CS 470: Unix & Linux Sysadmin
Spring 2025 Lab 1
OpenBSD and DNS with BIND9

errata v2, published 2/26/25:

- added back instructions in step #48 for making sure that `named` only uses IPv4 for back-end resolution traffic, in addition to only receiving connections over IPv4

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

- lab 0: setting up your SSH key pair
- lab 0: getting comfortable on the command line and with `vi`

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

- installing the VPN client so your name server will work on campus
- OpenBSD and `named` running and providing name service for cs470.internal
- SSH agent forwarding working properly to grade your work later

Time to set up the first machine on your lab network, using your desktop hypervisor's shared NAT network. This lab will start getting your hands dirty with one of the three most common operating systems descended from Berkeley Unix (commonly also called BSD, or BSD Unix). There are several more boutique ones out there still, like PicoBSD and DragonflyBSD, but the other two big ones are:

- FreeBSD, the most used of all the completely free and open source BSDs, coming in lab 2
- Darwin, the open-source Unix underpinnings of macOS. Yes, it's actually certified as a Unix, and a portion of it is really open source. See Apple's open source page.

OpenBSD was initially a source fork of the NetBSD operating system, formed when Theo de Raadt had finally angered enough fellow NetBSD developers so much, so they kicked him off the project. NetBSD's core mission has always been to port itself to every conceivable hardware platform, for a very long, long time … and it runs on **many** CPU and hardware architectures (see http://netbsd.org/ports/), but it never found much of a user base. The public access system SDF.org uses NetBSD, but it is the **only** place I have seen it used in production, **ever**. FreeBSD and OpenBSD I've both repeatedly seen used in production, mostly FreeBSD.

When founding the OpenBSD project, DeRaadt and crew decided that it would trim down to just a few common CPU architectures, in order to better focus on a different set of goals, including but not limited to:

- code correctness and security
- allowing only permissively-licensed (BSD licensed, not GPL'd) open source code in (even disallowing pre-compiled binary "blobs" for device drivers)
- making use of strong cryptography **everywhere** appropriate

It is also for this reason that OpenBSD is distributed from Calgary, in Canada. Canada's federal government also (like here in the US) took a scary turn against civil rights under its prior leadership, but it still doesn't forbid the export of strong encryption through its borders.

The OpenBSD project took that "code correctness" goal quite seriously: upon forking the NetBSD code tree, it began with a ten person-year audit of its code tree for software flaws ... new ones, old ones, and commonly repeated mistakes in new forms. Ever since that code audit, the OpenBSD project has taken an “evolutionary not revolutionary” approach and made a stable, hard-to-break release every six months or so, always twice a year, since 1995.

Whenever the OpenBSD team has historically developed a distaste for the licensing stance, security track record, or design of a critical component, the OpenBSD project develops a replacement for it, and the replacement often becomes the more popular product in the market.

- Once upon a time, the Finnish company initially behind the SSH protocol (F-Secure) realized it was giving away the farm and closed their source code, trying to make SSH a proprietary product in the process. The OpenBSD project forked the last open source version to write OpenSSH, which became far, far more popular than the original F-Secure SSH virtually overnight because of both cost (free) and license (BSD). It is now used on virtually every Linux, Unix, and embedded device in the world. As noted previously, it's even shipped with every copy of Windows now!

- When OpenSSL, the ubiquitous free and open source library for encryption and SSL/TLS APIs developed a string of high-severity flaws (notably HeartBleed), OpenBSD forked it, did a code audit, fixed a ton of bugs, removed a bunch of old code and old platform support, and released it re-branded as LibreSSL. Apple uses LibreSSL instead of OpenSSL in macOS now, and it was used by Microsoft in Windows 10 as well.

- When they didn’t like the licensing terms for IPFilter, the original bundled host firewall, they wrote `pf`, short for "packet filter," which is now used by FreeBSD, macOS, Solaris, and even by some commercial products for Windows. Then they wrote `pfsync`, which allows for `pf` firewalls to operate in redundant, fault-tolerant clusters.

- When they grew tired of `sendmail`’s hideous configuration files and the difficulty involved with auditing its code, they built their own mail server, OpenSMTPd.

- When they grew tired of the Apache web server’s configuration files and security history, and NGINX’s licensing terms were found to be unsuitable, they wrote their own web server, just called `httpd`.

- OpenBSD even forked its own version of the X11 graphical windowing system ("Xenocara"), because the project leaders didn’t like the canonical X.org server security

> model, its handling of privileged operations, and the source tree's long retention of legacy code for older operating systems.

OpenBSD is third in the size of its installed base, behind macOS and FreeBSD, amongst strongly BSD-based operating systems available today … but those users are typically security researchers, hackers, and sysadmins at the most paranoid entities and on the most security-conscious networks. It's a hotbed of security innovation (see `man unveil` and `man pledge` after you get it installed) and is the only OS that takes out the time to randomize each system's kernel and shared library address space to prevent return-oriented programming (ROP) attacks. It is for this reason that OpenBSD is often used as the basis for embedded firewall software, and as a general-purpose compute platform in extremely high-security and cryptography-intensive applications.

It's also heavily used in networking gear, because of its built-in firewall, `pf`, carp (firewall redundancy), `relayd` (a network proxy or application-layer firewall), OpenBGPD (for internet multihoming), and `rpki-client` (network backbone router-to-router authentication and encryption). The "PKI" in rpki-client is short for "public key infrastructure," a way of establishing trust for encrypted communications … we'll be setting up a certificate authority to for this purpose, on our OpenBSD VM, in lab 1b.

Significant pieces of OpenBSD have been taken and made a part of FreeBSD, macOS, numerous Linux distributions, Windows, commercial software, and embedded products, including firewalls and network hardware. Security functionality developed first by the OpenBSD project became the reference model, in many cases, for many other operating systems. Saying that OpenBSD is highly relevant and influential beyond its own installed base would be an understatement, to say the least … and this is why you're about to get hands on with it.

OpenBSD has the most painful learning curve of all the operating systems we'll be playing with. Even if it wasn't your first experience installing a new OS without a mouse or setting up inside a text-based terminal window, OpenBSD is not written with newbies in mind and has few in the way of creature comforts. It installs a light footprint, expects its users to know what to do, and can be very finicky. Binary patches for the operating system only became available within the last few years … but until then, to patch flaws in the operating system, you had no choice but to re-compile the appropriate portions of the operating system, and often their dependencies too. That said, if you can read and understand the documentation, you can do it, of course.

This lab also serves to plumb the underpinnings of your lab network with DNS, the naming glue for the TCP/IP suite of networking protocols. Nobody wants to remember IP numbers … humans are generally better at remembering names like www.sdsu.edu or openbsd.cs470.internal than they are at remembering 104.20.12.66 or 10.42.77.71.

In the early days of the internet, this was done with a file called simply `hosts`, and we'll play with that file later on in this lab, because it's still there and it still has a valid use. Back in the day, changes to `hosts` used to have to be synced across all the systems on the early internet.

As the internet grew, this rapidly became untenable. The architects of the internet realized that we needed a distributed database with delegated sources of authoritative answers, and a hierarchical naming system based upon organizational names, to allow for organizations to have as many named systems as they need and a flexible taxonomy for naming them.

That doesn't even take into account all of the things that we can do with DNS, like offering up multiple answers for the same name, in case one of them goes down, or providing different answers at different locations geographically … for instance, a DNS lookup answer for the Americas might not offer great performance for users in Asia or Africa, and you could give them a different answer there. There's also what's often called "split DNS," where you have two different views of a domain, whether you're inside or outside the internal network of that entity's domain. Internal DNS would offer answers with internal IPs, whereas external DNS might not even have lookups for most internal hosts. Sometimes we rotate the answers given to DNS queries in order to spread out the load across a fleet of service-providing systems; this is called "round robin" DNS.

When you set up an internet connection for any given entity, you typically stand up a public domain name with a registrar, and this gives you lookups in the WHOIS database. The WHOIS database is required to list an administrative, technical, and billing contact for each domain … and also serves to tell the rest of the internet where to find authoritative DNS servers for that domain. Lookups from a name like www.sdsu.edu to an IP address are called "forward lookups," and are denoted by A ("address") records in a DNS server. "Reverse lookups" are also possible, from an address to a name, a PTR ("pointer") record. Various selected protocols have their own ways of working within DNS to advertise their availability. For instance, every domain on the public internet that wants to receive e-mail has one or more public MX ("mail exchanger") records to tell others online where to send e-mail. You can also force a traversal of the DNS namespace with a CNAME ("canonical name") record, which is effectively an alias or a shortcut, allowing you to aim from one place in the DNS namespace to another. There are several other kinds of records, including NS (name server) records to tell you how to find a name server for a domain or a network, TXT (text) records to allow you to use DNS as a backing store for arbitrary system metadata, and many more … but that's enough for now.

When we stand up a server system on a network, we typically don't have it get its IP address dynamically, so that we can associate a name with a static address. We tend to manually configure the things that DHCP automatically sets up for us on each of our laptops or computers, every time we attach a client system to a network. To statically configure an internet connection, we need four things: an IP address for the system, a netmask for the network, the default gateway that gets us out of that network (if it's attached to the internet, if not we can do without), and the last of which is a name server to resolve names and help us find destinations online.

In our labs, we use cs470.internal for our domain. We use the top-level domain .internal, because .internal is a newly-reserved part of the public DNS namespace for systems internal to each organization connected to the internet. By using .internal, we are not going to have any

names that occur logically below it (to the left of .internal) collide with names of systems out there on the internet. Just in case this doesn't make sense, for instance, there's nothing physically stopping us from using sdsu.edu as our internal lab domain names … but if we stood up a name server that said it was authoritative for sdsu.edu, we would have to clone sdsu.edu's DNS records into our domain zone file on our name server, or our labs wouldn't be able to access anything under sdsu.edu. Whatever namespace we use here in our lab networks, we won't be able to access out there … so we're using something that doesn't conflict.

I have repeatedly looked at using a public domain name, and Microsoft even used to offer a free public domain name via Name.com, but there are many other challenges involved with making this work for small, mobile setups on consumer internet connections. Dynamic DNS would solve the mobility problem, but most consumer broadband blocks server-to-server e-mail traffic. Maybe one day, but for now … cs470.internal it is.

I've placed DNS first as our lab's first service because we have no choice; you can run little air-gapped networks with only IP numbers, but all the services we'll use on the internet are built on top of DNS. I've put it on top of OpenBSD as our lab's first operating system, because once you've conquered OpenBSD, everything else will look and feel easy by comparison. You'll be ready for anything.

part zero: DNS on the campus network

!! THIS SECTION IS REALLY IMPORTANT IF YOU LIVE ON CAMPUS, OR WILL BE USING A LAPTOP YOU BRING ON CAMPUS AS YOUR LAB COMPUTER. THIS VPN WILL BE SET UP AT THE ADD/DROP DEADLINE.

If you are ever curious about the status of network services on campus at SDSU, go to https://status.sdsu.edu …

For security purposes, SDSU's IT security mandates that all DNS traffic use the official campus servers. This is because you can filter out a lot of malware phoning home … by denying DNS lookups of the malware's command-and-control systems, if you know their DNS names.

You get the proper SDSU campus DNS servers from DHCP when your computer connects to an on-campus network. When we set up a recursive name server later in this lab, it will go directly to the internet, bypassing SDSU's local name service and starting with the root DNS servers to recursively resolve DNS names, starting with the TLD (.com, .edu, etc.). Because this setup bypasses SDSU's name servers, this causes lookups of internet sites to fail while on campus for labs in person, or if you live on campus. To work around this, you can either use a personal privacy VPN if you have one that works on the campus network, or you can use the campus' official VPN client for an officially-sponsored solution I worked out with campus IT Security.

To install the campus VPN client, please see SDSU IT Security's directions online at the following location: https://security.sdsu.edu/services/virtual-private-network

If your lab computer is running Windows as its native operating system, you *might* also need to disable the setting in the VPN client that's mentioned in this Palo Alto Networks knowledge base article:

https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA14u000000HB0rCAG

part one: installation

1. First, if you don't have a copy of it, let's grab the latest OpenBSD media and take a look at the repository. Go to the following URL.

   https://mirrors.mit.edu/pub/OpenBSD/7.6/

   Please note all the things that are distributed with a fresh copy, or a new version of the operating system upon release. There's a directory for each supported hardware architecture (`alpha` through `sparc64`, except the `packages` directories), an `ANNOUNCEMENT` file with changes and release notes, a `README` (common, but not much here), SHA256 files with hashes of distribution files to validate them, and a bunch of `.tar.gz` files (sometimes shortened to `.tgz`, more on this later).

   The 64-bit Intel x86-derived CPUs and their clones are often called `x86_64` but are also often called `amd64`. This is because Intel CPU clone-maker AMD created the first set of 64-bit instruction extensions for the 32-bit Intel architecture, while Intel preferred to build an all-new architecture from scratch. In the end, Intel was forced to adopt AMD's instruction set extensions in order to **not** fragment the market!

   Note that there are a bunch of `.tgz` files here; this is because OpenBSD can be booted off a very small mini-distribution (historically called a "mini-root" with many of the BSDs), such as off a classic 1.4 megabyte floppy disk, with enough of the operating system to run a live repair mode or to get online and do an installation of the operating system over the network.

   Think about that: with just 1.4 megabytes of data, one of those old 3.5 inch floppy disks that started disappearing from PCs about 25 years ago when Steve Jobs demoed the original iMac, you can boot and install the operating system over the network, or repair an existing installation.

   At the time of its introduction, many gasped out loud about the sheer gall of the iMac not including a floppy drive, then about the impending death of the floppy disk, but this image should be nostalgic for nobody now. We don't miss them at all.

I'm not trying to invoke nostalgia by bringing this up, though … I bring it up because it's impressive that a modern OS can **still** use this dated device and the comparatively pathetic amount of data it holds to boot a computer, get online, download, install, and configure an operating system. It goes to show how an operating system like OpenBSD can be used to revive an old piece of hardware with modern software that isn't unsafe to run on today's internet … and why OpenBSD is still a great first choice for a lot of small devices today, and for small device applications.

In fact, this is **exactly** what we're going to do, at least at first for a demonstration. We're not going to buy floppy drives or disks or CDs or DVDs, because virtual hardware in virtual machines means we can *emulate* having this hardware, and boot using the image of its removable media, which you can plainly see is just 1.4 MB.

No matter what instruction set your physical hardware provides, go into the `i386` folder and grab `floppy76.img`. Also, grab the file `SHA256.sig`. That's right, I want everybody to grab the 32-bit Intel installer … we're **all** going to boot up the installer for the 32-bit Intel version of OpenBSD.

2. Now, let's verify the hash on the file, and verify we got a good, unaltered copy. If you look at the top of `SHA256.sig`, you'll see a digital signature that can be used to verify that the contents of the file, and thus all the hashes, are accurate. Until we have OpenBSD up and running, we don't have the tools, so let's just validate the hash. If you're on a Mac or Linux system, you can use `openssl` to create a fresh SHA256 hash from the file.

   You should see this:

```
$ openssl sha256 floppy76.img
SHA2-256(floppy76.img)=
4219da0bffbbf9d6a7184b9625a5a637672874e4269ddab36c750b63a86daf5a
```

If you're on a Windows system, you can use the built-in utility `certutil` to hash the floppy image.

```
C:\Users\peter\Downloads>certutil -hashfile floppy76.img sha256
SHA256 hash of floppy76.img:
…
CertUtil: -hashfile command completed successfully.
```

Now compare the hash you generated with what's in the signatures file. You can run the following command to see what the hash was when the OpenBSD Foundation packaged it …

```
grep floppy76.img SHA256.sig
```

Taking this additional step of validating the installation medium would be very common in certain environments, if we **really** wanted to make sure we got the original OpenBSD binaries. Of course, somebody could have replaced the floppy image file on the web server **and** the file with the digital signatures … so how might you protect against this? Perhaps check the signatures for the floppy image (and the image itself, since it's such a a small download!) on a couple more mirrors, as an attack against several mirrors would be more difficult to mount.

3. Create a new, custom virtual machine.

   Another reason many choose OpenBSD for their products or systems is because of low resource requirements. OpenBSD doesn't need much memory and, as you will see, it also has a very small installation footprint.

   Intel CPUs: in whichever VMware desktop product you're using (Workstation or Fusion), choose to make a new, custom VM. You're at your own discretion, aside from the following settings:

   - guest OS: other -> other 64-bit
     we're installing a 32-bit OS, but a 64-bit virtual Intel CPU will run it just fine
   - boot firmware: legacy BIOS
   - create and use a new 16 GB disk
     (you may need to fix the disk size after VM creation, but before booting)
   - RAM: 2048 MB (2 GB)
     (OpenBSD will run with way less than this, but we'll reclaim it later)
   - two virtual CPU cores
   - make sure to use your "NAT" network (on macOS, sometimes "Share with my Mac")
   - customize the VM before booting it up to make changes to conform to the above … add a new floppy drive device, and aim the floppy drive at the image downloaded in step #1

ARM Macs: Make sure you're running the latest release of UTM, 4.6.4 at the time of the writing of this lab. Create a new custom VM, and right away, choose to **emulate**, **not to virtualize**.

- OS: use "other"
- boot device: choose floppy image, and aim it at `floppy76.img`
- architecture: i386 (x86)
  that's right, our ARM Macs are going to emulate 32-bit Intel … and they do okay at it
- system: choose "Standard PC (i440FX + PIIX, 1996) (alias of pc-i440fx-7.2) (pc)"
- RAM: 2048 MB (2 GB)
  (again, OpenBSD will run with way less than this, but we'll reclaim it later)
- two virtual CPU cores
- storage: 16 GB
- make sure to select "Open VM Settings"
- under QEMU, and Tweaks
  - disable UEFI boot
  - enable "force PS/2 controller"
  - make sure "use local time for base clock" is turned off
- under input, disable USB support entirely
- under display, set the emulated display card to "virtio-vga" if not that already
  do not use the GPU supported option
- under network, change your NIC to an Intel e1000 and use "Shared Network"

4. At the initial "Welcome to the OpenBSD installation …" prompt, choose `i` for install. Welcome to a text-based installer … **make sure you carefully read everything it's asking you for**. Getting you away from a mouse was a primary goal here … you're a computer science student, type!

   You're at your own discretion aside from the options I choose below. They are listed below, more or less in order of how you'll be asked for them during the installation script.

   For your system hostname, use "`openbsd`." Don't use the period, or the quotes.

   When asked to choose a network interface, you don't want `vlan0`. You want to choose the other one, probably `em0` since we took the step of configuring an Intel NIC even for those on ARM Macs.

   When asked for an IPv4 address, choose `autoconf`, which is the default. When asked for an IPv6 address: choose `none`, the default … we will not use IPv6 during the class.

   If asked, you do NOT need an HTTP proxy. Odds are you're directly connected to the internet.

‼ IMPORTANT NOTE: **READ** what you're being asked, and don't be afraid to web search. I'll give you a freebie here, though: you don't need a proxy. A proxy is a service that grabs things for you, something you typically only need to use on tightly-controlled networks. A proxy will (try to) make sure you don't go to bad places, learn how to do evil things, or bring home viruses … but we don't have one. A proxy, that is.

When asked whether to start `sshd` by default, choose `yes`. This is the default.

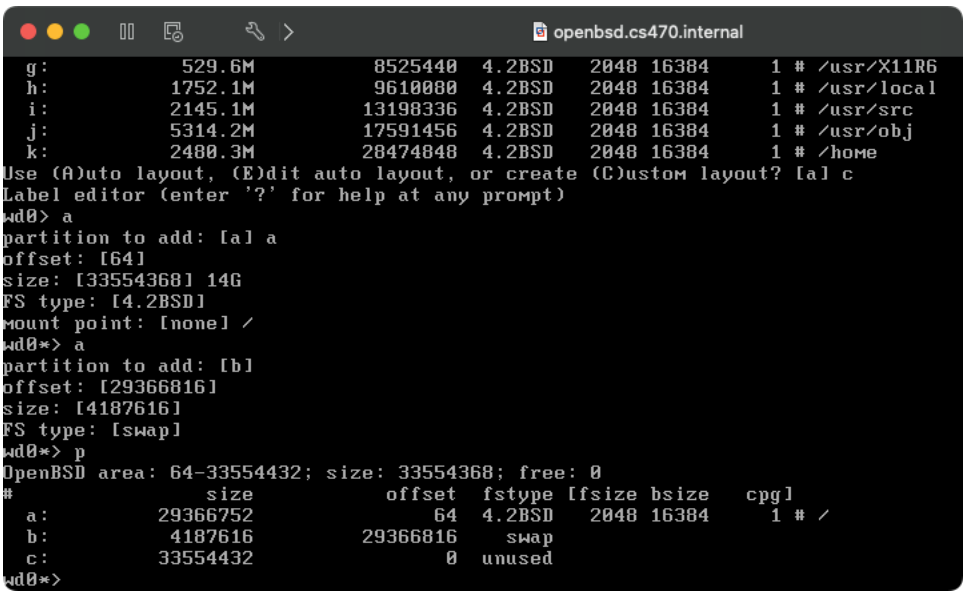
When asked if you'll run the X Window System, say `no`. This is sometimes the default.

When asked to setup a user … in lab zero, I asked you to use your first name … or whatever you wanted, but I don't want you to use **that** universal username here, but rather another universal username we'll be setting up on each of your VMs … use `failsafe` and `failsafe user` as their full name. In lab 1b, we'll set up an LDAP directory to share user accounts across all our VMs. The only account we want configured locally on each VM is a "failsafe" account, just in case LDAP authentication is broken.

DO NOT allow root to log in over SSH. Doing so is widely accepted as **bad** from a security point of view. You want to see users log in as themselves first and exercise root powers second, so that you know who is doing what.

When disk setup begins, do **not** choose to encrypt the root disk, if asked. Do, however, choose to use the whole disk; this is the default.

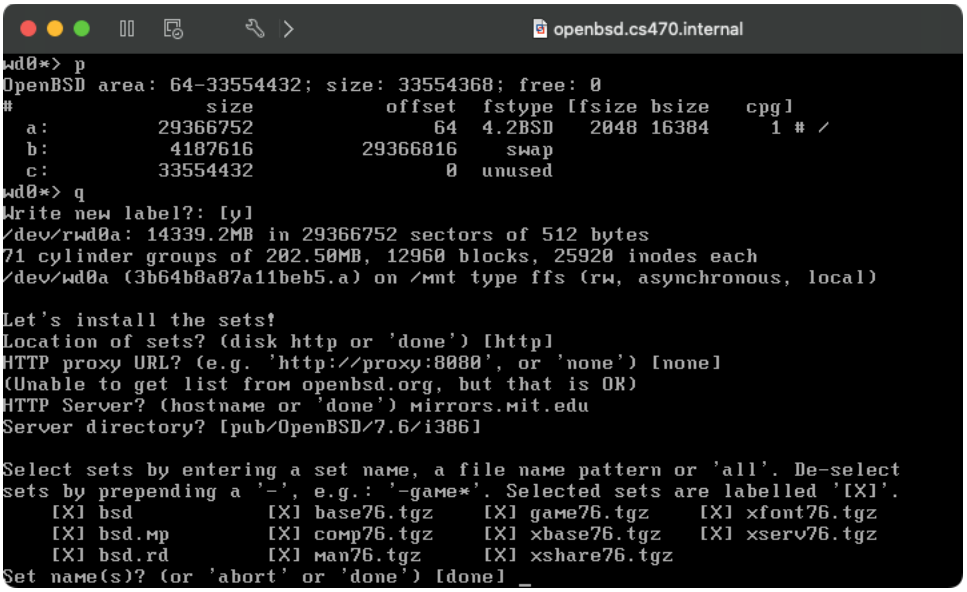
DO NOT USE THE DEFAULT LAYOUT! It uses way too many partitions. We want one.



Hit `c` to create a custom layout. Use `a` to create a partition. Choose partition `a` with the default offset, size `14G`, and mount point `/` … in a BSD, partition `a` is almost always our root filesystem. Use `a` again to create a swap space, partition letter `b`, with the rest of the disk – if you accept the defaults, it will use the rest of the disk. Use `p` to print out the disk label, and make sure it looks similar to the one above.

Use `q` to write out the new label and exit disk configuration.

When you're asked for the "location of the sets," note that you can install from over the network. Here in OpenBSD's installer, that's done by choosing `http` here … please try it now, installing from the internet instead of from local media. Our local media doesn't have the install sets anyways. Remember, no proxy. When asked for a server, choose the hostname of an OpenBSD mirror … for instance, I directed you to mirrors.mit.edu … the installer script should provide the rest.



Don't follow through with installing any sets here, because this was just for a demonstration, like I said earlier. I wanted to prove two points here:

a) The 64-bit Intel architecture is just an extension – more instructions available on 64-bit capable CPUs – of the 32-bit Intel instruction set. 64-bit Intel CPUs will run older 32-bit Intel software just fine, generally.

b) Where virtualization isn't an option, emulation sometimes is or can be. It should only be used as a fallback, because you pay a performance penalty (up to 30-40% sometimes) to pretend to have a different instruction set within a virtual machine. We did this since the beginning of extending support to ARM Macs, because the ARM ports of OpenBSD are still **very** incomplete. As of the April 2025 release of OpenBSD 7.5, we have finally had enough hardware support in the ARM port to run a native VM on ARM Macs.

Go ahead and power off your VM. This is something we'll **never** do again – you always want to gracefully shut down a virtual machine you care about – but we don't care about this VM, as we haven't even installed the operating system. We're going to install something else in a moment here.

5. Go back to the OpenBSD download site from a few steps ago, and get out of the `i386` folder if you're still there … go to the `amd64` folder if your host has an Intel CPU, and to the `arm64` folder if you're on an ARM Mac. In that folder, download the file `cd76.iso` and validate it like we did a few steps ago.

6. We're going to either reset or throw away our VM from the prior few steps now.

   Those of you using Intel CPUs, we're just going to tweak some VM settings:

   - change the boot firmware from legacy BIOS to UEFI
     (on Macs, this is in Advanced, on Windows in Options -> Advanced)
   - delete the VM's old hard disk and create a new 16 GB disk on a SATA or NVMe bus
     (when asked, you can move the old one to the trash or delete the files)
   - insert the ISO file `cd76.iso` into your VM's virtual optical (CD/DVD) drive
   - remove the floppy drive

   ARM Macs: delete your old VM, and create a new one … choose to **virtualize this time**, **not to emulate**.

   - OS: use "other"
   - boot device: choose CD/DVD image, and aim UTM at `cd76.iso`
   - RAM: 2048 MB (2 GB)
     (again, OpenBSD will run with way less than this, but we'll reclaim it later)
   - CPU: the default number of cores
   - storage: 16 GB
   - make sure to select "Open VM Settings"
   - under network, change your NIC to an Intel e1000 and use "Shared Network"
   - under storage, change your storage device interface to NVMe

7. Repeat the installation process from the prior steps, with the following differences:

   - on Intel CPUs, especially on Macs, you may need to play with the window size
     (UEFI enables high-resolution displays)
   - your disk device will probably change from `wd0` to `sd0`; this is okay
   - use the whole disk partitioning scheme still, but choose GPT on Intel CPUs
     if you don't get the choice for GPT on Intel, you forgot something

- use the same custom layout: a 14 GB root disk and the rest to swap, except you have to leave the MSDOS `i` partition there … it's the GPT boot code, and note how its offset puts it first on the geometry of the storage device

Your disk layout should look something **like** the following. It should have the same partitions, but it's okay so long as the offsets and sizes are **similar**.

```
OpenBSD area: 532544-33554399; size: 33021855; free: 0
#                size           offset  fstype [fsize bsize   cpg]
a:           29364415           532544  4.2BSD   2048 16384 12960 # /mnt
b:            3657439         29896960    swap
c:           33554432                0  unused
i:             532480               64   MSDOS
```

When you get to choose "set names" (the components of the operating system you want to install), please make sure **everything** is selected before proceeding.

VERY IMPORTANT: When the installation is over, choose `S` to exit to a shell, do not reboot. We want to explore a bit more.

8. Before starting to explore this post-installation shell, I want you to notice a few things. First, type `df -h` to see the filesystems that are currently mounted. Note that `/mnt` contains the complete system we're going to boot, with the real disk device (`wd0` or `sd0`) that you chose to configure near the beginning of the installer, and its much larger size. If we had set up multiple partitions on our virtual hard disk, they would all be on mount points below `/mnt`, like `/mnt/var` or `/mnt/usr`.

```
openbsd# df -h
Filesystem     Size    Used   Avail Capacity  Mounted on
/dev/rd0a      3.5M    3.2M    211K    95%    /
/dev/sd0a     13.6G    1.9G   11.0G    15%    /mnt
```

The root filesystem is currently the "miniroot" we booted into, a very small subset of what is required for a running system ... just that which is required to **install** the operating system. It's using an `rd` device driver, for a RAM disk, that is, a section of RAM mocked up by the kernel to look like a filesystem. This is done in part because it is small, and read/writable during usage. When the operating system installation miniroot is being loaded from a floppy, where the media is read/write-able but the image of it is compressed, it is thus not writable in this instance. When it is loaded from an optical disk, it's simply not writable. A RAM disk gives us a way around that.

Increasingly, this is happening with flash-based USB devices being used instead, but even then, we are often emulating an optical disk for compatibility's sake, and even then, there isn't a use case for keeping changes to the installer miniroot. After the operating system is installed, the pieces of configuration we want to keep get copied to that operating system instance, on persistent storage. In other, fringe use cases, like repairing an installed operating system instance, all the changes that need to be stored

also get stored on persistent storage. Note how it says "`saving configuration files`" … when we reboot, everything under `/` but not under `/mnt` will be completely gone, as only `/mnt` is on persistent or "non-volatile" storage.

Also note that the root of the installed operating system instance ... doesn't have to be mounted at the top of the active filesystem namespace, especially when it's being worked on. We use another copy of OpenBSD, in order to install OpenBSD.

Also note "`making all device nodes`" and "`relinking to create unique kernel`" before the end of the installation. What is the OpenBSD installer doing there? Web search it, be prepared to tell me later. It's okay if this step fails; just ignore it.

You've either seen me show you the `/dev` file tree of device file abstractions in lecture, or you will see me do so very soon. Many contemporary operating systems mount `/dev` as a virtual filesystem, with contents that are probed and re-generated at every boot and changed as devices and drivers dictate during run-time. Like with a lot of other things, OpenBSD does it the old-fashioned way, and actually writes a `/dev` tree of virtual device files out to a filesystem.

9. Compare the contents of the root directory with that of `/mnt` using the `ls` command… what's obviously different and missing, and why? Just think about it a bit.

10. Type `fdisk sd0` (or substitute the correct name of your "real" virtual storage device from the `df` output above if it's not `sd0`) to peek at the partition table. Note that though you have a swap and root partition, there's only one OpenBSD partition. This is because OpenBSD, like a lot of other operating systems, uses partitions inside partitions to protect and simplify disk contents. FreeBSD calls the outer wrapping partition a "slice."

    To see the contents of the OpenBSD partition/slice, type `disklabel sd0` … do the outputs of these commands look familiar, like tools we were using during the installation process? They should … the OpenBSD installer is just a big shell script, calling the necessary commands, some of them in an interactive mode, to perform their various pieces of the installation process.

    Also note how with UEFI, the EFI system partition is labeled as `i` to make it addressable within OpenBSD's partition nomenclature, which will ignore everything outside the OpenBSD "slice." The EFI system partition contains early-stage boot loaders, and may need to be accessible

11. `chroot` is a really useful command that changes a process' view of the filesystem, by changing where it sees the root of the filesystem name-space. Try getting the manual page for it now.

```
# man chroot
sh: man: not found
```

It's not found because this is the installation miniroot, and it doesn't include a lot of the comforts of the full operating system distribution, like the online manual pages. Just because we don't have a lot of these creature comforts doesn't mean they'd be hard to add ...

```
# PATH=/mnt/usr/bin:$PATH
```

We've added `/mnt/usr/bin` to the head of our PATH. Now try to `man chroot` again. It now finds the binary for `man`, but we still get an error, and this is to be expected ... the linker (`ld.so`) and terminal types database (`/etc/termcap`) aren't part of the miniroot, but there's a more elegant solution I mentioned earlier for seeing the manual, instead of having to fix one thing after another that's not there in the miniroot.

Run …

```
# ls -l /
```

… then type …

```
# chroot /mnt /bin/ksh
```

… and then …

```
# ls -l /
```

… again. The output changed. Where are we? This command tells us to run `ksh` inside a process that has its filesystem root changed to the location that is currently at `/mnt`. Thus, we're looking at the subset of the current file name space ... that we installed on our VM's virtual storage. This is a common way to boot into a maintenance or rescue: boot off something else that works, mount all the filesystems, `chroot` into the right namespace, and fix whatever's wrong.

Note that you can use any command there with `chroot`, not just a shell, and this can be done for many reasons, including for security, to limit what a process can see after it gets started, in case it gets hacked ... but just another barrier.

Now, the entirety of this process and everything started by it are under `/mnt`. Check out the output of `df` as well … note that the kernel still knows all filesystems are there, of course, but your process is only aware of a subset of the filesystem namespace. Now, try to run the `man` command … you get a different error! This is because man is now in a correct and common path in the filesystem namespace, as far as `ksh` is concerned. Type the following `set` command so that your shell's environment knows what kind of terminal you're using, and then `man` should work.

```
# TERM=vt220
# export TERM
# man chroot
```

What I wanted you to get out of this: 1) `chroot` is a great way to go into a full-featured recovery environment from external boot media, and 2) when you use `chroot`, single user mode, or boot media in general, there's always a little bit of manual setup (setting terminal types, preparing devices, and mounting things) required to make the environment useful.

12. Create a temporary file in `/tmp` with the following command.

```
# touch /tmp/testfile
```

We might expect this file to be there after we reboot, no? Let's check on it later.

13. What is your user ID number here? Use the following command to check it.

```
# grep failsafe /etc/passwd
failsafe:*:1000:1000:failsafe user:/home/failsafe:/bin/ksh
```

A reminder about tab completion here, try to remember it whenever it can save you keystrokes. Even though OpenBSD generally has less comfort-inducing functionality, and even though we're in a stripped-down installer mini-root, it is a modern operating system. Living without tab completion would be too much for many modern admins, even OpenBSD-types. Keying in `/e tab pa tab` is all that's required to address `/etc/passwd`, the user database in its common location since the dawn of epoch (Unix) time. Because `/etc` is the only thing that starts with `e` at the root of the filesystem and it's a directory, tab completion even gives you a `/` to save a keystroke descending into the directory.

The user ID # is the third field of the output line above (1000). The second instance of 1000 is your user account's default group ID. What group is 1000?

```
# grep 1000 /etc/group
failsafe:*:1000:
```

OpenBSD has, as many operating systems do nowadays, created a group for your user account so that you can dis-associate your files and processes from other groups in the system, by design.

The second-to-last field is the account's home directory (`/home/failsafe`).

Your default login shell is the final field in the passwd entry above. `ksh` is the Korn shell, the canonical Unix replacement for `sh`, the Bourne shell. Many old-school Unix users

see `bash` (the Bourne-again shell, get it?) and Linux as cheap knock-offs, or abominations. Some on the OpenBSD team are definitely this sort of purist, so `ksh` is the default shell for root and all other users on OpenBSD.

The second field, with a star in it (you almost never hear "asterisk" because of the extra syllables), is an artifact. We used to put a hashed version of the user's password in this field, but `/etc/passwd` needs to be legible by all authenticated users of the system for a variety of reasons, mostly for convenience … it is this file that maps user IDs (which are what's actually stored in memory and on a filesystem) to usernames. Once upon a time, we realized that storing these hashed passwords in a file readable by all users was a Bad Idea™, because any who could read that file could try to "crack" the passwords.

Since OpenBSD focuses on strong cryptography, it uses a very strong algorithm to hash passwords … check out your password hash with the following command:

```
# grep failsafe /etc/master.passwd
```

`/etc/master.passwd` can be keyed in as `/e tab mas tab` … and I'll stop reminding you about filename completion here and now. It's up to you now, to remind yourself to work smarter, not harder.

`/etc/master.passwd` is the "shadow" password database that contains the actual hashes. BSD-based operating systems typically use this filename, with the exception of macOS, which uses its own user information file format (under `/var/db/dslocal` if you are curious). AT&T-derived Unix OSs typically use `/etc/shadow` … as do most Linux distributions.

14. Run `ls -l /home` to check out its contents, and note how `ls` is aware of the names for UID and GID 1000. Where does it get this information from? Hint: you just saw it.

15. Now type `exit` to exit the `chroot` shell, and run `ls -l /mnt/home` … it's no longer aware of the names for those user IDs. Why not?

16. Type `halt` to tell your VM to gracefully spin down, then use your hypervisor's UI to power off the VM. Finally, use your hypervisor's UI to disconnect the installation ISO from your VM's virtual drive.

    Note that you will always tell your VMs to shut down. Never use the hypervisor (VMware/UTM) buttons or menus to shut down the VM, unless you absolutely **have to**, because of a hang or crash, or until your VM tells you it's safe to power down.

```
openbsd# halt
syncing disks… done

The operating system has halted.
Please press any key to reboot.
```

This means it's safe to shut down the VM. All Unix and Unix-like OSs make heavy use of caching, and file systems may go corrupt if you do not clearly shut down the machine, taking your hard work with them.

If your lab machine is a laptop, generally it's okay to close your laptop, but note that your VMs' clocks may go askew, as they cannot talk to power hardware that tells them they went to sleep with your host operating system.

!! IMPORTANT REMINDER: Reminder from the first lecture, in every lab, always read each step, digest it, understand it, and only then, do it. Read the next step before hitting a key to reboot your VM or powering it back up, and be ready to intervene right away.

17. At the `boot>` prompt, let's first take care of having to re-size your window for those of you on Intel/AMD CPUs. Continually resizing your VM's console window will be painful, as we're stuck here for a little bit, and may need to use it again from time to time.

    As with a lot of situations, you have choices … you can either continue to click and drag the window's corners for a temporary fix, or you can do a web search and find the right entry point into the manual pages, and a more automated way of doing this.

    As the manual page for the boot loader shows, the commands `machine gop` and `machine video` can be used to provide lists of supported settings for the resolution of the screen (`gop`) and the number of rows and columns of text (`video`). I personally prefer `gop 23` and the default `video 2`. These commands are not supported on ARM64.

    Now type `-s` to boot in single user mode.

    I want you to notice the lines roll by as the kernel boots; this is it detecting the virtual machine's CPU, system buses, and hardware.

    The kernel will detect all your hardware, and then you'll be prompted:

    ```
    Enter pathname of shell or RETURN for sh:
    ```

    Hit return for `sh`. We're now in single-user mode, commonly used for maintenance and rescue. It's called single-user mode because no services are enabled to allow another user to log in. It's just you, on the console.

    At the prompt, type `dmesg` to show the hardware inventory again. It goes by pretty fast, but please note that you can hold down `shift` with the `page-up` and `page-down` keys (on a Mac laptop keyboard, that might be `function-shift-up` and `function-shift-down`) to scroll in the console terminal. You have a very limited scroll-back buffer here in

console mode, which is just one of many reasons why we'll want to SSH into our new VM as soon as possible, to make use of a more flexible and powerful software terminal emulator. Those of you on ARM Macs are using a virtual serial console, and won't have to use funny key combinations at all to scroll backwards.

Note how `dmesg` spells out the CPU, details of storage devices, network hardware, keyboard, and everything. This echoes the output of the OpenBSD kernel as it enumerated detection of all hardware in our virtual machine at boot time. Often times these details can be useful, or help shine light on conflicts or misconfigurations.

18. Type `mount` to list all mounted filesystems, and note that the root filesystem is mounted read-only.

    We might want to be here to do things to our operating system instance without allowing logins, or allowing users to be able to access it as we're making these changes, among many other reasons. The root is mounted read-only at first so that you can repair it if it's corrupt.

    If we feared filesystem corruption, or if we were booting up this machine in single-user mode to try to address disk problems, odds are we'd want to check the integrity of the filesystems right away. The utility used to do this on most operating systems is `fsck`, shorthand for "filesystem check."

    `cat /etc/fstab` to see the table of all filesystems on this machine. This path is also common between all BSD and Linux systems we'll use in this class.

```
# cat /etc/fstab
85d40c0352eae142.a none swap sw
85d40c0352eae142.b / ffs rw,wxallowed 1 1
```

    First things first, notice how OpenBSD is using a unique ID (we often use UUIDs) of sorts to identify the filesystems, rather than the actual device file (the `/dev/sd0` we saw during installation). This is in case we have multiple physical disks, so that we choose the right one, regardless of the order they enumerate in. We'll come back to this.

    The Berkeley version of the Unix file system (UFS) is called FFS, short for fast system, after a huge number of optimizations made by Kirk McKusick at UC Berkeley to reduce fragmentation, increase filesystem resilience, and introduce the ability to background filesystem checks. A lot of the work done at Berkeley to improve the file system was absorbed by other vendors, many of whom use "UFS" and "FFS" interchangeably. You can read more about UFS and FFS here, and we'll be back to talking more about filesystems checks shortly as well.

    Also note the numbers to the right of the root filesystem. The first is to denote how often a filesystem should be backed up. The second is to denote the order in which

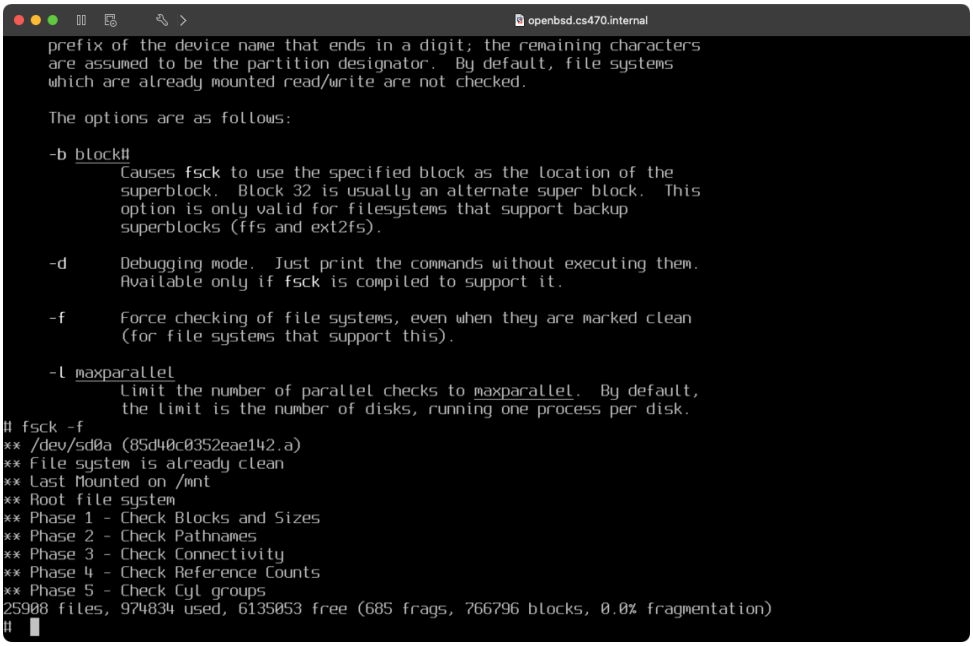
filesystem checks are done with `fsck`. The root filesystem should go first for what should be obvious reasons, even when it's not the only filesystem in a machine.

Run `fsck`. If your VM shut down cleanly, you should see something like this.

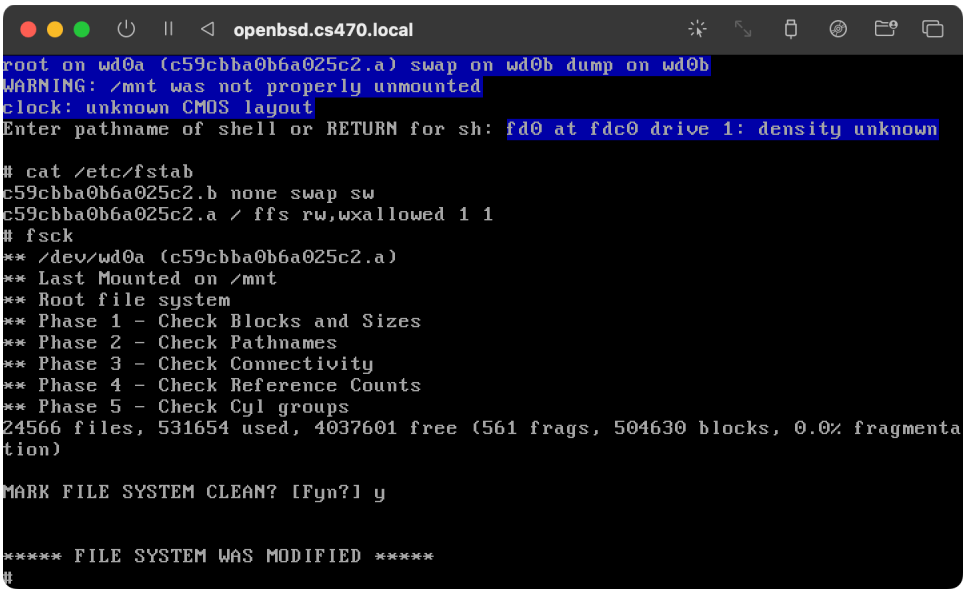
```
# fsck
** /dev/sd0a (85d40c0352eae142.a)
** file system is clean; not checking
```

… `fsck` is saying that it can tell the filesystem was unmounted properly the last time – there's a "clean" bit that is set and can be checked – and it does not need to run. There's typically an `-f` switch we can use to force it to do a check anyways.

As you can see in the below screenshot, I performed the same terminal setup from a few steps ago so I could use `man` with a pager, and verified that `-f` does, indeed, exist on OpenBSD's implementation of `fsck`.



The below screenshot is from my OpenBSD VM on my ARM Mac, also taken during a prior class, and it (intentionally) was not shut down cleanly for the purposes of demonstrating this to you. As you can see in the text near the top, `WARNING: /mnt was not properly unmounted`. There's a bit called "the clean bit" which is set when a filesystem is cleanly unmounted, and it wasn't set, so that's how it knows. Also, the filesystem is the root filesystem now, but as we can see in the output, the filesystem knows it was last mounted on `/mnt` and that's why OpenBSD reports it to us this way.



Note how `fsck` does not object to being run, because the clean bit was not set here, and how it asks if we want to set it afterwards. If `fsck` finds anything to fix, it will generally ask for each problem, if you want it fixed. For this reason, there's also commonly a `-y` flag that can be used to tell `fsck` to just go ahead and fix everything it can.

Had I not booted in single-user mode, OpenBSD, like most Unix-like OSs today, would automatically run `fsck -y` on any filesystems not marked clean during the boot process.

19. Now we're going to make some changes, so mount the root filesystem read-write with the following command.

    ```
    # mount -o rw /
    ```

    Type `mount` again to see that it's mounted read/write. It won't say read/write, it just won't say read-only, as read/write is the normal state.

20. Use `ls -l /tmp` to see if your `testfile` is still there. It should be. If it isn't, why not? Did you mess up getting into single-user mode on the first boot of your new virtual machine?

21. Use `cat` to check out the contents of the file `/etc/syslog.conf` … this is where logging is configured. We'll be back here later in the labs.

    Also, note that `cat` doesn't require the terminal to be configured – we do that in the next step – whereas a paging file viewer like `more` does. This can be handy to save a

couple steps during maintenance in single-user mode.

22. Try to `vi /etc/ttys` … and unless you already set up the terminal environment, you should have the same problem with terminal types as before, in the post-install shell. Go ahead and set it up the terminal environment as we did before for manual pages.

    ```
    # TERM=vt220
    # export TERM
    # vi /etc/ttys
    ```

    These steps will need to be replicated **each time** you boot into single-user mode or off the installation media, in order to use a lot of commands like `man` and `vi` that require knowledge of your terminal type and settings.

    `vi` will complain that it can't find `/tmp/vi.recover` … you can ignore this.

    Now, find the line starting with `console` in `/etc/ttys` and change `secure` at the end of that line to `insecure` and save the file. Note that single user mode didn't ask us for a password to become root … the idea here was, if your hardware was in a public place (`insecure`), you wouldn't want somebody to be able to power-cycle it or reboot it and become root.

    Also, if you're running on Intel/AMD, use `vi` to edit and create the file `/etc/boot.conf` and enter any video mode settings you'd prefer over the defaults a few steps ago.

    Reboot into single-user mode again to make sure your video mode setting works, and to verify that you are asked for the root password in order to get into single-user mode.

23. Nearly all Unix-like operating systems implement `shutdown` as a primary tool for changing the power state of your computer. In nearly all implementations of `shutdown`, `-h` halts the computer but does not power it off (macOS is an exception to this, and powers off with `-h`) and `-p` powers off the machine (often capital `-P` is used on Linux).

    The `shutdown` utility typically touches the file `/etc/nologin`, which is checked by most services that allow a login to occur. If the file exists, logins are rejected, until the file is removed after a full multi-user boot.

    `shutdown` also typically takes, or requires, a time until it commences. On a larger system with many users, this may be required to give users a grace period to save files and work before the system becomes unresponsive. During that grace period, `shutdown` will repeatedly send messages to each terminal that's logged in, to remind users to save their data and log out. If you have no users on your system, as with most of us, either `0` or `now` can be used to immediately shut down.

    Virtually all modern systems implement a utility called `reboot` that instantaneously

reboots the machine, unless modified with switches and arguments to do it on a delay, or with warning. Virtually all canonical AT&T Unix-like systems and Linux systems implement a utility called `poweroff` that initiates a hardware power-off of the machine. BSD systems generally do not implement `poweroff`.

In order to avoid filesystem corruption and a loss of work on your labs, whenever you intend to shut down one of your VMs, you should always run `shutdown` or `poweroff` before you try closing its window in VMware or UTM.

Reboot your OpenBSD VM with `shutdown -r now` …

This time, let's **not** boot into single-user mode. You can either hit `return` at the boot loader prompt, or let it time out (five seconds or so) to perform a normal, multi-user boot. Once it's done booting, log in as `failsafe`.

24. Type `ifconfig -a` to inspect all your network interfaces. Each is identified by a device type name (for instance, for your virtual loopback adapter, `lo`) and a number (`0`). Each device type is associated with a kernel device and driver for which there is a `man` page. Use `man` to determine which are pseudo-devices (virtual network interfaces in software) and which is your VM's "real" network interface, in case you didn't retain this information during installation … besides, it's good to explore.

    Note the IP address ("`inet`") of your VM's "real" network interface. It's divided up (by the dots) into four "octets." It should be of the form 192.168.x.y or 172.x.y.z or 10.x.y.z.

    Also note the number of bits in your network mask. People using VMware will almost certainly have 24-bit (`0xffffff00`) netmasks, but some of you using UTM may have 16-bit (`0xffff0000`) netmasks. **Keep track of what you have to use again later.**

25. As we said in the prologue to this lab, DHCP configures an IP address, netmask, DNS, and a default gateway … the four things you need to set up an internet connection. Check out the contents of the file `/etc/resolv.conf` (the configuration of the DNS client, or "resolver") with the `more` command. The IP address after "nameserver" … this is the IP address of the DNS server provided by your desktop hypervisor. Note it.

    Mine was 10.42.77.2, but it has moved back and forth between .2 and .1 in between previous versions of VMware. This file won't get blown away when we reboot, so we can leave it there.

26. Most servers are statically addressed, because we want to find them in the same place all the time. Whatever your virtual machine's "real" ethernet interface was named (probably `em0`), you should find a file with the name `/etc/hostname.if`, where "`if`" is the name of your network interface (don't forget the number). Type `man hostname.if` to find out what the expected syntax of this file should be.

We are going to stand up a static network with all our hosts – you're starting to see the method behind the madness here, I hope – with DNS for all our hosts. We will all be using the same final octets, listed below, for each of our hosts.

OpenBSD	openbsd.cs470.internal	.71
FreeBSD	freebsd.cs470.internal	.72
Rocky	rocky.cs470.internal	.73
Ubuntu	ubuntu.cs470.internal	.74
Solaris	solaris.cs470.internal	.75
Proxmox	proxmox.cs470.internal	.76
Kubernetes	k8s.cs470.internal	.78

Use the `su` command to become root from your failsafe user account. Note: `su` expects the password of the user you're becoming, in this case, root. After that, you can edit `/etc/hostname.if` with `vi`.

My network was 10.42.77.0/24, so I put this …

```
inet 10.42.77.71 255.255.255.0
```

… into the file `/etc/hostname.em0` (swap in the name of your network interface) **replacing everything that was previously there** (including the `autoconf` setting we entered during installation). You should also end your IP with .71, but you should make sure you keep the same first three octets of your IP address as your hypervisor's shared network is using, as well as the netmask, that you got from `ifconfig`.

27. DHCP set up all the details of our network previously. `/etc/resolv.conf` handled DNS resolution, `/etc/hostname.*` our network interface configurations … the only thing that's left is setting up a default gateway, or default router. In OpenBSD, this is handled by `/etc/mygate`. Edit this file – create it if it doesn't yet exist – and put **only** the IP address of the default router you got from DHCP. You can get this IP address by running the following command. Remember, the pound sign prompt means you're still root here.

    ```
    # netstat -rn | grep default
    ```

    The `netstat` command is a very handy network tool, and we'll be coming back to it repeatedly. For now, the `r` switch tells it we want the routing tables, and the `n` switch, as with many commands, tells `netstat` not to use DNS or anything to resolve names.

    After you've put your default router IP address into the file, use `more` to make sure it's got what you think.

‼ IMPORTANT NOTE: very few, if any, services ever watch configuration files proactively, and the network stack is no exception. These files feed data into variables when they are parsed by shell scripts that configure the network stack when the OS boots. After you change the contents of these files, you either have to manually reconfigure the network stack … *or you reboot to test your configuration and reconfigure the network*.

28. Let's do just that … `reboot` and make sure your network configuration works.

    !! YET ANOTHER IMPORTANT NOTE OR THREE: this is the part of the movie where we test our internet connection. Throughout the labs, when we set up a new service, we test it afterwards. I won't always say "we're testing!" because you should be able to tell, if you're paying attention. Whenever you set up a service the next time, on another system, look back to how you've tested it in the past.

    In the last few tasks, we set up static network addressing on your OpenBSD VM. If everything's fine after the reboot, you should still be able to access the internet. Try running `ping 4.2.2.2` and `ping 1.1.1.1` and `host www.openbsd.org` to test IP and DNS service, and make sure you still have functioning internet access.

    If your network doesn't work by IP, `ping` your default gateway, and make sure that works. If it doesn't, you have a problem with your IP address or netmask. If pinging your gateway works, but not 4.2.2.2 or 1.1.1.1, you have a problem with your routing/gateway configuration. If name to IP address doesn't work, your problem is either in `/etc/resolv.conf` or your name server.

    We haven't set up a name server, not yet at least, so you don't get to blame that yet. Your hypervisor (VMware or UTM) is playing the role of a DNS server for your VM at this point still … it helped us find the OpenBSD installation sets online, and we'll still be using it for a bit.

part two: installing the source packages, the ports tree, and third-party software

Next, we're going to extract the `ports`, `src`, and `sys` trees to `/usr/ports`, `/usr/src`, and `/usr/src/sys`, respectively. You can find these as tarballs in the same directory on the webserver where you found your installation ISO.

The `src` and `sys` trees are the userland and kernel sources for OpenBSD, respectively. The ports tree is full of automation for building thousands of third-party software packages, from source.

Historically, many have had their VMs hang or run out of RAM ("`LLVM ERROR: out of memory`") while building ports in this section of the lab. This is why we now start with 2 GB of RAM in our OpenBSD VMs, though it can run in much less. If you get out-of-memory (OoM)

errors, please feel free to shut down your VM and bump the VM's RAM a bit. After building all software in this section successfully, you can shut the VM down again and bump down the amount of RAM it uses back down.

29. There should already be a ~/.ssh directory inside your failsafe user's home directory. Copy your SSH public key into .ssh/authorized_keys from your host operating system.

    After you do this, you won't have to use password-based authentication with your VM. For each of your VMs moving forward, we're going to always repeat this setup step.

    Since we don't have DNS working yet for cs470.internal, you will have to use your OpenBSD VM's IP address as the target host in your scp command. No DNS server out there (yet!) can tell us how to get to openbsd.cs470.internal. That domain, if it exists out there, isn't connected to the public DNS namespace (.internal is not a valid TLD), so we are going to use it.

    This will make copying and pasting into command lines on your VM a **whole** lot easier, as you can minimize the VM, and VMware, and just log into your VM with a terminal window from your host operating system … using an easy-to-remember hostname.

    Log into it as your failsafe account, over SSH. Not only do you no longer have to type a password, but you're also able to open multiple terminal windows on your host, and log into your VM with multiple shells to get more done.

30. Now let's install the source code for the operating system. First, the "base" source tarball, with the "userland" (non-kernel) sources.

    If we unpack it in the wrong place, we're going to be in a world of hurt, and probably need to re-install our OpenBSD VM. So, first, let's look inside the tarball. There's a really handy way to look inside a tar archive rather than extract files, using the t switch of the tar command … "t" is short for "table of contents." Try this to see where src.tar.gz file will unpack …

    ```
    $ ftp -o - https://cdn.openbsd.org/pub/OpenBSD/7.6/src.tar.gz | tar
    ztvf - | head
    ```

    ‼ IMPORTANT NOTE: the above is a single-line command, wrapped. Don't copy/paste. Start developing muscle memory for some of these commands.

    … the head command, by default, grabs only the first few lines of the output. The -o switch chooses where output goes. I found this option, and how to send output to stdout with, you guessed it: man ftp …

Have problems with the OpenBSD mirror? Look up another mirror online, and just change your mirror!

You will see the output of this command wait before giving you back a command line; this is because it's waiting for `ftp` to receive an end of file (EOF) before all the processes in the pipe stop. You can hit `control-c` rather than wait here.

You will notice that actually unpacking `src.tar.gz` is going to make directories under wherever it's working from. It doesn't create a top-level `src` directory when it unpacks, therefore it expects to be unpacked when you're already in `/usr/src`, which is created for you during installation on OpenBSD. **This command is going to take a long time. Don't run it and expect to be able to move your laptop in between locations**, as we're grabbing it from the network and unpacking it, all at once.

Also note, the pound sign prompt below means you need to be root to run this command. Again, use the `su` command to become root from your regular user account.

```
# cd /usr/src && ftp -o -
https://cdn.openbsd.org/pub/OpenBSD/7.6/src.tar.gz | tar zxvf -
```

‼ IMPORTANT NOTE: the above is a single-line command, wrapped. Don't copy/paste.

You should remember the logical AND from lab zero.

Pretty cool, though, how we're using pipes to download and extract `src.tar.gz`, while NOT saving a copy of it, no?

31. `sys.tar.gz` is the tarball with the source tree for the OpenBSD kernel. Let's do the same thing to see if it's going to make its own directory …

```
$ ftp -o - https://cdn.openbsd.org/pub/OpenBSD/7.6/sys.tar.gz | tar
ztvf - | head
```

‼ IMPORTANT NOTE: the above is a single-line command, wrapped. Don't copy/paste.

… and it turns that it also expects to be unpacked under `/usr/src`, and creates a `sys` directory there, as you'll see with the command above.

```
# cd /usr/src && ftp -o -
https://cdn.openbsd.org/pub/OpenBSD/7.6/sys.tar.gz | tar zxvf -
```

‼ IMPORTANT NOTE: the above is a single-line command, wrapped. Don't copy/paste.

32. And finally, `ports.tar.gz`, the "ports tree" full of automation for building third-party software from source code …

```
$ ftp -o - https://cdn.openbsd.org/pub/OpenBSD/7.6/ports.tar.gz | tar
ztvf - | head
```

‼ IMPORTANT NOTE: the above is a single-line command, wrapped. Don't copy/paste.

… which will make a `ports` directory, as the command above shows. Note that you won't find a `ports` directory under `/usr`, so change directory to there and unpack it … but we don't want to install that copy of the ports tree from the mirror. We want a *version-tracked* copy of the ports tree we can keep up to date, so that we build the latest versions of third-party software we want.

OpenBSD keeps three branches of the source code for each version of the operating system, including the ports tree:

- -release, as in 7.6-release: the original release of each version
  this gets frozen in time on the release day of each version of the OS
- -stable, which includes patches issued for each version
  "stable" means it should compile without issue into presumed-stable binaries
  OpenBSD 7.6's patches are here: https://www.openbsd.org/errata76.html
- -current: this is the unstable, under-development branch
  it will become 7.7-beta just before the next version of the OS is released

See https://www.openbsd.org/faq/faq5.html for more information and https://www.openbsd.org/anoncvs.html for more information about using CVS to grab source. To install the "stable" version of the ports tree …

```
# cd /usr && cvs -qd anoncvs@anoncvs1.usa.openbsd.org:/cvs checkout
-rOPENBSD_7_6 -P ports
```

‼ IMPORTANT NOTE: yet again the above is a single-line command, wrapped.

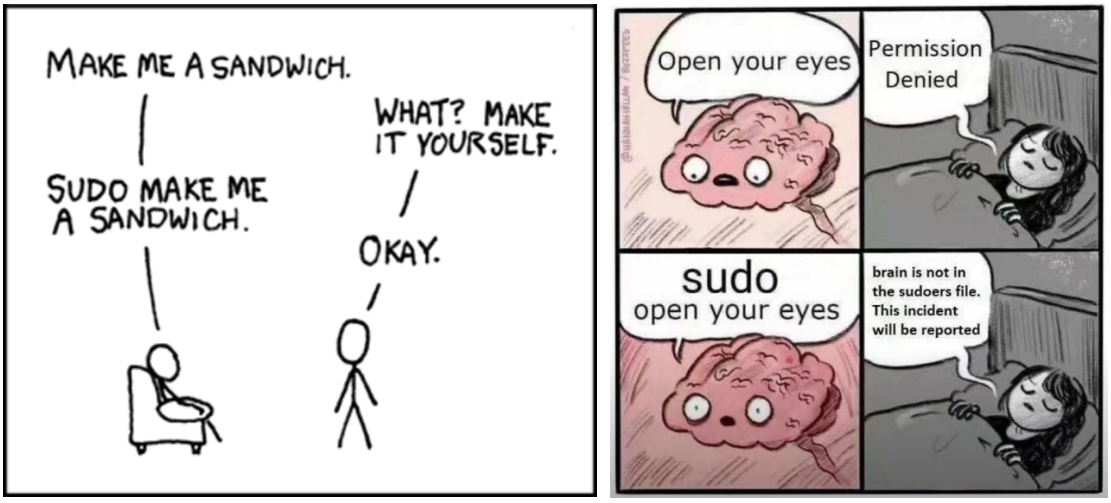
You'll notice that the first time you run this command, you're prompted to accept the host key for anoncvs1.usa.openbsd.org. This is because `cvs` is using `ssh` to provide transport security. Those of you who have done some software development may also recognize that we're now using CVS, a common source code versioning system, to grab and check out an initial version of the ports tree. This should be unsurprising, as the ports tree is a huge pile of shell scripts.

CVS will also be used to keep the ports tree up to date, we'll set up a periodic task to automate this later in this lab.

Now let's use the ports tree to install `sudo`, `tcsh`, GnuPG (either version 1 or 2 is fine), and ISC-BIND, the Internet Software Consortium's Berkeley Internet Name Daemon. ISC-BIND will provide a name server, the first major service target in our labs, and a requirement for other services to be found by name.

First, `ls /usr/ports` and note how the ports tree is broken up into categories. Individual software packages can be found in directories under each of these.

33. The most popular tool for administrative privilege escalation and user-switching is `sudo`.



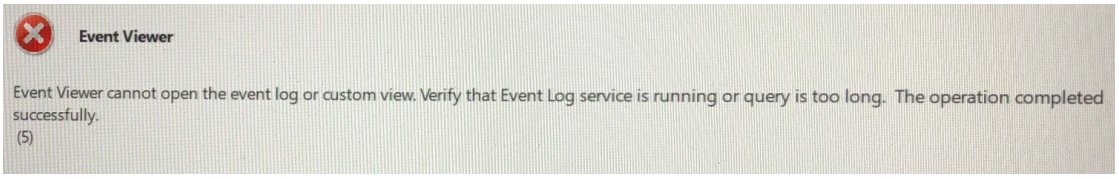
Even though alumni of the OpenBSD project wrote `sudo`, it's acquired a perception amongst a lot of the OpenBSD team as a bloated piece of software, so OpenBSD now has its own built-in replacement, `doas`. There's nothing wrong with `doas`, but let's get `sudo` here for uniformity, since we're going to be using it on every other VM.

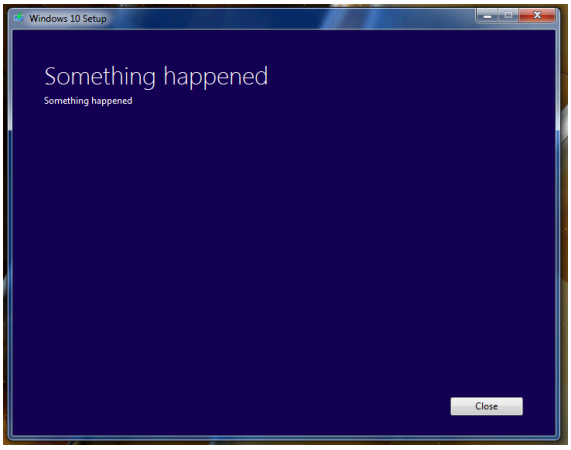
```
# cd /usr/ports/security/sudo && make install
```

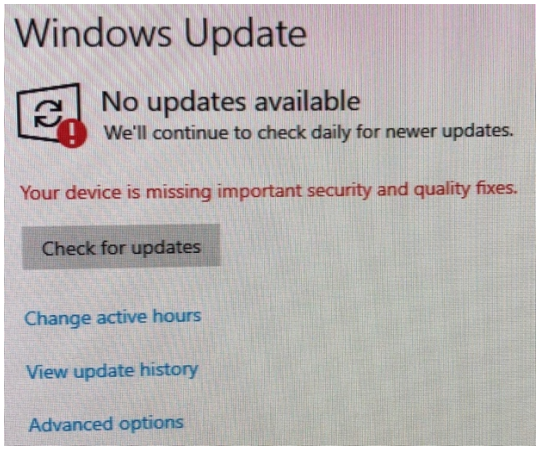
It might be okay for some people to make a symbolic link from `sudo` to `doas` – this is like a shortcut in Windows or an alias in macOS, since we haven't covered it in lecture yet – but this could be very bad if software sees that `sudo` is there, and tries to treat `doas` like `sudo` … the options and syntax for the two commands are generally not compatible.

The command above is going to build `sudo`. In order to get there, since we're building it from source, it's going to have to build a lot of dependencies first, other pieces of software that `sudo` is built on top of. This is going to take a little while, and we're going to use this as an excuse to talk about building software on Unix/Linux, and how this is done using the ports tree in BSD.

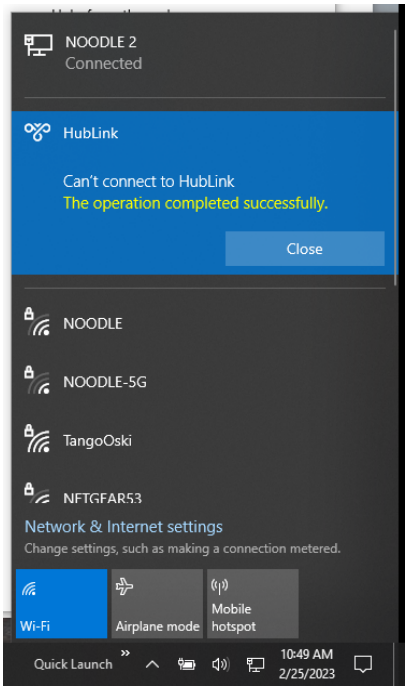
Many of us, especially those who became familiar with computers via Windows, have grown accustomed to the curse of bad error messages. Microsoft is famously really good at really bad error messages, and since we don't give them enough bad press here, all of these are screenshots I've taken myself personally …







I'm **not** going to say that you're not going ever to have the same problem in Unix or Linux – there are a multitude of developers from all over the world in in this space, with varying degrees of coding and communications and writing skills, and you will run into bad messaging every now and again … but nothing like this!



What I am going to say: **listen to what your computer is saying to you, all the time**. Learn not to panic, and to trust in what it is telling you, and read into it. Many students

go off the rails and create issues for themselves in troubleshooting, playing guessing games instead of simply dissecting what their computers are telling them.

Very few meaningless things are going to be printed out to your terminal. You don't have to watch the entire time it builds `sudo` and all its dependencies, but don't totally tune out.

‼ IMPORTANT NOTE: if you have a problem building a port, scroll back UP in your terminal session to find the root cause of the problem. The root cause is probably the very first of any error messages that appear, not the last. This is yet another huge reason why we prefer to SSH into our VMs to do things, unless we absolutely can't: the much larger and easier-to-search scroll-back buffer in software terminal programs.

Let's look at what's going on here when we tell the ports tree to install `sudo` … first, the ports tree checks to see if it has the files to build the port. When it finds that it doesn't have them, it goes and grabs them ….

```
===>  Checking files for sudo-1.9.15p5
>> Fetch https://www.sudo.ws/dist/sudo-1.9.15p5.tar.gz
sudo-1.9.15p5.tar.gz 100% |***********************************|  5182 KB    00:00
>> (SHA256) sudo-1.9.15p5.tar.gz: OK
```

… and just like we did with the OpenBSD installation media at the beginning of this lab, the ports tree does a SHA256 hash of the source tarball to verify, as best as possible, that it's the same source code they've extended support to. Different source might not work with local-to-OpenBSD patches provided by OpenBSD, and this also (somewhat) helps to prevent us from getting backdoored source. `xz` is in the dependency chain for `sudo`, so this is legit … if you've been reading the news lately.

After verifying the hash, the ports tree then proceeds to check for the presence of dependencies, other pieces of software, required to build `sudo` …

```
===> sudo-1.9.15.5p0 depends on: python->=3.11,<3.12 - not found
===>  Verifying install for python->=3.11,<3.12 in lang/python/3
===> Building from scratch Python-3.11.10
===>  Checking files for Python-3.11.10
>> Fetch https://www.python.org/ftp/python/3.11.10/Python-3.11.10.tgz
Python-3.11.10.tgz 100% |***********************************************|
25910 KB    00:04
>> (SHA256) Python-3.11.10.tgz: OK
===> python-3.11.10p1 depends on: metaauto-* - not found
===>  Verifying install for metaauto-* in devel/metaauto
===> Building from scratch metaauto-1.0p4
===>  Checking files for metaauto-1.0p4
>> Fetch https://pkg-config.freedesktop.org/releases/pkg-config-0.29.2.tar.gz
pkg-config-0.29.2.tar.gz 100% |***********************************************|
1969 KB    00:00
>> (SHA256) pkg-config-0.29.2.tar.gz: OK
===>  Extracting for metaauto-1.0p4
```

… the port of `sudo` depends on `python`, version 3.11 or greater, less than version 3.12, which isn't found, so it downloads and verifies the source for `python`, which in turn requires `metaauto`, so it downloads and verifies that project's known-good source ball.

This process recurses for each port, enumerating and grabbing missing dependencies for each port, until it finally hits a leaf node: a port with no unsatisfied dependencies, at which point the ports tree builds that project from source.

`metaauto` is just a collection of interpreted scripts for automating software builds, so it's both not a surprise that it's an early leaf node when building ports … and it's a really bad example of a port, since no compiler is involved. I'll dissect an example port build by looking at the aforementioned `xz` being built. Those of you smart enough to follow the links above will note: we're not building the backdoored version.

A typical port build starts with the ports tree patching the downloaded sources with local customizations for OpenBSD and better integration with other ports.

```
>> (SHA256) xz-5.4.5.tar.gz: OK
===> xz-5.4.5 depends on: dwz-* -> dwz-0.15
===>  Verifying specs:  c pthread
===>  found c.99.0 pthread.27.1
===>  Extracting for xz-5.4.5
===>  Patching for xz-5.4.5
===>   Applying OpenBSD patch patch-config_h_in
Hmm...  Looks like a unified diff to me...
The text leading up to this was:
--------------------------
|Index: config.h.in
|--- config.h.in.orig
|+++ config.h.in
--------------------------
Patching file config.h.in using Plan A...
Hunk #1 succeeded at 409.
done
===>   Applying OpenBSD patch patch-src_xzdec_xzdec_c
Hmm...  Looks like a unified diff to me...
The text leading up to this was:
--------------------------
|Index: src/xzdec/xzdec.c
|--- src/xzdec/xzdec.c.orig
|+++ src/xzdec/xzdec.c
--------------------------
Patching file src/xzdec/xzdec.c using Plan A...
Hunk #1 succeeded at 295.
done
```

`xz` now building version 5.6.2 for me as I type this, but I'm keeping the older output here because what it shows is still valid. You can plainly see from the older 5.4.5 output above that the ports tree patched a configuration header file and a C source file. This is fairly common – to only need to alter a couple of files – because the next step is to typically to configure the source tree …

```
===>  Generating configure for xz-5.4.5
===>  Configuring for xz-5.4.5
Using /usr/ports/pobj/xz-5.4.5/config.site (generated)
configure: WARNING: unrecognized options: --disable-gtk-doc
configure: loading site script /usr/ports/pobj/xz-5.4.5/config.site

XZ Utils 5.4.5

System type:
checking build system type... x86_64-unknown-openbsd7.6
checking host system type... x86_64-unknown-openbsd7.6
```

```
Configure options:
checking if debugging code should be compiled... no
checking which encoders to build... lzma1 lzma2 delta x86 powerpc ia64 arm
armthumb arm64 sparc
checking which decoders to build... lzma1 lzma2 delta x86 powerpc ia64 arm
armthumb arm64 sparc
checking which match finders to build... hc3 hc4 bt2 bt3 bt4
checking which integrity checks to build... crc32 crc64 sha256
checking if external SHA-256 should be used... yes
checking if MicroLZMA support should be built... yes
checking if .lz (lzip) decompression support should be built... yes
checking if assembler optimizations should be used... no
checking if small size is preferred over speed... no
checking if threading support is wanted... yes, posix
checking how much RAM to assume if the real amount is unknown... 128 MiB
checking if sandboxing should be used... maybe (autodetect)

checking for a shell that conforms to POSIX... /bin/sh

Initializing Automake:
checking for a BSD-compatible install... /usr/ports/pobj/xz-5.4.5/bin/install -
c
checking whether build environment is sane... yes
checking for a race-free mkdir -p... mkdir -p
checking for gawk... awk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether ln -s works... yes
checking whether make supports the include directive... yes (GNU style)
checking for gcc... cc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether the compiler supports GNU C... yes
checking whether cc accepts -g... yes
checking for cc option to enable C11 features... none needed
checking whether cc understands -c and -o together... yes
checking dependency style of cc... gcc3
checking dependency style of cc... gcc3
checking for stdio.h... yes
checking for stdlib.h... yes
checking for string.h... yes
…
```

… I use the ellipsis (dots) here to show that the output is truncated, because I've shown you enough example output. When you see a bunch of lines like this, starting with "checking for," this means the source package is using `autoconf` to help facilitate support for multiple versions of multiple platforms. `autoconf` is a collection of shell scripts used to check local APIs and define compile-time macros to activate the right support for the current target platform.

With that done, you will now typically see `make` get called to oversee the build process, followed by a bunch of invocations of `cc` or `clang` or whatever compiler is appropriate for the language used by the source, as the ports tree now actually compiles the software.

```
===>  Building for xz-5.4.5
make  all-recursive
Making all in src
Making all in liblzma
Making all in api
```

```
/usr/bin/libtool  --tag=CC   --mode=compile cc -DHAVE_CONFIG_H -I. -
I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma -I../..  -I/usr/ports/pobj/xz-
5.4.5/xz-5.4.5/src/liblzma/api  -I/usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/common  -I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/check
-I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/lz  -I/usr/ports/pobj/xz-
5.4.5/xz-5.4.5/src/liblzma/rangecoder  -I/usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/lzma  -I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/delta
-I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/simple  -I/usr/ports/pobj/xz-
5.4.5/xz-5.4.5/src/common  -DTUKLIB_SYMBOL_PREFIX=lzma_  -pthread -
fvisibility=hidden -Wall -Wextra -Wvla -Wformat=2 -Winit-self -Wmissing-
include-dirs -Wstrict-overflow=3 -Wfloat-equal -Wundef -Wshadow -Wpointer-arith
-Wbad-function-cast -Wwrite-strings -Wdate-time -Wsign-conversion -Wfloat-
conversion -Waggregate-return -Wstrict-prototypes -Wold-style-definition -
Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -Wc99-compat -
Wc11-extensions -Wc2x-compat -Wc2x-extensions -Wpre-c2x-compat -Warray-bounds-
pointer-arithmetic -Wassign-enum -Wconditional-uninitialized -Wdocumentation -
Wduplicate-enum -Wempty-translation-unit -Wflexible-array-extensions -Wmissing-
variable-declarations -Wnewline-eof -Wshift-sign-overflow -Wstring-conversion -
O2 -pipe -g -MT liblzma_la-tuklib_physmem.lo -MD -MP -MF .deps/liblzma_la-
tuklib_physmem.Tpo -c -o liblzma_la-tuklib_physmem.lo `test -f
'../common/tuklib_physmem.c' || echo '/usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/'`../common/tuklib_physmem.c
cc -DHAVE_CONFIG_H -I. -I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma -I../..
-I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/api -I/usr/ports/pobj/xz-
5.4.5/xz-5.4.5/src/liblzma/common -I/usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/check -I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/lz -
I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/rangecoder -I/usr/ports/pobj/xz-
5.4.5/xz-5.4.5/src/liblzma/lzma -I/usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/delta -I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/liblzma/simple
-I/usr/ports/pobj/xz-5.4.5/xz-5.4.5/src/common -DTUKLIB_SYMBOL_PREFIX=lzma_ -
pthread -fvisibility=hidden -Wall -Wextra -Wvla -Wformat=2 -Winit-self -
Wmissing-include-dirs -Wstrict-overflow=3 -Wfloat-equal -Wundef -Wshadow -
Wpointer-arith -Wbad-function-cast -Wwrite-strings -Wdate-time -Wsign-
conversion -Wfloat-conversion -Waggregate-return -Wstrict-prototypes -Wold-
style-definition -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls
-Wc99-compat -Wc11-extensions -Wc2x-compat -Wc2x-extensions -Wpre-c2x-compat -
Warray-bounds-pointer-arithmetic -Wassign-enum -Wconditional-uninitialized -
Wdocumentation -Wduplicate-enum -Wempty-translation-unit -Wflexible-array-
extensions -Wmissing-variable-declarations -Wnewline-eof -Wshift-sign-overflow
-Wstring-conversion -O2 -pipe -g -MT liblzma_la-tuklib_physmem.lo -MD -MP -MF
.deps/liblzma_la-tuklib_physmem.Tpo -c /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/../common/tuklib_physmem.c -fPIC -DPIC -o .libs/liblzma_la-
tuklib_physmem.o
…
```

Once the ports tree is done building from source, it does a “fake installation” to a temporary working directory, to archive the necessary files and gather metadata …

```
===>  Faking installation for xz-5.4.5
Making install in src
Making install in liblzma
Making install in api
 mkdir -p '/usr/ports/pobj/xz-5.4.5/fake-amd64/usr/local/include'
 mkdir -p '/usr/ports/pobj/xz-5.4.5/fake-amd64/usr/local/include/lzma'
 /usr/ports/pobj/xz-5.4.5/bin/install -c -m 644  /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/base.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/bcj.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/block.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/check.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/container.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/delta.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/filter.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/hardware.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/index.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/index_hash.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/lzma12.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/stream_flags.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/version.h /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma/vli.h '/usr/ports/pobj/xz-5.4.5/fake-
amd64/usr/local/include/lzma'
```

```
 /usr/ports/pobj/xz-5.4.5/bin/install -c -m 644  /usr/ports/pobj/xz-5.4.5/xz-
5.4.5/src/liblzma/api/lzma.h '/usr/ports/pobj/xz-5.4.5/fake-
amd64/usr/local/include/.'
 mkdir -p '/usr/ports/pobj/xz-5.4.5/fake-amd64/usr/local/lib'
 /usr/bin/libtool   --mode=install /usr/ports/pobj/xz-5.4.5/bin/install -c
liblzma.la '/usr/ports/pobj/xz-5.4.5/fake-amd64/usr/local/lib'
libtool: install: /usr/ports/pobj/xz-5.4.5/bin/install -c -m 644
./.libs/liblzma.a /usr/ports/pobj/xz-5.4.5/fake-amd64/usr/local/lib/liblzma.a
libtool: install: /usr/ports/pobj/xz-5.4.5/bin/install -c -m 644
./.libs/liblzma.so.2.2 /usr/ports/pobj/xz-5.4.5/fake-
amd64/usr/local/lib/liblzma.so.2.2
libtool: install: /usr/ports/pobj/xz-5.4.5/bin/install -c -m 644
./.libs/liblzma.lai /usr/ports/pobj/xz-5.4.5/fake-
amd64/usr/local/lib/liblzma.la
 mkdir -p '/usr/ports/pobj/xz-5.4.5/fake-amd64/usr/local/lib/pkgconfig'
 /usr/ports/pobj/xz-5.4.5/bin/install -c -m 644 liblzma.pc '/usr/ports/pobj/xz-
5.4.5/fake-amd64/usr/local/lib/pkgconfig'
…
```

… so that OpenBSD's package manager can be used to manage the installation, upgrading, or removal of the port as you please. The ports tree is also how the OpenBSD team builds binary packages for distribution.

```
===>  Building package for xz-5.4.5
Create /usr/ports/packages/amd64/all/xz-5.4.5.tgz
Creating package xz-5.4.5
reading plist|********************************************|100%
checking dependencies|********************************************|100%
checksumming|********************************************|100%
archiving|********************************************|100%
Link to /usr/ports/packages/amd64/ftp/xz-5.4.5.tgz
```

Then the software is finally installed from the resulting package.

```
===>  Installing xz-5.4.5 from /usr/ports/packages/amd64/all/
xz-5.4.5 (processing)|********************************************|100%
xz-5.4.5 (extracting)|********************************************|100%
xz-5.4.5 (installing)|********************************************|100%
xz-5.4.5: ok
```

Then it returns to the prior port or dependency, or returns you a shell if it has just completed the port you requested in the first place.

Enough reading now, go do it!

34. We can now go back to being our ordinary selves (not root) and use `sudo` to become root when we need those privileges. You might want to type `exit` to quit out of the root shell that was started by the `su` command.

    Try to run the following command …

    ```
    $ sudo grep failsafe /etc/master.passwd
    ```

    … you should see something like …

    ```
    failsafe is not in the sudoers file. This incident will be reported.
    ```

Fear not, it's your system you'll be reported on … and we haven't configured reporting yet. More importantly, you're getting this because we haven't set ourselves up with privileges to use `sudo` yet.

Use `su` to become root again, then edit the `sudo` configuration file, `sudoers` …

```
# visudo
```

… `sudo` comes with its own utility to edit its configuration file. The password database does too (`vipw`). The idea behind these utilities is that they'll lock the file to make sure concurrent edits don't happen, and do sanity checks on syntax before saving out the results, to make sure these critical system files don't get corrupted.

Find the line that allows the "wheel" group access to run all commands, but NOT without a password, and uncomment it by removing the pound sign (`#`) from the beginning of the line. Then write out the file. Then try to view your password hash again in the shadow file (the command starting with `sudo grep` above) … if you did it correctly, you will get results instead of a warning.

HISTORICAL NOTE: BSD-derived operating systems use the group wheel to contain all system administrators who are able to "take the wheel" of the system, so to speak. On a BSD-derived OS, you *must* be in the wheel group to use `su` to become root. Since your account was created during installation, it was assumed to be an administrator and naturally made a member of wheel.

You can confirm this with `id`.

```
$ id
uid=1000(failsafe) gid=1000(failsafe) groups=1000(failsafe), 0(wheel)
```

If you're ever curious what user you are, or what permissions you have, try either the `id` or `whoami` commands.

35. For those of you like me, who want a more modern version of `csh` …

    ```
    $ cd /usr/ports/shells/tcsh && sudo make install
    ```

    … the `reaper`, the grading script used later in the class, used to be written in `csh`, and your dinosaur of a professor may be using it still for some things, so we'll be doing this on all your VMs.

36. For interactive command lines, all of you should be making yourselves at home in `bash`, which is not included by default, so let's build that too.

```
$ cd /usr/ports/shells/bash && sudo make install
```

37. Now that it's installed, change your login shell to `bash` with the command `chsh`, which is of short for "change shell," of course. Before you do that, though, check out the contents of the file `/etc/shells`. This file commonly determines what are the acceptable contents of the login shell field in `/etc/passwd`.

    Please note here, if you didn't before this, that third-party software installed by the ports tree lives under `/usr/local`, though configuration files (like `/etc/sudoers`) don't.

    If you customized your shell prompt on EDORAS in lab zero, now would also be a good time to move those customizations over by making the proper edits to `~/.profile` or whichever of your shell startup scripts you used.

38. Often times in the past, people have had problems fetching distribution (source) files when building things out of the ports tree. In some cases, this is because people are (awesomely) using the virtual-ness of our VMs already, pausing and resuming them, closing their laptops … unfortunately, as said before, this lets our VM system clocks get out-of-phase with reality. In a couple cases I've witnessed, student OpenBSD VMs were not downloading software because a VM loses time and after it does, a remote web server's security certificate appears to that VM to be signed *in the future.* One student VM's clock was ten days behind. `date` will help you check and set your system clock.

    This is a common problem in VMs. VMs, since they're virtual, don't always have the CPU, and even when running miss some of the interrupts required over time to keep the system clock up to date. I will mention this at some point during the lectures, but the optimal solution is to run your hypervisor's guest tools in each VM. This typically exposes the host operating system's clock via a paravirtualized kernel device (see `man pvclock`). Those of you running OpenBSD on VMware and Intel products will be able to use dmesg, the system messages tool, to see the stub of paravirtualization …

    ```
    $ dmesg | grep pv
    pvbus0 at mainbus0: VMware
    vmt0 at pvbus0
    ```

    … but that the OpenBSD kernel apparently doesn't support VMware's `pvclock`. VMware has open sourced its guest tools (https://github.com/vmware/open-vm-tools/) to get the help of the community in maintaining them. They've been ported to FreeBSD and to NetBSD, but not to OpenBSD.

    Those of you running OpenBSD as a VM on ARM Macs will see that OpenBSD's ARM64 kernel doesn't even support the `pvbus` inside UTM/Qemu yet, which is a KVM hypervisor.

    In lieu of a better option, you should always run a network time protocol daemon (`ntpd`) in VMs, even if it's only good for minor adjustments here and there. `ntpd` is started by

default in OpenBSD nowadays …

```
$ grep ntpd /etc/rc.conf
ntpd_flags=
```

… the null flag here means `ntpd` should start with no special options. If you flip through the file `/etc/rc.conf` with `more`, you'll rapidly see that you should not change that file, because it's where the defaults are stored. Services that don't start by default have their flags set to `NO`. When we want to override these defaults, we'll use the file `/etc/rc.conf.local`, and we'll use that file soon.

In the meantime, use `ps` to list the process tables and confirm that `ntpd` is indeed running.

```
$ ps auxww | grep ntpd
_ntp      19655  0.0  0.1  1476  2792 ??  S<pc     3:42PM     0:00.86
ntpd: ntp engine (ntpd)
_ntp      15464  0.0  0.1  1348  2500 ??  Ip       3:42PM     0:00.03
ntpd: dns engine (ntpd)
root      38970  0.0  0.1  1356  1740 ??  I<pU     3:42PM     0:00.02
/usr/sbin/ntpd
failsafe 38579  0.0  0.1  2196  1356 p0  R+/1     8:27PM     0:00.00 grep
ntpd
```

You should see multiple processes, like I did in the output above. What's going on here is called privilege separation. Because of the low port number it occupies, like a lot of service processes, `ntpd` must be started as root by convention. However, it does not need to remain as root. From a security perspective, code that runs as root can be exploited to give up root access to your system. So `ntpd` splits itself up into multiple processes, and does most of its work using its own unprivileged user account.

That said, `ntpd` is best at performing minor adjustments to a system clock. If your VM gets hours off the mark, check out `man date` and figure out how to check your VM's clock and do one-time manual adjustments to your VM's clock; you'll use this tool both to check and set the clock on all your VMs, though syntax may vary slightly between operating systems. If you get certificate errors or OCSP errors while building ports, you'll need to use `date` to fix your VM's clock.

39. Let's install ISC-BIND, the Internet Software Consortium's Berkeley Internet Name Daemon, a DNS server … because that's what we're here to do, to stand up a name server.

```
$ cd /usr/ports/*/isc-bind && sudo make install
```

This is going to take a LONG time, and will probably be interrupted by the error I refer to in red, just below. You can't do part three of this lab (configuring DNS) until ISC-BIND is installed, but you can continue working on parts four, five, and six while ISC-BIND and its dependencies are built from source.

You should take note of any errors or notes; **read the entire output back to the first sign of an error, if you get an error**. Over time, you'll develop the knowledge of what to tune in to, and what doesn't matter.

If LLVM keeps running out of memory on you here while building `cmake`, go ahead and install `cmake` from the package repositories to get around it.

```
$ sudo pkg_add cmake
```

When you get notes like the following after installing ports, pay attention. This notice will come into play again.

```
The following new rcscripts were installed: /etc/rc.d/isc_named
See rcctl(8) for details.
```

40. We're going to need to install GnuPG on each system for grading later …

```
$ cd /usr/ports/*/gnupg && sudo make install
```

… I often use the wildcard this way to find the right place in the ports tree, so I don't need to know how the OpenBSD team has categorized a given port … very handy. This build takes way longer than `sudo` or `bash`, because `gnupg` has a lot more dependencies, and this is why I put it last.

This may need to run overnight. It's okay … let it run. This is why it's better to have an Intel CPU, and better to use a non-portable for your lab computer. **While your VM is building GnuPG, you can open another terminal, SSH into it again, and do other things with it … like move on to the next part of this lab.**

‼ IMPORTANT NOTE: If you ever want to do other stuff on your OpenBSD VM with your console, and you're booted into multiuser mode, the keystroke combinations `control-alt-F2` through `F4`, and `F6`, will give you alternate text-based consoles. If you're on a Mac, use the `option` key instead of `alt`, and you may additionally need to hold down the `fn` (function) key to escape any other local meanings on your Mac. `F5` doesn't work because it's reserved in OpenBSD for X11 … graphics mode. Most other Intel-based Unix and Linux OSs also support virtual consoles using these key combos.

Of course, you *could* and *should* just SSH into your VM again in another terminal window or tab too, but these hotkey combinations could come in handy in the future, if you ever find yourself unable to SSH into one of your VMs.

While either in another terminal or logged into your OpenBSD VM over SSH, run `top` to watch the compilation. `top` shows all running processes, sorted by CPU usage by default. An even better, more verbose, and thread-aware re-write of `top` is called `htop`

and can also be found in the ports tree.

41. Please also check out (with `ls`, of course) the contents of `/var/db/pkg` … this is where OpenBSD keeps information on the packages it has installed. You will note that a LOT of software got installed as default dependencies for other software we wanted during this lab. For each piece of software (each "port") the ports tree wanted to build, it went through the process of building a package, so that the package management tools could be used to manage them (query, version track, delete, etc.). Each "entry" in the package database is a directory in `/var/db/pkg`. Explore this directory tree, and check out the contents of the files in one or two packages.

    Run `pkg_info` to see a list of installed packages, essentially the same data as in `/var/db/pkg` presented from another point of view.

    Run `apropos pkg | grep ^pkg` for a list of package management utilities.

part three: setting up DNS

In the last part of this lab, we installed ISC-BIND, the Berkeley Internet Name Daemon. Now, we're going to set it up as a name server for the domain cs470.internal, and set up both forward lookups for cs470.internal and reverse DNS for the IP space we're using in the labs.

Once we're done setting it up, we'll use our ISC-BIND as the name server for the OpenBSD instance we're standing it up on. Then, we'll try out using it as the name server for our host operating system. From here on out, we're going to use it to look up internet sites by name for all our VMs moving forward. Everything builds on top of everything else, from here.

Not if, but when, you run into errors, always remember: logs are your friend. If you were paying attention earlier, you saw that virtually all the log files curated by the Unix logging daemon, `syslogd`, were all going to the directory `/var/log`.

42. Lots of ports, when they install stuff, litter the file system with things of all manner of usefulness. The `man` pages, for instance, go into `/usr/local/share/man`. Often times, sample files come with it. I, personally, ran this command to try to find them …

    ```
    $ find /usr/local -iname "*bind*"
    ```

    … this tells `find` to look in `/usr/local` for files containing the string "`bind`," not case sensitive ("`-iname`"). If we didn't put those asterisks in quotes, the shell would try to expand them to match filenames in the current directory rather than passing them through to `find`.

    There are about 40 results for me here, but this sure beats a manual directory traversal,

and the directory `/usr/local/share/examples/bind9` sure stands out. Thanks, `find`.

Compare to the output of `more /var/db/pkg/isc-bind*/+CONTENTS` … there's a lot more in this file, clearly, but as the master list of files installed with the port and package, this is another great way to find whatever you're looking for.

43. Remember the message the ports tree gave us about BIND? Installing BIND added a new "rcscript" to `/etc/rc.d/isc_named`. Look at the contents of this file with the `more` command. Looks like `rndc.key` contains a key used to manage BIND, may be found in `/etc` or `/var/named`, but everything expects to live in `/var/named`. I confirmed this by doing an `ls -l /var/named` and seeing the whole file tree there, and a sample `named.conf` in `/var/named/etc`.

    OpenBSD has completely stripped down its sample `named.conf` … a lot of configuration examples used to be in there, but no longer. You're going to need to look at `man named.conf` and/or online resources regarding its syntax to see how to implement what I ask you to do in the rest of this section of this lab. When you do, check out the still politically incorrect, fresh-from-the-dungeon syntax of the `named.conf` file. Yes, BIND still uses terminology neo-Marxists and aspiring dictators everywhere are trying to make taboo … even in software.

    `acl clients` with `localnets` in it will presumably allow the local IP subnet of our system to access the name server. There is a stanza with some general options that govern how BIND operates and how zones resolve, then we add in the "zones" themselves, the domains and IP ranges we want our name server to provide authoritative answers for. We'll be adding a couple zones, one for cs470.internal and the other for "reverse lookups" of IP addresses to names on our lab subnets.

    For me, I added the following to `/var/named/etc/named.conf`:

```
zone "cs470.internal" {
  type master;
  file "/master/cs470.internal";
};

zone "77.42.10.in-addr.arpa" {
  type master;
  file "/master/10.42.77";
};
```

    Note: filenames in `named.conf` are relative to `/var/named`. BIND uses `chroot` (remember `chroot` from after installation?) to limit its filesystem access to **only** this directory after it starts up, to (try to) prevent access to the rest of the filesystem if the name server is compromised.

    This means you'll need to make a directory at `/var/named/master` if it doesn't already exist.

Also note: we reverse the octets of our VMware NAT network in the zone name of reverse lookups because DNS goes from minor to major domains in each field (.com is greater than any of the companies below it), while the standard notation for IP addresses goes from major to minor (network-associated bits on the left, host bits on the right). This is also one place you can see the internet's ARPA/DARPA roots exposed, in the virtual namespace used for reverse lookups.

‼ YET ANOTHER IMPORTANT NOTE: You will need to change your IP addresses, zone file names, etc., to reflect your private NAT network's IP addresses for your VMs.

44. For `/var/named/master/10.42.77`, I just did a web search for a template for a DNS reverse mapping zone file, and came up with this page ...

    http://www.zytrax.com/books/dns/ch6/mydomain.html

    … and then I edited it, making sure it properly reflects our subnet address (192.168.X / 172.X.Y / 10.X.Y) as the "origin" of this zone file. I'm designating our OpenBSD VM the name server for our network, and properly publishing that here in DNS with the "NS" record below. The values in the header block define a serial number for the data (you will have to increment this value to let name servers know the zone file contains new data when you update it) and a variety of times the data from this file should live in caches (TTL/refresh/retry/expiration/minimum).

```
$ sudo vi /var/named/master/10.42.77

$ORIGIN 77.42.10.in-addr.arpa.
$TTL 6h
@       IN      SOA         localhost. root.localhost. (
                            1         ; serial
                            1h        ; refresh
                            30m       ; retry
                            7d        ; expiration
                            1h )      ; minimum

        IN      NS          openbsd.cs470.internal.
71      IN      PTR         openbsd.cs470.internal.
72      IN      PTR         freebsd.cs470.internal.
73      IN      PTR         rocky.cs470.internal.
74      IN      PTR         ubuntu.cs470.internal.
75      IN      PTR         solaris.cs470.internal.
76      IN      PTR         proxmox.cs470.internal.
78      IN      PTR         k8s.cs470.internal.
```

I wholeheartedly recommend the use of tabs to line up the text nicely.

Note how all the PTR records end with a period; DNS names decrease in significance to the left … `internal` is the top-level domain, `cs470` is ostensibly the name of our made up organization for the labs, and `openbsd` is the name of a system inside the organization at `cs470.internal`. It doesn't take a lot of imagination to see how this

would also work for www.sdsu.edu in the public DNS namespace. The period at the end denotes the root of the DNS hierarchy; very much like a leading slash (/) at the front of a Unix filename, it means that the name is an absolute, not a relative name.

45. For `/var/named/master/cs470.internal`, I started with the basic template I found for a forward lookup file at the same site above …

```
$ sudo vi /var/named/master/cs470.internal
```

… and then did the same treatment. Note that I have also added an "MX record" without a hostname. By omitting the hostname, I'm referencing the domain itself (cs470.internal), and I'm telling the network, via DNS, that if you want to send e-mail to anybody @cs470.internal, the place to go, priority 10, is freebsd.cs470.internal … our near-future lab 2 VM. This priority field allows us to stack-rank a pool of mail servers for the domain, for fault-tolerance or load-balancing purposes.

```
$ORIGIN cs470.internal.
$TTL 6h

@       IN      SOA     cs470.internal. root.cs470.internal. (
                        1       ; serial
                        1h      ; refresh
                        30m     ; retry
                        7d      ; expiration
                        1h )    ; minimum

                NS      openbsd.cs470.internal.
                MX      10      freebsd.cs470.internal.
openbsd         IN      A       10.42.77.71
freebsd         IN      A       10.42.77.72
rocky           IN      A       10.42.77.73
ubuntu          IN      A       10.42.77.74
solaris         IN      A       10.42.77.75
proxmox         IN      A       10.42.77.76
k8s             IN      A       10.42.77.78
```

46. If you'd like to test the syntax of your configuration file, I highly recommend you check out the manual page for `named-checkconf` and give it a spin.

Likewise, if you'd like to test the syntax of your zone files, I recommend you check out of the `man` page for `named-checkzone` and give it a trial run.

I test my configuration by starting up the service …

```
$ sudo /etc/rc.d/isc_named start
isc_named(ok)
```

… and I succeed the first time ... but that's me. May you too have such luck. Use the source, Luke.

These `/etc/rc.d` scripts, by convention, typically take `start`, `stop`, and `restart` as

arguments, so that you can bring down or restart a particular subsystem or service, without bringing down or restarting the whole computer … or any more than is necessary.

If you don't initially succeed, and `named` won't start, you'll be using the `start` argument again. If you don't initially succeed, but `named` started, or you make changes to your name server that requires re-reading its configuration files, you'll want to use the `restart` argument to its `rc.d` script.

You can also confirm named is running with our old friend `netstat` …

```
$ netstat -an | grep -w 53
tcp          0      0  10.42.77.71.53         *.*                    LISTEN
tcp          0      0  127.0.0.1.53           *.*                    LISTEN
udp          0      0  127.0.0.1.53           *.*
udp          0      0  10.42.77.71.53         *.*
tcp6         0      0  fe80::1%lo0.53         *.*                    LISTEN
tcp6         0      0  ::1.53                 *.*                    LISTEN
udp6         0      0  ::1.53                 *.*
udp6         0      0  fe80::1%lo0.53         *.*
```

The `-a` switch shows active connections using the system's TCP/IP stack, and port 53 is the ubiquitous port for name service. UDP is used for queries, TCP is used for synchronizing zone data in between name servers. If the service failed to start up, port 53 wouldn't be listening … but of course, this still doesn't mean that it's configured correctly.

If you were not so lucky, the reasons why will be spit out at the end of `/var/log/messages` by the system logging service, and that's where you want to start looking if you need to troubleshoot. Here's where `tail` comes in … whether or not it worked for you, remember `tail`? Use it, and see what `named` told you in that logfile.

47. Once there's a running name server, it's time to test it. Since we've set up the server, we're going to test it by aiming our OpenBSD VM's built-in DNS **client** at the server on the same system for name resolution. I changed my `/etc/resolv.conf` to look like the four lines below, and to reflect that .71 (itself) as the primary name server, and to **not** use the VMware name server (.2 or .1, depending on your setup) as a backup, and to search for "unqualified" host names under cs470.internal.

```
search cs470.internal
nameserver 10.42.77.71
#nameserver 10.42.77.2
lookup file bind
```

Then I test the name server.

```
openbsd# host openbsd
openbsd.cs470.internal has address 10.42.77.71
openbsd# host 10.42.77.71
```

```
71.77.42.10.in-addr.arpa domain name pointer openbsd.cs470.internal.
```

48. Even though we declined to configure IPv6 on our system's real interface, the IPv6 stack is still enabled (see `ifconfig lo0`), even if just for link-local addresses, and that's enough for OpenBSD to try to reach out via IPv6 and fail to look up internet hosts for your network. We need to tweak `named` to use only IPv4 for its own communications.

    !! ERRATA NOTE here in red. Look at `named` defaults in its startup script, `/etc/rc.d/isc_named`, and look at the default configuration file for all services, `/etc/rc.conf`. Figure out how to set the startup flags (arguments) for a particular service.

    Consult `man named` to find out how make `named` only use IPv4 networking, and create a service flags line for `named` in `/etc/rc.conf.local` that adds the required flag(s) to the defaults. Test that it works by restarting the service, then by doing a few lookups with `host`.

    Also, change the option `listen-on-v6` in `named.conf` appropriately to make sure it doesn't use IPv6, and restart the service to make it parse its configuration file. When you get this portion of this step right, the output of `netstat -an | grep -w 53` will not show port 53 open with TCPv6 or UDPv6 like it did above.

49. Often times, resolvers (DNS clients) ask for both DNS "A" records (IPv4 lookups, 32 bits) and "AAAA" records (IPv6 lookups, 128 bits); if you look up prominent internet sites, you should get back some IPv6 responses. This will cause occasional problems because our lab is not going to support IPv6.

    We also need to filter out these IPv6 answers (AAAA replies) from coming back to us over IPv4, because this will make a system that thinks it has IPv6 connectivity try it, and time out … or fail.

    The configuration for this has been moved out of `named.conf` as of version 9.18 and into a plug-in. Add the following into `named.conf` just *after* the options stanza.

    ```
    plugin query "filter-aaaa.so" {
            filter-aaaa-on-v4 yes;
    };
    ```

    You'll find this filter in `/usr/local/lib/bind` but because `named` runs inside a `chroot` it can't see `/usr/local/lib/bind`. We need to replicate the same directory structure under the `chroot` used by `named`, because `named` will be looking for filters in that folder…

    ```
    $ sudo mkdir -p /var/named/usr/local/lib/bind
    ```

… and link all the filters into that replica folder. Note that symbolic links would not work here because of the `chroot`; no file namespace path would lead to that folder. We have to use hard links.

```
$ sudo ln /usr/local/lib/bind/* /var/named/usr/local/lib/bind/
```

Then restart `named`.

50. Along with pushing filtering out to plug-ins in version 9.18, doing validation of DNSSEC signatures is now a default. While this is great for security, knowing that you're really getting DNS answers from authoritative sources, we're going to be validating most of the content we get back over the network. Making sure that DNSSEC doesn't get in our way is more trouble than we're ready to take on yet, so let's take a pass on validating DNSSEC signatures … you should look them up and read about them, though.

    Add the following line somewhere in the "options" stanza of `named.conf` …

    ```
    dnssec-validation no;
    ```

    … then restart `named` again.

51. We want to make sure our server starts up BIND at boot time, since it's now a name server. Remember that message from the ports tree about the `rcctl` command? `man rcctl` showed me that OpenBSD is starting to follow in the footsteps of `launchd` and `systemd`, because we apparently want to type this to make BIND start at boot:

    ```
    $ sudo rcctl enable isc_named
    ```

    I discovered that this put a new type of startup declaration (to OpenBSD) in the standard place for local changes, `/etc/rc.conf.local`:

    ```
    pkg_scripts=isc_named
    ```

    After a reboot, I saw the message during startup …

    ```
    starting package daemons: isc_named.
    ```

    … thus, the service is properly installed and configured. Note: we don't typically need or want to reboot to start up a new service, but we do often want to test that it starts up properly after a reboot.

52. Check out the contents of the file `/etc/myname` ... mine said `openbsd.localdomain` … make sure it contains only `openbsd` and nothing more.

    Like other network configuration files, this file is only parsed at startup ... but there's a

way to change the active setting, in this case what the system thinks its hostname is. If `/etc/myname` had anything besides "openbsd" in it, you only fixed the starting configuration of the system with that file. To save a reboot and apply that change to live, booted system, run `sudo hostname openbsd`.

53. Let's test your name server … make sure your host computer can use it to access the internet.

    To do this, we want to test various use cases.

    First, we want to test that your name server is able to resolve hosts on the internet. Using `host` on the command line on your OpenBSD VM, look up various internet website hostnames, and make sure you get the right answers, and that you only get IPv4 answers.

    Next, let's aim your lab's host computer at your OpenBSD VM for DNS as a client, by its IP address. Mac users will find this in System Preferences.app or System Settings.app, depending on the version of macOS you're running. Windows users will find it in network interface settings in the Settings app, and Linux users … typically `/etc/resolv.conf` but sometimes you're forced to use another UI for this. Consult your documentation. ☺

    ‼ IMPORTANT NOTE: some browsers, notably Firefox, have their own DNS settings; you may have to make sure your browser is using your system DNS settings to test web browsing using your new name server.

    ‼ ALSO NOTE: some people have failed this step until they disabled IPv6 on their host computer.

    That said, when it is on, you should now be able to ping it by name at openbsd.cs470.internal from your "host" operating system.

    ‼ ALSO ALSO NOTE: If you don't get this working, keep trying at it later as you move on through the labs, not a big deal. You will want to remove/undo this setting anyways, when you're not doing your lab, because if your VM isn't on, you won't get name service! You should be able to make this work, but we only need your other VMs to get name service from this machine in the end.

part four: configuring your host operating system's `hosts` file

Like we just said there, changing our host operating system's resolver configuration back-and-forth to use our OpenBSD VM and see the VMs on our lab network would be a pain. Let's just **not** do that moving forward, but get everything we want, shall we?

On every operating system, there's a `hosts` file that is consulted before the network-based DNS resolver, and asking name servers … so you can override the DNS address lookups for any name you like, with some limitations. Like we said in the opening exposition to this lab, prior to developing the DNS protocol, everybody on the early internet just shared this huge flat "hosts" text file, until it became untenable, and we needed a protocol like DNS, with a hierarchical namespace and distributed lookups and authority. For our labs, since we're just setting up half a dozen or so local VMs, it makes sense to configure them in the hosts file so we can look them up no matter what name server our lab host is aimed at. They're local VMs anyways.

‼ IMPORTANT NOTE: be careful here with the `hosts` file! It's a common mistake to leave stuff in here, forget about it, and wonder why your computer uniquely among all the others next to it can't get to *that* internet site in particular.

If you're using macOS or Linux as your host operating system, you want to (`sudo`) edit the file `/etc/hosts`. If you're using Windows and WSL, you want to edit the file `C:\WINDOWS\system32\drivers\etc\hosts`. I recommend you run Wordpad as administrator from the start menu. If you use another editor, it'll get the line breaks wrong. If you don't run it as administrator, you'll be denied access to the file. WSL updates its own `/etc/hosts` file from the file Windows on the Windows side of the house, so if you get the one in Windows, you get both Windows and WSL covered, and we need both. You'll have to close all your WSL shells and run `wsl --restart` in PowerShell to force a re-parse of this file. If that doesn't work, just reboot Windows … it's still the best way to fix a lot of things with Windows, sadly. ☺

Either way, you want to add the following entries to your hosts file:

```
10.42.77.71  openbsd openbsd.cs470.internal
10.42.77.72  freebsd freebsd.cs470.internal
10.42.77.73  rocky rocky.cs470.internal
10.42.77.74  ubuntu ubuntu.cs470.internal
10.42.77.75  solaris solaris.cs470.internal
10.42.77.76  proxmox proxmox.cs470.internal
10.42.77.78  k8s k8s.cs470.internal
```

Of course, change the first three octets to match **your** private IP subnet, and test these new lookups with `ping` and `ssh` on the command line on your host system, and your OpenBSD and FreeBSD VMs as targets. You should now be able to freely address these systems by name, instead of by IP, without aiming your host system at the name server on your OpenBSD VM.

part five: SSH agent forwarding (early grading configuration)

Now that we have a single properly-configured virtual machine up and running, we can pilot the grading configuration for your lab 1 VM. We'll be revisiting this setup near the end of class for the rest of your VMs, but lab 1 has a check-in date to receive full credit for non-procrastination.

54. Inside your OpenBSD VM, like we did on your host at the end of lab zero, create a ~/.ssh/config file and set it up so that your OpenBSD VM is also aware of your EDORAS account and the username you were given there.

```
Host edoras edoras.sdsu.edu
Hostname edoras.sdsu.edu
User csscXXXX
```

When we forward our SSH agent, we forward a socket that has access to our agent, and thus our key material, so you only want to forward your agent to a system you administer, and where you trust all the other administrators. We will **not** be forwarding our agent to EDORAS, just to our VMs.

On your host system, add the following lines to the end of your host's ~/.ssh/config file.

```
Host openbsd freebsd rocky ubuntu solaris proxmox k8s *.cs470.internal
        ForwardAgent yes
```

As you can see, this line calls out all our future VMs. Any time the SSH client (ssh) hits this configuration, it will parse all configuration until the next Host directive, and then stop. ssh will stop at the first configuration stanza matching a host you're connecting to, and will not evaluate any more configuration. Because of this, if you put in a separate line for your OpenBSD VM in ~/.ssh/config on your host, like the example near the end of lab 0, remove it. If you want to add any customization for your lab VMs, you'll want to do it under this configuration block … and make sure it matches all your later VMs. Since we'll use failsafe as the local admin account on all our VMs and LDAP to share regular user accounts, this shouldn't be difficult or controversial.

55. To test agent forwarding, first SSH into your OpenBSD VM from your host and run …

```
$ ssh-add -l
4096 SHA256:blahblahblahblahblahblahblahblah peter@shoggoth (RSA)
```

… that's a lowercase letter "L," for "list." This asks your agent to list the keys it has imported.

If ssh-add reports that it Could not open a connection your authentication agent, then your agent isn't running, or isn't forwarding to your OpenBSD VM.

If ssh-add reports that The agent has no identities, then it can talk to your agent and you probably have agent forwarding working but need to import keys with the ssh-add command on your host. If this is the case, check your shell scripting assignment from lab 0 out; it was supposed to automate this for you.

If you see your key, like I did above, you should be good to go. Now try to SSH into

EDORAS from your OpenBSD VM.

If you set this up correctly, you should be able to SSH into your OpenBSD VM without a password, and directly from your OpenBSD VM into EDORAS, also without a password. AGAIN, this is for grading … make sure it works!

part six: patching and cron jobs

In each of our labs moving forward, we'll close with the care-and-feeding steps that need to take place on each of your VMs. Notably, we always check for and install updates to the operating system and third-party software we've installed.

56. Run the following command:

```
$ sudo crontab -l
```

This displays root's "crontab" … the table of things to do, as root, on a schedule. See the syntax of the file. Let's add to root's crontab to run `syspatch` to check for available patches each night, update ports, and see which ones are out of date.

First, as root, `crontab -e` (`-e` is the switch to edit) and then add the following lines:

```
0 0 * * *   syspatch -c
0 1 * * *   cd /usr/ports && cvs -q up -Pd -rOPENBSD_7_6
0 2 * * *   /usr/ports/infrastructure/bin/pkg_outdated
```

This literally means, at 00:00 (midnight) every day of every month, regardless of which day of the week, run `syspatch -c` to check for any available updates to OpenBSD. The second line runs an hour later at 1:00, and its associated command keeps our ports tree up to date. The third command, another hour later, checks which installed ports are outdated from the versions provided by updates to the ports tree. When that happens, you will want to install updated versions from the ports tree or the OpenBSD package repositories.

When updating ports, these links from the OpenBSD project may be helpful:

- porter's handbook: https://www.openbsd.org/faq/ports/
- working with ports: https://www.openbsd.org/faq/ports/ports.html
- package management: https://www.openbsd.org/faq/faq15.html

If you ever run low on space on your OpenBSD VM, `make clean` can be used to remove temporary build artifacts below any point in your ports tree hierarchy. The ports tree will rapidly become the biggest consumer of storage on your OpenBSD VM's filesystem. If you issue `make clean` at the top of the ports tree (`/usr/ports`), you can often times save a lot of disk space on an established OpenBSD instance. This will also hold true for

your FreeBSD VM, which also uses a ports tree.

As you'll start to see after we set up a mail server in lab 2, `crond` e-mails the output of these "cron jobs" to the users who own them. Capturing these e-mails is a primary goal behind setting up that mail server, and the primacy of email for communications between people and systems, the reason why we get it running right away in lab 2.

57. We'll dip our toes into source patching in a bit here, and rebuild a customized OpenBSD kernel from scratch, because this is still how most long-time BSD users do it.

    OpenBSD recently became the last of the operating systems we'll use in this class to implement a binary patching system, `syspatch`. I read `man syspatch` (as should you) and it told me we need to put a URL for an OpenBSD mirror into `/etc/installurl` … check that this file exists, and ends with the OpenBSD folder, as in the URL below …

    https://cdn.openbsd.org/pub/OpenBSD

    … though maybe using the address of another OpenBSD mirror.

    Let's not wait to patch this system immediately … run `syspatch` as root or `sudo syspatch` with no options, then `reboot` to lock in the security fixes immediately. You should see output like this …

```
Get/Verify syspatch76-001_unbound... 100% |*****************|  2836 KB    00:00
Installing patch 001_unbound
Get/Verify syspatch76-002_xserver... 100% |*****************|  4308 KB    00:00
Installing patch 002_xserver
Get/Verify syspatch76-005_expat.tgz 100% |*****************|   656 KB    00:00
Installing patch 005_expat
Get/Verify syspatch76-006_wg.tgz 100% |*****************|   145 KB    00:00
Installing patch 006_wg
Relinking to create unique kernel... done; reboot to load the new kernel
Errata can be reviewed under /var/syspatch
```

    As it says, errata can be found under `/var/syspatch`. As I said before, it can also be viewed at https://www.openbsd.org/errata76.html ... there you will see why we didn't get a patch 002; it's only for the `alpha` CPU architecture.

    Unfortunately, `syspatch` doesn't patch the OpenBSD operating system source tree. It just patches the binary objects that would be built from a patched source tree … and we're about to build a custom kernel, so we need to patch our local source tree. Use the following command to look over one of the source patches from OpenBSD under `/var/syspatch` …

```
$ more /var/syspatch/76-002*/*patch.sig

untrusted comment: verify with openbsd-76-base.pub
RWTkuwn4mbq8ooVU88a3J6qJ9Yi87hwgaLoqT+acUCFBrdxzqSQGKyyGm0cL+Ti9qmoU/3OOSRaLEif
0wKXaSV0Ego9JCdHhEwQ=
```

```
OpenBSD 7.6 errata 002, October 29, 2024:

Fix memory allocation error in the Xkb X11 server extension. CVE-2024-9632

Apply by doing:
    signify -Vep /etc/signify/openbsd-76-base.pub -x 002_xserver.patch.sig \
        -m - | (cd /usr/xenocara && patch -p0)

And then rebuild and install the X server:
    cd /usr/xenocara/xserver
    make -f Makefile.bsd-wrapper obj
    make -f Makefile.bsd-wrapper build

Index: xserver/xkb/xkb.c
===================================================================
RCS file: /cvs/xenocara/xserver/xkb/xkb.c,v
diff -u -p -u -r1.24 xkb.c
--- xserver/xkb/xkb.c   22 Jan 2023 09:44:42 -0000      1.24
+++ xserver/xkb/xkb.c   12 Oct 2024 07:52:49 -0000
@@ -2992,13 +2992,13 @@ _XkbSetCompatMap(ClientPtr client, Devic
         XkbSymInterpretPtr sym;
         unsigned int skipped = 0;

-        if ((unsigned) (req->firstSI + req->nSI) > compat->num_si) {
-            compat->num_si = req->firstSI + req->nSI;
```

… I omit the rest of the patch here, for brevity's sake. You can see an "untrusted comment" at the top, followed by a cryptographic signature. Underneath `Index`, you can see the meat of the source patch, created by the utility `diff` (including the command used to create the patch file), which goes hand-in-hand with the utility `patch` to share changes to source code.

Also note, just under "apply by doing," directions for how to apply the patch. The tool `signify` is used to validate the cryptographic signature at the top of the patch file, and the output of that validation is piped into `patch` after changing to the directory of the source tree to fix. As I wrote this lab, there were obviously a few patches available for OpenBSD 7.6, that step would need to be done once for each of the patches, and not necessarily always in the same directory. More so, it'll need to be run each time new patches are installed by `syspatch`.

To make this easier, I wrote a quick shell script I called `srcpatch`. Download it to your OpenBSD VM, install it in `/usr/local/bin` …

```
$ cd /tmp && ftp https://slagheap.net/media/cs470/srcpatch
$ chmod 755 srcpatch
$ sudo mv srcpatch /usr/local/bin
$ sudo chown root:wheel /usr/local/bin/srcpatch
```

… and then run it with `sudo`.

```
$ sudo srcpatch
Password:
76-001_unbound ... patching in /usr/src
76-002_xserver ... skipping, no /usr/xenocara
76-005_expat ... patching in /usr/src
76-006_wg ... patching in /usr/src
```

Run `srcpatch` again, every time you run `syspatch` to install new patches.

Also, make sure to take a look at the shell script with `more` and see what I've done in there.

```
$ more /usr/local/bin/srcpatch
```

Automating things like this is the focus of the SAMS and Kochan texts, the spice of productivity, and the essence of jokes like the one on this t-shirt …

… please note the obviously monospaced terminal font. ☺

58. When doing this lab with ARM Macs for the first time, I had this annoying problem with the software RAID driver (`softraid`) causing every first boot of the OpenBSD VM to panic before making it to a boot prompt. This only ever affected me … things uniquely hitting me is something that happens a lot as the designer of these labs.

    We need something to do, however, to show how to customize a kernel, so we're going to continue removing that driver from the kernel. Software RAID is something we'll never use in a VM. **Even if you're on Intel, you're going to fix this too, just to join in on the fun of customizing the kernel.**

    This is where `uname` comes in. We'll dig into this tool in some depth later when we have more systems to poll, but it came about as Unix began to proliferate across different companies and hardware platforms. As software began to be ported to all these systems, we needed a tool to provide details of the particular operating system and instance we're logged into. For this reason, `uname` reports a variety of information. See `man uname` for a really quick dump on all the things it can do here on OpenBSD, and I do mean quick … OpenBSD's `man` page for `uname` is quite short. Run `uname` with `-a` for "all."

```
$ uname -a
OpenBSD openbsd 7.6 GENERIC.MP#338 amd64
```

The data it's presenting here are, in order, the name of the kernel (`OpenBSD`), the local hostname (`openbsd`), the version of the kernel (`7.6`), the name of the kernel configuration (`GENERIC.MP`) with a build number, and the architecture (`amd64`). Those of you on ARM64 will see slightly different output, of course.

Regardless of which architecture you're on, note how some of this information is historically propagated to `/etc/motd`.

Some of you will be using the `GENERIC` kernel, instead of `GENERIC.MP`. "MP" is short for "multi-processor," and this appears to be a function of how many sockets vs. cores your VM's hypervisor configuration shows to the VM. No matter which you're using, that kernel configuration can be found if you …

```
$ cd /sys/arch/`uname -m`/conf
```

With `pwd -P` you will discover you have been sent to `/usr/src/sys` because `/sys` is a symlink to `usr/src/sys`, and then rest appends either `amd64/conf` or `arm64/conf`, depending on which architecture you're on. I typically use the tool `arch` here (please check out its `man` page), but it's not yet behaving in line with its documentation on ARM64.

Please take a look at the file `GENERIC`, now. The kernel config file contains a lot of build macros, prefixed with `OPTION` …

```
option          USER_PCICONF    # user-space PCI configuration

option          APERTURE        # in-kernel aperture driver for XFree86
option          MTRR            # CPU memory range attributes control

option          NTFS            # NTFS support
option          SUSPEND
option          HIBERNATE
```

… and a litany of supported device drivers in an obvious taxonomy for detecting them on a variety of system buses, in order to properly detect a multitude of possible physical system configurations. Consider, for instance, the following set of lines, which declare host USB controllers will be probed.

```
# USB Controllers
xhci*   at pci?                         # eXtensible Host Controller
ehci*   at pci?                         # Enhanced Host Controller
ehci*   at cardbus?                     # Enhanced Host Controller
uhci*   at pci?                         # Universal Host Controller
(Intel)
uhci*   at cardbus?                     # Universal Host Controller
(Intel)
ohci*   at pci?                         # Open Host Controller
ohci*   at cardbus?                     # Open Host Controller

# USB bus support
usb*    at xhci?
```

```
usb*    at ehci?
usb*    at uhci?
usb*    at ohci?
```

OHCI and UHCI (together, USB 1.0), EHCI (USB 2.0), and XHCI (USB 3.0) can be detected on a PCI bus or expansion card, on a "cardbus" adapter (PCMCIA, or PC cards, a much older standard), or on another USB bus. You can probably see how this mirrors your use of USB devices, adapters, and hub … and why it's necessary.

To look at this file is to look at a compendium of devices supported by the OpenBSD kernel on this particular architecture, which IRQ (CPU interrupt) each uses, on which bus, at which address, and so on and so forth. If you were ever to have a configuration mismatch with a piece of hardware, showing up at the wrong bus address for instance, this is where you could fix that, and rebuild a kernel that hopefully would work.

Also, whether you got a `GENERIC` or a `GENERIC.MP` kernel, look at the file `GENERIC.MP` now as well. Note how it pretty much includes the `GENERIC` configuration by including that file, adds the option "MULTIPROCESSOR" and allows for detecting multiple CPUs on the main system bus.

When I took this class in 1996, one of the exercises was to take the output of `dmesg` and make a kernel that only had support for all the devices on the piece of hardware I was using. By means of commenting out all the devices in configurations that weren't required, one could build a much smaller kernel, minus all the device drivers that weren't required by this instance of hardware. When you keep in mind that the kernel is always in RAM, and can't swap out … this can save a lot of memory for processes. The same thing could be done with this virtual machine here, in order to save a lot of RAM … but RAM isn't so expensive or scarce anymore. You might want to do this if you were using OpenBSD inside lean, embedded hardware … and this is not at all uncommon out there. Remember, OpenBSD is used in a lot of small firewalls and network devices.

It's for this reason that we'll now be doing this exercise in lab 1b.

59. Note, however, that this `GENERIC` file is probably only devices, or device configurations, specific to this particular architecture. If you search for the `softraid` driver in the file `GENERIC` in this directory, you will come up empty.

```
$ grep softraid GENERIC
$
```

This is because kernel config files can include other files, and **do**, so that OpenBSD can be flexible in general, of course. OpenBSD is using this here to make the configuration modular. The files here in `/sys/arch/*/conf` contains device configurations specific to each architecture. Looking at the output of …

```
$ ls -l /sys/arch
```

… as you can see, if you didn't already while on OpenBSD's website, OpenBSD supports quite a few machine architectures, and we have the source for all of them. Let's look for other files included by our CPU architecture's generic configuration file.

```
$ grep include GENERIC
include             "../../../conf/GENERIC"
```

This incorporates the kernel configuration file from `/sys/conf` where there's a generic configuration file for OpenBSD on **all** architectures. If you `grep` for `softraid`, you will indeed find it here. So let's back up this configuration file, always a good idea before we modify files provided by the operating system.

```
$ sudo cp -p GENERIC CS470
```

Now, edit `CS470` and comment out (with a `#` at the beginning of each line) the **two lines** – there are only two and they're impossible to miss – right next to each other that clearly have something to do with the `softraid` driver. **Please also comment out the two lines just below them that mention the `vscsi` driver; I've seen that appear to cause panics too. If you have already compiled a fixed kernel, repeat the steps below and compile it again.**

Now, go back to the correct `/sys/arch/*/conf` directory for your CPU architecture, and copy the generic configuration file to one branded for this class, just like we just did in the architecture-independent kernel configuration directory.

```
$ sudo cp -p GENERIC CS470
```

Those of you who got a kernel called `GENERIC.MP`, also copy `GENERIC.MP` to `CS470.MP`.

Now, edit `CS470` (**and** `CS470.MP` if it applies to you) and change the include statement(s) near the top to reference the files `CS470` from `/sys/conf` instead of `GENERIC`.

60. Time to build a custom kernel. Tell the OpenBSD kernel source tree to spin up a folder for compiling the OpenBSD kernel. If you had the `MP` kernel, configure `CS470.MP` instead.

```
$ sudo config CS470
/usr/src/sys/arch/amd64/compile/CS470/obj ->
/usr/obj/sys/arch/amd64/compile/CS470
config  -b /usr/src/sys/arch/amd64/compile/CS470/obj  -s /usr/src/sys
/usr/src/sys/arch/amd64/conf/CS470
```

Under your architecture's section of the source tree, a directory `compile/CS470` (or `CS470.MP`) has been configured to build a kernel in, so `cd` to the appropriate directory and `make` a kernel.

```
$ cd /sys/arch/amd64/compile/CS470 && sudo make
```

This will take a little while. After it's done building you should be able to `sudo make install` to install it at the root of the filesystem as `/bsd`, the kernel to be loaded at the following boot.

```
$ sudo make install
mkdir -p -m 700 /usr/share/relink/kernel
rm -rf /usr/share/relink/kernel/CS470 /usr/share/relink/kernel.tgz
mkdir /usr/share/relink/kernel/CS470
tar -chf - Makefile makegap.sh ld.script *.o |  tar -C
/usr/share/relink/kernel/CS470 -xf -
[[ ! -f /bsd ]] || cmp -s bsd /bsd || ln -f /bsd /obsd
install -F -m 700 bsd /bsd && sha256 -h /var/db/kernel.SHA256 /bsd
```

We can see it mucking with `/usr/share/relink` – this should look familiar to something I asked you to research earlier … what is it doing? – and moving around kernels in the root directory of the filesystem.

```
total 115
drwxr-xr-x   2 root  wheel       512 Sep 30 07:33 altroot
drwxr-xr-x   2 root  wheel      1024 Sep 30 07:33 bin
-rwx------   1 root  wheel  28711831 Feb  2 21:34 bsd
-rwx------   1 root  wheel  28795222 Feb  2 15:35 bsd.booted
-rw-------   1 root  wheel   4710411 Feb  2 05:30 bsd.rd
-rw-------   1 root  wheel  28679978 Feb  2 05:30 bsd.sp
drwxr-xr-x   6 root  wheel     19456 Feb  2 15:42 dev
drwxr-xr-x  24 root  wheel      1536 Feb  2 23:09 etc
drwxr-xr-x   3 root  wheel       512 Feb  2 13:32 home
drwxr-xr-x   2 root  wheel       512 Sep 30 07:33 mnt
-rwx------   1 root  wheel  28807662 Feb  2 18:57 obsd
drwx------   3 root  wheel       512 Feb  2 13:32 root
drwxr-xr-x   2 root  wheel      1536 Feb  2 18:56 sbin
lrwxrwx---   1 root  wheel        11 Sep 30 07:33 sys -> usr/src/sys
drwxrwxrwt   8 root  wheel       512 Feb  2 21:33 tmp
drwxr-xr-x  17 root  wheel       512 Feb  2 16:10 usr
drwxr-xr-x  25 root  wheel       512 Feb  2 20:24 var
```

`bsd` is the kernel that will be used at the next boot. `bsd.booted` is the kernel being used, that is being used by the current boot cycle. `bsd.rd`, again, is a "RAM disk" kernel that can be used to perform maintenance, repair, or recovery if the rest of the filesystem is in a strange state. Some might have a `bsd.sp` too … this is a `GENERIC` uniprocessor kernel, not `GENERIC.MP`, as we saw earlier. `obsd` is the prior kernel, as you can see by the date stamp, just in case you need a known-good kernel to boot into, should you compile a bad one.

Note that it's possible to enter the name of a kernel at boot loader prompt – the same one we use to choose single-user mode – if your new kernel is bad.

If your new kernel is good, though … if your VM reboots cleanly – go ahead and `reboot`, please – the `softraid` driver shall bother us no longer. You should be able to verify this with `dmesg` and `grep`, and `uname` should report that you're using the first build (`#0`) of

your new, custom CS470 kernel.

61. A little bit of review here.

    Never, ever simply power off a Unix system without telling it you want to do so, first. Unix systems make extensive use of caches for performance reasons, especially for the filesystem. The live state and integrity of filesystems may be dependent on data in kernel memory (where filesystem code lives), not to mention the data of whatever processes are active on your system instance. To avoid data loss, these kernel data structures must be saved back to disk any time a storage device is disconnected or powered down.

    IN CASE YOU MISSED IT, LET ME PUT IT HERE IN RED AND ALL-CAPS ONE MORE TIME: UNLESS YOU HAVE NO OTHER CHOICE, **NEVER EVER** POWER OFF A UNIX SYSTEM WITHOUT TELLING IT YOU WANT IT TO DO SO FIRST.

    If you ever shut down or power off a VM by means of the UI widgets in VMware, you may well be doing the virtual equivalent of pulling the power cable on your server. DON'T DO IT.

    In order to shut down our VM, again, we want to tell it to do so first, and most modern operating systems will physically power down the hardware they're running on. In the case of VMware, this means that VMware will be able to tell the VM is off, and will release its RAM and resources for use by other processes on your computer.

    On BSD-based operating systems, we typically use the command `shutdown` …

```
$ shutdown -p now
/sbin/shutdown: Permission denied.
```

    … `shutdown` typically sets a flag (in `/etc/nologin` or a similar file) so that additional users are unable to log in, then tells the system to go through all of its usual processes for shutting down active services and software, before powering down the system.

    The `-p` switch tells `shutdown` to try to physically power down the hardware once completed.

    What about that "permission denied" error? Again, we use `ls -l` to see the permissions associated with a file, regardless of file type.

```
$ ls -l /sbin/shutdown
-r-sr-x---  1 root  _shutdown  268856 Sep 30 07:33 /sbin/shutdown
```

    This set of permissions means that root, the owner of the file, can run it, and grants access to the setuid API to *become* root while running the program. The group

associated with the file, _shutdown, is able to read and execute the file, and nobody else is allowed to do anything with it … hence the error we're running into.

OpenBSD is actually diverging from convention here by creating a separate group for those who can shut down the system. Whereas the wheel group is reserved for users who are able to become root and exert full control over the system, the operator group is the group historically reserved for people who have physical access to the system and are able to perform a limited subset of privileged operations. If they were to have all access, we'd add them to the wheel group, and/or give them the root password … in theory, at least.

What else could operators do on a BSD, you might ask? Well, let's take a look …

```
$ ls -l /dev | grep operator
brw-r-----  1 root      operator    6,   0 Feb  2 13:32 cd0a
brw-r-----  1 root      operator    6,   2 Feb  2 13:32 cd0c
brw-r-----  1 root      operator    6,  16 Feb  2 13:32 cd1a
brw-r-----  1 root      operator    6,  18 Feb  2 13:32 cd1c
crw-rw----  1 root      operator   17,   0 Feb  2 13:32 ch0
crw-r-----  1 root      operator   90,   0 Feb  2 13:32 diskmap
crw-rw----  1 root      operator   14,   3 Feb  2 13:32 enrst0
crw-rw----  1 root      operator   14,  19 Feb  2 13:32 enrst1
crw-rw----  1 root      operator   14,   2 Feb  2 13:32 erst0
crw-rw----  1 root      operator   14,  18 Feb  2 13:32 erst1
brw-r-----  1 root      operator    2,  16 Feb  2 13:32 fd0Ba
brw-r-----  1 root      operator    2,  17 Feb  2 13:32 fd0Bb
brw-r-----  1 root      operator    2,  18 Feb  2 13:32 fd0Bc
brw-r-----  1 root      operator    2,  24 Feb  2 13:32 fd0Bi
…
[output shortened here]
…
brw-r-----  1 root      operator    0,  55 Feb  2 13:32 wd3h
brw-r-----  1 root      operator    0,  56 Feb  2 13:32 wd3i
brw-r-----  1 root      operator    0,  57 Feb  2 13:32 wd3j
brw-r-----  1 root      operator    0,  58 Feb  2 13:32 wd3k
brw-r-----  1 root      operator    0,  59 Feb  2 13:32 wd3l
brw-r-----  1 root      operator    0,  60 Feb  2 13:32 wd3m
brw-r-----  1 root      operator    0,  61 Feb  2 13:32 wd3n
brw-r-----  1 root      operator    0,  62 Feb  2 13:32 wd3o
brw-r-----  1 root      operator    0,  63 Feb  2 13:32 wd3p
```

What you're seeing here is the subset of device files (in `/dev`) that refer to the group "operator." An operator can read CD-ROMs and optical media (`/dev/cd*`), read and write tape devices (`/dev/*rst*`), and has read access to all disk devices. This is because an important function of non-root operators is to perform backups, in the middle of the night, on production systems while the wheel group rests.

We want our failsafe user in both of these groups, operator and "_shutdown." Note: OpenBSD, like a few other OSs, uses an underscore in the front of user and group named to denote a service user or group local to this computer, so that a networked system could subscribe to a directory of users and groups, and easily distinguish between local service accounts and groups.

Become root or use `sudo` to `vi /etc/group` and add a comma and `failsafe` to the end of this line ….

```
operator:*:5:root
```

… and this line, where you won't need a comma …

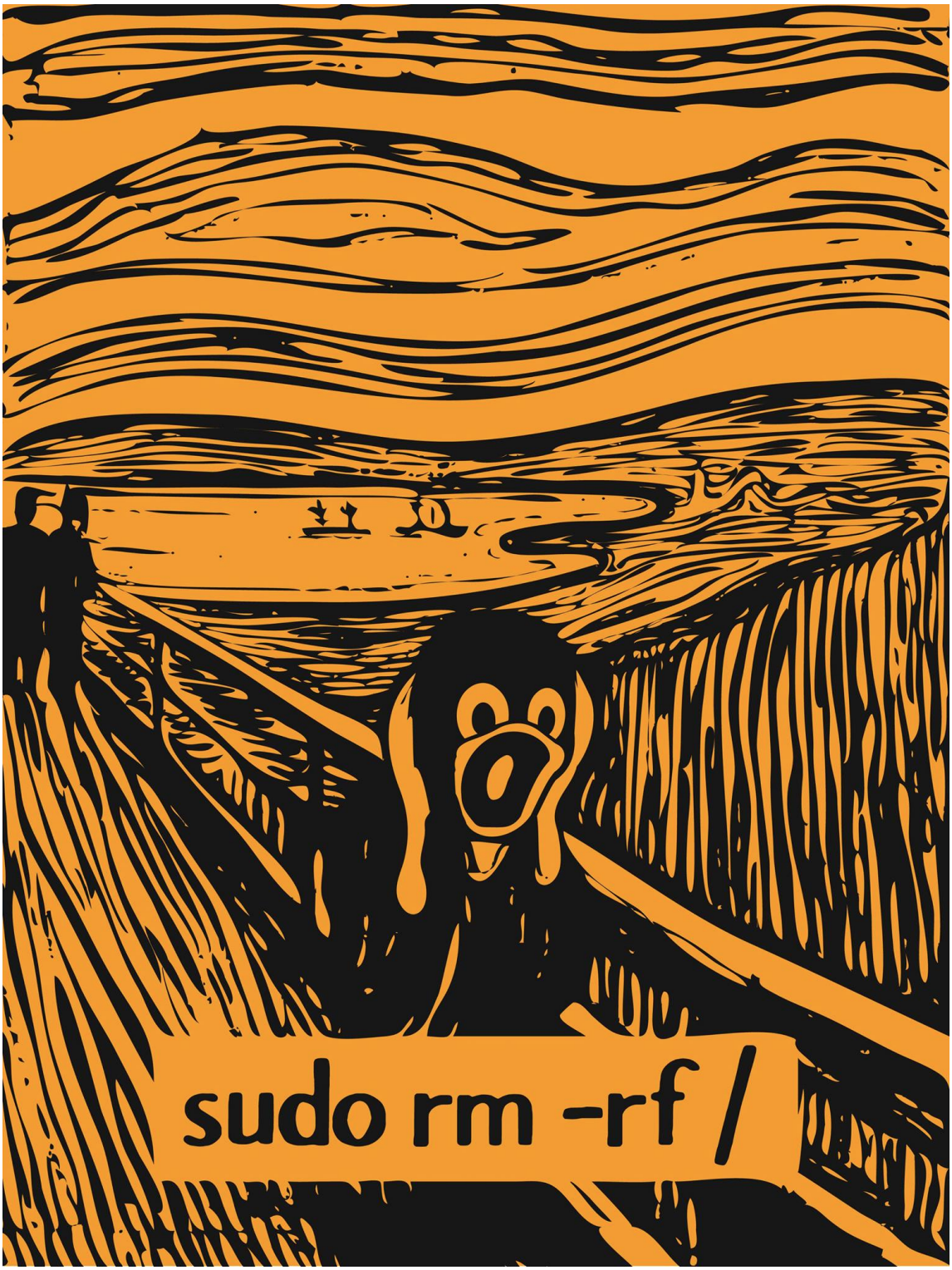
```
_shutdown:*:118:
```

… and note that editing this file does not, in and of itself, make you a member of the group. You have to log out and log back in to complete the process … these group IDs are associated with the processes and environment that belong to you, when you log in.

Log out, log back in, and shut down your OpenBSD VM without having to become root.

Well done … you've finished the first of the two hardest labs. Don't get too comfortable yet, though … lab 1b isn't a cake-walk, and it's also on OpenBSD. Before we stand up another operating system, we're going to set up two more services on this here VM, just over the horizon.

```
</lab1>
```



A reminder to be careful with root command lines and `sudo` … respect your new power. Please let this also serve as a reminder of both the calmative and healing properties of backups. If you don't have a backup configured yet, this really begs the question: what's wrong with you?