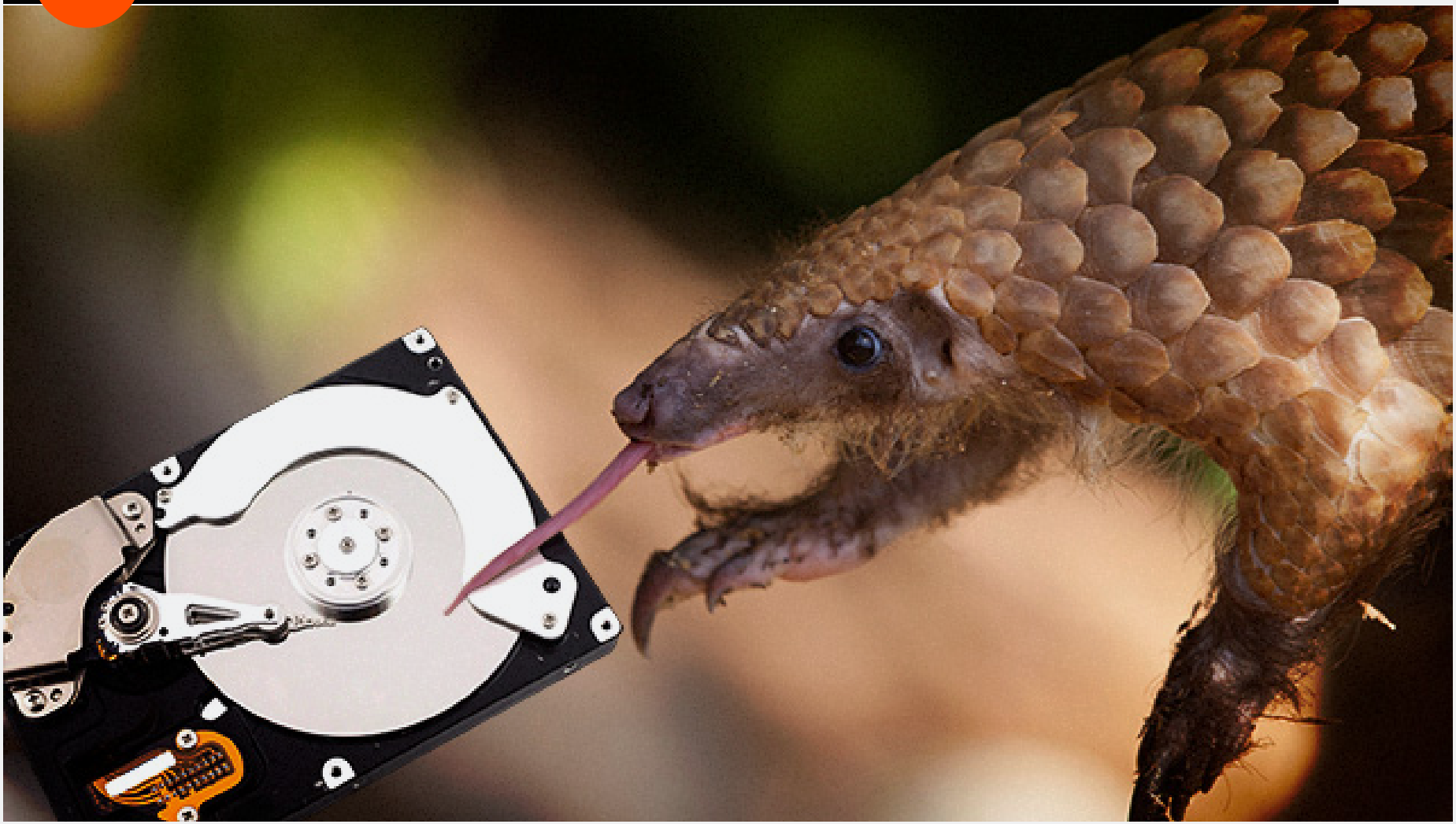
BIZ & IT —

Ars walkthrough: Using the ZFS next-gen filesystem on Linux

If btrfs interested you, start your next-gen trip with a step-by-step guide to ZFS.

JIM SALTER - 2/23/2014, 12:00 PM



Aurich Lawson / ThinkStock

If you're not an expert on armored anteaters that's a pangolin.

In my last article on next-gen filesystems, we did something in between a generic high altitude overview of next-gen filesystems and a walkthrough of some of btrfs' features and usage. This time, we're going to specifically look at what ZFS brings to the table, walking through getting it installed and using it on one of the more popular Linux distributions: Precise Pangolin. That's the most current Long Term Support (LTS) Ubuntu release.

Inside "next-gen" filesystems

Ubuntu in particular or even Linux in general. You can always visit http://zfsonlinux.org directly for help with the initial installation if you prefer RHEL or Fedora or Arch or what have you—and if you're a BSD fan, ZFS is available from the base installer in either PC-BSD or in the latest 10.0 release of FreeBSD itself.

In the interest of brevity, I'm going to assume you're already familiar with most of the generic terms and features associated with next-gen filesystems: atomic snapshots, asynchronous incremental replication, self-healing arrays, per-block checksumming, etc. If you *aren't* already familiar with those concepts, you might want to brush up on the last article to catch up.

Prerequisites and installation

You'll need a 64-bit PC or virtual machine with a recommended minimum of 8GB of RAM (you may be able to squeak by with less—possibly much less—but you're more likely to encounter performance degradation or odd behavior if you do) and several hard drives or partitions available to use with ZFS. We're going to use and cover Ubuntu Precise here, specifically. If you have a different distribution of Linux, you'll need to look at a guide for installation on your distribution at http://zfsonlinux.org. If you're using (a new version of) FreeBSD, PC-BSD, or one of the Solaris variants, you should have ZFS support built-in already.

Assuming that you've got a 64-bit PC running Ubuntu Precise just like I do here, actual installation is pretty mind-numbingly simple: first, we need to add the PPA (Personal Package Archive) for ZFSonLinux, then install the package itself:

```
you@box:~$ sudo apt-get install python-software-properties
you@box:~$ sudo apt-add-repository ppa:zfs-native/stable
you@box:~$ sudo apt-get update
you@box:~$ sudo apt-get install ubuntu-zfs
```

In the first command above, we gave ourselves access to the **apt-add-repository** command, which makes it much simpler to safely add PPAs to our repository list. Then we added the PPA, updated our source list to reflect that, and installed the package itself. Couldn't be (much) easier. One note: the final step isn't just slapping a binary in place. This command automatically compiles the module for your particular kernel live at the time of installation, so you should expect it to take a minute or five.

Initial tuning

y choose

One of the weaknesses in the ZFS implementation on Linux is that the ARC—which is ZFS' Advanced Replacement Cache, a smarter type of cache than the typical "first in first out" used in most filesystems—is unfortunately too slow to release RAM back into the system. *In theory*, it should behave as a normal filesystem cache does. When your system needs more RAM, the ARC should release RAM to it as necessary. In practice, large memory allocation requests—such as starting up a virtual machine or a large database—are likely to simply fail if they need RAM currently allocated to the ARC.

What this means for you is that you should manually *limit* the ARC to an appropriate maximum value for your system. A pure fileserver or NAS might want almost all RAM available for the ARC, or a virtual machine host might want as much RAM as possible available for the system itself. If you're in any doubt at all, half the system RAM is a great starting point, and you can adjust later if you need to. We'll set this value in **/etc/modprobe.d/zfs.conf**:

```
# /etc/modprobe.d/zfs.conf
#
# yes you really DO have to specify zfs_arc_max IN BYTES ONLY!
# 16GB=17179869184, 8GB=8589934592, 4GB=4294967296, 2GB=2147483648, 1GB=1073741824, 500MB=536870912, 250MB=2
#
options zfs zfs_arc_max=4294967296
```

Note that just creating this file doesn't actually apply the change. The file is only read when the kernel module is initially loaded, so you'll have to either load and unload the module or reboot the system to make it take effect. If you're adventurous, and you have a *really* new version of ZFS on Linux (newer than the version I am using today), you may also be able to just echo the value directly to the module for immediate effect:

```
you@box:~$ sudo -s
root@box:~# echo 4294967296 > /sys/module/zfs/parameters/zfs_arc_max
```

Be aware that changes echoed directly to the module as shown above take place immediately but *do not* persist across reboots. Extra-confusingly, you need to look at a *different* value if you want to check what your current in-use value of zfs_arc_max is:

Once you're satisfied that you've set zfs_arc_max to a value appropriate for your system (remember: half the available RAM is a good rule of thumb), you're ready to move on.

Learning ZFS lingo

Before we go nuts on the command line, let's briefly discuss some important ZFS terminology—that is, how we refer to the hierarchy that the actual physical disks will fit into. In the broadest terms, any data stored on ZFS is stored on one or more vdevs, which may populate one or more *higher-level* vdevs, which populate a single zpool (though you can have multiple zpools in one system). Confused yet? Let's start from the smallest term and move our way up.

vdev

A vdev is a Virtual DEVice. This can be a single physical drive or partition, or it can be a higher-level vdev consisting of multiple... uh... "lower-level" vdevs. To make everybody's life easier, from here on out I'm just going to refer to simple drives or partitions as *devices*, and when I use the term *vdev,* I will be referring to higher-level vdevs.

higher-level vdev

A "higher-level vdev" is a special ZFS raid array of lower-level vdevs, typically meaning of individual physical drives or partitions. These devices may be arranged in the vdev as a simple mirror or as a striped array with parity. Parity may be one, two, or three blocks per stripe—corresponding to RAID5, RAID6, or a mythical nonexistent RAID7 (referred to as raidz1, raidz2, and raidz3 respectively). Note that it's not possible to create a higher-level vdev out of *other* higher-level vdevs. However, if you have multiple higher-level vdevs in a single pool, the pool functions as a variable stripe width raid0 across them—so a pool filled with mirror vdevs is largely similar to a raid10 array, a pool filled with raidz1 vdevs is largely similar to a raid50 array, and so on.

One final note about vdevs: *vdevs are immutable.* This means that once you've created yourself a RAIDZ1 vdev with three drives in it, if you buy a fourth drive, you *cannot add it to your existing vdev* (sorry, Charlie). You can always add more vdevs to a zpool, but you *cannot grow or shrink a vdev once created*. There is *one* (and *only* one) exception: if you remove and then replace *each* drive in an existing vdev with a larger drive, one by painful one, once *every single drive* has been replaced with a larger one, you can resilver your vdev, and it will then be higher capacity. This could, of course, require *weeks* in a large and heavily populated vdev... but that's

our cup of

zpool

A zpool is a *named* device which consists of one or more vdevs in what is *effectively* a variable-stripe-width RAID0 configuration. Whew. What this actually means is that you can add a bunch of arbitrary devices or higher-level vdevs into a zpool, and the system will do its best to fill them all up at the same time, even if they are of different sizes. For example, say you create a new zpool named *ars* with two vdevs: a two-drive 1TB mirror vdev, and a two-drive 2TB mirror vdev. It will look something like this:

```
you@box:~$ sudo zpool status ars
  pool: ars
 state: ONLINE
 scrub: none requested
config:

        NAME                      STATE     READ WRITE CKSUM
        ars                       ONLINE       0     0     0
          mirror-0                ONLINE       0     0     0
            wwn-0x50014ee2080259c8  ONLINE       0     0     0
            wwn-0x50014ee2080268b2  ONLINE       0     0     0
          mirror-1                ONLINE       0     0     0
            wwn-0x50014ee25d4cdecd  ONLINE       0     0     0
            wwn-0x50014ee25d4ce711  ONLINE       0     0     0

errors: no known data errors
```

OK, so that's relatively easy to visualize—a zpool with two mirror vdevs in it. But what's this about "variable stripe width RAID0?" Basically, the zpool will do its best to fill all vdevs up at the same rate relative to their capacity. So in our example above, if mirror-0 is two 1TB drives, and mirror-1 is two 2TB drives, for every 3GB of data we write, 1GB of it goes on mirror-0, and the other 2GB goes on mirror-1. Each vdev will have the same *percentage* of space free at any given time, all the way up to when the pool's completely full... and mirror-1 is going to be receiving *twice* as much of the data as mirror-0 is, which will correspondingly hurt your overall performance.

e add a
ev with
1TB free (the half-full mirror-1), and a vdev with 2TB free (the new, empty mirror-2). If we write another 1400GB of data to the zpool now, 200GB goes on mirror-0, 400GB goes on mirror-1, and 800GB goes on mirror-2—so, again, the elements of our zpool are filling their *remaining* empty space at the same rate.

Yes, this means that now you're working the drives in vdev3 much harder than the drives in vdev2, and the drives in vdev2 much harder than the drives in vdev1. So this zpool will perform significantly slower than it would have if all of the vdevs had been in it to begin with *and* if all of the vdevs had been the same size. Such is life under ZFS; and you may very well choose *not* to use mismatched vdevs—or add new vdevs to an existing zpool—for exactly this reason.

Parity

For the purposes of this article, I'm going to be sloppy and refer to both mirror vdevs and raidz vdevs as vdevs *with parity*, meaning that if a single device in the vdev fails or returns corrupt data, the data can be reconstructed from other devices in the vdev. Note that a vdev *without* parity (such as a single drive or partition) *has no way of reconstructing corrupt or missing data*. More on that next.

Hating your data

Hating your data is a highly technical term referring to the addition of a single device to an existing zpool. I'm being tongue-in-cheek here, but I'm hoping it grabs your attention. The nature of a zpool means you can *really* get yourself badly in trouble if you aren't careful if you decide to add new vdevs to it later. What happens if, instead of adding another 2TB mirror vdev, we had added a *single* 2TB drive to our "ars" pool from the zpool example above?

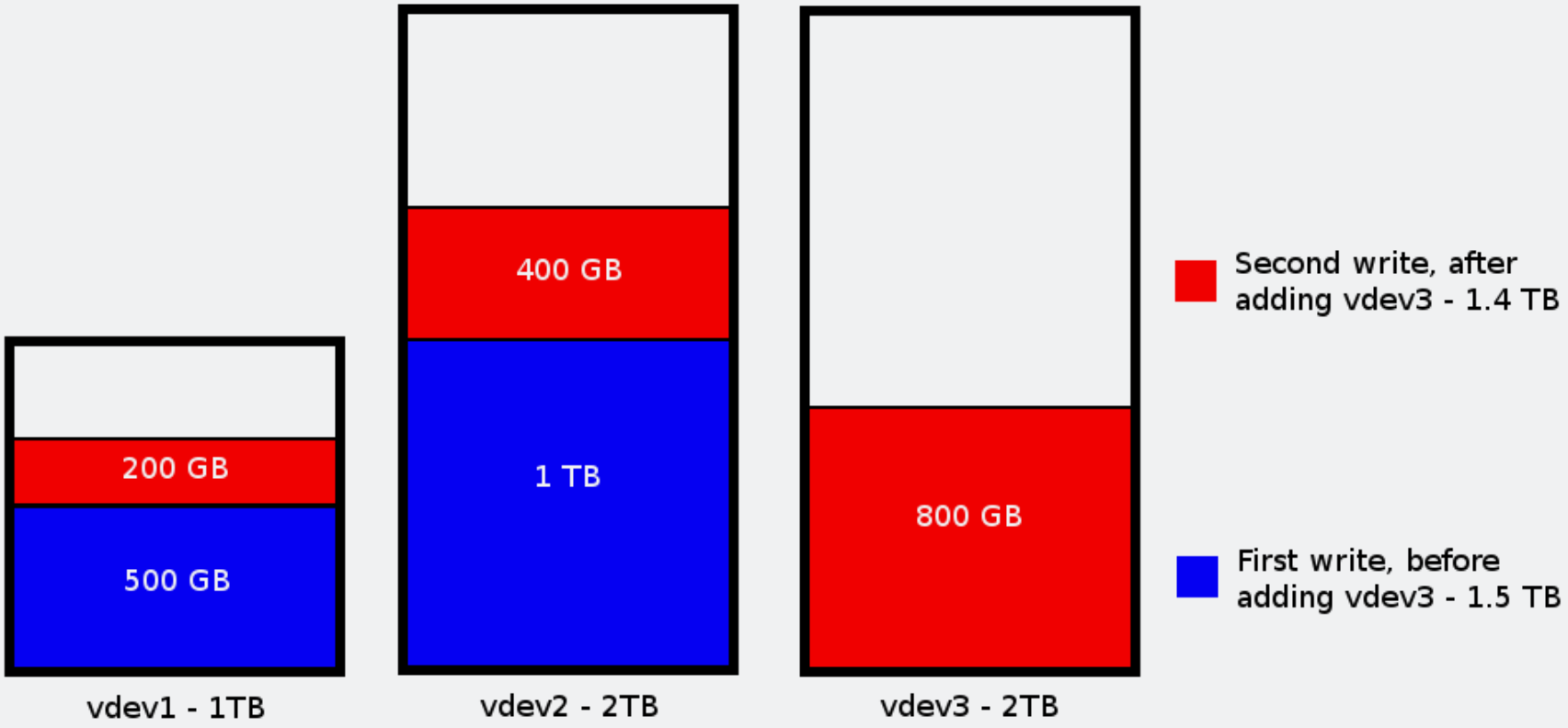
```
you@box:~$ sudo zpool status ars
  pool: ars
 state: ONLINE
 scrub: none requested
config:

        NAME                        STATE     READ WRITE CKSUM
        ars                         ONLINE       0     0     0
          mirror-0                  ONLINE       0     0     0
            wwn-0x50014ee2080259c8  ONLINE       0     0     0
```

```
            wwn-0x50014ee25d4cdecd  ONLINE       0     0     0
            wwn-0x50014ee25d4ce711  ONLINE       0     0     0
          wwn-0x50014ee25d4cr2d2    ONLINE       0     0     0

    errors: no known data errors
```

Keep in mind, the zpool stripes all data, including filesystem metadata, across *all* the top-level vdevs—and now, one of our top-level vdevs has no parity. This means that *any* failure on that single disk will bring the *whole array* down. All that space you devoted to parity on the first two vdevs won't do you any good at all if you lose a sector on the singleton!



ars TECHNICA

he 1.4TB of data we wrote after adding that third vdev went *to* that third vdev, which in this case is a single 2TB drive. This means that our performance just tanked hard, both on writes and on reads of the new data that's mostly stored on a single disk.

By the way, if you're thinking "well at least that first 1.5TB of data is safe," sorry, it isn't. Remember, *all* data, including filesystem metadata, is striped across all top-level vdevs with each new write. You lose that single disk after you added it, and your zpool will be left in an inconsistent state and will refuse to mount at all, no matter how prettily you cry.

Like I said, adding a single disk to a zpool is called *hating your data*. Don't do it. Please.

Page: 1 2 3 Next →

JIM SALTER

Jim Salter (@jrssnet) is an author, public speaker, small business owner, mercenary sysadmin, and father of three—not necessarily in that order.

TWITTER @jrssnet

← PREVIOUS STORY

NEXT STORY →

Related Stories

Today on Ars

Apple’s iOS 12 strategy: Take more time to squash the bugs

Teaser: Our Apollo series finale is coming tomorrow

Proposed NASA budget takes one small step toward the Moon

Daylight Saving Time isn’t worth it, European Parliament members say