Blivet Documentation

Release 1.20.3

David Lehman

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Introduction to Blivet

Blivet is a python module for system storage configuration.

The main thing that blivet offers is the ability to model a series of changes without necessarily commiting any of the changes to disk. You can schedule an arbitrarily large series of changes (called 'actions'), seeing the effects of each (within the `DeviceTree` instance) as it is scheduled. Nothing is written to disk, however, until you execute the actions.

CHAPTER 2

Building Blocks

Individual block devices are represented by the various subclasses of `StorageDevice`, while the formatting of the data they contain is represented by the various subclasses of `DeviceFormat`. The hierarchy of devices is represented by `DeviceTree`.

`DiskDevice`, `PartitionDevice`, `LVMLogicalVolumeDevice`, and `MDRaidArrayDevice` are some of the most important examples of `StorageDevice` subclasses.

Some examples of `DeviceFormat` subclasses include `SwapSpace`, `DiskLabel`, and subclasses of `FS` such as `Ext4FS` and `XFS`.

Every `StorageDevice` instance has a `format` that contains an instance of some `DeviceFormat` subclass – even if it is "blank" or "unformatted" (in which case it is an instance of `DeviceFormat` itself).

Every `DeviceFormat` has a `device` attribute that is a string representing the path to the device node for the block device containing the formatting. `StorageDevice` and `DeviceFormat` can represent either existent or non-existent devices and formatting.

`StorageDevice` and `DeviceFormat` share a similar API, which consists of methods to control existing devices/formats (`setup()`, `teardown()`), methods to create or modify devices/formats (`create()`, `destroy()`, `resize()`) , and attributes to store critical data (`status`, `exists`) . Some useful attributes of `StorageDevice` that are not found in `DeviceFormat` include `parents`, `isleaf`, `ancestors`, and `disks`.

`DeviceTree` provides `devices` which is a list of `StorageDevice` instances representing the current state of the system as configured within blivet. It also provides some methods for looking up devices (`getDeviceByName()`), for listing devices that build upon a device (`getDependentDevices()`), and for listing direct descendants of a given device (`getChildren()`).

CHAPTER 3

Getting Started

First Steps

First, create an instance of the `Blivet` class:

```
import blivet
b = blivet.Blivet()
```

Next, scan the system's storage configuration and store it in the tree:

```
b.reset()
```

Now, you can do some simple things like listing the devices:

```
for device in b.devices:
    print(device)
```

To make changes to the configuration you'll schedule actions, but `Blivet` provides some convenience methods to hide the details. Here's an example of removing partition '/dev/sda3':

```
sda3 = b.devicetree.getDeviceByName("sda3")
b.destroyDevice(sda3)    # schedules actions to destroy format and device
```

At this point, the StorageDevice representing sda3 is no longer in the tree. That means you could allocate a new partition from the newly free space if you wanted to (via blivet, that is, since there is not actually any free space on the physical disk yet – you haven't commited the changes). If you now ran the following line:

```
sda3 = b.devicetree.getDeviceByName("sda3")
```

sda3 would be None since that device has been removed from the tree.

When you are ready to commit your changes to disk, here's how:

```
b.doIt()
```

That's it. Now you have actually removed /dev/sda3 from the disk.

Here's an alternative approach that uses the lower-level `DeviceTree` class directly:

```
import blivet
dt = blivet.devicetree.DeviceTree()
dt.populate()
sda3 = dt.getDeviceByName("sda3")
action1 = ActionDestroyFormat(sda3)
action2 = ActionDestroyDevice(sda3)
dt.registerAction(action1)
dt.registerAction(action2)
dt.processActions()
```

Here's the Blivet approach again for comparison:

```
import blivet
b = blivet.Blivet() # contains a DeviceTree instance
b.reset()   # calls DeviceTree.populate()
sda3 = b.devicetree.getDeviceByName("sda3")
b.destroyDevice(sda3)   # schedules actions to destroy format and device
b.doIt()    # calls DeviceTree.processActions()
```

Scheduling a Series of Actions

Start out as before:

```
import blivet
from blivet.size import Size
b = blivet.Blivet()
b.reset()
sda3 = b.devicetree.getDeviceByName("sda3")
```

Now we're going to wipe the existing formatting from sda3:

```
b.destroyFormat(sda3)
```

Now let's assume sda3 is larger than 10GiB and resize it to that size:

```
b.resizeDevice(sda3, Size("10 GiB"))
```

And then let's create a new ext4 filesystem there:

```
new_fmt = blivet.formats.getFormat("ext4", device=sda3.path)
b.formatDevice(sda3, new_fmt)
```

If you want to commit the whole set of changes in one shot, it's easy:

```
b.doIt()
```

Now you can mount the new filesystem at the directory "/mnt/test":

```
sda3.format.setup(mountpoint="/mnt/test")
```

Once you're finished, unmount it as follows:

```
sda3.format.teardown()
```

Disk Partitions

Disk partitions are a little bit tricky in that they require an extra step to actually allocate the partitions from free space on the disk(s). What that means is deciding exactly which sectors on which disk the new partition will occupy. Blivet offers some powerful means for deciding for you where to place the partitions, but it also allows you to specify an exact start and end sector on a specific disk if that's how you want to do it. Here's an example of letting Blivet handle the details of creating a partition of minimum size 10GiB on either sdb or sdc that is also growable to a maximum size of 20GiB:

```
sdb = b.devicetree.getDeviceByName("sdb")
sdc = b.devicetree.getDeviceByName("sdc")
new_part = b.newPartition(size=Size("10 GiB"), grow=True,
                          maxsize=Size("20 GiB"),
                          parents=[sdb, sdc])
b.createDevice(new_part)
blivet.partitioning.doPartitioning(b)
```

Now you could see where it ended up:

```
print("partition %s of size %s on disk %s" % (new_part.name,
                                              new_part.size,
                                              new_part.disk.name))
```

From here, everything is the same as it was in the first examples. All that's left is to execute the scheduled action:

```
b.doIt()     # or b.devicetree.processActions()
```

Backing up, let's see how it looks if you want to specify the start and end sectors. If you specify a start sector you have to also specify a single disk from which to allocate the partition:

```
new_part = b.newPartition(start=2048, end=204802048, parents=[sdb])
```

All the rest is the same as the previous partitioning example.

CHAPTER 4

Testing Blivet

Note: The test suite documented here is available only from the git repository not as part of any installable packages.

In order to execute blivet's test suite from inside the source directory execute the command:

```
make test
```

Tests descending from `ImageBackedTestCase` or `LoopBackedTestCase` require root access on the system and will be skipped if you're running as non-root user. Tests descending from `ImageBackedTestCase` will also be skipped if the environment variable JENKINS_HOME is not defined. If you'd like to execute them use the following commands (as root):

```
# export JENKINS_HOME=`pwd`
# make test
```

To execute the Pylint code analysis tool run:

```
make check
```

Running Pylint doesn't require root privileges but requires Python3 due to usage of pocket-lint.

It is also possible to generate test coverage reports using the Python coverage tool. To do that execute:

```
make coverage
```

It is also possible to check all external links in the documentation for integrity. To do this:

```
cd doc/
make linkcheck
```

Test Suite Architecture

Blivet's test suite relies on several base classes listed below. All test cases inherit from them.

- `unittest.TestCase` - the standard unit test class in Python. Used for tests which don't touch disk space;
- `StorageTestCase` - intended as a base class for higher-level tests. Most of what it does is stub out operations that touch disks. Currently it is only used in `DeviceActionTestCase`;
- `LoopBackedTestCase` and `ImageBackedTestCase` - both classes represent actual storage space. `ImageBackedTestCase` uses the same stacks as anaconda disk image installations. These mimic normal disks more closely than using loop devices directly. Usually `LoopBackedTestCase` is used for stacks of limited depth (eg: md array with two loop members) and `ImageBackedTestCase` for stacks of greater or arbitrary depth.

In order to get a high level view of how test classes inherit from each other you can generate an inheritance diagram:

```
PYTHONPATH=.:tests/ python3-pyreverse -p "Blivet_Tests" -o svg -SkAmy tests/
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search