CS 470: Unix/Linux Sysadmin
Spring 2025 Lab 5
the ghost of commercial Unix past: Solaris, as told by OpenIndiana
centralized logging and configuration automation with Ansible

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

- labs 1-4: all prior VMs are required for the Ansible portion of this lab
- lab 3: Rocky Linux up, online, and sharing files via NFS

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

- having this lab's VM up and running is a requirement for grading lab 5b
- aside from the grading dependency, this lab can be worked out of order

co-authoring and editing credits:

- 2020 TA Obada Alagha, who came back in 2021 to write the first version of the Ansible lab
- 2023 TA Enrrique Torres, for updates to Ansible in 2023 and adding `expect`
- 2025 TA Max Carrillo, for adding in `pipx` and Python virtual environments

Labs 5 and 5b, collectively, were originally for completeness and for historical purposes. We've now built labs around two descendants of Berkeley Unix and the two most popular families of Linux distributions. Now we're going to cover two of the few remaining commercial operating systems that come from true, genetic AT&T Unix: Oracle's Solaris, and IBM's AIX in lab 5b, because they're still out there. Solaris is a dying operating system, but there is still a large installed base out there, and you are still likely to run into it in the field. AIX is a boutique operating system, with strong installed bases in health care, finance, and it's still quite alive and well.

There are a few more operating systems with genetic AT&T Unix that still survive, but only three are still actively developed:

- **Hewlett Packard's HP-UX**:

  Hewlett Packard Enterprise appears to be getting bored here. Very much like we saw with a dying Silicon Graphics and its formerly ground-breaking Unix, IRIX, HP-UX has been stuck on version 11i v3 since 2007, and updates are now released in bundles on an annual basis. As of the 2021 update to HP-UX, the operating system is now "intended for use on HPE Integrity servers, not on HPE 9000 PA-RISC systems," HP's now-terminated line of home-grown workstations with HP's own RISC CPU architecture.

  The Integrity line of servers has included both those HP-grown PA-RISC CPUs, which have not been sold since 2008 or supported since 2013, and the Itanium processor, jointly developed with and manufactured by Intel. However, Intel doesn't make or sell them anymore, as of the summer of 2021. Another year, another CPU instruction set bites the dust!

  Accordingly, HPE does not have a roadmap for HP-UX development past 2025, and seems to plan to end support for the operating system in 2028.

- **formerly Sun, now Oracle, Solaris:**

This is the only one of these three genetic AT&T OSs that ever had a significant port to commodity PC hardware. It still supports both PC hardware and some older SPARC-based workstations, but Solaris was clearly developed with different hardware in mind, especially Sun's – now Oracle's – line of SPARC-based data center big hardware solutions for data mega-warehouses, not really for workstations or for PC Unix. This is what remains of the Unix shop that built Java, RPC, NFS, NIS, and co-authored System V Release 4 with AT&T.

Solaris' development team at Oracle has been gutted by layoffs, and it laid off virtually all of its SPARC CPU and hardware platform design team too, in the fall of 2017. The SPARC CPU intellectual property has been placed in the public domain. Oracle still has paths defined for Solaris and SPARC hardware, but this product is clearly winding down. Its death hasn't been officially announced, but the writing is on the wall. Oracle has stated they'll keep supporting Solaris 11, the current version, through 2034.

Sun and Solaris were once so popular that there was a small but vibrant market of third-party vendors making SPARC-based hardware, even a line of SPARC-based laptops made by a small third-party hardware partner of Sun (see http://www.computinghistory.org.uk/det/32324/Tadpole-SPARCbook-3/). Only Fujitsu remains, but they have been making SPARC-based systems running Solaris for over 20 years, and they too have now signaled that they intend to stop building on the platform, with support for SPARC (not necessarily Solaris) ending in 2034. However, Solaris is all but dead; we've been stuck on Solaris version 11.4 since 2018. Again, the proverbial writing is on the wall.

https://www.fujitsu.com/global/products/computing/servers/unix/sparc/key-reports/roadmap/index.html

Silicon Valley is ever-innovative. "Sparc" already means something else to Silicon Valley, and you can't write life any funnier than it just happens sometimes. See https://sparc.co/ …

- **IBM's AIX**:

  IBM released AIX 7.3 in December 2021, and AIX is still actively developed and updated.

  Though it once had stints running on IBM's mainframes, minicomputers, workstations, and very briefly ran on early IBM PCs, it now primarily runs on IBM's POWER architecture. POWER descends from the PowerPC line of CPUs, long ago jointly developed with Apple and Motorola. IBM hasn't sold any PowerPC or POWER-based workstations since 2009, but its server hardware line still has a strong following in a few market segments (https://www.ibm.com/it-infrastructure/power), and I'm sure y'all have heard of the Jeopardy-playing Watson software.

  Of these three operating systems, AIX is the only one with a real roadmap; there are plans for AIX through 2035.

These are the last remnants of the workstation era. Each of these companies once had both a vibrant workstation and server business, each on their own home-grown CPU architecture. HP has long since completely gotten out of the CPU business and has now had the carpet pulled out from underneath them by Intel. Sun appears to be on the way out of the CPU business too. All three have gotten out of the Unix workstation business, arguably except for when HP sells consumer or business PCs with Linux installed.

As discussed in the history section of the first lecture, Bell Labs spun out Unix System Labs, then sold it to Novell. Novell kept ownership of a lot of the intellectual property to the original AT&T Unix source tree, but sold an exclusive license to sell it and continue its development to the Santa Cruz Operation, also known as SCO. SCO used to have a great Unix-on-PC business with a strong presence in retail, built on top of what was formerly Microsoft's Unix, Xenix. As Linux began to eat away at its market, SCO was sold to Caldera, which later renamed itself "The SCO Group" before suing IBM, still a licensee of AT&T Unix, over Linux, alleging that IBM violated the Unix copyrights by putting Unix code into the Linux kernel.

It was a long road, with the oddest twist of an ending. The court and the rest of us didn't discover until several painful years of the suit had passed, that Novell had never given up the intellectual property rights. Once the court was presented with the contract between Novell and SCO, SCO's case for copyright infringement rapidly disintegrated from a lack of standing to file the suit in the first place. Once it became clear SCO was going to lose, it sold its Unix business to UnXis, which became Xinuos. SCO remained alive to continue the suit against IBM, for loss of business.

Through all this, their OpenServer and UnixWare operating systems have only seen minor updates with no version number increases in almost 20 years. The current versions of each, UnixWare was released in 1998, and OpenServer 6 was released in 2005. A newer major version release of OpenServer, version 10, is really an unrelated product based on FreeBSD ... a candid admission that development wasn't happening on the old Unix codebase ... and now that this product has failed to find market share, Xinuos has also recently decided to sue IBM over Linux.

https://arstechnica.com/gadgets/2021/04/xinuos-finishes-picking-up-scos-mantle-by-suing-red-hat-and-ibm/

As often in life, you can't write it any funnier than it just happens sometimes. Many suspected that Microsoft was pulling the strings and providing the funding behind SCO's lawsuit, as the first entity called SCO was once a Microsoft subsidiary, there were lots of ties there, and Microsoft stood to gain a lot if Linux was found to be in violation of copyrights and lost some footing. Microsoft was also vehemently and vocally anti-Linux at the time. Who stands to gain the most in this suit is unclear. FreeBSD was named as the victim of IBM's alleged misdeeds this time, as it provides the underpinnings of both the newest version of Xinuos' OpenServer ... and also the underpinnings of the last and biggest commercial Unix, macOS.

Apple's macOS and its derivatives that run on familiar watches, phones, tablets, and set-top streaming devices are certified as Unix™ but are actually based on Carnegie-Mellon's "Mach" kernel and Berkeley Unix, not AT&T Unix. According to the BSD/AT&T settlement, modern BSDs definitely contain no actual AT&T code, so macOS doesn't contain any unless it came from open-sourced versions of … Solaris. Yes, the use case exists! I was validating my thinking as I wrote this, and funny how one web search and a link for macOS Catalina's source tree was able to validate my thinking …

https://opensource.apple.com/source/dtrace/dtrace-338.40.5/FileOrigins.auto.html

… Apple once flirted with making ZFS (from Solaris) its new filesystem in the early days of Mac OS X. Instead, it ended up making its own journaling and snapshotting filesystem with plenty of design cues from ZFS, and also put some of Solaris' kernel tracing tools into macOS.

Running macOS, however, requires either a Mac or a violation of Apple's license agreement, so covering the Mac in this class was never a feasible option. There are, however, numerous write-ups online to tell you how

to install macOS inside VMware. I'm not suggesting you do – just pointing out that they exist – if you were to go find them and felt like experimenting with macOS before it becomes an ARM-only operating system.

As of 2020, 32-bit HP-UX versions from HP's former line of PA-RISC CPUs can be run inside QEMU (www.qemu.org) with full graphics and network support. Within the last couple years, AIX (from IBM's PowerPC CPUs) has also been able to run on QEMU. Solaris, both the SPARC and Intel CPU variants, have been able to run inside QEMU for over a decade … but the Intel variant runs well inside VMware, so we'll put it there. I only offer up QEMU and these other OSs here for history's sake, and in case you'd like to discover more on your own later. You can find loads of abandonware operating systems (including older versions of macOS you can try if you want) at the following URL:

https://winworldpc.com/library/operating-systems

Back to Solaris. Again, Solaris is the product of the former Sun Microsystems, once the most powerful and seemingly invincible of Unix workstation companies. For many years, Solaris only ran acceptably on Sun's own hardware, and like many data center operating systems, its market was continually eroded by the increasing power and far lower price point of Linux running on commodity PC-based hardware. This trend forced Sun to finally get serious about making Solaris run well on PC hardware. Sun even began selling PC-based servers and workstations, and at one point, re-released Solaris under an open-source license as OpenSolaris, trying to pick up some of the open source community's steam. Unfortunately, neither of these initiatives held fate at bay. Sun's carcass was purchased by Oracle, largely for the rights to Java, and the desire to sue Google over not paying anything for using Java in Android. Oracle promptly made Solaris closed-source again, and even worse: it started charging for patches, essentially forcing you to pay to keep Solaris running, or to keep it safe.

However, the proverbial genie was out of the bottle. The community forked the last available OpenSolaris kernel as illumos, and today, illumos is to the OpenSolaris-based operating systems as Linux is to all the Linux-based operating systems … just a kernel and the toolchain around it. OpenSolaris was referred to as "Project Indiana" inside Sun, so when they forked the last available version of it to keep a public descendant alive, it became OpenIndiana. OpenIndiana is now the viewed as the reference distribution for the illumos kernel.

When compared with the Linux and BSD communities, there isn't a big open-source Solaris derivative community, but there are a couple companies that have made some cool and viable products with it (notably Nexenta's line of enterprise storage products, www.nexenta.com), and there is at least one good open-source distribution built around illumos: OpenIndiana. We're going to use OpenIndiana to avoid all of you having to sign up for accounts with Oracle for virtual machines that we won't be able to patch or maintain.

And this is only a small part of Solaris' long history! It superseded a far more loved BSD-based operating system, called SunOS, and suffered from multiple version numbers and multiple brand names for most of its life. You will **still** see Solaris and its derivatives advertise their guts as "SunOS," to this day. Even the OpenIndiana distribution we're about to install does too, twice removed from the old Sun Microsystems.

```
$ uname -a
SunOS openindiana 5.11 illumos-1edba515a3 i86pc i386 i86pc
```

I recommend light reading of https://en.wikipedia.org/wiki/Solaris_(operating_system) … especially the version history section. If you'd like to try the official Solaris operating system, please go for it … but you'll be graded on OpenIndiana. Oracle Solaris downloads can be found here …

https://www.oracle.com/technetwork/server-storage/solaris11/downloads/index.html

In this lab, not only are we going to explore more canonical AT&T-ish Unix, but we're going to go through the pain of resolving user ID conflicts and collisions. As all our systems subscribe to NFS (except OpenBSD, by design), all these systems must have user IDs match for all interactive users to access their content. OpenIndiana is the first operating system that doesn't use UID 1000 for the first non-root user. We'll need to reboot into single-user mode to fix this.

Those of you on ARM Macs auditing or also enrolled in CS 596 probably remember my attack demonstration failing just before spring break. It turns out there's a bug with networking in UTM version 4.6.x that affects most CPU architectures in UTM ... but ARM64 is not one of them. That's why we haven't noticed it yet over here in CS 470, and this why it caught me by surprise during that demonstration.

We'll be running an Intel version of OpenIndiana here, and the 64-bit Intel architecture is affected by the bug. You'll want to download and install UTM 4.5.4 from https://github.com/utmapp/UTM/releases to avoid network issues between your lab 5 VM and the rest of your CS 470 VMs. Gracefully shut down your VMs before you re-start them all in the older version of UTM. They should work just fine without needing to export and re-import them.

part zero: get it

Grab your wingtip shoes and twirl your handlebar moustache to web your way to the OpenIndiana download page and download the text installation DVD ISO for OpenIndiana "Hipster" 2024.10; `OI-hipster-text-20241026.iso` takes up just under 1 GB of storage.

The OpenIndiana team's web server has been flaky of late; I've posted a copy of this ISO file in the ISOs folder of the shadow website. If you have issues getting it from the OpenIndiana project's site, don't hesitate to grab it from me.

part one: install it

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

- guest OS:
  - Intel/VMware: Solaris 11 64-bit
  - UTM: select "emulate" on the first screen, then "other" for the operating system
- firmware:
  - Intel/VMware: legacy BIOS
  - UTM: disable "UEFI Boot" on QEMU VM settings pane
- CPU:
  - Intel/VMware: single virtual core/processor
  - ARM/UTM: x86_64 architecture, "Standard PC (Q35)" system, single core
    also on ARM/UTM, select Intel Core i7-9xx … (Nehalem-v1)

- RAM: 4 GB (4096 MB)
- hard disk: 30 GB

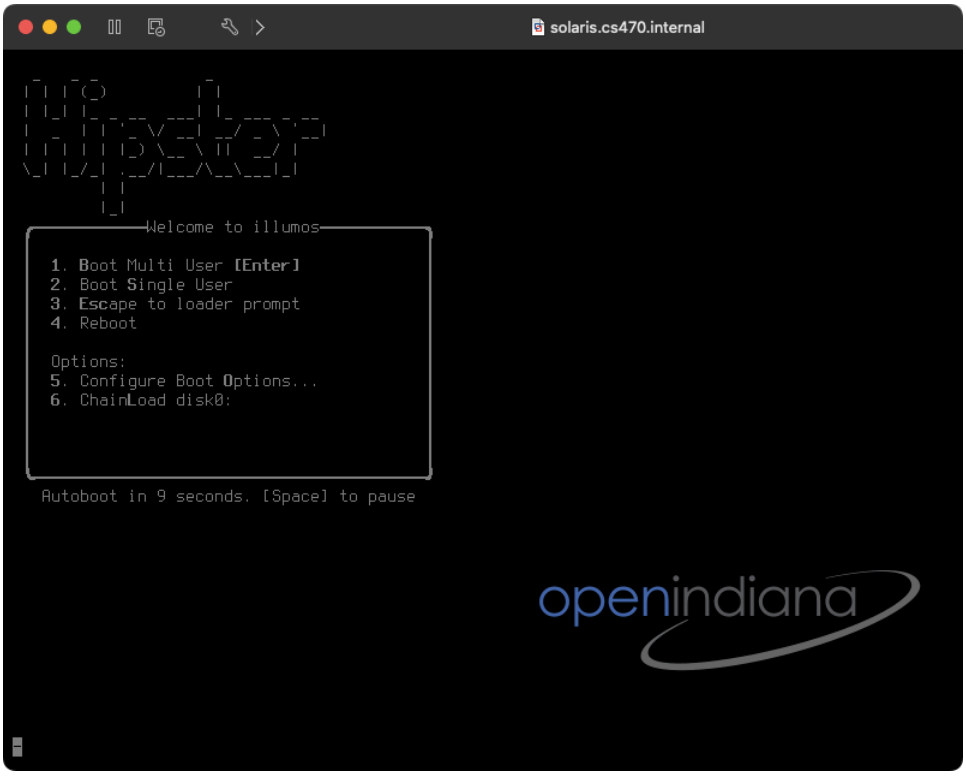
ARM Mac / UTM people: disable your VM's USB support, set your display card to virtio-vga, and your network card to an e1000, if they're not configured that way already.

Though its Solaris ancestor was widely regarded as the first and best at symmetric multi-processing in the Unix world, OpenIndiana seems to have lost some of that magic. You really **do** need to make sure you're only using one core, or OpenIndiana **will** hang or panic during boot.

‼ IMPORTANT NOTE: yeah, OpenIndiana's recommended minimum is 4 GB of RAM. If you're going to be running short on RAM during this class, it's after labs 5 and 6. If that happens, you can suspend or shut down your FreeBSD and Ubuntu VMs. Your OpenBSD and Rocky VMs will be necessary, for DNS, LDAP, and NFS.

!! ALSO IMPORTANT NOTE FOR ARM MACS: those of you on ARM Macs, you're emulating a 64-bit Intel VM because it and OpenIndiana were the best choice here to keep you with the pack. Like I warned you at the beginning, there will be three labs in which you make concessions to use your ARM Mac. Here, this is because the ARM port of the illumos kernel is not yet mature, and there's only one old remaining publicly-distributed version of a 32-bit Intel Solaris derivative out there (see www.tribblix.org). It isn't a workable alternative.

1. While booting up the VM for the installation … does the boot loader menu look familiar?



When Sun open-sourced Solaris, FreeBSD (in its desire to be the best free server out there) grabbed Solaris' acclaimed ZFS filesystem and was the first non-Solaris operating system to be able to boot off a ZFS filesystem. In return, many years later, OpenSolaris poached FreeBSD's boot loader … because the licensing terms were properly free (BSD) ... and it could already boot off ZFS!

You should select to boot multi-user … or you will time out into selecting it as the default option, which is also fine.

At this point in time with OpenIndiana as recently as 2019, you used to receive a little text-based parade through history … the kernel would identify itself during boot as `SunOS release 5.11`, also known as Solaris 11, and Solaris 11's open-source kernel fork would show itself in the kernel version string as `illumos`. You'd also see a very long range of dates in the copyright notice and were readily able to see this as a descendant of the original SunOS from way back in 1982, if you look at that Unix family tree from levenez.com again.

Though the kernel now identifies itself as `illumos-a8cd71ddc9`, and the operating system as OpenIndiana Hipster 2024.10. The illumos kernel seems to have shed itself of the advertising baggage associated with the old OpenSolaris license. For many years it referenced various prior brand names, including Sun, SunOS, and Oracle, but now none of them … until you get to a command line and run `uname`.

In the first step after booting, select your language and keyboard layout, as usual!

After selecting your language, the illumos kernel will go detect and "configure devices," then you'll get a menu. Door number 3, a shell, is to provide you a recovery environment if you ever should need it. Go ahead and bust open a shell right away and run `top`. Since OpenIndiana wants 4 GB, we're going to watch RAM use very closely. As usual, we'll hope to take some memory back later. It's not using much now, but when we get to playing with the über-bloated Solaris package manager later on, we'll be needing all of it.

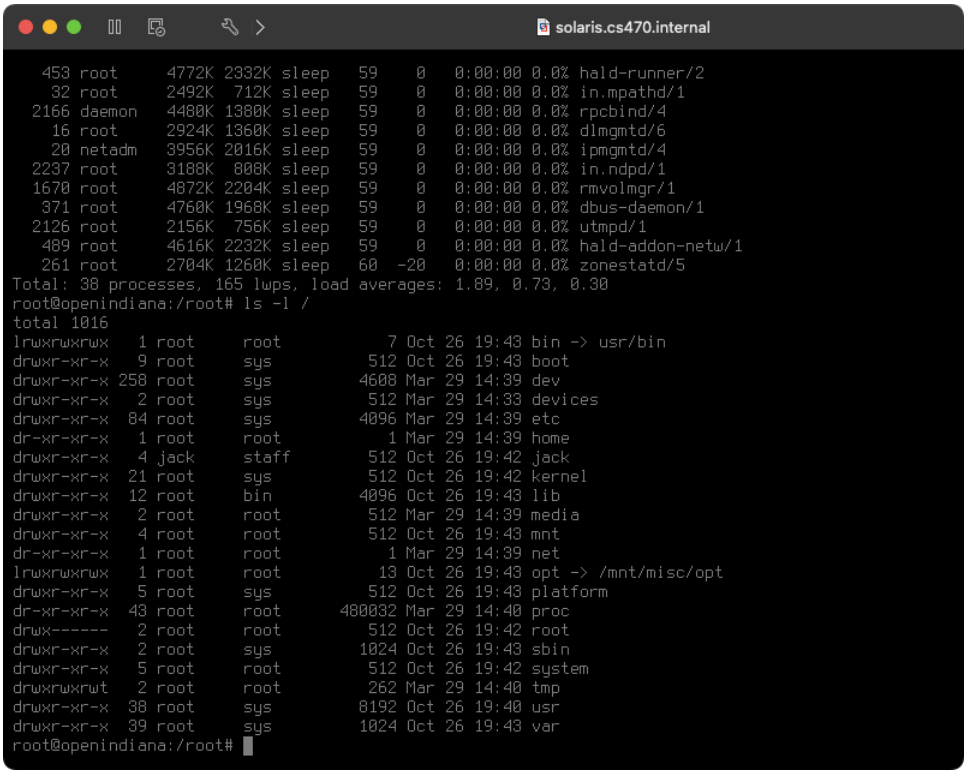
‼ Solaris also has its own neat version of `top`, called `prstat` … go ahead and try it!

First, run `uname -a` and run there, you can see it, the old Sun Microsystems roots of OpenIndiana …

```
root@openindiana:/root# uname -a
SunOS openindiana 5.11 illumos-a8cd71ddc9 i86pc i386 i86pc
```

… it still identifies as SunOS 5.11 (AKA Solaris 11) and calls the Intel x86 "i86pc" like only Sun Microsystems ever did, and always has, to distinguish from its early Intel-based but not PC-compatible systems.

If you `ls -l /` you'll easily be able to see that the username for the regular (non-root) user to run the installer, also the username used by the larger live ISO and graphical installer, is `jack`. Coincidentally, my son is named Jack, and I had a good friend named Jack who is a great Unix story himself … more on that later.



Why OpenIndiana or OpenSolaris might have an affinity for people named Jack, I cannot find conclusive evidence, but Jack has been the default non-root user on installation media going as far back as 2009. As best as I can tell from web searching, this is either a reference to an early 2002 video game or to a YouTuber trucker … nothing in OpenIndiana's documentation explains why they chose this username.

Run `df -h` and note the output … Solaris has a `/devices` tree, a virtual filesystem that contains a further abstraction logically **below** the `/dev` file tree. Almost everything in `/dev` (run `ls -l /dev | more`) is actually a softlink to the corresponding real device file under `/devices`. Also, look at (`ls`) the contents of `/dev/dsk` … this is where the disk devices (or at least, their abstraction layers) live, using the far more verbose AT&T-ish disk device naming format.

For instance, for the partition called `/dev/dsk/c1t0d0s2`, we're referring to controller (`c`) one, target (`t`) zero, disk (`d`) zero, and slice/partition (`s`) two. Note: precisely like BSD Unix, slice zero is almost always the root filesystem, slice one is almost always a swap area, and slice two (`c` in BSD Unix) is **by convention** a virtual "overlap" partition used to refer to the **entire disk** with a single device reference, even if it's got multiple partitions.

‼ IMPORTANT NOTE: You never, **ever**, ***ever*** want to make a filesystem on slice two (`s2`) of an AT&T-ish Unix, or the `c` partition of a BSD-ish Unix disk. Immediate and great pain, suffering, and data loss are a virtual lock to follow. Read the paragraph above again a second time for good measure and the appropriate amount of self-flagellation for lessons that dispel the possibility of data loss.

Also note, from the output of `df`, that Solaris and its descendants interchangeably use swap spaces and memory-based filesystems. When booting off read-only media like an optical disc (or an ISO file pretending to be one), we need a few areas of the file namespace to be read/write … notably, `/tmp`

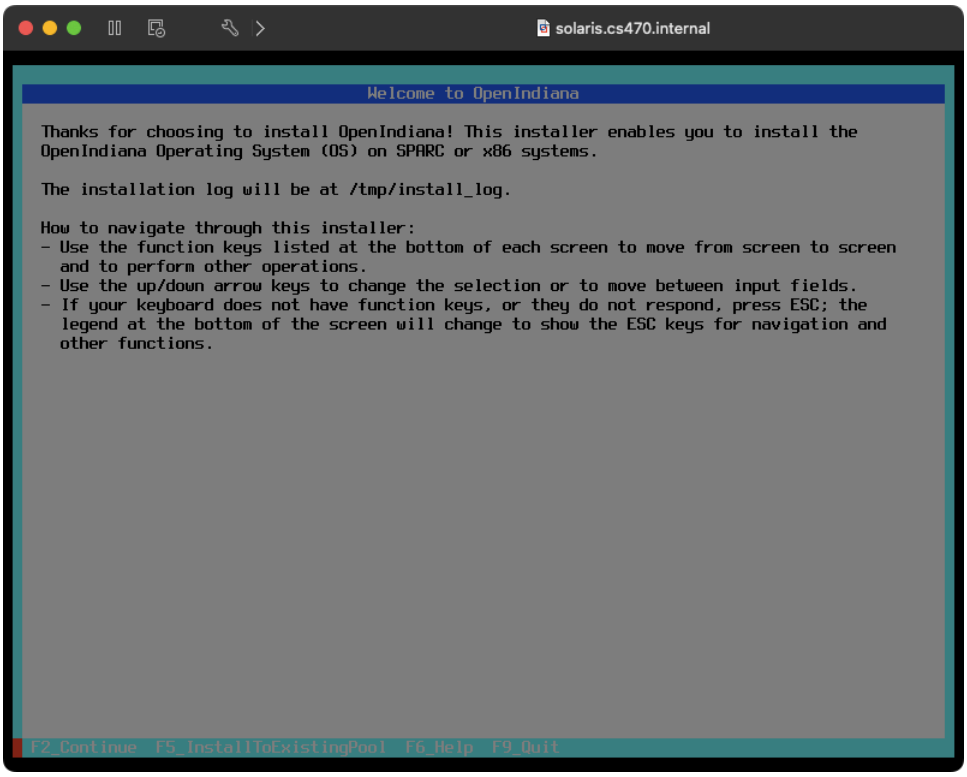
and `/var/run` (like a lot of Unix-ish OSs), home directories for users, `/var/pkg` for package tools, and `/etc/svc/volatile` for service configuration information.

2. `exit` the shell to go back to the install ISO's menu.

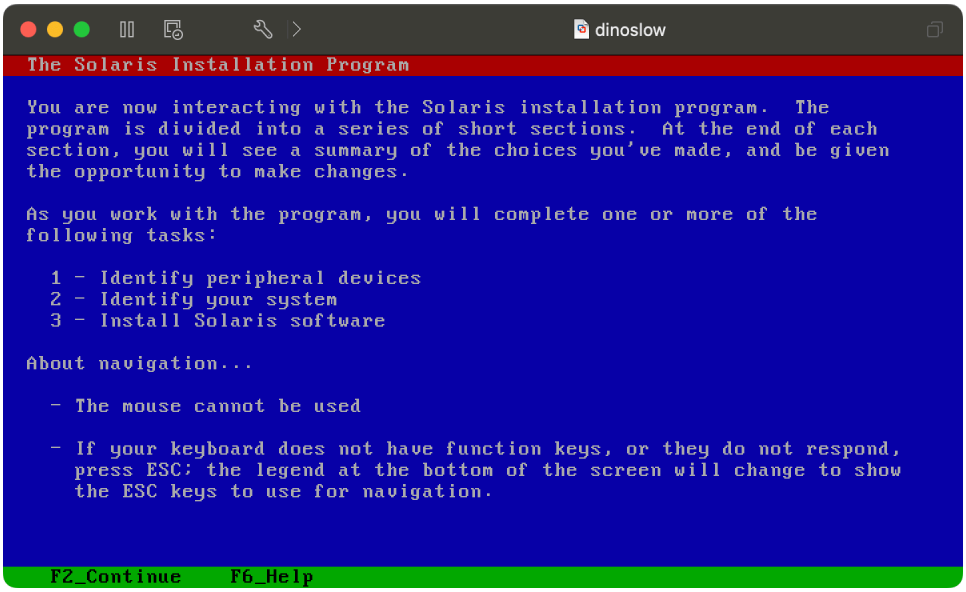
   First, take a moment to make sure that the terminal type (option 4) is set to `sun-color`, especially if you're on an ARM Mac. I know it's not one of the listed options, but manually key it in. If the installer runs while `$TERM` is not set to `sun-color`, it'll be pretty painful, as menus, borders, and input fields won't draw correctly. If you somehow get stuck with things not drawing correctly, make sure to read the notes in red in this step carefully, and lean on `control-L` to make sure you can clearly see what you're doing.

   !! REMINDER: `control-L` is a common way to ask your terminal to re-draw the contents of the window, if log entries should spontaneously pop up over that bottom line of the screen and obscure the hot keys, or if you have issues with character output on the console in UTM.

   Now let's go through door number 1, the installer. You'll soon be welcomed to OpenIndiana's text-based installer. The design cues give it away to those of us who have been playing with Solaris for decades …



   … the colors are different from the red and blue of the long-time Solaris text-based installation dialogs, but the insistence on using function keys for motion and putting those function key options in a colored margin at the bottom of the screen totally give it away. OpenIndiana's text-based installer is a direct code descendant of the original Solaris text-based installer. A screenshot from Solaris 2.4's text-based installer, released in 1994, is shown just below; you be the judge.



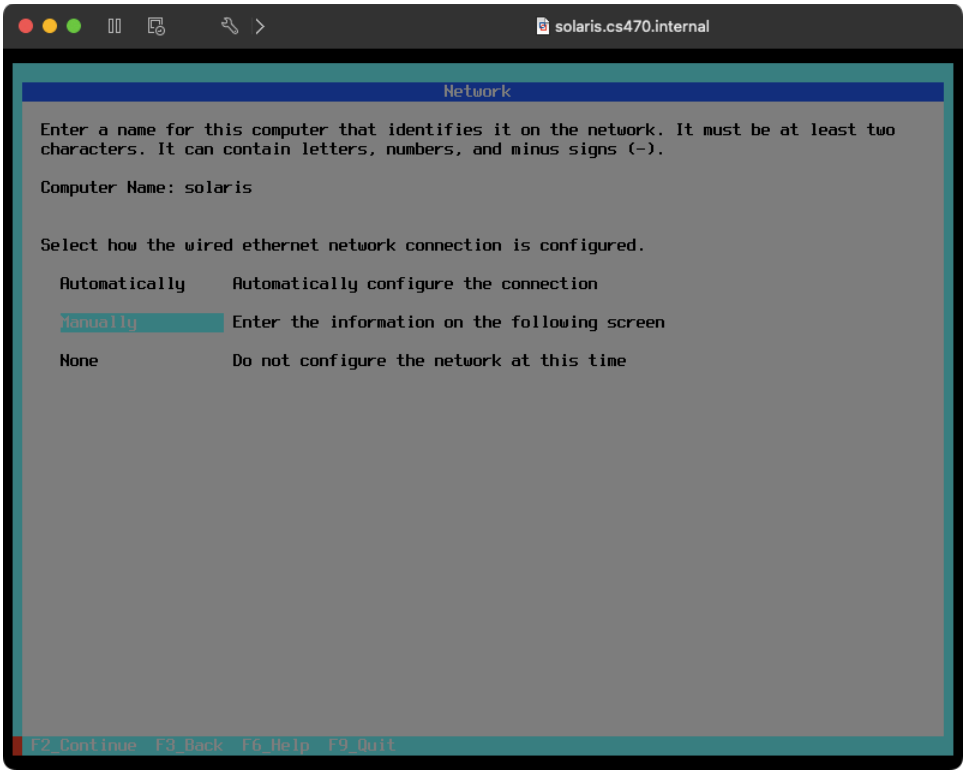
!! NOTE: You really **DO** have to use the function keys here to move around and to complete forms on the screen. Those of you on Mac keyboards may, depending on your Mac's configuration, have to use the `Fn` key to make sure your VM's console sees keystrokes as function keys. **Sometimes you'll see the command tips in the bottom change to Esc key combinations … they will both always work.** Escape keys are a concession to allow keyboards that don't have function keys to work.

Hit `F2` to continue and you'll get to the "disks" screen.

If you have issues with text-based menus not displaying as they should on UTM, be patient and keep in mind that your insertion point is initially at the end of default input strings. You can backspace to delete, and make liberal use of `control-L` – don't hesitate to spam it – to re-draw the window.

3. Choose your VM's solitary virtual hard disk … of course, there are no other options on the "disks" screen for installation targets. `F2` again, to the partitions screen. If you see a warning about the loss of all existing data, fear not, it's just your VM. Choose to use the whole disk … we're not going to do anything besides OpenIndiana with it. `F2` again. Lather, rinse, repeat.

4. Next, the "network" screen. For your computer name, `solaris`. Then use the down arrow to select a manual setup, and hit `F2` to continue as usual.



You should know what to do on the manual network configuration at this point in time, and you should be able to easily look up this VM's IP address in DNS or the prior labs, if it isn't seared into your brain already from all the data entry to date.

If you have issues with the installer taking the network configuration, or completing the installation with the network configured, choose "None" and configure the network immediately after the first reboot using the information in appendix two.

For years, we used the graphical installer for OpenIndiana, which offered no ability to set up the network during the installation. Then, it always made us go through a dozen-step incantation (preserved in appendix two) to set a static IP address, post-installation. You can kind of see their logic … if you're using a graphical desktop, it's probably a workstation, and a workstation likely doesn't need or want a static address.

5. Next, the ever-important time zone selection. Choose your region on the first page, `F2`, and keep iterating until you find your time zone and pick it. I don't care if you choose to be in Vanuatu, so long as your VMs can talk to each other.

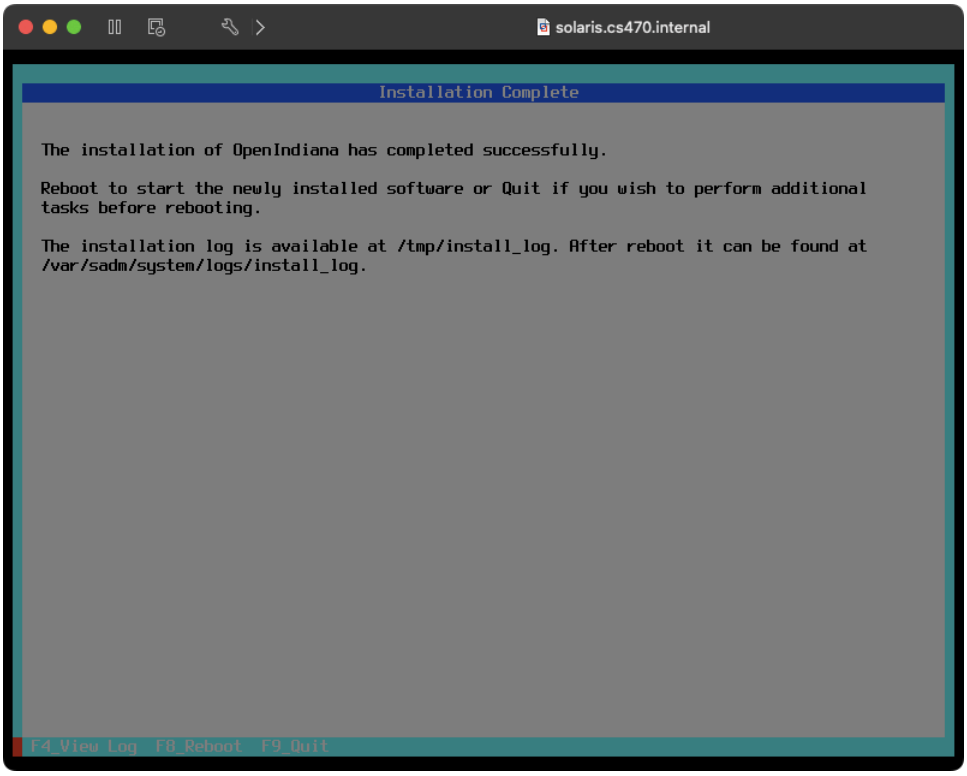
6. Check the system clock, adjust if you feel like playing with it, but we'll be setting up NTP after we get the operating system installed, as usual.

7. On the next screen, set a root password, and create your failsafe user account.

   Review and confirm your installation settings and hit `F2` to begin the installation.

   You should have a progress bar soon afterwards, and then installation will take a short while. **For**

those of you on ARM Macs, this short while is going to be a little while longer for you. Installation is one of those tasks where you're really going to be slowed down; it might take up to a couple hours. Go grab a bite to eat or take a break once you see the progress bar.



When it's done, take a second to look at the installation log; it is not very long at all. In particular, note the disk and filesystem configuration lines … OpenIndiana uses ZFS by default, which will use multiple name-spaces within a single actual disk partition or slice, very much like Apple's new-ish APFS filesystem does.

Also, please note that we didn't configure swap space. When you're done with the log, go back and reboot.

8. The first time a modern Solaris boots up, it initializes the `smf` (Service Management Framework) database. As discussed briefly in lecture, `smf` is the Solaris replacement for the old AT&T Unix system of `init.d` scripts. This initialization process takes a little while, but this is fortunately a one-time thing.

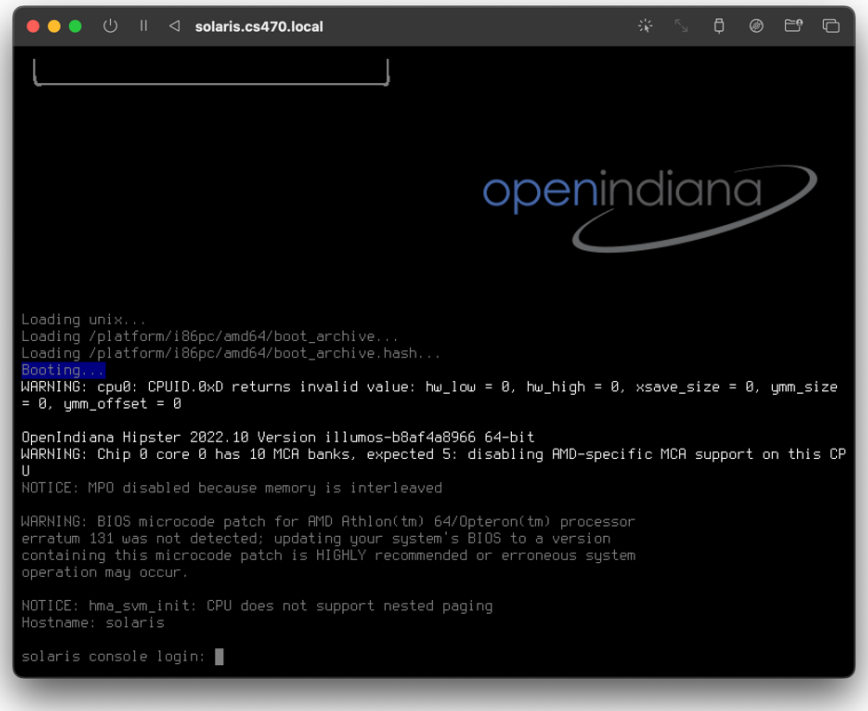
   ‼ IMPORTANT NOTE: it may seem like I use the terms "Solaris" and "OpenIndiana" interchangeably here, but I do not. OpenIndiana is a subset of Solaris. Most of commercial Solaris was open-sourced, not just the kernel (illumos), but the base operating system and userland as well. OpenIndiana is not just an illumos kernel … it's a direct descendant of Solaris, and it hasn't diverged much yet.

   After a minute or two – or a few more for ARM Mac users – you should see a login prompt.

   ARM Mac users: if you missed some of the directives about how to set up your VM, the kernel may pop off all kinds of notices and warnings now that it's booting up into a full installed operating system

instance, about CPU details, about unusable USB adapters, and more. You can ignore them, unless your VM completely hangs and fails to make it to a boot prompt after five minutes or so.

If you see any log messages pop out over your login prompt, worry not. Just like with Ubuntu's first boot, give it a carriage return (enter key) to get a new, fresh one.



Now that you have OpenIndiana installed, it's time for the obligatory reminder about shutting down your new VM gracefully, and telling you how to do it on this new operating system. OpenIndiana is still a canonical System V ("System Five") AT&T Unix, and while it has a `shutdown` command, it uses a different syntax than used in BSD, or Linux, which uses a BSD-ish implementation of `shutdown`. Whereas one might typically invoke BSD `shutdown` with …

```
$ shutdown -h now
```

… or `-p` to kill power after completion (the reason BSD didn't have a separate `poweroff` utility), on an AT&T-ish Unix, the same command would typically be …

```
# shutdown -y -g 0
```

… the `-y` is to bypass a confirmation you'd like to shut down the computer, the `-g` is to denote a grace period, and `0` is for the number of seconds in that grace period (none, now). Note the `#` prompt; there isn't typically a group that's associated with granting the rights to run `shutdown` in AT&T-ish Unix, so you'll either need to run it as root or with `sudo`. Some System V implementations of `shutdown` don't want a space between `-g` and the number of seconds, so as always, I wholeheartedly recommend you look at `man shutdown` before you run it.

The `reboot` and `poweroff` utilities are both here as well; as I believe I mentioned in a prior lab, the `poweroff` utility originated with System V AT&T Unix.

part two: configure it

If you had to defer network configuration until after installation, go ahead to appendix two before continuing.

9. At this point, you should be able to SSH into your new VM, and set up key-based authentication in your failsafe user's account. Do it.

10. Let's get our home directory over here to our new VM via NFS.

```
$ sudo mount rocky:/home /home
nfs mount: mount: /home: Device busy
```

Uh oh … `/home` is spoken for. It turns out `/home` isn't actually where your home directory is, because Solaris expects your system administrator to consider exporting it …

```
$ echo $HOME
/export/home/failsafe
```

… and `/home` itself was "busy." Turns out, Solaris is running a service called the automounter (also sometimes called `autofsd`), that we've talked about but are not going to use. Since we're going to "import" `/home` over NFS rather than export it, let's get it out of the automounter's configuration so it's not busy …

```
$ sudo vi /etc/auto_master
```

… comment out the line in `/etc/auto_master` starting with `/home` … and then let's send it the HUP signal to tell it to reprocess its configuration files.

```
$ sudo pkill -HUP automountd
```

`pkill` searches for all instances of a process matching a token (in this case, `automountd`), and sends it the specified signal. The equivalent in *BSD and Linux is usually `killall`. Now let's try to mount that NFS share again …

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

… success, I sure hope! Before we can mount `/srv/nfs`, we'll need to create that directory tree first. Remember, the `/srv` file tree is a Linux innovation.

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

Now make it permanent …

```
$ sudo vi /etc/vfstab
```

… note that Solaris and its derivates put their filesystem table in a different place. Also note how a switch is provided on this operating system to allow a filesystem to be registered system-wide in this

file, but not be mounted at boot time. Also also note how swap is mounted on `/tmp` and a swap partition is additionally mounted from disk. This guarantees that the contents of `/tmp` are transitory and will be gone after a reboot; they are stored only in virtual memory. Add these lines at the end:

```
rocky:/home     -               /home           nfs     -       yes     rw,nosuid
rocky:/srv/nfs  -               /srv/nfs        nfs     -       yes     rw,nosuid
```

The `nosuid` option disallows the setuid and setgid flags on the mounted filesystem … which is handy for security, so that you can block an unknown somebody from placing programs that allow for privilege escalation. An archetypal security coup de grace would be placing a setuid, root-owned shell on another computer. Without this option in place, you're effectively trusting the security of the NFS file server … a lot.

You can test your edit to `/etc/vfstab` by unmounting and re-mounting `/home` and `/srv/nfs`. If you're specifying a mount point as the argument to mount, generally mount will check its filesystem table to determine the filesystem to use, and vice-versa.

```
$ sudo umount /home
$ sudo umount /srv/nfs
$ sudo mount /home
$ sudo mount /srv/nfs
```

If you get no errors, awesome!

11. Strap in, folks … I've looked a few times at how to break up this step … but decided against it, each time. As of the current version of me editing this sentence right now, the end of this step is several pages away, but you learn some important lessons here.

    Sometimes having user IDs synced up between machines is really, really important. Any time you share filesystems in between systems with NFS is one of those times, and we are doing just that. First, we need to fix our home directory …

    ```
    $ grep failsafe /etc/passwd
    failsafe:x:101:10:failsafe user:/export/home/failsafe:/bin/bash
    ```

    … you may notice a couple problems here, but I'm not going to make you figure it out this time. Not only is our home directory **not** under `/home`, but our user ID is **not** 1000 like on our other VMs, and the default group ID doesn't match. The `id` command is pretty handy.

    Also, run this command …

    ```
    $ ls -l /home
    ```

    Note how the failsafe user doesn't have ownership of its own home directory here, because of the user ID mismatch. In order to fix this, we're going to have to log out … we're in the affected user account, and we need to change it all at once, from another account.

    Try to log in as root on the console, inside the VM's window or tab in VMware. It rejects you, even with the right password!

```
Roles can only be assumed by authorized users
```

OpenIndiana is enforcing proper security protocols on us, by not allowing us to log directly into what it considers to be a "role" account. Remember how we log in as ourselves first, then use `sudo` to do things as root? This is the same principle being applied here.

Because we don't have another user on the system besides root and the account we need to fix, we're going to need to reboot into single user mode. Sorry, popping a root shell with `sudo` just won't work for this.

‼ IMPORTANT NOTE: we're about to change a user's ID #; when you do this, you typically want to make sure the user is logged out and no processes are even running on their behalf (`cron` jobs?). Fortunately this is a fresh account with little in it. That said, going into single-user mode is still probably the best way to do this.

Reboot your Solaris VM, and when the boot menu comes by, choose option 2 to boot into single-user mode. After you do that, it will ask you to log in to do maintenance … **this** is the only place your root login will work.

After logging in, if you look at `/export/home` (with `ls`, of course), there's nothing there, of course. The ZFS root "rpool" has been imported, so that the kernel is aware of the filesystems in our ZFS partition. However, only the root rpool has been mounted from the ZFS filesystem. Use `zfs list` to display all the namespaces inside your ZFS filesystem; it should look fairly similar to my output.

```
$ zfs list
NAME                          USED  AVAIL     REFER  MOUNTPOINT
rpool                        8.34G  20.2G     33.5K  /rpool
rpool/ROOT                   4.21G  20.2G       24K  legacy
rpool/ROOT/openindiana       4.21G  20.2G     3.56G  /
rpool/ROOT/openindiana/var    659M  20.2G      657M  /var
rpool/dump                   2.00G  20.2G     2.00G  -
rpool/export                   76K  20.2G       24K  /export
rpool/export/home              52K  20.2G       24K  /export/home
rpool/export/home/failsafe     28K  20.2G       28K  /export/home/failsafe
rpool/swap                   2.13G  22.4G       12K  -
```

Fortunately, all we want to mount are `/export`, `/export/home`, and `/export/home/failsafe`. Even if we're moving our home area to `/home`, it would be sufficiently disruptive to a user to NOT own their own files … so we're going to fix them all, across the whole file namespace.

First, let's mount all the filesystems.

‼ IMPORTANT NOTE: be very careful … we're at a root shell for a while! Typos can be costly‼

```
# zfs mount rpool/export
# zfs mount rpool/export/home
# zfs mount rpool/export/home/failsafe
```

Then, use `mount` to mount everything from the filesystem tables, including our home directories from over NFS …

```
# mount -a
```

… and then confirm they're all mounted.

```
# df -h
```

Now, let's edit `/etc/passwd` …

```
# vi /etc/passwd
```

Go to the line for your failsafe user, and change its user ID to 1000, to be in line with all the other VMs, change your primary group ID to be 1000 too (which it should be on your other VMs as well), and remove the leading `/export` from your home directory so that we go to NFS for the home directory …

before:
```
failsafe:x:101:10:failsafe user:/export/home/failsafe:/bin/bash
```

after:
```
failsafe:x:1000:1000:failsafe user:/home/failsafe:/bin/bash
```

… and save `/etc/passwd`. Now, let's edit `/etc/group` …

```
# vi /etc/group
```

… in here, because failsafe's primary group was staff (10), we'll add failsafe to the staff group by putting it at the end of the line starting with "staff." There shouldn't be another user there, but if there is, add `failsafe` after adding a comma in to separate the usernames. Also add the following line to the end to properly add a failsafe group like on our other VMs. Solaris and its derivates just haven't yet gotten on-board with creating a group for each user; this is a relatively new practice in the bigger picture of things, and it really originated in Linux.

```
failsafe::1000:
```

When you get out of `vi`, run …

```
# ls -l /export/home/
```

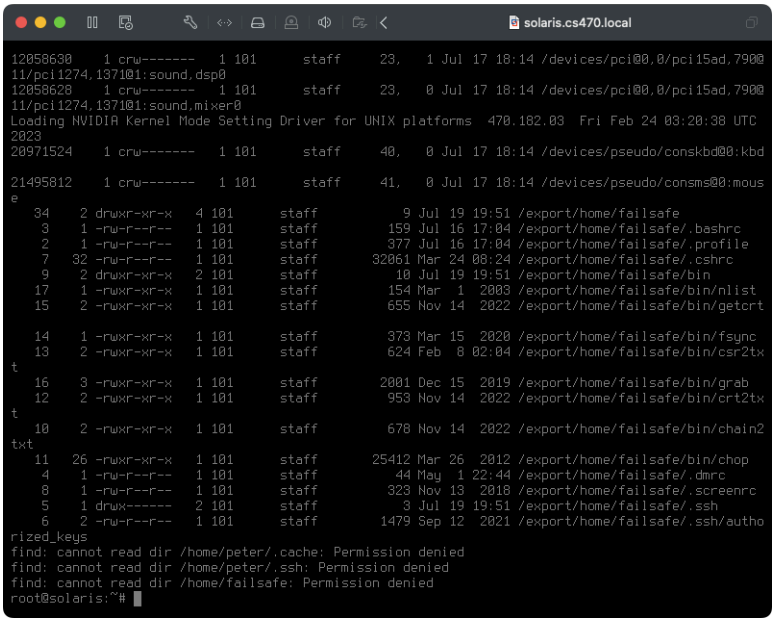
… and note that now, the failsafe account is no longer associated with the home directory that got created for it by the OpenIndiana installer, because that filesystem object is still associated with user ID 101, and failsafe is no longer user ID 101 anymore as far as your OpenIndiana VM is concerned. Now, let's fix this … with the `find` command.

```
# find / -user 101 -ls | more
```

This command tells `find` to look from the root of the filesystem for everything with user ID 101, and to give the long output of `ls` for everything that matches. If you get some complaints from NFS, don't worry about them too much … nothing in NFS should be owned by user ID 101, right?

At times, I've seen a few things outside the failsafe home directory, notably under `/devices`, that also had its ownership changed to user ID 101. Some of it, device files associated with the console, make

sense. Some of them, hard disk devices … should not be. I wondered if I'd just found a problem with OpenIndiana's installer. The point-and-click installer didn't do this, and the 2023.10 and 2024.04 versions of the operating system didn't do this either. The below screen shot is preserved to show off this older anomaly.



Let's get to fixing some files … first, let's change the user ID of everything matching our old account …

```
# find / -user 101 -exec chown -h failsafe {} \; -and -ls >
/export/home/failsafe/chown.out
```

… this command tells `find` to run (`-exec`) `chown failsafe` on all the files matching user ID 101. The curly brackets are where the name of each file matched by `find` goes into the `-exec` command. `-h` makes `chown` change the ownership of symbolic links too, rather than following them to what they point at. The backslash and semicolon terminate the `-exec` command … if we didn't escape out of the semicolon with the backslash, it'd be interpreted by our local shell, and terminate `find` rather than the command `find` is running. Also, instead of having `find` continue, we are asking it to also (`-and`) run `ls` on the file (`-ls`). The greater-than (`>`) sends the output to the file you name.

I got an error that `find` couldn't read `/home/failsafe` and `/home/peter`, and **you should get it too and know why it's happening** at this point.

When you look in that file you sent output to, you'll note that the change of ownership isn't made before the `ls` is run … this is annoying, but it is what it is. You can check that it worked another way.

```
# ls -la /export/home/failsafe
```

Before we bail out of single-user mode, we need to fix one last thing. Our failsafe account needs a home directory, and if for some reason NFS fails … we'd be able to log in as `failsafe`, but we would not have a home directory, and that could cause some annoyance. Fortunately, it's easy to fix.

We've told OpenIndiana the home directory is at `/home/failsafe`, so if NFS isn't on `/home` … the

contents of `/home` that would ordinarily be hidden underneath the mount point will be there, as we've seen before, and we can use that to our advantage. Unmount `/home` and provide a symbolic link to `/export/home/failsafe` from `/home/failsafe` …

```
# ln -s /export/home/failsafe /home/failsafe
```

… you can test by making `ls` evaluate where `~failsafe` is …

```
# ls -la ~failsafe/
```

… if you omit the trailing slash in the command above, you'll see the `ls` output for the symbolic link, rather than the files that demonstrate this workaround actually works. Your failsafe account will always have a home directory, whether or not `/home` is mounted from NFS. Just remember that your files might be in two destination folders while you're troubleshooting in that account, if NFS isn't working.

We're done with the big step! Hit `control-d` to exit single-user mode, and the system will complete a full multi-user boot. (This is common, by the way, using `control-d` to exit single-user or to log out, for virtually all Unix OSs.)

12. Go ahead and set a root password; grading will not work later if we don't do this again.

```
$ sudo passwd root
```

13. OpenIndiana's service framework, inherited directly from canonical Solaris, needs us to enable all of the NFS client's dependencies in `svcadm` to reliably mount an NFS share at boot time.

```
$ sudo svcadm enable -r nfs/client
```

This command spit out this output, which was merely informational:

```
svcadm: svc:/milestone/network depends on svc:/network/physical, which has multiple
instances.
```

After a reboot, `/home` will now mount nicely and automatically at boot time, provided your NFS server is up, of course.

14. !! READ OR SKIP BUT DO NOT DO THIS STEP, UNLESS YOU'RE FEELING ADVENTUROUS. I WAS UNABLE TO GET LDAP TO WORK WITH OPENINDIANA, EVEN WITH THE HELP OF A FRIEND WHO USED TO WORK ON THE LDAP AND X509V3 (CERTIFICATES) CODE AT SUN/ORACLE.

I'll give away a bunch of extra credit if you're able to get this to work, but I had to give up. The Solaris LDAP client is its own code, with its own configuration tool. There's a lot of documentation about the Solaris LDAP client and its configuration tool … a lot of really bad documentation. There's even more discussion out there as to how deeply broken it is, than there is documentation of it, and that's actually saying something.

Oracle's master documentation for setting up the LDAP client in Solaris 11.4:
https://docs.oracle.com/cd/E37838_01/html/E61012/clientsetup-1.html

Troubleshooting documentation:
https://docs.oracle.com/cd/E37838_01/html/E61012/setupproblems-1.html

Oracle's HTML version of the man page for `ldapclient`, the LDAP client configuration tool:
https://docs.oracle.com/cd/E88353_01/html/E72487/ldapclient-8.html

Let's add our CA certificate to OpenIndiana's trust store. You should know how to get a copy of your CA certificate from lab 1b over here by now, if you don't already have a copy of it in your failsafe user's NFS home directory like I do.

Make a copy of it, and in that copy, remove everything before `-----BEGIN CERTIFICATE-----` and anything after `-----END CERTIFICATE-----` …

https://docs.oracle.com/cd/E53394_01/html/E54783/kmf-cacerts.html

… Solaris' documentation states that some applications are not able to handle the extra text; we best be safe.

Use `sudo` to copy the file to `/etc/certs/CA/cs470.internal.pem` and then the following command to cause OpenIndiana to re-parse its trust anchor directory.

```
$ sudo svcadm refresh /system/ca-certificates
```

Check the file `/etc/certs/ca-certificates.crt` to make sure it has an updated modification date, and `grep` it for `CS470` to see that it contains your CA certificate. As the Solaris documentation link above states, you should also see a symlink to your certificate in `/etc/openssl/certs`.

Use `sudo` to make another copy of the certificate file, to `/var/ldap/certdb.pem`.

Set up OpenLDAP in `/etc/openldap/ldap.conf` … you should know what to do here. Aim `ldap.conf` at the certs database in `/etc/certs`.

Enumerate services with LDAP in their name in the service management facility.

```
$ svcs \*ldap\*
```

Check the status of the LDAP client with the service management facility.

```
$ svcs -l network/ldap/client:default
```

Enable the LDAP client in the services framework.

```
$ sudo svcadm enable /network/ldap/client:default
```

Add `dns` before `files` to the `hosts` and `ipnodes` lines in `/etc/nsswitch.ldap` on your Solaris VM.

Put `cs470.internal` into the file `/etc/defaultdomain`.

Solaris' LDAP client also ties itself up with stubs left over from NIS. This was surprising with OpenBSD, but isn't surprising from the remnants of Sun, who developed NIS. This was them eating their own dog food.

```
$ sudo mkdir /var/yp/binding/cs470.internal
```

Initialize the LDAP client's state and auth files.

```
$ sudo touch /var/ldap/ldap_client_cred /var/ldap/ldap_client_file
```

These pages contain sample syntax for `/var/ldap/ldap_client_file` …
https://docs.oracle.com/cd/E19253-01/816-4556/clientsetup-1/index.html
https://docs.oracle.com/cd/E19253-01/816-4556/clientsetup-49/index.html

Solaris and its derivatives include a utility for setting up the operating system's built-in LDAP client for authentication. Even though it's not supposed to need a username and password for the LDAP server in "anonymous" mode, the tool won't run if you don't provide credentials.

```
$ sudo ldapclient -vvv manual -a authenticationMethod=tls:simple  -a
certificatePath=/var/ldap/certdb.pem -a credentialLevel=anonymous -a
domainName=cs470.internal -a defaultServerList=openbsd.cs470.internal:636 -a
defaultSearchBase=dc=cs470,dc=internal -a adminDN=cn=admin,dc=cs470,dc=internal -a
adminPassword=r3dact3d
```

I gave up again.

An alumnus of the class got really, really close to getting this working last fall, so I'm passing along his knowledge here, which replicated some portion of how close I was able to get.

What he ended up doing:
a. Create an NSS db. The `certutil` utility is apparently in the latest version of `system/library/mozilla-nss`, but he couldn't install this so I just used the `libnss3-tools` package on Ubuntu.

```
$ certutil -A -n cacert -t "CT,CT,CT" -d db/ -i cs470.internal.crt
```

b. Configure `ldapclient` with proxy authentication; it claims to support anonymous authentication, but nobody can get it to work.

```
# ldapclient -vvv manual -a domainName=cs470.internal -a credentialLevel=proxy -a
defaultSearchBase=dc=cs470,dc=internal -a adminDN=cn=admin,dc=cs470,dc=internal -a
adminPassword=YourPassword -a enableShadowUpdate=true -a
authenticationMethod=tls:simple -a certificatePath=/var/ldap -a
proxyDN=cn=admin,dc=cs470,dc=internal -a proxyPassword=here -a
defaultSearchScope=sub -a defaultServerList=openbsd.cs470.internal:636 -a
serviceSearchDescriptor=passwd:ou=Users,dc=cs470,dc=internal?one -a
serviceSearchDescriptor=group:ou=groups,dc=cs470,dc=internal?one
```

c. Make configuration files world readable.

```
# chmod 644 /var/ldap/*
```

d. Update PAM configuration in `/etc/pam.conf` (check `man` page for `pam_ldap`).

```
login   auth requisite          pam_authtok_get.so.1
login   auth required           pam_dhkeys.so.1
login   auth required           pam_unix_cred.so.1
## CHANGE
login   auth binding            pam_unix_auth.so.1 server_policy
login   auth required           pam_ldap.so.1
```

Also do the same for `rlogin` auth and `other` auth. For some reason, you will get a password expired on SSH unless you also do this:

```
other   account requisite       pam_roles.so.1
## CHANGE
other   account required        pam_unix_account.so.1 server_policy
other   account required        pam_allow.so.1
```

15. Make a symlink from `/usr/local/bin/bash` to `/bin/bash` … you'll have to make `/usr/local/bin` first.

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

16. Create a local version of your LDAP user. First, create the self-named user group for your LDAP user …

```
$ sudo groupadd -g 1001 peter
```

… then create the user …

```
$ sudo useradd -d /home/peter -g peter -G staff -u 1001 -R root peter
```

… and set that user's password.

```
$ sudo passwd peter
```

You should now be able to SSH into your Solaris VM from as your LDAP user. No further action required, since your home directory is mounted from the Rocky VM, and you properly own all your files again. If you can't, something's wrong … go back and figure it out.

17. Add your failsafe and LDAP user accounts to the group sysadmin (14).

Give the sysadmin group permissions to run `sudo` in `/etc/sudoers`.

18. Network time protocol setup time!

Let's drill in on `ntp` … you can also use `svcs` to search through the database of services.

```
$ svcs ntp
STATE          STIME    FMRI
disabled       16:23:27 svc:/network/ntp:default
```

… and enable it with the `svcadm` command …

```
$ sudo svcadm enable /network/ntp
```

… AT&T-ish Unix is big on its `*adm` commands, as you can tell … and now let's verify it's running …

```
$ ps -eaf | grep ntp
```

… also note here that with AT&T Unix, the `ps` command has an entirely different set of switches, and where we use `aux` or `auxww` (dash optional) on BSD and Linux, we use `-eaf` on Solaris (dash not optional) and other canonical AT&T-ish Unix, including AIX … although AIX is not very canonical, as we'll see in lab 5b.

19. Let's prepare the system for updates and security fixes. I'll spare you some suspense; OpenIndiana's going to run out of swap space here.

```
$ swap -lh
swapfile             dev    swaplo   blocks     free
/dev/zvol/dsk/rpool/swap 272,2         4K  512.00M  512.00M
```

It's got 512 MB of swap. Let's give it 4 GB.

```
$ sudo zfs set volsize=4g rpool/swap
```

After a reboot, all that swap space is accessible.

```
$ swap -lh
swapfile             dev    swaplo   blocks     free
/dev/zvol/dsk/rpool/swap 273,2         4K    4.00G    3.68G
```

20. Now, the package manager. While each Linux went its own way for package management (`apt` and `rpm` are just the most common), BSD went another together (the ports tree or `pkg`), Solaris went in yet another direction all its own, though its tool is also called `pkg` like on the BSDs. Grab the latest status of the publication repository.

```
$ sudo pfexec pkg refresh --full
```

If you're on an ARM Mac, this is going to take a **long while**. After you're sure it's started, take a break and monitor it sporadically.

`pfexec` is being here to run the rest of the command in a "profile," in this case the root "role" ability to modify the operating system and the package registries. That's right, as we saw earlier, root is a "role" here in Solaris and its derivatives, not a user account, and gets extra protections.

I got the following error back from `pkg` …

```
pkg: 0/1 catalogs successfully updated:

1/1 repositories for publisher 'openindiana.org' could not be reached for catalog
refresh.

Unable to contact valid package repository: http://pkg.openindiana.org/hipster/
```

```
Encountered the following error(s):
  Framework error: code: E_MULTI_ADDED_ALREADY (7)
URL: 'http://pkg.openindiana.org/hipster' (happened 4 times)
```

If you don't have the same problem, read the rest of this step but skip to the next.

```
$ netstat -rn
Routing Table: IPv4
  Destination           Gateway           Flags  Ref     Use     Interface
-------------------- -------------------- ----- ----- ---------- ---------
127.0.0.1            127.0.0.1            UH        2         72 lo0
10.42.77.0           10.42.77.75          U         5        369 e1000g0

Routing Table: IPv6
  Destination/Mask            Gateway                   Flags Ref   Use    If
--------------------------- --------------------------- ----- --- ------- -----
::1                         ::1                         UH      2       0 lo0
```

Of course, no default route. Fortunately, we're not going to have to have to deal with any over-engineered network configuration manager. Like `netplan` on Ubuntu, for many years, we had to deal with NWAM ("network auto-magic") on Solaris. It was neither auto, nor was it magic … good riddance to foul garbage.

You just need to create the file `/etc/defaultrouter` and put the default gateway IP address into it, if this happened to you too. That file is parsed, like the way Solaris did it for almost 30 years, at boot time, and the default route is configured from it. Also, add the default route directly to the running routing table to avoid a reboot.

```
$ sudo route add default 10.42.77.2
```

Run the `pfexec` command above again now, and again, if you're on an ARM Mac, take a short break.

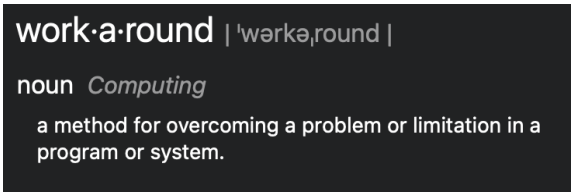
21. Now let's see a list of updated packages. You might want to run this command in a screen, or on the console of your OpenIndiana VM; OpenIndiana often lags on the network while using the package manager with limited resources. If you're on an ARM Mac, this is going to take a while. After you're sure it's working, take a medium-sized break. This one takes a while, even on Intel CPUs.

```
$ sudo pfexec pkg update -nv
```

OpenIndiana asks for all that RAM because, it turns out, their package management tool is the very definition of the word "bloated." Don't wait for the command above to run, to run this one in another terminal.

```
$ top | grep pkg
   682 root        1    0     0 1599M 1443M run      0:46  8.30% pkg
```

That's right … `pkg` is using over 1.5GB of RAM, though not much CPU. I logged in over SSH again to watch `top` output while it was running. This year, we didn't install the desktop environment, and it seems to be holding at around ~175 MB of free swap, but it's thrashing that swap. If you're running an SSD like I am, this doesn't feel good.



A kludge should only be lived with for the very short term, maybe to get us through a night, the weekend, or until help can arrive. A workaround may be okay a little longer, depending on its nature. Having to manually intervene or automate a reboot after every patch may just not work for very long at all, if we have many systems affected by the problem. Here, it'll probably work, but I'd rather not be thrashing all our SSDs, and `pkg` will just run way, way faster with more RAM.

Once it finishes, install all available updates, and create a separate "boot environment" for the patches, so we can go back if they don't work, or don't boot ... just like the loader menu we saw in Linux, with options for the last couple installed kernels, especially if you updated your kernel in Rocky. The option to create a boot environment is the `be-name` option.

```
$ sudo pfexec pkg update -v --be-name=240729
```

If you're on an ARM Mac, this is going to take a long old while. After you're sure it's working, this may well run overnight … or at least for a few hours. This one takes a while, even on Intel CPUs.

Note how I used a date-stamp for the boot environment name; update yours accordingly. This is the patching and packaging subsystem using ZFS' filesystem snapshotting capability to enable the ability to back out – and to go forward as well – with system patches.

Then `poweroff` to lock in your patches, and while you have your VM down, go ahead and reclaim some memory if you are running low and want to do so.

22. Installation of GnuPG for grading ...

```
$ sudo pfexec pkg install gnupg
```

… then let's find where it was installed …

```
$ which gpg
gpg: Command not found.
```

... it was not there for me …

```
$ which gpg2
```

This is a common configuration, to show that GnuPG is not version 1, but rather `gpg2` (version 2.x).

If you got `gpg` and `gpg2` with the package, skip ahead to the next step. If `which gpg` said "not found," let's make sure `gpg2` is also found as `gpg`. Note: I am using a soft link here, because if we used a hard link, and the built-in `gpg2` command got updated, the hard link would probably stay the old version. Using a soft link, we always point to what's there at `/usr/bin/gpg2`.

```
$ sudo ln -s gpg2 /usr/bin/gpg
$ ls -l /usr/bin/gpg
```

23. Mail forwarding … if you look in `/etc/mail`, you should recognize what you see there.

    In `/etc/mail/aliases`, I did not find a root alias, so I added one …

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

    … and ran `sudo newaliases`.

    Solaris has always has some peculiarities here in setting up mail … first and foremost, it wants to know who its "mailhost" is, like the smart host we used to use to kludge our way out of trouble on Ubuntu. Edit `/etc/inet/hosts` …

    ```
    $ sudo vi /etc/inet/hosts
    ```

    … and add a line for our mail server ...

    ```
    10.42.77.72 freebsd.cs470.internal mailhost.internal mailhost
    ```

    Then, let's edit `/etc/mail/submit.cf`, the config file for the submission mail server …

    ```
    $ sudo vi /etc/mail/submit.cf
    ```

    … the submission mail server just listens for local connections on loopback, and sends them on their way. Find the line starting with `Dj` … it's not far from the top, and let's forcibly tell `sendmail` our name. Uncomment it (remove the #) and fill in the name, replacing the example text …

    ```
    Djsolaris.cs470.internal
    ```

    Just a couple lines below, put our FreeBSD host after the `DS` option a couple lines below …

    ```
    DSfreebsd.cs470.internal
    ```

    … and then save the file and tell `sendmail` to re-read it.

    ```
    $ sudo pkill -HUP sendmail
    ```

    Then I used a similar mail command to test e-mail …

    ```
    $ echo "test" | mail -t peter@cs470.internal
    ```

    … and looked at the end of `/var/log/syslog`. Note that logs on Solaris are broken up, into `/var/log` and `/var/adm`.

    For further reading, I recommend: https://docs.openindiana.org/handbook/systems-administration/

part three: log server configuration

In this second half of the lab, we learn the basics of using Ansible for remote management of systems and use the tool to automate the configuration of centralized logging with `syslog` on each operating system.

Centralized logging allows us to accomplish two important goals. First, by offloading the logs from each system, we're able to put more trust in logs on a log server than we might trust logs on a compromised system … not very much. Also, we're able to view the logs of all our VMs in one place, rather than having to go on a hunt for logs in each VM. In this way, we're able to meaningfully compare logs across systems in an incident that affects multiple systems. It's for this reason that we've set up NTP (network time protocol) on each system, to make sure that timestamps are accurate.

Since Rocky Linux is our storage VM, it makes sense to store all our logs within that VM. We'll do this by having a `syslogd` listen on Rocky Linux and accept logs from other systems, while all the other VMs will act as clients … and we're going to use Ansible to help set it all up. Ansible uses YAML, and as you'll see over the course of this lab, YAML is **super picky** about indenting and formatting … for this reason, I've provided a screenshot of the final version of the primary configuration file in a monospaced text editor, in the first appendix to this lab. Use it as a reference before asking for help … or invite certain scrutiny for not heeding the support policy.

YAML is so picky, in fact, that there's a website dedicated to making fun of it: https://noyaml.com/

Speaking of picky configuration files, `syslogd` used to take the prize. Older implementations of `syslogd` rigidly enforced whether you used spaces or tabs as a delimiter in between fields of its configuration file, and would silently ignore violating lines in the file. The finicky nature of `syslogd` configuration, the number of different ways to configure derivative logging services, and the need to centralize configuration makes log centralization an ideal exercise for an automation lab. In field use, other additional common configuration, such as patching, `cron` jobs, NTP configuration, and mail forwarding could be implemented once in an Ansible "playbook," and that playbook used to automate common configuration on all supported operating systems.

Before we go on, it's important to note that there's the original `syslog`, and there's a more recent replacement called `rsyslog` which is more feature-rich. We'll be using both in this lab to utilize already-included packages rather than downloading additional ones on top of them. The original `syslog`, for example, doesn't support TCP connections – only UDP – so we'll be configuring our `rsyslog` server to listen for UDP, despite it also being compatible with TCP. The BSDs and other genetic Unix systems (OpenIndiana, AIX) have `syslog` installed by default, while Rocky Linux, Ubuntu, and other Linux-based distributions tend to come with `rsyslog` instead, if they come out of the box with a logging daemon at all.

First, let's configure `rsyslog` on Rocky Linux.

24. Double check that `syslog` is running.

    ```
    $ systemctl status rsyslog
    ```

    should return that `rsyslog.service` is active.

25. Use `vi` to edit `/etc/rsyslog.conf`. Uncomment the lines that specify the module and input for `imudp`. This will allow `syslog` to accept UDP connections on port 514. After those lines, add the

following two lines to configure the locations for each system's logs.

```
$template FILENAME,"/var/log/systems/%HOSTNAME%.log"
*.info  ?FILENAME
```

Note: in between `info` and `?FILENAME`, that's a tab, because really old versions of `syslogd` required use of tabs in their configuration files. Of course, only the Solaris derivative forces this on us now.

26. If you'll remember from lab 3, Rocky Linux comes with a firewall. We need to allow `rsyslog` as a service through this firewall …

```
$ sudo firewall-cmd --permanent --zone=public --add-service=rsyslog
```

… but this interestingly enough returns that `rsyslog` is not among existing services. This command …

```
$ sudo firewall-cmd --get-services | grep log
```

… shows us that it's referred to as just `syslog` within the firewall. Fix the above command and reload the firewall.

```
$ sudo firewall-cmd --permanent --zone=public --add-service=syslog
$ sudo firewall-cmd --reload
```

Verify that `syslog` is now in the firewall's allowed services.

```
$ sudo firewall-cmd --info-zone=public

public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens160
  sources:
  services: cockpit dhcpv6-client mountd nfs rpc-bind ssh syslog
```

27. Now that we added `syslog` as a valid service to our firewall, restart `rsyslog` itself for the configuration changes we made a couple steps ago to take effect.

```
$ sudo systemctl restart rsyslog
$ ^restart^status
```

We don't need to enable it ourselves, because we can see from the `systemctl` status command that it's enabled as a vendor default. Look in `/var/log/systems` … it has been created, and you should already be seeing logs in there for your Rocky Linux VM.

28. From the `rsyslog` configuration file, the port that `syslog` is listening on is 514. Confirm that with `netstat`:

```
$ netstat -an | grep 514
```

part four: first client configuration

29. We need to configure all of our systems to send their logs over to our main server. Let's start with OpenBSD. Check the status of `syslogd` …

    ```
    $ rcctl check syslogd
    ```

    … this should return `ok`. If it isn't running, just restart the service.

30. Use `vi` to edit `/etc/syslog.conf` and add the following line to the end of the file:

    ```
    *.info      @rocky
    ```

    This line tells `syslog` to forward all logs to our Rocky Linux VM. Once again, note that for `syslog.conf`, that's a tab. We want to use tabs instead of spaces.

    `syslogd` starts up before `named` on our OpenBSD VMs, so let's make sure it never has a problem finding its log server, even during the boot process. As we've done before, add a line for `rocky` in `/etc/hosts` on your OpenBSD VM.

    Next, we need to restart `syslogd` after making this change …

    ```
    $ sudo rcctl restart syslogd
    ```

    … it should echo `ok` twice, once for stopping it, once for starting it back up again, as usual with `rcctl`.

31. Now let's test logging …

    ```
    $ logger "openbsd test 1"
    ```

    You can see this log locally on OpenBSD by running …

    ```
    $ tail /var/log/messages
    ```

    To see it on your Rocky Linux VM …

    ```
    $ sudo tail /var/log/systems/openbsd.cs470.internal.log
    ```

    To confirm the logs are now being sent to Rocky Linux.

    Now that we've configured this for OpenBSD, we need to configure the remaining machines in our network … but we're not going to do this manually. We'll be using Ansible for the rest of the systems, but keep note of the following configurations for each system.

    On FreeBSD:

to manage the `syslog` daemon: `/etc/rc.d/syslogd`
`syslog` configuration file: `/etc/syslog.conf`

On Ubuntu:
to manage the `rsyslog` daemon: `systemctl rsyslog`
`syslog` configuration file: `/etc/rsyslog.conf`

On OpenIndiana:
to manage the `syslog` daemon: `svcs system-log`
`syslog` configuration file: `/etc/syslog.conf`

After configuring all of our local VMs, we've now successfully set up Rocky Linux as our centralized logging server. This is on top of it already being our network file share.

part five: Ansible host

We're going to setup and use Ansible. Ansible is a tool that allows you to manage multiple servers at once, rather than manually configuring each. We do this by configuring what and how we want everything to happen once, while Ansible takes care of the rest.

We'll use OpenBSD as the host of our Ansible configurations. When you use `ssh` to connect to your OpenBSD VM in this lab, always use the `-A` switch to forward your SSH agent to your OpenBSD VM. As you'll see, Ansible uses SSH for transport, and by forwarding your agent, you won't need to log in to any of the systems we're automating actions on. It wouldn't be very automated if you had to provide a password for each system, now would it?

32. On your OpenBSD VM, make a directory at `/home/ansible` for your Ansible configuration files, and make it sudoers group writable, so that your failsafe account and LDAP user can both use that directory. We'll come back to this later, as we'll be using OpenBSD to host and configure Ansible.

33. Ansible documentation shows `pipx` as the preferred way to install and manage Ansible. `pipx` is an extension of `pip`, Python's package manager, and Ansible is built on top of Python. Thus, Ansible requires Python on all systems it is used on (including the systems you target) to run … thankfully, all of our VMs in this course come with Python installed by default.

    Unfortunately, OpenBSD does not have an official port of `pipx`, which means we have no choice but to compile it from source.

    Navigate to `sysutils/py-pipx` under the ports tree, and get compiling …

    ```
    $ sudo make install
    ```

    This should only take a minute or two. Once it's done compiling, verify that it's installed.

```
$ pipx --version
```

34. Ansible comes in two main flavors: `ansible`, a community distribution with a variety of tools that we used in prior versions of the class, and `ansible-core`, a minimum distribution which only contains the Ansible engine and language. We won't be needing the extra tools from the community distribution, so we can save some resources here.

    Let's try and install `ansible-core` with `pipx`.

    ```
    $ pipx install ansible-core
    ```

    I got an error that certain dependencies, namely `maturin` and `installable`, failed to build, complaining that `rustc` was not found. `rustc` is the compiler for the Rust programming language, and seeing as `maturin` is written in Rust, the error checks out. You can compile Rust from the source tree … or get it installed far more quickly with `pkg_add`.

    ```
    $ sudo pkg_add rust
    ```

    Try installing `ansible-core` again using `pipx`. This will take a while longer than before.

    ```
    ⣿ installing ansible-core
      installed package ansible-core 2.18.4, installed using Python 3.11.10
      These apps are now globally available
        - ansible
        - ansible-config
        - ansible-console
        - ansible-doc
        - ansible-galaxy
        - ansible-inventory
        - ansible-playbook
        - ansible-pull
        - ansible-test
        - ansible-vault
    ⚠  Note: '/home/peter/.local/bin' is not on your PATH environment variable. These
    apps will not be globally accessible until your PATH is updated. Run `pipx
    ensurepath` to automatically add it, or manually modify your PATH in your shell's
    config file (e.g. ~/.bashrc).
    done! ✨ 🌟 ✨
    ```

    I was taken aback by the terminal emoji, but after installation is complete, try running …

    ```
    $ ansible --version
    ```

    … to get some details about Ansible's versioning and configuration. You should do this whenever you install a piece of software you expect to use, to see how it was built.

    I got an error telling me that ansible wasn't found. Now is a good time to take a step back here to understand what `pipx` actually does.

With `pip`, Python's built-in package manager, packages are installed system-wide, which comes with some problems. In many cases, we need to install different versions of the same package, which isn't really possible with a system package manager, since there'd be no way to tell the binaries from each version apart.

Python's solution to this is virtual environments … virtual like virtual machines, but virtual in the sense that one system can have multiple Python environments that are isolated from each other. Each virtual environment has its own set of installed packages, which allows us to install two different versions of the same package, so long as they are in different virtual environments. To use a virtual environment's packages, we'd typically add its configuration to our `PATH` environment variable to make all of its binaries available to us. This can get cumbersome, especially when all we care about right now is getting Ansible up and running.

Enter `pipx`. `pipx` is designed to seamlessly install Python applications while avoiding the problems mentioned earlier. It achieves this by creating a separate virtual environment for each application, keeping them and their dependencies isolated from one another.

With this in mind, look back at the warning `pipx` gave us after it finished installing `ansible-core`. It tells us why `ansible` isn't available, and tells us how to fix the problem. Read what it says, and do it.

Test again with `ansible --version`. You should now see some real output. Also note that the `ansible` binary is located in a `pipx` virtual environment, as explained earlier:

```
$ which ansible
/home/peter/.local/bin/ansible
```

35. Before we do our first configurations and run Ansible, there is one important thing to note about how Ansible works: it is *idempotent*. Idempotence means that you can run something once, or a dozen times, and still get the same result back once. An example of idempotence would be installing a package through a package manager like `apt` once, and if you run it with the `install` verb a second time, there won't be an additional instance of that package installed – the package manager, depending on its design, would either exit gracefully or update the package if it needs one, depending on the design.

    We can run Ansible in two ways. Much like Docker, we can specify flags on the command line to specify certain actions. Much more useful than that, however, are the configuration files we can write that allow us to specify a whole lot more than what can be reasonably fit into single command lines. There are three in Ansible that are important: Playbooks, the Ansible `.cfg` file, and Inventory files.

    The `ansible.cfg` and inventory files are purely configuration files. They help us define some defaults and our list (inventory) of machines we'd like to target. The first thing we want to do is to write out these two files.

    Navigate to `/home/ansible`. Generate an ansible config file with `ansible-config init --disabled`, but note that it just outputs to `stdout` by default, so we have to redirect it to `ansible.cfg`. Then, use

`touch` to create the files `inventory` and `syslog_clients.yml`.

Add the following to your `ansible.cfg` file above `[privilege_escalation]` …

```
INVENTORY = /home/ansible/inventory
remote_tmp = /tmp
```

… then add to `inventory` like so:

```
[cs470]
freebsd
ubuntu
solaris
```

The `[cs470]` heading in the `inventory` file is the token you will reference when running Ansible on that set of computers.

Note wherever you create your `ansible` directory and place your configuration files. Ansible prioritizes the current directory, and the `ansible.cfg` file in it, when you run `ansible`.

36. Before we get into playbooks, let's start off with running a simple command across all our systems. This is called an ad-hoc command in Ansible.

```
$ ansible cs470 -a "hostname"
```

This will run `hostname` on all systems listed under `cs470`, in the `inventory` file.

If you haven't yet logged into each of those VMs from your OpenBSD VM with SSH, it might be a good idea to do so manually first. As should be obvious, Ansible uses SSH for transport, and will get hung up on accepting SSH host keys if you haven't logged in before. Also worth noting, if you did your SSH agent and agent forwarding setup correctly from lab zero, you won't have to provide a password for each system.

You can try this with other commands, too, but beware: not all systems have the same syntax for each command. This way of running commands with Ansible is not considered best practice, and Ansible has a good replacement for this, called modules. We'll get into the module system with playbooks.

One thing you might have noticed is that the order of which machine goes first is not always the same. Ansible forks itself so that it can configure multiple systems in parallel, instead of waiting for one system to finish before configuring the next. By default, Ansible forks itself 5 times, so that it can configure up to 5 systems at once. This makes the order of which server gets hit, or returns first, completely non-deterministic.

37. Now, let's write the actual playbook. It's going to be our `syslog_clients.yml` file, and as the name implies, will configure `syslog` to forward logs on all the remote systems.

NOTE: When writing YAML files, you simply CAN'T use tabs to indent lines. You always have to use spaces for all indentation.

If you installed a more modern editor like `vim` or `neovim`, it'll know that you're editing a YAML file and try to help you out by automatically inserting two spaces whenever you press tab. However, if you DON'T want two spaces, you can input a literal tab character by entering insert mode and hitting `control-v` (letting go of those keys after) before hitting tab.

Let's start our file off with the following structure …

```
---
- name: configure syslog clients on the CS470 network
  hosts: all
  become: true
```

... and let's break down what it all means. The `name` field is simply a title for the specific task that you'll be running. Since this is at the top level, consider it as the name for the entire Playbook which will be echoed at the very beginning when you first run it. The `hosts` field specifies which hosts we want to run this on. Finally, the `become` field tells Ansible to run with privileged permissions … that is, **become** an admin during the duration of the tasks. Since this is at the top level, all tasks will run with `sudo`, but you can also specify `sudo` for each task.

A task in Ansible is a command you want to run, or configuration you want to change. A handler is like a task, but is something that you typically only want to run when a task makes a change on your remote system.

Write out the file you were in; we need to go back to the shell.

38. Make a directory called `host_vars` in `/home/ansible`. Then, make a YAML file for every host within that directory:

```
$ touch freebsd.yml ubuntu.yml solaris.yml
```

The daemon name and the location of the configuration file differs in between operating systems. As a result, we need to use these files to set variables that will contain the correct values for each system.

Within `freebsd.yml`, place the following:

```
syslog_service_name:
  - name: syslogd

syslog_conf_paths:
  - path: /etc/syslog.conf
```

Do the same with the files `ubuntu.yml` and `solaris.yml`, replacing the `name` and `path` fields as needed with the configurations found at the very end of part four of this lab.

39. Go back to editing the `syslog_clients.yml` file. We're going to model this file after the steps we took earlier to configure logging on OpenBSD. First, we write a task to check the status of the logging daemon:

```
tasks:
 - name: Check the status of the syslog daemon
   service:
     name: "{{ item.name }}"
     enabled: yes
     state: started
   with_items: "{{ syslog_service_name }}"
```

The `service` field refers to an Ansible module called "service." This module will use the built-in service management system (like `systemd` on Ubuntu and Rocky Linux) on each system, rather than having to write out commands specific to each system, with a bunch of conditionals.

The `name` subfield refers to the name of the service you want to interact with. You'll notice a unique syntax, we'll get back to this later. The `enabled` subfield tells Ansible if you'd like the service to be enabled at boot or not. The `state` subfield tells Ansible what state you'd like the service to be in currently.

The `with_items` field creates a loop over a list of items. This list of items is compiled by Ansible automatically since we used the default directory of `host_vars` and placed YAML files based on the hostname within that directory. Since we're referencing the `syslog_service_name` variable, `with_items` creates a list of our 3 values and exposes the `item` variable within that task to allow us to refer to that variable. This special templating syntax in Ansible is called Jinja.

Save the file and run the playbook. Notice that like Docker and Docker Compose, running Ansible on the command line and running an Ansible playbook are different commands.

```
$ ansible-playbook syslog_clients.yml -K
```

The `-K` flag here will prompt you for the "BECOME password", which will be your `sudo` password on the systems; since you should be using your LDAP user account here, your `sudo` password should be the same on all systems except maybe Solaris.

Note the results. Ansible nicely formats what happened at each stage and gives a synopsis of any changes, variables used, and the status of each task.

40. Before we move on to the next step, let's introduce another piece of versatile automation software that will make your lives easier. `expect` is a tool that can be used to automate other pieces of software, it's able to "talk" with other programs that require input. It can be used in virtually all modern operating systems and can make many processes easier through automation, as you will soon see. First things first, let's get it installed.

Using `pkg_info` again, find the right version of `expect` to install. The top result is what we need.

While Ansible has its own `expect` module, we will be using a separate module for what we're using `expect` for later. I'd advise you to look at it in your free time if you want to learn more about Ansible.

41. Next up, create a file named `become.sh` and make sure it's executable as well. The `.sh` signifies that the file is a `bash` script, which will allow us to run written commands in a file. No need to manually run multiple commands repeatedly, when you can just script it.

42. Within `become.sh` add the following lines:

```
#!/usr/local/bin/expect
set timeout -1
spawn ansible-playbook syslog_clients.yml -K
expect "BECOME password:"
send -- "<your root_password here>\r"
expect eof
```

Let's go over what this does.

`set timeout -1` tells the system to not set a timeout for this script, we don't need one. `spawn ansible-playbook syslog_clients.yml -K` is automating the command for us. `spawn` tells the system to create another process and runs the command we give it, in this case the command to run the playbook. This way instead of having to copy and paste, or scroll through the `bash` history, we can just run the script and it'll do it for us. `expect` tells the system to be on the lookout for output that requires input, otherwise we'd be sending data at every possible moment while a process was running. In this case, we know our playbook will always request the `BECOME password` every time.

`send` does exactly that; it sends the information in the quotation marks to what the script is expecting. I hope your root passwords are all the same, otherwise you have research to do. You can do this for any process or software you know what the requesting line is.

From here on out, you can use `./become.sh` to test out your playbook without needing to input your root password every time. Keep in mind, however, that this is **really** bad practice. In real life, we do not want to store your root password, or any password, in plaintext, anywhere, **ever**. Really insecure, but we needed something to demo `expect` with.

43. The next thing we need to do is to automate the addition of a log-forwarding configuration line to the end of each system's logging daemon configuration file. Create a new line after the previous task and model it like so:

```
- name: add configuration line to the end of the syslog config file
  lineinfile:
    path: "{{ item.path }}"
    line: "*.info   @rocky"
  with_items: "{{ syslog_conf_paths }}"
  notify: restart syslog daemon
```

The `notify` field here refers to the handlers we discussed earlier. This tells Ansible to invoke the handler if that task has a change.

44. Remember idempotence? Ansible will remember if it inserted a line to the end of those files in the previous task. If you were to run that task again, Ansible will detect that no changes are needed and make no changes. However, when there is a need to make changes, we'll need to restart the logging daemon on each system and do some test logging. Let's write the handlers for those…

    Above your `task` section, create a `handlers` section, and write out the following:

```
handlers:
  - name: restart syslog daemon
    service:
      name: "{{ item.name }}"
      state: restarted
    with_items: "{{ syslog_service_name }}"
    notify: log to console

  - name: log to console
    shell: 'logger "Testing from $(hostname)"'
    args:
      executable: /usr/local/bin/bash
```

    Notice that our first handler notifies the second handler to run.

    On your Solaris VM, make a symbolic link from `/usr/local/bin/bash` to `/usr/bin/bash` … you will have to make sure the directory tree is there first. If Peter had gotten LDAP working on Solaris, this symbolic link would have been there.

45. Now that we've finished writing our playbook, run the playbook again with the command from earlier. Make sure to type out your `sudo` password on the prompt.

    One thing we can do is encrypt values with `ansible-vault`. This will allow us to store the `sudo` password securely on the system, in a vault file, which is managed by Ansible.

    While in `/home/ansible`, run …

```
# openssl rand -base64 32 > .vaultpass
```

    … to generate a pseudorandom string of 32 bytes with Base64 encoding. We'll use this as our vault password. Get the contents of the file in a terminal using your command of choice, and copy the password.

    Now, create an Ansible Vault encrypted file using …

```
$ sudo ansible-vault create vault.yml
```

    … notice it prompts you for a new vault password. Use the password you generated earlier with `openssl` and paste it into the prompt.

    You'll be placed in a `vi` window. Write out the contents of the file as such:

```
---
sudo_password: <your_sudo_password>
```

Note that to interact with this file, `ansible-vault` provides encrypt, decrypt, and edit options. Encrypt and decrypt work on the vault file and will leave the file within the state you left it in, meaning you could accidentally leave your vault file decrypted if you don't encrypt it again. Preferably, you should use `ansible-vault edit` to view and edit the file in a temporary `vi` window, while the file itself doesn't get decrypted on the drive.

If you were to view this file in `vi` or output it to the console, you'll notice it's not human readable. Furthermore, the header tells us that `ansible-vault` uses AES-256 for encryption, the same symmetric encryption standard Uncle Sam uses for secret stuff.

46. Now that we've created a vault password to store our `sudo` password, modify your `syslog_clients.yml` file with the following lines between your "`become: true`" and "`handlers:`" lines:

```
vars_files:
  - /home/ansible/vault.yml

vars:
  ansible_become_pass: "{{ sudo_password }}"
```

The `vault.yml` file is owned by `root:wheel` by default. Take a look at the permissions of that file and change them so that you can pass that file into `ansible-playbook` without needing `sudo` to access that file … you probably just need to make it group-readable. Using `sudo` would also introduce other issues, like `ansible` attempting to `ssh` into your VMs as root.

And finally, to run the playbook without a prompt:

```
$ ansible-playbook syslog_clients.yml --vault-password-file
/home/ansible/.vaultpass
```

Voilà! You've now encrypted your `sudo` password on the disk and created a password to protect that `sudo` password. This means your playbooks can now be automated and run without a human being in the loop, and without exposing the `sudo` password. From here on out `become.sh` is no longer necessary, as trying to run the playbook without the `vaultpass` is impossible, unless you remove it from the playbook, and the command does exactly what `become.sh` does. Your password is also encrypted in the vault too.

If you had a different `sudo` password for each system, it's possible to vault-protect multiple `sudo` passwords with `host_vars` like we did with the daemon name and conf file path above, but we won't be doing that in this lab. Hopefully you had the same password for your LDAP user's local namesake on Solaris … if you didn't, just change it.

The screenshot of `syslog_clients.yml` on the following page is provided as a reference, because of

YAML's pickiness about formatting.

47. On your Rocky VM, do this to make sure your centralized logging destination is accessible:

```
$ sudo chgrp -R wheel /var/log/systems
$ sudo chmod 750 /var/log/systems
$ sudo chmod g+r /var/log/systems/*
```

appendix one: final Ansible configuration

```yaml
---
- name: Configure Syslog clients on the CS470 network
  hosts: all
  become: true

  vars_files:
    - /home/ansible/vault.yml

  vars:
    ansible_become_pass: "{{ sudo_password }}"

  handlers:
    - name: restart syslog daemon
      service:
        name: "{{ item.name }}"
        state: restarted
      with_items: "{{ syslog_service_name }}"
      notify: log to console

    - name: log to console
      shell: 'logger "Testing from $(hostname)"'
      args:
        executable: /usr/local/bin/bash

  tasks:
    - name: Check the status of the syslog daemon
      service:
        name: "{{ item.name }}"
        enabled: yes
        state: started
      with_items: "{{ syslog_service_name }}"

    - name: Add configuration line to the end of the Syslog config file
      lineinfile:
        path: "{{ item.path }}"
        line: "*.info     @rocky"
      with_items: "{{ syslog_conf_paths }}"
      notify: restart syslog daemon
```

appendix two: network auto-magic

This portion of lab 5 was brought back in 2025 from the 2021 version of lab 5, just in case you had an issue configuring the network on OpenIndiana with the installer. If so, you could use the "none" option for the network during installation, and then configure the network immediately after installation.

Solaris used to put its IP address information in an `/etc/hostname.if` file, just like OpenBSD currently does, and the default router in `/etc/defaultrouter`. However, Solaris and all its descendants have now moved on from this.

I checked out Oracle Solaris, and it appears to now put network configuration information into `/etc/zones` and `/etc/svc/profile/sysconfig` – I found it by doing a recursive `grep` with the `-r` switch for the network interface name in `/etc` … but both of these turned up empty in OpenIndiana if I didn't configure the network or chose the "Automatic" option during installation. It turned out, of course, that the best first step was a web search for "configure network OpenIndiana," and this link was the secret sauce … not only had the former OpenSolaris people named the network configuration subsystem after an annoying way of things fixing themselves without you knowing what was ever broken ("auto-magic"), but this new over-engineered tool that bucks 30+ years of simply editing text files is somehow "auto-magic."

The tool we're looking for is called `nwamcfg`.

First, though, let's make sure we know what network interface we're configuring. Run `ifconfig -a` to see what interfaces your VM has, and remember that `lo` interfaces are typically loopback adapters. My network interface is definitely called `e1000g0`.

Now let's run `nwamcfg`.

```
$ sudo nwamcfg
```

This should give you a `nwamcfg>` prompt. When I get a prompt for a specific application, I always try `help` or simply `?` to see what I *can* do, even if I know what I want to do, and that made me realize that what I *really* wanted to do was to poke around, with the list command, I saw a single NCP ("Network Configuration Profile," because we need more acronyms in tech) … "Automatic." "Automatic," unlike `nwam` and NCP, is a great name for a configuration that uses DHCP to automatically set up the network … so let's set up another profile, "static" … incorporating our VM's only network interface, whatever `ifconfig` told us it was before.

```
nwamcfg> create ncp static
nwamcfg:ncp:static> create ncu phys e1000g0
Created ncu 'e1000g0'.  Walking properties ...
activation-mode (manual) [manual|prioritized]>
enabled (true) [true|false]>
link-mac-addr>
link-autopush>
link-mtu>
```

Note: in the transcript above, where there's no input shown after the `>` prompt, I only supplied a carriage return to accept the default for each after creating the interface. All the defaults (in parenthesis here) seem to be fine, and for the MAC address, autopush, and MTU (maximum transmission unit), we want to provide no special settings.

Now we apparently save our changes with the `commit` command, and get out of raw interface configuration with the `end` command.

```
nwamcfg:ncp:static:ncu:e1000g0> commit
Committed changes
nwamcfg:ncp:static:ncu:e1000g0> end
nwamcfg:ncp:static>
```

At last, let's set up the interface's IP address. Use `create ncu ip` with the name of your network interface, and after you're done, `commit`, and `exit`.

```
nwamcfg:ncp:static> create ncu ip e1000g0
Created ncu 'e1000g0'.  Walking properties ...
enabled (true) [true|false]>
ip-version (ipv4,ipv6) [ipv4|ipv6]> ipv4
ipv4-addrsrc (dhcp) [dhcp|static]> static
ipv4-addr> 10.42.77.75
ipv4-default-route> 10.42.77.2
nwamcfg:ncp:static:ncu:e1000g0> commit
Committed changes
```

At this point previously, an `exit` just brought me up a logical level within `nwamcfg`, to `ncu`, and then to `static`. Now the first `exit` ejects you from the program … no problem, though, just run it again and now, let's create a location …

```
nwamcfg> create loc static
Created loc 'static'.  Walking properties ...
activation-mode (manual) [manual|conditional-any|conditional-all]>
enabled (false) [true|false]> true
nameservices (dns) [dns|files|nis|ldap]>
nameservices-config-file ("/etc/nsswitch.dns")>
dns-nameservice-configsrc (dhcp) [manual|dhcp]> manual
dns-nameservice-domain> cs470.internal
dns-nameservice-servers> 10.42.77.71
dns-nameservice-search> cs470.internal
nfsv4-domain>
ipfilter-config-file>
ipfilter-v6-config-file>
ipnat-config-file>
ippool-config-file>
ike-config-file>
ipsecpolicy-config-file>
```

… commit …

```
nwamcfg:loc:static> commit
Committed changes
```

… and `exit` again. Now, let's make our profile and location the defaults back out at a shell.

```
$ sudo nwamadm enable -p ncp static
$ sudo nwamadm enable -p loc static
```

You should be able to validate it's taken hold by running `ifconfig -a` … also check the contents of `/etc/resolv.conf`. Test it's working with the usual set of remote IPs and public website names, to make sure your OpenBSD name server is serving up internet DNS resolution, and that your new Solaris-ish VM is able to access them.

appendix three: Jack

OpenIndiana always brings Jack to mind, thanks to the username used on the OpenIndiana live ISO prior to installation. Not my son Jack, but my hacker buddy Jack, very much my brother in Unix.

One day I was working on a pen-test with Jack, when Jack thought he found a flaw on that customer's web server. It turned out to be a security issue in the underlying programming language itself (`perl`), and it affected pretty much every Unix/Linux system in the whole world, and a whole lot more.

Another day, Jack decided the ubiquitous portscanner `nmap` was a slow and bloaty piece of crap, and he decided to write a tool to portscan the entire routable internet in a matter of hours. When he finished the first distributed asynchronous TCP and UDP port scanner, this dyed-in-the-wool punk rock hacker decided to name it "Unicorn Scan," because unicorns were ***so*** not Jack, and so not his scanner either.

This one's for you, Jack. Jack's badass tattoo keeps this little detour completely on-point.

RIP Jack Louis 1977-2009

RIP Sun Microsystems 1982-2010

</lab5>