

# **The Linux Sysadmins Guide to Virtual Disks**

—————  
**From the basics to the advanced.**

---

Copyright © 2009, 2010, 2011 Tim Bielawa

This work is licensed under the Creative Commons Attribution 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

You can view the latest DocBook 5 XML source to this guide under version control on the web at <http://github.com/tbielawa/Virtual-Disk-Guide> and an HTML compiled version under <http://linox.cx/docs/vdg/>.

---

---

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> The Linux Sysadmins Guide to Virtual Disks		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Tim Bielawa	May 8, 2011	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

---

---

# Contents

<b>1</b>	<b>The Virtual Disk Cookbook</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Query an Image for Information . . . . .	1
1.3	Converting Between RAW and QCOW2 . . . . .	2
1.3.1	Convert an Image from RAW to QCOW2 . . . . .	2
1.3.2	Convert an Image from QCOW2 to RAW . . . . .	3
1.4	Creating Disks with Backing Images . . . . .	3
1.5	Creating Simple Images . . . . .	5
<b>2</b>	<b>Disk Concepts</b>	<b>6</b>
2.1	Introduction . . . . .	6
2.2	Learning by doing (Creating a 1GiB virtual disk from scratch) . . . . .	8
2.2.1	Background on the <b>dd</b> command . . . . .	8
2.2.2	Running <b>dd</b> . . . . .	8
2.2.3	Examining the created file . . . . .	9
2.2.4	Create a Partition Table . . . . .	10
2.2.5	What <b>parted</b> changed . . . . .	10
2.2.6	Creating Partitions . . . . .	11
2.2.6.1	Creating partitions on a loop device. . . . .	11
2.2.6.2	Creating a loop device . . . . .	12
2.2.6.3	Examine the loop device . . . . .	12
2.2.6.4	Creating partitions . . . . .	13
2.2.6.5	Formatting Partitions . . . . .	14
2.2.6.6	Cleaning Up . . . . .	16
<b>3</b>	<b>Advanced Operations</b>	<b>17</b>
3.1	Creating a Mountable Disk . . . . .	17
3.2	Cloning a Physical Disk . . . . .	17
<b>A</b>	<b>Appendices</b>	<b>18</b>
A.1	UNITS . . . . .	18

---

## List of Tables

---

## Introduction

I was motivated to write this document because I felt the quality of the information regarding commonly used functionality in virtual disk operation was lacking certain specific clear examples. The information that is available is not contained in a central location. Some concepts of the qemu system aren't covered at all. FAQs lead on to having an answer to a particular query, but many lead you to off site resources, some of which are no longer available on the Internet.

What I hope to do is provide a document which will demonstrate the core concepts of virtual disk management. This document will concern itself primarily with the **qemu-img** tool and common GNU/Linux disk utility tools like **fdisk**. Most importantly to me, in the case of non-trivial examples, I hope to identify what the relevant technical concepts are how they work up to the final result of each example.

## Chapter 1

# The Virtual Disk Cookbook

### 1.1 Introduction

In this section we're just going to cover things you'll find yourself needing to do from time to time. Theory and concepts will be covered later on. It's assumed that you're comfortable with the concepts already and don't need everything explained.

### 1.2 Query an Image for Information

How to query some basic information about virtual disks. The tools of the trade here are going to be **ls** to check disk usage, **file** for a quick check of the types, and **qemu-img** for more in-depth information.

---

#### Example 1.1 Querying an Image

```
tbielawa@deepfryer:/srv/images$ ls -lhs
total 136K
136K -rw-r-----. 1 tbielawa tbielawa 256K May  8 18:00 image-qcow.qcow2
   0 -rw-r-----. 1 tbielawa tbielawa  10G May  8 18:00 image-raw.raw

tbielawa@deepfryer:/srv/images$ file image-qcow.qcow2 image-raw.raw
image-qcow.qcow2: Qemu Image, Format: Qcow , Version: 2
image-raw.raw:    data

tbielawa@deepfryer:/srv/images$ qemu-img info image-qcow.qcow2
image: image-qcow.qcow2
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 136K
cluster_size: 65536

tbielawa@deepfryer:/srv/images$ qemu-img info image-raw.raw
image: image-raw.raw
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 0
```

---

---

**Note**

These images are freshly created and don't have any information on them yet. Both were created to be 10G images.

---

The interesting information we can get from using `ls -lhs` is how the files are actually sized. `image-raw.raw` appears to be 10G but doesn't have any actual blocks allocated to it yet. It is literally an empty file. What's good about these RAW disks is that you don't need any special kind of tools to know how large the disk is internally. The RAW image should always match it's reported file size on the host OS.

Our QCOW, on the other hand, is being deceptive and concealing it's true size. QCOWs will grow to their maximum size over time. What makes it different from our RAW image in this case is that it already has blocks allocated to it (that information is in the left-most column and comes from the `-s` flag to `ls`). The allocated space is overhead from the meta-data associated with the QCOW image format.

The `file` command tells us immediately what it thinks each file is. This is another simple to perform query that you can run from any system without special tools. Here we see it correctly reports `image-qcow.qcow2`s type. Unfortunately, without any content, all it can tell us about `image-raw.raw` is that it's 'data'.

---

**Note**

Its worth mentioning that RAW image types will be reported by `file` as "x86 boot sector, code offset 0xb8" once given a filesystem label and a partition table.

---

Using the `qemu-img` command we can get more detailed information about the disk images in a clearly presented format.

With `qemu-img` it's clear that `image-qcow.qcow2` is a QCOW2 type image and is only 136K on disk and internally (the 'virtual size' field) is a 10G disk image. If the QCOW had a backing image the path to that file would be shown here as an additional field.

For the RAW image there is no new information here that we didn't already get from the `ls`.

## 1.3 Converting Between RAW and QCOW2

### 1.3.1 Convert an Image from RAW to QCOW2

RAW images, though simple to work with, carry the disadvantage of increased disk usage on the host OS. One option to have is to convert the image into the QCOW2 format which uses zlib compression and optionally allows for 128 bit AES encryption.

---

**Example 1.2** Converting from RAW to QCOW2

---

```
tbielawa@deepfryer:/srv/images$ qemu-img convert -O qcow2 image-raw.raw image-raw-converted.qcow
tbielawa@deepfryer:/srv/images$ qemu-img info image-raw-converted.qcow
image: image-raw-converted.qcow
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 136K
cluster_size: 65536
```

---

It's that simple.

---

---

### 1.3.2 Convert an Image from QCOW2 to RAW

Here's how to do the last example, but in reverse.

---

#### Example 1.3 Converting from QCOW2 to RAW

```
tbielawa@deepfryer:/srv/images$ qemu-img convert -O qcow2 image-raw.raw image-raw-converted.qcow ↵  
image-raw-converted.qcow  
  
tbielawa@deepfryer:/srv/images$ qemu-img info image-raw-converted.qcow  
image: image-raw-converted.qcow  
file format: qcow2  
virtual size: 10G (10737418240 bytes)  
disk size: 136K  
cluster_size: 65536
```

---

It's that simple.

---

#### Note

When converted to the RAW format the image has the potential to take up much more disk space than before.

---

## 1.4 Creating Disks with Backing Images

Using backing images for work I was doing is a trick I figured out a few years ago. Back then I was doing lots of development on a host provisioning tools and needed to be able to quickly revert machines I was working on to a perfect clean state. Backing images were especially handy when working on features that frequently hosed the machine if it didn't work right.

Snapshots were an option, but how they worked wasn't documented very well at the time (Snapshots will be covered in another section). The technical concepts and practical skills describe in this section mostly applies to Snapshots too.

So I went with backing images, they worked great for what I needed to them do. I could work incrementally and "commit" changes in the COW image (copy-on-write) back to the master. I also could use the same base-image for multiple COWs at once which meant other people on my team could all be using the same base-image if we were working on the same thing.

---

#### Note

The terms `base-image` and `backing-image` are used interchangeably to refer to the read-only image that is used to back a read-write disk.

---

---

#### Note

When we refer to a COW image it has no connotation to the COW ("copy-on-write") disk format. Saying COW only serves to help make a distinction between the read-only base-image and the image that changes are copied to on writing.

---

---

### Example 1.4 Creating a Disk with a Backing Image

```
tbielawa@deepfryer:/srv/images$ mkdir base-images
tbielawa@deepfryer:/srv/images$ mkdir webserver01
tbielawa@deepfryer:/srv/images$ cd base-images

tbielawa@deepfryer:/srv/images/base-images$ qemu-img create -f qcow2 image- ↵
  webserver-base.qcow2 10G
Formatting 'image-webserver-base.qcow2', fmt=qcow2 size=10737418240 encryption=off ↵
  cluster_size=0
tbielawa@deepfryer:/srv/images/base-images$ cd ../webserver01

tbielawa@deepfryer:/srv/images/webserver01$ qemu-img create -b /srv/images/base- ↵
  images/image-webserver-base.qcow2 -f qcow2 image-webserver-devel.qcow2
Formatting 'image-webserver-devel.qcow2', fmt=qcow2 size=10737418240 backing_file ↵
  ='/srv/images/base-images/image-webserver-base.qcow2' encryption=off ↵
  cluster_size=0

tbielawa@deepfryer:/srv/images/webserver01$ qemu-img info image-webserver-devel. ↵
  qcow2
image: image-webserver-devel.qcow2
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 136K
cluster_size: 65536
backing file: /srv/images/base-images/image-webserver-base.qcow2 (actual path: / ↵
  srv/images/base-images/image-webserver-base.qcow2)
tbielawa@deepfryer:/srv/images/webserver01$
```

1. It's bad practice to just have a bunch of disk images in a directory so we made two here. `/srv/images/base-images/` to hold all the base-images on this system and `/srv/images/webserver01` to later hold the disk assigned to the virtual machine.
2. Next we go into the base images directory and create a small 10G image, type: QCOW2.
3. Normally what we used to do at this point is create a virtual machine that uses this disk for it's primary drive. It would get a base OS provisioned on it and any other tweaks we needed there each time it was wiped.
4. Once the machine was what we wanted in a "Golden Master" it was shutdown and the backing image would be made read-only.
5. The next step was creating the copy-on-write (COW) image. See how in the example we give the `-b` option with the *full path* to the base-image? Also note that no size is given after the file name. Size is constrained to the size of a disks base-image.
6. With the image preparation complete we would modify the virtual machines configuration and set its primary disk drive to the COW in the `webserver01` directory.

Sometimes we would want to update a base-image to resemble the contents of an attached COW image. Maybe we wanted to make the latest system updates a part of the base image, or a configuration setting needed to be updated. This was as simple as making the base-image read-write, and running **`qemu-img commit /srv/images/webserver01/image-webserver-devel.qcow2`**.

---



### Important

It's not crucial, but highly recommend that the virtual machine is turned off when running the **commit** command.

---

## 1.5 Creating Simple Images

The simplest operation (next to deleting an image) you can do is creating a new virtual disk image. Depending on what type you choose there are several options available when creating an image. For example, if you want encryption, if you want a format that supports compression, or even what existing image to base this new disk off of.

In this example we will start simple and only show how to create basic images in different formats. Each will appear to a virtual machine as being 10G in capacity.

---

### Example 1.5 Using **qemu-img** to Create RAW Images

---

```
tbielawa@deepfryer:/srv/images/$ qemu-img create webserver.raw 10G
Formatting 'webserver.raw', fmt=raw, size=10485760 kB
```

---

From the `fmt` attribute above you can see that the type of virtual disk created was RAW, this is the default when using **qemu-img**. You can also see that the disk will appear as a 10G drive when used in a virtual machine.

---

## Chapter 2

# Disk Concepts

### 2.1 Introduction

The best way to learn is by doing, so to learn the concepts of virtual disks we're going to create a 1GiB virtual disk from scratch. This information is applicable to the topic of disks in general, it's value is not limited to virtual disks.

What makes virtual disks any different from actual hard drives? We'll examine this question by creating a virtual disk from scratch.

What does your operating system think a disk drive is? I have a 320 GB SATA drive in my computer which is represented in Linux as the file `/dev/sda`. Using **file**, **stat** and **fdisk** we'll see what Linux thinks the `/dev/sda` file is.

Lets start out by looking at what a regular drive looks like to our operating system. Throughout this document the regular drive we'll be comparing our findings against will be a 320G SATA hard drive drive that Linux references as `/dev/sda`. The following example shows some basic information about the device.

---

**Example 2.1** Regular Disk Drive

---

```
<tbielawa>@(fridge) [~/images]
$ file /dev/sda
/dev/sda: block special
<tbielawa>@(fridge) [~/images]
$ stat /dev/sda
  File:  `/dev/sda'
  Size: 0   Blocks: 0           IO Block: 4096   block special file
Device: 5h/5d   Inode: 5217           Links: 1        Device type: 8,0
Access: (0660/brw-rw----)  Uid: (   0/   root)   Gid: (   6/   disk)
Access: 2010-09-15 01:09:02.060722589 -0400
Modify: 2010-09-12 11:03:20.831372852 -0400
Change: 2010-09-12 11:03:26.226369247 -0400
<tbielawa>@(fridge) [~/images]
$ sudo fdisk -l /dev/sda
```

```
Disk /dev/sda: 320.1 GB, 320071851520 bytes
255 heads, 63 sectors/track, 38913 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x12031202
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1		1	25496	204796588+	7	HPFS/NTFS
/dev/sda2		25497	31870	51199155	83	Linux
/dev/sda3		31871	33086	9767520	82	Linux swap / Solaris
/dev/sda4		33087	38913	46805377+	5	Extended
/dev/sda5	*	33087	38913	46805346	83	Linux

---

The term "block" is generally interchangeable with the term "sector". The only difference in their meaning is contextual. It's common usage to say block when referring to the data being referenced and to use sector when speaking about disk geometry. Officially the term "data block" was defined by ANSI ASC X3 in ANSI X3.221-199xata *AT Attachment Interface for Disk Drives (ATA-1)* §3.1.3 as:

**data block**

This term describes a data transfer, and is typically a single sector [...].

Storage units need to be clearly defined. Luckily some very smart people already took care of that. The International Electrotechnical Commission defined binary prefixes for use in the fields of data processing and data transmission. Below are some applied to bytes. See [UNITS\(7\)](#) for the full prefix listing.

Abbrev.	Measurement	Name
1B	= 8 Bits	The byte
1KiB	= 1B * 2 <sup>10</sup>	The kibibyte
1MiB	= 1KiB * 2 <sup>10</sup>	The mebibyte
1GiB	= 1MiB * 2 <sup>10</sup>	The gibibyte
	= 1B * 2 <sup>10</sup> * 2 <sup>10</sup> * 2 <sup>10</sup>	
	= 1B * 2 <sup>30</sup>	
	= 1B * 1073741824 (1,073,741,824)	

## 2.2 Learning by doing (Creating a 1GiB virtual disk from scratch)

### 2.2.1 Background on the dd command

We'll use the **dd** command to create the file that represents our virtual disk. Other higher level tools like **qemu-img** exist to do similar things but using **dd** will give us a deeper insight into what's going on. **dd** will only be used in the introductory part of this document, later on we will use the **qemu-img** command almost exclusively.

If we're creating a 1GiB disk that means the file needs to be exactly  $2^{30}$  bytes in size. By default **dd** operates in block sized chunks. This means that to create  $2^{30}$  bytes it needs to push a calculable number of these chunks into our target disk file. This number is referred to as the `count`. To calculate the proper `count` setting we need only to divide the total number of bytes required by the size of a each block. "The block size is given to **dd** with the `bs` option. It specifies the block size in bytes. If not explicitly defined, it defaults to 512 byte blocks ( $2^9$ )"

$$\text{count} = 2^{30} / 2^9 = 1073741824 / 512 = 2097152 \text{ (2,097,152)}$$

EQUATION 2.1: Calculating the Count

We need something to fill the file with that has a negligible value. On Unix systems the best thing to use is the output from `/dev/zero` (a special character device like a keyboard). We specify `/dev/zero` as our input file to **dd** by using the `if` option.

---

#### Note

`/dev/zero` doesn't provide endless "0"s. It actually provides endless NUL control characters (^@ in Caret Notation<sup>4</sup>). The NUL control character has the octal value 000. The actual ASCII zero character **0** has the octal value 060.

---

<sup>4</sup>[Caret Notation](#) on Wikipedia

NUL being a control character<sup>1</sup> means it's a non-printing character (it doesn't represent a written symbol), so if you want to identify it you can use **cat** like this to print 5 NUL characters in Caret Notation:

```
<tbielawa>@(fridge) [~/images]
$ dd if=/dev/zero bs=1 count=5 2>/dev/null | cat -v
^@^@^@^@^@
```

You can also convert the output from `/dev/zero` into ASCII **0** characters like this:

```
<tbielawa>@(fridge) [~/images]
$ if=/dev/zero bs=1 count=5 2>/dev/null | tr "\0" "\60"
00000
```

### 2.2.2 Running dd

With the information from the preceding sections we can now create the file that will soon be a virtual disk. The file we create will be called `disk1.raw` and filled with 2097152 blocks of NUL characters from `/dev/null`. Here's the command:

---

<sup>1</sup>[Control Characters](#) on Wikipedia

---

**Example 2.2** Running the **dd** command

---

```
<tbielawa>@(fridge) [~/images]
$ dd if=/dev/zero of=disk1.raw bs=512 count=2097152
```

---

Now that you know what `/dev/zero` is it's obvious this is just a file containing  $2^{30}$  bytes (1GiB) of data. Each byte literally having the value `'0'`.

### 2.2.3 Examining the created file

3 Like in Example 2.1 lets take a look at the file we created from the operating systems point of view.

```
<tbielawa>@(fridge) [~/images]
$ dd if=/dev/zero of=disk1.raw bs=512 count=2097152
2097152+0 records in
2097152+0 records out
1073741824 bytes (1.1 GB) copied, 10.8062 s, 99.4 MB/s
<tbielawa>@(fridge) [~/images]
$ file disk1.raw
disk1.raw: data
<tbielawa>@(fridge) [~/images]
$ stat disk1.raw
  File: `disk1.raw'
  Size: 1073741824  Blocks: 2097152    IO Block: 4096   regular file
Device: 805h/2053d  Inode: 151552      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 500/tbielawa)   Gid: ( 500/tbielawa)
Access: 2010-09-15 02:51:36.147724384 -0400
Modify: 2010-09-15 02:51:25.729720057 -0400
Change: 2010-09-15 02:51:25.729720057 -0400
<tbielawa>@(fridge) [~/images]
$ fdisk -l disk1.raw

Disk disk1.raw: 0 MB, 0 bytes
255 heads, 63 sectors/track, 0 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x00000000

Disk disk1.raw doesn't contain a valid partition table
```

From this it's quite clear that there isn't much `disk1.raw` has in common with the actual disk drive `sda`.

- **file** thinks it's "data", which the **file** manual page says is how it labels what are usually 'binary' or non-printable files.
- **stat** says it's just a regular file.
- **fdisk** doesn't know how big it is, nor can it find any partition information on it.

```
|      sda      |      disk1.raw      |
-----
```

---

```

file | block special | data |
stat | block special | regular file |
fdisk | Has partition | no partition |
      | table | table |
-----

```

These results make perfect sense. `disk1.raw` is just  $2^{30}$  0's in a row

## 2.2.4 Create a Partition Table

Use GNU **parted** to put a valid partition table on the image file.

```

<tbielawa>@(fridge) [~/images]
$ parted disk1.raw mklabel msdos
WARNING: You are not superuser. Watch out for permissions.

```

Lets examine the image again to see how the operating system thinks it has changed.

```

<tbielawa>@(fridge) [~/images]
$ file disk1.raw
disk1.raw: x86 boot sector, code offset 0xb8
<tbielawa>@(fridge) [~/images]
$ stat disk1.raw
  File: `disk1.raw'
  Size: 1073741824      Blocks: 2097160      IO Block: 4096   regular file
Device: 805h/2053d    Inode: 151552       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 500/tbielawa)  Gid: ( 500/tbielawa)
Access: 2010-09-15 19:38:30.516826093 -0400
Modify: 2010-09-15 19:38:25.934611550 -0400
Change: 2010-09-15 19:38:25.934611550 -0400
<tbielawa>@(fridge) [~/images]
$ fdisk -l disk1.raw
You must set cylinders.
You can do this from the extra functions menu.

Disk disk1.raw: 0 MB, 0 bytes
255 heads, 63 sectors/track, 0 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x000e44e8

```

Device	Boot	Start	End	Blocks	Id	System
--------	------	-------	-----	--------	----	--------

## 2.2.5 What parted changed

- Now when we run **file** instead of "data" **file** thinks it is an x86 boot sector. That sounds pretty accurate since we just put a partition table on it.
- **stat** still thinks it's a regular file (as opposed to a block special device, or a socket, etc...).
- **fdisk** was able to find a partition table in that boot sector that **file** found.

	sda	disk1.raw	disk1.raw (parted)
file	block special	data	x86 boot sector
stat	block special	regular file	regular file
fdisk	Has partition table	no partition table	valid partition table. Unknown
			---cylinder count---

## 2.2.6 Creating Partitions

After using **parted** `disk1.raw` has a partition table, but does that mean we can create partitions on it now? Lets run **fdisk** on `disk1.raw`.

```
<tbielawa>@(fridge) [~/images]
$ fdisk disk1.raw
You must set cylinders.
You can do this from the extra functions menu.

Command (m for help):
```

To calculate how many cylinders to specify you could work backwards from Equation 2.2<sup>2</sup> but the number of unknowns at this time make it prohibitively difficult for us to work out. It's provided in this documentation as a general informative reference.

Disk Capacity = (cylinders/disk) \* (heads/cylinder) \* (tracks/head) \* (sectors/track) \* (bytes/sector)

EQUATION 2.2: Total Capacity of a Disk

A much simpler way to create partitions (still using **fdisk**) is by is by accessing the file as if it were an actual device. Doing this requires creating loop devices.

### 2.2.6.1 Creating partitions on a loop device.

Instead of using **fdisk** on `disk1.raw` directly, we'll create a loop device and associate `disk1.raw` with it. From here on we'll be accessing our virtual drives through loop devices.

Why are we doing this? And what is a loop device?

Unfortunately for `disk1.raw`, it will never be anything more than just a file. The operating just system doesn't have interfaces for block operations against files. As the kernel creates the block special device `/dev/sda` to represent my hard drive, we need to create a block special device to represent our virtual disk. This is called a loop device<sup>3</sup> You can think of a loop device, e.g., `/dev/loop1`, like a translator.

With a loop device inserted between programs and our disk image we can view and operate on the disk image as if it were a regular drive. When accessed through a loop device **fdisk** can properly determine the number of cylinders, heads, and everything else required to create partitions.

<sup>2</sup>Disk Geometry

<sup>3</sup>Don't confuse the often misused term "loopback device" with a "loop device." In networking a loopback device refers to a virtual interface used for routing within a host. `localhost` is the standard hostname given to the loopback address `127.0.0.1`. See [rfc1700](#) for additional information.

### 2.2.6.2 Creating a loop device

#### Note

Since we'll be working with the kernel to create a device you'll need to have super user permissions to continue<sup>a</sup>.

---

<sup>a</sup>FUSE (Filesystem in Userspace) has a module called `MountIo` that allows non-root users to make loop devices.

To create a loop device. Run the **losetup** command with the `-f` option and an available loop device will be selected automatically and associated with `disk1.raw`.

#### Example 2.3 Creating a loop device

```
<tbielawa>@(fridge) [~/images]
$ sudo losetup -f disk1.raw
<tbielawa>@(fridge) [~/images]
$ sudo losetup -a
/dev/loop1: [0805]:151552 (/home/tbielawa/images/disk1.raw)
```

You can run **file**, **stat**, and **fdisk** on `disk1.raw` to verify that nothing has changed since we put a partition table on it with **parted**.

### 2.2.6.3 Examine the loop device

```
<tbielawa>@(fridge) [~/images]
$ file /dev/loop0
/dev/loop0: block special
<tbielawa>@(fridge) [~/images]
$ stat /dev/loop0
  File: '/dev/loop0'
  Size: 0          Blocks: 0          IO Block: 4096   block special file
Device: 5h/5d   Inode: 5102         Links: 1        Device type: 7,0
Access: (0660/brw-rw----)  Uid: (  0/   root)   Gid: (  6/   disk)
Access: 2010-09-15 01:22:09.909721760 -0400
Modify: 2010-09-12 11:03:19.351004598 -0400
Change: 2010-09-12 11:03:24.694640781 -0400
<tbielawa>@(fridge) [~/images]
$ sudo fdisk -l /dev/loop0
Disk /dev/loop0: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x000e44e8

   Device Boot      Start         End      Blocks   Id  System
```

Look back at Example 2.1 where I ran these commands against my actual disk drive (`/dev/sda`) and you'll see the results are quite similar.

- **file** detects `loop0` as a block special device.
- **stat** does too.
- **fdisk** no longer says we need to set the cylinders.

```
|   sda   |   disk1.raw   | disk1.raw (parted) |   /dev/loop0   ↵
-----|-----
file | block special | data           | x86 boot sector | block ↵
special |
stat | block special | regular file   | regular file    | block ↵
special |
fdisk | Has partition | no partition   | valid partition | Has ↵
partition table, |
| table       | table         | table. Unknown | cylinder count ↵
-----|-----
-----cylinder count--|--known ↵
-----|-----
```

Our virtual disk is starting to look like a real hard drive now!

To bring this section to a conclusion we'll create a partition, format it with ext3, and then mount it for reading and writing.

#### 2.2.6.4 Creating partitions

Open `/dev/loop0` (or whatever loop device your disk was associated with) in **fdisk** to create a partition.

---

**Example 2.4** Creating a partition with **fdisk**

---

```
<tbielawa>@(fridge) [~/images]
$ sudo fdisk /dev/loop0

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-130, default 1):
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-130, default 130):
Using default value 130

Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: Re-reading the partition table failed with error 22: Invalid argument.
The kernel still uses the old table. The new table will be used at
the next reboot or after you run partprobe(8) or kpartx(8)
Syncing disks.
<tbielawa>@(fridge) [~/images]
$ sudo fdisk -l /dev/loop0

Disk /dev/loop0: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Disk identifier: 0x000e44e8

   Device Boot      Start         End      Blocks   Id  System
/dev/loop0p1            1         130     1044193+  83  Linux
```

---

### 2.2.6.5 Formatting Partitions

Unlike `/dev/sda` we can't just create a partition on the first `loop0` partition by addressing it as `/dev/loop-0`. This is because the kernel has no device created to represent this. Instead we'll have to create another device associated with a specific offset in our device/file.

Why do we have to specify an offset? How do we know what offset to specifically?

An "offset" just means how far from the beginning of a device something is. We need to know this because we're going to create a device mapped to the first partition (Linux does this automatically for regular disks during the boot process). Partitions don't start on the first block of a device because filesystem information is stored there. To specify the offset we just need to know what sector the partition (`loop0p1`) starts on. **fdisk** can give us this information. (Spoiler: 9 times out of 10 the offset for the first partition will be  $512 * 63 = 32256$ ).

Print the partition table using **fdisk** with the `-u` option to switch the printing format to sectors instead of cylinders for units.

---

```
<tbielawa>@(fridge) [~/images]
$ sudo fdisk -ul /dev/loop0

Disk /dev/loop0: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Disk identifier: 0x000e44e8

   Device Boot      Start         End      Blocks   Id  System
/dev/loop0p1                63      2088449     1044193+  83  Linux
```

`/dev/loop0p1` is our first partition and from the table above we know that it starts on sector 63. Since we have to specify offsets in bytes we multiply 63 by 512 (the default block size) to obtain an offset of 32256 bytes.

```
<tbielawa>@(fridge) [~/images]
$ sudo losetup -o 32256 -f /dev/loop0
<tbielawa>@(fridge) [~/images]
$ sudo losetup -a
/dev/loop0: [0805]:151552 (/home/tbielawa/images/disk1.raw)
/dev/loop1: [0005]:5102 (/dev/loop0), offset 32256
```

Now that we have `/dev/loop1` representing partition of our virtual disk we can create a filesystem on it and finally mount it.

---

### Example 2.5 Formatting the partition

```
<tbielawa>@(fridge) [~/images]
$ sudo mkfs -t ext3 /dev/loop1
mke2fs 1.41.9 (22-Aug-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65536 inodes, 262136 blocks
13106 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376

Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 25 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
<tbielawa>@(fridge) [~/images]
$ sudo losetup -d /dev/loop1
<tbielawa>@(fridge) [~/images]
$ sudo losetup -d /dev/loop0
<tbielawa>@(fridge) [~/images]
$ mkdir partition1
<tbielawa>@(fridge) [~/images]
$ sudo mount -t ext3 -o loop,offset=32256 disk1.raw partition1/
<tbielawa>@(fridge) [~/images]
$ mount | grep partition1
/dev/loop0 on /home/tbielawa/images/partition1 type ext3 (rw,offset=32256)
<tbielawa>@(fridge) [~/images]
$ df -h partition1/
Filesystem                Size      Used Avail Use% Mounted on
/dev/loop0                 1008M    18M   940M   2% /home/tbielawa/images/partition1
```

---

#### Note

The same procedure applies to any arbitrary partition: obtain the starting sector, multiply by block size.

---

#### 2.2.6.6 Cleaning Up

You can detach the loop device (while leaving your file intact) by giving the `-d` option to **losetup**.

---

### Example 2.6 Detaching a loop device

```
<tbielawa>@(fridge) [~/images]
$ sudo losetup -d /dev/loop1
```

---

## Chapter 3

# Advanced Operations

### 3.1 Creating a Mountable Disk

In the previous section I went through creating a disk that you can use as the boot disk for a virtual machine. In that example it was expected that you would configure your virtualization software to use that disk and then the operating system installation would take care of creating the disk label and file systems. In this section I'll show you the steps required to do that by yourself.

### 3.2 Cloning a Physical Disk

"Everything's a file"

— Unknown

It is also possible to clone an existing physical hard disk using the **qemu-img** `convert` option. This is possible in part due to the original philosophy laid down by Dennis Ritchie and Ken Thompson when they first created Unix that everything's treated as a file. The syntax for the `convert` option is below.

```
convert [-c] [-f fmt] [-O output_fmt] [-o options] filename [filename2 [...]] ↔  
output_filename
```

I never fully grasped the "everything's a file" concept until I tried (expecting to fail) to use the `convert` option to create a virtual disk image of an actual hard drive. This section explains how to do just that.

---

**Example 3.1** Making a virtual disk from a physical disk

---

## Appendix A

# Appendices

### A.1 UNITS

units, kilo, kibi, mega, mebi, giga, gibi — decimal and binary prefixes

#### DESCRIPTION

##### Binary prefixes

The binary prefixes resemble the decimal ones, but have an additional 'i' (and "Ki" starts with a capital 'K'). The names are formed by taking the first syllable of the names of the decimal prefix with roughly the same size, followed by "bi" for "binary".

Prefix	Name	Value
Ki	kibi	$2^{10} = 1024$
Mi	mebi	$2^{20} = 1048576$
Gi	gibi	$2^{30} = 1073741824$
Ti	tebi	$2^{40} = 1099511627776$
Pi	pebi	$2^{50} = 1125899906842624$
Ei	exbi	$2^{60} = 1152921504606846976$

See also: <http://physics.nist.gov/cuu/Units/binary.html>

#### Discussion

Before these binary prefixes were introduced, it was fairly common to use  $k=1000$  and  $K=1024$ , just like  $b=\text{bit}$ ,  $B=\text{byte}$ . Unfortunately, the  $M$  is capital already, and cannot be capitalized to indicate binary-ness.

At first that didn't matter too much, since memory modules and disks came in sizes that were powers of two, so everyone knew that in such contexts "kilobyte" and "megabyte" meant 1024 and 1048576 bytes, respectively. What originally was a sloppy use of the prefixes "kilo" and "mega" started to become regarded as the "real true meaning"

when computers were involved. But then disk technology changed, and disk sizes became arbitrary numbers. After a period of uncertainty all disk manufacturers settled on the standard, namely  $k=1000$ ,  $M=1000k$ ,  $G=1000M$ .

The situation was messy: in the 14k4 modems,  $k=1000$ ; in the 1.44MB diskettes,  $M=1024000$ ; etc. In 1998 the IEC approved the standard that defines the binary prefixes given above, enabling people to be precise and unambiguous.

Thus, today,  $MB = 1000000B$  and  $MiB = 1048576B$ .

In the free software world programs are slowly being changed to conform. When the Linux kernel boots and says:

```
hda: 120064896 sectors (61473 MB) w/2048KiB Cache
```

the MB are megabytes and the KiB are kibibytes.

---