ZFS

Internals

“A Compilation of blog posts”

Version 0.1 (Draft)

- [Leal’s blog] -
http://www.eall.com.br/blog

ZFS Internals (part #1)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INCACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

Do we have a deal? ;-)

A few days ago i was trying to figure it out how ZFS copy-on-write semantics really works, understand the ZFS on disk layout, and my friends were the zfsondiskformat specification, the source code, and zdb. I did search on the web looking for something like what i´m writing here, and could not find anything. That´s why i´m writing this article, thinking it can be useful for somebody else.
Let´s start with a two disks pool (disk0 and disk1):

```
# mkfile 100m /var/fakedevices/disk0
# mkfile 100m /var/fakedevices/disk1
# zpool create cow /var/fakedevices/disk0 /var/fakedevices/disk1
# zfs create cow/fs01
```

The recordsize is the default (128K):

```
# zfs get recordsize cow/fs01
NAME      PROPERTY    VALUE     SOURCE
cow/fs01  recordsize  128K      default
```

Ok, we can use the THIRDPARTYLICENSEREADME.html file from “/opt/staroffice8/” to have a good file to make the tests (size: 211045). First, we need the object ID (aka inode):

```
# ls -i /cow/fs01/
         4 THIRDPARTYLICENSEREADME.html
```

Now the nice part...

```
# zdb -dddddd cow/fs01 4
... snipped ...
 Indirect blocks:
               0 L1  0:9800:400 1:9800:400 4000L/400P F=2 B=190
               0  L0 1:40000:20000 20000L/20000P F=1 B=190
           20000  L0 0:40000:20000 20000L/20000P F=1 B=190

                segment [0000000000000000, 0000000001000000) size   16M
```

Now we need the concepts in the zfsondiskformat doc. Let´s look the first block line:

```
0 L1  0:9800:400 1:9800:400 4000L/400P F=2 B=190
```

The **L1** means two levels of indirection (number of block pointers which need to be traversed to arrive at this data). The "**0:9800:400**" is: the device where this block is (**0** = /var/fakedevices/disk0), the offset from the begining of the disk (**9800**), and the size of the block (0x**400** = 1K), respectivelly. So, ZFS is using two disk blocks to hold pointers to file data...

ps.: **0:9800** is the Data virtual Address 1 (dva1)

At the end of the line there are two other important informations: **F=2**, and **B=190**. The first is the fill count, and describes the number of non-zero block pointers under this block pointer. Remember our file is greater than 128K (the default recordsize), so ZFS needs two blocks (**FSB**), to hold our file. And the second is the birth time, what is the same as the txg number(**190**), that creates that block.

Now, let´s get our data! Looking at the second block line, we have:

```
0  L0 1:40000:20000 20000L/20000P F=1 B=190
```

Based on zfsondiskformat doc, we know that **L0** is the block level that holds data (we can have up to six levels). And in this level, the fill count has a little different interpretation. Here the **F=** means if the block has data or not (0 or 1), what is different from the levels 1 and above, where it means "how many" non-zero block pointers under this block pointer. So, we can see our data using the -R option from zdb:

```
# zdb -R cow:1:40000:20000 | head -10
Found vdev: /var/fakedevices/disk1

cow:1:40000:20000
          0 1 2 3 4 5 6 7   8 9 a b c d e f  0123456789abcdef
000000:  505954434f44213c  50206c6d74682045 !DOCTYPE html P
000010:  2d222043494c4255  442f2f4333572f2f  UBLIC "-//W3C//D
000020:  204c4d5448204454  6172542031302e34  TD HTML 4.01 Tra
000030:  616e6f697469736e  2220224e452f2f6c  nsitional//EN" "
000040:  772f2f3a70747468  726f2e33772e7777  http://www.w3.or
000050:  6d74682f52542f67  65736f6f6c2f346c  g/TR/html4/loose
```

That´s nice! 16 bytes per line, that is our file. Let´s read it for real:

```
# zdb -R cow:1:40000:20000:r
... snipped ...
The intent of this document is to state the conditions under which
VIGRA may be copied, such that the author maintains some
semblance of artistic control over the development of the library,
while giving the users of the library the right to use and
distribute VIGRA in a more-or-less customary fashion, plus the
right to
```

ps.: Don´t forget that is the first 128K of our file...

We can assemble the whole file like this:

```
# zdb -R cow:1:40000:20000:r 2> /tmp/file1.dump
# zdb -R cow:0:40000:20000:r 2> /tmp/file2.dump
# cat /tmp/file2.dump >> /tmp/file1.dump
# diff /tmp/file1.dump /cow/fs01/THIRDPARTYLICENSEREADME.html
Warning: missing newline at end of file /tmp/file1.dump
5032d5031
<
```

Ok, that´s warning is something we can understand. But let´s change something on that file, to see the copy-on-write in action... we will use VI to change the

“END OF TERMS AND CONDITIONS” line (four lines before the EOF), to “FIM OF TERMS AND CONDITIONS”.

```
#  vi THIRDPARTYLICENSEREADME.html
# zdb -dddddd cow/fs01 4
... snipped ...
Indirect blocks:
               0 L1  0:1205800:400 1:b400:400 4000L/400P F=2 B=1211
               0  L0 0:60000:20000 20000L/20000P F=1 B=1211
           20000  L0 0:1220000:20000 20000L/20000P F=1 B=1211

                segment [0000000000000000, 0000000001000000) size   16M
```

All blocks were reallocated! The first **L1**, and the two **L0** (data blocks). That´s something a little strange… I was hoping to see all the block pointers reallocated (metadata), and the data block that holds the bytes i have changed. The first data block that holds the first 128K of our file, now is on the first device (**0**), and second block is still on the first device (**0**), but in another location. We can be sure by looking the new offsets, and the new txg creation time (**B=1211**). Let´s see our data again, getting it from the new locations:

```
zdb -R cow:0:60000:20000:r 2> /tmp/file3.dump
zdb -R cow:0:1220000:20000:r 2> /tmp/file4.dump
cat /tmp/file4.dump >> /tmp/file3.dump
diff /tmp/file3.dump THIRDPARTYLICENSEREADME.html
Warning: missing newline at end of file /tmp/file3.dump
5032d5031
<
```

Ok, and the old blocks, they are still there?

```
zdb -R cow:1:40000:20000:r 2> /tmp/file1.dump
zdb -R cow:0:40000:20000:r 2> /tmp/file2.dump
cat /tmp/file2.dump >> /tmp/file1.dump
diff /tmp/file1.dump THIRDPARTYLICENSEREADME.html
Warning: missing newline at end of file /tmp/file1.dump
5027c5027
< END OF TERMS AND CONDITIONS
---
> FIM OF TERMS AND CONDITIONS
```

```
5032d5031
<
```

Really nice! In our test the ZFS copy-on-write moved the whole file from on region on disk to another. But if we were talking about a really big file, let´s say 1GB? Many 128K data blocks, and just a 1K change. ZFS copy-on-write would reallocate all data blocks too? And why ZFS reallocated the "untouched" block in our example (the first data block **L0**)?
Something to look in another time. Stay tuned… ;-)
peace.

EDITED:

The cow is used to guarantee filesystem consistency without "fsck", so without chance of leaving the filesystem in an inconsistent state. For that, ZFS *never* overwrite a block, always that ZFS needs to write something, "data" or "metadata", it writes it to a brand new location. Here in my example, because my test was with a "flat" file (not structured), the VI (and not because it is an ancient program ;), needs to rewrite the whole file to update a little part of it. You are right in the point that has nothing to do with the cow. But i did show that the cow takes place in "data" updates too. And sometimes we forget that flat files are updated like that…
Thanks a lot for your comments!
Leal.

ZFS Internals (part #2)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the **MANUAL** page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

An interesting point about ZFS block pointers, is the Data virtual address (**dva**). In my last post about ZFS internals, i had the following:

```
# zdb -dddddd cow/fs01 4
... snipped ...
Indirect blocks:
               0 L1  0:1205800:400 1:b400:400 4000L/400P F=2 B=1211
               0  L0 0:60000:20000 20000L/20000P F=1 B=1211
           20000  L0 0:1220000:20000 20000L/20000P F=1 B=1211

                segment [0000000000000000, 0000000001000000) size   16M
```

Let´s look the first data block (**L0**):

```
0  L0 0:60000:20000 20000L/20000P F=1 B=1211
```

The dva for that block is the combination of **0** (indicating the physical vdev where that block is), and the **60000**, which means the offset from the begining of the physical vdev (starting after the vdev labels, plus the boot block), 4MB total.

So, let´s try to read our file using the **dva** information:

```
# perl -e "\$x = ((0x400000 + 0x60000) / 512); printf \"\$x\\n\";"
8960
```

ps.: 512 is the disk block size

```
# dd if=/var/fakedevices/disk0 of=/tmp/dump.txt bs=512 \
iseek=8960 count=256
256+0 records in
256+0 records out
# cat /tmp/dump.txt | tail -5
The intent of this document is to state the conditions under which
VIGRA may be copied, such that the author maintains some
semblance of artistic control over the development of the library,
while giving the users of the library the right to use and
distribute VIGRA in a more-or-less customary fashion, plus the
```

That´s cool! Don´t you think?

ZFS Internals (part #3)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INCACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

Hello all…
Ok, we are analyzing the ZFS ondiskformat (like Jack, in parts). In my first post we could see the copy-on-write semantics of ZFS in a VI editing session. It's important to notice a few things about that test:
1) First, we were updating a non-structured flat file (text);
2) Second, all blocks were reallocated (data and metadata);
3) Third, VI (and any software i know), for the sake of consistency, rewrite the whole file when changing something (add/remove/update);
So, that's why the "non touched block" was reallocated in our example, and this have nothing to do with the cow (thanks to Eric to point that). But that is a normal behaviour for softwares updating a flat file. So normal that we just remember that when we make such tests, because in normal situations those "other writes" operations did not have a big impatc. VI actually creates another copy, and at the end move "new" -> "current" (rsync for example, does the same). So, updating a flat file, is the same as creating a new one. That's why i did talk about mailservers that work with mboxes… i don't know about cyrus, but the others that i know, rewrite the whole mailbox for almost every update operation. Maybe a mailserver that creates a mbox like a structured database, with static sized records, could rewrite a line without need to rewrite the whole file, but i don't know any MTA/POP/IMAP like that. See, these tests let us remember why databases exist. ;-)

Well, in this post we will see some internals about ext2/ext3 filesystem to understand ZFS.
Let’s do it…
ps.: I did these test on an Ubuntu desktop

```
# dd if=/dev/zero of=fakefs bs=1024k count=100
# mkfs.ext3 -b 4096 fakefs
mke2fs 1.40.8 (13-Mar-2008)
fakefs is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
25600 inodes, 25600 blocks
1280 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=29360128
1 block group
32768 blocks per group, 32768 fragments per group
25600 inodes per group

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 29 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

Ok filesytem created, and you can see important informations about it, what i think is really nice. I think ZFS could give us some information too. i know, it’s something like “marketing”: We did not pre-allocated nothing… c’mon! ;-)
ps.: Pay attention that we have created the filesystem using 4K blocks, because it is the bigger available.
Now we mount the brand new filesystem, and put a little text file on it.

```
# mount -oloop fakefs mnt/
# debugfs
```

```
debugfs: stats
... snipped ...
Directories:             2
 Group  0: block bitmap at 8, inode bitmap at 9, inode table at 10
           23758 free blocks, 25589 free inodes, 2 used directories
debugfs: quit
# du -sh /etc/bash_completion
216K /etc/bash_completion
# cp -pRf /etc/bash_completion mnt/
```

Just to be sure, let's umount it, and see what we got from the debugfs:

```
# umount mnt/
# debugfs
debugfs: stats
Directories:             2
 Group  0: block bitmap at 8, inode bitmap at 9, inode table at 10
           23704 free blocks, 25588 free inodes, 2 used directories


debugfs: ls
2  (12) .    2  (12) ..    11  (20) lost+found    12  (4052) bash_completion
debugfs:  ncheck 12
Inode   Pathname
12      /bash_completion
```

Very nice! Here we already have a lot of informations:
- The file is there. ;-)
- The filesystem used 54 blocks to hold our file. I know it looking at the **free blocks** line (23758 - 23704 = 54)
- The inode of our file is 12 (don't forget it)
But let's go ahead…

```
debugfs: show_inode_info bash_completion
Inode: 12   Type: regular    Mode:  0644   Flags: 0x0   Generation: 495766590
User:  1000   Group:  1000   Size: 216529
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 432
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x48d4e2da -- Sat Sep 20 08:47:38 2008
```

```
atime: 0x48d4d858 -- Sat Sep 20 08:02:48 2008
mtime: 0x480408b3 -- Mon Apr 14 22:45:23 2008
BLOCKS:
(0-11):1842-1853, (IND):1854, (12-52):1855-1895
TOTAL: 54
```

Oh, that's it! Our file is using 53 data block (53 * 4096 = 217.088), and more one metadata (indirect block 1854). We already have the location too: 12 data blocks from the position 1842-1853, and 41 data blocks from the position 1855 to 1895. Yes, we don't believe it! We need to see it by ourselves…

```
# dd if=Devel/fakefs of=/tmp/out skip=1842 bs=4096 count=12
# dd if=Devel/fakefs of=/tmp/out2 skip=1855 bs=4096 count=41
# cp -pRf /tmp/out /tmp/out3 && cat /tmp/out2 >> /tmp/out3
# diff mnt/bash_completion /tmp/out3
9401a9402
>
\ Não há quebra de linha no final do arquivo
debugfs: quit
```

ps.: just the brazilian portuguese for "Warning: missing newline at end of file" ;-)
Now let's do the same as we did for ZFS…

```
# vi mnt/bash_completion
 change
unset BASH_COMPLETION_ORIGINAL_V_VALUE
 for
RESET BASH_COMPLETION_ORIGINAL_V_VALUE
# umount mnt
# sync
```

The last part, we wil look the new layout of the filesystem…

```
# mount -oloop fakefs mnt
# debugfs
debugfs: show_inode_info bash_completion
Inode: 12   Type: regular    Mode:  0644   Flags: 0x0   Generation: 495766590
User:  1000   Group:  1000   Size: 216529
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 432
Fragment:  Address: 0    Number: 0    Size: 0
```

```
ctime: 0x48d4f6a3 -- Sat Sep 20 10:12:03 2008
atime: 0x48d4f628 -- Sat Sep 20 10:10:00 2008
mtime: 0x48d4f6a3 -- Sat Sep 20 10:12:03 2008
BLOCKS:
(0-11):24638-24649, (IND):24650, (12-52):24651-24691
TOTAL: 54
```

Almost the same as ZFS... the block data are totaly new ones, but the metadata remains in the same inode (12/Generation: 495766590)! Without the copy-on-write, the metadata is rewriten in place, what in a system crash was the reason of the fsck (ext2). But we are working with a ext3 filesystem, so the solution for that possible fail is "journaling". Ext3 has a log (as you can see in the creation messages), where the filesystem controls what it is doing before it really does. That not solves the problem, but makes the recovery easier in the case of a crash.

But what about our data, we need to see our data, show me our........ ;-)
From the new locations:

```
# dd if=Devel/fakefs of=/tmp/out4 skip=24638 bs=4096 count=12
# dd if=Devel/fakefs of=/tmp/out5 skip=24651 bs=4096 count=41
# cp -pRf /tmp/out4 /tmp/out6 && cat /tmp/out5 >> /tmp/out6
# diff mnt/bash_completion /tmp/out6
9402d9401
<
\ Não há quebra de linha no final do arquivo
```

and the originals (old)data blocks...

```
# dd if=Devel/fakefs of=/tmp/out skip=1842 bs=4096 count=12
# dd if=Devel/fakefs of=/tmp/out2 skip=1855 bs=4096 count=41
# cp -pRf /tmp/out /tmp/out3 && cat /tmp/out2 >> /tmp/out3
# diff mnt/bash_completion /tmp/out3
9397c9397
< RESET BASH_COMPLETION_ORIGINAL_V_VALUE
---
> unset BASH_COMPLETION_ORIGINAL_V_VALUE
9401a9402
>
\ Não há quebra de linha no final do arquivo
```

That’s it! Copy-on-write is: “**Never rewrite a live block (data or metadata)**“. Here we can see one of the big diff about ZFS and other filesystems. VI rewrite the whole file (creating a new one), ok both filesystems did allocate other data blocks for it, but the metadata handling was using different approuches (what actualy do the whole difference). But if we change just one block? If our program did not create a new file, just rewrite the block in place? Subject to other post…
peace!

ZFS Internals (part #4)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

In this post we will do something simple, but that shows a great feature of ZFS. More one time we will use ext3 filesystem to understand a ZFS feature…
Here you can see the layout of the ext3 filesystem, and we will use the same file to our today's test:

```
# mount -oloop fakefs mnt
# debugfs
debugfs: show_inode_info bash_completion
Inode: 12   Type: regular    Mode:  0644   Flags: 0x0   Generation: 495766590
User:  1000   Group:  1000   Size: 216529
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 432
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x48d4f6a3 -- Sat Sep 20 10:12:03 2008
atime: 0x48d4f628 -- Sat Sep 20 10:29:36 2008
mtime: 0x48d4f6a3 -- Sat Sep 20 10:12:03 2008
BLOCKS:
(0-11):24638-24649, (IND):24650, (12-52):24651-24691
TOTAL: 54
```

The file we will work on is the only one on that filesystem: **bash_completion**.
So, let's see the head of that file:

```
#   bash_completion - programmable completion functions for bash 3.x
#                     (backwards compatible with bash 2.05b)
#
#   $Id: bash_completion,v 1.872 2006/03/01 16:20:18 ianmacd Exp $
#
#   Copyright (C) Ian Macdonald ian@caliban.org
#
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 2, or (at your option)
```

Ok, that's just ten lines of that file, and we have 4096 bytes of data per block of our filesystem. So, let's umount that filesystem, and read the first block that has the first ten lines (block 24638).

```
# umount fakefs
# fsck.ext3 -v fakefs
e2fsck 1.40.8 (13-Mar-2008)
fakefs: clean, 12/25600 files, 1896/25600 blocks
# dd if=fakefs of=/tmp/out bs=4096 skip=24638 count=1
```

Now, imagine some bad guy acting wild…

```
# vi /tmp/out
```

change

```
#   This program is free software; you can redistribute it and/or modify
```

for

```
#   This program isn't free software; you can't redistribute it ,  modify
```

ps.: That is the problem with flat files, to change it we need worry obout the size of what we are changing.
So, after we did that, we can put the block back to the filesystem.

```
# dd if=fakefs of=/tmp/fake2 skip=24639 bs=4096
# dd if=fakefs of=/tmp/fake1 count=24638 bs=4096
# dd if=/tmp/out of=/tmp/out2 ibs=4096 obs=4095 count=1
# cat /tmp/out2 >> /tmp/fake1
# cat /tmp/fake2 >> /tmp/fake1
# cp -pRf /tmp/fake1 fakefs
# fsck.ext3 -v fakefs
e2fsck 1.40.8 (13-Mar-2008)
fakefs: clean, 12/25600 files, 1896/25600 blocks
# mount -oloop fakefs mnt/
# debugfs
debugfs: open fakefs
debugfs: show_inode_info bash_completion
Inode: 12   Type: regular    Mode:  0644   Flags: 0x0   Generation: 495766590
User:  1000   Group:  1000   Size: 216529
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 432
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x48d4f6a3 -- Sat Sep 20 10:12:03 2008
atime: 0x48d4fac0 -- Sat Sep 20 10:29:36 2008
mtime: 0x48d4f6a3 -- Sat Sep 20 10:12:03 2008
BLOCKS:
(0-11):24638-24649, (IND):24650, (12-52):24651-24691
TOTAL: 54
debugfs: quit
# head mnt/bash_completion

#   bash_completion - programmable completion functions for bash 3.x
#                     (backwards compatible with bash 2.05b)
#
#   $Id: bash_completion,v 1.872 2006/03/01 16:20:18 ianmacd Exp $
#
#   Copyright (C) Ian Macdonald ian@caliban.org
#
#   This program isn't free software; you can't  redistribute it ,  modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 2, or (at your option)

# ls -l mnt/bash_completion
-rw-r--r-- 1 leal leal 216529 2008-09-20 10:12 bash_completion
```

ps.: I did use dd because i think is simple, and i am simulating a HD with a 100mb file. But remember that in a real scenario, that task can be done rewriten just that block, and could be done exactly like this.

Ok, a silent data “corruption”! A really bad one… and the filesystem does not know anything about it. And don’t forget that all the attrs of that file are identi-cals. We can use a “false” file for days, without know… What about ZFS? I tell you: in ZFS that would not gonna happen! Don’t believe me? So you will need to wait for the next part… ;-)
peace.

ZFS Internals (part #5)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

Today, we will see if what we did with the ext3 filesystem can be done with ZFS. We start creating a brand new filesystem, and putting our file into it…

```
# mkfile 100m /var/fakedevices/disk0
# zpool create cow /var/fakedevices/disk0
# zfs create cow/fs01
# cp -pRf /root/bash_completion /cow/fs01/
# ls -i /cow/fs01/
4 bash_completion
```

Ok, now we can start to play..

```
# zdb -dddddd cow/fs01 4
... snipped...
        path    /bash_completion
        atime   Sun Sep 21 12:03:56 2008
        mtime   Sun Sep 21 12:03:56 2008
        ctime   Sun Sep 21 12:10:39 2008
        crtime  Sun Sep 21 12:10:39 2008
        gen     16
        mode    100644
        size    216071
        parent  3
        links   1
        xattr   0
        rdev    0x0000000000000000
Indirect blocks:
               0 L1  0:a000:400 0:120a000:400 4000L/400P F=2 B=16
               0  L0 0:40000:20000 20000L/20000P F=1 B=16
           20000  L0 0:60000:20000 20000L/20000P F=1 B=16

                segment [0000000000000000, 0000000001000000) size   16M
```

So, now we have the **DVA**'s for the two data blocks (**0:40000** and **0:60000**). Let's get our data, umount the filesystem, and try to put the data in place again. For that, we just need the first block...

```
# zdb -R 0:40000:20000:r 2> /tmp/file-part1
# head /tmp/file-part1

#   bash_completion - programmable completion functions for bash 3.x
#                     (backwards compatible with bash 2.05b)
#
#   $Id: bash_completion,v 1.872 2006/03/01 16:20:18 ianmacd Exp $
#
#   Copyright (C) Ian Macdonald
#
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 2, or (at your option)
```

Let's change the file's content, and put it there again. But let's do a change more difficult to catch this time (just one byte).

```
# vi /tmp/file-part1
change
#   bash_completion - programmable completion functions for bash 3.x
for
#   bash_completion - programmable completion functions for bash 1.x

# zpool export cow
# dd if=/var/fakedevices/disk0 of=/tmp/fs01-part1 bs=512 count=8704
# dd if=/var/fakedevices/disk0 of=/tmp/file-part1 bs=512 iseek=8704 count=256
# dd if=/var/fakedevices/disk0 of=/tmp/fs01-part2 bs=512 skip=8960
# dd if=/tmp/file-part1 of=/tmp/payload bs=131072 count=1
# cp -pRf /tmp/fs01-part1 /var/fakedevices/disk0
# cat /tmp/payload >> /var/fakedevices/disk0
# cat /tmp/fs01-part2 >> /var/fakedevices/disk0
# zpool import -d /var/fakedevices/ cow
# zpool status
  pool: cow
 state: ONLINE
 scrub: none requested
config:

        NAME                        STATE     READ WRITE CKSUM
        cow                         ONLINE       0     0     0
          /var/fakedevices//disk0   ONLINE       0     0     0

errors: No known data errors
```

Ok, everything seems to be fine. So, let’s get our data…

```
# head /cow/fs01/bash_completion
#
```

Nothing? But our file is there…

```
# ls -l /cow/fs01
total 517
-rw-r--r--   1 root     root      216071 Sep 21 12:03 bash_completion
# ls -l /root/bash_completion
-rw-r--r--   1 root     root      216071 Sep 21 12:03 /root/bash_completion
```

Let’s see the zpool status command again…

```
# zpool status -v
  pool: cow
 state: ONLINE
status: One or more devices has experienced an error resulting in data
        corruption.  Applications may be affected.
action: Restore the file in question if possible.  Otherwise restore the
        entire pool from backup.
```

```
   see: http://www.sun.com/msg/ZFS-8000-8A
 scrub: none requested
config:

        NAME                         STATE     READ WRITE CKSUM
        cow                          ONLINE       0     0     2
          /var/fakedevices//disk0    ONLINE       0     0     2

errors: 1 data errors, use '-v' for a list
```

Oh, trying to access the file, ZFS could see the checksum error on the block pointer. That's why is important to schedule a scrub, because it will traverse the entire pool looking for errors like that. In this example i did use a pool with just one disk, in a real situation, don't do that! If we had a mirror for example, ZFS would fix the problem using a "good" copy (in this case, if the bad guy did not mess with it too). What zdb can show to us?

```
# zdb -c cow

Traversing all blocks to verify checksums and verify nothing leaked ...
zdb_blkptr_cb: Got error 50 reading <21, 4, 0, 0>  -- skipping

Error counts:

        errno  count
           50  1
leaked space: vdev 0, offset 0x40000, size 131072
block traversal size 339456 != alloc 470528 (leaked 131072)

        bp count:              53
        bp logical:        655360        avg:  12365
        bp physical:       207360        avg:   3912     compression:   3.16
        bp allocated:      339456        avg:   6404     compression:   1.93
        SPA allocated:     470528       used:  0.47%
```

Ok, we have another copy (from a trusted media ;)…

```
# cp -pRf /root/bash_completion /cow/fs01/bash_completion
# head /cow/fs01/bash_completion

#   bash_completion - programmable completion functions for bash 3.x
#                     (backwards compatible with bash 2.05b)
#
#   $Id: bash_completion,v 1.872 2006/03/01 16:20:18 ianmacd Exp $
#
#   Copyright (C) Ian Macdonald
#
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 2, or (at your option)
```

Now everything is in a good shape again…
see ya.

ZFS Internals (part #6)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

In this post let's see some important feature of the **ZFS filesystem**, that i think is not well understood and i will try to show some aspects, and my understanding about this technology. I'm talking about the **ZFS Intent Log**.
Actually, the first distinction that the devel team wants us to know, is about the two pieces:
- **ZIL**: the code..
- **ZFS Intent log** (this can be in a separated device, thus slog)
The second point i think is important to note, is the purpose of such ZFS' feature. All filesystems that i know that use some logging technology, use it to maintain filesystem consistency (making the fsck job faster). ZFS does not…
ZFS is always consistent on disk (transactional all-or-nothing model), and so there is no need for fsck, and if you find some inconsistency (i did find one), it's a bug. ;-) The copy-on-write semantics was meant to guarantee the always consistent state of ZFS. I did talk about it on my earlier posts about ZFS Internals, even the name of the zpools used in the examples was **cow**. ;-)
So, why ZFS has a log feature (or a slog device)? **Performance is the answer!**
A primary goal of the development of ZFS was consistency, and honor all the filesystems operations (what seems to be trivial), but some filesystems (even disks) do not.

So, For ZFS a sync request is a sync done.
For ZFS, a perfect world would be:

```
 +------------------+                +--------------------------------+
 |  write requests  | ---------->    | ZFS writes to the RAM's buffer |
 +------------------+                +--------------------------------+
```

So, when the RAM’s buffer is full:

```
+---------------------------+                +------------------------+
 | ZFS RAM's buffer is full | ---------->    | ZFS writes to the pool |
+---------------------------+                +------------------------+
```

For understanding, imagine you are moving your residence, from one city to another. You have the TV, bed, computer desk, etc, as the “write requests”… and the truck as RAM’s buffer, and the other house as the disks (ZFS pool). The better approuch is to have sufficient RAM (a big truck), put all stuff into it, and transfer the whole thing at once.
But you know, the world is not perfect…
That writes that can wait the others (all stuff will go to the new house together), are asynchronous writes. But there are some that can not wait, and need to be written as fast as possible. That ones are the synchronous writes, the filesystems nightmare, and is when the ZFS intent log comes in.
If we continue with our example, we would have two (or more trucks), one would go to the new city full, and others with just one request (a TV for example). Imagine the situation: bed, computer, etc, put in one truck, but the TV needs to go now! Ok, so send a special truck just for it… Before the TV truck returns, we have another TV that can not wait too (another truck).
So, the ZFS intent log comes in to address that problem. *All* synchronous requests are written to the ZFS intent log instead to write to the disks (main pool).
ps.: Actually not *all* sync requests are written to ZFS intent log. I will talk about that in another post…

The sequence is something like:

```
 +----------------+          +--------------------------------+
 | write requests | ---->    | ZFS writes to the RAM's buffer |-------+
 +----------------+          +--------------------------------+       |
                                                                      |
   ZFS acks the clients   +------------------------------------+      |
   <--------------------- |syncs are written to ZFS intent log |<-----+
                          +------------------------------------+
```

So, when the RAM’s buffer is full:

```
+----------------------------+                +-------------------------+
| ZFS RAM's buffer is full   | -------------->| ZFS writes to the pool |
+----------------------------+                +-------------------------+
```

See, we can make the world a better place. ;-)
If you pay attention, you will see that the flow is just one:

```
 requests ------> RAM --------->disks
```

or

```
 requests ------> RAM--------->ZFS intent log
                           |
                           +---->disks
```

The conclusion is: The intent log is never read (while the system is runnning), just written. I think that is something not very clear: the intent log needs to be read after a crash, just that. Because if the system crash and there are some data that was written to the ZFS intent log, and not written to the main pool, the system needs to read the ZFS intent log and replay any transactions that are not on the main pool.

From the source code:

```
/*
 * The zfs intent log (ZIL) saves transaction records of system calls
 * that change the file system in memory with enough information
 * to be able to replay them. These are stored in memory until
 * either the DMU transaction group (txg) commits them to the stable pool
 * and they can be discarded, or they are flushed to the stable log
 * (also in the pool) due to a fsync, O_DSYNC or other synchronous
 * requirement. In the event of a panic or power fail then those log
 * records (transactions) are replayed.
 *
...
```

ps.: More one place where ZFS intent log is called ZIL. ;-)

Some days ago looking at the SUN's fishworks solution, i did see that there is no redundancy on the fishworks slog devices. Just one, or stripe… so i did realize that the "window" of problem is around five seconds! i mean, the system did write the data to the ZFS intent log device (slog), and just after that the slog device did fail, and before the system write the data on the main pool (what is around five seconds), the system crash.

Ok, but before you start to use the SUN's approuch, i think there is at least one bug that need to be addressed:

6733267 "Allow a pool to be imported with a missing slog"

But if you do not export the pool, i think you are in good shape…

In another post of ZFS internals series, i will try to go deeper into ZFS intent log, and see some aspects that i did not touch here. Obviously, with some action… ;-)

But now i need to sleep!

peace.

ZFS Internals (part #7)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

Ok, we did talk about **dva's** on the first and second posts, and actually that is the main concept behind the block pointers and indirect blocks in ZFS. So, the **dva** (data virtual address), is a combination of an **physical vdev** and an **offset**. By default, ZFS stores one DVA for user data, two DVAs for filesystem metadata, and three DVAs for metadata that's global across all filesystems in the storage pool, so any block pointer in a ZFS filesystem has at least two copies. If you look at the link above (the great post about Ditto Blocks), you will see that the ZFS engineers were thinking about add the same feature for data blocks, and here you can see that this is already there.

```
# ls -li /cow/fs01/
4 file.txt
# md5sum /cow/fs01/file.txt
6bf2b8ab46eaa33b50b7b3719ff81c8c  /cow/fs01/file.txt
```

So, let's smash some ditto blocks, and see some ZFS action…

```
# zdb -dddddd cow/fs01 4
... snipped ...
Indirect blocks:
               0 L1  1:11000:400 0:11000:400 4000L/400P F=2 B=8
               0  L0 1:20000:20000 20000L/20000P F=1 B=8
           20000  L0 1:40000:20000 20000L/20000P F=1 B=8

                segment [0000000000000000, 0000000000040000) size  256K
```

You can see above (**bold**) the two **dva's** for the block pointer **L1**(level 1): **0:11000** and **1:11000**.

```
# zpool status cow
  pool: cow
 state: ONLINE
 scrub: none requested
config:

        NAME                          STATE     READ WRITE CKSUM
        cow                           ONLINE       0     0     0
          /var/fakedevices/disk0      ONLINE       0     0     0
          /var/fakedevices/disk1      ONLINE       0     0     0

errors: No known data errors
```

Ok, now we need to export the filesystem, and simulate a silent data corruption on the first vdev that clears the first block pointer (**1K**). We will write **1K of zeros** where should be a bp copy.

```
# zfs export cow
# perl -e "\$x = ((0x400000 + 0x11000) / 512); printf \"\$x\\n\";"
8328
# dd if=/var/fakedevices/disk0 of=/tmp/fs01-part1 bs=512 count=8328
# dd if=/var/fakedevices/disk0 of=/tmp/firstbp bs=512 iseek=8328 count=2
# dd if=/var/fakedevices/disk0 of=/tmp/fs01-part2 bs=512 skip=8330
# dd if=/dev/zero of=/tmp/payload bs=1024 count=1
# cp -pRf /tmp/fs01-part1 /var/fakedevices/disk0
# cat /tmp/payload >> /var/fakedevices/disk0
# cat /tmp/fs01-part2 >> /var/fakedevices/disk0
```

That's it, our two disks are there, whith the first one corrupted.

```
# zpool import -d /var/fakedevices/ cow
# zpool status cow
  pool: cow
 state: ONLINE
 scrub: none requested
config:
        NAME                          STATE     READ WRITE CKSUM
```

```
      cow                         ONLINE       0     0     0
        /var/fakedevices/disk0  ONLINE       0     0     0
        /var/fakedevices/disk1  ONLINE       0     0     0

errors: No known data errors
```

Well, as always the import procedure was fine, and the pool seems to be perfect. Let’s see the md5 of our file:

```
# md5sum /cow/fs01/file.txt
6bf2b8ab46eaa33b50b7b3719ff81c8c  /cow/fs01/file.txt
```

Good too! ZFS does not know about the corrupted pointer? Let’s try it again...

```
# zpool status
  pool: cow
 state: ONLINE
status: One or more devices has experienced an unrecoverable error.  An
      attempt was made to correct the error.  Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
      using 'zpool clear' or replace the device with 'zpool replace'.
   see: http://www.sun.com/msg/ZFS-8000-9P
 scrub: none requested
config:

      NAME                        STATE     READ WRITE CKSUM
      cow                         ONLINE       0     0     0
        /var/fakedevices/disk0  ONLINE       0     0     1
        /var/fakedevices/disk1  ONLINE       0     0     0
```

As always, we need to try to access the data to ZFS identify the error. So, following the ZFS concept, and the message above, the first dva must be good as before at this point. So, let’s look..

```
# zpool export cow
# dd if=/var/fakedevices/disk0 of=/tmp/firstbp bs=512 iseek=8328 count=2
# dd if=/var/fakedevices/disk1 of=/tmp/secondbp bs=512 iseek=8328 count=2
# md5sum /tmp/*bp
70f12e12d451ba5d32b563d4a05915e1  /tmp/firstbp
70f12e12d451ba5d32b563d4a05915e1  /tmp/secondbp
```

;-)
But something is confused about the ZFS behaviour, because i could see a better information about what was fixed if i first execute a scrub. Look (executing a scrub just after import the pool):

```
# zpool scrub cow
# zpool status cow
  pool: cow
 state: ONLINE
status: One or more devices has experienced an unrecoverable error.  An
      attempt was made to correct the error.  Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
      using 'zpool clear' or replace the device with 'zpool replace'.
   see: http://www.sun.com/msg/ZFS-8000-9P
 scrub: scrub completed after 0h0m with 0 errors on Sun Dec 28 17:52:59 2008
config:

      NAME                          STATE     READ WRITE CKSUM
      cow                           ONLINE       0     0     0
        /var/fakedevices/disk0      ONLINE       0     0     1  1K repaired
        /var/fakedevices/disk1      ONLINE       0     0     0

errors: No known data errors
```

Excellent! **1K** was the bad data we have injected on the pool (and the whole diff between one and another output)! But the automatic repair of the ZFS did not show us that, and does not matter if we execute a scrub after it. I think because the information was not gathered in the automatic resilver, what the scrub process does. if there is just one code to fix a bad block, both scenarios should give the same information right? So, being very simplistic (and doing a lot of specutlation, without look the code), seems like two procedures, or a little bug…

Well, i will try to see that on the code, or use dtrace to see the stack trace for one and other process when i got some time. But i'm sure the readers know the answer, and i will not need to…
;-)
peace

ZFS Internals (part #8)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

That's is something trick, and i think is important to write about… i'm talking about ZFS vdevs.
ZFS has two types of vdevs: logical and physical. So, from the ZFS on-disk specification, we know that a physical vdev is a writeable media device, and a logical vdev is a grouping of physical vdevs.
Let's see a simple diagram using a RAIDZ logical vdev, and five physical vdevs:

```
                      +----------------------+
                      |      Root vdev       |
                      +----------------------+
                                 |
                         +--------------+
                         |    RAIDZ     |            VIRTUAL VDEV
                         +--------------+
                                |
                           +----------+
                           |  128KB   |              CHECKSUM
                           +----------+
                                |
    32KB         32KB         32KB          32KB          Parity
 .-------.     .-------.     .-------.    .-------.      .-------.
|-______-|    |-______-|    |-______-|   |-______-|     |-______-|
|  vdev1 |    | Vdev2  |    | Vdev3  |   | Vdev4  |     | Vdev5  | PHYSICAL VDEV
 '-____-'      '-____-'      '-____-'     '-____-'       '-_____-'
```

The diagram above was just an example, and in that example the data that we are handling in the RAIDZ virtual vdev is a block of 128KB. That was just to make my math easy, so i could divide equal to all phisycal vdevs. ;-)

But remember that with RAIDZ we have always a full stripe, not matter the size of the data.
The important part here is the filesystem block. When i did see the first video presentation about ZFS,

i had the wrong perception about the diagram above. As we can see, if the system reclaims a block, let's say the 128KB block above, and the **physical vdev 1** gives the wrong data, ZFS just fix the data on that physical vdev, right? Wrong... and that was my wrong perception. ;-)
ZFS RAIDZ virtual vdev does not know which physical vdev (disk) gave the wrong data. And here i think there is a great level of abstraction that shows the beauty about ZFS... because the filesystems are there (on the physical vdevs), but there is not an explict relation! A filesystem block has nothing to do with a disk block. So, the checksum of the data block is not at the physical vdev level, and so ZFS cannot know directly what disk gave the wrong data without a "combinatorial reconstruction" to identify the culprit. From the vdev_raidz.c:

```
static void
vdev_raidz_io_done(zio_t *zio)
{
...
        /*
         * If the number of errors we saw was correctable -- less than or equal
         * to the number of parity disks read -- attempt to produce data that
         * has a valid checksum. Naturally, this case applies in the absence of
         * any errors.
         */
...
```

That gives a good understanding of the design of ZFS. I really like that way of solving problems, and to have specialized parts like this one. Somebody can think that this behaviour is not optimum. But remember that this is something that should not happen all the time.

In mirror we have a whole different situation, because all the data is on any device, and so ZFS can match the checksum, and read the other vdevs looking for the right answer. Remember that we can have n-way mirror…

In the source we can see that a normal read is done in any device:

```
static int
vdev_mirror_io_start(zio_t *zio)
{

...
          /*
           * For normal reads just pick one child.
           */
          c = vdev_mirror_child_select(zio);
          children = (c >= 0);
...
```

So, ZFS knows if this data is OK or not, and if it is not, it can fix it. But without to know which disk but which physical vdev. ;-) The procedure is the same without the combinatorial reconstruction. And as a final note, the resilver of a block is not copy on write, so in the code we have a comment about it:

```
                /*
                 * Don't rewrite known good children.
                 * Not only is it unnecessary, it could
                 * actually be harmful: if the system lost
                 * power while rewriting the only good copy,
                 * there would be no good copies left!
                 */
```

So the physical vdev that has a good copy is not touched. As we need to see to believe...

```
mkfile 100m /var/fakedevices/disk1
mkfile 100m /var/fakedevices/disk2
zpool create cow mirror /var/fakedevices/disk1 /var/fakedevices/disk2
zfs create cow/fs01
cp -pRf /etc/mail/sendmail.cf /cow/fs01/
ls -i /cow/fs01/
 4 sendmail.cf
zdb -dddddd cow/fs01 4
Dataset cow/fs01 [ZPL], ID 30, cr_txg 15, 58.5K, 5 objects, rootbp [L0 DMU ob-
jset]  \\
400L/200P DVA[0]=<0:21200:200> DVA[1]=<0:1218c00:200> fletcher4 lzjb LE conti-
guous \\
birth=84 fill=5 cksum=99a40530b:410673cd31e:df83eb73e794:207fa6d2b71da7

    Object  lvl   iblk   dblk  lsize  asize  type
         4    1    16K  39.5K  39.5K  39.5K  ZFS plain file (K=inherit) (Z=inh-
erit)
                                 264  bonus  ZFS znode
        path    /sendmail.cf
        uid     0
        gid     2
        atime   Mon Jul 13 19:01:42 2009
        mtime   Wed Nov 19 22:35:39 2008
        ctime   Mon Jul 13 18:30:19 2009
        crtime  Mon Jul 13 18:30:19 2009
        gen     17
        mode    100444
        size    40127
        parent  3
        links   1
        xattr   0
        rdev    0x0000000000000000
Indirect blocks:
               0 L0 0:11200:9e00 9e00L/9e00P F=1 B=17

                segment [0000000000000000, 0000000000009e00) size 39.5K
```

 So, we have a mirror of two disk, and a little file on it... let's do a little math, and
smash the data block from the first disk:

```
zpool export cow
perl -e "\$x = ((0x400000 + 0x11200) / 512); printf \"\$x\\n\";"
dd if=/tmp/garbage.txt of=/var/disk1 bs=512 seek=8329 count=79 conv="nocreat,n-
otrunc"
zpool import -d /var/fakedevices/ cow
cat /cow/fs01/sendmail.cf > /dev/null
zpool status cow
  pool: cow
 state: ONLINE
status: One or more devices has experienced an unrecoverable error.  An
        attempt was made to correct the error.  Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
        using 'zpool clear' or replace the device with 'zpool replace'.
   see: http://www.sun.com/msg/ZFS-8000-9P
 scrub: none requested
config:

        NAME            STATE     READ WRITE CKSUM
        cow             ONLINE       0     0     0
          mirror        ONLINE       0     0     0
            /var/disk1  ONLINE       0     0     1
            /var/disk2  ONLINE       0     0     0

errors: No known data errors
```

Now let's export the pool and read our data from the same offset in both disks:

```
dd if=/var/fakedevices/disk1 of=/tmp/dump.txt bs=512 skip=8329 count=79
dd if=/var/fakedevices/disk2 of=/tmp/dump2.txt bs=512 skip=8329 count=79
diff /tmp/dump.txt /tmp/dump2.txt
head /tmp/dump.txt
#
# Copyright (c) 1998-2004 Sendmail, Inc. and its suppliers.
#       All rights reserved.
# Copyright (c) 1983, 1995 Eric P. Allman.  All rights reserved.
# Copyright (c) 1988, 1993
#       The Regents of the University of California.  All rights reserved.
#
# Copyright 1993, 1997-2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

head /etc/mail/sendmail.cf
#
# Copyright (c) 1998-2004 Sendmail, Inc. and its suppliers.
#       All rights reserved.
# Copyright (c) 1983, 1995 Eric P. Allman.  All rights reserved.
# Copyright (c) 1988, 1993
#       The Regents of the University of California.  All rights reserved.
#
# Copyright 1993, 1997-2006 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
```

So, never burn your physical vdevs, because you can (almost) always get some files from it.
Even if the ZFS can't. ;-)
peace

ZFS Internals (part #9)

PLEASE BE AWARE THAT ANY INFORMATION YOU MAY FIND HERE MAY BE INACCURATE, AND COULD INCLUDE TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS, AND EVEN SPELLING ERRORS.

```
From the MANUAL page:
The zdb command is used by  support  engineers  to  diagnose
failures and gather statistics. Since the ZFS file system is
always consistent on disk and is self-repairing, zdb  should
only be run under the direction by a support engineer.
```

DO NOT TRY IT IN PRODUCTION. USE AT YOUR OWN RISK!

Some builds ago there was a great change in OpenSolaris regarding ZFS. It was not a change in the ZFS itself because the change was the adition of a new scheduling class (but with great impact on ZFS).

OpenSolaris had six scheduling classes until then:
- Timeshare (**TS**);
This is the classical. Each process (thread) has an amount of time to use the processor resources, and that "amount of time" is based on priorities. This scheduling class works changing the process priority.
- Interactive (**IA**);
This is something interesting in OpenSolaris, because it is designed to give a better response time to the desktop user. Because the windown that is active has a priority boosts from the OS.
- Fair Share (**FSS**);
Here there is a division of the processor (fair? ;-) in units, so the administrator can allocate the processor resourcers in a controlled way. I have a screencast series about Solaris/OpenSolaris features so you can see a live demonstration about resource management and this FSS scheduling class. Take a look at the containers series...
- Fixed Priority (**FX**);
As the name suggests, the OS does not change the priority of the thread, so the time quantum of the thread is always the same.
- Real Time (**RT**);

This is intended to guarantee a good response time (latency). So, is like a special queue on the bank (if you have special necessities, a pregnant lady, elder, or have many, many dollars). Actually this kind of person do not go to bank.. hmmm bad example, anyway…
– System (**SYS**);
For the bank owner. ;-)
Hilarious, because here was the problem with ZFS! Actually, the SYS was not prepared for ZFS's transaction group sync processing.

There were many problems with the behaviour of ZFS IO/Scheduler:

6471212 need reserved I/O scheduler slots to improve I/O latency of critical ops
6494473 ZFS needs a way to slow down resilvering
6721168 slog latency impacted by I/O scheduler during spa_sync
6586537 async zio taskqs can block out userland commands
ps.: I would add to this list the scrub process too…

The solution on the new scheduling class (#7) is called:

System Duty Cycle Scheduling Class (**SDC**);

The first thing that i did think reading the theory statement from the project was not so clear why fix a IO problem changing the scheduler class, actualy messing with the management of the processor resources. Well, that's why i'm not a kernel engineer… thinking well, seems like a clever solution, and given the complexity of ZFS, the easy way to control it.
As you know, ZFS has IO priorities and deadlines, synchronous IO (like sync/writes and reads) have the same priority. My first idea was to have separated slots for different type of operation. It's interesting because this problem was subject of a post from Bill Moore about how ZFS was handling a massive write keeping up the reads.
So, there were some discussions about why create another scheduling class and not just use the SYS class. And the answer was that the sys class was not designed to run kernel threads that are large consumers of CPU time. And by definition, SYS class threads run without preemption from anything other than real-time and interrupt threads.
And more:

Each (**ZFS**) sync operation generates a large batch of I/Os, and each I/O may need to be compressed and/or checksummed before it is written to storage. The taskq threads which perform the compression and checksums will run nonstop as long as they have work to do; a large sync operation on a compression-heavy dataset can keep them busy for seconds on end.

ps.: And we were not talking about dedup yet, seems like a must fix for the evolution of ZFS…
You see how our work is wonderful, by definition NAS servers have no CPU bottleneck, and that is why ZFS has all these wonderful features, and new features like dedup are coming. But the fact that CPU is not a problem, actually was the problem. ;-) It's like give to me the MP4-25 from Lewis Hamilton. ;-)))
There is another important point with this PSARC integration, because now we can observe the ZFS IO processing because was introduced a new system process with the name: zpool-poolname, which gives us observability using simple commands like **ps** and **prstat**. Cool!
I confess i did not had the time to do a good test with this new scheduling class implementation, and how it will perform. This fix was commited on the build 129, and so should be on the production ready release from OpenSolaris OS (2010.03). Would be nice to hear the comments from people that is already using this new OpenSolaris implementation, and how the dedup is performing with this.
peace