CS 264 Disk Image Lab

Spring 2020

1 Intro

In this activity, we are going to write a `bash` script that makes a disk image, partitions it, and writes a filesystem to the partition. A `bash` script is not the same as a Makefile, which I introduced in an earlier video. A Makefile is a special kind of script that is taylor made for compiling programs. `bash` scripts are more general-purpose and are often used for automating complex tasks. In this exercise, we'll use a `bash` script to automatically build a disk image.

1.1 Bash Scripts

`bash` is the name of the shell that we use in the terminal program in Linux. You can run `bash` scripts in Mac OS also but not in Windows. We can write a text file called a *script* with a list of commands that will be executed by `bash`. Writing a `bash` script is kind of like programming but weirder.

The workflow for `bash` script programming is basically typing in commands in the terminal (like you usually do) and then copy-pasting those commands into a text file once they're working.

1.2 Writing Bash Scripts

`bash` scripts have a couple of common components:

1. First line is called a *hashbang* and tells the shell what ineterpreter to use to run the script. Our hashbang will look like the following:

   ```
   #!/bin/bash
   ```

   This is the absolute path to the `bash` executable.

2. Comments: they usually start with a `#` (hash) character.

3. File extension: we will use the `.sh` extension for all BASH scripts. This is not mandatory, but it helps others figure out what kind of file it is.

Your First `bash` Script

```
#!/bin/bash

echo "Hello world!!!"
```

The `echo` command in `bash` is kind of like `print` in Python—it just prints stuff to the screen. To run this script, save the file—you can call it `hello.sh`—and make it executable. In a shell, once you've saved it, run the following command:

```
$ chmod +x hello.sh
$ ./hello.sh
```

Variables in a `bash` Script Just like in other programming languages, we can delcare and use variables in shell scripts. The weird thing about assigning a value to a variable in a shell script is that you **must not put spaces between the variable name, the equals sign, and the variable value**. If you put spaces in the wrong place in a shell script, the interpreter will complain.

```
#!/bin/bash

myname="Neil" # Note no spaces in variable assignment

echo "My name is $myname"
```

Loops in a `bash` Script We often use loops to apply the same operation to a bunch of files. For example, say we want to search for a string in all the C files in the current directory and all its subdirectories. This is a pretty common operation—say you are looking for all the places where a variable or function is used in a complex program, but you don't want to see the occurances of that variable in binary files.

We can use the program `find` to search for all files that match a certain pattern: for example, all files that end with "`.c`":

```
find . -iname '*.c'
```

This command will print out the path of each C file in the current directory (`.`) and all its subdirectories. We can use the program `grep` to search for a string within a file.

```
grep "stage1main" ./src/file.c
```

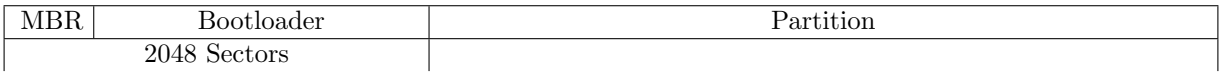
This will search for the string "stage1main" in the file `src/file.c`. We want to compose these two commands to (1) find all files with a `.c` extension and (2) search for a pattern within each individual C file.

```
#!/bin/bash

for f in `find . -iname *.c`
do
    echo $f; # Prints $f, a line output by the find command
    grep "stage1main" $f
done
```

In the above for loop, we run the `find` command inside of backticks (right next to the 1 on the keyboard). Each line that gets printed by the `find` command causes the loop to iterate once. For each loop iteration, we assign one line of the output of the `find` command to the variable `f`. We can then use the variable `f` inside the loop as an argument of the `echo` and `grep` commands.

2 Creating a Disk Image



A disk image is a file that lives on the hard disk and contains a byte-for-byte copy of the contents of some disk. Disk images are used for lots of things. They can be used to reproduce identical copies of some disk, for example if you are trying to mass-produce phones. They are also used by virtual machines to hold the contents of the hard disk.

We have previously talked about how a physical disk can be read/written using a file in the `/dev` filesystem. Usually the root filesystem is located on `/dev/sda1`. The file associated with a physical device in the `/dev` filesystem is not actually present on the disk. It is just a link that is used to get easy access to the hardware device.

Disk images, however, are real files that actually exist on disk. Since images and physical drives both look like files in Linux, we can use the same techniques to read and write them.

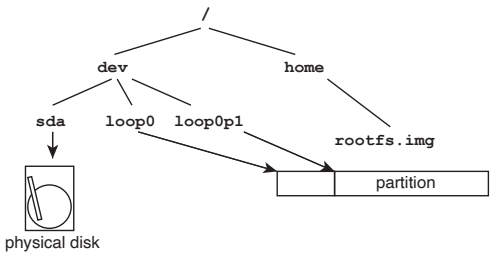


Figure 1: In the device filesystem, `sda` is linked to a physical disk, and `loop0` is linked to a disk image.

To work on this project, I suggest creating a new directory in your Ubuntu VM to hold all the files you are going to create. The deliverable for this project will be one script file that will create a disk image. The sections below will walk you through how to write that script. Basically you will just have to copy/paste the commands below into a script file and turn that in.

In this guide, I will use the `$` to indicate what you are supposed to type in to the terminal. Lines that don't start with at `$` or a `#` are output of a command.

2.1 Step 1: Create a Big Empty File

The first step is to create a file on the hard disk and fill it up with zeros. We can do this using the `dd` command:

```
$ dd if=/dev/zero of=rootfs.img bs=512 count=16k
```

This command will create a file called `rootfs.img` that is 16k × 512 bytes = 8 MiB filled with zeros. To verify its size, do this:

```
$ ls -l rootfs.img
-rw-r--r--  1 neil  staff   8.0M Mar 29 21:25 rootfs.img
```

`rootfs.img` is a binary file, so we can't really open it in a text editor[1]. The right way to view the contents of a binary file is to use the program `hexdump`, which prints out the hex representation of each byte in a file.

```
$ hexdump -C rootfs.img
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00800000
```

This is confirming what we already knew: the file is filled up with zeros. `hexdump` doesn't print every single zero, it just tells us that they repeat up to 8 MiB.

2.2 Step 2: Create a Partition Table

So far, you've created a filesystem that can be written to a disk partition. For this thing to be usable as a hard disk, we need to partition the disk and write the filesystem to the first partition.

Loopback Devices When we mount a filesystem that lives on a disk image (as opposed to a physical disk like a hard drive or USB key), we say that it is mounted in *loopback mode*. When you mounted `rootfs.img`, that mount was done in loopback mode.

You can associate a special file in the `/dev` filesystem with a disk image. This is similar to how we have special files in the `/dev` filesystem that are associated with physical disks. Fig. 1 shows a diagram of how files in the device filesystem are linked to disk images or physical devices.

[1] Actually, we can open it in a text editor, but we won't really see anything useful.

```
$ sudo losetup -v -f disk.img
$ sudo fdisk /dev/loop0
elcome to fdisk (util-linux 2.33.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x1a56f8e6.

Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-32767, default 2048):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-32767, default 32767):

Created a new partition 1 of type 'Linux' and of size 15 MiB.

Command (m for help): p
Disk disk.img: 16 MiB, 16777216 bytes, 32768 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x1a56f8e6

Device     Boot Start   End Sectors Size Id Type
disk.img1        2048 32767   30720  15M 83 Linux

Command (m for help): a
Selected partition 1
The bootable flag on partition 1 is enabled now.

Command (m for help): p
Disk disk.img: 16 MiB, 16777216 bytes, 32768 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x1a56f8e6

Device     Boot Start   End Sectors Size Id Type
disk.img1  *     2048 32767   30720  15M 83 Linux

Command (m for help): w
The partition table has been altered.
Syncing disks.
$ sudo partx -v --add /dev/loop0
```

2.3 Step 3: Write a Filesystem to your Disk Image

Now we will format our empty disk image with a filesystem. In Linux, we can use the `mkfs.XXX` command to initialize a disk image or drive with an empty filesystem. In Mac OS, we use `newfs_XXX` to do the same thing. There are lots of different

filesystems supported under Ubuntu[2]. We will use ext4 because it's the most commonly used one:

```
$ sudo mkfs.ext4 /dev/loop0p1
mke2fs 1.45.5 (07-Jan-2020)
Discarding device blocks: done
Creating filesystem with 8192 1k blocks and 2048 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

2.4 Step 4: Put Some Files on your Filesystem

Now let's put some stuff on our filesystem. First we need to mount the filesystem on your new partition:

```
$ sudo mkdir /mnt/disk
$ sudo mount disk.img /mnt/disk
$ cd /mnt/disk
$ ls
lost+found
```

The first command creates a directory called `/mnt/disk` which we will use as the mountpoint for our disk image. If you already have that directory, you'll get an error message (no big deal, everything else will work). The second command mounts our filesystem onto the directory `/mnt/disk`. In the last two commands, we change directory to the mountpoint and list the files in the newly created disk image. If everything worked correctly, you should have only one directory, `lost+found`.

To see a complete list of all mounted filesystems, type:

```
$ mount
/dev/sda1 on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,nosuid,relatime,size=10240k,nr_inodes=742885,mode=755)
...
/home/neil/disk.img on /mnt/disk type ext4 (rw,relatime)
```

You'll see a printout of a bunch of mounted filesystems. In the list of mounts on my VM, the first one is the rootfs, and the last one is the disk image that we're working on. Most will be pseudo-filesystems like `devtmpfs` mounted on `/dev` which actually don't have any disk sectors associated with them.

You can now set up a rootfs directory tree in here. Create the `/bin`, `/boot`, and `/etc` subdirectories.

2.5 Step 5: Unmount your Disk Image

Once you've set up the directory structure for your filesystem, you should change directory out of `/mnt/disk` because you're about to unmount your disk image, and the OS won't let you do it if you have a terminal open in the directory where it's mounted.

```
$ cd -
$ umount /mnt/disk
$ ls /mnt/disk
```

After you unmount `ls /mnt/disk`, you shouldn't see any of the files or directories you created.

3 What to Turn In

You should copy/paste all these commands into a bash script and submit that file. Make sure your script has a hashbang on the first line, and test your script by running it on your computer.

[2]To see a list, type `mkfs` and push tab three or four times.