The Linux Sysadmins Guide to Virtual Disks

From the Basics to the Advanced

Copyright © 2009-2016 Tim Bielawa

The Linux Sysadmin's Guide to Virtual Disks by Tim Bielawa is licensed under the *Creative Commons Attribution-ShareAlike 4.0 International License* (CC BY-SA 4.0). To view a copy of the CC BY-SA 4.0 License, please visit:

https://creativecommons.org/licenses/by-sa/4.0/

Second Edition DRAFT - In Progress - 20XX

Printed in The United States

Published by Scribe's Guides

New York, NY, USA

Web Site: https://scribesguides.com/

Editors:

Henry Graham

Jonathan Connell

Jyoti Sabharwal

Cover Designer: Tim Bielawa

The Disk Components Image (Figure B.1, "Disk Drive Components" [67]) which appears on both covers and in the Disk Drive History appendix of this book is remixed from the Wikipedia image `Disk-structure2.svg`. The original image [1] is licensed as a work in the worldwide public domain. Original image created by Wikipedia users MistWiz [2] and Heron2 [3]. The image was last updated by Tim Bielawa as displayed in this book.

Library of Congress Control Number: 2016903293

Soft Cover ISBN-13: 978-0-692-64121-7

[1] https://commons.wikimedia.org/wiki/File:Disk-structure2.svg

[2] https://en.wikipedia.org/wiki/User:MistWiz

[3] https://en.wikipedia.org/wiki/User:Heron2

Contents

Dedication

This book is dedicated to the loving memories of Seth Vidal and Donald Brewer.

Seth, you were the kind of person who always stuck to what they believed in once they decided what that was. There are few people you can truly say that about. Your creativity and technical prowess changed the world in ways most people couldn't dream of. It's a tragedy we lost you so early, but I can promise you one thing: your contributions to the world were inspirational and it's going to take a lot of people to pick up where you left off.

As you used to say, Don, *lemme be honest wit'cha*.... You were always a stand up gentleman, sharp as a tack, and loyal as hell to anyone you were responsible for. You may be gone now, but you will never be forgotten by those whose lives you touched.

Acknowledgments

This book wouldn't have been possible without the gracious advice, contributions, and support I received from so many people. In fact, there's so many people that I can't remember them all! You know who you are — coworkers, friends, the people reporting errors, the people sending random emails saying they enjoyed the book, the people sending emails describing a tough situation they were in that the guide helped them get out of. Thanks everybody!

Andrew "Hoss" Butcher. You're a badass friend and an even more talented individual. You and I could hack or riff on anything together and have fun with it. Give Hampus and Ripley a kitty treat for me some time, will ya?

Thanks to John Eckersberg and Chris Venghaus for copious amounts of feedback early on. It really helped to stoke my fire and get things moving.

And a special thanks to Chris for being my biggest word-of-mouth referral. I have no idea how you meet all these people with burning needs to have their virtual disk questions answered, but I'm glad you refer them to me when you do.

Alex Wood, your eclectic interests never fail to serve my personal interests in some way. Thanks for the assist with that XSLT a while back. Coincidentally, that XSLT [4] pertained to rendering the very *Acknowledgements* section you're reading right now. So I guess we all owe you debt of gratitude for that!

Jorge Fábregas, you were a fabulous unexpected resource when most of the major writing was happening for this book. Thanks for repeatedly reaching out to me with your feedback and suggestions and pointing out errors. This book is better because of your unique contributions.

[4] See my blog post about dblatex+docbook+acknowledgements sections for the interesting details: `https://blog.lnx.cx/2013/03/27/dblatex-docbook-acknowledgements-and-pdf-output/`

Thank you Jon Connell, Henry Graham, and Jyoti Sabharwal for editing! Mark Dalrymple, Jason Hibbets, and Christopher Negus, thanks for the reviews and general authoring/publishing advice and encouragement.

Obligatory shout-outs to Norman Walsh, the man I consider the living personification of DocBook XML [5] , and Bob Stayton, author of DocBook XSL: The Complete Guide [6] .

Thank you Wikipedia contributor MistWiz for creating the original image, Figure B.1, "Disk Drive Components" [67], used in the appendix. Also, thanks to Wikipedia contributor Heron2 for making later updates. The disk components image featured in this book has subsequent changes I personally made. Image used and remixed according to permissions detailed in the Licensing section of the image's Wikipedia page [7] .

To my wife, Alicia, thanks for making me so happy and keeping me sane. Thank you for encouraging me to work on this book when I didn't want to. And, thank you for *everything else*.

Finally, my biggest "thanks" goes to David Krovich. A mentor, friend, and button-pusher for many years now. You always encouraged me to be better than I was. Without the opportunities you offered me, and the radical influence you had on my life, this book would never have even reached conception. Truth be told, Chapter 2, The Virtual Disk Cookbook [7] section is mostly a merge and refresh of a lot of the notes I took, and staff documentation I wrote, working on one of our provisioning projects. Those notes became the first chapters of this book.

It was while working under your instruction that I discovered my passion for documenting everything I learned. This book is a testament to that passion. Thanks, Kro.

[5] DocBook Homepage: `http://www.docbook.org/`

[6] Read DocBook XSL: The Complete Guide online: `http://www.sagehill.net/docbookxsl/index.html`

[7] Original Disk Components Image: `https://en.wikipedia.org/wiki/File:Disk-structure2.svg`

Chapter 1

Introduction

1.1 Introduction

I was motivated to write this book 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 provide is a book which will demonstrate the core concepts of virtual disk management. This book will concern itself primarily with the **qemu-img** tool and common GNU/Linux disk utility tools like **fdisk**, **parted**, and **resize2fs**. Most importantly to me, in the case of non-trivial examples, I hope to identify what the relevant technical concepts are and how they work up to the final result of each example.

1.2 Typographical Conventions

The following describes the typographical conventions used throughout this book.

References

References to other sections will look like this: Chapter 7, Troubleshooting/FAQs [57]. The format is: *Chapter/section title* followed by the page number in [brackets].

Footnotes

References to footnotes[1] appear as small superscripted numbers flowing inline with the current discussion.

Terminology & Emphasis

The introduction of a new or alternative term, as well as phrases which have been given emphasis, are formatted in italics:

- The disk image has been *sparsified*
- You should *always* wear clean socks

Commands & Options

The name of commands are formatted in bold, an option you would give to a command is formatted in a monospaced sequence, for example: give the `-ltrsh` options to the **ls** command.

Filesystem Paths

Names or paths to files, directories, and devices on the filesystem are formatted in a monospaced sequence: `/dev/loop0p1`

Examples

Examples are formatted in a gray box with a title bar which provides the example number and title.

Example 1.1 An example of examples

```
[~/vdg] 18:38:17  (master)
$ cat /etc/redhat-release
Fedora release 19 (Schrödinger's Cat)
```

Notes, Warnings, and Other Important Information

Note

A note will provide additional information relevant to the current discussion.

[1] Hello! I am a footnote.

Important
Warnings and other important information which you should know before executing any commands will appear in an admonition such as this.

1.3 Units & Prefixes

Throughout this book you will see file sizes specified with an assortment of units. For example: 42 kB, 42 Mb, 42 GiB, 42 G, 42 GiB.

Without an explanation this may seem confusing, random, and inconsistent. However, there is a method to this madness:

1. The unit used in discussion preceding/following an example is consistent with the convention used in the example
2. Without any scope or context, binary prefixes are used (e.g., 1024 KiB, 35565 MiB)

For additional literature on why this necessary, I refer you to Appendix A, Appendix: Man Pages [64]

1.4 Getting Help/Feedback

If you find a typographical or any other error in this book, or if you have thought of a way to make this book better, I would love to hear from you! Please submit a report in GitHub [2] . You can also read or clone the entire book's DocBook 5.1 XML source from GitHub.

If you have a suggestion for improving the book, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so I can find it easily. I also recommend you review the suggestions in Chapter 7, Troubleshooting/FAQs [57].

If you're submitting an error with an example, please try and include as much relevant information about your setup as possible. This includes (but is not limited to): your operating system and version, the version of the software the example happens with, if you are running the command as the root user or not, and the exact commands to run to reproduce the error.

[2] `https://github.com/tbielawa/Virtual-Disk-Guide/issues`

1.5 Updates and Alternative Formats

The latest version of this book is always available online for **free** in the following digital formats:

- PDF
- HTML Single Page

If you find this book useful, please consider supporting the author by purchasing a hard-copy. Visit the publisher's website at `http://scribesguides.com/` for purchasing options and links to alternative formats.

This book was generated from commit `98bafa1` [3] on Sun 13 Mar 2016 11:41:51 PM EDT.

1.6 About The Author

Tim Bielawa (or *Shaggy* if you knew him in college) has been a system administrator since his humble beginnings in 2007. Back then he was working in the Systems Staff Group in the West Virginia University Computer Science Department.

Now-a-days Tim works at Red Hat, Inc., makers of Red Hat Enterprise Linux, and sponsors of the Fedora Project. Tim has been involved in Open Source communities for around a decade, and a contributor/maintainer for about half that time. In his spare time he enjoys writing documentation [4] , solving interesting problems (and blogging [5] about them), and building things [6] .

`bitmath` is a Python library Tim wrote which simplifies a lot of work required to manipulate (add, subtract, convert) prefix units (MiB, kB, TB, etc). Much of the inspiration for writing the `bitmath` library came directly from working on this book. `bitmath` is loaded with features: converting between units, arithmetic, best human-readable representation, rich comparison, sorting, the list of features goes on. Check out the bitmath docs [7] or GitHub project [8] for more information on getting started.

[3] `https://github.com/tbielawa/Virtual-Disk-Guide/commit/98bafa1`
[4] Docs on lnx.cx: `http://lnx.cx/docs/`
[5] My blog, Technitribe: `https://blog.lnx.cx`
[6] GitHub: tbielawa `https://github.com/tbielawa/`
[7] Read the bitmath docs online: `https://bitmath.readthedocs.org/en/latest/`
[8] bitmath on GitHub.com: `https://github.com/tbielawa/bitmath`

Chapter 2

The Virtual Disk Cookbook

In this section we're just going to cover things you'll find yourself needing to do from time to time. It's assumed that you're comfortable with the concepts already and don't need everything explained. Theory and concepts will be covered later on in Chapter 3, Disk Concepts [30].

2.1 Creating Simple Images

The simplest operation you can do (next to deleting an image) is creating a new virtual disk image. Depending on what format you choose there are several options available when creating an image:

- Encryption
- Compression
- Backing images [1]
- Snapshots

In this example we will start simple and only show how to create basic images in different formats. Each image we create will appear to a virtual machine as a drive with 10GB of capacity.

[1] Creating Disks with Backing Images: Section 2.5, "Creating Disks with Backing Images" [26]

Example 2.1 Using **qemu-img** to Create RAW Images

```
$ qemu-img create webserver.raw 10G
Formatting 'webserver.raw', fmt=raw, size=10485760 kB
```

From the `fmt` attribute in the output above we can see that the format of the virtual disk we created is of type *RAW*[2], this is the default when using **qemu-img**. Where it says `size=...` we see that the disk was created with a capacity of 10485760 kB, or 10gB.

2.2 Resizing Disk Images

In this section we'll resize two different virtual disk images. The first will be a RAW image, the other will be a QCOW2 image. The RAW section is more involved in that we'll do all of the resizing operations *outside* of a virtual machine. In the QCOW2 section I'll show the (simpler) steps which take place both outside and inside of a virtual machine.

2.2.1 Resizing RAW Images

In this part we'll add 2GiB to a disk image I created of a 1GiB USB thumb drive [3] The thumb drive has two roughly equal sized partitions, both are EXT4.

At the end of this section we'll have done the following:

- Enlarged the disk by 2GiB with **qemu-img**
- Shifted the the second partition 1024MiB right into the new space with **gparted**
- Enlarged the first partition by about 1GiB with **gparted**
- Resized the first filesystem to use the new space on its partition with **resize2fs**

[2] Section 5.1, "RAW" [49]
[3] See Section 2.7, "Cloning a Physical Disk" [28] for instructions on how to do this yourself.

Example 2.2 Resize a RAW Image

```
# qemu-img info thumb_drive_resize.raw
image: thumb_drive_resize.raw
file format: raw
virtual size: 966M (1012924416 bytes)
disk size: 914M

# qemu-img resize thumb_drive_resize.raw +2G
Image resized.

# qemu-img info thumb_drive_resize.raw
image: thumb_drive_resize.raw
file format: raw
virtual size: 2.9G (3160408064 bytes)
disk size: 914M
```

Next we need to create device maps and devices linking to the enlarged disk image so we may interact with it. We will use the **kpartx** command [4] to automatically create loop devices [5] [6] and device maps to the partitions. The `-a` option means we're adding partition mappings and the `-v` option means to do it with increased verbosity so we know the names of the created devices.

Example 2.3 Create devices with **kpartx**

```
# kpartx -av ./thumb_drive_resize.raw
add map loop0p1 (253:8): 0 3082432 linear /dev/loop0 2048
add map loop0p2 (253:9): 0 996030 linear /dev/loop0 3084480

# dmsetup ls | grep loop
loop0p2 (253:9)
loop0p1 (253:8)
```

Now we're going to use **gparted** to resize the partitions in the disk image. There are two important things to keep in mind:

[4] For more information on the **kpartx** command, see Chapter 4, Helper Utilities [44]

[5] 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 *Assigned Numbers* for additional information (`http://tools.ietf.org/html/rfc1700`).

[6] We'll revisit loop devices in Chapter 3, Disk Concepts [30]

1. **gparted** expects to find the `loop0p*` devices in `/dev/`, not in `/dev/mapper/`
2. **gparted** won't list loop devices in its device selection menu

When we ran **kpartx** it created symbolic links to the new devices (`/dev/dm-*`) which map to the partitions on `/dev/loop0`. We can use this information to create the symlinks necessary for **gparted** to locate `loop0p*`.

Example 2.4 Create the symbolic links

```
# ls -l /dev/mapper/loop0p*
lrwxrwxrwx 1 root root 7 Jan 21 15:07 /dev/mapper/loop0p1 -> ../ ↩
    dm-8
lrwxrwxrwx 1 root root 7 Jan 21 15:07 /dev/mapper/loop0p2 -> ../ ↩
    dm-9

# ln -s /dev/dm-8 /dev/loop0p1
# ln -s /dev/dm-9 /dev/loop0p2

# ls -l /dev/loop0p[12]
lrwxrwxrwx 1 root root 9 Jan 21 15:23 /dev/loop0p1 -> /dev/dm-8
lrwxrwxrwx 1 root root 9 Jan 21 15:23 /dev/loop0p2 -> /dev/dm-9
```

Once the symlinks are created we can run **gparted** from the command line with `/dev/loop0` as the device argument.

Example 2.5 Run **gparted**

```
# gparted /dev/loop0
======================
libparted : 3.0
======================
```

Now **gparted** should open and show the two existing partitions, as well as the 2GiB of unallocated space we just added to the image:

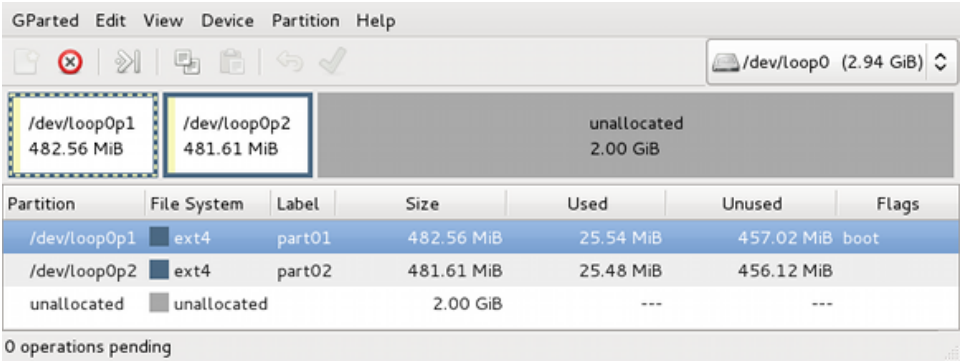

Figure 2.1: Welcome to **gparted**

Right click the second partition, `loop0p2`, and select the *Resize/Move* option:

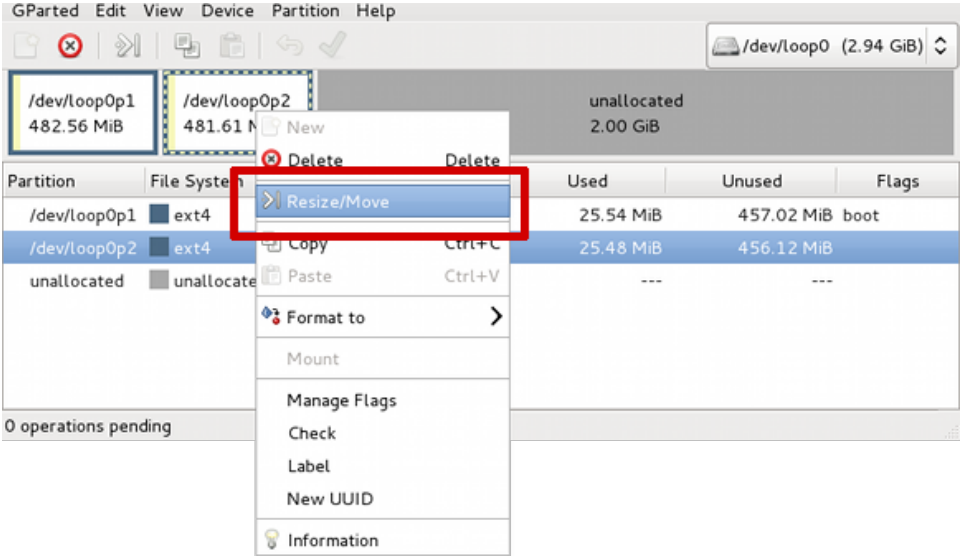

Figure 2.2: Resize/Move `loop0p2`

We're not going to resize the second partition. We just want to make room for the first par-

tition to expand into. Enter `1024` into the *Free space preceding (MiB)* box. That will move this partition to the right far enough to leave the first partition enough room to expand to 1024 MiB. Also, in the *Align to* drop-down menu select *Cylinder* [7] :

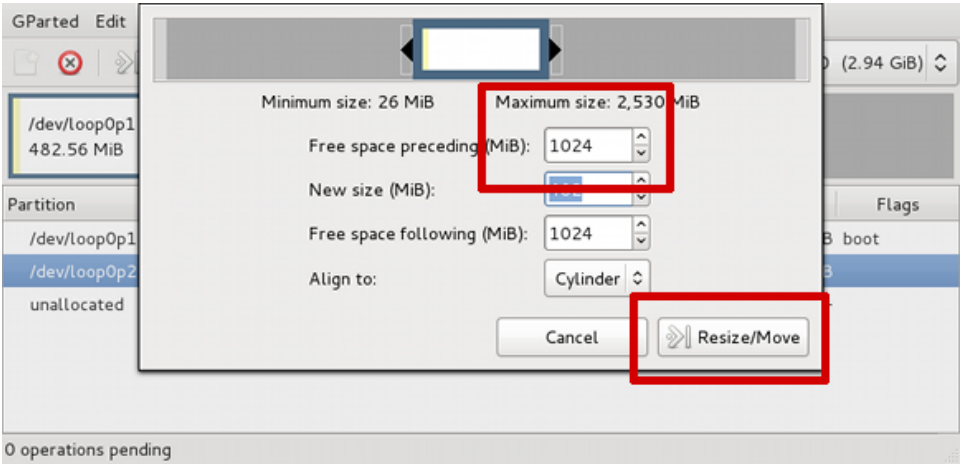

Figure 2.3: Moving `loop0p2`

gparted will now show 1 operation pending:

[7] On aligning Partitions: Section B.3, “The Master Boot Record” [69]

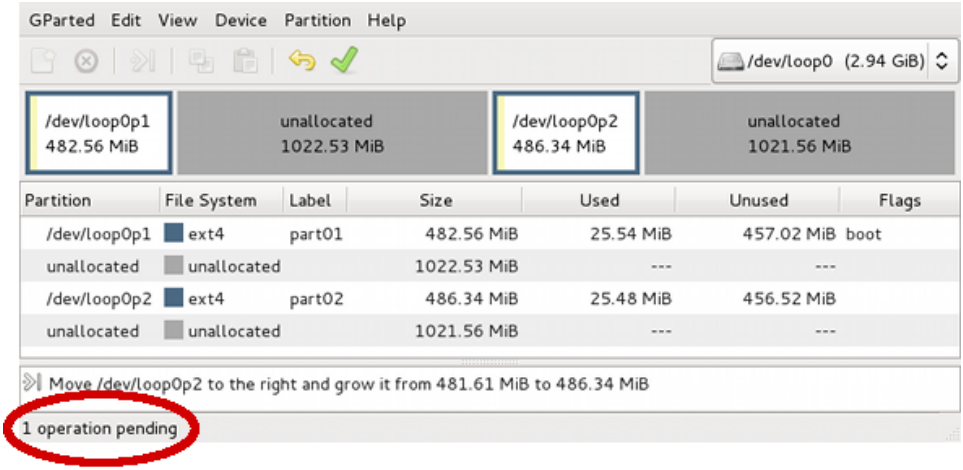

Figure 2.4: Pending move operation

Now right click the first partition and select *Resize/Move* like we did with the second partition. We'll make the first partition use the free space preceding the second partition by setting the *Free space following (MiB)* input box to `0`. Again, in the *Align to* drop-down menu select *Cylinder*:

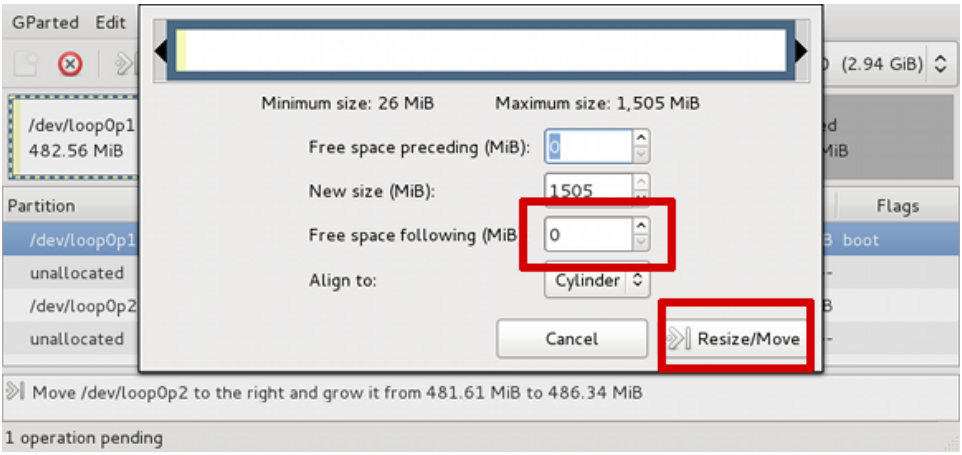

Figure 2.5: Resize `loop0p1`

There is a summary of the two pending actions below the partition table. Click the green check mark button to apply the changes:

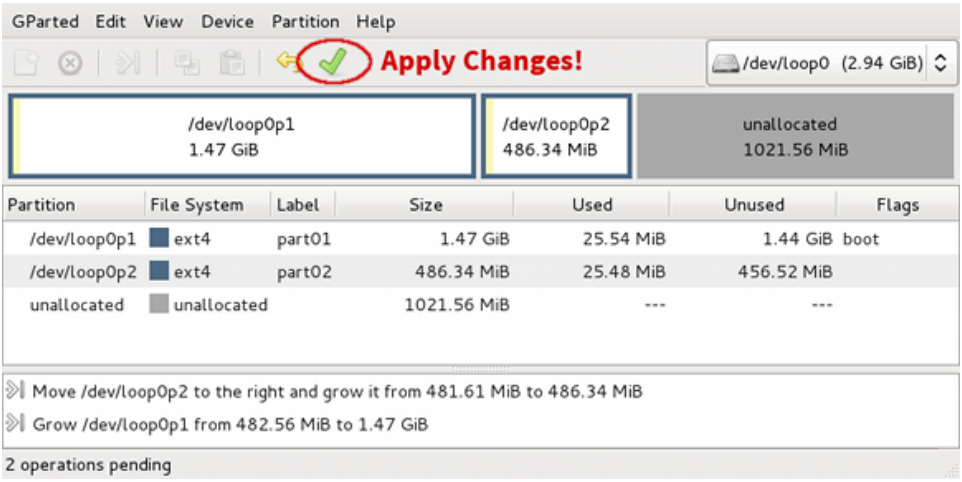

Figure 2.6: Apply the changes

After you click apply you'll get this confirmation dialog:

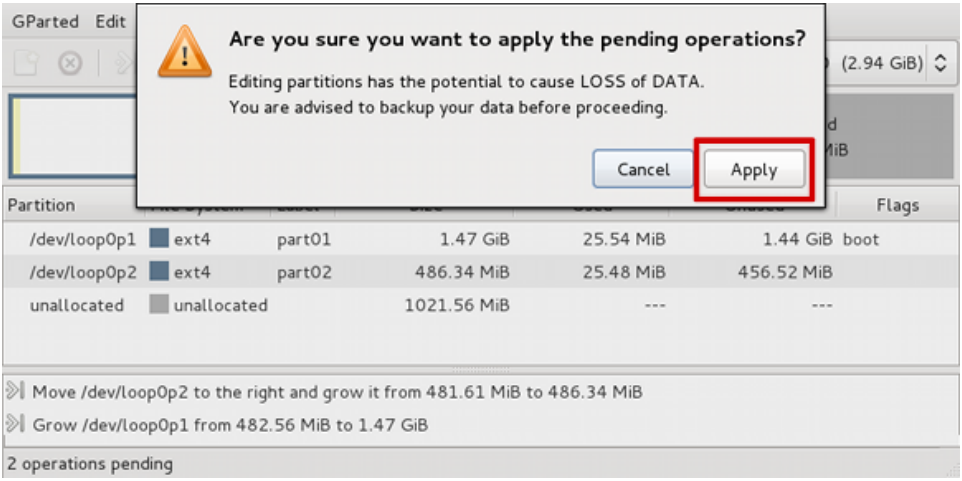

Figure 2.7: Scary warning!

Once you click apply again this window will show the progress:

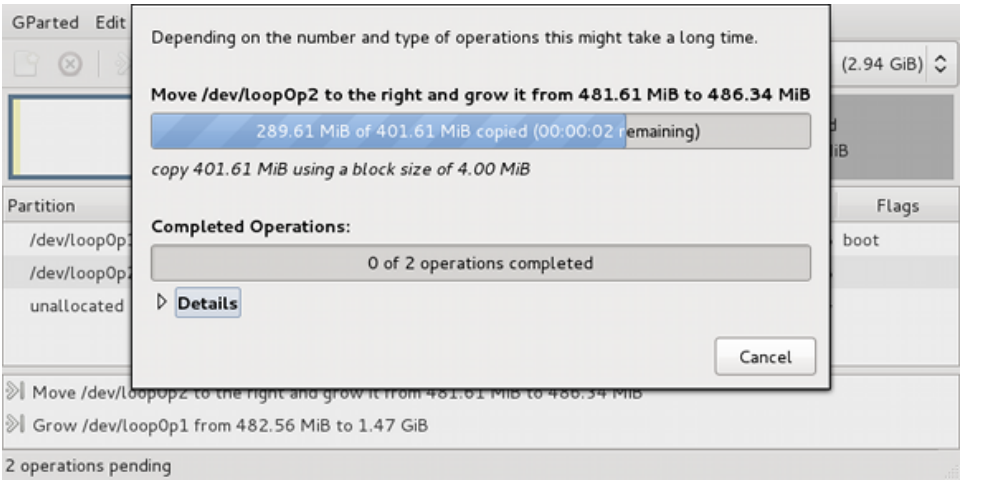

Figure 2.8: Progress happening

You should see this screen if there were no errors:

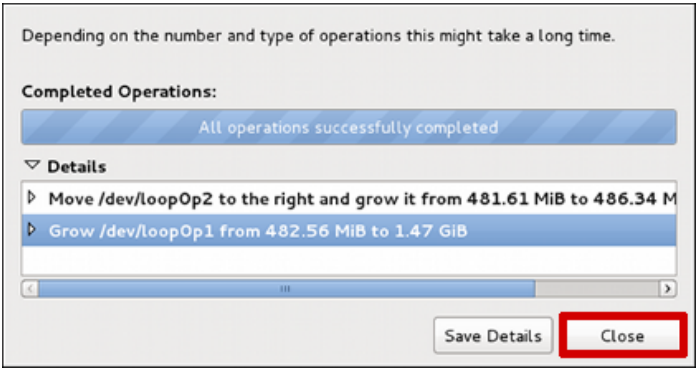

Figure 2.9: No errors!

All done! Click *Close* to return to the main **gparted** screen:

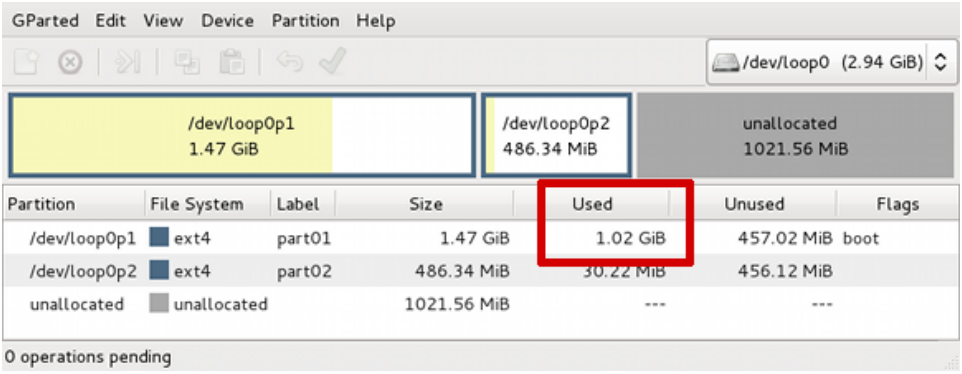

Figure 2.10: **gparted** has resized our partitions

But wait, what's this on the last screen here? **gparted** says `loop0p1` is using 1.02GiB of 1.47GiB. That can't be right. Before resizing the partition **gparted** said `loop0p1` was only using 25.54MiB out of 482.56MiB. Let's take a look at it on the command line:

Example 2.6 Compare **gparted** and **df** output

```
# mount /dev/loop0p1 /mnt/vdg01

# df -h /mnt/vdg01
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/loop0p1   468M   11M  433M   3% /mnt/vdg01

# du -sh /mnt/vdg01
14K     /mnt/vdg01

# umount -l /mnt/vdg01
```

All of that is incorrect too, as if nothing we did in **gparted** had an effect. What's going on here?

After the partitions were resized the partition table was updated with the new information but we never updated the device maps in the kernel. The **kpartx** command also accepts a `-u` option to *update* partitions mappings. Let's try that and see if it fixes our problem:

Example 2.7 Create device maps with **kpartx**

```
# kpartx -uv /dev/loop0
add map loop0p1 (253:8): 0 3082432 linear /dev/loop0 2048
add map loop0p2 (253:9): 0 996030 linear /dev/loop0 3084480
```

The partition sizes and offsets reflect the changes we made, but mounting the first partition still doesn't show the added capacity:

Example 2.8 Still missing added capacity

```
# mount /dev/loop0p1 /mnt/vdg01

# df -h /mnt/vdg01
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/loop0p1  468M   11M  433M   3% /mnt/vdg01
```

We've already resized the partition, but we haven't resized the *filesystem* on the partition. That's the last thing we have to do to finish this whole resizing operation. We'll use the **resize2fs** command and let it automatically resize the filesystem to fill the available space on the `/dev/loop0p1` partition.

Example 2.9 Resize the filesystem with **resize2fs**

```
# resize2fs /dev/loop0p1
resize2fs 1.42.3 (14-May-2012)
Resizing the filesystem on /dev/loop0p1 to 1541216 (1k) blocks.
The filesystem on /dev/loop0p1 is now 1541216 blocks long.

# mount /dev/loop0p1 /mnt/vdg01

# df -h /mnt/vdg01
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/loop0p1  1.5G   11M  1.4G   1% /mnt/vdg01
```

Don't forget to clean up those lingering symlinks we made earlier:

Example 2.10 Cleanup lingering symlinks

```
# rm -f /dev/loop0p[12]
```

Note
The **resize2fs** command can also shrink partitions, print the minimum possible size, and a couple other things. Check **man 8 resize2fs** for more information.

2.2.2 Resizing QCOW2 Images

In this section we'll resize a QCOW2 image, making it 5GB larger. This process will differ from the RAW image resizing section in that we'll do some operations outside of the virtual machine and some operations inside of the virtual machine.

The virtual machine we'll be working with is called `f18`, which is running `Fedora Linux` and has no LVM managed partitions. The disk image for this virtual machine is located at `/var/lib/libvirt/images/f18.qcow2`, and the root partition is `vda3`.

Outside of the virtual machine the disk looks like this:

Example 2.11 Examine `f18.qcow2` on the host

```
# qemu-img info f18.qcow2
image: f18.qcow2
file format: qcow2
virtual size: 12G (12884901888 bytes)
disk size: 4.7G
cluster_size: 65536
```

Inside of the virtual machine the disk and root partition look like this:

Example 2.12 Examine `vda` in the guest

```
# parted /dev/vda print
Model: Virtio Block Device (virtblk)
Disk /dev/vda: 12.9GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system     Flags
 1      1049kB  525MB   524MB   primary  ext4            boot
 2      525MB   4686MB  4161MB  primary  linux-swap(v1)
 3      4686MB  12.9GB  8199MB  primary  ext4
```

```
# df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda3       7.6G  3.8G  3.4G  53% /
```

Warning
Before we begin: make sure you shutdown any virtual machines the disk might be attached to! For example: `virsh shutdown f18`

Once the virtual machine is shutdown the process for resizing QCOW2 images starts similar to the process for resizing RAW images. Use the **qemu-img resize** sub-command, specify the disk to operate on (`f18.qcow2`), and how much to increase the size by (`+5G`):

Example 2.13 Resize a QCOW2 Image

```
# qemu-img resize f18.qcow2 +5G
Image resized.

# qemu-img info f18.qcow2
image: f18.qcow2
file format: qcow2
virtual size: 17G (18253611008 bytes)
disk size: 4.7G
cluster_size: 65536
```

Once you've resized the disk image you can turn the virtual machine back on, for example: `virsh start f18`

Important
The following steps happen *inside* of the running virtual machine.

Once the machine is back online we can resize the partition with the **fdisk** command. Technical note here: when we "resize" the partition with **fdisk** what we're *actually* doing

is deleting the partition and then re-creating it starting at the same position [8] .

Example 2.14 Resize /dev/vda with **parted**

```
# fdisk /dev/vda
Command (m for help): p

Disk /dev/vda: 18.3 GB, 18253611008 bytes, 35651584 sectors
Units = cylinders of 1008 * 512 = 516096 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00020891

   Device Boot      Start         End      Blocks   Id  System
/dev/vda1   *           3        1018      512000   83  Linux
/dev/vda2            1018        9080     4063232   82  Linux  ↩
    swap / Solaris
/dev/vda3            9080       24967     8006656   83  Linux

Command (m for help): d
Partition number (1-4): 3
Partition 3 is deleted

Command (m for help): n
Partition type:
   p   primary (2 primary, 0 extended, 2 free)
   e   extended
Select (default p): p
Partition number (1-4, default 3): 3
First cylinder (9080-35368, default 9080):
Using default value 9080
Last cylinder, +cylinders or +size{K,M,G} (9080-35368, default  ↩
    35368):
Using default value 35368
Partition 3 of type Linux and of size 12.7 GiB is set

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

Calling ioctl() to re-read partition table.
```

[8] While performing research for this section, I found some examples where the **parted resize** sub-command was used. As of **parted** version 2.4 the **resize** subcommand no longer exists.

```
WARNING: Re-reading the partition table failed with error 16:  ↩
    Device or resource busy.
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.
```

Note
In the above example we use the defaults for some of the new partition creation prompts. The defaults work out to selecting the first and last available cylinders, respectively.

Restart the virtual machine again. Now we can see the partition size has increased from 7.6G to 13.6GB:

Example 2.15 New capacity now detected

```
# parted /dev/vda print
Model: Virtio Block Device (virtblk)
Disk /dev/vda: 18.3GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system     Flags
 1      1049kB  525MB   524MB   primary  ext4            boot
 2      525MB   4686MB  4161MB  primary  linux-swap(v1)
 3      4686MB  18.3GB  13.6GB  primary  ext4
```

Just like when we resized the filesystem on the RAW disk image we'll use the **resize2fs** command inside the QCOW2 image. The root partition, `/dev/vda3`, is the last partition on the disk and is followed by free space which we'll grow it into:

Example 2.16 Grow the filesystem on `/dev/vda3`

```
# resize2fs /dev/vda3
resize2fs 1.42.5 (29-Jul-2012)
Filesystem at /dev/vda3 is mounted on /; on-line resizing  ↩
    required
```

```
old_desc_blocks = 1, new_desc_blocks = 1
The filesystem on /dev/vda3 is now 3312304 blocks long.

# df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda3        13G  3.6G  8.3G  31% /
```

2.3 Query an Image for Information

This section is going to show how to query some basic information from a virtual disk. 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. [9]

Example 2.17 Querying an Image

```
$ ls -lhs
total 136K
136K -rw-r-----. 1 tim tim 256K May  8 18:00 image-qcow.qcow2
   0 -rw-r-----. 1 tim tim  10G May  8 18:00 image-raw.raw

$ file image-qcow.qcow2 image-raw.raw
image-qcow.qcow2: Qemu Image, Format: Qcow , Version: 2
image-raw.raw:    data

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

$ qemu-img info image-raw.raw
image: image-raw.raw
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 0
```

[9] The **qemu-img** command manipulates virtual machine disks and is part of the QEMU suite. “QEMU” is a “Quick EMUlator”. It emulates hardware for virtual machines.

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. 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. `image-raw.raw` appears to be 10G but doesn't have any actual blocks allocated to it yet. It is literally an empty file. 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 query which is simple to perform and we can run on any system without special tools. In the last example 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** command.

2.4 Converting Between RAW and QCOW2

2.4.1 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 we have is to convert the image into the QCOW2 format which uses zlib[10] compression and optionally allows your disks to be secured with 128 bit AES encryption[11].

Example 2.18 RAW to QCOW2

```
$ qemu-img convert -O qcow2 original-image.raw image-converted. ↩
    qcow

$ qemu-img info image-converted.qcow
image: image-converted.qcow
file format: qcow2
virtual size: 10G (10737418240 bytes)
disk size: 140K
cluster_size: 65536
```

2.4.2 Convert an Image from QCOW2 to RAW

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

Example 2.19 QCOW2 to RAW

```
$ qemu-img convert -O raw image-converted.qcow image-converted- ↩
    from-qcow2.raw

$ qemu-img info image-converted-from-qcow2.raw
image: image-converted-from-qcow2.raw
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 0
```

[10] From the zlib homepage (`http://zlib.net/`): zlib is designed to be a free, general-purpose, legally unencumbered — that is, not covered by any patents — lossless data-compression library for use on virtually any computer hardware and operating system.

[11] For more information on AES encryption, see *FIPS PUB 197: Advanced Encryption Standard* - `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

Note
When converted to the RAW format the image has the potential to take up much more disk space than before. RAW images may use up up their allocated space immediately, whereas QCOW images will grow to their pre-determined maximum size over time.

2.5 Creating Disks with Backing Images

A few years ago I found out about *backing images* (or "base-images")[13]. Back then I was doing lots of development on host provisioning tools and needed to be able to quickly revert machines I was working on to a desired initial state. In this use case backing images were especially handy when working on features that frequently destroyed the machine if it didn't work right.

Snapshots were an option, but how they worked wasn't documented very well at the time. I went with backing images instead, as they worked perfectly for what I needed them to do. I could work iteratively and *commit* changes in the COW image [12] (copy-on-write) back to the base-image [13] when I was satisfied. I also could use the same base-image for multiple COWs at once. This meant that other people on my team working on the same project could all use the same base-image.

Example 2.20 Creating a Disk with a Backing Image

```
$ mkdir base-images
$ mkdir webserver01
$ cd 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
$ cd ../webserver01
```

[12] In this section when we refer to a *COW* image it is not apropos 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.

[13] The terms "base-image" and "backing image" are used interchangeably

```
$ 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

$ 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)
```

Steps in Detail

1. I consider it bad practice to a bunch of bunch of disk images in a directory so we made two directories 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[14]? Also note that no size is given after the file name. Size is implicitly the size of the disks backing 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.

[14] Some versions of **qemu-img** can not handle *relative* paths)

2.6 Comitting changes to a backing image

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** on the created file.

Important
You should turn off or suspend the virtual machine when running the **commit** command. Failure to do so could result in data corruption.

Example 2.21 Commiting changes

```
# qemu-img create -f qcow2 /srv/base-images/base-image01.qcow2 10 ↩
    G
Formatting '/srv/base-images/base-image01.qcow2', fmt=qcow2 size ↩
    =10737418240 encryption=off cluster_size=65536

# qemu-img create -b /srv/base-images/base-image01.qcow2 -f qcow2 ↩
     /srv/images/with-backing-image.qcow2
Formatting '/srv/images/with-backing-image.qcow2', fmt=qcow2 size ↩
    =10737418240 backing_file='/srv/base-images/base-image01. ↩
    qcow2' encryption=off cluster_size=65536

# qemu-img commit /srv/images/with-backing-image.qcow2
Image committed.
```

2.7 Cloning a Physical Disk

> " Everything in the UNIX system is a file. "
>
> — *The UNIX Programming Environment* - Chapter 2

I never fully grasped the "everything's a file" concept until I tried (*expecting* to fail) to use the **qemu-img convert** sub-command to create a virtual disk image of an actual hard

drive. This is possible in part due to the philosophy laid down by Dennis Ritchie and Ken Thompson when they first created UNIX: everything's treated as a file. The synopsis of the **convert** sub-command is below.

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

In this section we'll look at a standard 1GB USB thumb drive and then clone it into a disk image. Using **parted**, here's what that disk looks like to the host system:

Example 2.22 Thumb Drive Properties

```
# parted /dev/sdb print
Model: Generic Flash Disk (scsi)
Disk /dev/sdb: 1013MB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size   Type     File system  Flags
 1      31.2kB  506MB   506MB  primary               boot
 2      506MB   1013MB  507MB  primary
```

To convert the thumb drive we're first going to unmount the drive, then use the **qemu-img** command to perform the actual conversion. While unmounting the drive I use the `-l` option which means to unmount *lazily*, i.e., to wait until there is no activity going on before attempting to unmount. [15]

Example 2.23 Conversion Steps

```
# umount -l /dev/sdb1
# time qemu-img convert -O raw /dev/sdb ./thumb_drive.raw

real    1m8.206s
user    0m0.161s
sys     0m2.593s
```

[15] See also: Chapter 7, Troubleshooting/FAQs [57]

Chapter 3

Disk Concepts

The best way to learn is by doing, so to learn the concepts of virtual disks we're going to create a 1GiB [1] 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.

Let's start out by looking at what a regular drive looks like to our operating system. Throughout this section the regular drive we'll be comparing our findings against will be a 320G [2] SATA hard drive drive that Linux references as `/dev/sda`. The following example shows some basic information about the device.

Example 3.1 Regular Disk Drive

```
$ file /dev/sda
/dev/sda: block special

$ stat /dev/sda
```

[1] Check out Appendix A, Appendix: Man Pages [64] for a review of binary/decimal prefixes if "GiB" is foreign to you.

[2] If you're wondering why I didn't say 320GiB here, it's because "320GB" is the capacity as defined by the manufacturer.

```
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

$ 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 X3.221-199x - AT Attachment Interface for Disk Drives (ATA-1)* [3] [4] §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[5] already took care of that. The International Electrotechnical Commission [6] defined binary prefixes for

[3] *ANSI X3.221-199x Working Draft*: `http://www.t10.org/t13/project/d0791r4c-ATA-1.pdf`

[4] Technical Committee (T13) Homepage: `http://www.t10.org/t13/`

[5] *IEC 60027-2, Second edition, 2000-11, Letter symbols to be used in electrical technology* - Part 2: Telecommunications and electronics: `http://webstore.iec.ch/webstore/webstore.nsf/artnum/034558`

[6] The IEEE also adopted this method for unit prefixes. Within the IEEE it is known as *IEEE Std 1541-2002*: `http://ieeexplore.ieee.org/servlet/opac?punumber=5254929`

use in the fields of data processing and data transmission. Below are some prefixes as they apply to bytes. See Appendix A, Appendix: Man Pages [64] for the full prefix listing.

Abbrev.	Measurement	Name
1B	= 8 bits	The byte
1KiB	= 1B * 2^{10}	The kibibyte
1MiB	= 1KiB * 2^{10}	The mebibyte
1GiB	= 1MiB * 2^{10}	The gibibyte

3.1 Creating a 1GiB virtual disk from scratch

3.1.1 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).

count = 2^{30} / 2^9 = 1,073,741,824/512 = 2,097,152

EQUATION 3.1: Calculating the Count

We need to fill the file with something 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 zero characters. It actually provides endless NUL control characters(^@ in Caret Notation). The NUL control character has the octal value `000`. The actual ASCII "zero" character has the octal value `060`.

NUL being a control character [7] 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 [8]:

```
$ 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:

```
$ if=/dev/zero bs=1 count=5 2>/dev/null | tr "\0" "\60"
00000
```

3.1.2 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/zero`. Here's the command:

Example 3.2 Running the **dd** command

```
$ 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`.

3.1.3 Examining the Created File

Like in Example 3.1, "Regular Disk Drive" [30] let's take a look at the file we created from the operating system's point of view.

[7] Wikipedia.org - Control Characters: `http://en.wikipedia.org/wiki/Control_code`
[8] Wikipedia.org - Caret Notation: `http://en.wikipedia.org/wiki/Caret_notation`

Example 3.3 Examining the Created File

```
$ 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

$ file disk1.raw
disk1.raw: data

$ 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/tim)   Gid: (  500/tim)
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

$ 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 that `disk1.raw` has in common with the actual disk drive `sda`. Using this information, let's put the physical disk and the virtual disk size-by-size and make some observations about their properties.

- **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 knows how big it is, nor can it find any partition information on it.

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

Command	sda	disk1.raw
file	block special	data
stat	block special	regular file
fdisk	Contains partition table	Missing partition table

Table 3.1: Attribute Comparison

3.1.4 Create a Partition Table

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

Example 3.4 Create a Partition Table

```
$ parted disk1.raw mklabel msdos
WARNING: You are not superuser.  Watch out for permissions.
```

Let's examine the image again to see how the operating system thinks it has changed.

Example 3.5 Overview - What Changed

```
$ file disk1.raw
disk1.raw: x86 boot sector, code offset 0xb8

$ 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/tim)   Gid: (  500/tim)
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

$ 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
```

- Now, instead of “data”, the **file** command thinks it is an “x86 boot sector”. That sounds pretty accurate as 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 the boot sector which **file** found.

Command	**`sda`**	**`disk1.raw`**	**`disk1.raw` (via 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

Table 3.2: What **parted** Changed

3.2 Devices and Partitions

3.2.1 Introduction

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

```
$ fdisk disk1.raw
You must set cylinders.
You can do this from the extra functions menu.

Command (m for help):
```

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

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 system just 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. 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.

3.2.2 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.

To create a loop device run the **losetup** command with the `-f` option. The first available loop device will be selected automatically and associated with `disk1.raw` [9].

Example 3.6 Creating a loop device with **losetup**

```
$ sudo losetup -f disk1.raw

$ sudo losetup -a
/dev/loop1: [0805]:151552 (/home/tim/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**.

[9] FUSE (Filesystem in Userspace) has a module called `Mountlo` that allows non-root users to make make loop devices.

3.2.3 Examine the loop device

Example 3.7 Examining the Loop Device

```
$ file /dev/loop0
/dev/loop0: block special

$ 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

$ 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 3.1, “Regular Disk Drive” [30] 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.

Our virtual disk is starting to look like a real hard drive now! To conclude this section we’ll:

- create a partition
- format it with an ext3 filesystem
- mount it for reading and writing

Command	sda	disk1.raw	disk1.raw (via parted)	/dev/loop0
file	block special	data	x86 boot sector	block special
stat	block special	regular file	regular file	block special
fdisk	has partition table	no partition table	valid partition table. unknown cylinder count	valid partition table. known cylinder count

Table 3.3: Examining the Loop Device

3.2.4 Creating partitions

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

Example 3.8 Creating a partition with **fdisk**

```
$ 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.

$ 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
```

3.2.5 Formatting Partitions

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

Q: *What is an `offset`, and why do we have to specify one?*

A: An `offset` indicates how far from the beginning of a device something is. The first partition is not located at the beginning of the device. That is where the Master Boot Record (MBR) is stored (`offset=0`). If we tried to create a partition at `offset=0` we would overwrite the MBR. Knowing the offset will allow us to create a device mapped to where the first partition begins without overwriting the MBR. Linux does this automatically for regular disks during the boot process.

Q: *How do we calculate the offset to specify?*

A: To calculate the offset we 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`).

Q: *Why doesn't the first partition begin after the MBR? Specifically, why is there empty space between the first sector (where the MBR is stored) and the first partition?*

A: It's complicated but worth learning about. See Appendix B, Appendix: Disk Drive History [66] for a complete explanation. Here's the short answer: In current PC MBRs there

may be up to 446B of executable code and a partition table containing up to 64B of data. When you add in another 2B to record a *Boot Signature* you have 512B, which up until recently happened to be the typical size of one sector. Partitioning tools historically left the space between the MBR and the second cylinder empty. Modern boot loaders (NTLDR[10], GRUB[11], etc) use this space to store *additional* code and data necessary to boot the system [12] [13]. Some software, such as licensing managers and virus scanners, also use this space to store files [14] .

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

```
$ sudo fdisk -u -l /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.

```
$ sudo losetup -o 32256 -f /dev/loop0

$ sudo losetup -a
/dev/loop0: [0805]:151552 (/home/tim/images/disk1.raw)
/dev/loop1: [0005]:5102 (/dev/loop0), offset 32256
```

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

[10] NT Loader (NTLDR): `http://en.wikipedia.org/wiki/NTLDR`

[11] The Grand Unified Bootloader (GRUB): `http://www.gnu.org/software/grub/`

[12] GRUB: BIOS Installation: `http://www.gnu.org/software/grub/manual/grub.html#BIOS-installation`

[13] Simon Kitching: Booting Linux on x86 using Grub2: `http://moi.vonos.net/linux/Booting_Linux_on_x86_with_Grub2/#installing-grub`

[14] Ubuntu Forums - Sector 32 FlexNet Problem -- Grub: `http://ubuntuforums.org/showthread.php?t=1661254`

Example 3.9 Formatting and mounting the partition

```
$ 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.

$ sudo losetup -d /dev/loop1

$ sudo losetup -d /dev/loop0

$ mkdir partition1

$ sudo mount -t ext3 -o loop,offset=32256 disk1.raw partition1/

$ mount | grep partition1
/dev/loop0 on /home/tim/images/partition1 type ext3 (rw,offset ↩
    =32256)

$ df -h partition1/
Filesystem            Size  Used Avail Use% Mounted on
/dev/loop0           1008M   18M  940M   2% /home/tim/images/ ↩
    partition1
```

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

3.2.6 Cleaning Up

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

Example 3.10 Detaching a loop device

```
$ sudo losetup -d /dev/loop1
```

Chapter 4

Helper Utilities

Up until now most of the commands we've been using have been very low-level. Just the section on resizing images [1] is about 8 pages of this book (depending on what format you're reading it in). Let's get real here: it's not pragmatic to run ten commands when one or two will suffice. Luckily for us some very helpful utilities exist.

This section will introduce those utilities. I'll highlight some key features in each, show demos, and tell you where you can find more information. Let's get started by introducing our new heros using their official descriptions.

libguestfs
: `libguestfs` is a way to create, access and modify disk images. You can look inside disk images, modify the files they contain, create them from scratch, resize them, and much more. It's especially useful from scripts and programs and from the command line.

virt-manager
: The "Virtual Machine Manager" application (virt-manager for short package name) is a desktop user interface for managing virtual machines. It presents a summary view of running domains, their live performance & resource utilization statistics. The detailed view graphs performance & utilization over time. Wizards enable the creation of new domains, and configuration & adjustment of a domain's resource allocation & virtual hardware. An embedded VNC client viewer presents a full graphical console to the guest domain.

[1] Section 2.2, "Resizing Disk Images" [8]

4.1 libguestfs

`libguestfs` make managing virtual disks (and machines) a lot simpler. Included is a C library (with bindings available for Perl, Python, Ruby, Java, OCaml, PHP, Haskell, Erlang, Lua and C#), as well as a collection of 34 utilities (at the time of writing).

I won't even attempt to cover all of it's features in this book. Instead, I'll go over some of the most useful utilities. For more information on `libguestfs` you should go to the project website [2] where they have a complete 250 page manual fully describing all aspects of `libguestfs`.

4.1.1 guestmount

> The guestmount program can be used to mount virtual machine filesystems and other disk images on the host. It uses libguestfs for access to the guest filesystem, and FUSE (the "filesystem in userspace") to make it appear as a mountable device.
>
> —man 1 guestmount

foo

4.1.2 virt-filesystems

> This tool allows you to discover filesystems, partitions, logical volumes, and their sizes in a disk image or virtual machine.
>
> —man 1 virt-filesystems

virt-filesystems is the Sherlock Holmes [3] of virtual disk management. What delights me most about **virt-filesystems** is how well it integrates with LVM (Logical Volume Manager) to show you LVM device paths. This tool is most useful in combination with other tools, such as **virt-resize**, **virt-sparsify**, or **guestmount**.

[2] `libguestfs` homepage: `http://libguestfs.org/`

[3] Sherlock Holmes is a fictional detective. Read some of the books online for free on Project Gutenberg: `http://www.gutenberg.org/ebooks/1661`

4.1.3 virt-rescue

> virt-rescue is like a Rescue CD, but for virtual machines, and without the need for a CD. virt-rescue gives you a rescue shell and some simple recovery tools which you can use to examine or rescue a virtual machine or disk image.
>
> —man 1 virt-rescue

4.1.4 virt-resize

> **virt-resize** is a tool which can resize a virtual machine disk, making it larger or smaller overall, and resizing or deleting any partitions contained within.
>
> —man 1 virt-resize

4.1.5 virt-sparsify

> **virt-sparsify** is a tool which can make a virtual machine disk (or any disk image) sparse a.k.a. thin-provisioned. This means that free space within the disk image can be converted back to free space on the host.
>
> —man 1 virt-sparsify

Depending on your deployment strategy the **virt-sparsify** command could potentially save you a lot of disk space. This is especially the case in "cloud"-type setups where new machines are commonly created from a single "golden-master" image. If you're making copies of *any* disk image then you need to make sure that you aren't unnecessarily wasting space on your disk.

That's where **virt-sparsify** comes in. When you *sparsify* a disk image (or any other file for that matter) you're potentially reducing the number of blocks on the backing storage volume [4] which are allocated to the disk image. This frees up room on the backing volume for storing other files. Sparsifying a disk image is only effective as there is space that can be freed. More on that next.

Important
Sparsifying files doesn't divorce you from the inherent size limitations of your backing storage volume. I.e., you can not expect to fill two 100GiB sparse disk images with data if the volume they're stored on is only 50GiB.

[4] The phrase "backing storage volume" refers to the actual storage device which the disk image is saved on.

In this example I'll sparsify the disk image we cloned from a thumbdrive earlier in the book [5]. Let's start by using the **qemu-img info** subcommand [6] to see how much total space the disk image has allocated to it presently:

```
# qemu-img info ./thumb_drive.raw
image: ./thumb_drive.raw
file format: raw
virtual size: 966M (1012924416 bytes)
disk size: 914M
```

The output you want to take note of here is on the `disk size` line: 914M. For a thumb drive that only had two small text files on it, we sure are wasting a lot of space. Let's attempt to reclaim that space with **virt-sparsify**. We'll call **virt-sparsify** with two parameters, the first is the source disk name, `thumb_drive.raw`, and the second is the name of the sparsified disk image we're going to create, `thumb_drive_sparse.raw`:

Example 4.1 Sparsify a disk image

```
# virt-sparsify ./thumb_drive.raw ./thumb_drive_sparse.raw
Create overlay file to protect source disk ...
Examine source disk ...
 100% [...] 00:00
Fill free space in /dev/vda1 with zero ...
 100% [...] --:--
Fill free space in /dev/vda2 with zero ...
 100% [...] --:--
Copy to destination and make sparse ...

Sparsify operation completed with no errors.  Before deleting the ↩
     old
disk, carefully check that the target disk boots and works  ↩
    correctly.
```

As you can see, the output from **virt-sparsify** is straightforward and easy to grok [7]. Right away we can tell that **virt-sparsify** is protecting our source file from undesired modifications by creating an "overlay file". Next it examines the disk, identifying entities such as partition tables, LVM volumes, and space to be freed. Then the freeable space is zeroed out.

[5] Example 2.23, "Conversion Steps" [29]

[6] We could also use the **ls -lsh** command. The `-s` option prints the *allocated size* (actually used space), and the `-h` options prints sizes in "human readable" formats, e.g., 915M or 4.0K

[7] *grok* - verb - "To understand. Connotes intimate and exhaustive knowledge". Source: `http://www.catb.org/jargon/html/G/grok.html`

Remember that we haven't modified the source image yet! All of the potential changes were made to an overlay file. The final step to sparsify the file is combining the delta present in the overlay file with the source file and writing the result out to disk. Observe the following important note from the **virt-sparsify** man page:

Important
virt-sparsify may require up to 2x the virtual size of the source disk image (1 temporary copy + 1 destination image). This is in the worst case and usually much less space is required.

Let's use **qemu-img info** again and examine the sparsified disk image (`thumb_drive_sparse.raw`). Recall that we're primarily concerned with the `disk size` field and that the starting size was 914M:

```
# qemu-img info ./thumb_drive_sparse.raw
image: ./thumb_drive_sparse.raw
file format: raw
virtual size: 966M (1012924416 bytes)
disk size: 6.2M
```

From this we can see that after the image was sparsified the allocated space is only 6.2M. That's a net savings of 907.8M! Don't let this result give you unreasonable expectations though. This example demonstrated an ideal case, where the source disk was virtually 100% empty to begin with.

virt-sparsify has other options available as well. For example, it can convert between formats (e.g., vdmk to qcow2), ignore specific filesystems on the source disk, or enable compression in qcow2 images. Read the man page for a complete description of all the available options.

4.2 virt manager

Up until now most of the commands we've been using have been very low-level. Just the section on resizing images [8] is about 8 pages of this book (depending on what format you're reading it in). Let's get real here: it's not pragmatic to run ten commands when one or two will suffice. Luckily for us some very helpful utilities exist.

[8] Section 2.2, "Resizing Disk Images" [8]

Chapter 5

Disk Formats

In this chapter we'll review some of the formats available for virtual disks. Along the way we'll discuss features of each format, performance options (tunables), and use case considerations.

5.1 RAW

Words to introduce the feature set.

- Simple
- Exportable
- Supports sparse files

Words about performance and use-cases.

5.2 QCOW

Words to introduce the feature set. http://people.gnome.org/~markmc/qcow-image-format-version-1.html

Words about performance and use-cases.

- Smaller file size, even on filesystems which don't support holes (i.e. sparse files)
- Snapshot support, where the image only represents changes made to an underlying disk image
- Optional zlib based compression
- Optional AES encryption
- Superseded by QCOW2

5.3 QCOW2

Words to introduce the feature set. http://people.gnome.org/~markmc/qcow-image-format.html

Words about performance and use-cases.

- Smaller file size, even on filesystems which don't support holes (i.e. sparse files)
- Copy-on-write support via *backing images*, where the image only represents changes made to an original separate disk image
- Snapshot support, where the image can contain multiple snapshots of the images history
- Optional zlib based compression
- Optional AES encryption
- Options for performance/data integrity tuning

5.4 Other Formats

In addition to the formats we've already reviewed, QEMU has varying levels of support for several other disk image formats. See the documentation[1] for a complete description of their supported options.

The following formats are supported by QEMU in a *read-write* mode:

[1] QEMU User Docs: 3.6.6 Disk image file formats - `http://qemu.weilnetz.de/qemu-doc.html#disk_005fimages_005fformats`

qed
: Old QEMU image format with support for backing files and compact image files (when your filesystem or transport medium does not support holes).

cow
: User Mode Linux Copy On Write image format. It is supported only for compatibility with previous versions.

vdi
: VirtualBox 1.1 compatible image format.

vmdk
: VMware 3 and 4 compatible image format.

vpc
: VirtualPC compatible image format (VHD).

The following formats are also supported by QEMU in a *read-only* mode:

bochs
: Bochs images of *growing* type.

cloop
: Linux Compressed Loop image, useful only to reuse directly compressed CD-ROM images present for example in the Knoppix CD-ROMs[2].

dmg
: Apple disk image.

parallels
: Parallels disk image format.

[2] KNOPPIX: bootable Live Linux system on CD/DVD - `http://www.knopper.net/knoppix/index-en.html`

Chapter 6

Performance Considerations

Managing disk images doesn't stop at file manipulation and storage pool monitoring. After you create a disk image something else is going to use it. That's where performance tuning considerations come into play. This section straddles the line between system administrator and application developer roles. What I mean to say is that application of some techniques in this section may require knowledge which is outside of your domain as a system administrator. To help bridge the knowledge gap I'll include notes on how to identify what you're looking for when tuning the system.

Many performance tuning decisions come down to one question: *In the event of catastrophic system failure, how expensive is it to replace the data?* If that value is low you can reach higher levels of performance at the cost of higher risk of data loss. If that value is high you can reach greater levels of data integrity at the cost of performance.

In this section we'll cover the following topics:

- Selecting the right disk caching mode
- Selecting the right virtual device
- Selecting the right I/O scheduler
- Balancing resources with cgroups

You may also be interested in reading over Chapter 5, Disk Formats [49].

6.1 I/O Caching

I/O caching requirements differ from host to host. I/O caching refers to the *mode* (or *write policy*) by which the kernel writes modified data to both the cache and the caches backing store. There are two general modes to consider, *write-back* and *write-through*. Let's review them now:

Write-back
: Writes are done *lazily*. That is, writes initially happen in cache, and then are propagated to the appropriate backing storage device. Also known as *write-behind*.

Write-through
: Writes are done synchronously to cache and the backing store (main system memory/disk drive).

Selecting the correct cache mode can increase or decrease your overall system performance. Selecting the correct mode depends on several factors, including:

- Cost of data loss
- System latency vs. throughput requirements
- Operating System support
- Hypervisor feature support
- Virtualization deployment strategy

In addition to write-back and write-through modes there is a third pseudo-mode called *none*. This mode is considered a write-back mode. In this mode the onus is on the guest operating system to handle the disk write cache correctly in order to avoid data corruption on host crashes[2]. In a supported system where latency/throughput are valued over data integrity you should consider choosing the "none" mode[1] Next we'll review the two cache mode options in greater detail. At the end of the chapter we'll summarize the use cases for each mode.

[1] https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Tuning_and_Optimization_Guide/chap-Virtualization_Tuning_Optimization_Guide-BlockIO.html

6.1.1 Write-back Caching

Write-back caching means that as I/O from the virtual guest happens it is reported as *complete* as soon as the data is in the virtual hosts page cache[2]. This is a shortcut around the I/O process wherein the data is written into the systems cache and then subsequently written into the backing storage volume. Whether that volume be volatile system memory (such as ram), or a non volatile source (such as a disk drive). In write-back caching the new data is not written to the backing store until a later time.

I remember the phrase write-back by thinking of it like this: "As soon as a *write* happens on the guest a response is sent *back* to indicate that the operation has 'completed'."

Using write-back caching will have several side-affects:

PRO: Increased performance
: Both the guest and host will experience increased I/O performance due to the lazy nature of cache-writes.

CON: Increased risk of corruption
: Until the data is `flush`'d there is an increased risk of data corruption/loss due to the volatile properties of system cache.

CON: Doesn't support guest migrations
: You can not use the guest migration hypervisor feature if you are using write-back cache mode.

CON: Not supported by all Operating Systems
: Not all OSs may support write-back cache. For example, RHEL releases prior to 5.6[3].

Though the CONs out-number the PROs, In reality, write-back is not as dangerous as it may appear to be. The QEMU User Documentation[2] says the following:

> By default, the `cache=writeback` mode is used. It will report data writes as completed as soon as the data is present in the host page cache. This is safe as long as your guest OS makes sure to correctly flush disk caches where needed. If your guest OS does not handle volatile disk write caches correctly and your host crashes or loses power, then the guest may experience data corruption.

[2] QEMU User Docs: `-drive` options - http://qemu.weilnetz.de/qemu-doc.html#sec_005finvocation

[3] https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Tuning_and_Optimization_Guide/chap-Virtualization_Tuning_Optimization_Guide-BlockIO.html

If your guest is ineligible for the “none” mode, because it doesn’t manage its disk write cache well, then write-back mode is a great secondary option.

6.1.2 Write-through Caching

Write-through caching means that modified data is written synchronously into both system cache, as well as a backing store (RAM/disk drive). Because the writing happens in the backing store as well, write-through cache introduces a performance hit.

Because write-through caching puts a larger load on the host it is best used in moderation. You should avoid enabling write-through caching on a host with many guests, as this configuration is prone to scaling issues. You should only consider enabling write-through caching in situations where data integrity is paramount above all else or where write-back caching is not available on the guest.

6.2 I/O Schedulers

Scheduling algorithms, sometimes referred to as *elevators*[4], are methods used by the operating system to decide the order in which block I/O operations (read/write) take place. Different algorithms exist because no single one is best suited for all workloads.

A database server would want to prioritize latency over throughput, thus making the deadline scheduler an ideal choice, whereas in an interactive desktop you would favor the CFQ (“completely fair queueing”) scheduler. Workload isn’t the only parameter to consider when selecting a scheduler. The properties of the backing storage device also play an important role (SSD or spinning disk?). In a virtualized environment the choice of scheduler becomes even more involved because you may wish to consider the scheduler used by the hypervisor as well.

As you can see, the topic of selecting the proper I/O scheduler is neither short, nor is it simple. That being said, in this chapter I’ll attempt to provide you with sufficient information to make an informed decision as well as several resources which discuss I/O schedulers in greater detail. Together we’ll review the scheduler options available, the procedure for setting one permanently, and typical use cases.

This chapter is incomplete. Please come back for updates.

[4] TODO: Explain why

6.2.1 Additional Resources

White papers:

- *Does Virtualization Make Disk Scheduling Passé?* - `http://www-users.cs.umn.edu/~chandra/papers/hotstorage09/paper.pdf`
- *On Disk I/O Scheduling in Virtual Machines* - `http://sysrun.haifa.il.ibm.com/hrl/wiov2010/papers/kesavan.pdf`
- *Understanding the Effects of Hypervisor I/O Scheduling for Virtual Machine Performance Interference* - `http://www.seas.gwu.edu/~howie/publications/CloudCom12.pdf`
- *I/O Scheduling for SAN and Virtualization* - `http://www.monperrus.net/martin/IO+scheduling+for+san+and+virtualization`

Performance tests:

- *Ceph Bobtail Performance – IO Scheduler Comparison* - `http://ceph.com/community/ceph-bobtail-performance-io-scheduler-comparison/`
- *I/O Scheduler Comparison On The Linux 3.4 Desktop* - `http://www.phoronix.com/scan.php?page=article&item=linux_iosched_2012&num=1`

Chapter 7

Troubleshooting/FAQs

Q: *Why are my cloned disks so big, I thought QCOWs would be smaller if my disk was mostly empty* ? [1]

A: Creating a disk image from a device copies *all* blocks from the source device. This includes data which has been deleted on the filesystem. When you delete a file from the filesystem the operating system will not signal to the disk that it should mark the formerly occupied blocks as free [2] . The additional overhead associated with the operation would hurt disk performance. What option do you have available if you want to minimize the size of the created disk image? You have two options, a free utility called zerofree [3] , and **virt-sparsify**. I refer you to Section 4.1.5, "virt-sparsify" [46] for more information on **virt-sparsify**.

Q: *Why do I get a* `device busy` *error message when unmounting* `$THING`*?*

A: A process is accessing files on the mounted volume. Possible fixes:

- Sometimes the solution is a simple as *lazily* unmounting the device. Do this by giving the `-l` option to **umount**.
- Make sure you don't have any open shells whose present working directory is in the path you're trying to unmount.

[1] This character "?" is called the *interrobang*. I just blew your mind.
[2] This is what allows data recovery software to work
[3] zerofree homepage: `http://intgat.tigress.co.uk/rmy/uml/sparsify.html`

- If that doesn't work you can try using the **fuser** command to find what processes are accessing the device. For example: `fuser /mnt/thumbdrive`. This command also accepts an optional −k option, which will try to kill all processes accessing the busy path.
- If none of that works you can try the **lsof** command (superuser permissions required to see *everything* being accessed). For example: `lsof | grep /mnt/path`.

Chapter 8

Glossary

AES encryption

Advanced Encryption Standard - very fast and secure; the de facto standard for symmetric encryption. See Also ”zlib compression”.

ASCII

American Standard Code for Information Interchange. It is a 7-bit code. ASCII encodes characters as you would enter them into a computer (like this book)

Backing image

A (typically) read-only disk image which can be used as a starting point for new read-write images. See Also ”Snapshot”.

Base-image

Placeholder. See Also ”Backing image”, ”Snapshot”.

block

block special

Caret notation

cat

A utility program which concatenates files and print them the standard output.

Control character

cylinder

dd

A utility program which can copy files, converting and formatting them according to the options given by the user.

Device map

Software which creates devices from partition tables which you can interact with. See Also ”Partition table”, ”kpartx”, ”GParted”, ”Partition”.

dev null

dev zero

fdisk

A utility program which manipulates disk partition tables. See Also ”Partition”, ”Partition table”.

file

A utility program which is used to determine file types

Filesystem

fuse

GParted

A graphical application used for manipulating (creating, resizing, moving, copying) the filesystems of partitions.

Guest OS

An operating system which is installed and ran on emulatd, virtual, or paravirtual hardware which is managed by hypervisor software on the Host OS. See Also ”Hypervisor”, ”Host OS”.

head

Host OS

The running system (server, OS) which provides resources and facilities for running several virtual Guest Operating Systems. See Also ”Guest OS”.

Hypervisor

Software blabla.

IDE

Image

A file which virtualization software can use as a hard disk, similar to a snapshot. See Also ”Snapshot”.

kpartx

Reads partition tables on a specified device and create device maps over partition segments detected. See Also ”Partition”.

Lookback device

Loop device

losetup

A utility program which sets up and controls loop devices. See Also ”Loop device”, ””.

ls

A command which lists directory contents and file attributes.

LVM

MBR

The *Master Boot Record* holds the information on how the logical partitions, containing file systems, are organized on a storage device. See Also ”Partition”, ”Partition table”.

meta-data

Data which describes other data; e.g., virtual disk configuration parameters.

mount

A utility program which attaches a filesystem to a directory tree. See Also ”umount”, ”Filesystem”.

NUL

offset

OS

Short for *Operating System*.

parted

Utility program which manipulates storage partitions See Also ”fdisk”.

Partition

In storage devices, the definition of storage allocation on a device; the capacity of that region is less than or equal to that of the backing storage device; multiple partitions may exist.

Partition table

Meta-data stored on a storage volume which describes the partition layout, i.e., begin/end locations, types, and other properties. See Also ”Device map”, ”kpartx”, ”GParted”, ”Partition”, ”meta-data”.

QCOW2

QEMY Copy On Write image format (version 2); improves v1 with few features: snapshots, performance tuning options.

QCOW

QEMU Copy On Write image format (version 1); supports sparse files, backing files, and encryption.

qemu-img

Virtual disk manipulation tool bundled with the QEMU (Quick Emulator) software collection.

RAW

The simplest type of virtual disk format, as the file contains no extra meta-data about itself, often usable without requiring special software. See Also ”QCOW”, ”QCOW2”, ”meta-data”.

resize2fs

Utility program which can resize ext2, ext3, or ext4 file systems.

SATA

sector

Snapshot

An virtual disk feature representing a moment in time, not represented as a disk image. See Also ”Backing image”, ”Base-image”.

socket

superuser

symlink

UNIX

umount

Utility program which detaches the file system(s) mentioned from the file hierarchy. See Also ”mount”.

virsh

x86 boot sector

zlib compression

general-purpose, patent-free, lossless data compression library. See Also ”AES encryption”.

Appendix A

Appendix: Man Pages

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".

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=bit, B=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"

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

Table A.1: Binary Prefixes

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.

Appendix B

Appendix: Disk Drive History

Disk drives, and how they are accessed, is a broad subject which has changed greatly over time. Some "facts" are actually just misconceptions which are taken as canon. This section will attempt to sort the facts from fiction and give some sort of historical account of how the software and hardware has changed over time.

B.1 Disk Drive Components

In the early days of computing, direct access storage devices (i.e., "hard disk drives") were much simpler. A simple device meant a less complex method for interaction was necessary. Two standards define how communication with disk drives may happen: The IDE/ATA standard for communicating with disk drives, and the BIOS Int 13h standard ("disk services") for how operating systems can interact with disk drives via software interrupts [1] [2].

A disk drive was originally composed of a few simple components:

- One called a *head* which is mounted on a swinging arm. The arm swings across a disk platter to move the head to the sector requested for a read or write operation. More platters in a disk drive mean more heads and arms.

[1] BIOS Enhanced Disk Drive Specification v3: `http://www.t10.org/t13/technical/d98120r0.pdf`

[2] PC Guide - Int 13h: `http://www.pcguide.com/ref/hdd/bios/bios_Int13h.htm`

- An array of magnetized spinning disks called *platters*. Because each side of a platter is used to store data there must be two heads for each platter.
- For the purpose of addressing a specific location on a platter each platter is further broken down into *cylinders* (or *tracks*), and *sectors*.

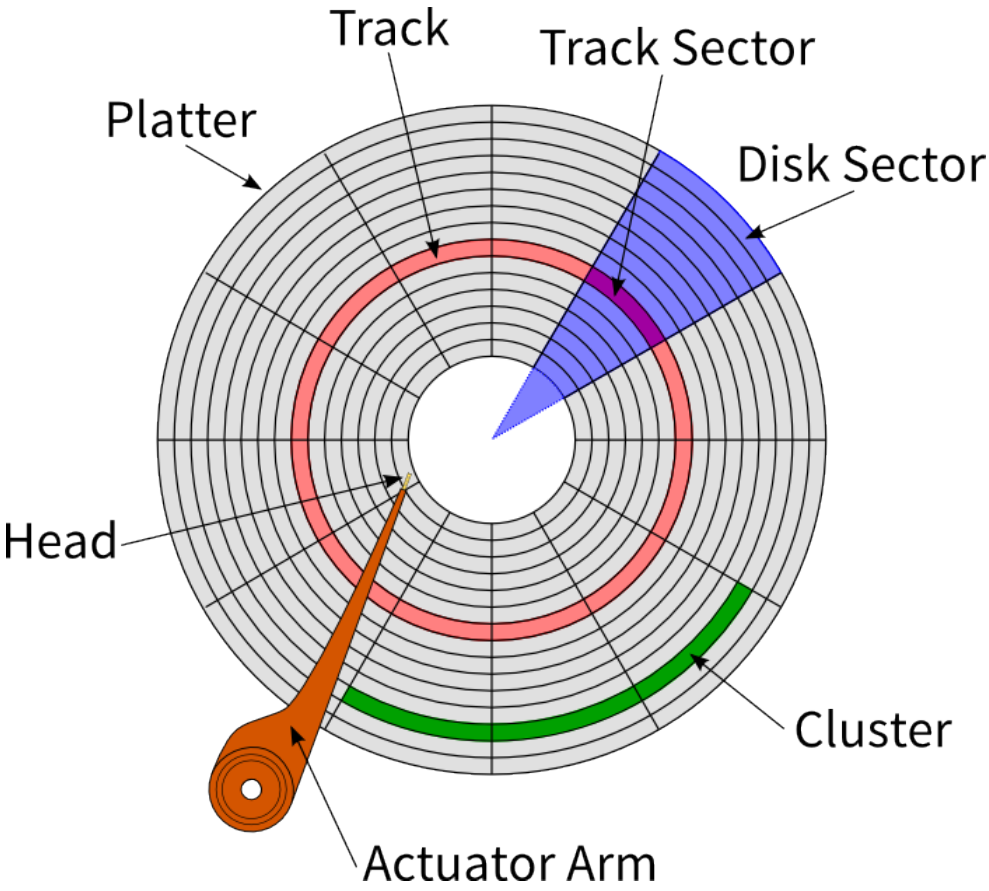

Figure B.1: Disk Drive Components

B.2 Access Modes

Addressing data blocks can be done in multiple ways. The older ways (CHS, ECHS) operate in terms of physical disk properties (geometry). The second system for addressing blocks (LBA) has been an option in almost every disk drive since 1996 [3].

Note
Later we'll see the problems caused by the radically different and conflicting way the ATA/`Int 13h` standards are defined.

B.2.1 CHS Addressing

In the beginning data on disk drives was addressed by describing the physical geometry of the disk using a combination of its distance from the center of the disk (*track*), its rotation around the disk (*sector*), and the read-write head which accesses its side of the platter. This addressing system is called *Cylinder-Head-Sector* (CHS) [4]. This method of access was provided via a BIOS service commonly referred to as `Int 13h`. While this system was quite straightforward, it provided no abstraction between the physical location of data and the act of requesting data from the drive. To read/write data you simply called `Int 13h` and specified the physical cylinder, head, and sector on the disk drive of what you were requesting. It began breaking down when drive capacities exceeded what the standards at the time were capable of describing. You can think of this like running out of telephone numbers.

One way this was addressed was through the *Int 13h Extensions*. The original `Int 13h` system used 24 bits for addressing data, the extensions bumped that number up to 64 bits. To put that into perspective, the maximum addressable range of data went from 8.46 GB up to 9,400,000,000,000 GB [5].

At the same time this was happening, technology was advancing to the point where it was becoming logically impossible to represent the physical drive geometry to the BIOS in a

[3] Wikipedia.org - Logical Block Addressing: `http://en.wikipedia.org/wiki/Logical_block_addressing#Enhanced_BIOS`

[4] PC Guide - Cylinder-Head-Sector: `http://www.pcguide.com/ref/hdd/geom/geom.htm`

[5] PC Guide: `Int 13h` Extensions `http://www.pcguide.com/ref/hdd/bios/bios_Extensions.htm`

way compatible with the ATA/`Int 13h` systems [6]. To work around this, disk drives began reporting their *Logical Geometry* to the BIOS. In this way only the disk drive knows it's actual physical (CHS) geometry. Access requests from the BIOS are *translated* internally on the hard disk controller into actual physical disk geometry. A disk's logical geometry will have a number of sectors approximately equal to, but never more than, the physical number of sectors on the disk. The reported logical geometry fits within the limits of the ATA standard, but not necessarily (most often not) within the limits of the `Int 13h` standard.

B.2.2 LBA Addressing

Up to this point we've been discussing addressing modes based on the properties of the physical disk drive. Now the discussion will transition to the modern *Logical Block Addressing*.

Another important thing that happened was the introduction of geometry translation at the BIOS level. This is an addressing mode which the BIOS will enable that translates the logical drive geometry [7] into CHS tuples compatible with the `Int 13h` system. This addressing mode is often called *Extended CHS*, or *Large mode* [8].

In LBA mode there is an abstraction between the operating system and the devices where the data is stored. Using LBA the operating system accesses data by unique identifiers. Each block is addressed by a simple identifier which increases sequentially. This system requires that all involved components are LBA aware: the disk drive controller, the BIOS, and the operating system.

Eventually disk drive capacities exceeded the maximum addressable range defined in original ATA-1 standard. In 2002 the T13 group released the ATA-6 standard [9] which introduced 48b addressing. This increased the maximum addressable capacity to 128PiB.

B.3 The Master Boot Record

The Master Boot Record (MBR) is located in the first sector of the primary disk drive. The MBR may be up to 446B of code, and partition tables may be up to 64B of data. When

[6] Zoned Bit Recording (PC Guide: `http://www.pcguide.com/ref/hdd/geom/tracks_ZBR.htm`) is an example of something logically impossible to represent with `Int 13h`

[7] Recall: this "logical geometry" has already been translated once to fit ATA standards for the BIOS by the disk controller

[8] PC Guide: Extended CHS/Large Mode: `http://www.pcguide.com/ref/hdd/bios/modesECHS-c.html`

[9] *INCITS 361-2002 (1410D): AT Attachment - 6 with Packet Interface (ATA/ATAPI - 6)*: `http://www.t13.org/documents/UploadedDocuments/project/d1410r3b-ATA-ATAPI-6.pdf`

you add in another 2B to record a *Boot Signature* you have 512B, which up until recently happened to be the typical size of one sector [10] [11] . This first sector is referred to by a special name, the `boot sector`.

Size (in bytes)	Percent	Purpose
440B	86%	Bootable Code (such as GRUB [12]/LILO [13])
004B	0.8%	Disk signature
002B	0.4%	Nulls
064B	13%	Partition Table
002B	0.4%	MBR Signature

Table B.1: Master Boot Record Contents

In the old days a disk cylinder (or track) was typically 63 sectors. This would represent one concentric ring of storage on a physical disk. Some people believe that early operating systems (notably MS−DOS) enforced requirements which dictated that partitions begin on cylinder boundaries, or that the OS needed to begin and end on a cylinder boundary. Jonathan de Boyne Pollard (JDBP) disputes that claim [14] , saying:

> It is often believed that disc partitions have to be aligned to cylinder or track boundaries. This is not in fact true and never really has been. There are alignment considerations for disc partitions, but they have nothing to do with cylinders, and they aren't mandatory. Operating systems will still work with misaligned partitions, just more slowly for some (not all) disc unit models.
>
> The idea that disc partitions have to aligned to cylinder boundaries is nonsense on its face. Millions of people have had discs where the first primary partition began on track zero, sector one, head one with no ill effect whatsoever on operating systems from MS-DOS through Windows NT to OS/2. That was, after all, the default that fdisk/Disk Manager on those operating systems used for almost two decades. At best, the purported alignment requirement would have been a track alignment, with all partitions starting at sector one (Sectors are numbered from one, remember.) of any given track.

[10] Seagate.com - Transition to Advanced Format 4K Sector Hard Drives: `http://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/`

[11] Pixel Beat - Details of GRUB on the PC: `http://www.pixelbeat.org/docs/disk/`

[12] The Grand Unified Boot Loader (GRUB): `http://www.gnu.org/software/grub/`

[13] Linux Loader (LILO): `http://lilo.alioth.debian.org/`

[14] The gen on disc partition alignment: `http://homepage.ntlworld.com/jonathan.deboynepollard/FGA/disc-partition-alignment.html`

> But this is not true, either. No version of any operating system has actually required this. Even MS-DOS was quite happy to have disc partitions starting at sectors other than 1. The only things that have required this have been disc partitioning utilities. There's been a bit of circular logic about this. The disc partitioning utilities enforced the requirement because their authors thought that it was a requirement, but people only thought that it was a requirement because fdisk and the like enforced it. It was what the partitioning utility programs enforced — so the logic went — so it must have been a restriction. In fact it never was, and no operating system itself has any trouble with this.
>
> —Jonathan de Boyne Pollard

What we can take away from JDBP here is this: Operating systems, not even MS−DOS, require partition's to begin (or end) on cylinder or track boundaries.

The very idea that partitions have such restrictions is a complete falsehood. A story passed down from hacker generation to generation, accepted as canon and never questioned.

JDBP goes on to also discuss the 4KiB alignment rule:

> There is a disc partition alignment rule that *does* reflect the actual hardware. It is the rule that partitions be aligned to 4KiB boundaries. This rule only makes sense for *some* hard disc models, however.
>
> In some hard disc models, the internal sector size has been increased from 0.5KiB to 4KiB. At the I/O command level, as system softwares access the disc, the sector size is still 0.5KiB, however. Such discs are known as "512 byte emulation" discs […]
>
> What happens on such "512e" discs is that whenever the operating system or the firmware reads a 0.5KiB sector, the disc unit itself is actually reading a whole 4KiB and handing the firmware/operating system the appropriate one-eighth; and whenever the firmware/operating system writes a 0.5KiB sector, the disc unit is actually reading a whole 4KiB sector, modifying one eighth, and writing the whole 4KiB back again.
>
> […]
>
> So it's simply necessary to ensure that those eight 0.5KiB sectors are contiguous and aligned to an actual 4KiB sector on the disc. The "natural" I/O boundaries used by the operating system must align with the internal, hidden, 4KiB boundaries of the physical disc. The eight 0.5KiB sectors in the I/O command must not span two or more 4KiB physical sectors; but must be exactly one 4KiB sector, and in the right order within that sector.

What we should first observe from this second quote is that there *is* a rule regarding sector alignment. But that rule has nothing to do with operating system requirements. Furthermore, this is only a rule and we are not obligated to follow it. Failure to follow the rule simply results in degraded I/O performance.

I recommend reading the entire page for a complete overview of these topics. JDBP does an excellent job separating the fact from fiction and explains how you can achieve correct 4KiB alignment, or realignment if you need to fix an existing system.

Note
For more information on the "native" 4KiB disk drive topic I recommend reviewing footnote [10].

Colophon

This book was created using free/open source software. All media within was created and saved in formats unencumbered by patents.

The standard typeface used in this book is *Source Sans Pro* [15] , the monospaced sequences use *Source Code Pro*. [16] Both of these beautiful fonts families were designed by Paul D. Hunt [17] at Adobe Systems Incorporated. Moreover, both of these families are available for use under the Open Font License version 1.1 [18] .

This book was written in 100% *lint-free* DocBook 5.1 XML [19] .

Composition of this book took place entirely in Emacs (nXML/RNG mode if you're curious), on an assortment of Fedora Linux [20] releases ranging from Fedora 11, *Leonidas*, to Fedora 23 (first edition published).

The single-page HTML version of this book [21] uses Twitter Bootstrap [22] for styling.

The print and PDF versions of this book were produced using an xsltproc → dblatex → xetex → xdvipdfmx toolchain.

[15] Source Sans Pro Announcement (2012-08-02): `http://blogs.adobe.com/typblography/2012/08/source-sans-pro.html`

[16] Source Code Pro Announcement (2012-09-24): `http://blogs.adobe.com/typblography/2012/09/source-code-pro.html`

[17] Paul D. Hunt on Adobe.com: `http://www.adobe.com/products/type/font-designers/paul-hunt.html`

[18] SIL Open Font License: `http://scripts.sil.org/OFL`

[19] DocBook 5: The Definitive Guide: `http://docbook.org/tdg51/en/html/`

[20] Visit the Fedora Project Homepage: `https://getfedora.org/`

[21] Single-page HTML: `http://scribesguides.com/books/vdg/latest/Virtual-Disk-Operations.html`

[22] Twitter Bootstrap: `http://getbootstrap.com/`