Filesystems: the next generation

Jonni Bidwell provides a primer on ZFS and btrfs: two of the most talked-up filesystems around.

Last issue, we created a glorious NAS box with 24TB of drives set up as a RAID 6 array formatted as ext4 [see *Homebrew your own NAS, p46,* LXF192]. This issue, we'll show you how to set up an alternative filesystem.

While ext4 is fine for volumes up to 100TB, even principal developer Ted Ts'o admitted that the filesystem is just a stop-gap to address the shortcomings of ext3 while maintaining backwards-compatibility. Ext4 first appeared in the kernel in 2008; up until then the most exciting filesystem around was ReiserFS. It had some truly next-gen features, including combined B+ tree structures for file metadata and directory lists (similar to btrfs). However, interest in this filesystem flagged just a touch when its creator, Hans Reiser, was found guilty of murdering his wife. Development of its successor, Reiser4, continues in his absence, but the developers have no immediate plans for kernel inclusion.

However, we now have a new generation of filesystems, providing superior data integrity and extreme scalability. They break a few of the old rules too: traditional ideologies dictate that the RAID layer (be it in the form of a hardware controller or a software manager such as mdadm) should be independent of the filesystem and that the two should be blissfully ignorant of each other. But by integrating them we can improve error detection and correction – if only at the cost of traditionalists decrying 'blatant layering violations'.

> "Interest in ReiserFS flagged when its creator was found guilty of murdering his wife."

The (comparatively) new kids on the block are btrfs (B-tree filesystem: pronounced 'butter-FS' or 'better-FS'), jointly developed by Oracle, Red Hat, Intel, SUSE and many others, and ZFS, developed at Sun Microsystems prior to its acquisition by Oracle. ZFS code was originally released in 2005 as part of OpenSolaris, but since 2010 this has been disbanded and Oracle's development of ZFS in Solaris is closed source. Open source development continues as a fork, but since ZFS is licensed under the CDDL, and hence incompatible with the GPL, it's not possible to incorporate support into the Linux kernel directly. However, support via a third-party module is still kosher and this is exactly what the ZFS on Linux project (**http://zfsonlinux.org**) does. This project is largely funded by the Lawrence Livermore National Laboratory, which has sizeable storage requirements, so ZFS can support file sizes up to 16 exabytes (2^{24} TB) and volumes up to 256 zettabytes (2^{38} TB).

Being an out-of-tree module, ZFS will be sensitive to kernel upgrades. DKMS-type packages will take care of this on Debian-based Linux distros, Fedora, CentOS, and so on, but for other distros you'll need to rebuild the module every time you update your kernel.

Failure to do so will be problematic if your root filesystem is on ZFS. Ubuntu users will want to add the PPA **zfs-native/stable** and then install the package **ubuntu-zfs**. The ZFS on Linux homepage has packages and information for everyone else.

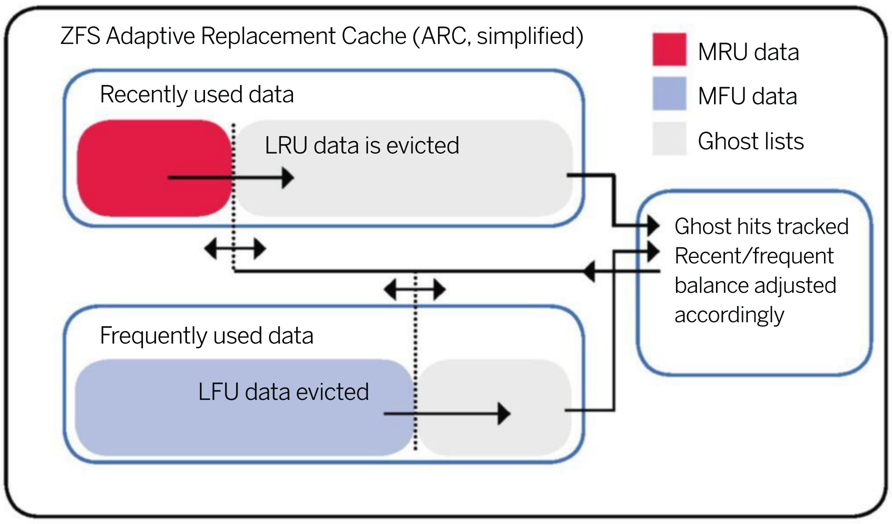
Let's cover the common ground first. One quite startling feature is that neither of these filesystems require disks to be partitioned. In ZFS parlance you can set up datasets within a single-drive zpool which offers more isolation than directories and can have quotas and other controls imposed. Likewise you can mimic traditional partitions using subvolumes within btrfs. In both cases the result is much more flexible – the 'neopartitions' are much easier to resize or combine since they are purely logical constructs. ZFS actively discourages its use directly on partitions, whereas btrfs largely doesn't care.

Both of the filesystems incorporate a logical volume manager, which allows the filesystem to span multiple drives and contain variously named substructures. Both also have their own RAID implementations, although, confusingly, their RAID levels don't really tie in with the traditional ones: ZFS has three levels of parity RAID, termed RAID-Z1, -Z2 and -Z3. These are, functionally, the same as RAID 5, RAID 6 and what would be RAID 7, meaning they use 1, 2 and 3 drives for parity and hence can tolerate that many drives failing. RAID 5 and 6 are supported in btrfs, but it would be imprudent to use them in a production environment, since that part of the codebase is significantly less mature than the rest. RAID 0, 1 and 10 support is stable in both filesystems, but again the levels have a slightly different interpretation. For example, a conventional RAID 1 array on three 1TB drives would mirror the data twice, making for a usable capacity of 1TB. With btrfs, though, RAID 1 means that each block is mirrored once on a different drive, making (in the previous example) for a usable capacity of 1.5TB at the cost of slightly less redundancy. You can also use multiple drives of different sizes with btrfs RAID 1, but there may be some unusable space (hence less than half of the total storage present is available) depending on the combinatorics. Additionally btrfs enables you to specify different RAID levels for data and metadata; ZFS features mirroring in much the same manner as RAID 1, but it does not call it that.

"Startlingly, neither of these filesystems require disks to be partitioned."

Mirroring with both of the filesystems is actually more advanced than traditional RAID, since errors are detected and healed automatically. If a block becomes corrupted (but still readable) on one drive of a conventional RAID 1 mirror and left intact on another, then mdadm has no way of knowing which drive contains the good data; half of the time the good block will be read, and half of the time you'll get bad data. Such errors are called silent data errors and are a scourge – after all, it's much easier to tell when a drive stops responding, which is what RAID mitigates against. ZFS stores SHA-256 hashes of each block and btrfs uses CRC32C checksums of both metadata and data. Both detect and silently repair discrepancies when a dodgy block is read. One can, and should, periodically perform a scrub of one's next-generation volumes. This is an online check (no need to unmount your pools), which runs in the background and does all the detecting and repairing for you.

All this CoW-ing (Copy-on-Writing) around can lead to extreme fragmentation, which would manifest itself through heavy disk thrashing and CPU spikes, but there are safeguards in place to minimise this. ZFS uses a slab allocator with a large 128k block size, while btrfs uses B-trees. In both approaches the idea is the same: to pre-allocate sensible regions of the disk to use for new data. Unlike btrfs, ZFS has no defragmentation capabilities, »


ZFS Adaptive Replacement Cache (ARC, simplified)
MRU data
MFU data
Ghost lists
Recently used data
LRU data is evicted
Ghost hits tracked Recent/frequent balance adjusted accordingly
Frequently used data
LFU data evicted

Caching in ZFS: two lists, for recently and frequently used data, share the same amount of memory. Most recently used (MRU) data is stored to the left and falls into the ghost list if not accessed. Memory is apportioned according to how often ghost entries are accessed.

A brief history of filesystems

In the beginning, data was stored on punch cards or magnetic tape. The concept of a file didn't exist: data was stored as a single stream. You could point to various addresses in that stream (or fast-forward, using the tape counter to find where you recorded something), but it was all essentially a single amorphous blob. Single-directory, or flat, filesystems emerged in the mid '80s. These enabled discrete files, but not subdirectories, to exist on a device. Their release coincided with increasing usage of floppy disks, which enabled random access of data (you can read/write at any region of the disk). Early Mac file managers abstracted a hierarchical directory structure on top of a flat filesystem, but this still required files to be uniquely named.

By the late '80s filesystems that enabled proper directories were necessary to support growing storage technologies and increasingly complex operating systems. These had in fact been around since the days of IBM PC-DOS 2, but the poster child for this generation is FAT16B, which allowed 8.3 filenames and volumes of up to 2GB. Windows 95 finally brought long filenames and the ability to access drives bigger than 8GB, but since 1993 Linux users had already seen these benefits thanks to ext2. This marked another step forward, featuring metadata such as file permissions, so that the filesystem becomes intrinsically linked with the user control mechanism. Ext3 and later revisions of NTFS introduced the next innovation: journaling, which allows filesystems to be easily checked for consistency, and quickly repaired following OS or power failure.

» which can cause serious performance issues if your zpools become full of the wrong kind of files, but this is not likely to be an issue for home storage, especially if you keep your total storage at less than about 60% capacity. If you know you have a file that is not CoW-friendly, such as a large file that will be subject to lots of small, random writes (let's say it's called **ruminophobe**), then you can set the extended attribute **C** on it, which will revert the traditional overwriting behaviour:

```
$ chattr +C ruminophobe
```

This flag is valid for both btrfs and ZFS, and in fact any CoW-supporting filesystem. You can apply it to directories as well, but this will affect only files added to that directory after the fact. Similarly, one can use the **c** attribute to turn on compression. This can also be specified at the volume level, using the **compress** mount option. Both offer zlib compression, which you shouldn't enable unless you're prepared to take a substantial performance hit. Btrfs offers LZO, which even if you're storing lots of already-compressed data won't do you much harm. ZFS offers the LZJB and LZ4 algorithms, as well as the naïve ZLE (Zero Length Encoding scheme) and the ability to specify zlib compression levels.

Note that while both btrfs and ZFS are next-generation filesystems, and their respective feature sets do intersect significantly, they are different creatures and as such have their own advantages and disadvantages, quirks and oddities.

Let's talk about ZFS, baby

The fundamental ZFS storage unit is called a vdev. This may be a disk, a partition (not recommended), a file or even a collection of vdevs, for example a mirror or RAID-Z set up with multiple disks. By combining one or more vdevs, we form a storage pool or zpool. Devices can be added on-demand to a zpool, making more space available instantly to any and all filesystems (more correctly 'datasets') backed by that pool. The image below shows an example of the ZFS equivalent of a RAID 10 array, where data is mirrored between two drives and then striped across an additional pair of mirrored drives. Each mirrored pair is also a vdev, and together they form our pool.

Let's assume you've got the ZFS module installed and enabled, and you want to set up a zpool striped over several drives. You must ensure there is no RAID information present on the drives, otherwise ZFS will get confused. The recommended course of action is then to find out the ids of those disks. Using the **/dev/sdX** names will work, but these are not necessarily persistent, so instead do:

```
# ls -l /dev/disk/by-id
```

and then use the relevant ids in the following command, which creates a pool called **tank**:

```
# zpool create -m <mountpoint> tank <ids>
```

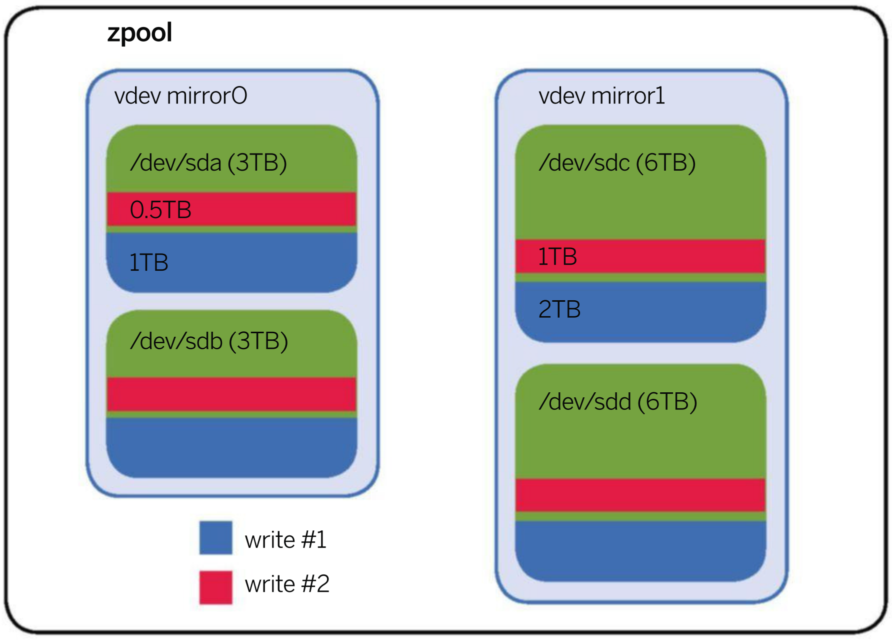
If your drives are new (post-2010), then they probably have 4kB sectors, as opposed to the old style 512 bytes. ZFS can cope with either, but some newer drives emulate the old-style behaviour so people can still use them in Windows 95, which confuses ZFS. To force the pool to be optimally arranged on newer drives, add **-o ashift=12** to the above command. You also don't have to specify a mountpoint: in our case, omitting it would just default to **/tank**. Mirrors are set up using the keyword **mirror**, so the RAID 10-style pool in the diagram (where we didn't have room to use disk ids but you really should) could be set up with:

```
# zpool create -o ashift=12 mirrortank mirror /
dev/sda /dev/sdb mirror /dev/sdc /dev/sdd
```

We can use the keyword **raidz1** to set RAID-Z1 up instead, replacing **1** with **2** or **3** if you want double or triple parity. Once created, you can check the status of your pool with:

```
# zpool status -v tank
```

You can now add files and folders to your zpool, as you would any other mounted filesystem. But you can also add filesystems (a different, ZFS-specific kind), zvols, snapshots and clones. These four species are collectively referred to as datasets, and ZFS can do a lot with datasets. A filesystem inside a ZFS pool behaves something like a disk partition, but is easier to create and resize (resize in the sense that you limit its maximum size with a quota). You can also set compression on a per-filesystem basis.



› **ZFS will stripe data intelligently depending on available space: after a 3TB write and then a 1.5TB write, all drives are half-full (or half-empty, depending on your outlook).**

Have a CoW, man

Even if you have no redundancy in your next-gen filesystem, it will be significantly more robust than its forbears. This is thanks to a technique called Copy-on-Write (CoW): a new version of a file, instead of overwriting the old one in-place, is written to a different location on the disk. When, and only when, that is done, the file's metadata is updated to point to the new location, freeing the previously occupied space. This means that if the system crashes or power fails during the write process, instead of a corrupted file, you at least still have a good copy of the old one Besides increased reliability, CoW allows for a filesystem (or more precisely a subvolume) to be easily snapshotted. Snapshots are a feature, or even the feature, that characterises our next-generation filesystems. A snapshot behaves like a byte-for-byte copy of a subvolume at a given time (for now think of a subvolume as a glorified directory – the proper definition is different for btrfs and ZFS), but when it is initially taken, it takes up virtually no space. In the beginning, the snapshot just refers to the original subvolume. As data on the original subvolume changes, we need to preserve it in our snapshot, but thanks to CoW, the original data is still lying around; the snapshot is just referred to the old data, so the filesystem will not mark those blocks as unused, and old and new can live side by side. This makes it feasible to keep daily snapshots of your whole filesystem, assuming most of its contents don't change too drastically. It is even possible to replicate snapshots to remote pools via SSH.

Let's create a simple filesystem called **stuff**. Note that our pool **tank** does not get a leading **/** when we're referring to it with the ZFS tools. We don't want it to be too big, so we'll put a quota of 10GB on there too, and finally check that everything went OK:

```
# zfs create tank/stuff
# zfs set quota=10G tank/stuff
# zfs list
```

A zvol is a strange construction: it's a virtual block device. A zvol is referred to by a **/dev** node, and like any other block device you can format it with a filesystem. Whatever you do with your zvol, it will be backed by whatever facilities your zpool has, so it can be mirrored, compressed and easily snapshotted. We've already covered the basics of snapshots (see *Have a CoW, man*), but there are some ZFS-specific quirks. For one, you can't snapshot folders, only filesystems. So let's do a snapshot of our **stuff** filesystem, and marvel at how little space it uses:

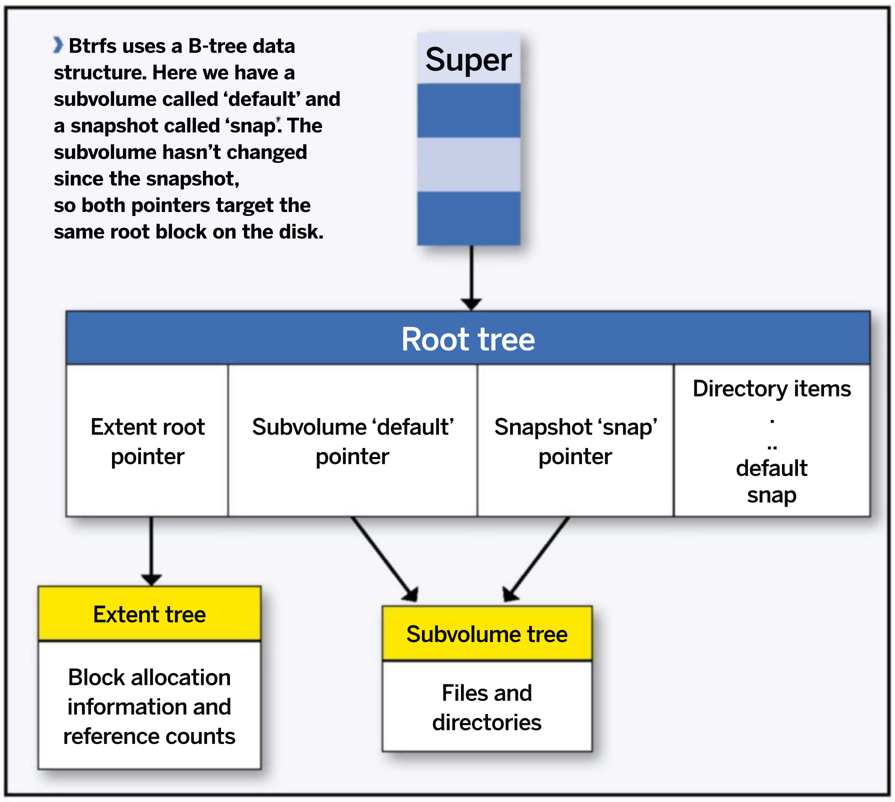
```
# zfs snapshot tank/stuff@snapshot0
# zfs list -t all
```

The arobase syntax is kind of similar to how a lot of systemd targets work, but let's not digress. You can call your snapshot something more imaginative than **snapshot0** – it's probably a good idea to include a date, or some indication of what was going on when the snapshot was taken. Suppose we now do something thoughtless resulting in our **stuff** dataset becoming hosed. No problem: we can roll back to the time of **snapshot0** and try and not make the same mistake again. The **zfs diff** command will even show files that are new (+), modified (M) or deleted (-) since the snapshot was taken:

```
# zfs diff tank/stuff@snapshot0
M       /pool/stuff
+       /pool/stuff/newfile
-       /pool/stuff/oldfile
# zfs rollback tank/stuff@snapshot0
```

Snapshots are read-only, but we can also create writable equivalents: the final member of the dataset quartet, called clones.

It would be remiss of us to not mention that ZFS works best with lots of memory. Some recommendations put this as high as a GB per TB of storage, but depending on your purposes you can get away with less. One reason for this is ZFS's Adaptive Replacement Cache. This is an improvement on the patented IBM ARC mechanism, and owing to its consideration of both recent and frequent accesses (shown in the diagram on *p49*) provides a high cache hit rate. By default it uses up to 60% of available memory, but you can tune this with the module option **zfs_arc_max**, which specifies the cache limit in bytes. If you use the deduplication feature then you really will need lots of memory – more like 5GB to the TB – so we don't recommend it. A final caveat: use ECC memory. All the benefits offered by ZFS checksums will be at best useless and at worst harmful if a stray bit is flipped while they're being calculated. Memory errors are rare but they do happen, whether it's dodgy hardware or stray cosmic rays to blame.


› **Btrfs uses a B-tree data structure. Here we have a subvolume called 'default' and a snapshot called 'snap'. The subvolume hasn't changed since the snapshot, so both pointers target the same root block on the disk.**

Btrfs me up, baby

As well as creating a new btrfs filesystem with mkfs.btrfs, one can also convert an existing ext3/4 filesystem. Obviously, this cannot be mounted at the time of conversion, so if you want to convert your root filesystem then you'll need to boot from a Live CD or a different Linux. Then use the **btrfs-convert** command. This will change the partition's UUID, so update your fstab accordingly. Your newly converted partition contains an image of the old filesystem, in case something went wrong. This image is stored in a btrfs subvolume, which is much the same as the ZFS filesystem dataset.

As in ZFS, you can snapshot only subvolumes, not individual folders. Unlike ZFS, however, the snapshot is not recursive, so if a subvolume itself contains another subvolume, then the latter will become an empty directory in the snapshot. Since a snapshot is itself a subvolume, snapshots of snapshots are also possible. It's a reasonable idea to have your root filesystem inside a btrfs subvolume, particularly if you're going to be snapshotting it, but this is beyond the scope of this article.

Subvolumes are created with:

```
# btrfs subvolume create <subvolume-name>
```

They will appear in the root of your btrfs filesystem, but you can mount them individually using the **subvol=<subvolume-name>** parameter in your fstab or mount command. You can snapshot them with:

```
# btrfs subvolume snapshot <subvolume-name> <snapshot-name>
```

You can force the snapshot to be read-only using the **-r** option. To roll back a snapshot:

```
# btrfs subvolume snapshot <snapshot-name> <subvolume-name>
```

If everything is OK then you can delete the original subvolume.

Btrfs filesystems can be optimised for SSDs by mounting with the keywords **discard** and **ssd**. Even if set up on a single drive, btrfs will still default to mirroring your metadata – even though it's less prudent than having it on another drive, it still might come in handy. With more than one drive, btrfs will default to mirroring metadata in RAID 1.

One can do an online defrag of all file data in a btrfs filesystem, thus:

```
# btrfs filesystem defragment -r -v /
```

You can also use the **autodefrag** btrfs mount option. The other piece of btrfs housekeeping of interest is **btrfs balance**. This will rewrite data and metadata, spreading them evenly across multiple devices. It is particularly useful if you have a nearly full filesystem and **btrfs add** a new device to it.

Obviously, there's much more to both filesystems. The Arch Linux wiki has great guides to btrfs (**http://bit.ly/BtrfsGuide**) and ZFS (**http://bit.ly/ZFSGuide**). LXF