CS 470: Unix/Linux Sysadmin
Spring 2025 Lab 6
backup and restore, P2V conversion, and private cloud with Proxmox

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

- getting your lab 2 VM up and online
- getting NFS running in lab 3
- grading this lab is done from your lab 2 FreeBSD VM

Nothing in this lab is on the critical path for upcoming labs.

Those of us with Intel CPUs have been using VMware's desktop hypervisor extensively thus far in the class; at this point in the labs, you've arguably set up a little data center in a private cloud. What if you wanted to use data center grade products, rather than desktop products? What if you wanted to take a physical machine and turn it into a virtual machine? We're going to answer all these questions in lab six, while we also pick up and utilize valuable skills performing backups and manually partitioning disks so we understand how to put the pieces of a machine back together in the event of a serious problem, or the need to convert a machine from a physical machine into a virtual instance.

First, we are going to install Proxmox to pilot a private cloud hypervisor. Then we are going to back up our FreeBSD VM to the file server on our Rocky VM, and restore a replica using our NFS and web servers.

We used to use VMware's ESXi hypervisor here, but VMware no longer offers that hypervisor for free for hobbyists, nor does it offer the academic licensing program that used to give us VMware's full suite of data center products for free for academic use.

Subject: ACTION REQUIRED: End of Life for VMware IT Academy & Software Licensing Programs
From: VMware IT Academy - No Reply <email@itacademy.brightspace.com>
Date: 8/8/24, 2:38 PM
To: pbartoli@sdsu.edu

Hello Peter,

We regret to inform you that Broadcom has unilaterally made the decision to initiate the end-of-life process for the **VMware IT Academy** and **Academic Software Licensing** programs effective **August 15, 2024**. As a result, customers will no longer be able to purchase new subscriptions in the VMware IT Academy store after **5:00pm E.T. on August 15, 2024**. We understand that the IT Academy has been a reliable resource for many years, and we are disappointed to be delivering this message.

Proxmox is the next best thing, so that's what we're going to use. It's free and open source, but a commercial version of the product is available if you want advanced features or want support for it in a business environment.

When extending support in this class to ARM Macs, this lab was one of my greatest concerns. I was really trying to avoid a situation where users of one platform had entirely different directions than the other, but it couldn't be avoided in this lab. For a portion of this lab, there will be a set of directions for each platform.

Those of you on Intel-based systems can actually do nested virtualization and will replicate your FreeBSD VMs as a nested virtual machine inside your Proxmox VM. Those of you using ARM Macs will only be able to test drive Proxmox in a VM for a bit, then will have to create another VM to make a replica of your FreeBSD instance on the side.

Because of the need to do nested virtualization here in this lab, if you're going to have system configuration issues, or conflicts with other software on your computer, this is when it's going to happen, if it hasn't already.

CPU hypervisor resource conflicts on Windows:
VT-x and other virtualization extensions in the CPU must be enabled for Intel CPU users, similar ones also enabled for AMD CPU users, and Hyper-V must be disabled for Windows users. If you failed to set the default WSL version to 1 in lab zero, you're likely to catch a problem here. You might also need to turn off the Virtual Machine Platform and Windows Hypervisor Platform features in the Windows Features dialog, and if you're on an AMD CPU, you'll have to turn off memory integrity features.

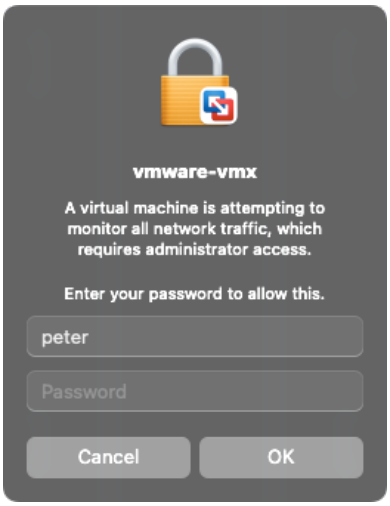
The links below and the proposed solutions in them helped a lot of people get their setups working.

disabling Hyper-V on Windows:
https://stackoverflow.com/questions/30496116/how-to-disable-hyper-v-in-command-line

disabling memory integrity on an AMD CPU:
Open Windows Security from the Start Menu, select "Device Security" in the menu on the left, then click on "core isolation details." Turn off "Memory Integrity."

enabling nested virtualization in a VM:
https://www.itechtics.com/nested-virtualization
https://bhanuwriter.com/virtualized-amd-v-rvi-is-disabled/

virtual network interface permissions and promiscuous mode on Linux:
In order to pretend to be other computers with other ethernet MAC addresses, to facilitate the software-defined networking capabilities in its hypervisors, VMware needs full control to read and re-write traffic on all network devices, which of course we do with file permissions. People running lab six on VMware Fusion on Intel Macs will get a login prompt to login as an admin when firing up an Proxmox VM.



Native Linux users must find out how to allow promiscuous mode through to their VMs; see this VMware knowledge base article for more information. The best solution for this annoyance is to modify the VMware service startup scripts on your host operating system. I recommend adding the following two lines to the end of the function `vmwareStartVmnet()` in `/etc/init.d/vmware` so that administrators of your Linux host (notably you) can start up VMs with full read/write access to the hypervisor's virtual network interfaces.

```
chgrp sudo /dev/vmnet*
chmod g+rw /dev/vmnet*
```

In all things, as always, if you get an error message, web search it first and ask questions second. Usually there's a quick answer.

part zero: get it

Go to proxmox.com and hit "downloads" in the main toolbar. Under latest releases, to the right of "Proxmox VE 8.4 ISO Installer," click Download. As the page says, `proxmox-ve_8.4-1.iso` is approximately 1.6 GB.

part one: install it

As always, create a custom VM.

- guest OS:
  - Intel/VMware: Debian 12.x 64-bit
  - ARM/UTM: emulate, other
- CPU:
  - Intel/VMware: two virtual cores or processors
    one processor, two cores if you're on VMware Workstation
  - ARM/UTM:
    - architecture: x86_64

    - system: Standard PC (Q35)
    - CPU: Nehalem-v2
    - two virtual cores
- boot firmware: UEFI
- RAM: 4096 MB / 4 GB
- hard disk: 20 GB

Whether you're using VMware or UTM, you'll probably need to customize settings right away.

Those of you with Intel CPUs will need to enable nested virtualization.

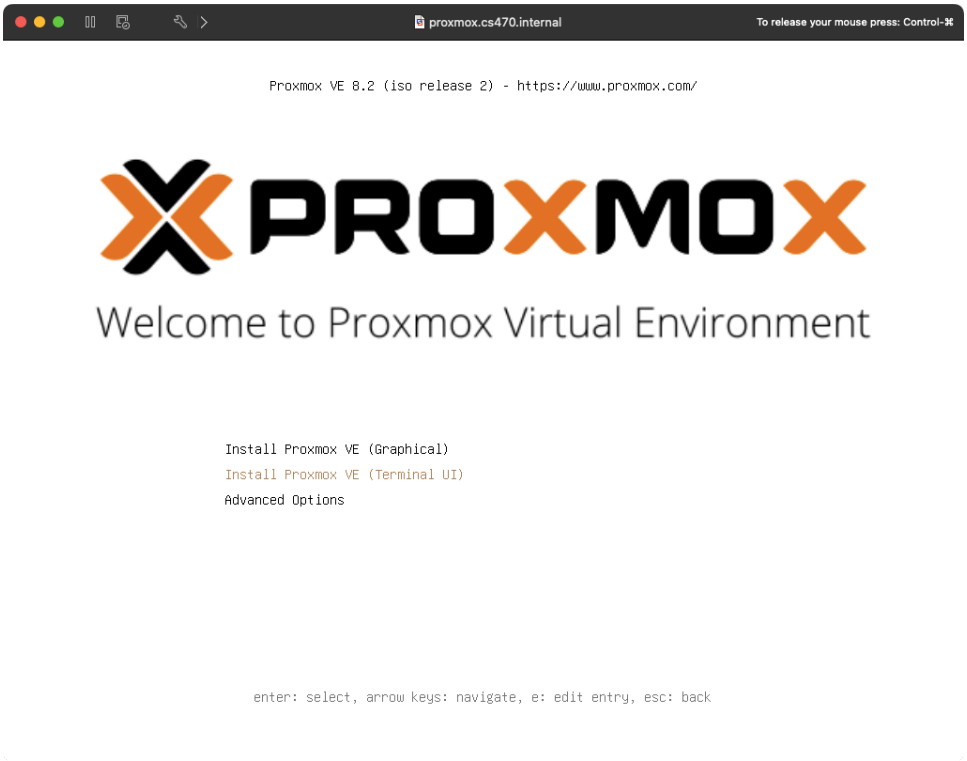
- In VMware Fusion on Intel Macs, go to your VM's settings, then the processors/memory pane, click on the triangle next to "advanced options," then click on the button to enable hypervisor applications in the virtual machine if it's not already turned on.
- In VMware Workstation on Windows or Linux, go to your VM's settings, select the hardware tab, then choose "processors" in the device list, and click on the box next to "Virtualize Intel VT-x/EPT or AMD-V/RVI" if it's not already turned on.

For those using ARM Macs, nested virtualization isn't going to work with emulation, but the following settings will help you at least run Proxmox in emulation:

- under QEMU and "tweaks"
    - enable UEFI boot, the RNG, and balloon devices if not already
    - make sure "use local time for base clock" is turned off
- under display, change the emulated display card to "virtio-vga" if not already that
- under network, change your emulated NIC to an e1000 if not already and use "Shared Network"

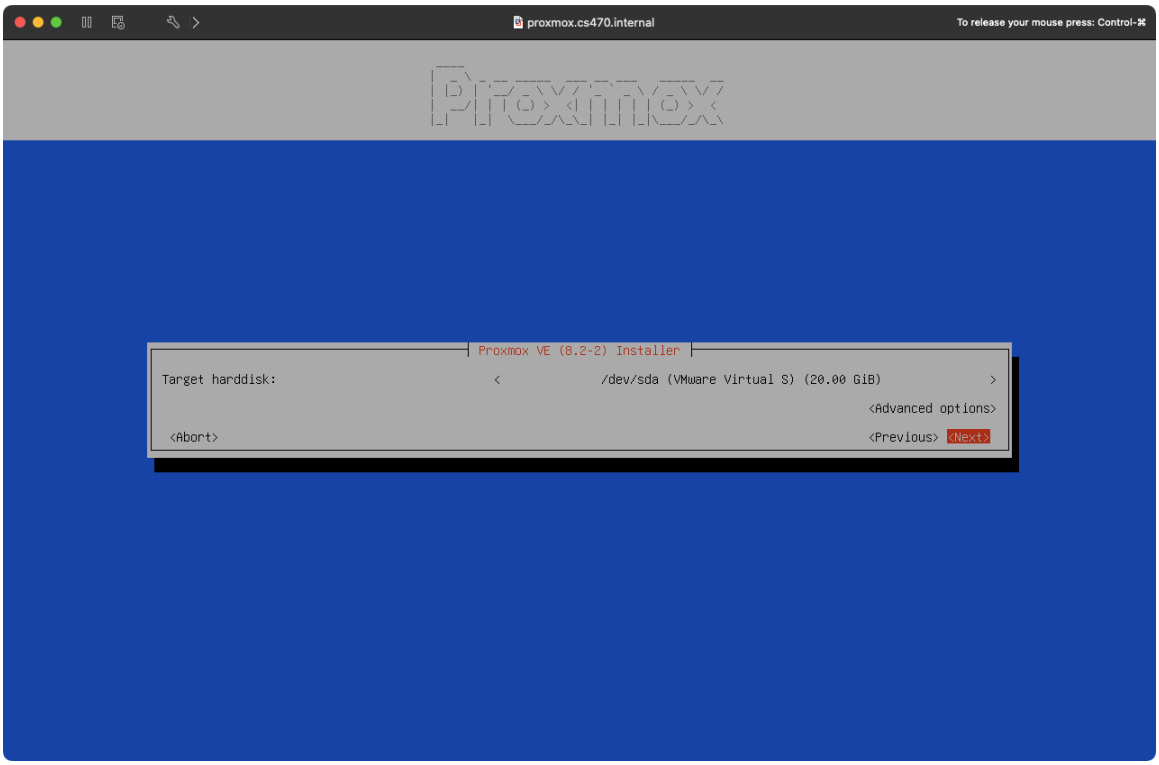
1. You may want to watch your RAM consumption in Activity Monitor.app on macOS, in the Task Manager on Windows, or using `top` on Linux. If you're running low on RAM, and swapping a lot, **you can shut down your OpenIndiana VM during part one, starting now**. Remember to tell them each to shut down on the command line first, not to just tell your hypervisor to shut down the VMs.

2. On booting your VM to do the installation, you'll be greeted by this menu. Choose the terminal UI … we don't need no stinkin' mice here anymore. Before you end up in the installer, you'll see it sense its devices and its network, and grab an IP address from the network, similar to other OSs you're seen recently. Proxmox is Linux, Debian Linux to be precise.

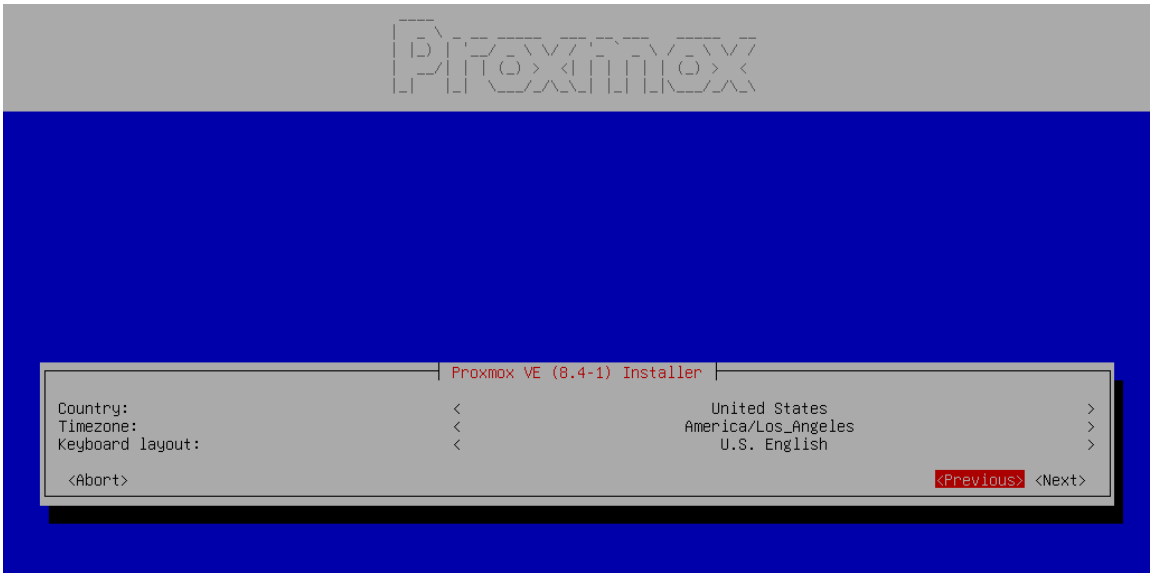


On the first screen, agree to the end user license agreement … consult an attorney, if needed.

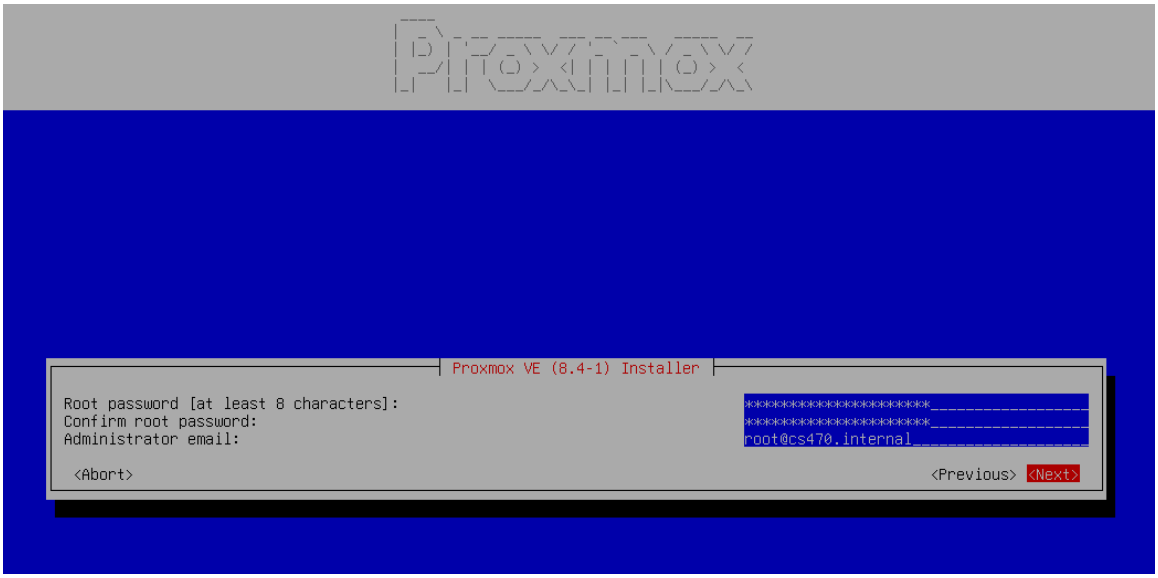
At the disks screen, choose your VM's only virtual hard drive.



At the localization screen, you should know what to do.



At the next screen, you'll be asked to set a root password and an administrator e-mail. Use either root@cs470.internal or your LDAP user's e-mail address.



At the network configuration screen, use the hostname proxmox.cs470.internal and the IP address ending in .76.

Confirm all your settings on the installer synopsis screen. You can leave the checkbox on to automatically reboot after the installation is over.

part two: configure it

After reboot, you'll see Proxmox go through a very familiar-looking configuration of GRUB on its way into its standard no-frills login screen, this time, not the installer.

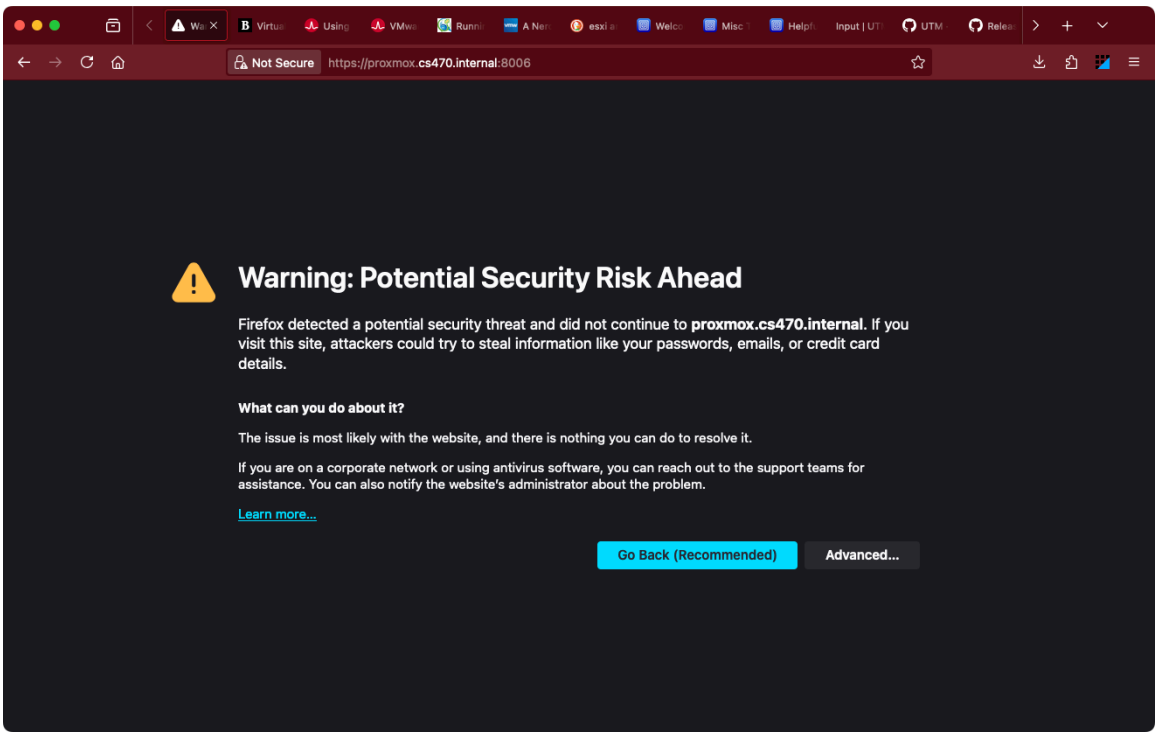
```
Welcome to the Proxmox Virtual Environment. Please use your web browser to
configure this server - connect to:

  https://10.42.77.76:8006/

proxmox login:
```

While starting up Proxmox, VMware may ask for admin rights to your computer's network, especially those of you on Macs. This is because Proxmox is intended to pretend to be many systems with many different IP and MAC addresses, and is putting its network interface into "promiscuous mode" to allow it to see all packets on the network interface. It does this out of the box, every time.

Typically, a data center hypervisor is a lean operating system installed on physical hardware, either manually on the console or by a "jumpstart" network boot and installation in large data center environments where it will be added to a hypervisor cluster. All configuration is usually done remotely by an operator from their computer/desk, because data center hypervisors are the epitome of the "set it and forget it" mentality. Typically, a piece of hardware running a data center hypervisor just becomes more CPU and RAM in the service of a larger fleet in a "private cloud" environment. Storage provided by any given node can also be added to the pool, but a common configuration of hypervisor nodes is to use redundant, mirrored (RAID from lecture) small physical disks to boot the "bare metal" hypervisor, and just the base environment, and then

attach to networked storage, from where VMs are loaded.

3. As the login screen invites you to do, aim your host's web browser at https://proxmox.cs470.internal:8006/ … which should now work if you got the `hosts` file down in the last step. I'm using Firefox here, and you might see a certificate warning, like the following …

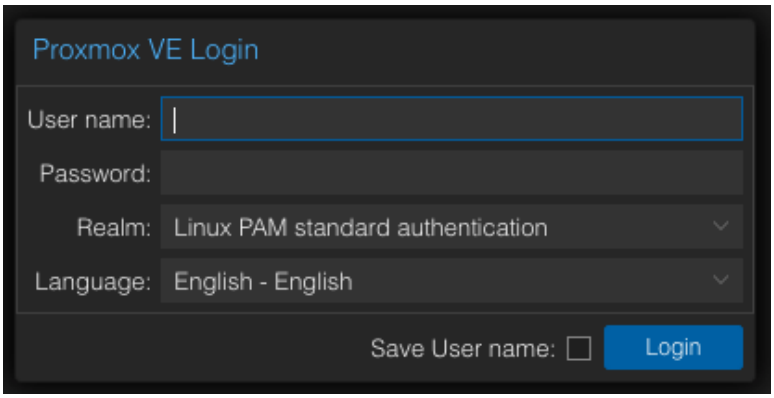


… by clicking the "Advanced" button here in Firefox, I saw what Firefox's problem was with the certificate, that it was self-signed ("unknown issuer"), as expected. Unless you have somebody or something you trust (like the CA on your OpenBSD VM) attesting to the validity of a certificate, anybody could be claiming to be proxmox.cs470.internal … or your bank, or doctor, or insurance company, you get the idea. This is why we need certificates, not merely to enable secure exchange of authentication credentials, but also to enable a modicum of trust in between web servers and each other, and between servers and clients.

For now, though, this is on our private network, at the IP address where we *juuuuuust* set up a Proxmox instance. Clicking to view the certificate should show that it's a Proxmox instance …

```
        Subject Name

Organizational Unit     PVE Cluster Node
        Organization    Proxmox Virtual Environment
         Common Name    proxmox.cs470.internal

         Issuer Name

         Common Name    Proxmox Virtual Environment
Organizational Unit     00bb111b-542f-4449-b799-79b2143237fd
        Organization    PVE Cluster Manager CA
```

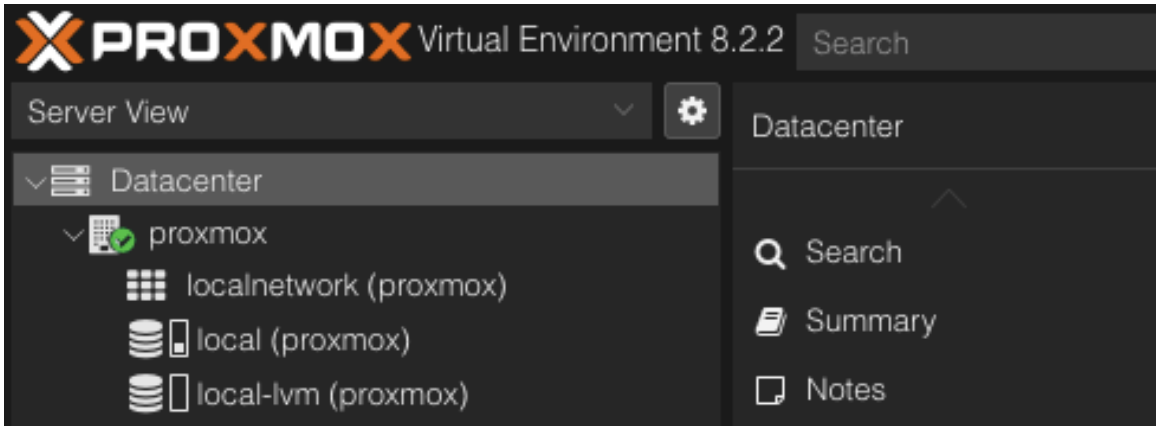
… so it's pretty safe to assume we're not logging into some malicious phishing website. Let's just click the button to "Accept the Risk and Continue." All browsers allow a similar incantation to set trust for an otherwise-unacceptable certificate on a particular system. Then, at last, we come to Proxmox's more-used login interface, via its built-in web server.



It refers to authentication "realms" … worth noting that it's really easy to connect Proxmox to a directory, and we'll try to do that shortly. "Linux PAM standard authentication" refers to the built-in user database in the underlying operating system (`/etc/passwd` and `/etc/shadow`) … so log in as root with the password you set during installation, and find yourself at Proxmox's web-based administrative interface.

Proxmox will warn you about not having a valid subscription. This is because Proxmox is free and open-source software under a "freemium" system, with the base product available for free and advanced features and support available as a subscription. This is as close as it's going to get to asking you for money.

By default, the Proxmox web UI drops you into the "server view" in the left-hand column. Like in a lot of type one hypervisors, only networks (denoted by the nine dots icon) and storage (cylinders, like in spinning hard disks) are visible for configuration.



Compute capacity (RAM and CPU) is available on demand as offered by each hypervisor. Once we fire

up a VM, you'll see it appear below the hypervisor that provides its resources.

4. Let's get cracking setting up our hypervisor. SSH into your Proxmox instance as the root user, and notice that Proxmox is built on top of Debian Linux. Please also notice that Proxmox includes its own custom Linux kernel ("`PMX`").

```
Linux proxmox 6.8.4-2-pve #1 SMP PREEMPT_DYNAMIC PMX 6.8.4-2 (2024-04-10T17:36Z)
x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Nov 19 22:31:57 2024
```

   Debian Linux is the upstream parent project of the Ubuntu Linux distribution. Since you're familiar with Ubuntu, you'll find yourself right at home with Debian. It uses a similar directory structure, similar built-in tools, and the familiar `apt` package manager.

   However, lots of creature comforts are missing … for instance, we don't have `sudo` here.

```
root@proxmox:~# which sudo
root@proxmox:~#
```

   This is because Proxmox expects us to use the root user for administration, or to set up another authentication realm like it alluded to in the login dialog.

   Go ahead and update your `apt` cache. You can ignore the warnings about it not loading from the Proxmox "enterprise" repos; this is because we don't have a subscription … we'll fix that later. The Debian repos are all that's important now.

   Also, copy your SSH public key into `~root/.ssh/authorized_keys` on Proxmox to save some typing.

5. Time synchronization is of critical importance in private clouds; each VM's clock is seeded by the host's clock. Even though Proxmox is a VM here, to its guests, it's the host.

   As the Proxmox wiki states, `chronyd` is the default NTP daemon with Proxmox. Go ahead and check the NTP server's status. Use `systemctl status chronyd` to check that it's properly getting updates from its configured upstream time server.

6. Proxmox stores its web server certificate and key data in `/etc/pve/nodes/` … the folder underneath is named for the hostname of each node, so each of our certificates should be located in `/etc/pve/nodes/proxmox/` …

```
# ls -l /etc/pve/nodes/proxmox
total 2
-rw-r----- 1 root www-data   83 Nov 19 23:28 lrm_status
drwxr-xr-x 2 root www-data    0 Nov 11 19:08 lxc
```

```
drwxr-xr-x 2 root www-data    0 Nov 11 19:08 openvz
drwx------ 2 root www-data    0 Nov 11 19:08 priv
-rw-r----- 1 root www-data 1704 Nov 11 19:08 pve-ssl.key
-rw-r----- 1 root www-data 1801 Nov 11 19:08 pve-ssl.pem
drwxr-xr-x 2 root www-data    0 Nov 11 19:08 qemu-server
-rw-r----- 1 root www-data  560 Nov 19 21:14 ssh_known_hosts
```

Let's get a new certificate in place; this process should be familiar by now. First, move `/etc/ssl/openssl.cnf` to `/etc/ssl/openssl.cnf.orig`, then use `scp` to get `/etc/ssl/openssl.cnf` over from your OpenBSD VM to the same place on Proxmox. Also `scp /home/pki/data/cacert.pem` into `/etc/ssl` on Proxmox for safe keeping. Since we're running around as root here, it'll require less operations, but you should be very careful.

Edit the `alternate_names` section we added to the end of openssl.cnf and change it to reflect the correct hostname for this Proxmox instance, so that we can next generate a new key and a certificate request in `/etc/pve/nodes/proxmox/` …

```
# cd /etc/pve/nodes/proxmox
# openssl genrsa -out pveproxy-ssl.key 4096
# openssl req -new -key pveproxy-ssl.key -out pveproxy-ssl.csr
```

Copy the CSR over to OpenBSD for signature …

```
# scp -p pveproxy-ssl.csr peter@openbsd:/home/pki/newreq.pem
```

… over on your OpenBSD VM, sign it; you should know what to do here by now, then copy it back.

```
# scp -p peter@openbsd:/home/pki/newcert.pem pveproxy-ssl.pem
```

Proxmox expects a "full chain" PEM file in `pveproxy-ssl.pem`, so add the contents of your root CA certificate file to it.

```
# cat /etc/ssl/cacert.pem >> pveproxy-ssl.pem
```

Just like on Ubuntu, we also want to add the CA certificate to the system trust anchors …

```
# cp -p /etc/ssl/cacert.pem /usr/local/share/ca-certificates/cs470.internal.crt
```

… and tell the operating system to re-parse its cache of CA certificates.

```
# update-ca-certificates
```

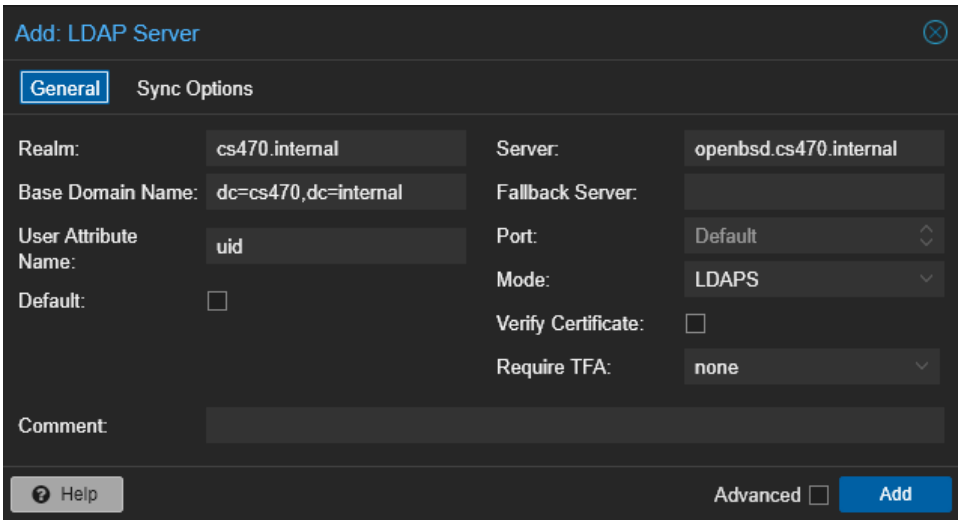
Finally, you should be able to restart Proxmox's proxy process that governs all HTTP/HTTPS traffic and re able to re-load your Proxmox instance's web interface in your web browser with a trusted certificate.

```
# systemctl restart pveproxy
```

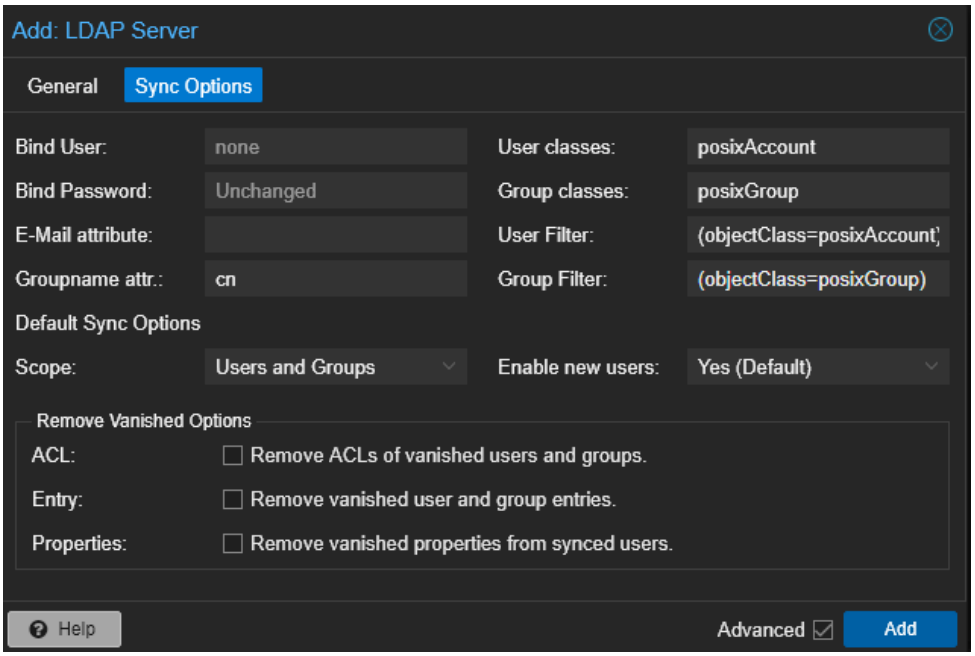
7. Now that we have our root CA's chain of trust configured into Proxmox, it's time to add LDAP.

Go back to the web UI, click on "Datacenter," then "Permissions," then "Realms." Pull down the "Add" drop-down menu to "LDAP Server."

I recommend you call your realm cs470.internal … you should know what to put in here, for the most part. Use your user OU as your base domain name, `uid` as the user name attribute (I think it has name and attribute backwards here), LDAPS for the mode, and there's no fallback server … that'd be the hostname if we had a backup LDAP server, but we don't.



Then click on "Sync Options." For user class, use `posixAccount`. For the group class, use `posixGroup`. Group name attribute: `cn` … user filter: `(objectClass=posixAccount)` … group filter: `(objectClass=posixGroup)` … scope: "Users and Groups."

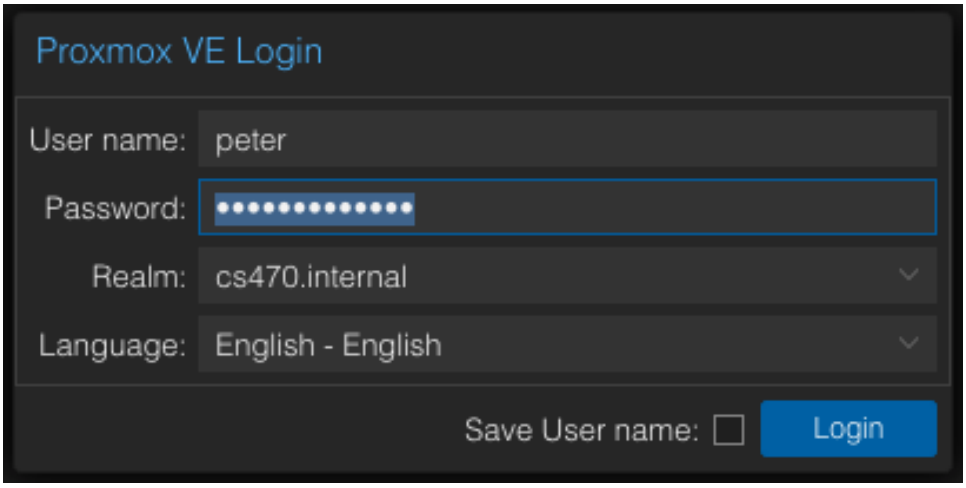


Hit OK. You should then see cs470.internal listed in the realms on the right. Select it, then click on

"Sync" to force Proxmox to grab a copy of pertinent users and groups matching the filter. Note any errors in the output it shows.

Next, click the "Permissions," just above the "Realms" section we were just configuring, then pull down the "Add" drop-down to "User Permission." Set the path to the root (`/`), choose your LDAP user, and make it an administrator with the "Role" drop-down.

You should now be able to log in as your LDAP user for the rest of the lab.



8. In this lab, we'll use NFS to store both our backup data and our Proxmox VMs. We're going to use the spare disk we created in lab 3 and mounted on `/srv/nfs` to store this data, so that we can dispose of this data later to save space if you want to keep your labs.

   Fortunately, we often don't need as much storage as the provisioned size of our VM's virtual storage devices to store a VM, because of thin provisioning. When thin provisioning is configured with a virtual hard disk, the virtual hard disk only consumes filesystem space on the host when virtual storage blocks are written by the guest. Though we configured 16 GB of disk space on our FreeBSD VMs, as you can confirm by the root filesystem line in `df` output …

```
$ ssh freebsd df -h
Filesystem          Size    Used   Avail Capacity  Mounted on
/dev/da0p2           14G    6.8G    6.8G    50%    /
devfs               1.0K      0B    1.0K     0%    /dev
/dev/da0p1          260M    1.3M    259M     1%    /boot/efi
rocky:/home          18G    2.3G     14G    14%    /home
rocky:/srv/nfs       39G    1.7G     35G     5%    /srv/nfs
```

   … but thus far it's only used about 6.8 GB of that space on its root filesystem. More than that 6.8 GB is used, by content now deleted, or by swap space, but as is visible when I look at the actual size of my VM on my host's disk, it's still using nowhere near 16 GB.

```
$ du -sh ~/doc/sdsu/VMs/cs470/freebsd.cs470.internal.vmwarevm
8.2G   /Users/peter/doc/sdsu/VMs/cs470/freebsd.cs470.internal.vmwarevm
```

This is the default behavior of virtual hard disks in VMware desktop hypervisors, and with Qemu's default virtual hard disk format, the `.qcow` file (short for "Qemu copy-on-write"). Those of you on ARM Macs can see your VMs in `~/Library/Containers/com.utmapp.UTM/Data/Documents` if you're curious about the host footprint they take up. Windows users, look in `/mnt/c/Users/*/Virtual Machines` via WSL.

!! IMPORTANT NOTE: **take note of your storage device name** in your FreeBSD from the output of `df`. If it starts with `ada`, it's on a virtual SATA or PATA disk in your VM. If it starts with `da`, it's on a SCSI disk. Those of you on UTM probably have a `vtbd` device, which is a VirtIO device. Whatever you had in your original FreeBSD VM, we're going to replicate later.

To address the (intentional) lack of local storage on our Proxmox VM, we're going to plug them into the larger pool of storage accessible via the NFS server on our Rocky VMs. Go into your Rocky VM, and make a directory for Proxmox storage …

```
$ sudo mkdir -m 1777 /srv/nfs/proxmox
```

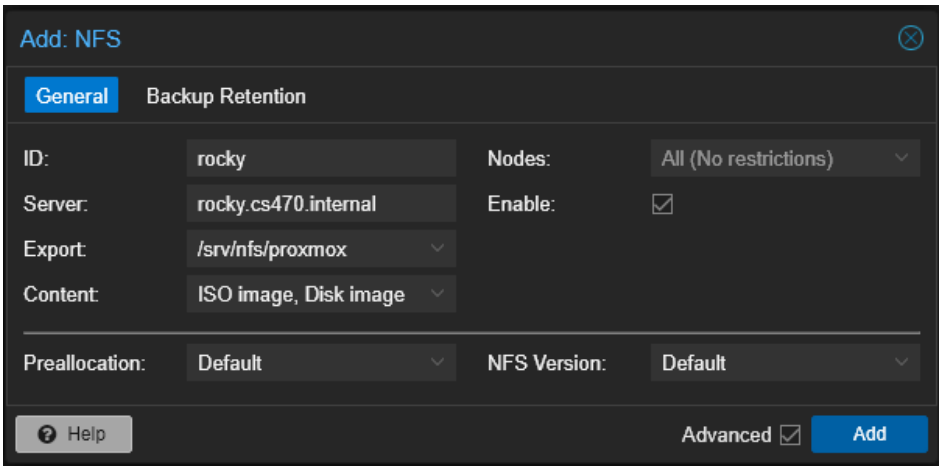
… add the following line to `/etc/exports` to share this new directory as a mount point with only your new Proxmox VM …

```
/srv/nfs/proxmox        10.42.77.76(rw,sync)
```

… then export it to the network with `exportfs -a` on your Rocky VM.

Log back into the Proxmox web interface, click on "Datacenter" in the "Server View" pane at the left, then "Storage" in the datacenter screen at the right. Pull down the "Add" drop-down to NFS, and supply `rocky` as the ID for the share and `rocky.cs470.internal` as the server. When you click the drop-down widget to the right of "Export:" you'll see it pause for a second while it scans available shared folders on the NFS server. If you did everything correctly, you'll be able to select `/srv/nfs/proxmox`.

In the "Content" drop-down, select to store ISO images and disk images … the options in this drop-down don't look like they are each on/off selections, but they are. Whereas VMware stores all back-end resources for each VM together in a folder, Proxmox stores all VM configurations in selected places, the disk images in others, the templates and ISOs for VMs in others, and so on. Each of these directories corresponds to a tab that shows up in Proxmox when we select our NFS server from the storage options. On `/srv/nfs`, we want to store both ISO files for creating VMs, and virtual disks for our VMs.

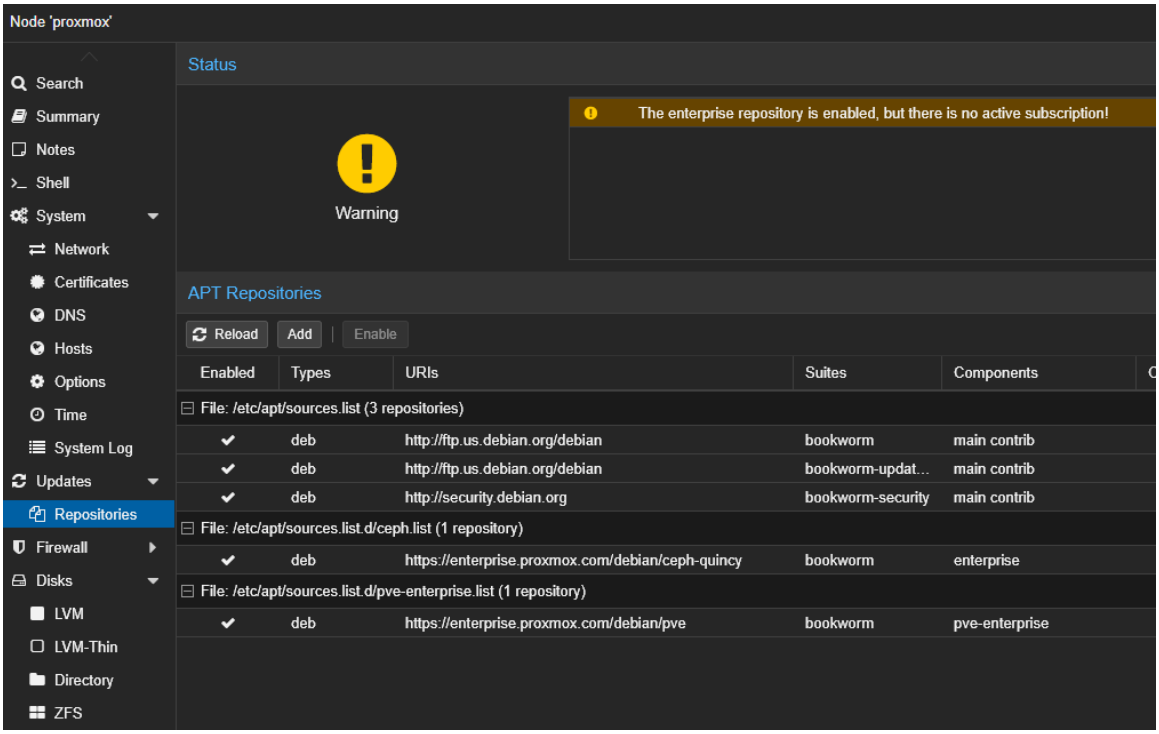


Finally, click on "Add" to enable storage to NFS in your Proxmox instance. After you do, it should be active. If you look inside that new Proxmox folder on your NFS server, you should see that Proxmox has made itself at home.

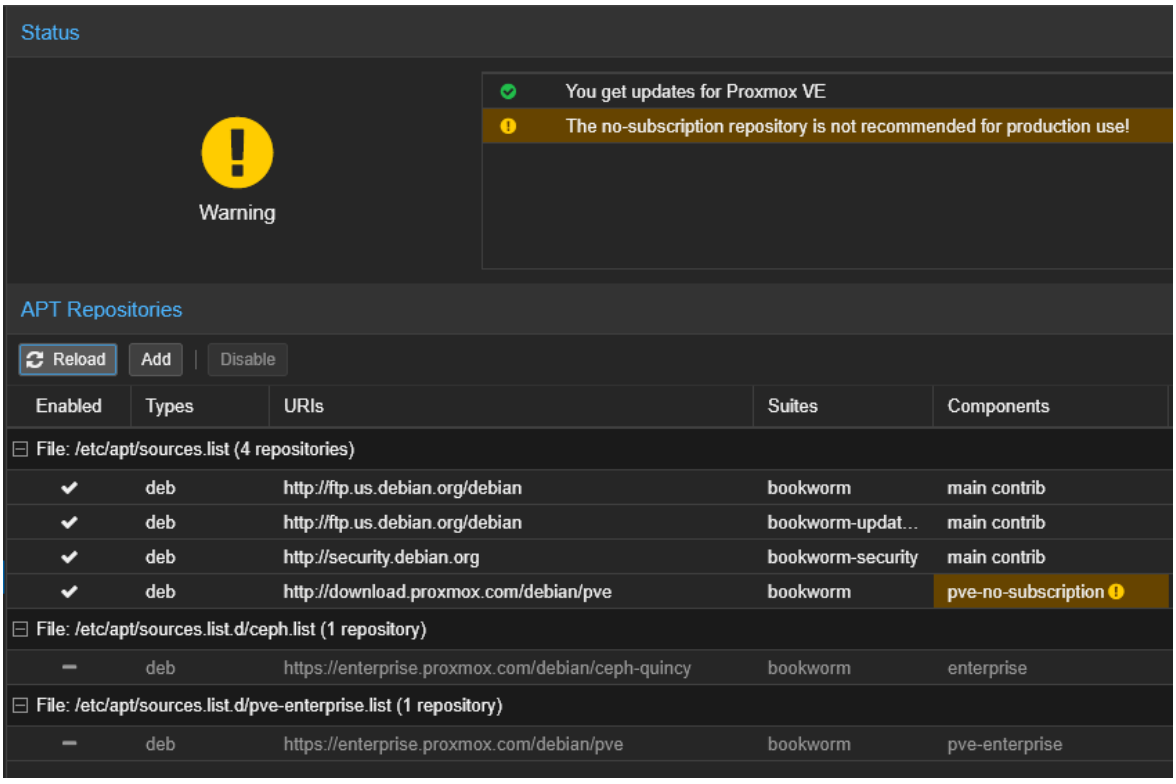
```
$ ssh rocky ls -l /srv/nfs/proxmox
total 1
drwxr-xr-x. 2 nobody nobody 4096 Apr 17 01:04 dump
drwxr-xr-x. 2 nobody nobody 4096 Apr 17 01:04 images
drwxr-xr-x. 2 nobody nobody 4096 Apr 17 01:04 private
drwxr-xr-x. 2 nobody nobody 4096 Apr 17 01:04 snippets
drwxr-xr-x. 4 nobody nobody 4096 Apr 17 01:04 template
```

9. Just like Ubuntu in lab 4 or a regular Debian system, you can update the base operating system with `apt`. Proxmox does a couple minor things to make updates harder to do out of the box without a subscription, and only allows updates from "no-subscription" repositories, which may include less-tested and potentially unstable updates.

   We're going to ride the bleeding edge and do it anyways. Back in Proxmox's web interface, find the Proxmox node, and select the Repositories tab under Updates.



First, select both enterprise repositories and disable them. Next, add a new repository, and select No-Subscription from the dropdown. Click “Add” to finalize it and close the dialog, and click “Reload” to put your changes into effect. You can ignore Proxmox whining about the no-subscription repository.



Then, update the package cache with `apt update`, and to install all patches, `apt dist-upgrade`.

10. Now let's test e-mail; the Proxmox installer has `postfix` in a very lean but effective configuration in `/etc/postfix/main.cf` if you take a look. My instance worked pretty much right away.

    Edit `/etc/aliases` and insert an alias for root to aim at your LDAP user's e-mail address, then run `newaliases`, then test. Proxmox already has a `mail` binary; no need to go hunting for packages to test e-mail here.

    We're done with Proxmox now, for a little bit. Those of you on ARM Macs are done with it, period – even if we could run a nested VM on an ARM Mac (we can't) – Proxmox is an amd64 VM, and your FreeBSD instance is an arm64 VM. You wouldn't want to try to emulate an ARM CPU inside an emulated Intel CPU … if you have an ARM Mac, you can shut down the Proxmox VM altogether, and be glad you got to taste-test it.

part three: backup

It's time to take a backup of your FreeBSD VM.

11. Because we're running this under controlled circumstances, we're able to shut down our VM to ensure a clean image – no services will be altering data while we take a backup – but we rarely have this luxury in production. More often than not, you'll want to back up a production system live to keep services running … and will want to find the correct switches to use with your backup utility for backing up a live, mounted filesystem.

    Let's boot our FreeBSD VM into single user mode. First, log in on the console, and then tell it to reboot.

    ```
    $ sudo reboot
    ```

    When you get back to FreeBSD's boot menu, hit `2` to go into single-user mode. Sometimes, just like with the installer, device detection continues past the presentation of a prompt … here, in the screenshot below, you can see FreeBSD note that authentication backends have an issue – how could LDAP work without a network? – just before FreeBSD asks for the full pathname of a shell, or return for `/bin/sh`. Hit `return` or `enter`, and find yourself at a `#` prompt. We're in single-user mode now, with a kernel loaded and not much else … I want you to think of this as a blank slate, and you're about to see why.

    Please note the `ifconfig` output in the next screenshot, confirming that the network is not configured. `em0` may have a cable connected ("`status: active`") but it has no IP address and is not marked as "UP."

12. Bring your network interface up by configuring it manually. For me, the right command was the following. You may have to substitute another network interface name, and as always, will almost certainly have to swap out some octets in my IP address for yours.

    ```
    # ifconfig em0 10.42.77.72 netmask 255.255.255.0
    ```

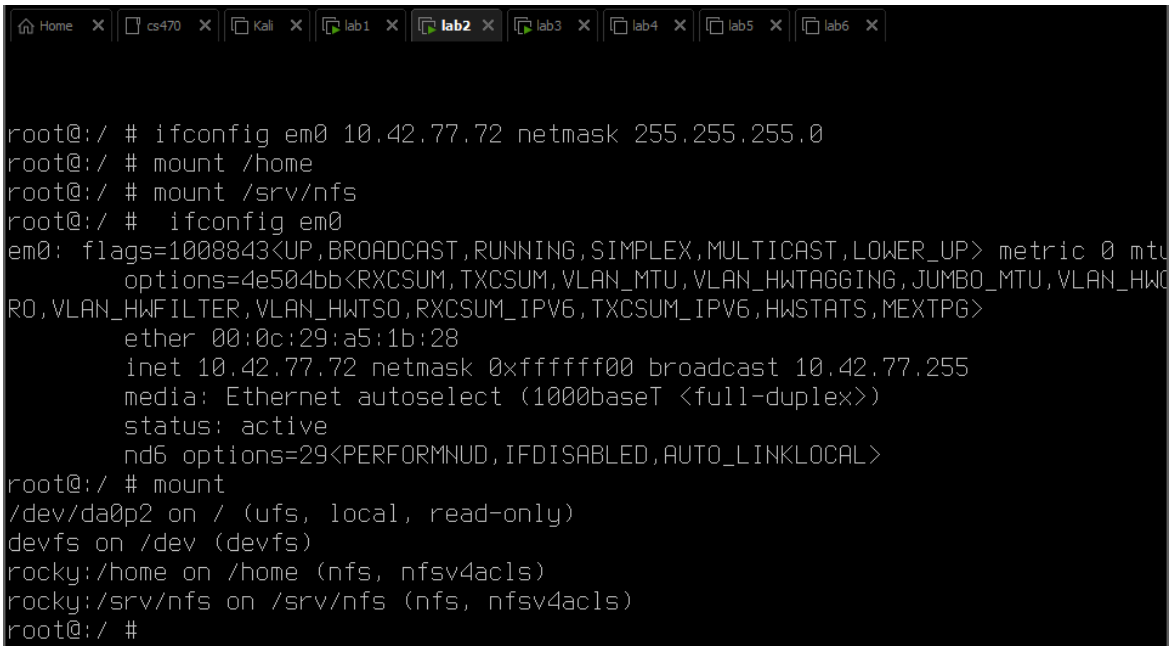
    You might see a boldface kernel log message stating that this command has brought your interface's link state to "UP." Using `ifconfig` again should confirm it.

    Since your name server is already configured in `/etc/resolv.conf`, and your name server and your lab VMs are on the same subnet as your FreeBSD VM – you don't need a default route configured to talk to them – you should now be able to ping your other lab VMs by name.

13. Let's try mounting both shares from our Rocky VM.

    ```
    # mount /home && mount /srv/nfs
    ```

    Now we have our backup source exactly as we want it: our FreeBSD VM's root filesystem is mounted read-only, so nothing is in motion and we know we'll get a clean image, and more than enough disk space is available on a separate device (the NFS share) to accept the backup. Note: this could be an external hard disk, or a flash drive ... but network storage is much faster, and as you have now seen, it's also not very difficult to get connected.



14. Let's start `sshd` on our FreeBSD VM.

    ```
    # /etc/rc.d/sshd start
    ```

    The startup script for `sshd` reported that it was "performing [a] sanity check on [the] `sshd` configuration," and then that it was starting `sshd`. The output of `ps` should now show that you're running `sshd`.

    You should now be able to log into your FreeBSD VM using SSH.

    Even though your FreeBSD VM has its root filesystem mounted read-only, has no default route configured, and has literally nothing besides the kernel, `sshd`, and a couple shells started up … it's still useful, up to the point that you configure it. If you're like me and your hostname is a part of your shell prompt you might notice that the system doesn't even have a hostname configured right now.

    Again, it is a blank slate. Remember: this can be very useful for isolating issues when troubleshooting.

```
$ ps auxww
USER  PID  %CPU %MEM    VSZ    RSS TT  STAT STARTED     TIME COMMAND
root   11 200.0  0.0      0     32  -  RNL  22:13   30:19.11 [idle]
root    0   0.0  0.2      0    480  -  DLs  22:13    0:00.58 [kernel]
root    1   0.0  0.5  11700   1076  -  ILs  22:13    0:00.00 /sbin/init -s
root    2   0.0  0.0      0     32  -  WL   22:13    0:00.34 [clock]
root    3   0.0  0.0      0     48  -  DL   22:13    0:00.00 [crypto]
root    4   0.0  0.0      0     48  -  DL   22:13    0:01.35 [cam]
root    5   0.0  0.0      0     16  -  DL   22:13    0:00.00 [busdma]
root    6   0.0  0.0      0     16  -  DL   22:13    0:00.00 [mpt_recovery0]
root    7   0.0  0.0      0     16  -  DL   22:13    0:00.06 [rand_harvestq]
root    8   0.0  0.0      0     48  -  DL   22:13    0:00.05 [pagedaemon]
root    9   0.0  0.0      0     16  -  DL   22:13    0:00.00 [vmdaemon]
root   10   0.0  0.0      0     16  -  DL   22:13    0:00.00 [audit]
root   12   0.0  0.3      0    688  -  WL   22:13    0:00.24 [intr]
```

```
root    13    0.0  0.0      0     48  -   DL    22:13     0:00.01 [geom]
root    14    0.0  0.0      0     16  -   DL    22:13     0:00.00 [sequencer 00]
root    15    0.0  0.1      0    160  -   DL    22:13     0:00.10 [usb]
root    16    0.0  0.0      0     32  -   DL    22:13     0:00.01 [bufdaemon]
root    17    0.0  0.0      0     16  -   DL    22:13     0:00.00 [vnlru]
root    18    0.0  0.0      0     16  -   DL    22:13     0:00.00 [syncer]
root    30    0.0  0.0      0     16  -   DL    22:22     0:00.00 [nfscl]
root    33    0.0  0.0      0     16  -   DL    22:22     0:00.00 [nfscl]
root    57    0.0  4.4  23620  10032  -   Is    22:25     0:00.00 sshd: /usr/sbin/sshd
[listener] 0 of 10-100 startups (sshd)
root    58    0.0  5.5  24312  12372  -   Is    22:26     0:00.04 sshd: peter [priv](sshd)
peter   60    0.0  5.7  26872  12836  -   S     22:26     0:00.03 sshd: peter@pts/0 (sshd)
root    19    0.0  3.0  22304   6788 v0   Is+   22:13     0:00.05 -sh (tcsh)
peter   61    0.0  5.2  25440  11696  0   Ss    22:26     0:00.05 -tcsh (tcsh)
peter   73    0.0  1.3  13452   2888  0   R+    22:28     0:00.00 ps auxww
```

As mentioned previously in lecture, all of the processes with names inside hard brackets are parts of the kernel, mocked up to look like processes, so that they can be monitored and manipulated like processes.

15. In that shell in SSH, let's start a backup of the root filesystem, to that folder … note you may have to substitute in the name of your root filesystem's device (`/dev`) file, and may want to adjust your output filename accordingly …

```
$ dump auf /srv/nfs/freebsd-da0p2.dump /dev/da0p2
```

… I want you to note a few things here.

First, you don't need to be root to back up the filesystem (look at the permissions on `/dev/da0p2` or whatever your disk partition device file is) because you're in the operator group, unless you missed that step. This is a historic privilege delegation so that rootly powers didn't need to be granted to overnight sysadmins ("operators") to allow them to run backups.

```
$ ls -l /dev/da0p2
crw-r-----  1 root operator 0x63 Jul 26 22:13 /dev/da0p2
```

Moreover, I have retained the name of the hostname and the filesystem in the name of the backup file. Not only is this just helpful, but it will become really important if you have multiple hosts and multiple filesystems in backup storage. I'd typically also recommend you include a date stamp in each backup file name, but we can skip that here.

Finally, the syntax for `dump` is designed to mirror the syntax of the `tar` command it was designed to replace, for many use cases. The archaic `a` switch prevents `dump` from caring about any predefined tape size limits … `a` is short for "all," as in "dump it all." The `u` switch tells dump to update the file `/etc/dumpdates` where it keeps track of backups, for determining which files to grab in incremental backups … this will generate an error, because our root filesystem is still mounted read-only, but we'd want to use this switch ordinarily. The `f` switch says the next argument is the filename of the backup target. Historically, this could be the device file name of a tape drive, an optical disk, or as we provided above, a regular file name.

You will see `dump` immediately spin up through several rounds of imaging your root filesystem. I went

ahead and interrupted the backup with a `control-c` and confirmed with a `yes` to interrupt the backup, and get a command line back. Please remove the incomplete backup … and don't worry, we're going to do this another way … a really cool way that'll help set you up for how to do distributed backups in the future.

```
$ rm /srv/nfs/freebsd-da0p2.dump
```

16. **Log out of the SSH session. Get back on the console of your FreeBSD for this step.**

    Let's make a directory to separate storage for artifacts from this lab.

    ```
    # mkdir -m 770 /srv/nfs/freebsd
    ```

    Note that we made the `freebsd` directory group-writable, but not world-writable with the permissions `770` above. It would've been really hard for somebody to compromise your NAT'd VM network, but this is to offer *some* protection of your backup files somewhat from imaginary bonus users of our private network. *Just because you're paranoid doesn't mean they're not after you.*

    Now let's copy over copies of the `dump` and `restore` binaries, in case we need them while we're restoring the backup later, and a copy of the filesystem table, `/etc/fstab`, into the NFS share.

    ```
    # cp -p /sbin/dump /sbin/restore /etc/fstab /srv/nfs/freebsd/
    ```

    Note that the output of this command …

    ```
    # ldd /sbin/dump /sbin/restore
    ```

    … shows that `dump` and `restore` are dynamically-linked binaries. Traditionally, binaries under `/sbin` are statically linked (hence the `s` in `sbin`) versions that can be used when the operating system isn't fully started up, and the dynamic linker isn't ready to go, such as in single-user mode. Single-user mode has apparently evolved, as our first `dump` started to go.

17. SSH into your Rocky VM from your host operating system, and look at the directory we made recently to host the backup, using the `d` switch with `ls` to look at the permissions and the directory itself, instead of its contents.

    ```
    $ ls -ld /srv/nfs/freebsd/
    drwxrwx---. 2 nobody nobody 4096 Apr 17 23:22 /srv/nfs/freebsd
    ```

    Note that the directory is owned by the nobody pseudo-user, because we were root (by way of `sudo`) in that shell, and as we said in lab 3, NFS squashes the root user into the nobody account over NFS shares, for security purposes. Just because somebody is root over there, doesn't mean we want them to bypass access controls here, on our file server.

    Also note that you are currently unable to use those group permissions and list the contents of the directory itself ...

```
$ ls -l /srv/nfs/freebsd/
ls: cannot open directory '/srv/nfs/freebsd/': Permission denied
```

... because you're not a member of the nobody group ...

```
$ id
uid=1001(peter) gid=1001(peter) groups=1001(peter),998(render),999(input)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

... and typically, that's the idea behind the name "nobody." Not only is root nobody on our file server, but nobody should really be using root rights to push files around and accidentally disclose things they shouldn't. This time, however, we want to be able to share files with that remote user, and to prevent our imaginary friends in our lab environment (we really have no other *intended* users besides ourselves in them) from seeing the sensitive backup file we're about to create.

!! IMPORTANT NOTE: if we had finished creating that backup file, and left it in `/srv/nfs`, then anybody with access to our file server would be able to read ... a backup of our FreeBSD VM and our mail server. Since it's just each of us on our own network here … I'm just going to point that out, and we're going to keep moving ... but if this was a production environment, and there was a real live name server and directory server in that dump-file ... wow, that would be bad.

Please edit `/etc/group` and add yourself to the nobody group and make sure you can read and write to that directory with your regular user account on Rocky Linux. You should know how to do this by now, and what you have to do to get your new permissions.

18. With that set up, now use `ssh` from your Rocky VM to get into your FreeBSD VM and remotely dump its root filesystem to stdout (using a dash as the `dump` filename, like we did in lab one while unpacking the `src` and `ports` trees). This causes the output – the backup data itself – to come back to the local system, where we use that to pipe it into a local `gzip` command on your Rocky VM, to compress the backup before stuffing it into a file. This command will ask you for a password, the login password for your FreeBSD VM, before it begins … unless, of course, you're using `ssh-agent` forwarding.

    As always, make sure the device file names match yours, in the command below before running it …

    ```
    $ ssh freebsd dump af - /dev/da0p2 | gzip -c > /srv/nfs/freebsd/da0p2.dump.gz
    ```

    If you did your lab two and three setups correctly, this should work. If not, you should know what to do by now. That last command may take a while to run, depending on the speed of your computer, primarily your CPU and storage device (an SSD helps a bunch here). Let it run until finished this time.

    ```
    DUMP: Date of this level 0 dump: Fri Jul 26 23:26:10 2024
    DUMP: Date of last level 0 dump: the epoch
    DUMP: Dumping /dev/da0p2 (/) to standard output
    DUMP: mapping (Pass I) [regular files]
    DUMP: mapping (Pass II) [directories]
    DUMP: estimated 6542463 tape blocks.
    DUMP: dumping (Pass III) [directories]
    DUMP: dumping (Pass IV) [regular files]
    ```

```
  DUMP: 46.07% done, finished in 0:05 at Fri Jul 26 23:37:03 2024
  DUMP: 93.62% done, finished in 0:00 at Fri Jul 26 23:36:52 2024
  DUMP: DUMP: 6543904 tape blocks
  DUMP: finished in 640 seconds, throughput 10224 KBytes/sec
  DUMP: DUMP IS DONE
```

When it is done, you'll have a full backup of your FreeBSD VM's root filesystem on your NFS share, under `/srv/nfs/freebsd`. We used `gzip` because the FreeBSD version of `dump` doesn't have compression hooks; the size of the backup to be about the size of the used space on the filesystem (6.8 GB), but the compressed backup was just under 3 GB for me. If your results are slightly different, it's not anything worth caring about, so long as compression is demonstrably working to save some space.

```
$ ls -l /home/nfs/freebsd
total 3.0G
-rw-r--r--. 1 peter  peter  3.0G Jul 26 23:36 da0p2.dump.gz
-r-xr-xr-x. 1 nobody nobody  57K May 31 02:05 dump
-rw-r--r--. 1 nobody nobody  245 Jun 24 03:32 fstab
-r-xr-xr-x. 1 nobody nobody  82K May 31 02:05 restore
```

Also note: if our VM NAT network were a company network, we would never, never ever expose a full backup of a system like this (especially a backup of the mail server!).

Also also note: if we had multiple filesystems on the computer we were backing up … we'd just repeat `dump`, for each of the filesystems, each to its own output file. That is, of course, if they're filesystems intended for Unix or Linux … if they're not, `dump` won't know what to do.

19. Of course, we have precisely one of those cases here, an EFI system partition used to store boot loaders. On your FreeBSD VM, check out `/etc/fstab` …

    ```
    $ cat /etc/fstab
    ```

    … unless you messed up picking your firmware in lab 2, you've got an MS-DOS (`msdosfs`) filesystem that is ordinarily mounted at `/boot/efi`. If you don't have a filesystem mounted on `/boot/efi` take no action, but make sure you understand the rest of this step anyways.

    ```
    $ sudo dump 0af /srv/nfs/freebsd/da0p1.dump /dev/da0p1
    sudo: unable to open /var/run/sudo/ts/1001: Read-only file system
    Password:
      DUMP: Date of this level 0 dump: Fri Jul 26 23:48:22 2024
      DUMP: Date of last level 0 dump: the epoch
      DUMP: Dumping /dev/da0p1 to /srv/nfs/freebsd/da0p1.dump
    dump: Cannot find file system superblock: No error: 0
    ```

    You can ignore this warning …

    ```
    sudo: unable to open /var/run/sudo/ts/peter: Read-only file system
    ```

    … all it means is that `sudo` isn't able to store its temporary files. This is because the root filesystem is mounted read-only by design right now, in order to get a cleaner backup. This means you'll not be given a grace period after each successful use of `sudo`, and will have to type your password each time you use it. Not a big deal, all the more so with the failure of the overall operation.

With `Cannot find superblock`, `dump` is telling you that it's looking for evidence of a Unix filesystem and failing to find it. This is one of those places where `tar` still excels, as a (mostly) filesystem-agnostic tool. Use `tar` to back up the EFI system partition.

```
$ cd / && sudo tar zcvf /srv/nfs/freebsd/da0p1.tar.gz ./boot/efi
```

You should see just a few files and a couple directories; it's not much.

20. On your FreeBSD VM's console, shut it down and power it off.

```
# shutdown -p now
```

part four: replica setup (Intel CPUs)

If you have an Intel CPU, you'll create a VM in Proxmox – a VM inside a VM! – and restore your backup to it. If you have an ARM Mac, please don't hesitate to read this to see what you're missing, but take no action until the version of part four for your ARM Mac, a couple pages ahead.

21. Let's download a copy of the correct FreeBSD ISO to where our Proxmox can use it.

    Then, in Proxmox's web UI, open up "Datacenter," then "proxmox," then your NFS storage. On the right, click on "ISO Images." You can then either click "Upload" then find the file on your host computer's local filesystem, or you can click "Download from URL" and find the download URL for the FreeBSD 14.1 amd64 ISO we used in lab 2.

22. Next, select "proxmox" and click "Create VM" in the upper right-hand corner of the web interface.

    In the first pane, "General," all you need to do is make sure the node is set to proxmox, your hypervisor, and pick a name for your VM. I used freebsd.cs470.internal. You don't need to touch anything else here.

    In the second pane "OS," choose to use an ISO and select your FreeBSD ISO from the prior step from your NFS server. Under "Type," choose "Other." Next.

    In the "System" pane, choose to use a "q35" machine, an UEFI firmware (BIOS), and a VirtIO SCSI controller. Create an EFI disk using your NFS server as the backing store, and uncheck the box to pre-enroll keys; this is for secure boot and FreeBSD doesn't support secure boot. Next.

    In the "Disks" pane, **select to use a SCSI bus if you had a `da` device in your original FreeBSD VM, or a SATA bus if you had a `wd` device.** Choose to use your NFS server as backing storage, and a disk size of 16 GB unless you used more in your original VM. That's it here; there's nothing to do in the "Bandwidth" pane. Next.

    ‼ IMPORTANT NOTE: Again, remember your hard disk type from your original FreeBSD VM… whether it

was a `wd` device (SATA/PATA) or an `sd` device (SCSI), make sure you make the correct substitutions in the rest of this lab.

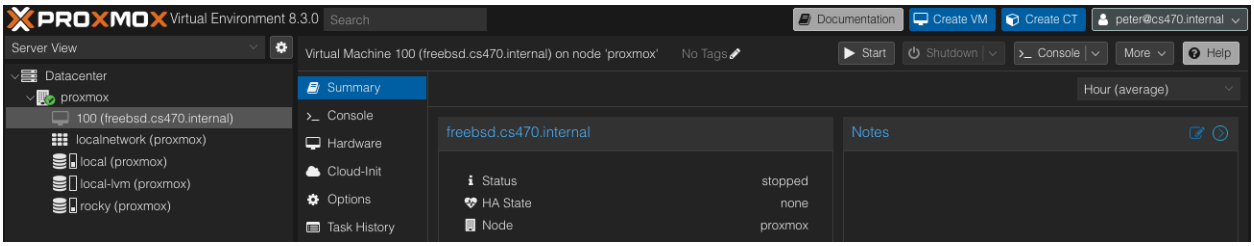
In the "CPU" pane, choose to use one socket and two cores. The standard CPU type is fine. Next.

In the "Memory" pane, choose to use 512 MiB of RAM. Next.

On the "Network" pane, choose to use the only bridge available – we haven't set up any network abstractions here – to share the network with your Proxmox host and be directly connected to your VMware NATwork. The default mode, an Intel E1000, is fine too. Turn off "firewall." Next.

Review your settings in the "Confirm" pane, and hit "Finish" to conclude the VM setup.

23. You should now see your first VM, freebsd.cs470.internal, under your hypervisor in the Proxmox UI.



Note the web UI's "console" menu in your new VM's view; you'll use a "NoVNC" console to provide the UI here for the remaining steps in this lab. When you open it up, you'll get a power button in that console window.

part four: replica setup (ARM Macs)

On ARM Macs, we're going to create another VM in UTM to use as our replica and restore target.

Create a new VM in UTM, with virtualization:

- guest OS: other
- CPU: two cores
- RAM: 512 MB
- hard disk: 20 GB

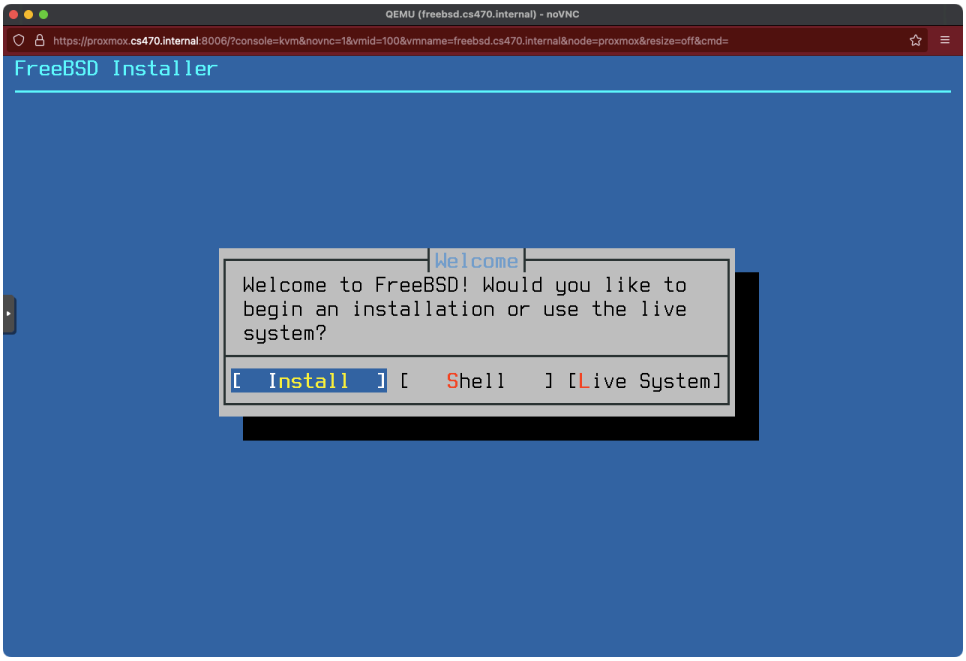
Put `FreeBSD-14.2-RELEASE-arm64-aarch64-disc1.iso` from lab two into the virtual CD/DVD drive.

part five: restore from backup

Generally, as you'll see, re-creating a system, whether it's a physical-to-virtual machine conversion, a disaster recovery, or whatever involves configuring and partitioning storage, creating filesystems, restoring content

into those filesystems, adjusting for any changes to configuration, and making sure the new storage devices boot properly. We go through all these steps in this part of the lab.

24. Boot up your replica VM – it may take a bit to fire up on Proxmox – and choose a "multi-user" boot at the installer boot loader menu. You should be greeted by familiar options.

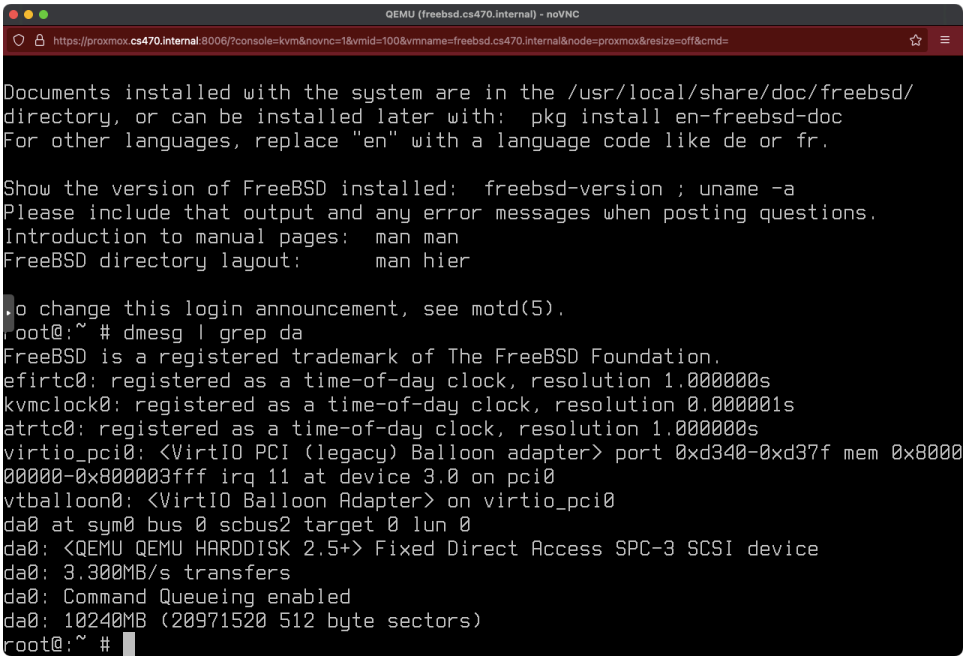


This time, let's choose the "live system" option to give us an interactive rescue environment we can use to restore our FreeBSD VM from the backup we just made. After you choose "live system" you'll be dropped out to a login prompt. Use `root` as your username; it will not ask for a password.

When you get to a shell, use the `dmesg` command to confirm you've got a device named `ada0` or `da0` or `vtbd0` or whatever you had in your original VM, of the appropriate size …

```
# dmesg | grep da
```

… `dmesg` is short for "device message," and it spits out the same list of device detection messages the FreeBSD kernel shows during boot time. This is really handy if you're ever curious what's attached to a machine, you want to confirm your storage bus type like we're doing now, or you just missed it as it went by during the boot process. My output looked like the below … yours may differ slightly, within reason, but should generally be the same.



If you're having issues finding the device using `grep` as a filter, try piping `dmesg` through `more` and slowly reading what you see. The storage device should be obvious once you set eyes on it. On my ARM Mac, in UTM, my original storage device was `vtbd0`, but FreeBSD/arm64 declares the presence of `vtblk0` instead. I don't know why the ARM64 port does this – it seems to break with decades of tradition, but the command …

```
# geom disk list
```

… showed that even though `dmesg` declared `vtblk0`, the disk device is really `vtbd0`. Looking in `/dev` confirmed this as well. Strangely, there is no `man` page for either `vtbd` or `vtblk`. There's a page for `virtio_blk` that seems to be appropriate, but doesn't say why or how. I'll continue searching for this later; if any of you find anything, please let me know.

Substitute whatever disk device you have in the commands below. I tried to set it up so that you'd not have to change `/etc/fstab` after unpacking the backup image, but if you have a different disk device, remember to change `fstab` appropriately.

25. Let's partition the replica VM's virtual hard disk. These commands will parallel not only what was done for FreeBSD by its installer in lab 2, but also what's been done with every other VM to set up the disk.

    First, let's lay down a fresh GUID/GPT partition table on `/dev/da0`, my VM's virtual hard disk …

    ```
    # gpart create -s gpt da0
    ```

    `gpart` should reply with `da0 created` … if not, something's wrong, and please trace back.

26. **ARM Mac and Intel people who did it the right way the first time ☺ with EFI firmware in their FreeBSD VMs:** Create an EFI system partition. EFI firmware requires a dedicated partition for boot

loaders, hence the label "`gpefiboot`" below.

```
# gpart add -t efi -l gpefiboot -a 4K -s 256M da0
```

People with Intel CPUs and legacy BIOS FreeBSD VMs who missed the EFI ask ☹ in lab 2: Let's create a boot partition …

```
# gpart add -t freebsd-boot -l gpboot -a 4K -s 492K da0
```

… you can ignore the warning about the partition not being aligned properly. Note: we are creating a BIOS boot partition here. BIOS firmware can boot GPT disks, and we're using that to our advantage here to keep the direction simple for those of you who were asleep at the wheel in lab 2. **WAKE UP!**

No matter which command you ran above, you should get an affirmative reply from `gpart`.

27. Next, let's create a partition for a FreeBSD filesystem, with the label for a root filesystem …

```
# gpart add -t freebsd-ufs -l gprootfs -s 15G da0
```

… I used 9G here for a 15 gigabyte (gibibyte?) filesystem, because I used a 16 GB replica virtual storage device. Whatever you picked for the size of your replica's storage device, save 1 GB or so for virtual memory.

It should reply `da0p2 added`.

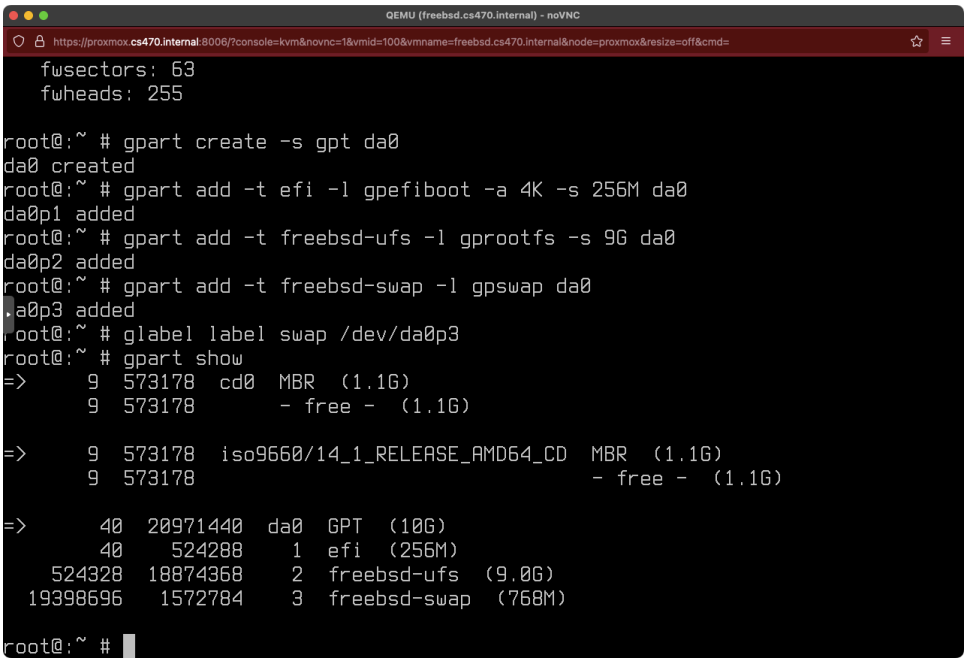
28. Now, add a swap partition …

```
# gpart add -t freebsd-swap -l gpswap da0
```

… this command should reply `da0p3 added`. Note we did not include a size, which implicitly tells `gpart`'s `add` verb that this partition can use the remainder of available space on the storage device. Let's also "label" the swap area as swap. This command returns nothing on success …

```
# glabel label swap /dev/da0p3
```

`gpart show` should help you validate your work.



29. Use `ifconfig` to determine the name of your network interface device, and note that it might be different from what you had when your VM was on Fusion or Workstation … mine was `em0` in Fusion, and an Intel E1000 is an `em` too. (Remember: `lo0` is your "loopback" interface, a virtual interface.)

    Whatever it is, let's configure your network now …

    ```
    # ifconfig em0 up 10.42.77.72 netmask 255.255.255.0
    ```

    … and you should now be able to ping your other VMs by IP address, but not by name … see `/etc/resolv.conf` … mine is a link to a file under `/tmp` (because `/etc` is on a read-only filesystem, an optical disk). Your resolver is not configured, but it could be if you really wanted.

30. Next, let's create a new filesystem within the partition set aside for the root filesystem, the one with the bulk of the space. Note that all we've done thus far is create the various abstractions, boot code partitions, and define what is where within your virtual machine's virtual disk. Now, we're going to start re-creating content.

    ```
    # newfs -L rootfs -U /dev/da0p2
    ```

    Depending on the size of the filesystem, `newfs` can generate a lot of output … as it creates the filesystem, it uses an algorithm to calculate where backups of the all-important filesystem "superblock" should go, in case a drive problem kills one, or a few … we'd still really, really like to get our data back. Once upon a time, I used to save the output of this command to an offline text file, because I **have** indeed been there with a dead disk, hoping a superblock backup would light the way for `fsck` to save **some** data.

    Mount the new root filesystem on `/mnt` …

```
# mount /dev/da0p2 /mnt
```

Now, `cd` into the new root filesystem …

```
# cd /mnt
```

… make a temporary directory for `restore` to use for temporary space (`/tmp` is the default and will run out way, way too fast) …

```
# mkdir /mnt/restore
```

… configure that directory in the environment so that restore knows where to put its temporary files …

```
# TMPDIR=/mnt/restore
# export TMPDIR
```

… and then pull down the backup of the lab two VM from the lab three NFS server, shared via the web server in the `nginx` container from lab four.

‼ IMPORTANT NOTE: you'll need to modify the below single-line command for it to run, notably your IP and backup file name, if yours is different. Once you get it working properly, it will take a little while to run. Let it.

```
# fetch -q -o - https://10.42.77.74/nfs/freebsd/da0p2.dump.gz | gunzip -c | restore
rvf - | tee /mnt/restore/restore.out
```

`tee` is like a plumbing connector; output from `restore` will still go to console, but will also go to the file `/mnt/restore/restore.out` … when you see the final line of output, "checkpointing the restore," it's done.

Predictably, **this command should produce a certificate error**, because we're accessing our Ubuntu VM's secure web server by its IP address. What's in its certificate? Its hostname. By adding `--no-verify-hostname` after `fetch` we remove complaints about the hostname not matching, but still it complains about not trusting where our web server's certificate came from.

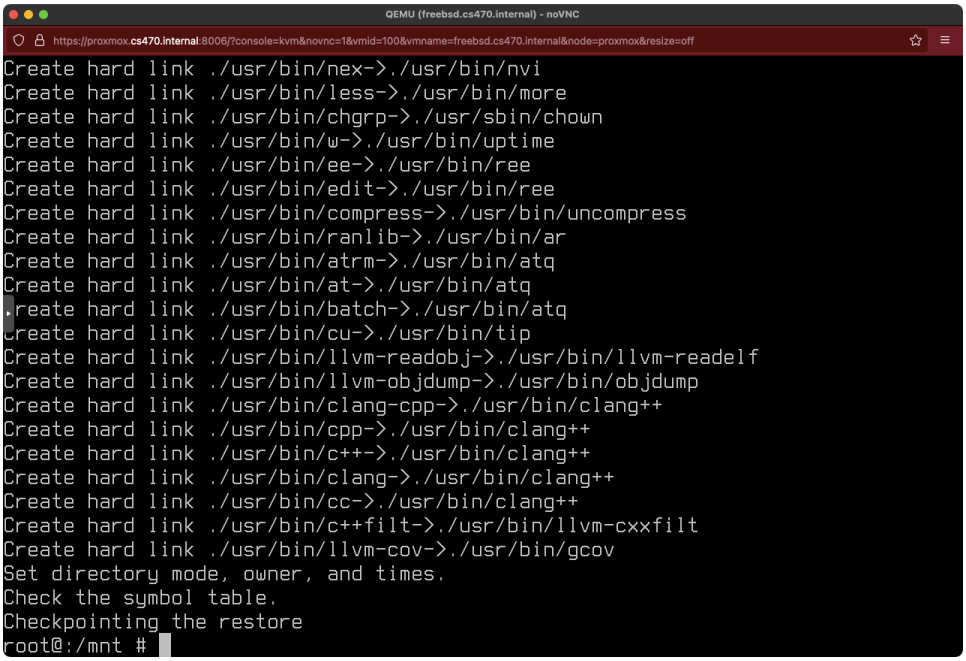
By adding `--no-verify-peer`, we fix this as well, but we **still** get an error … either "forbidden" or "not found." Of course you did ... because the user `nginx` is running as on our Ubuntu VM isn't in the nobody group, either. Let's **not** add it there, though ... go back over to your Rocky VM and run the following command to add world-execute permissions to the folder we're fetching this backup from ...

```
$ sudo chmod +x /srv/nfs/freebsd
```

... read permissions are required to see the listing of a directory ...

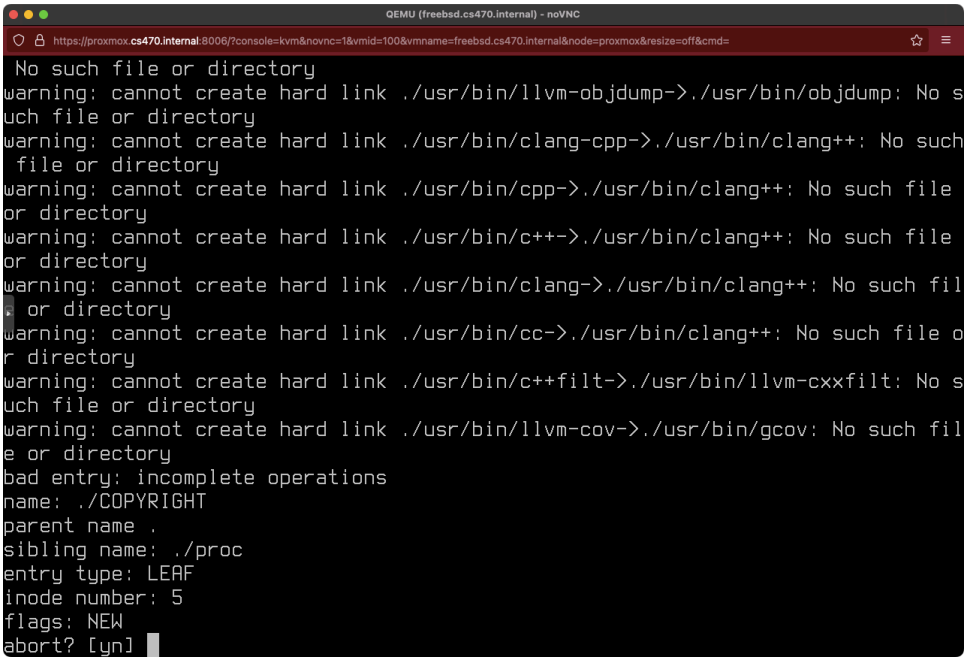
```
$ ls -ld /srv/nfs/freebsd/
drwxrwx--x. 2 nobody nobody 4096 Apr 17 23:51 /srv/nfs/freebsd/
```

... but now that everybody can *execute* the directory, they're allowed to access files in it **if and only if they know the name of the file they're trying to access.** Go back to your new FreeBSD VM's console, and try that `fetch` command above again. It should pause briefly, then explode an output of files it's copying from the backup … `restore` is running. It might take a while. It's way slower on Proxmox than it was on VMware. When it's done, you should see something like this if it was successful ...



... and this is what it looked like for me the last time it actually worked through until the end, for the first time in a few years, on the first attempt. This worked reliably for several years, from the time I wrote this lab until a diagnosis-defying bug was introduced somewhere in the stack around the time of FreeBSD 13.0's release. On each of my first three times running `restore` in 2022, it failed on me. I was hoping it would be fixed with FreeBSD 14, but apparently, it hasn't. It has repeatedly failed on me ever since, both with VMware ESXi as the hypervisor and with Proxmox … so I've eliminated the hypervisor, but still not pinned down the root cause … it could be VMware, Rocky Linux, `nginx`, NFS, or FreeBSD. So I've worked around it … see below.

The below screenshot is of a hard failure of `restore` ... if you have problems, I'm leaving in troubleshooting information from prior years here below. If this happens to you, key in `y` until it's done asking you questions to tell it you want to abort, and read below.



!! YOU SHOULD ONLY DO THE REST OF THIS STEP IF `restore` FAILS … AND PLEASE E-MAIL ME TOO. IF THIS AFFECTS YOU, I WANT TO KNOW. IF THIS AFFECTS ANY PEOPLE ON ARM MACS AND UTM, I'M REALLY CURIOUS … I HAVE NO RECORD OF ANYBODY EVER RUNNING INTO THIS PROBLEM ON ARM64.

If `restore` fails on you, you can try a couple things to solve it. I've had both work. Sometimes.

The first: try again. Maybe a few times. Go back and unmount, re-initialize the new root filesystem, mount it, create the temporary directory, and run `restore` again. I don't make this suggestion idly, to just try it again. The practice unmounting, re-mounting, and re-initializing the filesystem of the FreeBSD replica will be worth it if this affects you.

In order to try again, `cd` to the root of the file tree to get out from under `/mnt`, then `umount /mnt`, run the `newfs` command again, remount the filesystem, re-make `/mnt/restore`, `cd /mnt`, and then run the `fetch`/`restore` pipeline again. I recommend unmounting and wiping the filesystem with `newfs` after each failure.

The second method: read the backup via NFS instead of HTTPS. Make a new mount point under `/tmp` and mount `/home` from your Rocky VM on it. Make sure to appropriately adjust the path of your backup file when restoring, something like …

```
# mkdir /tmp/nfs && mount 10.42.77.73:/srv/nfs /tmp/nfs
# cd /mnt && gunzip -c /tmp/nfs/freebsd/da0p2.dump.gz | restore rvf - | tee
/mnt/restore/restore.out
```

… this typically works for me. If neither of these methods work for you, you need to let me know ASAP.

31. If your network device name changed, fix the network configuration portion of `/etc/rc.conf` … or, as it's mounted right now, `/mnt/etc/rc.conf` … if so, this should just be changing the name of the

variable `ifconfig_em0` or the variable name on whatever line has your FreeBSD VM's IP configuration.

32. If your storage device name changed, edit `/etc/fstab` (under `/mnt` of course) and make sure each device file reference in that file reflects your new storage device name, or you'll be booting into single-user mode later to fix it.

33. **This step is only for ARM Mac and Intel people with EFI firmware in their FreeBSD VMs.**

    Next, let's make the drive bootable by adding boot code to the EFI system partition. First, let's lay down a filesystem. By standard, the filesystem used for the EFI system partition is FAT (the old MS-DOS filesystem).

    ```
    # newfs_msdos /dev/da0p1
    ```

    Mount it.

    ```
    # mount -t msdos /dev/da0p1 /mnt/boot/efi
    ```

    Grab the backup of the EFI system partition and unpack it.

    ```
    # cd /mnt && fetch --no-verify-hostname --no-verify-peer -q -o -
    https://10.42.77.74/nfs/freebsd/da0p1.tar.gz | tar zxvf -
    ```

    You should see a few files unpack.

34. **This step is only for ARM Mac and Intel people with EFI firmware in their FreeBSD VMs.**

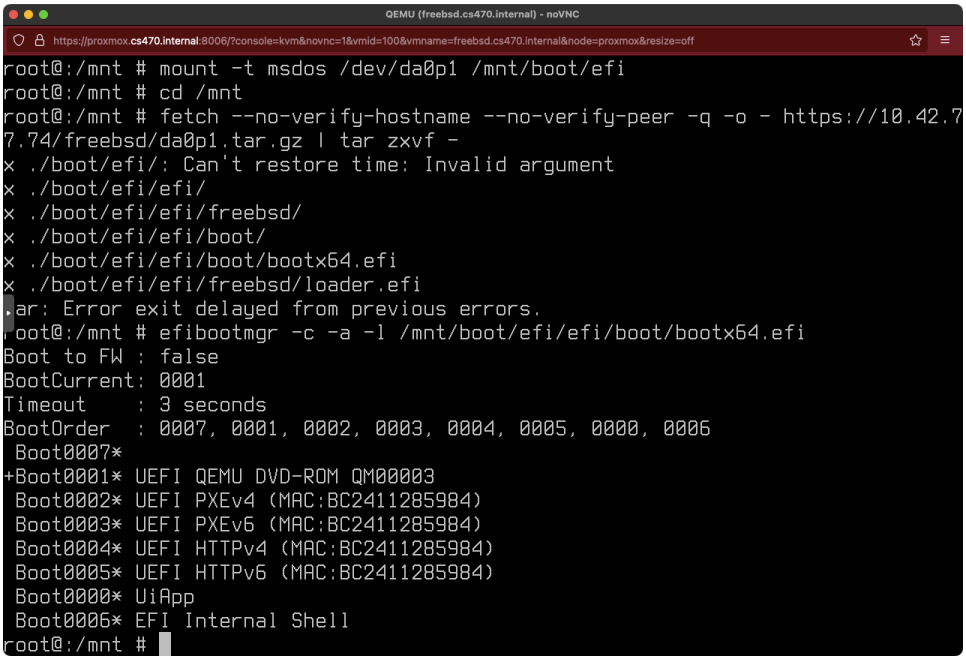
    We need to tell the VM's EFI firmware that there's a new boot option …

    ```
    # efibootmgr -c -a -l /mnt/boot/efi/efi/boot/bootx64.efi
    ```

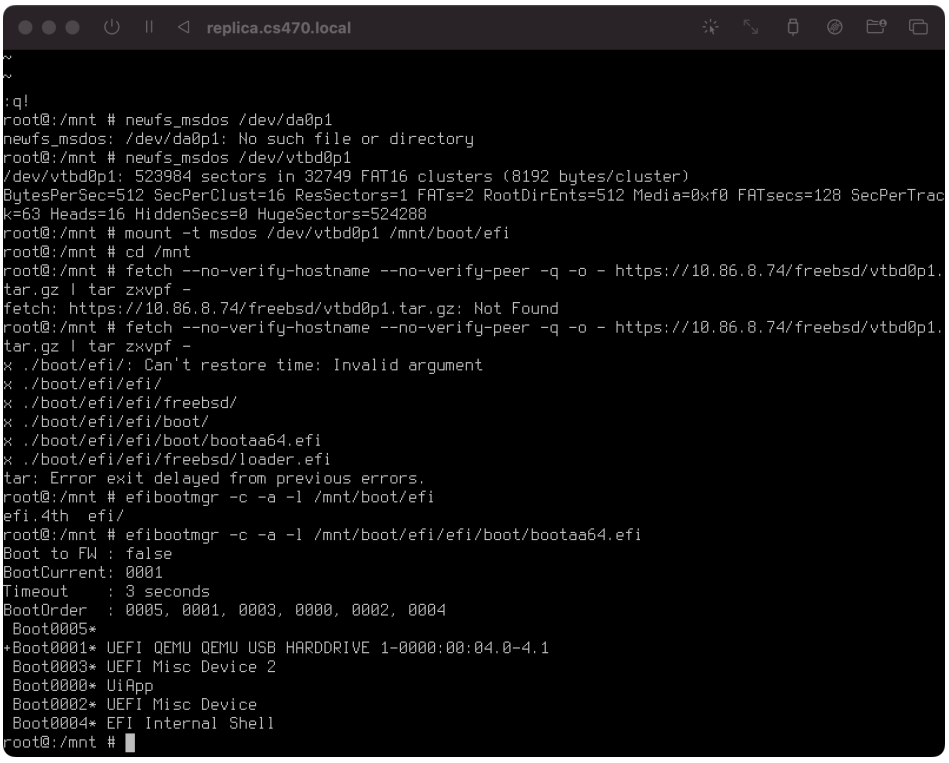
    … you should see that you created a new boot method that supersedes the current one.

    If you're on an ARM Mac, clearly `bootx64.efi` doesn't apply (`x64` refers to 64-bit Intel). It should be obvious which filename you should supply instead from the files you extracted in the prior step.

    Here's a screenshot from an Intel VM.



Here's a screenshot from an ARM VM from last semester, when we used cs470.local.



35. **This step is only for people with Intel CPUs and legacy BIOS FreeBSD VMs.**

Let's make the drive bootable by adding boot code to it.

```
# gpart bootcode -b /boot/pmbr -p /boot/gptboot -i 1 da0
```

36. Shut down your replica VM.

    ```
    # shutdown -p now
    ```

    In either the Proxmox web UI or UTM's interface, make sure the optical disk ISO is disconnected from the VM.

37. Start your new FreeBSD VM back up. It should boot up, be on the network, and you should be able to SSH into it too without key warnings. For all intents and purposes, it's the same machine. Mission accomplished‼

    People on Intel CPUs only: This one's a web search exercise. Configure your FreeBSD VM to start up when your Proxmox host does. Hint: it's in the Proxmox web-based UI. I know, not much of a hint.

38. Once you have your FreeBSD VM booted back up, open up your backup under `/home/tmp` with the command …

    ```
    $ gunzip -c /srv/nfs/freebsd/da0p2.dump.gz | restore ivf -
    ```

    … and when you get a prompt, enter `?` for help. Note that you have a small set of familiar commands you can use here, inside `restore`, to interactively browse through files created by `dump`, and select just the files you want to restore. You don't need to restore anything here … I just wanted you to see this interactive mode.

    ‼ DO NOT DELETE YOUR BACKUPS … WE WILL USE THEM FOR GRADING ‼

part six: saving backup and replication artifacts for grading

That's it … you've done a backup, and a restore, a mock physical-to-virtual conversion, and gotten experience with a data center hypervisor. If you have an Intel CPU, you're also running a VM inside another VM. Pretty cool. Go ahead, play around with the nested FreeBSD VM you just created. It works, everything works.

Also, from the pretty cool column, note what you did here. You backed up your lab 2 VM to a file server on your lab 3 VM, and downloaded it from your lab 4 VM to create a replica inside your lab 6 VM. Your lab 1 VM provided DNS, identity, and encryption for data security, of course. The reason we opted to use the FreeBSD VM here, is because it and your OpenIndiana VMs were the only bystanders here.

We've built on top of each lab with each subsequent lab, and it's all coming together now.

Please run the following commands on your FreeBSD clone to save artifacts for grading.

```
$ df -k > /tmp/df.lab6.txt
$ df -h >> /tmp/df.lab6.txt
$ sudo gzip /restore/restore.out
$ sudo cp -p /restoresymtable /restore/restore.out.gz /tmp/df.lab6.txt /srv/nfs/freebsd
```

Make sure all your lab 6 artifacts are readable for grading later on. On your Rocky Linux VM, run …

```
$ sudo chmod 644 /srv/nfs/freebsd/*
```

When you're done playing with your FreeBSD replica, you can shut it down and power off your whole lab 6 VM, Proxmox and all. Don't throw them away yet, though, just in case, but we won't be using it. Proxmox eats up too much RAM on top of our FreeBSD instance, so we'll go back to using your lab 2 VM to provide mail service for our lab networks.

Windows users, please feel free to re-enable any features you disabled at the start of the lab; so long as you have the artifacts above, you won't need your Proxmox VM for grading later.

Please do, however, feel free to experiment if you have the RAM and space to do so!

```
</lab6>
```