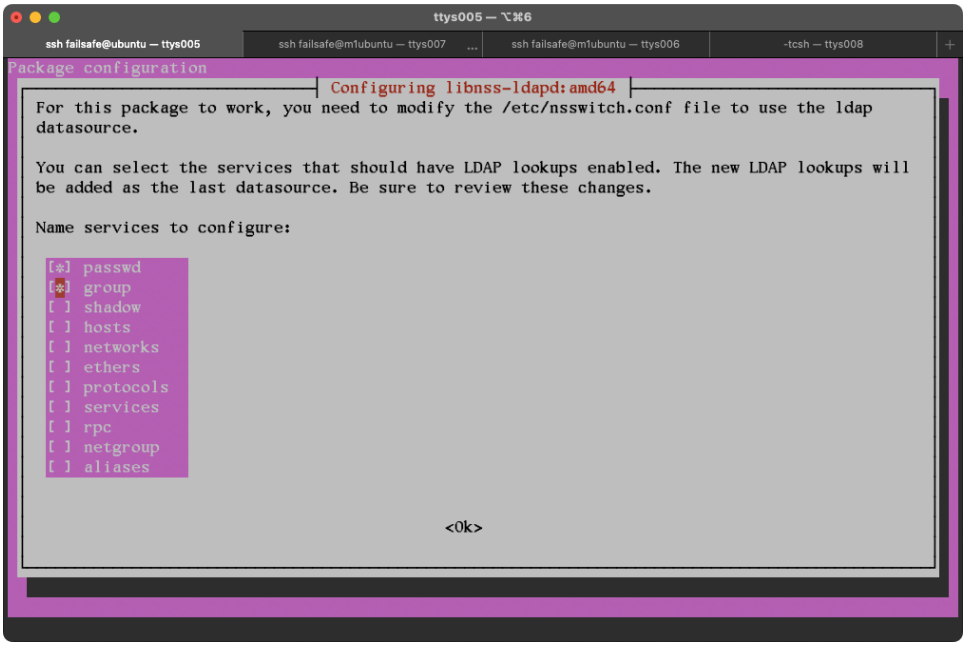
You'll be prompted by the now-familiar package manager dialog asking you how you'd like your LDAP integration set up, first for `nslcd`. First, the URI, `ldaps://openbsd.cs470.internal` just like the configuration of our other LDAP clients.



Then, of course, provide the search base `dc=cs470,dc=internal` if it's not already in there.

If asked to set a policy for certificate validation for `nslcd`, choose "demand."

When asked to choose which name services to tie to LDAP, choose users (`passwd`) and groups (`group`).



You're also going to be asked to set the same options again for `ldap-auth-config`.

When asked which version of the LDAP protocol to use, choose version 3.

Choose `<Yes>` when asked whether to make local root a database admin.

The LDAP database does not require login to search it.

When asked for the LDAP account to be used by root, provide the admin DN from lab 1b, and the directory admin password on the following screen.

You may be prompted for which services to restart next. Since no other systems or services are counting on this Ubuntu VM yet, go ahead and allow it to restart whichever services it identifies as candidates on your instance, if asked.

Ubuntu's package configuration automation has largely removed the need to edit configuration files; our Ubuntu VM's LDAP client is now aimed at our OpenBSD VM's LDAP server unless you made any mistakes during data entry. If you need to do any troubleshooting for this step, you can look at the files `/etc/ldap.conf` and `/etc/nslcd.conf` … or consult the `man` pages and get them to cough up how to reconfigure a package.

23. As usual, though, we need to pull over our CA certificate from the OpenBSD VM for the LDAP client.

```
$ scp -p openbsd:/home/pki/data/cacert.pem ~failsafe
```

Ubuntu adds certificates to its trust store in a different way than other operating systems. If you look in the familiar `/etc/ssl` directory with `ls -l`, you'll see there's no `cert.pem` but inside the certs directory, you'll find soft links to CA certificates stored elsewhere on the filesystem.

Ubuntu has a whole tool suite for managing the trust store, and you can find more details here: https://ubuntu.com/server/docs/security-trust-store

Since our certificate is already in PEM (text-encoded) format, we just need to move it to the correct directory and end it with `.crt` …

```
$ sudo mv ~failsafe/cacert.pem /usr/local/share/ca-certificates/cs470.internal.crt
```

… oops … `mv` wasn't able to move it … with `sudo` … why not, and why does the following work instead? You need to understand this.

```
$ mv ~failsafe/cacert.pem /tmp
$ sudo cp -p /tmp/cacert.pem /etc/ssl/cacert.pem
```

Note: we're also saving another copy of the root CA certificate to a familiar place, above, while we move it to Ubuntu's preferred place, below.

```
$ sudo mv /tmp/cacert.pem /usr/local/share/ca-certificates/cs470.internal.crt
$ ls -la /usr/local/share/ca-certificates/
total 1
drwxr-xr-x 2 root     root     4096 Mar 23 22:08 .
drwxr-xr-x 7 root     root     4096 Mar 12 04:53 ..
-rw-rw-r-- 1 failsafe failsafe 7091 Feb  8 20:51 cs470.internal.crt
```

Let's hope it being owned by `failsafe` isn't a problem, and tell Ubuntu to re-parse its cache of CA certificates.

```
$ sudo update-ca-certificates
```

Both my Ubuntu VMs said `1 added` as a part of that command's output … not a problem. Fix it anyways!

```
$ sudo chown root:root /etc/ssl/cacert.pem
$ sudo chown root:root /usr/local/share/ca-certificates/cs470.internal.crt
$ sudo chmod 644 /etc/ssl/cacert.pem /usr/local/share/ca-certificates/cs470.internal.crt
```

Now our Ubuntu VM's local LDAP client should be able to trust the certificate handed to it by the LDAP server on our OpenBSD VMs, and thus trust the service too.

24. Set up `/etc/ldap/ldap.conf` as is appropriate for your LDAP server. You've seen this file, the OpenLDAP client configuration file, before. You know what to do here. Do it.

25. Check the file `/etc/nsswitch.conf` on your new Ubuntu VM, and make sure `ldap` is at the end of the `passwd` and `group` lines, and test.

    At this point in time, your LDAP user should resolve with `id` and the output of `ls -l /home` should show names, not numbers, for all users and groups associated with each folder.

26. Ubuntu, like upstream Debian, uses a group called "sudo" to grant `sudo` rights; the installer added our failsafe user to this group. Add your LDAP user to the `sudo` group in `/etc/group` too.

27. Create a symbolic link for `bash`.

    ```
    $ sudo ln -s /bin/bash /usr/local/bin/bash
    ```

    … and add `/usr/local/bin/bash` to the list of acceptable shells in `/etc/shells`.

28. Test LDAP authentication!

    Test logging into the console of your VM, in its VMware window, as both your failsafe local user and your LDAP user. You can disable key-based authentication at the client side by doing this …

    ```
    $ ssh -o PreferredAuthentications=password -o PubkeyAuthentication=no peter@ubuntu
    ```

    Try logging in via SSH as both users, with passwords and with SSH keys. Make sure both authentication methods work, for both users.

    Test using `sudo` as your LDAP user. Note, again, that you can just use `sudo` to run `ls` to test that it works, even though you don't *need* rootly powers to run `ls`.

    Go ahead and add a line to `/etc/sudoers` for your LDAP sudoers group too.

    Once you're sure it works, as usual, abandon the failsafe account. It's only to fix things when needed.

part four: web server, containerized with Docker

We already talked about what we're doing here and why, in plenty of detail during the preamble for this lab, so let's jump right in.

29. First, let's install `docker` and its container authoring tool, `docker-compose`.

    ```
    $ sudo apt install docker.io docker-compose-v2
    ```

    Note that as a part of the truncated output below, `docker` is being set up with a sandbox group account, and is being registered with `systemd` as a service.

    ```
    Setting up docker.io (26.1.3-0ubuntu1~24.04.1) ...
    info: Selecting GID from range 100 to 999 ...
    ```

```
info: Adding group `docker' (GID 114) ...
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service →
/usr/lib/systemd/system/docker.service.
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket →
/usr/lib/systemd/system/docker.socket.
```

`docker` itself can be used to grab stock application containers from the Docker Hub (https://hub.docker.com), and from arbitrary repositories. The web server we'll be using in this lab, `nginx`, is one of, if not **the** most popular out there. As such, it's available as a container from the default repository hosted by the Docker project itself.

Now let's grab the `docker` container we're actually going to be running.

```
$ sudo docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
6e909acdb790: Pull complete
5eaa34f5b9c2: Pull complete
417c4bccf534: Pull complete
e7e0ca015e55: Pull complete
373fe654e984: Pull complete
97f5c0f51d43: Pull complete
c22eb46e871a: Pull complete
Digest: sha256:124b44bfc9ccd1f3cedf4b592d4d1e8bddb78b51ec2ed5056c52d3692baebc19
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

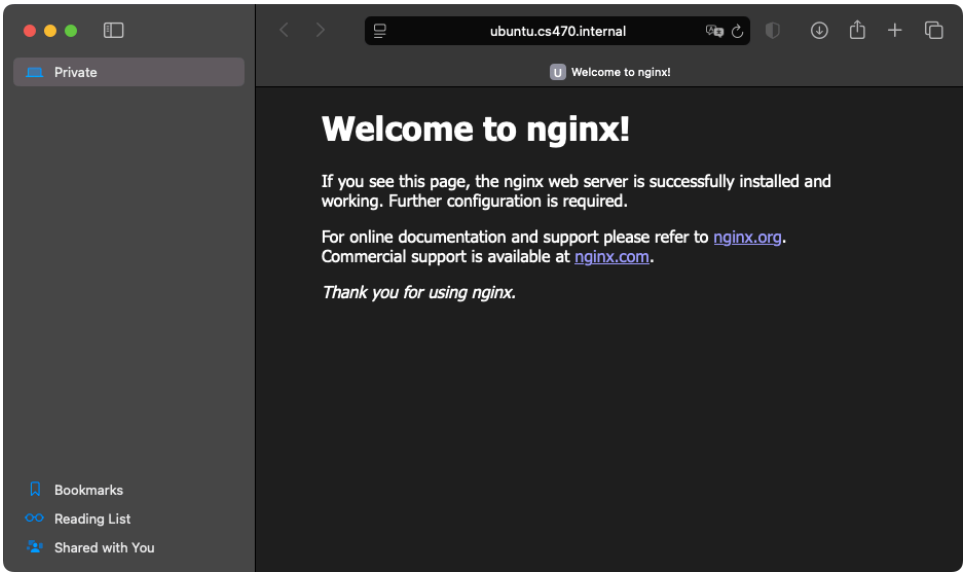
30. Let's fire up the container. In the command below, `-p 80:80` tells `docker` to map port 80 of our host system (the Ubuntu VM) to port 80 inside the container. This allows the `nginx` service *inside* the container to answer HTTP requests on TCP port 80 destined for our Ubuntu VM's primary network interface and IP address.

    ```
    $ sudo docker run -p 80:80 -d nginx
    ```

    If you (and I) did everything correctly to this point, the `docker` command will return a long hexadeximal identifier for the container you just started up, and you should be able to go to the following URL …

    http://ubuntu.cs470.internal/

    … and when it loads it, you should see a webpage welcoming you to the `nginx` web server.



You have a lightweight VM inside a VM now, and that VM-in-a-VM is a web server. Pretty cool, huh?

31. Using `sudo` every time we want to run `docker` commands will get old; if you add `failsafe` and your LDAP accounts to the `docker` group, you won't have to use `sudo` anymore. Remember, you have to log out and then back in to get new group IDs attached to your security context.

    If you try to use `docker` without `sudo` before logging out and back in, you'll see an error message saying permission was denied to `/var/run/docker.sock` … filesystem permissions (use `ls -l`) are how this is enforced.

32. In order to see all the containers you have running, the `docker` command has a complete syntax of subcommands underneath it, and `docker ps` will, like the command `ps`, show you a list of running things.

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS
NAMES
2abf8e2072df   nginx     "/docker-entrypoint.…"   24 minutes ago   Up 24 minutes
0.0.0.0:80->80/tcp, :::80->80/tcp   stoic_bouman
```

    Note that `docker ps` gives you not only the important handle for the container, in a shorter form of its container ID, but shows ports mapped into the containers.

33. Let's stop the container now, so that we can do some reconfiguration …

```
$ docker stop 2abf8e2072df
```

    … it will echo back the container ID, presumably because you can provide the `docker` command multiple container IDs on a single line, if you're doing the same operation (stop/start/whatever).

    Also note that now, if you reload that page in your browser, it will either throw an error right away, or

time out and then throw an error. The web service is no longer running, not answering anymore.

34. In order to bend the configuration of `nginx` to meet our needs, we'll need a copy of its configuration files. The easiest way to get those configuration files? Let's grab the `nginx` package for Ubuntu.

```
$ sudo apt install nginx
```

As we've seen before, installing service packages often enables and starts them, so let's make sure to stop and disable the copy of `nginx` we just installed on our Ubuntu VM.

```
$ sudo systemctl stop nginx
$ sudo systemctl disable nginx
```

35. Next let's make a certificate for our new webserver. All web traffic, unless we **know** all content is not sensitive.

As before, the first thing we want to do is to import our organizational settings for our key infrastructure. Copy the OpenSSL configuration file over to this system from our OpenBSD VM.

```
$ scp -p openbsd:/etc/ssl/openssl.cnf /tmp/openssl.cnf
```

Just like we last did with FreeBSD, move Ubuntu's default OpenSSL configuration file out of the way.

```
$ sudo mv /etc/ssl/openssl.cnf /etc/ssl/openssl.cnf.orig
```

Now, move the file from OpenBSD into its place, and fix permissions just like before.

```
$ sudo mv /tmp/openssl.cnf /etc/ssl/openssl.cnf
$ sudo chown root:root /etc/ssl/openssl.cnf
```

Finally, also just like we did in lab 2, edit the file, and change the hostname at the very end to `ubuntu.cs470.internal`.

Make a directory for storing encryption key material … note: we're leaving this group-read/executable for a reason.

```
$ sudo mkdir -m 750 /etc/nginx/ssl
```

… generate a key …

```
$ sudo openssl genrsa -out /etc/nginx/ssl/ubuntu.key 3072
```

… and then generate a certificate request.

```
$ sudo openssl req -new -key /etc/nginx/ssl/ubuntu.key -out /etc/nginx/ssl/ubuntu.csr
```

You should recognize the next steps … no locality name. For the common name, use `ubuntu.cs470.internal` and `root@cs470.internal` as the e-mail address. Once the CSR is done,

copy it over to OpenBSD for the CA to sign it. We do the two-step dance below because using `ssh` as root is frowned on for security reasons – don't run around as root if you don't have to.

```
$ sudo cp -p /etc/nginx/ssl/ubuntu.csr /tmp/ubuntu.csr
$ scp -p /tmp/ubuntu.csr openbsd:/home/pki/newreq.pem
```

Log back into OpenBSD as `failsafe` and sign the CSR.

```
$ cd /home/pki && CA.pl -sign
```

As always, double-check all the information provided before typing `y` to sign the certificate … especially the hostname. Then copy it back to Ubuntu.

```
$ scp -p newcert.pem ubuntu:/tmp/ubuntu.pem
```

Again, clean up the new cert and CSR here on OpenBSD. Then back on Ubuntu, remove the CSR from `/tmp` and slide the certificate into place …

```
$ sudo mv /tmp/ubuntu.pem /etc/nginx/ssl/ubuntu.pem
```

Set its permissions properly. Note: Red Hat and its derivatives use the BSD-derived `wheel` group here. Ubuntu doesn't have a direct equivalent. The `sudo` or `adm` groups would be the closest thing, if we were trying to keep system administrators in mind. Here, we're not. Encryption keys should belong to root and root alone.

```
$ sudo chown root:root /etc/nginx/ssl/ubuntu.pem
```

36. Let's configure the web server.

    Take a moment and check out the tree of configuration files under `/etc/nginx`.

```
$ ls -l /etc/nginx/
total 1
drwxr-xr-x 2 root root 4096 Sep 10  2024 conf.d
-rw-r--r-- 1 root root 1125 Nov 30  2023 fastcgi.conf
-rw-r--r-- 1 root root 1055 Nov 30  2023 fastcgi_params
-rw-r--r-- 1 root root 2837 Nov 30  2023 koi-utf
-rw-r--r-- 1 root root 2223 Nov 30  2023 koi-win
-rw-r--r-- 1 root root 5465 Nov 30  2023 mime.types
drwxr-xr-x 2 root root 4096 Sep 10  2024 modules-available
drwxr-xr-x 2 root root 4096 Sep 10  2024 modules-enabled
-rw-r--r-- 1 root root 1446 Nov 30  2023 nginx.conf
-rw-r--r-- 1 root root  180 Nov 30  2023 proxy_params
-rw-r--r-- 1 root root  636 Nov 30  2023 scgi_params
drwxr-xr-x 2 root root 4096 Mar 24 18:20 sites-available
drwxr-xr-x 2 root root 4096 Mar 24 18:20 sites-enabled
drwxr-xr-x 2 root root 4096 Mar 24 18:20 snippets
drwxr-x--- 2 root root 4096 Mar 25 03:22 ssl
-rw-r--r-- 1 root root  664 Nov 30  2023 uwsgi_params
-rw-r--r-- 1 root root 3071 Nov 30  2023 win-utf
```

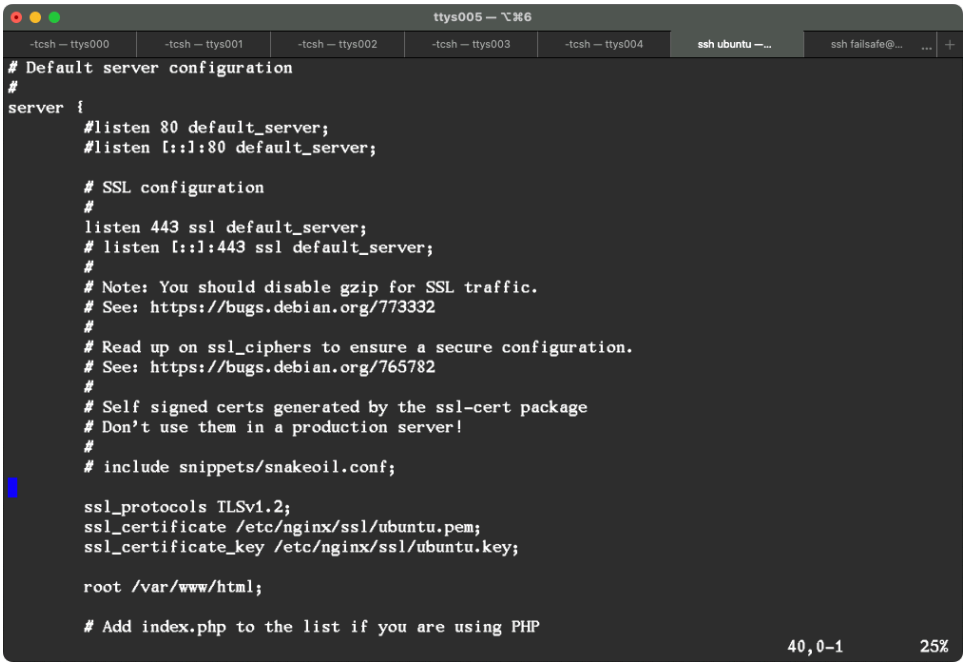
    There's a `conf.d` directory for modular configuration files. There are a few files for CGI (Common Gateway Interface, a way for external code to interact with web sites). There are a pair of directories

each for web server modules and for web sites, one each for “available” sites and modules, and one each for “enabled” sites and modules. The idea here is that all sites and modules are in the “available” directories, and you symlink to files in the “enabled” directories with those modules and sites you want enabled. The master `nginx.conf` file refers to files under `conf.d` and the two “enabled” folders.

To customize the configuration of `nginx`, the file we’re after is `/etc/nginx/sites-enabled/default`. First, let’s also add our certificate into the configuration … comment out both the first `listen` line for port 80 as well as the second `listen` line that listens for port 80 on IPv6 (with the colons). Then, uncomment the listen line for port 443, but NOT the second line that listens for port 443 on IPv6. Just after that stanza, add the following:

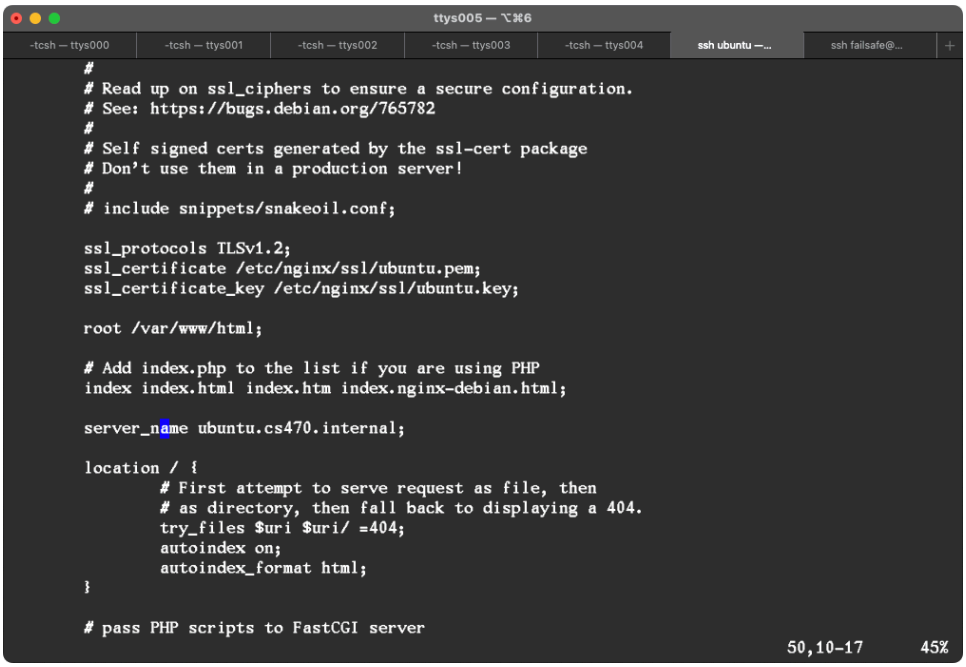
```
ssl_protocols TLSv1.2;
ssl_certificate /etc/nginx/ssl/ubuntu.pem;
ssl_certificate_key /etc/nginx/ssl/ubuntu.key;
```

The first part of the file should look like the screenshot below. Note the option `root`; this denotes the root of documents to be served up by the file server, and will be important later.



Go down a little within the file and change `server_name` to `ubuntu.cs470.internal`.

We want to add the option to autoindex folders to the root location of the web server and to give them proper download links by HTML-formatting the folder for our web browser. After you make these changes, you can save out the file. The second part of the edits are in the screenshot below.



37. In this step, we lay out all the content we expect our containerized web server to display. To the container, it's all underneath the folder `/var/www/html`, the default location for web content under `nginx`. Outside the container, we're going to map host directories to that path inside the container, and we're going to continue to use the the `/srv` tree for local service data.

    Make a copy of the default nginx website to `/srv/www`.

    ```
    $ sudo cp -pr /var/www/html /srv/www
    ```

    It's very common to have a group for people with permissions to change web content; Ubuntu has one built-in …

    ```
    $ grep www /etc/group
    www-data:x:33:
    ```

    … many OSs make this group ID 80, to match the default port 80 for HTTP. Ubuntu strangely doesn't configure the web root to be group readable …

    ```
    $ ls -ld /var/www/html /srv/www
    drwxr-xr-x 2 root root 4096 Mar 24 18:20 /srv/www
    drwxr-xr-x 2 root root 4096 Mar 24 18:20 /var/www/html
    ```

    … so let's fix that on our newly-minted web root directory, and make it group-writable. Here, you can see another convenience of `/srv` … `/srv/www` is far easier than `/var/www/html`, and doesn't create a need for web developers to go into `/var` if you don't want them in there.

    ```
    $ sudo chgrp www-data /srv/www
    $ sudo chmod g+w /srv/www
    ```

Add your LDAP user to the www-data group.

In lab three, we created `/srv/nfs` with wide-open permissions so that any user on any of our NFS clients (every VM except our OpenBSD name server) could use the disk space on our file server. Not only are we going to make the default `nginx` website available via the web server, we're also going to use `nginx` to make the content in `/srv/nfs` accessible via the web server. Make a mount point directory for that content. If you log out and back in quickly, you shouldn't have to use `sudo`.

```
$ mkdir /srv/www/nfs
```

With all that in place, we're going to restart the container, this time with encrypted HTTP (HTTPS) available on the standard port 443 instead of cleartext HTTP on port 80, with our host Ubuntu system's `/etc/nginx` in place of the container's default `nginx` configuration directory, `/srv/www` in place of `/var/www/html`, and `/srv/nfs` at `/var/www/html/nfs`.

```
$ docker run -p 443:443 -v /etc/nginx:/etc/nginx:ro -v /srv/www:/var/www/html:ro -v
/srv/nfs:/var/www/html/nfs:ro -d nginx
```

If you did everything correctly, you should now be able to load the website in your browser using HTTPS now – https://ubuntu.cs470.internal/ – and see the default nginx welcome page again. If you add go to https://ubuntu.cs470.internal/nfs you should see the list of files in `/srv/nfs` … listed in your browser. If this didn't work, try removing the `-d` flag and looking at the output.

!! If you get a certificate warning here in your browser, you messed something up or missed a step.

Make some files in `/srv/nfs` and refresh your web browser tab or window.

We'll be using this again shortly, but take a step back and look at what we're doing here … we're using Docker on Ubuntu and `nginx` to serve up files from our Rocky VM to web browser clients, over an encrypted channel with the trust chain created on our OpenBSD VM. We're starting to pull it all together here.

Pat yourself on the back, and then stop your `nginx` container.

38. Now let's make sure our Docker containers start when our Ubuntu VM starts. First, we need to enable `docker` as a service:

```
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

Next, let's create our containers the right way using `docker-compose`. `docker-compose` (note the hyphen) used to be a standalone program, but it has since been integrated as a subcommand of `docker` (hence the removal of the hyphen). It allows us to store a container configuration – even configurations for multiple containers – as a YAML file that will make more complicated configurations easier to troubleshoot. Use `sudo` and `vi` to create and edit a file `/etc/docker/nginx.yml` with the following contents:

```
version: '3.1'

services:
  nginx:
    image: nginx
    container_name: nginx
    restart: always
    networks:
      - cs470
    ports:
      - 443:443
    volumes:
      - /etc/nginx:/etc/nginx:ro
      - /srv/www:/var/www/html:ro
      - /srv/nfs:/var/www/html/nfs:ro

networks:
  cs470:
```

Heads up: YAML files are very rigid about spacing and indentation, all the more reason for monospaced fonts. If you have problems here, that'll likely be the cause. We'll be playing with them more in lab 5.

Next, we need to create a custom `systemctl` service unit to start up our `nginx` container, at `/etc/systemd/system/docker-nginx.service` …

```
[Unit]
Description=nginx docker container
Requires=docker.service
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker compose -f /etc/docker/nginx.yml up
ExecStop=/usr/bin/docker compose -f /etc/docker/nginx.yml down

[Install]
WantedBy=default.target
```

Now, we need to enable and start our new custom service unit.

```
$ sudo systemctl enable docker-nginx.service
$ sudo systemctl start docker-nginx.service
```

Check the status of the service to confirm it is running as a service, and use `docker` to confirm the details of the container.

```
$ systemctl status docker-nginx.service
$ docker ps
```

Restart to test out the configuration completely, making sure your container starts automatically after a reboot. Your web browser should trust the web server's certificate, or you missed something.

If you need to troubleshoot, remember your old friend `netstat` … it should show that you have an HTTPS service or a listener on port 443.

The command `docker inspect` will require you to provide a container ID, but will give you extremely rich information about how the container is configured.

The command `docker exec -it <ID> bash` will attempt to run a shell inside the container, and give you interactive access to the inside of the container … replace `<ID>` with the ID of your container. Keep in mind, a container is intended to be light, so a lot of commands will be missing, but you might be able to gain valuable insight as to what's going on, or going wrong, inside your container.

part five: Vaultwarden, containerized with Docker, plus `nginx` reverse proxy setup

Now we're going to set up an instance of Vaultwarden, an unofficial (but API-compatible) free and open-source re-implementation of Bitwarden. Bitwarden is a popular password manager service, and along with Vaultwarden, these are good choices for useful services to get running on your actual home network in your free time.

In a prior iteration of this lab, I went through documenting the Bitwarden installation, only to find at the very end that it doesn't support the ARM architecture. Oddly enough, Bitwarden uses Microsoft SQL Server … for Linux. No matter how much time has passed, it's still just really bizarre to see "Microsoft" next to "Linux," let alone in the name of a product, so I've left this here from my journey putting together these labs. This is from `/opt/bitwarden/logs/mssql/errorlog` …

```
2024-06-29 19:02:50.86 Server  Microsoft SQL Server 2022 (RTM-CU12) (KB5033663) - 16.0.4115.5 (X64)
       Mar  4 2024 08:56:10
       Copyright (C) 2022 Microsoft Corporation
       Express Edition (64-bit) on Linux (Ubuntu 22.04.4 LTS) <X64>
2024-06-29 19:02:50.87 Server  UTC adjustment: 0:00
2024-06-29 19:02:50.88 Server  (c) Microsoft Corporation.
2024-06-29 19:02:50.88 Server  All rights reserved.
2024-06-29 19:02:50.88 Server  Server process ID is 356.
```

… apparently, the reason Bitwarden won't work on ARM is because of this dependency, and that Microsoft SQL Server for Linux doesn't yet support ARM. It's for this reason that we moved to Vaultwarden.

Most of the directions here are taken directly from Vaultwarden's wiki on their GitHub.

39. Create the directory where we'll be storing Vaultwarden's data.

    ```
    $ sudo mkdir -m 750 /opt/vaultwarden
    ```

    Vaultwarden graciously provides the template of a `docker compose` file with the bare minimum of environment variables and configuration that Vaultwarden needs to run. Let's save this to `/etc/docker/vaultwarden.yml` like we did with `nginx`.

    ```
    services:
      vaultwarden:
        image: vaultwarden/server:latest
        container_name: vaultwarden
        restart: always
        networks:
          - cs470
    ```

```
    environment:
      SIGNUPS_ALLOWED: "true"
    volumes:
      - /opt/vaultwarden:/data
    ports:
      - 12345:80

networks:
  cs470:
```

Take note of the `networks:` lines telling this container to run on the `cs470` container network – the `nginx.yml` file we set up for the primary webserver on this system has the same setup. In a few steps, we'll want `nginx` and `vaultwarden` containers to be able talk to each other. In general, to enable inter-container network communication, either two containers must be sharing their hosts network stack and communicating over that, or they must be part of the same "container network."

```
$ docker network ls
NETWORK ID     NAME             DRIVER    SCOPE
8b8fba68e4d5   bridge           bridge    local
a30ef6948e8f   docker_cs470     bridge    local
eb6dfdca03bb   host             host      local
3a573f8d4c68   none             null      local
```

In the context of Docker, a bridge network on our Ubuntu VM is like the VM network on your host. Containers are assigned IPs, and the host machine (here, our Ubuntu VM) acts as the network's gateway. Something else that's neat about container networks is that containers can reference each other on the network by their names. Here's a quick showcase of that with a busybox container…

```
$ docker run --network=docker_cs470 -it busybox

# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue 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
90: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:16:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.22.0.4/16 brd 172.22.255.255 scope global eth0
       valid_lft forever preferred_lft forever

# ping nginx
PING nginx (172.22.0.2): 56 data bytes
64 bytes from 172.22.0.2: seq=0 ttl=64 time=0.651 ms
^C
^D
```

40. We're able to bring up the container straight away.

```
$ docker compose -f /etc/docker/vaultwarden.yml up

...

/-------------------------------------------------------------------\
|                       Starting Vaultwarden                        |
|                          Version 1.33.2                           |
|-------------------------------------------------------------------|
| This is an *unofficial* Bitwarden implementation, DO NOT use the  |
| official channels to report bugs/features, regardless of client.  |
| Send usage/configuration questions or feature requests to:        |
```

```
|   https://github.com/dani-garcia/vaultwarden/discussions or        |
|   https://vaultwarden.discourse.group/                             |
| Report suspected bugs/issues in the software itself at:            |
|   https://github.com/dani-garcia/vaultwarden/issues/new            |
\--------------------------------------------------------------------/

[vaultwarden::auth][INFO] Private key 'data/rsa_key.pem' created correctly
[start][INFO] Rocket has launched from http://0.0.0.0:80
```

Vaultwarden automatically creates its own private RSA key for all the crypto operations it needs to perform, before launching a Rocket web server listening on all interfaces on port 80. The wildcard address and port number is only from the perspective of the container. Outside the container, the port we want to use is 12345, because of the `12345:80` port mapping in `vaultwarden.yml`.

Navigate to http://ubuntu.cs470.internal:12345, et voilà!

We're nowhere near done, though. If you noticed "http" in that URL, the scary red lock icon in the URL bar, or the lack of a multi-step certificate copying-and-signing routine, then you'd know you're on a totally unencrypted connection. In fact, if you try to use Vaultwarden in this unsecure (HTTP) context, your web browser should straight-up block the use of JavaScript cryptography APIs, and rightfully so.

`control-c` out of the container in the terminal you started it in, if you haven't already.

41. As you saw below its startup banner, Vaultwarden uses Rocket, a Rust web framework HTTP server. We could generate a new cert for Vaultwarden to use, but since we already made a cert for this hostname for `nginx`, let's just reuse it and show off `nginx`'s reverse proxy feature.

    Fundamentally, forward and reverse proxies are just systems that sit between some network and the wider Internet, handling the traffic in between.

    Forward proxies forward traffic from a network to the outside Internet. You might be familiar with this use case: if you wanted to connect to some system online but were concerned about them seeing who you are, you could *proxy* your traffic through, well, a forward proxy, and they would see the proxy's IP instead of yours in their logs. If you want to examine all traffic heading outbound in order to contain potential data leakages, a forward proxy is also a great way to handle that particular problem too.

    A *reverse* proxy sits at the public-facing side of a network, relaying traffic from clients on the Internet to the inside of the network. A business might have separate servers for HTTP, FTP, and email – a reverse proxy would let clients access any of these on the same hostname, like someone at a front desk giving directions to different places in a building. This is great for simplifying load balancing, as well as handling transport-layer encryption (TLS/SSL) without needing to issue a new certificate for each server.

    Our goal state is simple: a public-facing `nginx` server configured to set up SSL connections and pass requests for Vaultwarden, to Vaultwarden. We'd like `nginx` to keep serving our NFS share directory on port 443, and listen on another port, say 8443, for Vaultwarden traffic.

Edit `/etc/docker/nginx.yml` and add another port mapping from 8443 on the VM to 8443 in the container.

Because we'll be using `nginx` to proxy connections to Vaultwarden, we *don't* want any ports in our Vaultwarden container to be exposed to our Ubuntu VM anymore. Go ahead and edit `/etc/docker/vaultwarden.yml` and comment out or delete the port mapping list.

42. We're going to set up the reverse proxy behavior for `nginx` now, so open up `/etc/nginx/sites-enabled/default` again.

    For some communications, Vaultwarden uses WebSocket, a two-way protocol, which web clients can switch to from HTTP with the `Upgrade` header. However, `Upgrade` and `Connection` HTTP headers are "hop-by-hop," meaning they're passed to the proxy, but not the next hop, the Vaultwarden server. The proxy needs to explicitly reconstruct those two headers as it passes the request along.

    Add the following `map` block below the default `server` block.

```
map $http_upgrade $connection_upgrade {
        default upgrade;
        ''      "";
}
```

    This defines a variable called `connection_upgrade`, based on the value of the client's `http_upgrade` header. If the `Upgrade` header is empty, then we don't need to define a `Connection` header, but if `http_upgrade` has any value, then `connection_upgrade` is set to "upgrade."

    We don't have the time, space, or drive to fully dissect HTTP here and now, so if you're curious about what the heck that all means, here's some light reading.

    https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
    https://nginx.org/en/docs/http/websocket.html

43. Now we can define the proxy server by placing the following `server` block below the `map` block.

    The request proxying lines look more complicated than they really are. When a client connects to port 8443 and hits `/`, `nginx` simply takes their request, sets up relevant HTTP headers, and passes it to our Vaultwarden server's hostname and port. Since our `nginx` and `vaultwarden` containers are on the same `cs470` container network, they can resolve each other by name, as demonstrated a few steps earlier.

```
server {
        listen 8443 ssl;
        server_name ubuntu.cs470.internal;
        resolver 127.0.0.11;

        ssl_protocols TLSv1.2;
        ssl_certificate /etc/nginx/ssl/ubuntu.pem;
        ssl_certificate_key /etc/nginx/ssl/ubuntu.key;
```

```
        client_max_body_size 525M;

        location / {
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection $connection_upgrade;

                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header X-Forwarded-Proto $scheme;

                set $proxy_vaultwarden vaultwarden:80;
                proxy_pass http://$proxy_vaultwarden;
        }
}
```

Though each container's `/etc/resolv.conf` file is already set to use the Docker daemon itself (addressable on the loopback adapter at 127.0.0.11) for DNS, `nginx` is unable to do lookups on other containers unless we explicitly specify a `resolver` for some reason.

44. Create a custom `systemd` service in `/etc/systemd/system/docker-vaultwarden.service` like you did with `nginx`. Since they're so similar, let's just …

    ```
    $ sudo cp /etc/systemd/system/docker-nginx.service /etc/systemd/system/docker-vaultwarden.service
    ```

    … and use `vi` to replace any mention of `nginx` with `vaultwarden`.

    Then, open `/etc/systemd/system/docker-nginx.service` in `vi` to change some `systemd` service settings. Append `docker-vaultwarden.service` to the end of the `After=` line. Below the `Requires=` line, add the following line to set a soft dependency on Vaultwarden. We don't want `nginx` to crash if something's wrong with Vaultwarden.

    ```
    Wants=docker-vaultwarden.service
    ```

45. Reload `systemd`, recreating the service dependency tree to lock in our changes to `docker-nginx.service`, then enable the `docker-vaultwarden` service and restart everything.

    ```
    $ sudo systemctl daemon-reload
    $ sudo systemctl enable docker-vaultwarden
    $ sudo systemctl restart docker-nginx
    $ sudo systemctl restart docker-vaultwarden
    ```

    You should now be able to pull up the Vaultwarden web interface at https://ubuntu.cs470.internal:8443/. Make an account with your LDAP user's email address, and start storing passwords!

    Since we're all alone on our little network, after you make an account for yourself, edit `vaultwarden.yml` again and set the environment variable `SIGNUPS_ALLOWED` to `"false"`

If you suspect an error with `docker compose`, you can see `systemd` service startup logs with:

```
$ journalctl -x -u <service>
```

If you get a "502 Bad Gateway" error in your browser, then something's preventing `nginx` from proxying the request to Vaultwarden. Try the `docker logs -f` command, or getting a shell inside the `nginx` or `vaultwarden` containers.

part six: container management with Portainer

46. Wouldn't it be nice – *the best and perhaps only happy song ever written about **not** getting what you want* – if there was a way to easily manage all the containers you install for `docker`? We're going to install an open-source container manager called `portainer` that allows you to do just that.

    First, let's create a volume for `portainer` to separately store persistent state data ...

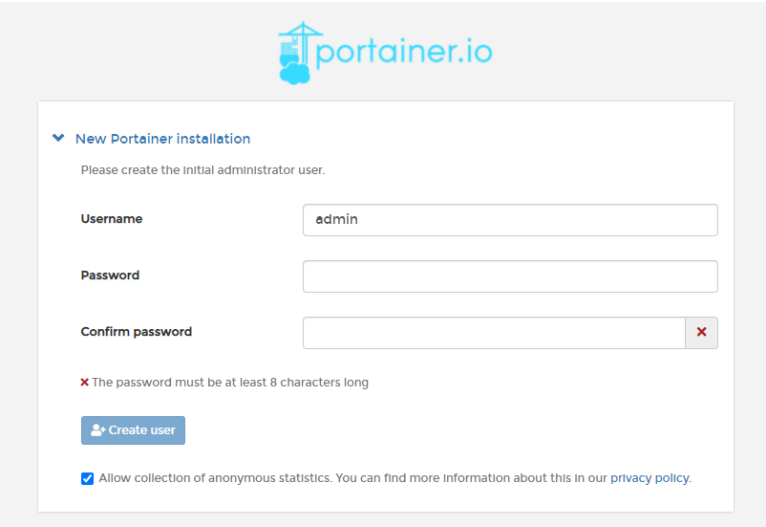
    ```
    $ docker volume create portainer_data
    ```

    ... and then tell `docker` to run `portainer`. What?! It's not been downloaded yet ...

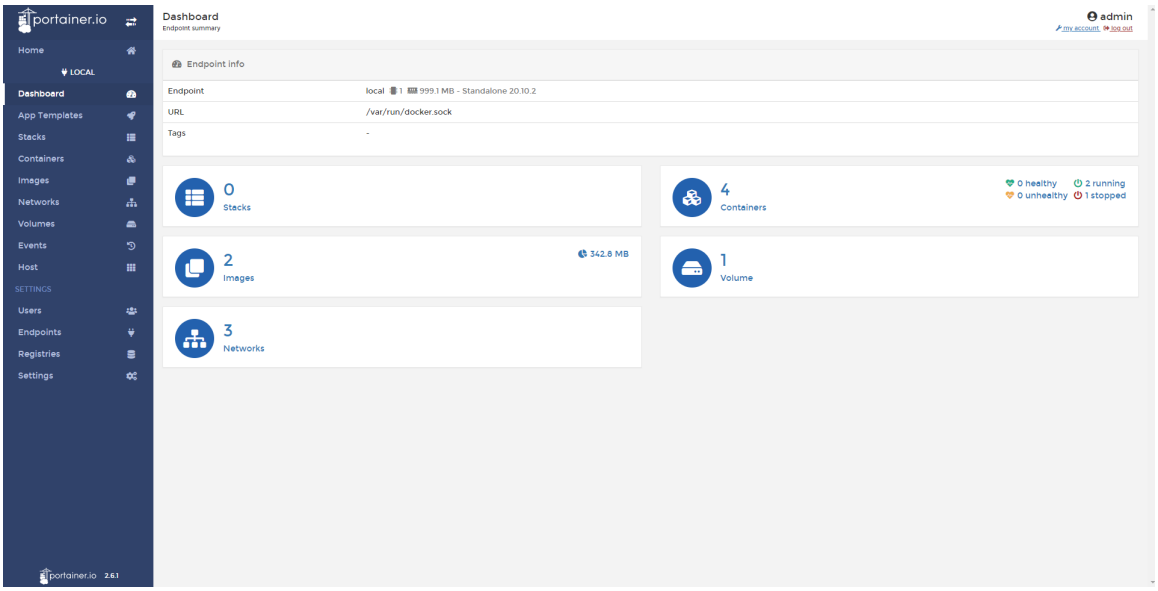
    ```
    $ docker run -d -p 8000:8000 -p 9000:9000 --name=portainer --restart=always -v
    /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data
    portainer/portainer-ce
    ```

    ... and of course, `docker` has an answer for that. Realizing it doesn't have `portainer`, it just pulls it.

47. In a web browser on your host operating system, go to http://ubuntu.cs470.internal:9000/ and you should be prompted to choose a username and password for the admin user. Note that we're using port 9000 for this web service.



48. Select to add an environment, then Docker as your container environment, and you should be able to go to the local environment to see an interface that looks like this.



You can do a lot in this web UI. You can add images, create new containers, stop or resume containers, look at container logs, and much more. This can be pretty useful when you have a lot of containers running.

49. Now, do the same with this container as with your `nginx` container, placing the YAML file in `/etc/docker/portainer.yml`, and creating a custom `systemd` service in `/etc/systemd/system/docker-portainer.service` … for your `portainer.yml` file, you will need to make some edits compared to last time, adding in a network and specifying your named volumes:

```
version: '3.1'

services:
  portainer:
    image: portainer/portainer-ce
    container_name: portainer
    restart: always
    networks:
      - cs470
    ports:
      - 8000:8000
      - 9000:9000
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - portainer_data:/data

networks:
  cs470:

volumes:
  portainer_data:
    name: portainer_data
```

Create a `systemd` unit file for `portainer` as well, just like before with `nginx`, and test it and make sure `portainer` starts up with your Ubuntu VM as well.

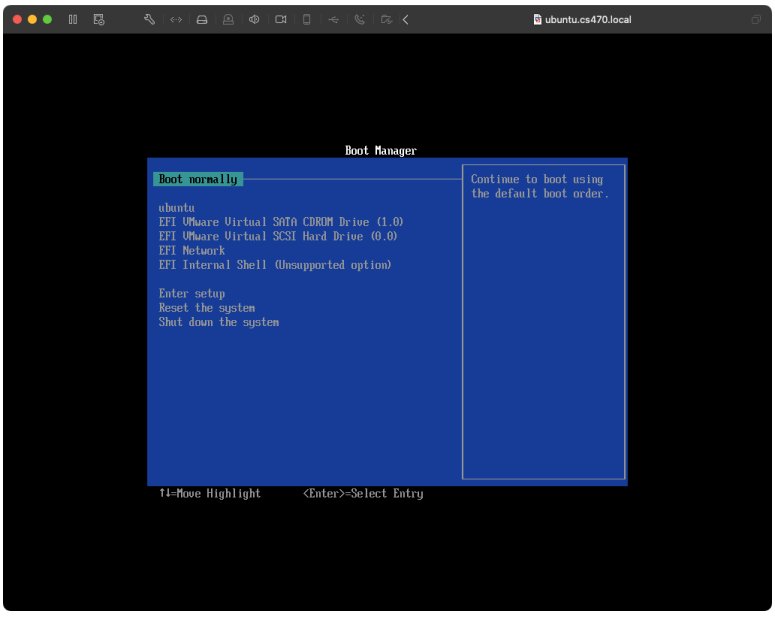
You might need to delete an old image … when trying to troubleshoot, remember to consult `systemctl` and `journalctl`, or wherever else `systemctl` tells you to go. Also try manually starting the container/service directly with `docker` on the command line (not `docker compose`) and see if you get a more solid error message.

part seven: playing with GRUB for fun, profit, and single user mode

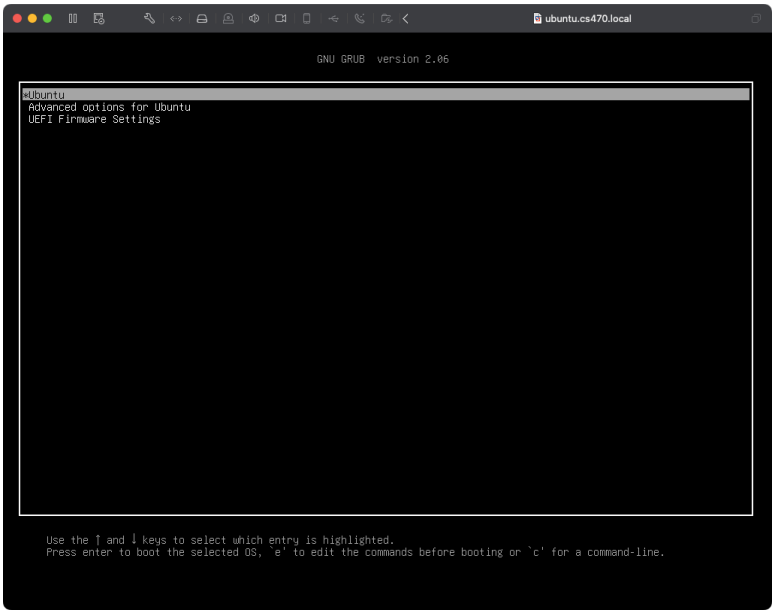
GRUB, the "grand unified boot loader," is far and away the most common boot loader used in conjunction with Linux. It supports other operating systems for dual-booting, allows for the reading of many kinds of filesystems from an early-stage boot loader, and allows operation from a menu pre-populated with various boot options or from a command line.

To reach the Linux boot loader menu, to boot into single user mode or boot with something besides the defaults, you hold down the left `shift` key right after rebooting or powering on the machine, if booting with "legacy" BIOS firmware, or the `escape` key if using UEFI.

Unfortunately, the `escape` key is also the key used to get into the UEFI boot firmware menus for VMware and UTM. If you end up at the menu in the screenshot below, or one similar to it, just use the arrow keys and enter to choose the hard disk boot option, and then get ready to lean on the escape key again right after that to get into GRUB.

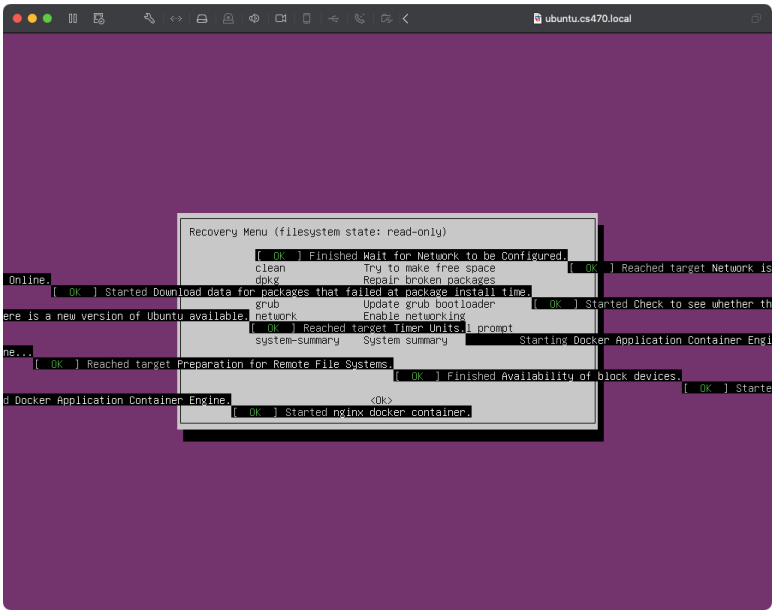


Also unfortunately, the `escape` key Is also used to break out of GRUB's menu interface into a command line. If this happens to you, just enter the command `normal` at a GRUB command line and you'll be returned to the menu. After doing this, you may want to hit `escape` one more time to kill GRUB's auto-boot timer if one is configured. You will not be returned to the command line interface a second time.

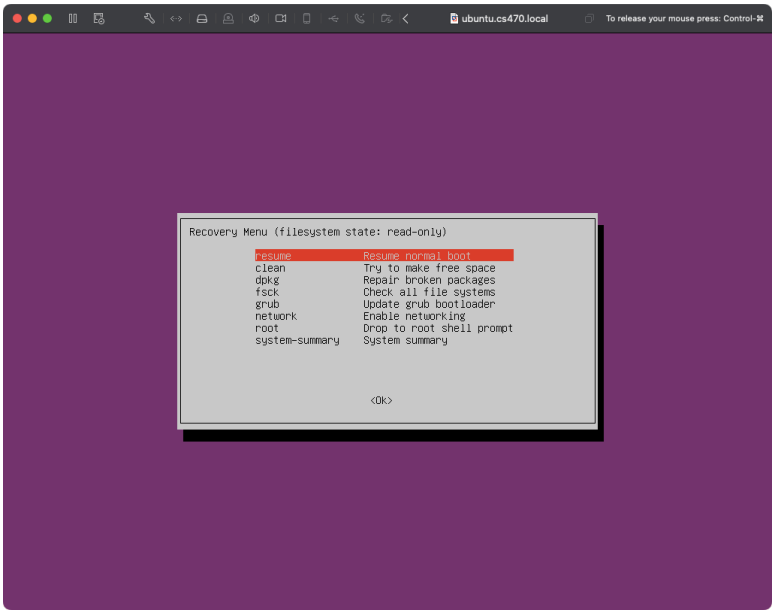


Instead, you should see something like the above menu. If you select “advanced options” you’ll end up at a menu where you can choose between all the installed versions of the kernel, including the option to boot into “recovery mode.” Recovery mode presents a menu with automation for various common maintenance tasks, including freeing up disk space, cleaning broken packages, and updating the GRUB bootloader. The latter is often very useful, as GRUB tends to use filesystem UUIDs to configure booting, and any number of things you might need to do including replacing or upgrading a storage device, or restoring from a backup, might torpedo your ability to cleanly boot up your system.

If the boot process continues and sprays text over your menu like in this screenshot …



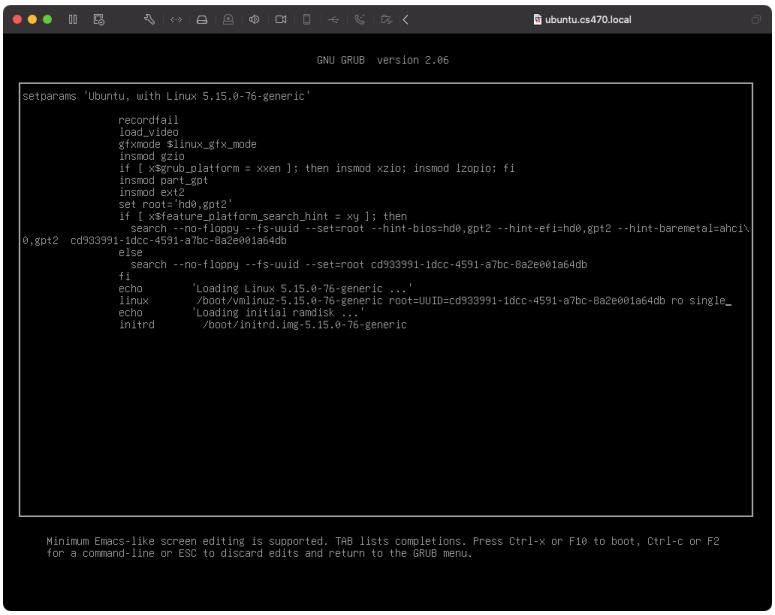
… remember that `control-L` will force most terminals including this console to re-draw the current contents.



Selecting the "root" option at this menu brings you into single user mode for interactive maintenance, just like we did in lab 1 with OpenBSD, only you won't be surprised to find that single user mode in Ubuntu is far more friendly … the root filesystem is already mounted read/write if possible, and you don't have to configure the terminal.

Note that the above menu is specific to Ubuntu … if you're trying to get into single user mode on another Linux distribution (like Rocky), you will probably need to hit `e` to do a one-time (non-persistent) edit to one of the existing menu options, and add `single` at the end of the line that starts with `linux`, to let the Linux kernel know you'd like to boot into single-user mode. Then, hit `control-x` or `F10` to boot into single user mode.

In my experience, trying to boot into single user mode often causes the kernel to hang in ARM VMs, especially with Red Hat derivatives like Rocky Linux. You're better off trying the rescue or recovery option.



If your GRUB configuration is completely poked up, I've often done something like in this link …

https://superuser.com/questions/1237684/how-to-boot-from-grub-shell

… to get my instance booted. Again, for those of you in UTM, using a serial console as your interface with your VM instead of an emulated display may help out greatly here.

The GRUB menu can understand filesystems, to help you easily locate a kernel and an `initrd` (initial RAM disk), as these are generally the only two things you need to boot a Linux-based operating system up.

part eight: Linux special virtual filesystems

This section is just expository; look at things … there are no changes made in this section.

The `/proc` and `/sys` filesystems are virtual filesystems, used to expose process functionality (`/proc`) and system hardware and kernel implementation details (`/sys`). There are a number of convenience features here in both cases.

50. Along with union filesystems and various other innovations from Plan 9, the `/proc` file tree (also often called `procfs`) was ported over to Linux to provide a way to abstract process functions, memory, and metadata as a file tree, so that processes could be interacted with using file-based APIs.

    The `/proc` file tree was initially intended to be only a clearinghouse for process information, but it suffered from scope creep almost immediately, gaining information about system configuration, kernel details, and devices … none of which are processes. If you look at the root level of the `/proc` directory on your Ubuntu VM, you can see how true this is. Information abounds about buses, and devices, and power management, even the CPU itself … all things which would make more sense under `/sys`.

    ```
    $ more /proc/cpuinfo
    ```

    If you pick a particular process, for instance the `docker-compose` process running our containerized web server on this VM, you can explore various attributes of the process by probing the file tree under its process ID number …

    ```
    $ ps auxww | grep docker-compose
    root        2121  0.4  0.5 1725804 45312 ?       Sl   01:33   0:00 /usr/libexec/docker/cli-
    plugins/docker-compose compose -f /etc/docker/nginx.yml up
    root        2122  0.3  0.5 1799792 45056 ?       Sl   01:33   0:00 /usr/libexec/docker/cli-
    plugins/docker-compose compose -f /etc/docker/portainer.yml up
    root        2126  0.3  0.5 1799536 45568 ?       Sl   01:33   0:00 /usr/libexec/docker/cli-
    plugins/docker-compose compose -f /etc/docker/vaultwarden.yml up
    ```

    … so for my `docker-compose` process for my `nginx` container, I wanted to look under `/proc/2121` … use `ls` now, substitute in your web server process' ID number in this and the commands below, and look at what you find in there.

For instance, the `cmdline` "file" yields a bastardization of the command line options we supplied to `docker-compose` in our `systemd` unit file.

```
$ more /proc/2121/cmdline
/usr/bin/python3/usr/bin/docker-compose-f/etc/docker/nginx.ymlup
```

Note that we don't need `sudo` rights here, because this information is typically held to be public within the system, and is also visible in the `ps` output above.

The `environ` "file" can be probed, if you own the process or leverage `sudo` rights, to display the full set of environment variables being used by the process.

```
$ sudo cat /proc/2121/environ
LANG=en_US.UTF-
8PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/snap/binUSER=rootINVOCATIO
N_ID=defecb92bf1648b0a55dd8559e36bfeaJOURNAL_STREAM=8:19189SYSTEMD_EXEC_PID=2070MEM
ORY_PRESSURE_WATCH=/sys/fs/cgroup/system.slice/docker-
nginx.service/memory.pressureMEMORY_PRESSURE_WRITE=c29tZSAyMDAwMDAgMjAwMDAwMAA=DOCK
ER_CLI_PLUGIN_ORIGINAL_CLI_COMMAND=/usr/bin/dockerOTEL_RESOURCE_ATTRIBUTES=docker.c
li.cobra.command_path=docker%20composeDOCKER_CLI_PLUGIN_SOCKET=@docker_cli_16bddc55
44f3dd6ff53deb3f84b18ce5
```

The "file" `mem` … this is a virtual file, as are all the rest, that can be used to probe the process' virtual memory space, and the file `maps` can be viewed to show the regions being used within that process' virtual memory and their respective permissions.

```
$ sudo cat /proc/2121/maps
```

I'm not going to share the output here, because it's long, but suffice it to say that you should see the interpreter for `docker-compose`, the heap, the stack, variable areas, and virtual address space called out for dynamically-linked libraries.

Using the information in both of these files, a script can be easily written to dump the contents of memory from a particular process. Just like walking all of a web server's pages and links, this activity is often called "scraping." I've written such a script, and I'm sharing it here with you so that you can take a look at it and see what it does as an example of a short-yet-powerful shell script, and a good example of the things that can be done with the file abstractions here in `/proc`.

```
$ wget https://slagheap.net/media/cs470/memscrape
```

You'll have to make it executable or an argument to `bash` to run it, and it will also require `sudo` rights. Don't have `wget`? You should know what to do by now.

51. The `/sys` filesystem has a noble design goal but isn't quite so useful, in my humble opinion.

    `sysfs` was developed to take the system-level information out of `/proc` and `/proc/sys`, though a bunch of it still remains, as you saw in the last step. `sysfs` is the filesystem type for `/sys`. If you look at the output of `mount` without any options on your Ubuntu VM, it's very noisy. It's even still if you

filter it through grep sys, but this should stick out near the top of the output.

```
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
```

If you run `man sysfs` on Ubuntu, you'll probably find – like I did – that there's an API called `sysfs` that'll fetch information about a mounted filesystem.

```
$ man -k sysfs
sysfs (2)            - get filesystem type information
sysfs (5)            - a filesystem for exporting kernel objects
```

So it's `man 5 sysfs` that we're looking for here, but if you look at the `man` page, a lot is still "to be documented," and this is to be expected since sysfs is still very much in motion at taking over work from `procfs`.

`sysfs` – and the things intended to move into `/sys` from `/proc/sys` – behave in many ways like what the BSD operating systems use `sysctl` for: to expose kernel memory structures for the purpose of allowing kernel-level system configuration information to be polled, tweaked, and tuned. Whereas `sysctl` provides both an API and a command line for changing values within a dot-notated hierarchical namespace, `sysfs` changes the dots to slashes and makes a file namespace play out of it.

If you want to configure your Linux instance as a router or firewall, one of the first things you'd do (again, don't do this) would be to enable routing in your network stack. The old, BSD-ish way of doing this would be …

```
$ sudo sysctl -w net.ipv4.ip_forward=1
```

… now, you'd just change the contents of the file `/proc/sys/net/ipv4/ip_forward` from 0 to 1.

Yep, this is intended `sysfs` functionality, but it's still under `/proc`. `/proc/sys` is considered part of sysfs, and it is the stated intent of the kernel team to move `/proc/sys` out into `/sys` and to make `sysfs` the official way to change kernel settings; the `sysctl` API has been deprecated since version 5.5 of the Linux kernel, and the `sysctl` command is just an interface on top of `sysfs`.

What can you do with `/sys` already? You can enumerate your network interfaces …

```
$ ls -l /sys/class/net
```

… or the enabled features and instances of various kinds of filesystems …

```
$ ls -l /sys/fs/ext4
```

… access firmware …

```
$ ls -l /sys/firmware
```

… or kernel modules, amongst many other things.

```
$ ls -l /sys/modules
```

part nine: building a kernel

In many use cases, for instance "IoT" devices and network gear, you'll often have to build a custom version of the Linux kernel. If you're in a memory-strapped or low-powered device, you'll want it to be as small as possible for the sake of economy. As we discussed in lecture, the kernel is necessarily a part of every process' virtual memory space and cannot be swapped out, so the less kernel there is, the more there is for every other process on the system. If you're in a funny piece of hardware, you may have to build the kernel because your hardware isn't built-in.

We started this lab with 8 GB RAM because the Linux kernel will flatly fail to build with less than 4 GB and a bunch of additional virtual memory space. Virtual memory is slower, though, and will wear out an SSD if you're using one, so let's not. As usual, we'll close this lab with tuning and will take most of that memory back.

52. First, let's install all the packages required for compiling a kernel …

    ```
    $ sudo apt install bc binutils bison dkms dwarves flex gawk gcc git gnupg2 gzip
    libelf-dev libncurses-dev libssl-dev libudev-dev libpci-dev make openssl pahole
    perl-base rsync tar xz-utils
    ```

    … and support files for the current version of the Linux kernel that we're been given by Ubuntu, through the package manager. This may reply that it's already installed; obviously not a big deal if it is.

    ```
    $ sudo apt install linux-image-$(uname -r)
    ```

    The use of the dollar sign and parenthesis above is logically equivalent to how we use backticks to embed a nested command. It's equivalent to this command.

    ```
    $ sudo apt install linux-image-`uname -r`
    ```

    With `uname -r` above, we're referencing the current release version of the kernel.

    ```
    $ uname -r
    6.8.0-55-generic
    ```

53. Head over to kernel.org and find the latest release. At the time this lab was written, kernel.org had a big yellow button advertising the latest release version, 6.14. When you read this, it will almost certainly be something different, but that's okay … just build the most recent. You'll want to right-click on that big bright yellow button, and select to copy the link to the clipboard.

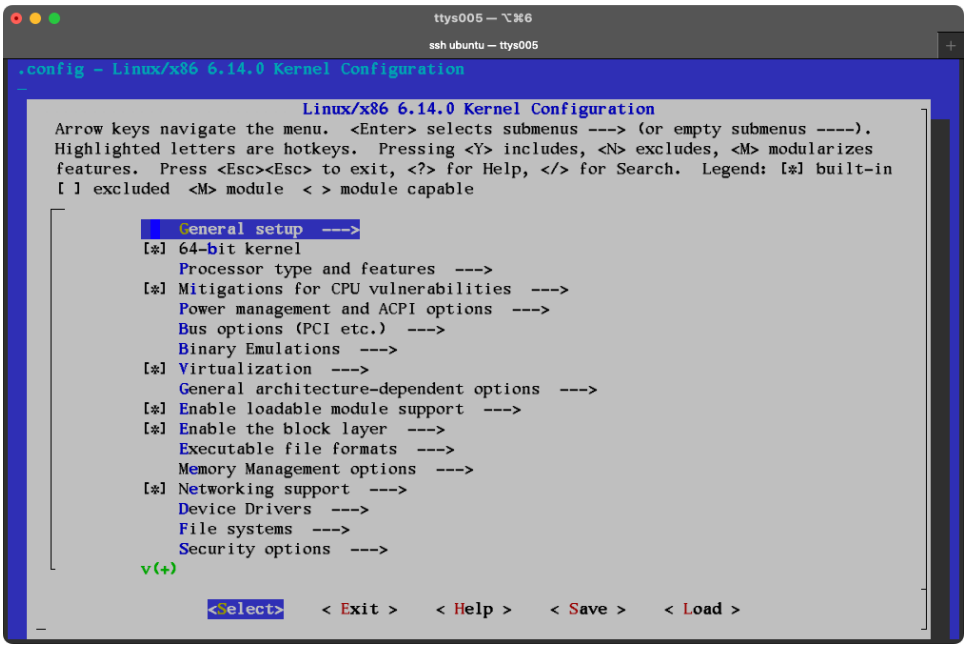
    In your Ubuntu VM, `cd /srv/nfs` and use `wget` again to download the kernel source tarball from the link you just copied to the clipboard as an argument, just like when you grabbed `memscrape` from me just a few steps ago.

As you'll note, the download is a `.tar.xz` … because it's x-zipped, you'll want to use the caps `J` switch with `tar` to unpack it. This will take a while. The Linux kernel is big … really big, and this is why they chose x-zip … it's more computationally expensive, but it really saves space, as you can see.

```
$ du -sh ./linux*
1.7G    ./linux-6.14
143M    ./linux-6.14.tar.xz
```

54. Change directories into the Linux source tree you just extracted, and run `make help`. You'll see that the kernel source tree supports multiple build targets, and has multiple "configuration targets" where the source tree will help you build a configuration file for the Linux kernel.

    Try `make menuconfig` … you'll see that it goes off, compiling a small-ish piece of software to help you generate `.config`, the configuration file used when compiling the kernel. It then runs `menuconfig`, which is somewhat reminiscent of text-based menus presented by FreeBSD's installer and ports tree.



Please feel free to explore and poke around in this menu interface. You won't screw anything up, because once you're done, exit out, and to be safe, we're going to recycle the configuration of our running kernel, the one provided by Ubuntu …

```
$ cp /boot/config-`uname -r` .config
```

… and let's update that configuration.

```
$ make olddefconfig
```

Warnings you receive here are probably okay. Errors are not. Near the end of the output, you should see …

```
# configuration written to .config
```

… and you'll notice that the kernel's build scripts backed up your old configuration to `.config.old` …

```
$ ls -l .config*
-rw-r--r-- 1 peter peter 295777 Mar 25 22:00 .config
-rw-r--r-- 1 peter peter 287562 Mar 25 22:00 .config.old
```

The configuration file is pretty big as far as text files go, almost 300k, and if you page through it with `more`, you'll see why … it's literally a dictionary of all macros used in the kernel source for various options, all the possible options you could explore in `make menuconfig`. Yes, it's way more configurable, but it still makes me miss the relative simplicity and elegance of BSD kernel configuration files.

55. By default, Debian and its derivatives (of which Ubuntu is one) use a certificate to sign all kernel modules, but we don't have that certificate. For now, on this one little VM here with its custom kernel, we don't care enough to get it. Let's disable this capability.

    Use `vi` to edit `.config`, find the settings `CONFIG_MODULE_SIG_KEY`, `CONFIG_SYSTEM_TRUSTED_KEYS`, and `CONFIG_SYSTEM_REVOCATION_KEYS`. Remove everything inside the quotes for each of these three settings, leaving each as a null string.

    Find the setting `CONFIG_MODULE_SIG_ALL`, and set it to `n` for "no."

    Also find the variable `CONFIG_LOCALVERSION`, and inside the quotes on that line, insert `-CS470`, our class department and number with a dash before it to separate it from the rest of the version string.

    With those changes done, save out `.config`.

56. At last, let's fire up some compilers. This will go way faster if your Ubuntu VM has more cores. You can find out how many logical cores you have by running the command `nproc`. I say "logical cores" here, because on some architectures, this command may take hyper-threading features of the CPU into account, and give you double the number of physical cores.

    On my ARM Mac, `nproc` returned 2, on my Intel Mac, it also returned 2. If you'd like to shut down your VM and give it more cores, these will just be more threads/processes on your host system, and I am not going to grade you on the number of cores your VM uses. Keep in mind, however: more cores here means more compilers running, which means more RAM being consumed, and when figuring out the right mix, I've had OoM (out of memory) show up only on the console window and kill my compile job mid-stream.

    Once you're done with that, if you did anything, invoke `make` with the `-j` switch to actually build the kernel. If you're thinking about doing this over an SSH session, this would be really good to do on the VM's console or inside a `screen` instance like you were showed on demo day, to make sure a loss of connectivity in between your host and Ubuntu VM didn't kill your kernel build …

```
$ make -j`nproc` |& tee -a /srv/nfs/kernel.make.log
```

… or just run one compile task at a time to play it safe.

```
$ make |& tee -a /srv/nfs/kernel.make.log
```

The `-j` switch tells it to run as many parallel compile jobs as you have logical cores, and we're redirecting standard error (`&`) along with standard output in the pipe, and additionally directing them both to a logfile for later perusal, if required. `tee` allows output to be sent both to the terminal *and* to a file, so that you can watch the output without having to `tail` a file, and then break out of `tail` when you think it's done. You will get both a log trail of the compile job and a prompt when it is done or stops for an error.

This will take a while; back on kernel version 6.5.5, it took a couple hours on my ARM Mac with four cores, and almost four hours on my Intel Mac with two cores. I didn't time it this time around.

If your kernel build ends in error like this (note `Killed` after the second line below, starting `BTF`) …

```
  LD      .tmp_vmlinux.btf
  BTF     .btf.vmlinux.bin.o
Killed
  LD      .tmp_vmlinux.kallsyms1
  NM      .tmp_vmlinux.kallsyms1.syms
  KSYMS   .tmp_vmlinux.kallsyms1.S
  AS      .tmp_vmlinux.kallsyms1.S
  LD      .tmp_vmlinux.kallsyms2
  NM      .tmp_vmlinux.kallsyms2.syms
  KSYMS   .tmp_vmlinux.kallsyms2.S
  AS      .tmp_vmlinux.kallsyms2.S
  LD      vmlinux
  BTFIDS  vmlinux
libbpf: failed to find '.BTF' ELF section in vmlinux
FAILED: load BTF from vmlinux: No data available
make[2]: *** [scripts/Makefile.vmlinux:37: vmlinux] Error 255
make[2]: *** Deleting file 'vmlinux'
make[1]: *** [/srv/nfs/linux-6.14/Makefile:1159: vmlinux] Error 2
make: *** [Makefile:240: __sub-make] Error 2
```

… then it's a virtual lock that your kernel build is dying because your VM is out of memory, including swap. Just as well … it's good that we get to use this as an excuse to show you how to add swap to a Linux system if you need it.

First, create a file with appropriate permissions.

```
$ sudo install -o root -g root -m 0600 /dev/null /var/swap
```

Then make sure it's the appropriate size, 4 GB.

```
$ sudo fallocate -l 4G /var/swap
```

Prepare it to be used for virtual memory.

```
$ sudo mkswap /var/swap
```

Finally, activate it as swap.

```
$ sudo swapon /var/swap
```

The command `free -h` should show that you have roughly 6 GB of swap now, and if you cat `/proc/swaps`, you should see that most of it is free.

This swap file could be added to `/etc/fstab` if we wanted to keep it permanently … but we don't. Once you get the kernel to finish building, you should be good to remove the additional temporary swap space from virtual memory …

```
$ sudo swapoff /var/swap
```

… and then from storage entirely.

```
$ sudo rm /var/swap
```

While you're having fun not being able to use your computer for the time being, let's preview the next step with `lsmod`. `lsmod` just lists out all loaded kernel modules. The first column is the name of the module, the second is the module's size in bytes, the third column is how many instances of said module are being used at that particular moment, and the last is a list of other kernel modules dependent on it.

Open up another shell and use the `lsmod` command to see what Ubuntu has before we install this new kernel. We're going to compare this output to what we get after we build and install.

```
$ lsmod | sort -o /srv/nfs/lsmod-`uname -r`
```

57. Here begins the installation of the new kernel. First, we install kernel modules built along with the kernel. If you messed up disabling module signatures, you'll find out soon enough.

```
$ sudo make modules_install
```

!! IMPORTANT NOTE: Remember that root is mapped to nobody in an NFS share. If you get an error, especially a clumsy one about spaces in the file path, check the permissions from the root of the filesystem down into your kernel source tree.

If everything goes as planned, you'll see `/lib/modules/6.14-CS470` get filled with `.ko` files (kernel objects, or modules) followed by a `DEPMOD` (module dependency) scan.

Kernel modules, if you remember from lecture 6, are pieces of code you can dynamically load into the kernel and unload from the kernel to provide support for different hardware, filesystems, networking protocols, virtualization, and so on. If ever, in the real world, your wi-fi interface, Nvidia GPU, or laptop trackpad randomly vanishes from your computer after a kernel update, you're probably just missing the kernel module(s) that provides support for it … for your new kernel.

The size of the installed modules, however, is preposterous by default. Here, you can see the module directories supplied by Ubuntu on my Ubuntu VM compared with the one I just installed.

```
$ du -sh /lib/modules/*
7.6G  /lib/modules/6.14.0-CS470
154M  /lib/modules/6.8.0-55-generic
```

In kernel 6.11.3, in the last iteration of this lab, it was only 2.0 GB! In order to fix this ghastly overconsumption of storage, we need to "strip" the modules. The utility `strip` is intended specifically to save disk space like this, and modifies the existing binary, rather than making a modified copy of it. We used to invoke strip ourselves with something like this …

```
$ sudo find /lib/modules/6.14.0-CS470 -name "*.ko" -exec strip --strip-unneeded {} \;
```

… `find` is a really powerful utility … we'll explore it more in lab 5. In the command above, we're finding all files in our new kernel modules directory ending in `.ko` – the quotes prevent the shell from trying to expand/glob the star – and running `strip --strip-unneeded` on all matching files.

Fortunately and unfortunately, starting recently in Ubuntu 23.10 and also to save disk space, kernel modules are compressed by default, using Zstandard (also called "Zstd") compression. You should have noticed `ZSTD` scrolling by as it was invoked while installing the modules … you'll definitely notice it happening now. The modules being compressed, however, means we can't `strip` them directly … but there is a switch we can provide the kernel `Makefile` to get it to `strip` them before compression.

Remove the modules you installed …

```
$ sudo rm -rf /lib/modules/6.14.0-CS470
```

… then re-install the stripped versions of them.

```
$ sudo make modules_install INSTALL_MOD_STRIP=1
```

After running the above, the new module directory uses way less space … about 7 GB less. Much better.

```
$ du -sh /lib/modules/6.14.0-CS470
588M  /lib/modules/6.14.0-CS470
```

I recommend you look at the `man` pages for both `strip` and `find`.

58. This step only needs to be done on ARM Macs / ARM VMs and other architectures, not on Intel/AMD CPUs.

    If you're running on a host with an Intel or AMD CPU, you can skip this step.

    ```
    $ sudo make dtbs_install
    ```

59. Now, install the kernel.

```
$ sudo make install
```

You should see a few things happen; the kernel makes a new `initramfs` (initial RAM filesystem), then installs that and the new kernel (`vmlinuz-*`) to `/boot`. Finally, it tells GRUB to sense all the kernels under `/boot` and rebuild its configuration file. If you ever have to do this manually, the command you want to run is right here.

```
$ sudo grub-mkconfig -o /boot/grub/grub.cfg
```

You could also run `update-grub`, a tiny script that just runs the above command for you.

```
$ more `which update-grub`
#!/bin/sh
set -e
exec grub-mkconfig -o /boot/grub/grub.cfg "$@"
```

60. Let's take a minute here to review the Linux boot process.

Whether you're working with a real or virtual machine, after you press the power button, the BIOS (UEFI) picks a boot entry saved in the EFI variables stored in UEFI, a chip on your motherboard if it's a physical computer. EFI variables can be set from the UEFI menu, or from the OS with the `efibootmgr` command. The chosen boot entry stores where the bootloader, in this case GRUB, can be found within the disk's EFI system partition (what we mount at `/boot/efi`) and runs it. If you had multiple operating systems installed on your computer's storage device(s), you would typically have a boot entry for each OS.

After GRUB has been executed, we get to choose to boot from a handful of GRUB menu entries, as defined in `/boot/grub/grub.cfg`. Even though the root filesystem is nowhere close to being mounted yet, GRUB comes statically linked with a handful of filesystem drivers, allowing it to find where `/boot/grub/grub.cfg` resides on our disk, even if that disk is actually an LVM volume, a RAID array, heavily compressed or heavily encrypted ... GRUB even comes with a network stack to be able to search for its config file on an NFS share.

On your Ubuntu VM, look at `/boot/grub/grub.cfg`, and look for configuration stanzas starting with `menuentry`. You should see two menu entries, one each for our new and old kernels, as well as sub-entries for booting into recovery mode. Pay attention to the following lines, which define where the Linux kernel image and `initramfs` – called `initrd` (initial RAM disk) for historical reasons – exist for that particular menu entry.

```
echo    'Loading Linux 6.14.0-CS470 ...'
linux   /boot/vmlinuz-6.14.0-CS470 root=UUID=1db32191-380e-4445-9e5a-2e548c454c91 ro
echo    'Loading initial ramdisk ...'
initrd  /boot/initrd.img-6.14.0-CS470
```

Notice how the UUID of the root filesystem partition is passed as a parameter to the kernel image. You could also use device notation here, like `root=/dev/sda2`, but using UUIDs is preferred so that we

always choose the right partition, even in increasingly common cases of `/dev/sda` becoming `/dev/sdb` when a another storage device is added to the system. This also makes name mismatches or collisions between our partitions are astronomically unlikely.

Isn't it awesome to once again behold the "everything is a file" doctrine of the Unix philosophy? Try running `file` on the kernel image and `initramfs`. You can straight-up trash your kernel just like any other file, and could probably swap in a friend's kernel image (along with his kernel modules, GRUB configuration lines, etc.), to replace it for the next time you boot. Good luck trying anything remotely similar to that on Windows.

Anyways, when you hit the "enter" key to pick a GRUB menu entry, GRUB first loads that entry's respective kernel image, located at `/boot/vmlinuz-*`. The kernel on old Unix systems was traditionally stored at `/unix`. With the advent of virtual memory, the kernel would be called `vmunix`, with the advent of Linux, `vmlinux`, and when we started compressing our kernel images to save disk space and get past size restrictions of some architectures, `vmlinuz`. From our kernel config, we can see that Zstd compression was enabled by default on our kernel image, not any of the other standard compression algorithms.

```
$ grep CONFIG_KERNEL_ /srv/nfs/linux-6.14/.config
# CONFIG_KERNEL_GZIP is not set
# CONFIG_KERNEL_LZMA is not set
# CONFIG_KERNEL_XZ is not set
# CONFIG_KERNEL_LZO is not set
# CONFIG_KERNEL_LZ4 is not set
CONFIG_KERNEL_ZSTD=y
CONFIG_KERNEL_MODE_NEON=y
```

After loading the kernel image, the root filesystem isn't immediately mounted. The kernel first mounts the `initramfs`, which holds filesystem drivers and other utilities required to mount the real root filesystem. We also typically want to run tools like `fsck` on our root filesystem before mounting it—the `initramfs` stage of booting is a perfect time for this.

Finally, the root filesystem is mounted, `init` is run, our `systemd` services start (or our `rc` scripts if we were on a BSD), and we get a login prompt.

Hopefully the Linux boot process and its modularity makes more sense to you now.

61. Your new kernel should be the new default; reboot your Ubuntu VM, go into the "advanced options" menu, and confirm this. Reboot Ubuntu with your new kernel, check out the GRUB menu on the way back in because you'll likely have to manually pick your kernel, and check out your new kernel with `uname`. You should see something similar to the below.

```
$ uname -a
Linux ubuntu 6.14.0-CS470 #1 SMP PREEMPT_DYNAMIC Wed Mar 26 21:08:15 UTC 2025
x86_64 x86_64 x86_64 GNU/Linux

$ uname -r
6.14.0-CS470
```

Once you're fairly sure your new kernel is installed correctly and working acceptably, clean up your

kernel source tree but don't remove it entirely … we'll use parts of it later for grading, but it takes up a pretty crazy amount of space and we'll need that storage space later on, in lab 6.

```
$ du -sh /srv/nfs/linux-6.14
28G   /srv/nfs/linux-6.14
```

Now you know why I asked for this additional space back in lab 3 … `make clean` is exactly what the doctor ordered here.

```
$ cd /srv/nfs/linux-6.14 && make clean && du -sh /srv/nfs/linux-6.14
…
  CLEAN   security/apparmor
  CLEAN   security/selinux
  CLEAN   security/tomoyo
  CLEAN   usr
  CLEAN   .
  CLEAN   modules.builtin modules.builtin.modinfo .vmlinux.objs .vmlinux.export.c
1.7G  /srv/nfs/linux-6.14
```

Much better again, to the tune of over 26 gigabytes of space savings.

62. Remember that file we made a few steps ago to demonstrate what `lsmod` does? Let's use that command again to dump out all the modules in this new kernel. Now `diff` the two files to find the differences between those two files and pipe that into `more` to get nice looking output like this …

```
2c2
< aesni_intel          356352  0
---
> aesni_intel          122880  0
9c9
< auth_rpcgss          184320  1 rpcsec_gss_krb5
---
> auth_rpcgss          180224  1 rpcsec_gss_krb5
12,14c12,14
< blake2b_generic       24576  0
< bluetooth           1028096  6 btrtl,btmtk,btintel,btbcm,btusb
< bridge               421888  1 br_netfilter
---
> blake2b_generic       20480  0
> bluetooth            983040  6 btrtl,btmtk,btintel,btbcm,btusb
> bridge               405504  1 br_netfilter
17,19c17,19
< btintel              57344  1 btusb
< btmtk                12288  1 btusb
< btrfs              2015232  0
---
> btintel              65536  1 btusb
> btmtk                36864  1 btusb
> btrfs              2019328  0
21,22c21,22
< btusb                77824  0
< cfg80211           1323008  0
---
> btusb                69632  0
> cfg80211           1335296  0
29,32c29,30
< drm_ttm_helper       12288  1 vmwgfx
< e1000               180224  0
< ecc                  45056  1 ecdh_generic
< ecdh_generic         16384  1 bluetooth
---
> drm_ttm_helper       16384  2 vmwgfx
> e1000                172032  0
37c35
< hid                  180224  2 usbhid,hid_generic
---
> hid                  262144  2 usbhid,hid_generic
39a38
> i2c_smbus             16384  1 i2c_piix4
41c40
< intel_rapl_common      40960  1 intel_rapl_msr
---
> intel_rapl_common      49152  1 intel_rapl_msr
53c52
< mptspi               24576  4
--More--
```

Notice any interesting differences? The bluetooth module stood out to me. It got smaller in the new kernel.

63. Another handy utility is `modprobe`; it loads and removes kernel modules. Based on what you know about the information `lsmod` gives you, you could try to remove the modules that are not needed. To confirm you actually got rid of the kernel modules, you can use a combination of `lsmod` and `grep`.

    Congratulations; you've just built, installed, and explored a fresh Linux kernel from source.

part ten: patching and tuning

64. Let's patch our systems. We've been using `apt` repeatedly to fetch packages, and just like with Rocky Linux and other Red Hat derivatives, the package manager is also used to patch and upgrade the whole operating system.

    The `apt` package manager's "update" verb tells it to synchronize its caches against lists of currently-

available packages in its configured repositories.

```
$ sudo apt update
```

The command should show passing through each of its configured repositories and sub-repositories, and then produce output similar to the following …

```
41 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

… you should also start seeing similar output in e-mails from the `crontab` entry we set up earlier in this lab, to let you know when you have things to patch. When you do, the "upgrade" verb is the droid you're looking for.

```
$ sudo apt upgrade
```

You might see `apt` tell you that several packages are no longer needed; you'll also find that this is a common occurrence. Packages are often built by default in different ways with different dependencies, and if an installed package is a dependency, not one that administrator(s) specifically requested by name, then `apt` will do you the service of telling you its services are no longer required.

If you see any such packages that are no longer needed, and you want to reclaim the disk space, do this, as `apt` recommends …

```
$ sudo apt autoremove
```

To remove packages, it's as simple as …

```
$ sudo apt remove <package>
```

To remove packages as well as their configuration files with extreme prejudice …

```
$ sudo apt purge <package>
```

65. My ARM64 Ubuntu VM had been springing this on me the whole time, since the first reboot:

```
1 device has a firmware upgrade available.
Run `fwupdmgr get-upgrades` for more information.
```

Running `fwupdmgr get-upgrades` produced several advisories for problems with secure boot and the associated firmware patches. Since this is a VM, there are no chips to flash to update the firmware of the compute instance. The firmware are files provided by UTM, and the update will be properly done in an update to UTM.

Take no action here; this was just for everybody to read and us to not ignore that heads up from Ubuntu's default login scripts.

66. Using the `top` command, I found my Ubuntu instance in VMware was using way less than 8 GB RAM.

Power it off. Remove some RAM, then power it back on. Experiment with how little RAM you can get away with, while your Ubuntu VM operates normally with all its containers.

When you're done with that, you are also done with lab 4.

</lab4>