PYPARTED: PYTHON DOES DISK PARTITIONING

Build a custom, command line disk partitioning tool by joining the user-friendliness of Python and the power of C.

partitioning is a traditional way to split disk drives into manageable chunks. Linux comes with variety of tools to do the job: there are fdisk, cfdisk or GNU Parted, to name a few. GNU Parted is powered by a library by the name of libparted, which also lends functionality to many graphical tools such as famous GParted. Although it’s powerful, libparted is written in pure C and thus not very easy for the non-expert to tap into. If you’re writing your own custom partitioner, to use libparted you’re going to have to manage memory manually and do all the other elbow grease you do in C. This is where PyParted comes in – a set of Python bindings to libparted and a class library built on top of them, initially developed by Red Hat for use in the Anaconda installer.

So why would you consider writing disk partitioning software? There could be several reasons:

- You are developing a system-level component like an installer for your own custom Linux distribution
- You are automating a system administration task such as batch creation of virtual machine (VM) images. Tools like ubuntu-vm-builder are great, but they do have their limitations
- You’re just having fun.

PyParted has made its way into the Python Package Index (PyPI) yet, but you may be lucky enough to find it in your distribution’s repositories. Fedora (naturally), Ubuntu and Debian provide PyParted packages, and you can always build PyParted yourself from the sources. You will need the libparted headers (usually found in libparted-dev or similar package), Python development files and GCC. PyParted uses the distutils package, so simply enter python setup.py install to build and install it. It’s a good idea to install PyParted you’ve built yourself inside the virtualenv (see http://docs.python-guide.org/en/latest/dev/virtualenvs for details), to keep your system directories clean. There is also a Makefile, if you wish. This article’s examples use PyParted 3.10, but the concepts will stay the same regardless of the version you actually use.

Before we start, a standard caution: partitioning may harm the data on your hard drive. Back everything up before you do anything else!

Basic concepts

The PyParted API has two layers. At the bottom one is the _ped module. Implemented entirely in C, it tries to keep as close as possible to the native libparted C API. On top of that, the ‘parted’ package with high-level Python classes, functions and exceptions is built. You can use _ped functions directly if you wish; however, the parted package provides a more Pythonic approach, and is the recommended way to use PyParted in your programs unless you have some special requirements. We won’t go into any details of using the _ped module in this article.

Before you do anything useful with PyParted, you’ll need a Device instance. A Device represents a piece of physical hardware in your system, and provides the means to obtain its basic properties like model, geometry (cylinders/heads/sectors – it is mostly fake for modern disks, but still used sometimes), logical and physical sector sizes and so on. The Device class also has methods to read data from the hardware and write it back. To obtain a Device instance, you call one of the global functions exported by PyParted (in the examples below, >>> denotes the interactive Python prompt, and ... is an omission for readability reasons or line continuation if placed at the beginning):

```python
>>> import parted
>>> # requires root privileges to communicate
... with the kernel
>>> [dev.path for dev in parted.getAllDevices()]
['/dev/sda', '/dev/mapper/ubuntu--vg-swap_1',
 '/dev/mapper/ubuntu--vg-root',
 '/dev/mapper/sda5_crypt']
```

```python
>>> # get Device instance by path
>>> sda = parted.getDevice('/dev/sda')
```

```python
>>> sda.model
'ATA WDC WD1002FAEX-0'
```

“A word of caution: partitioning may harm the data on your hard drive. Back everything up!”

PyParted was developed to facilitate Red Hat’s installer, Anaconda.
Next comes the Disk, which is the lowest-level operating system-specific abstraction in the PyParted class hierarchy. To get a Disk instance, you'll need a Device first:

```python
>>> disk = parted.newDisk(sda)
```

Traceback (most recent call last):
...
```
<parted.DiskException: /dev/sda: unrecognised disk label>
```

This reads the disk label (ie the partitioning scheme) from `/dev/sda` and returns the Disk instance that represents it. If `/dev/sda` has no partitions (consider the `sda.clobber()` call before), `parted.DiskException` is raised. In this case, you can create a new disk label of your choice:

```python
>>> disk = parted.newDisk(sda, 'msdos') # or 'gpt'
```

You can do it even if the disk already has partition table on it, but again, beware of data-loss. PyParted supports many disk labels. However, traditional 'msdos' (MBR) and newer 'gpt' (GUID Partition Table) are probably most popular in PC world.

Disk's primary purpose is to hold partitions:

```python
# Will be empty after clobber() or freshDisk()
```

```python
>>> disk.partitions
```

```python
[parted.partition.Partition object at 0x104305050, ...]
```

Each partition is represented by Partition object which provides 'type' and 'number' properties:

```python
>>> existing_partition = disk.partitions[0]
```

```python
>>> existing_partition.type, existing_partition.number
('parted.PARTITION_NORMAL', 0)
```

Besides `parted.PARTITION_NORMAL`, there are other partition types (most importantly, `parted.PARTITION_EXTENDED` and `parted.PARTITION_LOGICAL`). The 'msdos' (MBR) disk label supports all of them, however 'gpt' can hold only normal partitions.

Partitions can also have flags like `parted.PARTITION_BOOT` or `parted.PARTITION_LVM`. Flags are set by the `Partition.setFlag()` method, and retrieved by `Partition.getFlag()`. We'll see some examples later.

The partition's position and size on the disk are defined by the `Geometry` object. Disk-related values (offsets, sizes, etc) in PyParted are expressed in sectors; this holds true for `Geometry` and other classes we'll see later. You can use the convenient function `parted.sizeToSectors(value, 'B', device, sectorSize)` to convert from bytes (denoted as 'B'), other units such as 'MB' are available as well. You set the Geometry when you create the partition, and access it later via the `partition.geometry` property:

```python
>>> # 128 MiB partition at the very beginning of the disk
```

```python
>>> geometry = parted.Geometry(start=0, length=128, 'MiB', sda.sectorSize, device=sda)
```

```python
>>> new_partition = parted.Partition(disk=disk, geometry=geometry)
```

```python
>>> new_partition.geometry
<parted.geometry.Geometry object at 0xdc9050>
```

Partitions (or geometries, to be more precise) may also have an associated `FileSystem` object. PyParted can't create new filesystems itself (parted can, but it is still recommended that you use special-purpose utilities like `mkfs`). However, it can probe for existing filesystems:

```python
>>> parted.probeFileSystem(new_partition.geometry)
```

```python
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

```python
>>> parted.probeFileSystem(existing_partition.geometry)
```

```python
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

Caution: partitioning may void your warranty

Playing with partitioning is fun but also quite dangerous: wiping the partition table on your machine's hard drive will almost certainly result in data loss. It is much safer to do your experiments in a virtual machine (like VirtualBox) with two hard drives attached. If this is not an option, you can 'fake' the hard drive with an image file ($S$ is a normal user and # is a superuser shell prompt):

```bash
$ dd if=/dev/zero of=<image_file_name> bs=512 count=<disk_size_in_sectors>
```

This will almost work; however, Partition.getDeviceNodeName() will return nonexistent nodes for partitions on that drive. For more accurate emulation, use:

```python
>>> parted.
```

```python
# losetup -f <image_file_name>
```

```python
# losetup -a /dev/loopX
```

```python
# losetup -d /dev/loopX
```

where $X$ is the losetup device assigned to your image file (get it with `losetup -a`). After that, you may refer to the partitions on your image file via `/dev/loopXP` (or `/dev/mapper/loopXP` depending on your distribution).

This will require root privileges, so be careful. You can still run your partitioning scripts on an image file as an ordinary user, given that the file has sufficient permissions (ie is writable for the user that you are running scripts as). The last command removes the device when it is no longer needed.

If you feel adventurous, you can also fake your hard drive with a qcow2 (as used by Qemu), VDI, VMDK or other image directly supported by virtual machine, Oracle VirtualBox or VMware Workstation/Player. These images can be created with `qemu-img` and mounted with `qemu-nbd`.

```python
>>> parted.probeFileSystem(existing_partition.geometry)
```

```python
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

```python
>>> parted.probeFileSystem(new_partition.geometry)
```

```python
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

Sometimes, it may look like changes you make to such virtual drives via external tools (like `mkfs`) are silently ignored. If this is your case, flush the disk buffers:

```python
# blockdev --flushbufs <device_node_name>
```

Playing with partitioning is fun but also quite dangerous: wiping the partition table on your machine's hard drive will almost certainly result in data loss. It is much safer to do your experiments in a virtual machine (like VirtualBox) with two hard drives attached. If this is not an option, you can 'fake' the hard drive with an image file ($S$ is a normal user and # is a superuser shell prompt):

```bash
$ dd if=/dev/zero of=<image_file_name> bs=512 count=<disk_size_in_sectors>
```

This will almost work; however, Partition.getDeviceNodeName() will return nonexistent nodes for partitions on that drive. For more accurate emulation, use:

```bash
>>> parted.
```

```bash
# losetup -f <image_file_name>
```

```bash
# losetup -a /dev/loopX
```

```bash
# losetup -d /dev/loopX
```

where $X$ is the losetup device assigned to your image file (get it with `losetup -a`). After that, you may refer to the partitions on your image file via `/dev/loopXP` (or `/dev/mapper/loopXP` depending on your distribution).

This will require root privileges, so be careful. You can still run your partitioning scripts on an image file as an ordinary user, given that the file has sufficient permissions (ie is writable for the user that you are running scripts as). The last command removes the device when it is no longer needed.

If you feel adventurous, you can also fake your hard drive with a qcow2 (as used by Qemu), VDI, VMDK or other image directly supported by virtual machine, Oracle VirtualBox or VMware Workstation/Player. These images can be created with `qemu-img` and mounted with `qemu-nbd`.

```bash
>>> parted.probeFileSystem(existing_partition.geometry)
```

```bash
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

```bash
>>> parted.probeFileSystem(new_partition.geometry)
```

```bash
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

Sometimes, it may look like changes you make to such virtual drives via external tools (like `mkfs`) are silently ignored. If this is your case, flush the disk buffers:

```bash
# blockdev --flushbufs <device_node_name>
```

Playing with partitioning is fun but also quite dangerous: wiping the partition table on your machine's hard drive will almost certainly result in data loss. It is much safer to do your experiments in a virtual machine (like VirtualBox) with two hard drives attached. If this is not an option, you can 'fake' the hard drive with an image file ($S$ is a normal user and # is a superuser shell prompt):

```bash
$ dd if=/dev/zero of=<image_file_name> bs=512 count=<disk_size_in_sectors>
```

This will almost work; however, Partition.getDeviceNodeName() will return nonexistent nodes for partitions on that drive. For more accurate emulation, use:

```bash
>>> parted.
```

```bash
# losetup -f <image_file_name>
```

```bash
# losetup -a /dev/loopX
```

```bash
# losetup -d /dev/loopX
```

where $X$ is the losetup device assigned to your image file (get it with `losetup -a`). After that, you may refer to the partitions on your image file via `/dev/loopXP` (or `/dev/mapper/loopXP` depending on your distribution).

This will require root privileges, so be careful. You can still run your partitioning scripts on an image file as an ordinary user, given that the file has sufficient permissions (ie is writable for the user that you are running scripts as). The last command removes the device when it is no longer needed.

If you feel adventurous, you can also fake your hard drive with a qcow2 (as used by Qemu), VDI, VMDK or other image directly supported by virtual machine, Oracle VirtualBox or VMware Workstation/Player. These images can be created with `qemu-img` and mounted with `qemu-nbd`.

```bash
>>> parted.probeFileSystem(existing_partition.geometry)
```

```bash
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

```bash
>>> parted.probeFileSystem(new_partition.geometry)
```

```bash
<parted FileSystem<parted._ped.FileSystem object at 0xdc9050>... type=parted.PARTITION_NORMAL, ...>
```

Sometimes, it may look like changes you make to such virtual drives via external tools (like `mkfs`) are silently ignored. If this is your case, flush the disk buffers:

```bash
# blockdev --flushbufs <device_node_name>
```
Let’s start with **Alignment**, which is defined by two values: offset and grain. Any sector number \( X \) with \( X = \text{offset} + N \times \text{grain} \) (with \( N \) being non-negative integer) complies with Alignment. When you need to tell PyParted that your partitions should start (or end) at a 1MiB (or some other) boundary, Alignment is the way to do it. Any value satisfies Alignment(0, 1) which is equivalent to no alignment at all.

**Constraint** is basically a set of six conditions on **Geometry** (not a Partition!) that are wrapped together to control the following:

- How the Geometry’s boundaries are aligned (**startAlign**/**endAlign** properties).
- Where the Geometry can start or end (**startRange**/**endRange**).
- What the Geometry’s minimum and maximum sizes are (**minSize**/**maxSize**).

You do not always need to specify all of them. The Constraint constructor provides the shortcuts **minGeom**, **maxGeom** and **exactGeom**, which create a Constraint that fully embraces, is fully contained by, or exactly coincides with the Geometry you pass as an argument. If you use one of these, any alignment will satisfy the Constraint check. As another special case, **Constraint(device=dev)** accepts any Geometry attached to the Device `dev`.

It isn’t easy to catch the meaning of all these properties at once. Have a look at the diagram below, which depicts all of them in graphical form. Both Alignment and Constraint provide the **intersect()** method, which returns the object that satisfies both requirements. You can also check that the given Geometry satisfies the Constraint with the **Constraint.isSolution(geom)** method. The **Constraint.solveMax(geom)** method returns the maximum permitted geometry that satisfies the Constraint, and **Constraint.solveNearest(geom)** returns the permitted geometry that is nearest to the `geom` that you’ve specified. What’s ‘nearest’ is up to the implementation to decide.

### Partitioning on Ye Olde Windows NT

Imagine for a moment you need to create system partition for Windows NT4 prior to Service Pack 5 (remember that weird creature?). As the hardware requirements suggest ([http://en.wikipedia.org/wiki/Windows_NT#Hardware_requirements](http://en.wikipedia.org/wiki/Windows_NT#Hardware_requirements)), it must be no more than 4GB in size, contained within the first 7.8GB of the hard disk, and begin in the first 4GBs. Here’s how to do this with PyParted:

```python
>>> optimal = sda.optimumAlignment
>>> start = parted.Geometry(device=sda, ... start=0, ... end=parted.sizeToSectors(4, 'GB', ... sda.sectorSize))
>>> end = parted.Geometry(device=sda, ... start=0, ... end=parted.sizeToSectors(7.8, 'GB', ... sda.sectorSize))
>>> min_size = parted.sizeToSectors(124, 'MB', ... sda.sectorSize) # See [ref:4]
```
In order to assign a label to a disk, which is especially useful for removable media (/media/BobsUSBStick says more than /media/sdb1). But they are not the disk labels that libparted refers to.

When we speak of disk labels on these pages, we mean partition tables. It is very uncommon for a hard disk to not to have one (although many flash drives comes with no partitions). Linux usually sees unpartitioned devices as /dev/sdX (with X being a letter); partitions are suffixed with an integer (say, /dev/sda1).

There are many different partitioning schemes (or disk labels). Traditionally, the 'msdos' (MBR) disk partitioning scheme was the most popular one for PCs. By today's standards, it's very limited: it may contain at most four partitions (called 'primary') and stores partition offsets as 32-bit integers. If you need more, one partition should be marked as 'extended', and it may contain as many logical partitions as you want. This is the reason why logical partitions are always numbered starting with 5 in Linux.

The newer GUID Partition Table ('gpt') is much more flexible. It's usually mentioned in connection with UEFI, however it is self-contained and can be used on BIOS systems as well. In a 'gpt' disk label, partitions are identified by Globally Unique Identifiers (GUID) values. Their starting and ending offsets are 64-bit, so there is some safety margin for hard disks of tomorrow's capacities.

Each disk needs a label

Many modern operating systems enable you to assign a label to a disk, which is especially useful for removable media (/media/BobsUSBStick says more than /media/sdb1). But they are not the disk labels that libparted refers to.

When we speak of disk labels on these pages, we mean partition tables. It is very uncommon for a hard disk to not to have one (although many flash drives comes with no partitions). Linux usually sees unpartitioned devices as /dev/sdX (with X being a letter); partitions are suffixed with an integer (say, /dev/sda1).

There are many different partitioning schemes (or disk labels). Traditionally, the 'msdos' (MBR) disk partitioning scheme was the most popular one for PCs. By today's standards, it’s very limited: it may contain at most four partitions (called ‘primary’) and stores partition offsets as 32-bit integers. If you need more, one partition should be marked as ‘extended’, and it may contain as many logical partitions as you want. This is the reason why logical partitions are always numbered starting with 5 in Linux.

The newer GUID Partition Table (‘gpt’) is much more flexible. It’s usually mentioned in connection with UEFI, however it is self-contained and can be used on BIOS systems as well. In a ‘gpt’ disk label, partitions are identified by Globally Unique Identifiers (GUID) values. Their starting and ending offsets are 64-bit, so there is some safety margin for hard disks of tomorrow’s capacities.

Your very own fdisk

fdisk is probably the most basic partitioning tool. It’s an console program: it reads single-letter commands entered by a user and acts accordingly, printing results on the screen. Originally, it supported MBR and also BSD/SUN disk labels; we'll stick to MBR only.

Our example (let’s call it fdisk.py) is somewhat more elaborate version of fdisk/fdisk.py found in the PyParted sources https://git.fedorahosted.org/cgit/pyparted.git/tree/src/fdisk/fdisk.py, but it’s a bit simplified compared with the real fdisk. Since parted and fdisk are not 100% compatible (although parted is more advanced in many ways), there are some discrepancies (see comments in the sources for details). However, fdisk.py implements all the basic functions you’d expect from the partitioning software: it can create partitions (both primary and logical), list them, delete them, and even mark them as bootable.

All of these options are implemented as methods of the Fdisk class, which is instantiated when the program starts. In Fdisk.__init__(), we check whether the drive already has a partition table and create it if necessary. If the disk has any partition table other than MBR, the program exits immediately. The main loop simply dispatches commands entered by a user to Fdisk’s instance methods. If any of them raise an ExitMainLoop exception, the program ends.

Let’s start with the code that displays a partition table. In the real fdisk, it looks like the image at the top of page 96. And the following is the relevant part of fdisk.py code:

```python
print ""
```

Disk (path): {size_mbytes:d} MB, {size:d} bytes
(heads:d) heads, {sectors:d} sectors/track, \n(cylinders:d) cylinders, total {sectors:d} sectors
Units = 1 * sectors of (unit:d) = (unit:d) bytes

```
libparted provides the power behind many well-known free software tools, including GParted.

Sector size (logical/physical): \( \{\text{sector\_size:d}\} \) \( \text{bytes} / \{\text{physical\_sector\_size:d}\} \text{ bytes} \)
I/O size (minimum/optimal): \( \{\text{minimum\_io\_size:d}\} \) \( \text{bytes} / \{\text{optimal\_io\_size:d}\} \text{ bytes} \)

```python
"{}.*format(**data)"
```

width = len(disk.partitions[0].path) \nif disk.partitions else len('Device') + 1
print "{0:>{width}} Boot      Start         
End      Blocks   Id  System".format('Device', width)
width = len(disk.partitions[0].path) 
```

We can deduce the maximum and optimum I/O sizes from corresponding alignment values (see the previous section). Since we don’t allow our user to change units (as the real `fdisk` does), unit variable is always equal to sector size. Everything else is straightforward.

Parted has no concept of DOS disk label types such as ‘Linux native’, ‘Linux swap’, or ‘Win95 FAT32’. If you were to install good old Slackware using `fdisk` back in 1999, you would almost certainly use some of these. So we emulate disk labels to some extent on top of the partition and filesystem types provided by PyParted. This is done in the `Fdisk._guess_system()` method. We recognise things like ‘Linux LVM’ and ‘Linux RAID’, `parted.PARTITION_SWAP` maps to ‘Linux swap’, ext2/3/4, btrfs, ReiserFS, XFS, and JFS are displayed as ‘Linux native’, and we even support FAT16/32 and NTFS. As a bonus, PyParted enables you to identify hidden or service partitions added by some hardware vendors (https://git.fedorahosted.org/cgit/pyparted.git/tree/src/fdisk/fdisk.py). If the heuristic doesn’t work, we print ‘unknown’.

Creating partitions

It is also easy to delete a partition. The only thing to remember is that partitions on the disk can be out of order, so you can’t use the partition number as an index in the `disk.partitions` array. Instead, we iterate over it to find the partition with the number that a user has specified:

```python
for p in self.disk.partitions:
    if p.number == number:
        break
```

If we try to delete an extended partition that contains logical partitions, `parted.PARTITION_EXCEPTION` will be raised. We catch it and print a friendly error message. The last `break` statement is essential. PyParted automatically renumerates the partitions when you delete any of them. So, if you have, for instance, partitions 1–4, and delete the one numbered 3, the partition that was previously number 4 will become the new 3, and will be deleted at the next iteration of the loop.

The largest method, not surprisingly, is the one that creates partitions. Let’s look at it step by step. First of all, we check how many primary and extended partitions are already on the disk, and how many primary partitions are available:

```python
# Primary partitions count
pri_count = len(self.disk.getPrimaryPartitions())
# HDDs may contain only one extended partition
ext_count = 1 if self.disk.getExtendedPartition() else 0
# First logical partition number
lpart_start = self.disk.maxPrimaryPartitionCount + 1
# Number of spare partitions slots
parts_avail = self.disk.maxPrimaryPartitionCount + 1
(1, 0, 3, 0) (0, 0, 1, 0)
```

Then we check if the disk has some free space and
return from the method if not. After this, we ask the user for the partition type. If there are no primary partitions available, and no extended partition exists, one of primary partitions needs to be deleted, so we return from the method again. Otherwise, a user can create either a primary partition, an extended partition (if there isn’t one yet), or a logical partition (if an extended partition is already here). If the disk has fewer than three primary partitions, a primary partition is created by default; otherwise we default to an extended or logical one.

We also need to find a place to store the new partition. For simplicity’s sake, we use the largest free region available. Fdisk._get_largest_free_region() is responsible for this; it’s quite straightforward except one simple heuristic. It ignores regions shorter than optimum alignment grain (usually 2048 sectors); they are most likely alignment gaps.

Any logical partition created must fit inside the extended partition, and we use Geometry.intersect() to ensure that this is the case. On the contrary, a primary partition must lie outside the extended, so if the intersection exists, we return from the method. The code is similar in both cases; below is the former check (which is a bit shorter):

```python
try:
    geometry = ext_part.geometry.intersect(geometry)
except ArithmeticError:
    print(“No free sectors available”)
    return

if there is no intersection, Geometry.intersect() raises ArithmeticError.

All the heavy lifting is done in the Fdisk._create_partition() method, which accepts the partition type and the region that will hold the new partition. It starts as follows:

```python
alignment = self.device.optimalAlignedConstraint
constraint = parted.Constraint(maxGeom=geometry).\n    intersect(alignment)
data = {
    ‘start’: constraint.startAlign,\n    alignUp(region, region.start),\n    ‘end’: constraint.endAlign,\n    alignDown(region, region.end),
}
```

As in the real fdisk(), we align partitions optimally by default. The partition created must be no larger than the available free space (the region argument), so the maxGeom constraint is enforced. Intersecting these gives us a Constraint that aligns partitions optimally within boundaries specified. data[start] and data[end] are used as guidelines when prompting for the partition’s boundaries, and they shouldn’t be misleading. Thus we perform the same calculation that libparted does internally: find start or end values that are in a specified range and aligned properly. Try to play with these; for example, change the alignment to self.device.minimalAlignedConstraint and see what changes when you create a partition on an empty disk.

Resources

- PyParted homepage: https://fedorahosted.org/pyparted
- Partition types: properties of partition tables: www.win.tue.nl/~aeb/partitions/partition_types-2.html
- Windows NT4 Hardware Requirements: http://en.wikipedia.org/wiki/Windows_NTHardware_requirements
- fdisk.py sources (this article’s version) https://github.com/vsinitsyn/fdisk.py
- PyParted’s fdisk.py sample code https://git.fedorahosted.org/cgit/pyparted.git/tree/src/fdisk/fdisk.py

After that, Fdisk._create_partition() asks for the beginning and the end of the partition. Fdisk._parse_last_sector_expr() parses expressions like +100M, which fdisk(1) uses as the last sector specifier. Then, the partition is created as usual:

```python
try:
    partition = parted.Partition(\n        disk=self.disk,\n        type=type,\n        geometry=parted.Geometry(\n            device=self.device,\n            start=part_start,\n            end=part_end)),\n    self.disk.addPartition(partition=partition,\n        constraint=constraint)\nexcept (parted.PartitionException,\n    parted.GeometryException,\n    parted.CreateException) as e:
    raise RuntimeError(e.message)
```

If part_start or part_end are incorrect, the exception will be raised (see the comments in the source code for the details). It is caught in the Fdisk.add_partition() method, which displays error messages and returns.

To save the partition table on to the disk, a user enters the w command at the fdisk.py prompt. The corresponding method (Fdisk.write()) simply calls disk.commit() and raises MainLoopExit to exit.

Afore ye go

Python is arguably the scripting language of choice in today’s Linux landscape, and is widely used for various tasks including the creation of system-level components. As an interpreted language, Python is just as powerful as its bindings, which enable scripts to make use of native C libraries. In this perspective, it’s nice to have tools like PyParted in our arsenal. Implementing partitioners is hardly an everyday task for most of us, but if you ever face it, the combination of an easy-to-use language and a production-grade library can greatly reduce your programming efforts and development time.

Dr Valentine Sinitsyn edited the Russian edition of O’Reilly’s Understanding the Linux Kernel, has a PhD in physics, and is currently doing clever things with Python.