

ZFS Images From Scratch, or `makefs -t zfs`

BY MARK JOHNSTON

For a long time, the FreeBSD project has made virtual machine (VM) disk images available on its download site: just go to <https://download.freebsd.org/snapshots/VM-IMAGES> to find a selection of pre-built images for download. These come in a variety of formats recognized by common hypervisors such as QEMU, VirtualBox, VMWare and bhyve. The FreeBSD project similarly distributes images for various cloud platforms, such as EC2, Azure and GCP. As a FreeBSD user, all you need to do is select the image and create an instance, and within a few seconds a fully installed FreeBSD system will be available.

For many users, the pre-built images are sufficient, but you might have special requirements that are not met by those images. In particular, until recently, all of the FreeBSD project's official images used UFS for the root filesystem. Of course, it was still possible to use ZFS in a VM by one of several strategies:

1. Keep the root filesystem on UFS but add extra disks and use them to back a ZFS pool.
2. Boot the FreeBSD installation media in a VM and use it to install FreeBSD on a virtual disk with ZFS as the root filesystem. The resulting VM image can be cloned and used as a template for other images.
3. Manually create an image by setting up a `md(4)` disk and then creating and importing a ZFS pool on top of that disk, into which FreeBSD can be installed. `poudriere image` currently works this way, for example.

While these strategies work, they all come with caveats:

- **option 1)** makes it difficult to use boot environments;
- **option 2)** requires manual effort to create and customize the template image;
- **option 3)** requires root privileges and cannot be done at all from within a jail (currently ZFS pool creation is forbidden in jails).

For a long time, I had wanted to build ZFS-based VM images locally so that I could run the FreeBSD regression test suite on both UFS and ZFS, so in 2022, I started looking at how difficult it would be to extend the tools we already use to make UFS images (i.e., `makefs(8)`) to support ZFS in some form.

`makefs(8)`

The FreeBSD project builds official VM images using `makefs(8)`, a utility originating from NetBSD. It takes one or more paths as input and generates a single file containing a filesystem image populated with the contents of those paths.

`makefs` supports several filesystems, including UFS, FAT and ISO9660. The basic idea behind its use here is to install FreeBSD to a temporary directory (e.g., with `make install-world`), and then point `makefs` at that directory. The result is a file containing a filesystem image whose root directory contains the FreeBSD installation.

For users familiar with building FreeBSD from source, the following example might provide a clearer picture:

```
# make installworld installkernel distribution DESTDIR=/tmp/foo
# makefs -t ffs fs.img /tmp/foo
# mdconfig -f fs.img
md0
# mount /dev/md0 /mnt
# ls /mnt/bin/sh
/mnt/bin/sh
```

This installs a pre-built copy of a FreeBSD distribution to `/tmp/foo` and then uses `makefs` to generate a filesystem image in `fs.img`. This image can be mounted by using `mdconfig(8)` to create a character device backed by the file. Attributes of the files in `/tmp/foo`, such as mode bits and timestamps, are preserved in the resulting image.

In contrast, a traditional “live” installation of FreeBSD might look more like this:

```
# truncate -s 50g fs.img
# mdconfig -f fs.img
md0
# newfs /dev/md0
/dev/md0: 51200.0MB (104857600 sectors) ...
# mount /dev/md0 /mnt
# cd /usr/src
# make installworld installkernel distribution DESTDIR=/mnt
# umount /mnt
```

Here, we use the `newfs(8)` utility to initialize an empty filesystem on the target device, then copy files onto it. While this works, of course, it has downsides similar to the problems with creating ZFS pools that I mentioned earlier: `newfs(8)` requires root privileges, and the resulting image is not reproducible. That is, if two images are created this way from the same pre-built FreeBSD distribution, they will not be byte-for-byte identical, for example, because the file access and modification times will be slightly different between the two images. Reproducibility is an important security property of build systems since it makes it easier to detect malicious tampering of build outputs.

I was already familiar with `makefs` from writing scripts to create VM images for my own use. As mentioned, I had wanted to be able to similarly build ZFS images, and I was not alone; a common user complaint was that all of FreeBSD’s official cloud images were UFS-based, even though ZFS is a very popular choice for the root filesystem on FreeBSD. So, I spent some time thinking about how `makefs`-generated ZFS images might look.

makefs(8) Meets ZFS

So what does `makefs` actually do? A technical answer to this question requires some knowledge of filesystem internals, but briefly: `makefs` initializes some global filesystem metadata, such as a UFS/FFS superblock, and then traverses the input directory tree(s), copying their contents into the image and adding metadata, such as directory entries, which point to file data. Whereas one traditionally starts with an empty filesystem and then asks the kernel to add data to it via utilities such as `cp(1)`, `makefs` generates a populated filesystem

tem in a single operation. So while this means that **makefs** needs to know about how a filesystem's data and metadata is arranged on disk, it can be considerably simpler than the kernel's implementation of the filesystem.

makefs never needs to look up a file by name, handle out-of-space conditions, perform buffer caching, or delete files, for example.

ZFS is large and complex, but per the observation above, a hypothetical **makefs -t zfs** can ignore a lot of the details. This was important for me: I was and am currently not a ZFS expert, and at the time, had little understanding of its on-disk format, so simplicity was the name of the game. At this point we can ask: what exactly should **makefs -t zfs** do?

My goal was to support creation of VM images with ZFS as the root filesystem. More specifically, **makefs** would need to:

1. Create a ZFS pool that has a single disk vdev. There is no need to support RAID or mirroring layouts, since for a VM image the extra redundancy is not very useful.
2. Create at least one dataset in the pool. The dataset needs to be mountable as the root filesystem. In practice, a FreeBSD-on-ZFS installation comes with a dozen or so datasets pre-created, but an initial proof-of-concept can ignore this for simplicity's sake.
3. Populate the dataset with the contents of the input directory trees. More specifically, for each input file, **makefs** needs to allocate a **dnode** and copy the file into the image somewhere. It also needs to copy attributes, such as file permissions, of the input files.

In particular, quite a few ZFS features which affect on-disk layout can simply be ignored. There is no need for **makefs** to bother with compression or snapshots, for example. So while the task still seemed somewhat daunting, by excluding all but the minimum necessary features, it seemed quite doable.

Attempt #1: libzpool

As a FreeBSD kernel developer I already had some experience with OpenZFS internals, but ZFS is a complex beast. The code is partitioned into quite a few different subsystems, most of which have no knowledge of how data actually gets laid out on disk, and I had no experience with the ones that do. However, it turns out that one can compile the OpenZFS kernel module into a userspace library: **libzpool.so**. This is used primarily for testing the OpenZFS code itself, but seemed like an excellent starting point for my project: **libzpool.so** knows all about the ZFS on-disk layout, so I thought I could avoid learning too much about it and instead write code that used high-level operations, similar to how commands like **zpool create** simply ask the kernel to create a pool on a set of **vdevs**.

Without going into too much detail, this approach ended up yielding a working prototype, but turned out to be a dead end. A few of the reasons:

- **libzpool.so** is really not suitable for "production" applications: it has no stable interface, and my prototype was effectively making use of undocumented kernel APIs. If I were to press on with this approach, the result would be fragile and difficult to maintain.
- The code in **libzpool.so** is mostly unmodified kernel code, and thus creates lots of

**ZFS is large
and complex,
but a hypothetical
makefs -t zfs
can ignore a lot
of the details**

threads and caches file data in the ARC, all of which is unnecessary for **makefs**'s purposes. A consequence of this is that the prototype was very slow and would consume gobs of system memory, sometimes triggering the out-of-memory killer.

- The result was not reproducible. If I ran the prototype twice with identical inputs, the output would not be byte-for-byte identical.

While I had to throw away most of the prototype, writing it was a useful learning experience and helped motivate me to try a different approach.

At this point, it seemed I would have to get my hands dirty and learn about the ZFS on-disk layout. I realized that the FreeBSD boot loader would have a similar problem: in order to boot FreeBSD from a ZFS pool, the loader needs to be able to find the kernel file and load it into memory. The boot loader runs in a constrained environment and thus cannot use the kernel's ZFS code, so clearly other people had already solved similar problems.

Attempt #2: ZFS From Scratch

Fortunately, there is a [ZFS on-disk layout specification](#) floating around the Internet; it is rather incomplete and outdated, but it was much better than nothing. On top of that, I had the boot loader code to look at. In some sense it solves the inverse problem that **makefs** does: it just opens and reads data from a ZFS pool without writing anything, whereas **makefs** creates a new pool but does not need to be able to read existing pools.

The duality with the boot loader was very useful: I could write code to create a pool, and then test it by trying to use the boot loader to read a file (the kernel) from the pool. More specifically, I would first install a FreeBSD kernel to a temporary directory:

```
$ cd /usr/src
$ make buildkernel
$ make installkernel DESTDIR=/tmp/test -DNO_ROOT
```

Then I can create a ZFS image and try to load it using the legacy bhyve loader:

```
$ makefs -t zfs -o poolname=zroot zfs.img /tmp/test
$ sudo bhyveload -c stdio -d zfs.img test
```

Here, **bhyveload** is using **/boot/userboot.so**, which is a copy of the FreeBSD boot loader that is compiled to run in userspace. It has most of the functionality of the real boot loader, but rather than using, say, BIOS calls or EFI boot services to read data from disk, it uses the familiar **read(2)** system call to fetch data from the image file, **zfs.img**.

The initial goal was to get **userboot.so** to find and load the kernel located at **/boot/kernel/kernel** in **zfs.img**. This was a very convenient test harness since I could easily attach a debugger to **bhyveload** or add print statements to the loader and recompile **userboot.so**. My first milestone was to get [vdev_probe\(\)](#) to recognize the image as a valid ZFS pool.

vdev Labels and the uberblock

vdev_probe() looks at a disk to see if it belongs to a ZFS pool; that is, it determines whether the disk appears to be a **vdev**, and if so, starts loading more metadata:

```
/*
 * Ok, we are happy with the pool so far. Lets find
 * the best uberblock and then we can actually access
```

```
* the contents of the pool.
*/
vdev_uberblock_load(vdev, spa->spa_uberblock);
```

Chapter 1 of the ZFS on-disk specification describes `vdev` labels and uberblocks in a good amount of detail. The summary is that a `vdev` contains a block of metadata, the `vdev` label, which contains metadata describing the pool to which the `vdev` belongs, as well as copies of the “uberblock”, which points to the root of the `vdev`’s metadata tree. So, in order to get `userboot.so` to find my pool, I wrote [code](#) which adds `vdev` labels to the output image file.

At this point `makefs` was already making use of ZFS-specific data structures, such as `vdev_label_t` and `uberblock_t`. Rather than duplicating the definitions used by the boot loader, `makefs` shares with it a [large header](#) that contains many useful on-disk data structure definitions.

Object Sets and the MOS

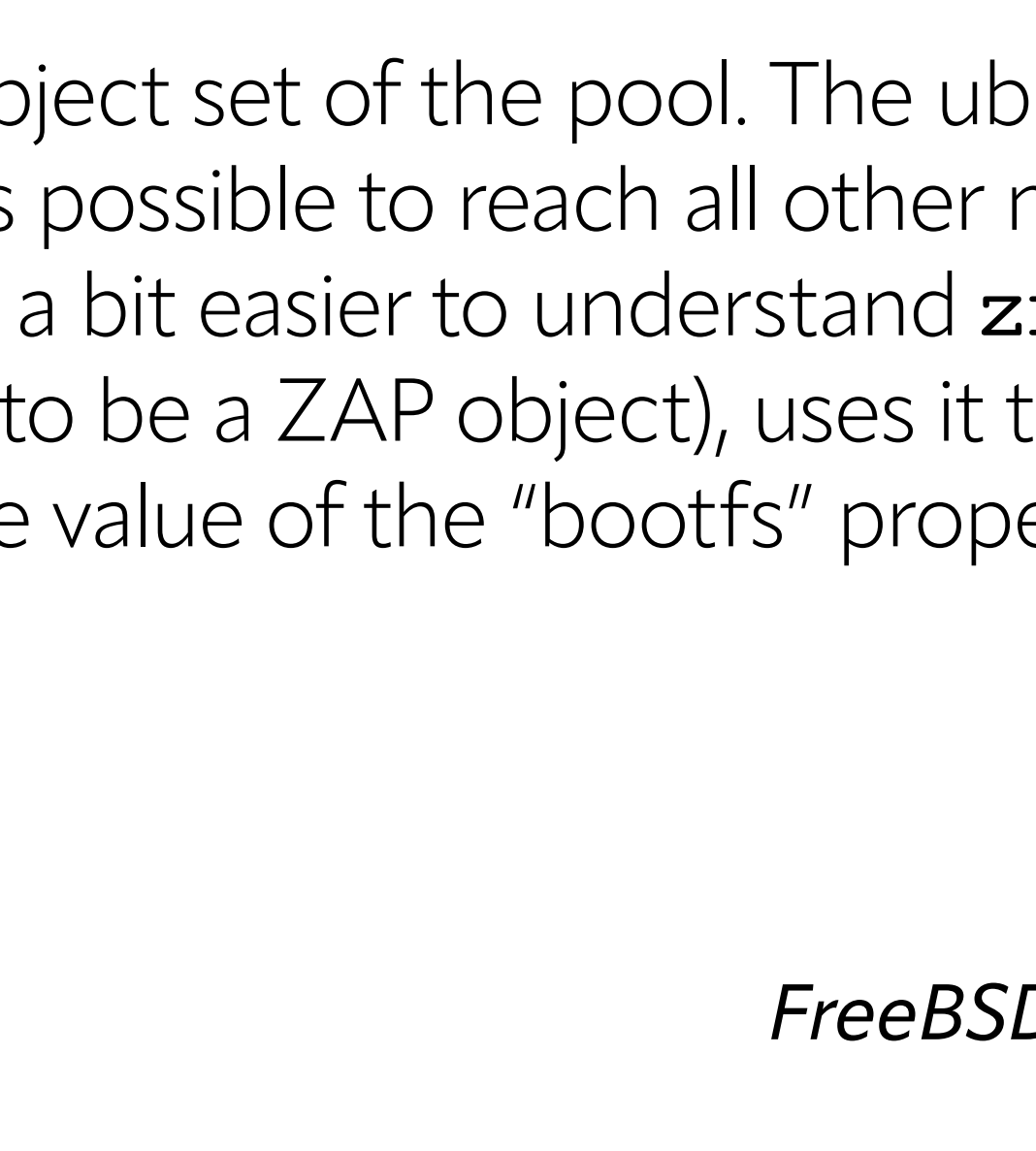
Once the loader was able to probe and recognize `makefs`-generated images, the next step was to get it to mount a dataset from within the image. The loader code that handles this is mostly contained in [zfs_get_root\(\)](#).

To understand the implementation of `zfs_get_root()`, it is worth reading chapter three of the ZFS on-disk specification, which describes object sets. While the specification quickly gets into the gory details, it is worth reviewing the high-level structures that are used to represent data in ZFS.

ZFS has “block pointers”, which really just refer to the physical location of a block of data on a `vdev` (from `makefs`’s perspective, this is just an offset into the output image file). A ZFS metadata object, of which there are several dozen types, is represented by a 512-byte “dnode”. A dnode contains various bits of metadata about the object, such as its type and size, and may also contain block pointers referring to additional data. For example, a file stored in a ZFS dataset is represented by a dnode (of type `DMU_OT_PLAIN_FILE_CONTENTS`), much like an inode in a traditional Unix filesystem. Finally, an “object set” is a structure which contains an array of dnodes; a dnode is uniquely identified by the object set to which it belongs and its index in the array.

A ZAP (ZFS Attribute Processor) is a dnode which contains a set of key-value pairs. ZAPs are used to represent many higher-level ZFS metadata structures. For example, a Unix directory is represented by a ZAP whose keys are filenames and values are dnode IDs for the corresponding files.

The MOS (meta object set) is the root object set of the pool. The uberblock contains a pointer to the MOS, and from the MOS it is possible to reach all other metadata (and thus, data) in the pool. With this information, it is a bit easier to understand `zfs_get_root()`: it takes the dnode with ID 1 (which it expects to be a ZAP object), uses it to find a ZAP object containing pool properties and looks up the value of the “bootfs” property, which is used to find the dnode of the root dataset.



The next step was to get it to mount a dataset from within the image.

When creating a pool, `makefs` allocates and begins populating the MOS in `pool_init()`. Once `userboot.so` was able to process the MOS, it became possible to import a `makefs`-generated pool, at which point I started using `zdb(8)` to inspect the generated pool. `zdb`'s command-line usage is rather obscure, but simple invocations like

```
# zdb -dddd zroot 1
```

which dumps dnode 1 from the MOS, were very useful for figuring out what OpenZFS expects to see when importing a pool. For example, when dumping a ZAP object, `zdb` can print all of the key-value pairs in the ZAP. Many configuration ZAP keys have values which are dnode IDs, so `zdb` can easily be used to inspect different "layers" of the pool and dataset configuration.

Datasets and Files

ZFS datasets have names and are organized into a tree. The root dataset is named after the pool itself (e.g., "zroot"), and names of child datasets are prefixed by the parent's name. While my initial prototype of ZFS support for `makefs` automatically placed all files in the root dataset, this was not sufficient to be able to create root-on-ZFS VM images:

`bsdinstall` and other FreeBSD installers automatically create a number of child datasets. Some, such as `zroot/var`, are never mounted but only exist to provide settings which are inherited by child datasets, such as `zroot/var/log`. My goal was for `makefs` to be able to create a tree of datasets which matches the layout provided by `bsdinstall`.

The release image-building script [demonstrates](#) the syntax for creating multiple datasets. Each dataset is described by a `-o fs` option which contains the dataset name and a semicolon-separated list of properties. Only a small number of properties - as described in the `zfsprops(8)` manual page — are currently supported.

When `makefs -t zfs` finishes initializing various structures, it begins to [process](#) the input directory trees. Each input file is represented by a `fsnode` structure, and these structures are organized into a tree which represents the file tree. First, `makefs` determines which `fsnode` corresponds to the root of each mounted dataset. Then, it traverses the tree of `fsnodes`, allocating a dnode for each file; this happens in the context of a dataset which determines the object set from which the dnode is allocated.

To [copy a regular file](#) `makefs` allocates a dnode from the current object set and, in a loop, allocates blocks of space from the output file, and copies data from the input file into the allocations. ZFS supports power-of-2 block sizes ranging from 4KB to 128KB, so smaller files do not create excessive internal fragmentation. All allocated blocks in the image are tracked using a bitmap which is updated by the `vdev_space_alloc()` function.

Allocated space tracked by the bitmap must be recorded in the output image; ZFS uses a central data structure, the "space map," to track which regions of a vdev are currently allocated. `makefs` uses the bitmap as an internal representation of all block allocations, and uses it to generate the space map as one of the final steps of image generation, once all block allocations have been done.

The root dataset is named after the pool itself and names of child datasets are prefixed by the parent's name.

Conclusion

Adding ZFS support to `makefs` took a fair bit of effort but ultimately resulted in an implementation that I believe will be useful to many FreeBSD users, while avoiding a large maintenance burden. There is roughly 2,600 lines of ZFS-specific code in `makefs` (out of 15,000 lines in total), which is reasonably small. There is also a [regression test suite](#) which provides a good amount of coverage.

Of those 2,600 lines, over 100 are calls to `assert()` and so simply verify invariants. These assertions were very useful during development, since a lot of code was written in an incomplete manner just to get the boot loader working, and fleshed out more fully later on; they served to document the limitations of various functions and helped catch many bugs as I added more and more functionality.

Now that FreeBSD 14.0 has shipped and root-on-ZFS VM images are available, I hope that many users are taking advantage of this new feature. A number of bugs were found and fixed during the release cycle, so at least some users have been trying it out. Currently there are no enhancements planned for `makefs -t zfs` but this may change in response to feedback — please submit a bug report if you see any room for improvement.

MARK JOHNSTON is a software developer and FreeBSD src developer living in Toronto, Ontario, Canada. When not sitting in front of a computer he enjoys playing in a city dodgeball league with friends.

Write For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)

