

ZFS On-Disk Data Walk (or: Where's my Data?)

DRAFT

Copyright 2008, Max Bruning

Abstract

This paper will examine the on-disk format of ZFS by using modified versions of mdb and zdb to actually look at the data structures of a ZFS file system on disk. The paper will cover the steps taken to convert a pathname in a ZFS file system to get to the actual data of the file at the end of the pathname. At each step, the data structure(s) being examined will be described. You may try to follow along by using a ZFS file system on your machine, but it is recommended that the file system not be very large. For the example that is covered here, the total amount of space taken by the file system is 3.1MB out of a single disk volume that is 37GB large.

The techniques shown should work for larger file systems, and should also work for file systems that span multiple disks in any configuration. The technique has been tried on a much larger file system, and on a 2-way mirror, and works. However, there may be more steps that need to be taken, particularly with respect to indirection. To get the most from this paper, it is recommended you have a copy of the "ZFS On_Disk Specification", available at: <http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf>. and the source code, available at:

The paper is divided into the following sections:

- Overview of On-Disk Data Structures
- On-Disk data structure walk.
- Future Work.
- Conclusion.

Overview of On-Disk Data Structures

The following is a list of most of the data structures found on the disk(s) making up a ZFS pool. This

is not a complete list. Rather, we concentrate on the structures that we'll be using in the "On-Disk data structure walk" portion of this paper. Header files for these structures can be found at `uts/common/fs/zfs/sys/*.h`. For each structure, we'll list the header file where the structure can be found. There are additional structures used to maintain information about a mounted ZFS file system in memory, but these structures will have to be the subject of another paper.

For a more complete description of these structures, please see the "ZFS On-Disk Specification" paper available at <http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf>.

- `uberblock_t` - This is the starting point for locating any structure within a ZFS file system. These are organized in an array, starting at physical location 0x20000 bytes within a vdev label. There are 4 copies of the vdev label on a given physical disk in use with ZFS. These are at the beginning of the disk at physical location 0x0 and at 0x40000 (256k), and at the end of the disk, N-512k and N-256K (where "N" is the size of the disk). Each `uberblock_t` in the 128k array of uberblocks is 1k large (making 128 `uberblock_t` in the array. At any point in time, only 1 `uberblock_t` in the array is "active". The `uberblock_t` with the highest transaction group id and valid checksum is the active uberblock. You can use "zdb -uuu zpool_name" to display the active `uberblock_t` for a "zpool_name". The `uberblock_t` is defined in `uts/common/fs/zfs/sys/uberblock_impl.h`.
- `blkptr_t` - Structure used to locate, describe, and verify blocks on disk. These are embedded within other structures (most commonly, `dnode_phys_t` structs), and also exist in arrays as "indirect" blocks. They are defined in `uts/common/fs/zfs/sys/spa.h`.
- `dnode_phys_t` - Almost everything in ZFS exists as an object. The `dnode_phys_t` is a 512-byte structure that describes an object. It is defined in `uts/common/fs/zfs/sys/dnode.h`. The `dnode_phys_t` contains a `blkptr_t` and may also contain data in a "bonus buffer" (`dn_bonusbuffer`). The types of objects a `dnode_phys_t` describes are defined in `uts/common/fs/zfs/sys/dmu.h`, and the type of object for a given `dnode_phys_t` is in the `dn_type` structure member.
- `objset_phys_t` - The `objset_phys_t` describes a group of objects. There will be one (at least) of these for the set of all objects (the "meta object set", or MOS), and one for ZFS file system objects (files and directories). This is defined in `usr/common/fs/zfs/sys/dmu_objset.h`.
- ZAP Objects - These are block(s) containing name/value pair attributes. (ZAP is "ZFS Attribute Processor"). There are two basic types of ZAP objects, "microzap" and "fatzap". Microzap objects are used when the attributes can fit in one block (there are other restrictions, not important here). Fatzap objects allow for more attributes than fit in one block. The first 64 bits of a ZAP object define whether it is a microzap object or a fatzap object. Microzap objects use the `mzap_phys_t` type defined in `uts/common/fs/zfs/sys/zap_impl.h`. The last field in the `mzap_phys_t` is an `mzap_ent_phys_t`. There may be more than one `mzap_ent_phys_t` in a microzap object. The number of `mzap_ent_phys_t` depends on the size of the block where the microzap resides. Not all entries need be in use. A fatzap object is made up of two structures. The starting point is a `zap_phys_t`. This contains a pointer table of blocks containing `zap_leaf_chunk_t` structs. We won't be looking at fatzaps in this paper. Take a look at `uts/common/fs/zfs/sys/zap_leaf.h` for more information.

The following objects exist in the bonus buffer of some `dnode_phys_t` structures.

- `dsl_dir_phys_t` - This is in the bonus buffer of a `DMU_OT_DSL_DIR` (DSL Directory) `dnode_phys_t` for the MOS. It contains the object id of a `DMU_OT_DSL_DATASET`

`dnode_phys_t`. See `uts/common/fs/zfs/sys/dsl_dir.h`.

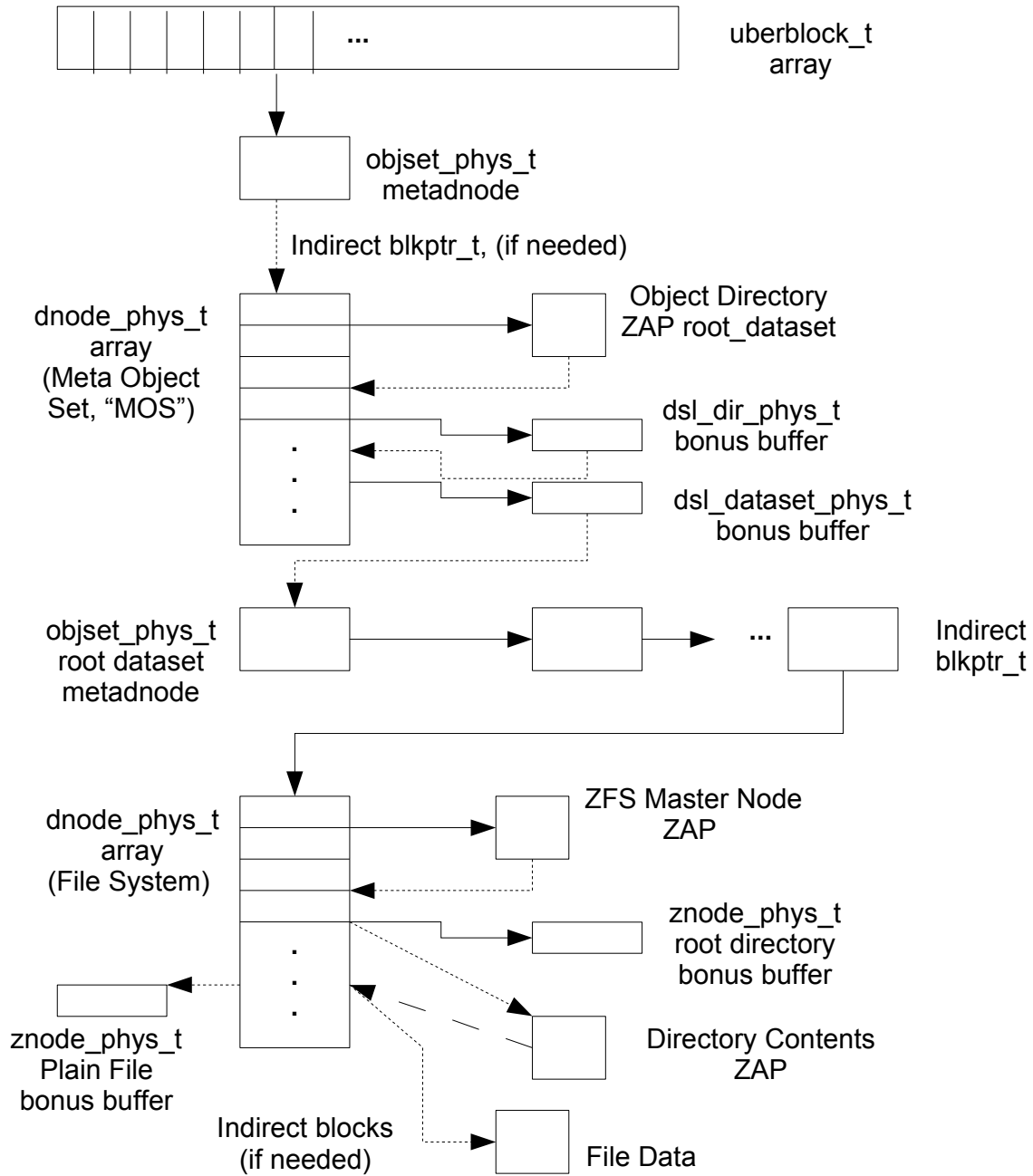
- `dsl_dataset_phys_t` - This is in the bonus buffer of a `DMU_OT_DSL_DATASET` `dnode_phys_t` in the MOS. This contains a `blkptr_t` which (indirectly) leads to a second array of `dnode_phys_t` for objects within a ZFS file system.
- `znode_phys_t` - This is in the bonus buffer of `dnode_phys_t` structures for plain files and directories. It contains attributes of the file/directory (time stamps, ownership, size, etc.). This structure is the closest equivalent to the disk inode for UFS file system files/directories.

On-Disk Data Structure Walk

In this section of the paper, the data for a file is found on a ZFS file system. To do this, we start at the `uberblock_t` and follow data structures on disk until we get to the data. If you want to "follow along" on your machine, you'll need a version of `mdb` that allows you to use the `:::print` dcmd to print data structures from a raw file, a version of `zdb` that allows you to decompress blocks before dumping them out, and an `mdb` dmod that has the `:::blkptr` command. These are available here??? Note: the ZFS file system I am using for this does not span multiple disks. If your file system spans multiple disks, the technique should still work, but there may be more indirect blocks in use, and the `vdev` used with the `zdb` command will probably change for different `blkptr_t` structs. In other words, pay attention to the output you get from each step before proceeding to the next step.

At a high level, the following steps are taken:

1. Copy a file with known contents into the top directory in a mounted ZFS file system. This step isn't completely necessary. It is done here so that when we get to the data, we know we have the correct data.
2. Display the active `uberblock_t`, along with its `blkptr_t`.
3. Display the `objset_phys_t` for the MOS.
4. Display the Object Directory `dnode_phys_t`, and its ZAP object for the MOS.
5. Display the DSL Directory `dnode_phys_t` and its bonus buffer for the MOS.
6. Display the DSL Dataset `dnode_phys_t` and its bonus buffer for the MOS.
7. From the DSL Dataset `dnode`, display the `objset_phys_t` for the ZFS file system.
8. Using the `objset_phys_t` for the ZFS file system, display the Master Node `dnode_phys_t`, and its ZAP object. This will almost certainly involve walking through indirect blocks.
9. From the Master Node ZAP object, display the root directory `dnode_phys_t` for the ZFS file system.
10. From the `blkptr_t` in the root directory `dnode_phys_t`, find the object id of the file we want to display. (Note, the copy of the file in step 1 placed the file in the root directory of the file system). If you want to find a file in a subdirectory, you will need to take more steps.
11. From the `dnode_phys_t` of the file we want to examine, display the data via the `blkptr_t` structure(s) for the file. The following diagram shows the path we are going to take. You might find it useful to refer to this as we go through the steps.



Note: Bonus buffers are fields within `dnode_phys_t`, all other objects occupy separate blocks on disk. Blocks are variable size, and not shown to scale.

ZFS On-Disk Data Structure Organization

First, we'll copy a file with known contents into the top directory of a zfs file system.

```
# cp /usr/dict/words /lacedisk/words
#
```

Next, we'll use zdb to read the current uberblock for the pool.

```
# zdb -uuu lacedisk
Uberblock

    magic = 0000000000bab10c
    version = 10
    txg = 19148
    guid_sum = 17219723339164464949
    timestamp = 1203801884 UTC = Sat Feb 23 22:24:44 2008
    rootbp = [L0 DMU objset] 400L/200P DVA[0]=<0:4e00:200>
    DVA[1]=<0:1c0004e00:200> DVA[2]=<0:380001200:200> fletcher4 lzjb LE contiguous
    birth=19148 fill=18 cksum=89aca5d29:38d271ef883:be5570b26779:1af282de579a51
#
```

The above output shows that the `objset_phys_t` contains a `blkptr_t` with 3 copies of the DVA (Data Virtual Address), one at disk block 0x4e00, the second at 0x1c0004e00, and the third at 0x380001200. The three copies of the same data are called “ditto blocks”. Ditto blocks are used for most of the metadata in a zfs file system, and provide a redundancy mechanism in case of failures. Each of these is 0x200 bytes on the disk, and is on vdev 0 (the only disk in the pool). The `objset_phys_t` is compressed with lzjb compression, and after compression it is 0x400 bytes (400L in the above output). We'll need to find which device(s) make up the pool. We'll use `zpool(1)` for this.

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

NAME	STATE	READ	WRITE	CKSUM
lacedisk	ONLINE	0	0	0
c4t0d0p1	ONLINE	0	0	0

```
errors: No known data errors
#
```

Now we'll use zdb to dump the raw data referred to in the uberblock. This is an `objset_phys_t`. The syntax of the “-R” option to read zfs blocks is:

```
# zdb -R pool:vdev_specifier:offset:size:flags
```

where:

- “pool” is the zfs pool.
- “vdev_specifier” is either device name or id. In this case, c4t0d0p1 is specified. Alternatively, one can use “0” for a zfs file system on a single device. In the case of a 2-way mirror, you can use the device name as here, or “0.0” to specify the first device, and “0.1” to specify the second device. See the comment at the beginning of the `zdb_vdev_lookup()` function in `usr/src/cmd/zdb/zdb.c` for naming conventions.
- “offset” is the byte offset following the disk label (L1 and L2). The L1 and L2 labels (along with boot block info) takes up 4MB (0x400000). So the physical address on the disk is “offset + 0x400000”.
- “size” is the physical size in bytes on the disk.
- “flags” are a set of characters specifying options. The “d” flag used here is not supported in the current version of zdb. The version of zdb being used here has been extended to support the “d” option. With the “d” option, one specifies the compression algorithm and decompressed size.

For more information on the “-R” command, see the beginning of the `zdb_read_block()` routine in the zdb source code (`usr/src/uts/cmd/zdb/zdb.c`). Note that the `zdb(1M)` manual page states that the Interface Stability is “Unstable”, indicating that the syntax of the `zdb(1M)` command is subject to change.

The command below reads a 0x200 bytes from the `lacedisk` zpool and the `/dev/dsk/c4t0d0p1` device in the pool. It reads block number 0x4e00 and performs `lzjb` decompression. The size after decompression is 0x400 bytes. The output is placed in `/tmp/metadnode`.

```
# zdb -R lacedisk:c4t0d0p1:4e00:200:d,lzjb,400 2> /tmp/metadnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

`/tmp/metadnode` contains the `objset_phys_t` data. (The beginning of the `objset_phys_t` is a `dnode_phys_t` which describes the meta object set, i.e., the “metadnode”). We use (a modified) `mdb` to print the structure contents. With the modified `mdb`, the current kernel CTF information is loaded (`::loadctf`), and a `dmod` is loaded. The kernel CTF information allows one to `::print` data structures.

```
# mdb /tmp/metadnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::print -a -t zfs`objset_phys_t
{
  0 dnode_phys_t os_meta_dnode = {
    0 uint8_t dn_type = 0xa    <-- DMU_OT_DNODE (see uts/common/fs/zfs/sys/dmu.h)
    1 uint8_t dn_indblkshift = 0xe
    2 uint8_t dn_nlevels = 0x1  <-- no indirect blocks
    3 uint8_t dn_nblkptr = 0x3  <-- 3 copies in the blkptr_t
    4 uint8_t dn_bonustype = 0   <-- no data in bonus buffer
... <--output omitted
    40 blkptr_t [1] dn_blkptr = [ <-- blkptr is at address 0x40 in tmp file
      {
        40 dva_t [3] blk_dva = [
          {
```

```

    40 uint64_t [2] dva_word = [ 0x4, 0x28 ]
    }
    {
    50 uint64_t [2] dva_word = [ 0x4, 0xe00028 ]
    }
    {
    60 uint64_t [2] dva_word = [ 0x4, 0x1c0001c ]
    }
]
70 uint64_t blk_prop = 0x800a07030003001f
78 uint64_t [3] blk_pad = [ 0, 0, 0 ]
90 uint64_t blk_birth = 0x4acc
98 uint64_t blk_fill = 0x11
a0 zio_cksum_t blk_cksum = {
    a0 uint64_t [4] zc_word = [ 0x8348a7aa95, 0x8a9a1c0eb664,
0x5b0e32bd611ac7, 0x2d691acc5e1f456f ]
}
}
]
... <-- output omitted
>

```

So, now will use the blkptr dcmd to get a "nicer" view of the blkptr_t in the objset_phys_t.

```

> 40::blkptr
DVA[0]: vdev_id 0 / 5000
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 80000000000
DVA[0]: :0:5000:800:d
DVA[1]: vdev_id 0 / 1c0005000
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 80000000000
DVA[1]: :0:1c0005000:800:d
DVA[2]: vdev_id 0 / 380003800
DVA[2]:   GANG: FALSE GRID: 0000 ASIZE: 80000000000
DVA[2]: :0:380003800:800:d
LSIZE: 4000           PSIZE: 800
ENDIAN: LITTLE           TYPE: DMU dnode
BIRTH: 4acc           LEVEL: 0   FILL: 1100000000
CKFUNC: fletcher4           COMP: lzjb
CKSUM: 8348a7aa95:8a9a1c0eb664:5b0e32bd611ac7:2d691acc5e1f456f
> $q
#

```

The objset_phys_t contain block number(s) of either indirect blocks or block numbers of a block containing an array of dnode_phys_t. This is from the dn_type and dn_nlevels fields in the dnode_phys_t, or the "TYPE" information (and LEVEL) information in the blkptr_t. For this objset_phys_t, the blkptr_t is for an array of dnode_phys_t, (no indirection since LEVEL is 0). This array of dnode_phys_t makes up the meta object set, or "MOS". Now we'll go back to zdb to dump the decompressed dnode_phys_t array, using block number 5000.

```
# zdb -R lacedisk:c4t0d0p1:5000:800:d,lzjb,4000 2> /tmp/mos
Found vdev: /dev/dsk/c4t0d0p1
#
```

The physical size of the block is 0x800 bytes, the logical size (after decompression) is 0x4000 bytes (from the output of ::blkptr above).

```
# mdb /tmp/mos
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> ::sizeof zfs`dnode_phys_t <-- how large is a dnode_phys_t?
sizeof (zfs`dnode_phys_t) = 0x200
> 4000%200=K <-- how many dnode_phys_t are there in the block?
20
```

```
> 0,20::print -a -t zfs`dnode_phys_t <-- dump the 32 dnode_phys_t of the objset
{
  0 uint8_t dn_type = 0 <-- DMU_OT_NONE (not in use)
  ... <-- output truncated
}
{
  200 uint8_t dn_type = 0x1 <-- DMU_OT_OBJECT_DIRECTORY
  ... <-- output omitted
  240 blkptr_t [1] dn_blkptr = [
    {
      240 dva_t [3] blk_dva = [
        {
          240 uint64_t [2] dva_word = [ 0x1, 0x1 ]
        }
        {
          250 uint64_t [2] dva_word = [ 0x1, 0xe00001 ]
        }
        {
          260 uint64_t [2] dva_word = [ 0x1, 0x1c00000 ]
        }
      ]
      270 uint64_t blk_prop = 0x8001070200000000
      278 uint64_t [3] blk_pad = [ 0, 0, 0 ]
      290 uint64_t blk_birth = 0x4
      298 uint64_t blk_fill = 0x1
      2a0 zio_cksum_t blk_cksum = {
        2a0 uint64_t [4] zc_word = [ 0x5a6f58679, 0x1cf035fcb09,
0x528ae171d3a8, 0xaabf09f1373ae ]
      }
    }
  ]
  2c0 uint8_t [320] dn_bonus = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ... ]
}
```

```

{
  400 uint8_t dn_type = 0xc <-- DMU_OT_DSL_DIR
... <-- output omitted
}
{
  600 uint8_t dn_type = 0xf <-- DMU_OT_DSL_PROPS
... <-- output omitted
}
{
  800 uint8_t dn_type = 0xd <-- DMU_OT_DSL_DIR_CHILD_MAP
... <-- output omitted
}
{
  a00 uint8_t dn_type = 0x10 <-- DMU_OT_DSL_DATASET
... <-- output omitted
}
... <-- output omitted, several other objects, and some unused entries
>

```

In the above, the first `dnnode_phys_t` is not used. The second (object id #1) is for a `DMU_OT_OBJECT_DIRECTORY`. The object directory is always object id 1 in the `objset_phys_t`. The `blkptr_t` for the object directory is a zap object that contains object names and ids to allow one to find all other objects (besides the object directory).

Now, we'll dump the `blkptr_t` at 0x240 (this is the `blkptr_t` for the `DMU_OT_OBJECT_DIRECTORY`).

```

> 240::blkptr
DVA[0]: vdev_id 0 / 200
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 2000000000
DVA[0]: :0:200:200:d
DVA[1]: vdev_id 0 / 1c0000200
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 2000000000
DVA[1]: :0:1c0000200:200:d
DVA[2]: vdev_id 0 / 380000000
DVA[2]:   GANG: FALSE GRID: 0000 ASIZE: 2000000000
DVA[2]: :0:380000000:200:d
LSIZE: 200          PSIZE: 200
ENDIAN: LITTLE          TYPE: object directory
BIRTH: 4            LEVEL: 0   FILL: 10000000
CKFUNC: fletcher4      COMP: uncompressed
CKSUM: 5a6f58679:1cf035fcb09:528ae171d3a8:aabf09f1373ae
> $q
#

```

The `blkptr` is for an object directory, and is uncompressed. We'll use `zdb` to get the raw output of the object directory.

```
# zdb -R lacedisk:c4t0d0p1:200:200:r 2> /tmp/objdir
```

```
Found vdev: /dev/dsk/c4t0d0p1
#
```

An object directory (dn_type == DMU_OT_OBJECT_DIRECTORY) is a ZAP (ZFS Attribute Processor) object. ZAP objects can be "fatzaps" or "microzaps". ZAP objects contain name-value pairs used to store attributes of an object. The first 64-bit value in a ZAP object is used to identify the type of ZAP contents contained within the ZAP block. A value of ZBT_MICRO (=0x8000000000000003) indicates a microzap. A microzap contains all of the attributes within a single block. A value of ZBT_HEADER (= 0x8000000000000001) is used for the first block of a fatzap. All other blocks for fatzap objects have a value of ZBT_LEAF (=0x8000000000000000) in the first 64-bits of the block.

```
# mdb /tmp/objdir
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0/J
0:      8000000000000003 <-- this is a microzap

> 0::print -a -t zfs`mzap_phys_t <-- print an mzap_phys_t (see
uts/common/fs/zfs/sys/zap_impl.h)
{
  0 uint64_t mz_block_type = 0x8000000000000003
  8 uint64_t mz_salt = 0x41d10a3
  10 uint64_t mz_normflags = 0
  18 uint64_t [5] mz_pad = [ 0, 0, 0, 0, 0 ]
  40 mzap_ent_phys_t [1] mz_chunk = [
    {
      40 uint64_t mze_value = 0x2
      48 uint32_t mze_cd = 0
      4c uint16_t mze_pad = 0
      4e char [50] mze_name = [ "root_dataset" ]
    }
  ]
}
```

The structure used to maintain microzap objects is a mzap_phys_t. The last member of the structure is a variable sized array of mzap_ent_phys_t structures. Each mzap_ent_phys_t contains a name-value pair. Possible attributes for the object directory include: "root_dataset", "config", "sync_bplist", and "deflate". The root_dataset attribute contains the object number of the root DSL (Dataset and Snapshot Layer) directory for the pool. The root DSL contains information about all top level datasets within the pool. It is an object of type DMU_OT_DSL_DIR. We'll take a closer look at this object shortly.

Below, we look at the next 2 mzap_ent_phys_t structures in the object directory. One is a "config" object with contains the object number of a DMU_OT_PACKED_NVLIST, the other is a "deflate" object. (For more information on these, refer to the source code).

```
> ::sizeof zfs`mzap_ent_phys_t
sizeof (zfs`mzap_ent_phys_t) = 0x40
> 80::print -a -t zfs`mzap_ent_phys_t
```

```

{
  80 uint64_t mze_value = 0xb
  88 uint32_t mze_cd = 0
  8c uint16_t mze_pad = 0
  8e char [50] mze_name = [ "config" ]
}
> c0::print -a -t zfs`mzap_ent_phys_t
{
  c0 uint64_t mze_value = 0x1
  c8 uint32_t mze_cd = 0
  cc uint16_t mze_pad = 0
  ce char [50] mze_name = [ "deflate" ]
}

> $q
#

```

Now, we take a look at object id 2, the "root_dataset". This is the third `dnode_phys_t` in the array of `dnode_phys_t` referenced by the `objset_phys_t`. For this, we go back to the `dnode_phys_t` array from the `objset_phys_t` referenced by the uberblock, and look at object id 2. The object starts at 0x400 (2 * `sizeof(dnode_phys_t)`) from the beginning of the array.

```

# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 400::print zfs`dnode_phys_t
{
  400 uint8_t dn_type = 0xc <-- DMU_OT_DSL_DIR
  ... <-- output omitted
  404 uint8_t dn_bonustype = 0xc
  ... <-- output omitted
  40a uint16_t dn_bonuslen = 0x100
  ... <-- output omitted
  440 blkptr_t [1] dn_blkptr = [
    {
      440 dva_t [3] blk_dva = [
        {
          440 uint64_t [2] dva_word = [ 0, 0 ] <-- blkptr not used
        }
      ]
    }
  ]
  ... <-- output omitted
  4c0 uint8_t [320] dn_bonus = [ 0x34, 0x9c, 0xb9, 0x47, 0, 0, 0, 0, 0x5, 0, 0,
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ... ]
}
>

```

This `dnode_phys_t` has information in the `dn_bonus` section. The type of this information (`dn_bonustype == 0xc`) is a `DMU_OT_DSL_DIR` (see `uts/common/fs/zfs/dmu.h`). DSL directory objects store a `dsl_dir_phys_t` in the bonus buffer (`dn_bonus`) of the `dnode_phys_t`. Note that the


```

        b50 uint64_t [2] dva_word = [ 0x1, 0xe00026 ]
    }
    {
        b60 uint64_t [2] dva_word = [ 0, 0 ]
    }
]
... <-- output omitted
>

```

Here, there is a blkptr_t at 0xb40 bytes. We'll take a look at this.

```

> b40::blkptr
DVA[0]: vdev_id 0 / 4c00
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 20000000000
DVA[0]: :0:4c00:200:d
DVA[1]: vdev_id 0 / 1c0004c00
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 20000000000
DVA[1]: :0:1c0004c00:200:d
LSIZE: 400          PSIZE: 200
ENDIAN: LITTLE          TYPE: DMU objset
BIRTH: 4acc          LEVEL: 0   FILL: 1500000000
CKFUNC: fletcher4          COMP: lzjb
CKSUM: a9a691571:477e2285748:f44e24c94c3d:23418ff35bb270
> $q
#

```

Note that this is the second DMU objset we have seen. The first was pointed to by the uberblock (see output of `zdb -uuu lacedisk` above). The first objset is for all objects in the pool. The second object set is for the DSL objects. The metadnode in this object set is for the root dataset of the file system. The blkptr_t specifies physical size 0x200 and logical size 0x400, with lzjb compression. We'll use `zdb` to dump the block.

```

# zdb -R lacedisk:c4t0d0p1:4c00:200:d,lzjb,400 2> /tmp/root_dataset_metadnode
Found vdev: /dev/dsk/c4t0d0p1
#

```

And now back to `mdb` to examine the (decompressed) data of the block. This is an `objset_phys_t`.

```

# mdb /tmp/root_dataset_metadnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::print -a -t zfs`objset_phys_t
{
  0 dnode_phys_t os_meta_dnode = {
    0 uint8_t dn_type = 0xa <-- DMU_OT_DNODE
    1 uint8_t dn_indblkshift = 0xe
    2 uint8_t dn_nlevels = 0x7 <-- levels of indirection
... <-- output omitted
    40 blkptr_t [1] dn_blkptr = [

```

```

    {
      40 dva_t [3] blk_dva = [
        {
          40 uint64_t [2] dva_word = [ 0x2, 0x24 ]
        }
        {
          50 uint64_t [2] dva_word = [ 0x2, 0xe00024 ]
        }
        {
          60 uint64_t [2] dva_word = [ 0, 0 ]
        }
      ]
    }
... <-- output omitted
>

```

Decoding the first blkptr_t gives:

```

> 40::blkptr
DVA[0]: vdev_id 0 / 4800
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 40000000000
DVA[0]: :0:4800:400:id
DVA[1]: vdev_id 0 / 1c0004800
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 40000000000
DVA[1]: :0:1c0004800:400:id
LSIZE: 4000          PSIZE: 400
ENDIAN: LITTLE          TYPE: DMU dnode
BIRTH: a1d0          LEVEL: 6   FILL: 1400000000
CKFUNC: fletcher4          COMP: lzjb
CKSUM: 5b1be31814:3f256d9bc248:172d1fb48446d2:5f55736a7cf2c95
> $q
#

```

To get to the `dnode_phys_t` structure(s) that are referenced by the `blkptr_t`, we must go through 6 levels of indirection to get to the `blkptr_t`'s that contain the `dnode_phys_t`'s (7 levels to get to the `dnode_phys_t` structures themselves).

Note the `LEVEL` value of 6. This `blkptr_t` is for a block containing `blkptr_t`'s of `blkptr_t`'s. It is 0x400 bytes on the disk, and, after `lzjb` decompression is 0x4000 bytes large. We want the `dnode_phys_t` for the root of the file system. To get this, we need the `dnode_phys_t` whose `dn_type` is `DMU_OT_MASTER_NODE`. This `dnode_phys_t` is always object id 1. This means that regardless of the levels of indirection, we always want the first indirect block at every level. When we get to the block containing `dnode_phys_t`, we want the second `dnode_phys_t` in the array.

```

# zdb -R lacedisk:c4t0d0p1:4800:200:d,lzjb,4000 2> /tmp/blkptr6
Found vdev: /dev/dsk/c4t0d0p1
#
# mdb /tmp/blkptr6
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so

```

```

> 0::blkptr
DVA[0]: vdev_id 0 / 4400
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[0]: :0:4400:400:id
DVA[1]: vdev_id 0 / 1c0004400
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[1]: :0:1c0004400:400:id
LSIZE: 4000           PSIZE: 400
ENDIAN: LITTLE           TYPE: DMU dnode
BIRTH: a1d0           LEVEL: 5   FILL: 1400000000
CKFUNC: fletcher4       COMP: lzjb
CKSUM: 5b9be7d355:3f9ee0f14633:1766faff22becd:607cb255f0cef14

```

Here is the level 5 indirect block. Let's read it and dump it out.

```

# zdb -R lacedisk:c4t0d0p1:4400:400:d,lzjb,4000 2> /tmp/blkptr5
Found vdev: /dev/dsk/c4t0d0p1
#

```

Now, back to mdb to display the blkptr_t array. Each blkptr_t takes up 0x80 bytes, so in a 0x4000 byte array of blkptr_t, there are 0x80 blkptr_t.

```

# mdb /tmp/blkptr5
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 4000
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[0]: :0:4000:400:id
DVA[1]: vdev_id 0 / 1c0004000
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[1]: :0:1c0004000:400:id
LSIZE: 4000           PSIZE: 400
ENDIAN: LITTLE           TYPE: DMU dnode
BIRTH: 71d2           LEVEL: 4   FILL: 1400000000
CKFUNC: fletcher4       COMP: lzjb
CKSUM: 572ea43879:3c2b4d3c6fed:15ebe34a088538:59553188934fe94
> $q
#

```

Note that the LEVEL value is now 4. Also, out of 128 blkptr_t at level 6, only one is being used. When do others get used? When more files are added to the file system. This file system only uses about 3.1MB out of an available 37GB (from "df -h" output). How do we pick the blkptr_t that we need? This depends on the object id (in UFS, the "inumber") in which we are interested. Here, we want to start from the root of the filesystem and traverse pathnames. So we start from the DMU_OT_MASTER_NODE, which is always object id 1. So, we'll use the first blkptr_t as our starting point (which is good in this case, since there is only one valid blkptr_t in the array).

Now, we'll use zdb to dump the blkptr_t array from this level of indirection. The blkptr_t array is at

physical block number 0x4000 and the (lzjb compressed) size on disk is 0x400 bytes. The uncompressed size is 0x4000 bytes.

```
# zdb -R lacedisk:c4t0d0p1:4000:400:d,lzjb,4000 2> /tmp/blkptr4
Found vdev: /dev/dsk/c4t0d0p1
#
```

And again mdb to display the blkptr_t array. Again, we'll only look at the first indirect blkptr_t.

```
# mdb /tmp/blkptr4
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 2000
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 40000000000
DVA[0]: :0:2000:400:id
DVA[1]: vdev_id 0 / 1c0002000
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 40000000000
DVA[1]: :0:1c0002000:400:id
LSIZE: 4000          PSIZE: 400
ENDIAN: LITTLE          TYPE: DMU dnode
BIRTH: 4acc          LEVEL: 3   FILL: 1400000000
CKFUNC: fletcher4          COMP: lzjb
CKSUM: 5b1a31ac6b:3f202983b5d8:1728de4368d22a:5f36cda2d00cc77
```

```
$q
#
```

The first blkptr_t in the array is for level 3 indirection. Again, it is lzjb compressed and is 0x400 bytes on the disk. The uncompressed (LSIZE) is 0x4000 bytes. The second blkptr_t is unallocated (as, we'll assume, are the remaining blkptr_t).

Now, back to zdb to get the next level of indirection.

```
# zdb -R lacedisk:c4t0d0p1:2000:400:d,lzjb,4000 2> /tmp/blkptr3
Found vdev: /dev/dsk/c4t0d0p1
#
```

And back to mdb to display the array of 0x80 entries (4000%80, where 0x4000 is the uncompressed size and 0x80 is the size of a blkptr_t). And again, we're only interested in the first entry.

```
# mdb /tmp/blkptr3
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 1800
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 40000000000
DVA[0]: :0:1800:400:id
DVA[1]: vdev_id 0 / 1c0001800
```

```
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[1]: :0:1c0001800:400:id
LSIZE: 4000           PSIZE: 400
ENDIAN: LITTLE           TYPE: DMU dnode
BIRTH: 4acc           LEVEL: 2   FILL: 1400000000
CKFUNC: fletcher4     COMP: lzjb
CKSUM: 5971752f5e:3d8cfbe170db:1668b663cd16fc:5b61debef02b3cd
```

```
> $q
#
```

And we'll go back to zdb to get the blkptr_t array specified for level 2 indirection.

```
# zdb/i386/zdb -R lacedisk:c4t0d0p1:1800:400:d,lzjb,4000 2> /tmp/blkptr2
Found vdev: /dev/dsk/c4t0d0p1
#
```

And back to mdb. (It would be nice if one could do all of this within mdb or zdb. Or maybe a new tool, mzdb/zmdb anyone?).

```
# mdb /tmp/blkptr2
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 1400
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[0]: :0:1400:400:id
DVA[1]: vdev_id 0 / 1c0001400
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[1]: :0:1c0001400:400:id
LSIZE: 4000           PSIZE: 400
ENDIAN: LITTLE           TYPE: DMU dnode
BIRTH: 4acc           LEVEL: 1   FILL: 1400000000
CKFUNC: fletcher4     COMP: lzjb
CKSUM: 5853930aed:3d385d7c3c5f:166a8b5f2e721c:5bd9492aa861e02
> 80::blkptr
LSIZE: 0             PSIZE: 200
ENDIAN: BIG           TYPE: unallocated
BIRTH: 0             LEVEL: 0   FILL: 0
CKFUNC: inherit      COMP: inherit
CKSUM: 0:0:0:0
> $q
#
```

And again to zdb to get the blkptr_t array at level 1.

```
# /zdb -R lacedisk:c4t0d0p1:1400:400:d,lzjb,4000 2> /tmp/blkptr1
Found vdev: /dev/dsk/c4t0d0p1
#
```

And once more to mdb...

```
# mdb /tmp/blkptr1
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0::blkptr
DVA[0]: vdev_id 0 / 3600
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: a000000000
DVA[0]: :0:3600:a00:d
DVA[1]: vdev_id 0 / 1c0003600
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: a000000000
DVA[1]: :0:1c0003600:a00:d
LSIZE: 4000           PSIZE: a00
ENDIAN: LITTLE           TYPE: DMU dnode
BIRTH: 4acc           LEVEL: 0   FILL: 1400000000
CKFUNC: fletcher4           COMP: lzjb
CKSUM: ad27cf4b34:ecbe6f671f03:c231a96352c039:7726f4dc30e0e44b
> $q
#
```

This is the level 0 blkptr_t. It is 0xa00 bytes on the disk, and logical (i.e., decompressed) size is 0x4000 bytes. We'll go back to zdb once more to get this block. This should be an array of dnode_phys_t structures.

```
# zdb -R lacedisk:c4t0d0p1:3600:a00:d,lzjb,4000 2> /tmp/dnode
Found vdev: /dev/dsk/c4t0d0p1
#
```

Let's use mdb again. This time, instead of indirect blkptr_t, we should have dnode_phys_t.

```
# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0,20::print -a -t zfs`dnode_phys_t <-- ",20" is the number of dnode_phys_t in 0x4000
<-- bytes (4000 divided by sizeof dnode_phys_t)
{
  0 uint8_t dn_type = 0 <-- first entry not used
  ... <-- output omitted
  {
    200 uint8_t dn_type = 0x15 <-- DMU_OT_MASTER_NODE (from
uts/common/fs/zfs/sys/dmu.h)
    201 uint8_t dn_indblkshift = 0xe
    ... <-- output omitted
    240 blkptr_t [1] dn_blkptr = [
      {
        240 dva_t [3] blk_dva = [
          {
            240 uint64_t [2] dva_word = [ 0x1, 0x63 ]
```

```

    }
... <-- output omitted
{
  600 uint8_t dn_type = 0x14 <-- DMU_OT_DIRECTORY_CONTENTS (a directory)
... <-- output omitted
}
{
  800 uint8_t dn_type = 0x13 <-- DMU_OT_PLAIN_FILE_CONTENTS (a regular file)
... <-- output omitted
}
... <-- output omitted. The remaining dnode_phys_t are either
  <-- DMU_OT_PLAIN_FILE_CONTENTS or DMU_OT_NONE
>

```

The DMU_OT_MASTER_NODE dnode_phys_t is a ZAP object. It is used to identify the root directory, delete queue, and version information for a file system. Let's look at the blkptr_t.

```

> 240::blkptr
DVA[0]: vdev_id 0 / c600
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 2000000000
DVA[0]: :0:c600:200:d
DVA[1]: vdev_id 0 / 1c000c600
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 2000000000
DVA[1]: :0:1c000c600:200:d
LSIZE: 200          PSIZE: 200
ENDIAN: LITTLE          TYPE: ZFS master node
BIRTH: 6           LEVEL: 0   FILL: 100000000
CKFUNC: fletcher4      COMP: uncompressed
CKSUM: 264216abd:f8ce291b17:347052edfaa6:79edede0bbfd6
> $q
#

```

So, this is for a 0x200 byte block and is uncompressed. We'll use zdb to dump out the block.

```

# zdb -R lacedisk:c4t0d0p1:c600:200:r 2> /tmp/zfs_master_node
Found vdev: /dev/dsk/c4t0d0p1
#

```

And back to mdb to display the contents.

```

# mdb /tmp/zfs_master_node
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0/J
0:      8000000000000003 <-- this is a microzap
> 0::print -t -a zfs`mzap_phys_t
{
  0 uint64_t mz_block_type = 0x8000000000000003
... <-- output omitted

```

```

40 mzap_ent_phys_t [1] mz_chunk = [
  {
    40 uint64_t mze_value = 0x3
... <-- output omitted
    4e char [50] mze_name = [ "VERSION" ]
  }
]
}
80::print -a -t zfs`mzap_ent_phys_t
{
... <-- output omitted
  8e char [50] mze_name = [ "DELETE_QUEUE" ]
}
c0::print -a -t zfs`mzap_ent_phys_t
{
  c0 uint64_t mze_value = 0x3 <-- the object id for the root directory of the fs
... <-- output omitted
  ce char [50] mze_name = [ "ROOT" ]
}
$q
#

```

Now, back to the `dnode_phys_t` array to look at object id 3, the object id for the root directory of the ZFS file system.

```

# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 3*200::print -a -t zfs`dnode_phys_t
{
  600 uint8_t dn_type = 0x14 <-- DMU_OT_DIRECTORY_CONTENTS from before
... <-- output omitted
  604 uint8_t dn_bonustype = 0x11 <-- DMU_OT_ZNODE (from dmu.h)
... <-- output omitted
  640 blkptr_t [1] dn_blkptr = [
    {
      640 dva_t [3] blk_dva = [
        {
          640 uint64_t [2] dva_word = [ 0x1, 0x51088 ]
        }
      ]
    }
  ]
... <-- output omitted
}
>

```

This `dnode_phys_t` has both `blkptr_t` and bonus buffer. The bonus buffer is a `znode_phys_t`. Let's take a look.

```

> 6c0::print -a -t zfs`znode_phys_t
{
  6c0 uint64_t [2] zp_atime = [ 0x47c08f1b, 0x31e41b57 ]
}

```

```
... <-- output omitted, ownership, other time stamps, size, etc.
}
> 6c0/Y
0x6c0:      2008 Feb 23 22:24:43  <-- when the fs was last accessed(?)
```

Now, let's look at the blkptr_t.

```
> 640::blkptr
DVA[0]: vdev_id 0 / a211000
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 20000000000
DVA[0]: :0:a211000:200:d
DVA[1]: vdev_id 0 / 1c003a400
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 20000000000
DVA[1]: :0:1c003a400:200:d
LSIZE: 600          PSIZE: 200
ENDIAN: LITTLE          TYPE: ZFS directory
BIRTH: 46b6          LEVEL: 0   FILL: 100000000
CKFUNC: fletcher4          COMP: lzjb
CKSUM: 172e5bed24:7e94f1c3dba:179eeeb7275:325d97fad3423c
> $q
#
```

This blkptr_t is for a ZFS directory (should be the root directory of the file system). We'll use zdb to dump this. It is 0x200 bytes on the disk, and the decompressed size is 0x600 bytes.

```
# zdb -R lacedisk:c4t0d0p1:a211000:200:d,lzjb,600 2> /tmp/zfs_root_directory
Found vdev: /dev/dsk/c4t0d0p1
#
```

The ZFS directory blkptr_t is a ZAP object. We'll look at the first 64-bits to determine the type of ZAP.

```
# mdb /tmp/zfs_root_directory
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 0/J
0:      8000000000000003 <-- this is a microzap
```

Ok. It's a microzap. Let's take a look.

```
> 0::print -a -t zfs`mzap_phys_t
{
  0 uint64_t mz_block_type = 0x8000000000000003
<-- output omitted
  40 mzap_ent_phys_t [1] mz_chunk = [
    {
      40 uint64_t mze_value = 0x8000000000000004 <-- the object id (inumber equivalent)
      48 uint32_t mze_cd = 0
      4c uint16_t mze_pad = 0
```

```

    4e char [50] mze_name = [ "foo" ] <-- file in the root directory of the file system
  }
]
}

```

Let's dump all of the `mzap_ent_phys_t` in the block. The block is 0x600 bytes large, and the size of an `mzap_ent_phys_t` is 0x40 bytes. We start 0x40 bytes into the block and dump the remainder of the block as `mzap_ent_phys_t` structures. Each entry in this microzap is a directory entry.

```

> ::sizeof zfs`mzap_ent_phys_t
sizeof (zfs`mzap_ent_phys_t) = 0x40
> 40,(600-40)%40::print -a -t zfs`mzap_ent_phys_t
{
  40 uint64_t mze_value = 0x8000000000000004 <-- low-order bits are object id numbers
                                     <-- 8 at the high bits is type info
  48 uint32_t mze_cd = 0
  4c uint16_t mze_pad = 0
  4e char [50] mze_name = [ "foo" ]
}
... <-- output omitted
{
  440 uint64_t mze_value = 0x8000000000000015 <-- ls -i should show 21 for this file
  448 uint32_t mze_cd = 0
  44c uint16_t mze_pad = 0
  44e char [50] mze_name = [ "words" ] <-- here's the file we want
}
{
  480 uint64_t mze_value = 0 <-- unused
... <-- output omitted
> $q
#

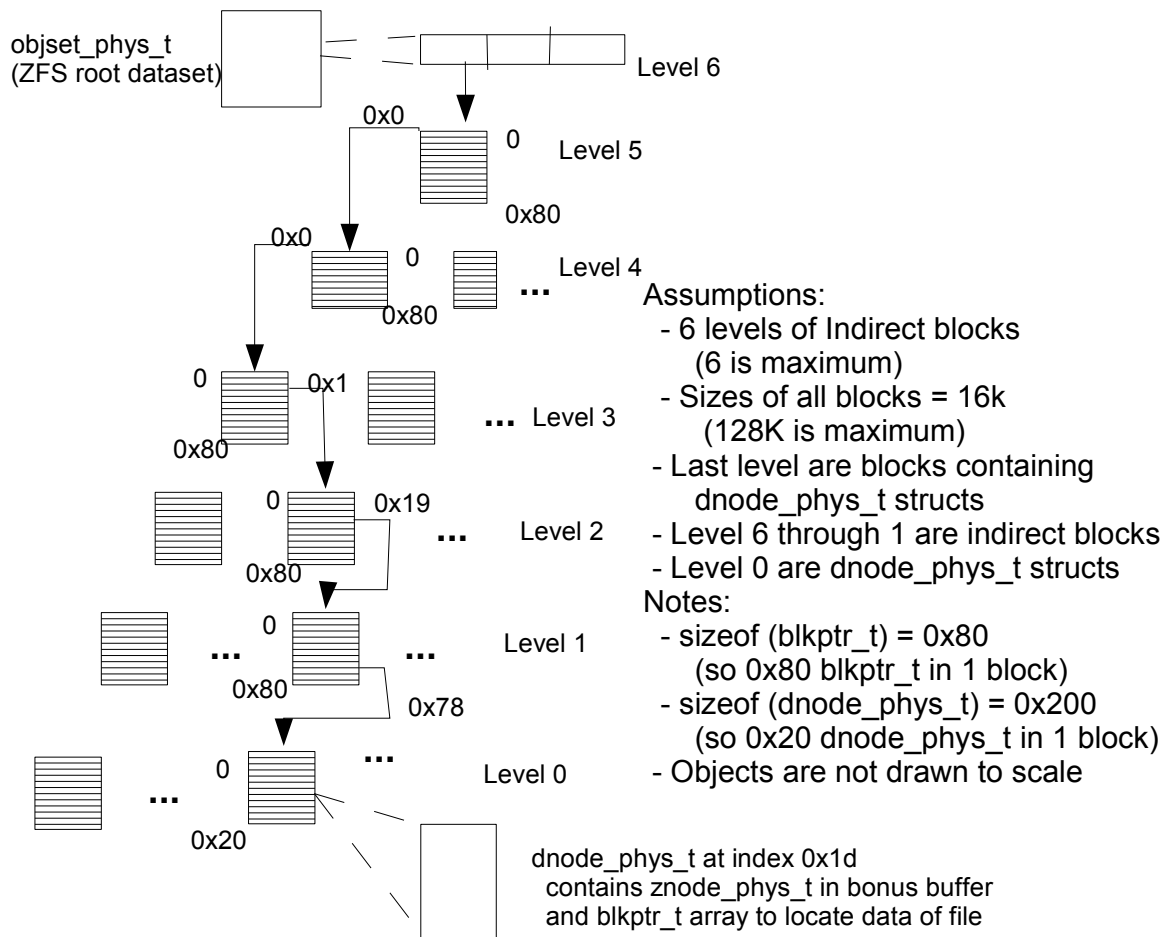
```

Now, we go back to the `dnode_phys_t` array for object id 0x15. Note that if the object id is $\geq 0x20$, we would need to look at other (indirect) `blkptr_t` to get the `dnode_phys_t` we want. Here, because the file system does not have many files, everything "fits" so that we don't have to go through too many levels of indirection.

Let's say, for example, that the object id is 0x99f1d, and the number of levels of indirect blocks for the `DMU_OT_DNODE` (`dn_type = 0xa`) for the DSL dataset is 7, as in the above. To get the index into the indirect `blkptr_t` array at each level requires a little arithmetic. Assuming each block of `dnode_phys_t` contains 0x20 `dnodes`, the index into the array is simply $0x99f1d \& 0x1f (= 0x1d)$. Also assuming that a block of indirect `blkptr_t`'s contains 0x80 entries (as in the above case), the index into the level 1 indirect `blkptr_t` is $(0x99f1d \gg 5) \& 0x7f (= 0x78)$. The second level index is $(0x99f1d \gg 0xc) \& 0x7f (= 0x19)$. The third level is $(0x99f1d \gg 0x13) \& 0x7f (= 0x1)$. The fourth level is $(0x99f1d \gg 1a) \& 0x7f (= 0x0)$. The fifth level is $(0x99f1d \gg 21) \& 0x7f (= 0x0)$. The sixth level index is $(0x99f1d \gg 28) \& 7f (= 0x0)$. So, at Level 6, 5, and 4, you use the 0th `blkptr_t`. At Level 3, the 1th (second) `blkptr_t`. At level 2, the 0x78th `blkptr_t`. And at level 0 (the array of `dnode_phys_t`), the 0x1d entry. (I have done this with the modified `mdb/zdb`, but for this paper, it is left as an exercise for the reader. Here, we'll use the nice small object id because it avoids the arithmetic. We basically take the

0th entry of all the indirect blocks and the 0x15 entry from the dnode_phys_t array at the end).

The following diagram shows the traversal of indirect blocks for object id 0x99f1d.



Example – Find dnode_phys_t for object id = 0x99f1d

Level 0 index = 0x1d (0x99f1d & 0x1f)
 Level 1 index = 0x78 ((0x99f1d >> 5) & 0x7f)
 Level 2 index = 0x19 ((0x99f1d >> 0xc) & 0x7f)
 Level 3 index = 0x1 ((0x99f1d >> 0x13) & 0x7f)
 Level 4 index = 0 ((0x99f1d >> 0x1a) & 0x7f)
 Level 5 index = 0 ((0x99f1d >> 0x21) & 0x7f)
 Level 6 index = 0, 1, or 2 (normally, there are 3 copies of the data)

Note: With larger block sizes (for instance 128k), shifts and masks change accordingly. Correct shift and mask values are left as an exercise for the reader.

```

# mdb /tmp/dnode
> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 15*200::print -a -t zfs`dnode_phys_t
{
  2a00 uint8_t dn_type = 0x13 <-- DMU_OT_PLAIN_FILE_CONTENTS
  2a01 uint8_t dn_indblkshift = 0xe
  2a02 uint8_t dn_nlevels = 0x2 <-- one layer of indirect blocks
  2a03 uint8_t dn_nblkptr = 0x1
  2a04 uint8_t dn_bonustype = 0x11 <-- DMU_OT_ZNODE (for "words" file)
... <- output omitted
  2a40 blkptr_t [1] dn_blkptr = [
    {
      2a40 dva_t [3] blk_dva = [
        {
          2a40 uint64_t [2] dva_word = [ 0x2, 0x51054 ]
        }
      ]
    }
  ]
... <- output omitted
}

```

The dn_bonus buffer contains file system attributes for the file in a znode_phys_t (time stamps, ownership, size, etc.). We just want the data, so let's dump the first level indirect blkptr_t.

```

> 2a40::blkptr
DVA[0]: vdev_id 0 / a20a800
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[0]: :0:a20a800:400:id
DVA[1]: vdev_id 0 / 1c003a800
DVA[1]:   GANG: FALSE GRID: 0000 ASIZE: 4000000000
DVA[1]: :0:1c003a800:400:id
LSIZE: 4000          PSIZE: 400
ENDIAN: LITTLE          TYPE: ZFS plain file
BIRTH: 46b6          LEVEL: 1   FILL: 200000000
CKFUNC: fletcher4          COMP: lzjb
CKSUM: 5d33a847e1:3e4f850ab0b0:16415a51e8464d:59b54146c060c42
> $q
#

```

And we'll use zdb to get the block.

```

# zdb -R lacedisk:c4t0d0p1:a20a800:400:d,lzjb,4000 2> /tmp/words_indirect
Found vdev: /dev/dsk/c4t0d0p1
#

```

And back into mdb to display it.

```

# mdb /tmp/words_indirect

```

```

> ::loadctf
> ::load /export/home/max/source/mdb/i386/rawzfs.so
> 4000%80=K
      80 <-- There is space for 0x80 blkptr_t in this indirect block.
> 0::blkptr
DVA[0]: vdev_id 0 / a220000
DVA[0]:   GANG: FALSE GRID: 0000 ASIZE: 2000000000000
DVA[0]: :0:a220000:20000:d
LSIZE: 20000          PSIZE: 20000
ENDIAN: LITTLE          TYPE: ZFS plain file
BIRTH: 46b6          LEVEL: 0   FILL: 100000000
CKFUNC: fletcher2          COMP: uncompressed
CKSUM: 281ad9d864b9dc57:79fe4143faf3e2b7:5e064a117c12a92e:5b788125e084c6b2
>
> $q
#

```

This is the blkptr_t for a 0x20000 byte (128KB) block of uncompressed file data. The block should contain the first 128KB of the copy of /usr/dict/words. We'll use zdb to dump it out. Note that here, there is only one entry in the DVA in use, I.e., only 1 ditto block. This is the default for file data. In SXCE and Solaris 10 Update 6(?), the “zfs set copies” command will allow the number of ditto blocks to be set for regular file data.

```

# zdb -R lacedisk:c4t0d0p1:a220000:20000:r
Found vdev: /dev/dsk/c4t0d0p1
10th
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
a
AAA
AAAS
Aarhus
Aaron
AAU
ABA
Ababa
aback
... <-- output omitted

```

And here is the first 128KB of the copy of the /usr/dict/words file.

Future Work

It would be very nice to have a version of mdb with zdb integrated into it. I have tried various combinations (for instance, including a decompress dcmd within mdb), but I took the easiest route.

Examination of snapshots, ZIL (ZFS Intent Log), and various other features of ZFS should be possible using the same techniques.

A similar walk through of the data structures in kernel memory would be nice. This is much simpler, and one can use mdb for this.

Conclusion

I have found that in writing this paper, access to the source code, and to the ZFS On-Disk Specification paper have proven to be invaluable in understanding the disk layout of ZFS. The paper really does cover everything about the on-disk layout. I found that the ability to "walk through" the data helped me to understand what the on-disk specification paper is saying. The layout is complex, but not unnecessarily so.