TOM HAYNES

introduction to ZFS

Tom Haynes is an NFS developer for Sun Microsystems, Inc., and is interested in the cost differential between open source and commercial offerings. He is exploring those costs by using OpenSolaris to design a NAS appliance.

tdh@excfb.com

THE ZETTABYTE FILE SYSTEM (ZFS) IS the replacement file system for UFS. In a nutshell, ZFS creates pools of data across multiple disks. It manages the complexity of formatting, partitioning, mirroring, and other tasks for the administrator.

You can search on Google and find many glowing testimonials about how ZFS was deployed, about how easy it was, about how great the software is, etc. But how much fun is it to just read about everything working out as expected? How often does that occur in your experience? Do we tune into "I Shouldn't Be Alive" on the Discovery Channel or "I Haven't Died Yet" on the Established Channel?

When I proposed this article, I wanted to write about a 1-terabyte NAS file server based on OpenSolaris. To minimize cost, all of the components were to be commodity parts and the drives would be SATA. What I'm going to write about is an exploration of ZFS on a 300 GB IDE drive. Oh, and I'm going to illustrate how the best-laid plans go astray. I'm also not going to define all of the ZFS or filesystem terminology—again, you can pick this stuff up online.

When you deploy either Solaris 10 or OpenSolaris on hardware not manufactured by Sun Microsystems, you need to do some research for compatibility. The best resource is the BigAdmin HCL, maintained at http://www.sun.com/bigadmin/hcl/. This Hardware Compatibility List details experiences with various x86 systems and components with the different flavors of Solaris. I picked the MSI K8N Master2-FAR motherboard because of the support for the NVIDIA nForce4 chip set, the support for the two GigE Ethernet controllers, and the ability to support four SATA drives without an additional controller card. Note that this MSI MB utilizes an NVIDIA nForce4 Host Bus Adapter. The Sun Ultra 20 also utilizes NVIDIA nForce chip sets to handle the HBA duties.

Right after I ordered this MB, bug 6363449 was filed on the Ultra 20. Basically, the NVIDIA nForce4 gets confused with the ZFS label written to the SATA drives. There are some measures mentioned in the bug report to get the drives working, but they do not work on my MB.

I had finally constructed my system, loaded Nevada b27 on it, done some fun things with ZFS, and powered the machine off for the night. That's when I found out about the bug. See, the fan was very loud. The system would not reboot the next morning. Considering the bad luck I had with the system, I named it wont, as in "wont work." I was able to identify the bug with help on the OpenSolaris discussion forums. I tried booting with the drive entries set to "none" in the BIOS, but still no joy. I disconnected the SATA drives and the system booted fine. By the way, the SATA connectors are very fragile; I broke one, and I would advise you not to reinsert the cables too often.

A limiting factor in getting parts working in a home office is that you might just have one of everything. I don't have another system in which I can put a different VTOC on the drives. (Just like I only had one power supply, one MB, one case, and one video card when I was troubleshooting the original reason the system would not boot: The video card was not compatible with the MB.)

I actually learned a lot about OpenSolaris during this very frustrating process. Among other things, I figured out how to use kmdb (kernel debugger), how to boot the system into the console from grub, how to wire the console, and how to force a core.

So I've hit the cutting edge of OpenSolaris and it appears I have two choices:

1. Convince the Solaris SATA developers that the bug needs to be fixed ASAP.
2. Hunker down and fix the issue myself.

The only reason there was any urgency on this bug was the deadline for this article. And I've been too busy with my new job to tackle the code myself.

But is there a third choice, besides RMAing the whole mess and trying my luck again?

Yes, there is actually a cheap alternative—just add another IDE drive. ZFS is quite capable of working with slices and not just disks. Sure, you introduce a single point of failure and bypass many of the benefits of having mirrored storage. But the goal is to play with ZFS, and to do so cheaply. I must admit I struggled with this decision; I'm used to NAS boxes that have a single storage partition spread over multiple disks, not a NAS box that has multiple storage partitions spread across a single disk.

I took a 300 GB IDE drive and created four equal slices of 68 GB. You can do this with the following format:

Note that, under OpenSolaris, disks are assigned names of the form controller

```
# format
Searching for disks...done

AVAILABLE DISK SELECTIONS:
        0. c0d0 <DEFAULT cyl 4862 alt 2 hd 255 sec 63>
                /pci@0,0/pci-ide@6/ide@0/cmdk@0,0
        1. c0d1 <DEFAULT cyl 36477 alt 2 hd 255 sec 63>
                /pci@0,0/pci-ide@6/ide@0/cmdk@1,0
Specify disk (enter its number): 1
format> partition
partition> p
Current partition table (original):
Total disk cylinders available: 36477 + 2 (reserved cylinders)

Part  Tag       Flag    Cylinders        Size                    Blocks
0     stand     wm      3 -  8879        68.00 GB        (8877/0/0)      142609005
1     stand     wm      8880 - 17756     68.00 GB        (8877/0/0)      142609005
```

```
2   backup      wm   0 - 36476       279.43 GB   (36477/0/0)   586003005
3   stand       wm   17757 - 26633   68.00 GB    (8877/0/0)    142609005
4   stand       wm   26634 - 35510   68.00 GB    (8877/0/0)    142609005
5   stand       wm   35510 - 36476   7.41 GB     (967/0/0)     15534855
6   unassigned  wm   0               0           (0/0/0)       0
7   unassigned  wm   0                           0             (0/0/0)     0
8   boot        wu   0 -             0           7.84 MB       (1/0/0)     16065
9   alternates  wu   1 -             2           15.69 MB      (2/0/0)     32130
```

ID and disk ID. So "c0d1" is the slave on the first controller. We can further reference the different slices on the disk. For now, think of slices as partitions. It isn't entirely accurate, but it is the concept we want to work with here.

The first thing we can try is to create a ZFS pool; if we were using entire disks, we could think of the pool as a volume of disks. If we were to add RAID to the mix, you would then be able to remove a disk from the volume, if it failed, and replace it with a spare. The file system would then rebuild the missing data on that new disk.

For right now, we want to construct a pool of storage that is larger than any single available unit. Perhaps we need some scratch space for a computational job.

```
# zpool create zoo c0d1s0 c0d1s1
# zpool list
NAME   SIZE   USED   AVAIL  CAP   HEALTH     ALTROOT
Zoo    135G   57.5K  135G   0%    ONLINE     -
```

So the system has a 135GB pool to use for storage. What this means is that the data can span the two slices. With this configuration, there is no redundancy.

We could instead have created a mirrored pool—one that halves your available storage but keeps an exact copy of the contents. Under this model, if one side becomes corrupt, you can break the mirror and activate the other side. With normal RAID configurations, you can survive a single disk failure. Mirroring allows you to survive multiple disk failures on one of the sides.

```
# zpool destroy zoo
# zpool create zoo mirror c0d1s0 c0d1s1
# zpool list
NAME   SIZE   USED   AVAIL  CAP   HEALTH     ALTROOT
zoo    67.5G  57.5K  67.5G  0%    ONLINE     -
```

Note that the mirror does indeed halve the storage. Also, we lost some space for ZFS overhead. Perhaps we want to add some additional storage:

```
# zpool add zoo c0d1s2 c0d1s3
invalid vdev specification
use '-f' to override the following errors:
/dev/dsk/c0d1s2 overlaps with /dev/dsk/c0d1s0
# zpool add zoo mirror c0d1s3 c0d1s4
invalid vdev specification
use '-f' to override the following errors:
/dev/dsk/c0d1s4 contains a ufs filesystem.
/dev/dsk/c0d1s4 overlaps with /dev/dsk/c0d1s5
```

The zpool command is keeping me from shooting myself in the foot. Slice 2 should never be used, and slice 4 earlier had a UFS file system. That should be easy to fix, but I'm more concerned with the data that exist on slice 5. Notice that it was just when I moved to ZFS that I found out about the overlap. I'm in the process of exploring how OpenSolaris DVDs are made bootable, and /dev/dsk/c0d1s5 contains the contents of the x86 DVD—see http://www.kanigix.org for more details on this project. So if I lose the data, I have it on DVD.

Now let's make the new file system real and use it to save the data. We have a ZFS pool, but now we need to create a file system on that pool and allow it to be utilized. A good question is, Why take the extra step? Why not make the pool the base unit? The reason is that we want to be able to store multiple file systems in a pool. What if we want to clone a file system? What if we want to take a snapshot of a file system? Taking this design path from the start saves the pain of trying to retrofit this functionality later—say, when many customers have vital data to be protected during an upgrade.

```
# zfs create zoo/x86
# df -h | grep zoo
zoo        67G     99K     67G     1%      /zoo
zoo/x86    67G     98K     67G     1%      /zoo/x86
# ls -la /zoo
total 6
drwxr-xr-x        3 root          sys     3 Mar 19 23:16 .
drwxr-xr-x        42 root         root    1024 Mar 19 23:08 ..
dr-xr-xr-x        3 root          root    3 Mar 19 23:17 .zfs
drwxr-xr-x        2 root          sys     2 Mar 19 23:16 x86
```

ZFS created the file system and mounted it for me. One of the ease-of-use factors of ZFS is that it automates many of the manual steps used with creating other file systems and making them ready for use.

I can use cpio to safely copy the data over to the new file system:

```
# chown tdh:staff /zoo/x86
# cd /kanigix/
# find . -depth -print | cpio -pudm /zoo/x86
6608816 blocks
# df -h /kanigix /zoo /zoo/x86
Filesystem        size    used    avail   capacity Mounted on
/dev/dsk/c0d1s5   7.3G    3.1G    4.1G    44%      /kanigix
zoo               67G     99K     64G     1%       /zoo
zoo/x86           67G     3.2G    64G     5%       /zoo/x86
```

Notice that although /zoo and /zoo/x86 appear to be different file systems, they share the same storage.

We copied the data over because we need to recreate the slice on which it resided—slices 4 and 5 shared a block. We now need to fix the two slices (remembering to comment out the entry in /etc/vfstab). After using format (and the subcommand of partition), these slices now look like this:

```
4  stand    wm   26634 - 35510    68.00GB    (8877/0/0)     142609005
5  stand    wm   35511 - 36476    7.40GB     (966/0/0)      15518790
```

Although I modified slice 5, I did not do so for slice 4. zpool will still think there is a valid UFS file system on that slice, so we need to force it to use that slice:

```
# zpool add -f zoo mirror c0d1s3 c0d1s4
# zpool list
NAME    SIZE    USED    AVAIL   CAP     HEALTH          ALTROOT
zoo     135G    3.21G   132G    2%      ONLINE          -
# df -h /zoo /zoo/x86
```

```
Filesystem       size    used    avail   capacity       Mounted on
zoo              134G    99K     131G    1%             /zoo
zoo/x86          134G    3.2G    131G    3%             /zoo/x86
```

We are now using about 268GB of raw disk space to provide a mirrored pool. Again, by using a single disk, the mirroring will only provide minimal benefit. Conceivably, someone could corrupt the slices with the format command—but we don't expect that. But as a cheap tour of the ZFS feature set, this setup works.

A common ZFS task is to create NFS exported home directories with a quota. We use inheritance to say that any file systems created inside /export/zfs are to exported via NFS, are to be compressed, and will have a 10GB quota. Note that we are creating file systems within other file systems. We are setting defaults, which can be overridden at any time.

```
# zfs create zoo/home
# zfs set mountpoint=/export/zfs zoo/home
# zfs set sharenfs=on zoo/home
# zfs set compression=on zoo/home
# zfs set quota=10G zoo/home
# zfs create zoo/home/nfsv2
# zfs create zoo/home/nfsv3
# zfs create zoo/home/nfsv4
# zfs list
NAME             USED    AVAIL   REFER   MOUNTPOINT
zoo              3.21G   131G    99.5K   /zoo
zoo/home         395K    10.0G   99.5K   /export/zfs
zoo/home/nfsv2   98.5K   10.0G   98.5K   /export/zfs/nfsv2
zoo/home/nfsv3   98.5K   10.0G   98.5K   /export/zfs/nfsv3
zoo/home/nfsv4   98.5K   10.0G   98.5K   /export/zfs/nfsv4
zoo/x86          3.21G   131G    3.21G   /zoo/x86
```

One thing to note here is that zoo/x86 is only available as /zoo/x86. Since it is not under zoo/home, the defaults we provided do not apply. Also note that it is not exported. Finally, if we do go to /zoo, we will not see "home."

And we check that the home directories are exported on the box wont:

```
[tdh@adept ~]> showmount -e wont
Export list for wont:
/export/zfs         (everyone)
/export/zfs/nfsv2   (everyone)
/export/zfs/nfsv3   (everyone)
/export/zfs/nfsv4   (everyone)
```

By the way, I never enabled the NFS server on wont. I know how to do it, but I did not have to do anything, since ZFS did it for me. Note that I am responsible for creating user accounts and changing ownership of the root of the file systems.

A cautionary note here is that the quotas are on the file system and not per user. ZFS does a lot for you behind the scenes, but it doesn't know that these are user accounts we are creating. So if the user nfsv2 were to copy files under the /export/zfs/nfsv3 file system, the charge would be against the file system and not against either of the two user accounts.

```
# useradd -m -u 1094 -g 100 -c "Mr. NFSv2" -d /export/zfs/nfsv2 nfsv2
# chown nfsv2:100 /export/zfs/nfsv2
# ls -al /export/zfs
total 10
```

```
drwxr-xr-x       5 root    sys      5 Mar 20 00:33 .
drwxr-xr-x       4 root    sys      512 Mar 20 00:31 ..
dr-xr-xr-x       3 root    root     3 Mar 20 00:40 .zfs
drwxr-xr-x       2 nfsv2   protos   2 Mar 20 00:33 nfsv2
drwxr-xr-x       2 nfsv3   protos   2 Mar 20 00:33 nfsv3
drwxr-xr-x       2 nfsv4   protos   2 Mar 20 00:33 nfsv4
```

We can test snapshots to see whether we can safeguard our data, in this case a copy of this article. When you take a snapshot of a file system, you are basically telling the OS that if the contents are changed, keep a copy of the old contents. This copy stays until the snapshot is deleted.

There are different ways to achieve this, but a common approach employs copy-on-write. Initially the two file systems (the original and the copy) point to the same inodes and blocks. The savings here is that the snapshot consumes minimal storage. We can see that here when we create the snapshot:

```
# zfs snapshot zoo/home/nfsv4@monday
# zfs list
NAME                    USED    AVAIL   REFER   MOUNTPOINT
zoo                     3.21G   131G    99.5K   /zoo
zoo/home                404K    10.0G   100K    /export/zfs
zoo/home/nfsv2          98.5K   10.0G   98.5K   /export/zfs/nfsv2
zoo/home/nfsv3          98.5K   10.0G   98.5K   /export/zfs/nfsv3
zoo/home/nfsv4          108K    10.0G   108K    /export/zfs/nfsv4
zoo/home/nfsv4@monday   0       -       107K    -
zoo/x86                 3.21G   131G    3.21G   /zoo/x86
```

The accounting shows that only zoo/home/nfsv4 has any storage. When the contents are changed, the original blocks are weaved into the snapshot space and the new ones are created inside the live file system. We can see that when we delete the file:

```
> ls -la
total 23
drwxr-xr-x    2 nfsv4    protos    4 Mar 20 01:38       .
drwxr-xr-x    5 root     sys       5 Mar 20 00:33       ..
dr-xr-xr-x    3 root     root      3 Mar 20 01:43       .zfs
-rw-r--r--    1 nfsv4    protos    0 Mar 20 01:04       it
-rw-r--r--    1 nfsv4    protos    11808 Mar 20 01:38 zfs.txt
> rm zfs.txt
> ls -la
total 5
drwxr-xr-x    2 nfsv4    protos    3 Mar 20 01:43       .
drwxr-xr-x    5 root     sys       5 Mar 20 00:33       ..
dr-xr-xr-x    3 root     root      3 Mar 20 01:43       .zfs
-rw-r--r--    1 nfsv4    protos    0 Mar 20 01:04       it
> zfs list | grep nfsv4
zoo/home/nfsv4            206K 10.0G   98.5K   /export/zfs/nfsv4
zoo/home/nfsv4@monday     107K -       108K    -
```

The storage has now transferred over to the snapshot. Also, the snapshot storage is coming from the containing file system. Note how the other numbers (USED and REFER) increased.

We can recover either the entire snapshot or just the file. To get the file back:

```
> ls -la .zfs/snapshot/monday/
total 21
```

```
drwxr-xr-x     2 nfsv4  protos   4 Mar 20 01:38 .
dr-xr-xr-x     3 root   root     3 Mar 20 01:43 ..
-rw-r--r--     1 nfsv4  protos   0 Mar 20 01:04 it
-rw-r--r--     1 nfsv4  protos   11808 Mar 20 01:38 zfs.txt
> cp .zfs/snapshot/monday/zfs.txt .
> ls -la
total 6
drwxr-xr-x     2 nfsv4  protos   4 Mar 20 01:44 .
drwxr-xr-x     5 root   sys      5 Mar 20 00:33 ..
dr-xr-xr-x     3 root   root     3 Mar 20 01:44 .zfs
-rw-r--r--     1 nfsv4  protos   0 Mar 20 01:04 it
-rw-r--r--     1 nfsv4  protos   11808 Mar 20 01:44 zfs.txt
```

At first the snapshot consumed no space, but as we caused it to deviate from the original, it was forced to keep the content.

```
> zfs list | grep monday
zoo/home/nfsv4@monday  106K    -       107K    -
```

As we change the copy in the live file system, we can see that the two files differ:

```
> diff zfs.txt .zfs/snapshot/monday/zfs.txt  | wc -l
    55
```

As alluded to earlier, we could restore the entire snapshot. Perhaps an errant script did an rm -rf or a virus corrupted everything. With our example:

```
# zfs rollback zoo/home/nfsv4@monday
# zfs list | grep nfsv4
zoo/home/nfsv4         108K    10.0G   108K    /export/zfs/nfsv4
zoo/home/nfsv4@monday  0       -       108K    -
```

I have tried to provide a taste of what ZFS can do for you and how you do not need to spend a lot of money on disks to take it for a spin. I did not explore all of the features—for example, creating a RAID pool, backing up a snapshot to tape, or cloning a file system. I showed perhaps the most common example, creating user accounts, and while I could have picked something different, for example, staging areas for external Web servers, I picked it for a reason.

When I taught undergraduate CS courses, I would have loved the ability to couple ZFS with Zones. Imagine that each student or group has its own virtual server and its own file system. One student cannot inadvertently rob the rest of the use of the machine and students cannot go look at each other's source code. They cannot complain that they accidentally deleted their files (i.e., a snapshot will keep that dog away from their homework). Also, if the due time is 5 p.m., just take a snapshot of the file systems. There is no need to worry about some industrious student changing the timestamps on the files.