Unix for Developers
Lab 1:
Installing and Configuring a Unix System

Benedict Reuschling

February 9, 2019

Contents

1 Objective: Installing a Unix System

In this lab, we will install and perform the basic configuration of a Unix system. There are a lot of Unix systems out there, but the basics for installing them are essentially the same. While there is usually an official installer that can be used to install the system, we will use terminal commands to get to know what is going on behind the scenes. At the end of this lab, we will have a basic Unix system running that has everything we need for future labs.

Note: We will use this installation in future lab exercises, so make sure that you save your work!

2 First Steps

Download a virtualization software (so called "hypervisor") such as VirtualBox[1] and install it on your computer. With that, you can run different operating systems (such as Unix) on top of your current operating system. Make sure that you can emulate and run 64-bit operating systems with it (if not, team up with someone that can). If you want to use a different hypervisor, then make sure that you know how to use it. Make sure that you have enough disk space available as we'll need at least 15 GB for the virtual disk. Also ensure that you can at least assign 1 GB of memory to the virtual machine (more is better).

Download the ISO-Image of the operating system that we are going to install. To do that, point your browser to the following URL:

`http://www.freebsd.org`

Click on the "Download FreeBSD" button and on the next page, select from the 12.0-RELEASE section, the column labeled **Installer Images**, the first one called *amd64*. This is the target architecture we are using in VirtualBox (64 bit). From the FTP listing that follows, select the one called `FreeBSD-12.0-RELEASE-amd64-disc1.iso`. Make sure to download the ISO-Image at home and not at the start of the lab as it can take a long time, especially over WiFi! You can also chose a mirror closer to you by changing the FTP URL to `ftp://ftp.de.freebsd.org/...` or `ftp://ftp2.de.freebsd.org/...`.

Once you have downloaded the ISO image, you are ready to begin with the installation. Note that this will install a system without a graphical desktop. Such an environment can be installed and configured afterwards, but requires more resources from the virtual machine. If you want to do that, take a look at TrueOS[2] (formerly known as PC-BSD).

3 Preparing the Virtual Machine

Create a new virtual machine in VirtualBox using the **New** button from the main menu. Enter *FreeBSD* as the name of the machine and make sure that **type** and **version** are *BSD* and *FreeBSD (64-bit)*, respectively.

On the next screen, chose the amount of memory you want to dedicate to the machine (1 GB is recommended, more is better).

[1] `https://www.virtualbox.org`, available for Windows, Linux, BSD, and Mac OS X systems.

[2] `http://trueos.org/`

Next, create a virtual disk that is between 15 GB and 20 GB in size (you can go with the VirtualBox recommendation here). The format of the virtual machine disk does not matter much to what we do with it, selecting any of the options is fine. On the next screen, make sure to select **fixed size**. The next screen allows you to name that virtual disk and select its size, but the defaults are fine. After clicking the **Create** button, VirtualBox will allocate the disk space for the VM, which can take some time. Afterwards, the new machine is listed in the VirtualBox main menu.

Before we launch the virtual machine for the first time to begin the installation, we need to make some configuration settings. To do that, select the virtual machine and click on the **Edit** button. On the *system control* page, **uncheck the floppy**[3] from the boot order. Also, select the **ICH9** chipset and check the box next to **hardware clock in UTC**.

In the **Display** tab, select **VMSVGA** as the graphics controller and check the box next to **3D-Acceleration**.

In the **mass storage** page, click on the disc icon, then click on the disc icon on the right side to browse to the downloaded `FreeBSD-12.0-RELEASE--amd64-disc1.iso` on your hard drive. This will enable the virtual machine to boot from that ISO to begin the installation.

Next, go to the *networking* page and activate at least one adapter. Pick **NAT** or **bridged** depending on what kind of network you are currently connected to (home vs. university network). You can switch between the two options when the machine is running later without having to stop and restart it first. Pick one of the Intel network cards as the adapter type.

That's all that is needed for now. Click **OK** when you're done with the settings to return to VirtualBox' main menu. We are now ready to start the installation.

4 Installing the System

Select the virtual machine and hit the **Start** button to begin booting from the downloaded ISO image. First, the FreeBSD boot menu is presented which offers a couple of options to control how the machine will start. But since this is the first time we boot, we are not interested in these options. Either let the countdown reach zero or hit enter to boot the kernel. The kernel (white messages on the screen) will go through various routines to detect the (virtual) hardware on the system. It will enumerate devices and assign names to it, followed by a number starting from zero if there are multiple devices of

[3] Do you even know what this is? ;-)

the same type. After the kernel has finished booting, it will hand over control to the userland part of the boot process. These messages are displayed in grey color. The system will then automatically start the bsdinstall installer and present the first screen, see fig. 1.

Figure 1: FreeBSD's first install screen

Select the middle option (**Shell**) using the cursor keys and press enter. You will be dropped into a shell prompt from where we will perform the rest of the installation. The system provides a number of shells[4] and by default, `/bin/sh` is used as it is available on all Unix systems. To get a few more comfortable options, we can chose to use `csh` or `tcsh`, so that we can get command history and tab completion. Just enter the name of the shell you want to use.

If you want to change the keyboard layout from the english default to another one, enter `kbdmap` (short for *keyboard map*) and select the one you want to use from the list. If you already know which layout to use, you can enter something like this: `kbdcontrol -l de` (the `-` can be found on the ?-key on a standard german qwertz keyboard). Test your keyboard so that the z-key is correct and that you can generate characters like these: `|`, `{` and `}`, as well as `[` and `]`.

[4]The bash shell is not part of the base system and must be installed separately.

4.1 Partitioning the Virtual Disk

In this section, we will take care of preparing the virtual hard disk for the operating system installation so that the system can boot from it without the need for an ISO image. Follow these steps to prepare the disk.

1. First of all, we need to figure out what device name the kernel has assigned to our virtual hard disk during boot. FreeBSD uses a system called CAM (common access method) for all kinds of storage devices, regardless of whether they are IDE, SATA, or SCSI devices. That way, it abstracts away common device functions and presents the user with a unified interface without having to deal with device specific ways of storing and accessing data (see cam(4) for more information). To list all detected devices on the bus, use `camcontrol devlist`.

   This will list two devices: one is the virtual CD-ROM drive that our ISO-file is connected to and the other one is the harddrive we are looking for. Next to the device name (`VBOX HARDDISK 1.0` in our case) is the device node the kernel has generated for that device, something like `da0` or `ada0`. Use `ls` on the `/dev` directory to see whether such a device node was created.

2. Now that we know which disk we are working with, it is time to partition it. FreeBSD uses a system called GEOM (short for geometry) to do all kinds of disk transformations with devices. First of all, we destroy all previous partitions on the disk. This is usually not needed when starting on a new disk, but when you are restarting the installation for some reason, it is better to remove any previous partitions.

   Warning: If you are doing this step, make sure you are doing it in a virtual machine. If you are issuing this command on a real disk that shares the partitions with another operating system, you can destroy the data stored on those partitions as well. Make sure you have proper backups!

   The command for partitioning geom disks is called `gpart` and has a number of subcommands. In this case, we need the `destroy` command. If there are any active partitions on the disk, the system will warn about destroying these. The parameter `-F` forces the deletion. Lastly, provide the device name to the command.

   ```
   gpart destroy -F <devicename>
   ```

3. We can now partition the disk with a new partition scheme. We will be using the modern **GPT** (Globally Unique ID Partition Table) which

supports modern disks on Intel-based architectures. The subcommand to use is *create* and the partition scheme is provided with the `-s` option to the target disk.

```
gpart create -s gpt <devicename>
```

A message will be issued to the console stating that the partition was created. To display the partition table as it is now, use `gpart show`.

4. We have created a basic partition now. Next, we must make sure that the computer knows how to start the operating system from that disk. Hence, we need to install a boot loader into the so called master boot record (MBR) of the disk. This is where the system is looking for startup information and if these are missing, the system will stop booting. There are two types of boot systems now: BIOS-based and (U)EFI-based systems. UEFI is still new, but it is expected to replace BIOS-based booting in the long term, so we'll use that one to familiarize ourselves with it.

   First, we add the efi boot partition. It does not require much disk space. We create a label (called efi, you can name it whatever you want) so that we can easily identify it later in the output of `gpart show`. We also align it to a sector size of 4k for optimal performance, even if your underlying physical disk is using 512 bytes internally.

   ```
   gpart add -t efi -l efi -s 800k -a 4k <device>
   ```

   Next, we add the traditional boot sector of type `freebsd-boot`, which is a small `512k` partition to hold the bootcode. We create a label so that we can easily identify it later in the `gpart show` output and distinguish it from the efi boot partition. We again align it to 4k sectors for optimal performance. The command is similar to the one above, except for the type (`-t` option) and the partition size.

   ```
   gpart add -t freebsd-boot -l bios -s 512k -a 4k <device>
   ```

   When running `gpart show`, you can see the two new partitions we just added and their labels (when providing the `-l` option).

5. We will now fill the disk with other partitions like swap and a ZFS partition to hold the bulk of our data for the operating system. A swap partition is used when the system is low on memory and needs to move memory pages to disk or back again. It should have at least the size of main memory (1 GB in our case). The type is `freebsd-swap` and we label it with `swap` so that we can identify it among the other

partitions. The rest of the parameters are familiar from the previous `gpart add` executions.

```
gpart add -t freebsd-swap -l swap -s 1g -a 4k <device>
```

Again, we check our results with `gpart show -l <device>`.

6. After the swap partition was successfully added, it is time to create the `freebsd-zfs` partition to fill the rest of the disk. As the label, we want to use the serial number of the disk so that we can uniquely identify it in case we add more disks in the future. To display the device serial number among other capabilities of the virtual hard disk, we use the `camcontrol` command with the `identify` subcommand on the disk. As the listing is quite long, even for a virtual disk, we pipe it to a pager (`less`) so we can scroll back and forth.

   ```
   camcontrol identify <device>|less
   ```

   Now that we have found the disk serial number, we can use it in our label command of `gpart add`. Note that we do not provide a size (`-s` option) this time to tell `gpart` to use the rest of the available disk space for this partition.

   ```
   gpart add -t freebsd-zfs -l <serial-number> -a 4k <device>
   ```

   In item 4 on the previous page we have created the boot partitions. By themselves, they can not do much without actual boot code in them. We will now tell the system that it should look for a ZFS-specific boot block on both of these partitions. We take the bootcode from the live-system we booted from the ISO and write it to these boot partitions. To identify which partition the `bootcode` should be written to, we provide the ID from the output of `gpart show -l`. EFI has the ID 1, because it was added as the first partition.

   ```
   gpart bootcode -p /boot/boot1.efifat -i 1 <device>
   ```

   Similarly, we write the legacy bootcode into the protected master boot record (`pmbr`) for the partition labeled `bios`, which is the second one we added.

   ```
   gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 2
   <device>
   ```

   We are done now partitioning the disk and can continue with the ZFS part of the installation.

4.2 ZFS Setup and Configuration

We will be using the ZFS filesystem and its features for our system. ZFS combines hard disks into a so called pool that provides disk space to ZFS filesystems that sit on top of it. Each filesystem (called a dataset in ZFS terms) can use the full storage provided by the pool and can set individual options on them. We will create the necessary datasets for the operating system files and directories[5].

1. Load the ZFS kernel module using the `kldload zfs` command. A white kernel message will appear if you have allocated less than 4 GB of memory for the virtual machine. ZFS expects to use that much RAM, but for our purposes, it should run fine if you do not have that much memory to give to the virtual machine. Check that the module was properly loaded by listing all current kernel modules with `kldstat`[6]. The `opensolaris.ko` module was automatically loaded as a dependency.

   We also have to set the minimum `ashift` value so that the pool will use a sector size of 4k (2^{12}), the same we have set in our partitions.

   ```
   sysctl vfs.zfs.min_auto_ashift=12
   ```

2. We will now create the pool that will abstract the underlying disks away so that we do not have to care anymore which disk each file is stored on. Instead, we just refer to the name of the pool which forms the root of the directory structure we are about to create. The `zpool` command is used with the subcommand `create` to initialize a new pool on our hard disk device. We are providing an alternative root directory for now, because we do not want it to interfere with the current ISO-based system we are in right now. We specify two options, `compression` and `atime`.

   With compression, all data stored on the pool is automatically compressed when written and uncompressed when read. This does not only save significant amounts of disk space, but also makes I/O faster for data that can be compressed well enough.

   The `atime` property stores the last access time of each file, which is rarely used nowadays and results in an extra I/O each time the file is accessed, so we deactivate it. We also do not mount the pool right now, we will do that in a later step. You can chose your own name for the

[5] See `hier(7)` for a description of the filesystem layout

[6] `kldstat` can output human-readable sizes using `-h` if hex isn't your thing. ;-)

pool, but note that you have to enter it a couple of times, so choose a short one if you want to safe yourself some typing.

Take a look at the output of `gpart show`. We want to create our pool on the partition of the type `freebsd-zfs`. As we know by now, each partition has been assigned a unique name under `/dev`, in the form `<devicename>p<ID>`. For example, the swap partition was added as the third partition, so it has the device node entry `/dev/ada0p3`. Accordingly, we can now specify the `freebsd-zfs` partition for the `zpool create` command.

```
zpool create -o altroot=/mnt -O compress=lz4 -O atime=off
-m none -f <poolname> <partition>
```

Note: This is not a very redundant pool because it is based on only one disk. For our purposes, it is enough, but for a real production system, more disks should be part of the pool in a redundant configuration.

After the pool has been created, you can check the status with `zpool status` and `zpool list`.

3. We will now create datasets on the pool to hold the operating system directory structure. Note that ZFS automatically inherits properties (like `compression` and `atime`) set on the pool level down to the datasets. When there is a dataset that needs special options, we need to specify them with the dataset to override the inherited settings.

   First, we create the root (`/`) directory as a boot environment. Boot environments are a feature of ZFS to allow the user to switch between different root directories. This is useful for testing different system configurations, to recover from a failed update of system software and other situations that would normally require a complete reinstall or restore from backups. We will use them later, so we just take note that `/` is stored on `<poolname>/ROOT/default` for now. Other datasets we create below it that have the property `canmount=off` are also part of the boot environment.

   ```
   zfs create -o mountpoint=none <poolname>/ROOT
   zfs create -o mountpoint=/ <poolname>/ROOT/default
   ```

   Next, we create the `/tmp` and `/usr` directories. We allow users to execute programs on them, but disallow those with the setuid bit set for security reasons.

   ```
   zfs create -o mountpoint=/tmp -o exec=on -o setuid=off
   <poolname>/tmp
   zfs create -o mountpoint=/usr -o canmount=off <poolname>/usr
   ```

The /usr/obj directory is used for storing temporary files when building the base system and kernel from the source code. This is really just a scratch space area that stores a lot of compilation results (objects, binaries).

```
zfs create -o mountpoint=/usr/obj <poolname>/usr/obj
```

This creates the home directory for all users (i.e. you) that will use the system later. `zfs create <poolname>/usr/home`

In /usr/ports, all makefiles to build and install third-party applications are stored in a directory structure to easily find them. We will later create the directory structure that will form the ports tree. For example, the vim text editor is stored in `editors/vim` below /usr/ports.

```
zfs create -o setuid=off <poolname>/usr/ports
```

All source files to build the operating system (kernel and userland applications) are stored in /usr/src. We will later check out the current sources for the operating system into that directory.

```
zfs create <poolname>/usr/src
```

The /var dataset stores various files like logs and print spool directories. It is also part of the boot environment as it also holds some important databases the system relies on. `zfs create -o mountpoint=/var -o canmount=off <poolname>/var`

The next two directories below /var store crash dumps (if configured) and the system logs, respectively. `zfs create -o exec=off -o setuid=off <poolname>/var/crash`
`zfs create -o exec=off -o setuid=off <poolname>/var/log`

Local system mail stored in /var/mail uses the access time property, so we switch it on for this particular dataset only. `zfs create -o atime=on <poolname>/var/mail`

Lastly, we create /var/tmp, which is used by some applications to store temporary data instead of /tmp. `zfs create -o setuid=off <poolname>/var/tmp`

Now we only have to set the `bootfs` property to tell the system from which of the datasets to boot from. This is a specialty from the boot environments and with that property, we can easily switch between multiple different environments if needed.

```
zpool set bootfs=<poolname>/ROOT/default <poolname>
```

After you've created all the datasets, you can list them with `zfs list`. You can see where they are mounted and that each can use the total

amount of disk space that the pool provides based on the disk space capacity.

Note: If you made a mistake with one of the datasets, you can destroy them with `zfs destroy <poolname>/datasetname` and re-create them.

4.3 Installing the Operating System Files

In this part of the installation, we will extract the files and directories that make up the operating system into the datasets we created earlier. This (along with partitioning and disk setup according to the users inputs) is what a typical installer does, too.

1. The files are available as xz-compressed tarballs that simply need to be extracted to the destination. We specify our `ALTROOT` destination to the `-C` parameter as this is where the pool is currently mounted. The base system is extracted onto the ZFS datasets.

   ```
   tar --unlink -xvpJf /usr/freebsd-dist/base.txz -C /mnt
   ```

2. Next, we extract the kernel, which is a separate component stored in `kernel.txz`. The kernel contains debugging symbols for each module. This takes up additional disk space, so we exclude them from the extraction.

   ```
   tar --unlink -xpvJf /usr/freebsd-dist/kernel.txz -C /mnt
   --exclude '*.symbols'
   ```

3. Next, we ensure that `/var/tmp` has the proper permissions needed after the extract is over.

   ```
   chmod 1777 /mnt/var/tmp
   ```

4. The swap partition needs to be listed in `/etc/fstab`, so that it can be activated upon boot. ZFS does not require `/etc/fstab` entries, as all information about partitions and mountpoints are stored within the pool itself and managed there. First, we create the `/etc/fstab` file:

   ```
   touch /mnt/etc/fstab
   ```

   Then, we can check the `gpart show` output to see which device node was assigned to the `freebsd-swap` entry. Then, we use the `swapon` tool to activate the swap partition.

   ```
   swapon /dev/<swappartition>
   ```

   We can check the output of `swapinfo` to see whether the swap partition was successfully added.

To automatically add the swap partition each time the system boots, we create an entry in `/mnt/etc/fstab` (using the `ee` editor on it) that looks like this:

```
/dev/ada0p3   none    swap    sw    0    0
```

4.4 Configuring the Operating System

We need to make a few modifications within the operating system to be able to boot from it and set some basic options for our first boot.

1. First, we use `chroot` to temporarily change our `/` filesystem to be `/mnt`, where our pool with the newly extracted operating system resided. That way, we are already in the newly created system and can use the same paths as we would have when the new system is active after a reboot.

   ```
   chroot /mnt
   ```

   We can also start a `csh` within that environment to get tab completion and command history.

2. We need to add a few lines to `/boot/loader.conf` so that the system knows that is should boot from ZFS. Use either `ee` to edit the file directly or issue the following command:

   ```
   sysrc -f /boot/loader.conf zfs_load=yes
   ```

3. Another file that holds a lot of configuration data is `/etc/rc.conf`, which is responsible for all managing of third party applications (startup, options) as well as those of the operating system itself. Here, we also specify that we want to boot from ZFS and the datasets we created.

   ```
   sysrc zfs_enable=yes
   ```

   Additionally, ZFS requires the hostid service to run to manage the pool IDs. We also set a hostname for our system here (replace `<myhostname>` with a name of your choice and add `.local` to it). The third line sets the keyboard layout (keymap) to german QWERTZ (only if you have one, otherwise leave this setting alone). Lastly, we enable the OpenSSH server for remote logins.

   ```
   sysrc hostid_enable=yes
   sysrc hostname=<myhostname>
   sysrc keymap=german.iso
   sysrc sshd_enable=yes
   ```

Run `cat` on `/etc/rc.conf` and `/boot/loader.conf` to see whether the settings were added.

4. Next, we want to set up networking. The `ifconfig` commmand shows all currently available adapters on the system, as well as the local loopback device (`lo0`). FreeBSD uses a different name to identify the network adapters than Linux. Instead of `eth`, it is based on the driver and vendor of the NIC[7]. In our case, `ifconfig` should list `em0`, which is the Intel Network driver we selected in the VirtualBox settings earlier. Each such driver has a man page describing the capabilities of the driver, in this case, it is `em(4)`. We assume that we are using DHCP in the network to configure the card and supply an IP address to it automatically. We add this to our configuration in `/etc/rc.conf`.

   ```
   sysrc ifconfig_em0=DHCP
   ```

   Note: We can also configure a static IP address, but for now, we assume DHCP is fine. We'll see after we reboot the machine, depending on the network we are in.

5. We need to set the timezone and system time. When we set up our VirtualBox (section 3 on page 3), we made a setting that defined that our virtual hardware clock is in UTC.

   ```
   tzsetup
   ```

   First, we chose **Yes** on the UTC question and then select the region (*Europe*), *Germany*, and *most locations*. We can check the system time with `date`. If it is still off, we will sync with a time server later once we got the network set up.

6. Add an non-root user to the system. To do that, execute the `adduser` program and follow the prompts. When prompted for adding that user into other groups, enter `wheel operator` to be able to switch to `root` and shutdown the system.

   After you've successfully added the user, change the password for the root user with `passwd root`.

7. You can now reboot into the newly installed system by entering `reboot` or `shutdown -r now`. Make sure to remove the ISO-image from the system (the little CD icon in the upper right corner of the VirtualBox window).

[7] Network Interface Card

5 Getting to know the system

In this part, we'll take a closer look at the installed system and perform some basic tasks. This will help you in future lab exercises.

1. Log in with the non-root user account that you have created in `adduser`. Use the `id` command to see whether you are member of the groups `wheel` and `operator`. Switch to the `root` account using `su` and enter `root`'s password. Switch back to your normal (non-root) account by pressing Ctrl + D.

2. Check that networking works. Use `ping` with some web address to see whether you can reach the other end (use Ctrl + C to end the ping). If it does not work, try switching from NAT networking to bridged networking and vice versa in the VirtualBox settings (you can do that while the machine is running). Use `reboot` to reboot your virtual machine as the `root` user. Check whether you have a gateway assigned to your `em0` NIC by entering `netstat -rn` and checking the `FLAGS` column for an entry containing a `G`. Also check the name resolution by looking at the contents of `/etc/resolv.conf` with `cat`. There should be a `search` and `nameserver` entry there.

3. Synchronize the system clock with a timeserver after you've established network functionality (as `root`).

   ```
   ntpdate de.pool.ntp.org
   ```

   Use `date` to check the new system time. You can use `sysrc` to add a line to `/etc/rc.conf` to synchronize the clock each time the system boots (assuming it has a network connection).

   ```
   sysrc ntpdate_enable="YES"
   ```

4. Setting up the package system. We can bootstrap the FreeBSD package system so that we can download and install pre-packaged software.

   ```
   pkg update
   ```

   The first time this command is run, the package database is initialized from the FreeBSD package repository, which requires network access. After that, packages can be installed.

   ```
   pkg install <packagename>
   ```

   To upgrade packages with a newer version, the `upgrade` subcommand can be used.

   ```
   pkg upgrade
   ```

5. Now that we can install packages from the internet, it's time to create a boot environment we can fall back to in case we have some kind of misconfiguration of the base system. The tool is available in the base system since 12.0-RELEASE and is called `bectl` (Boot Environment Control).

   ```
   bectl list
   ```

   The output shows the name of the boot environment (BE) as well as the mountpoint, used space, and when it was created. We can create a new boot environment with the `create` subcommand.

   ```
   bectl create <name>
   ```

   Listing the boot environments again, we can now see its name in the first column, below the `default` BE. Look at the `Active` column. The **N** means that the BE is currently active (*now*). The `R` indicates that the boot environment will be active upon reboot. We will now activate our newly created BE.

   ```
   bectl activate <name>
   ```

   Once again, check the `bectl list` output. The `R` has moved from the `default` BE to the one you created. Reboot the system to boot into your boot environment.

   ```
   reboot
   ```

   After the rebooted and logging in again, the output of `bectl list` tells us that our own boot environment is now active and will still be active when we reboot. Whenever we do something that could damage the system and its configuration files (specifically, everything in `/`, `/usr`, and `/var`), we can go back to the `default` BE. We can select the boot environment from the loader screen (option 7), in case our current system does not boot anymore. The BE will only store the differences between the `default` and itself, so it does not take up additional disk space and only grows over time depending on how many changes are being made.

   You can also create additional BEs to fall back to (i.e. between system upgrades) or delete old, inactive ones using `bectl destroy <name>`.

Show me the following before leaving:

1. `gpart` output with your custom label (serial no.) for the disk
2. Demonstrate that the network has been set up and works
3. your custom generated Boot Environment in `bectl list`.