Contents

Chapter 1

On-disk Format

1.1 Vdev

ZFS storage pools are essentially a collection of *virtual devices*, or *vdevs*, which are arranged in a tree with physical vdevs sitting at leaves. Figure. 1.1 illustrates such a tree representing a sample pool configuration containing two mirrors, each of which has two disks.

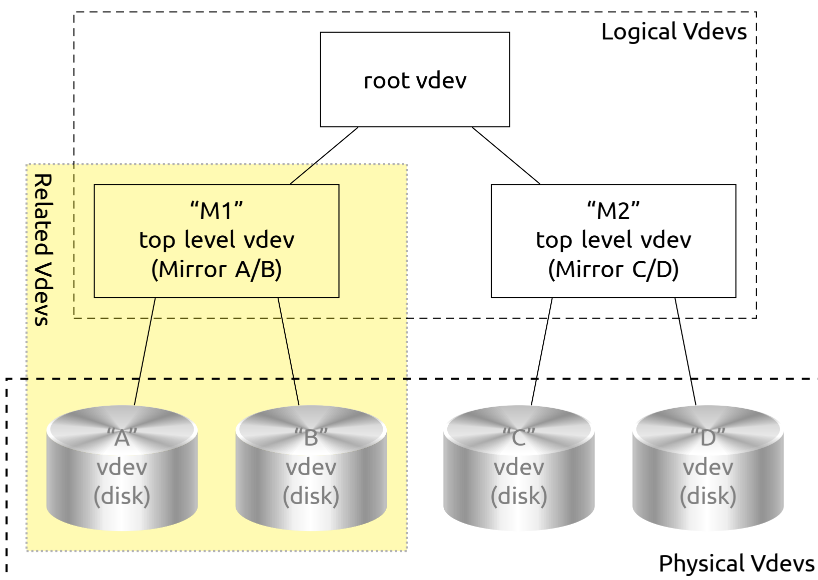

Illustration 1.1: Vdev Tree

1.2 Label and Uberblock

Each physical vdev within a storage pool contains four identical copies of 256KB structure called a vdev *label* (`vdev_label_t`). The vdev label contains information describing this physical vdev and all other vdevs which share a common top-level vdev as an ancestor. For example, in Figure. 1.1, the vdev label on D3 would contain information describing D3, D4, and M2. Any copy of the labels can be used to access and verify the pool. There are two labels at the front of the device and two labels at the back. Figure. 1.2 illustrates the physical layout of vdev labels.

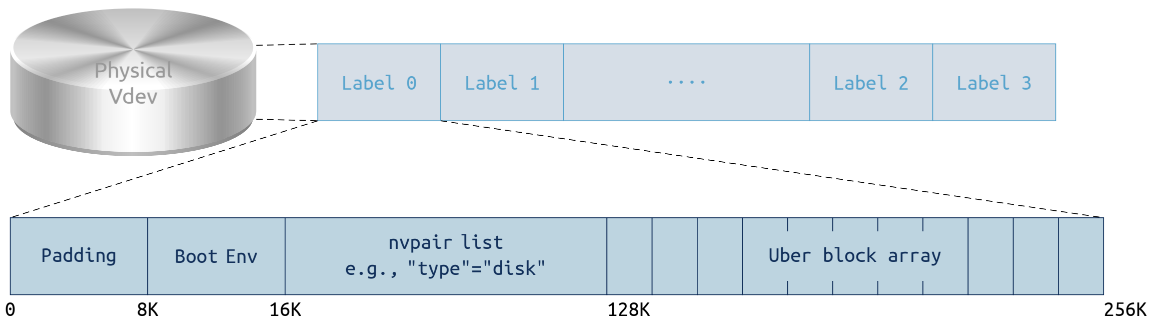

Illustration 1.2: From label to MOS

An *uberblock* (`uberblock_t`) contains information necessary to access the contents of the pool. Only one uberblock in the pool is *active* at any time. The uberblock with the highest transaction group number and valid checksum is the active uberblock. Figure. 1.3 illustrates the layout of an active uberblock, and how accessing the storage pool can start from it.

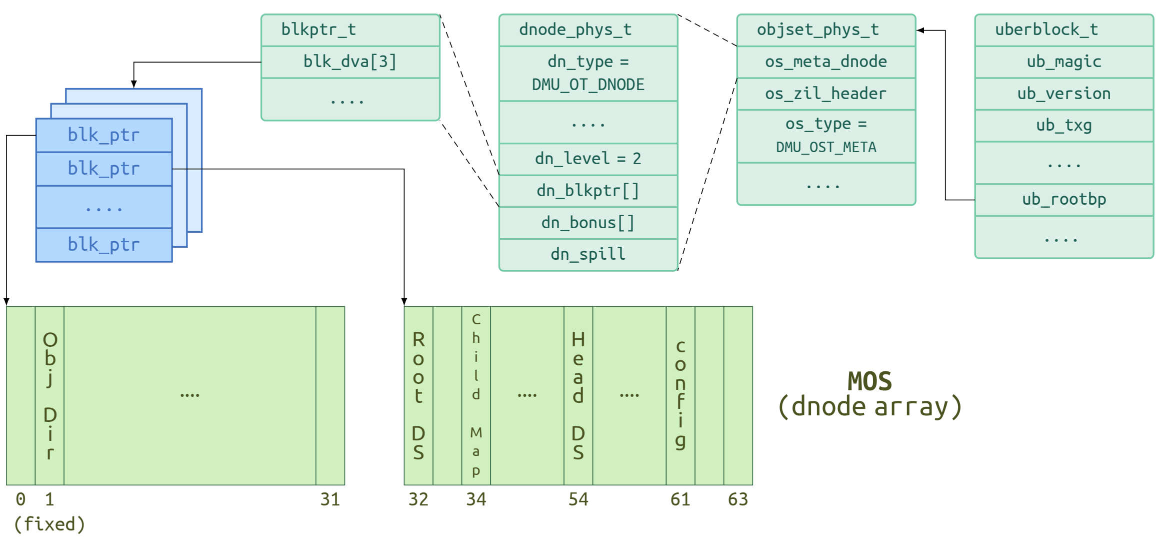

Illustration 1.3: From Uberblock to MOS

1.3 Block Pointer

Data is transferred between disk and main memory in units called blocks. A block pointer (`blkptr_t`) is a 128 byte ZFS structure used to physically locate, verify, and describe blocks of data on disk. The layout of a normal blkptr is shown as Figure. 1.4. Note that, the size of a block is described by three different fields: *psize*, *lsize*, and *asize*.

- *lsize*: Logical size. The size of the data without compression, raidz or gang overhead.
- *psize*: Physical size of the block on disk after compression

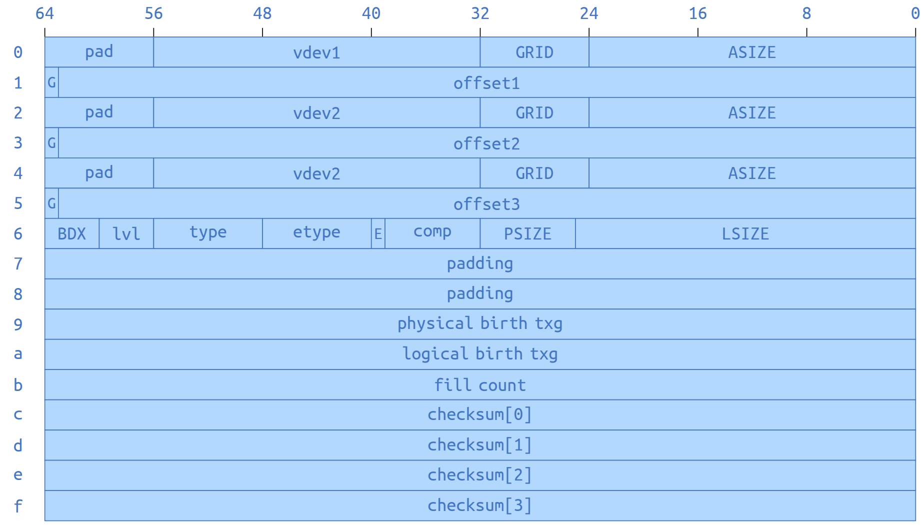

Illustration 1.4: Block Pointer Layout

- *asize*: Allocated size, total size of all blocks allocated to hold this data including any gang headers or raid-Z parity information

Normally, block pointers point (via their DVAs) to a block which holds data. If the data that we need to store is very small, this is an inefficient use of space, Additionally, reading these small blocks tends to generate more random reads. Embedded-data Block Pointers was introduced. It allows small pieces of data (the "payload", upto 112 bytes) embedded in the block pointer, the block pointer doesn't point to anything then. The layout of an embedded block pointer is as Figure. 1.5.

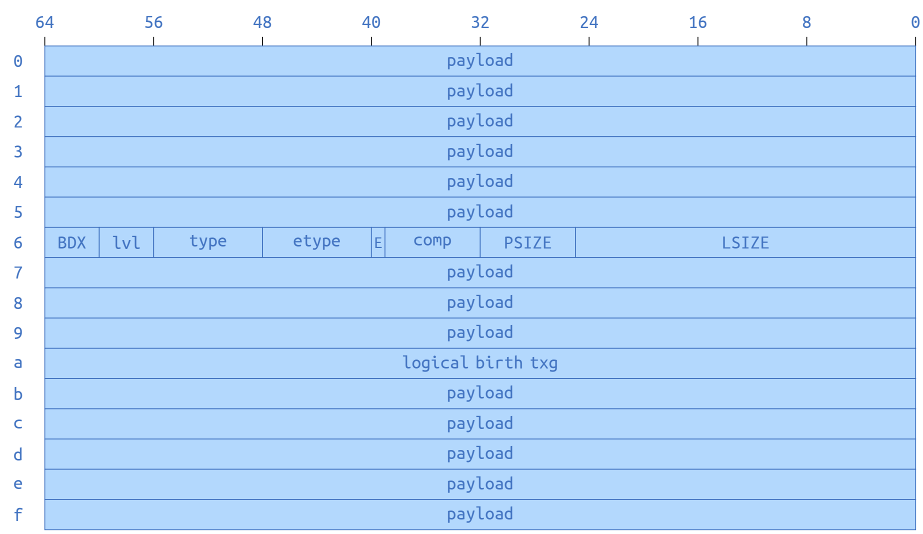

Illustration 1.5: Embedded Block Pointer Layout

1.4 DMU

The *Data Management Unit*, or *DMU* groups blocks into logical units called *objects*. Almost everything in ZFS is an object. Objects are defined by 512 bytes structures called *dnodes* (`dnode_phys_t`). A dnode describes and organizes a collection of blocks making up an object, for example, `dn_type` determines the type of the object, `dn_blkptr` stores the block pointers for block addressing. A dnode has a limited number (`dn_nblkptr`) of block pointers to describe an object's data. For a dnode using the largest data block size (128KB) and containing the maximum number of block pointers (3), the largest object size it can represent is 384 KB. To allow larger objects, indirect blocks are introduced, the largest indirect block (128KB) can hold up to 1024 block pointers, so that 384MB object can be represented without the next level of indirection. The `dn_nlevel` field tells total levels of addressing. Figure. 1.6 illustrates a 3-levels indirect addressing.

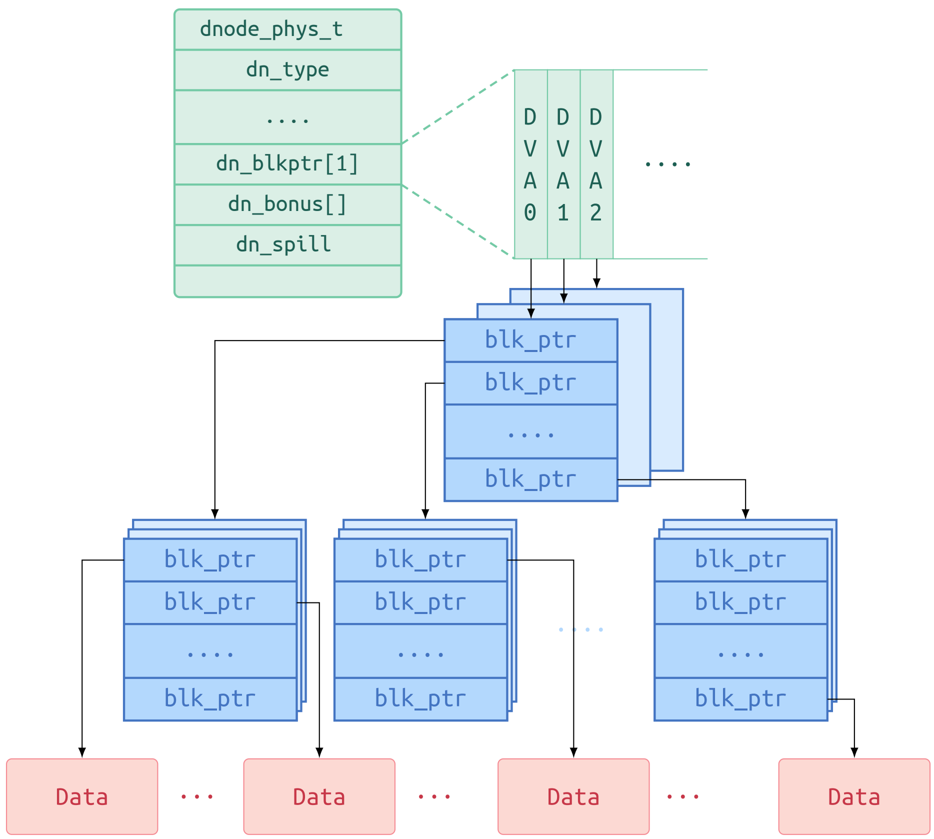

Illustration 1.6: Indirect block addressing

Related objects can be further grouped by the DMU into *object sets*. ZFS allows users to create four kinds of object sets: *filesystems*, *clones*, *snapshots*, and *volumes*.

1.5 DSL

The *Dataset and Snapshot Layer*, or *DSL* is for describing and managing relationships-between and properties-of object sets.

Each object set is represented in the DSL as a *dataset* (`dsl_dataset_phys_t`). Datasets are grouped into collections called *Dataset Dirctories*, which manages a related grouping of datasets and the properties associated with that grouping. A DSL directory always has exactly one *active* dataset. All other datasets under the directory are related to the active dataset through *snapshots*, *clones*, or *child/parent dependencies*.

1.6 MOS

The DSL is implementd as an object set of type `DMU_OST_META`, which is often called the *Meta Object Set*, or *MOS*. There is a single distinguished object in the Meta Object Set, called the *object directory*. Object directory is always located in the 1st element of the dnode array (index starts from 0). All other objects can be located by traversing through a set of object references starting at this object.

The object directory is implemented as a *ZAP* object that is made up of name/value pairs. The object directory contains *root_dataset*, *config*, *free_bpobj*, and some other attribute pairs.

Figure. 1.7 illustrates a realistic layout of the MOS of a sample pool, from which a data set (e.g., file system) can be accessed.

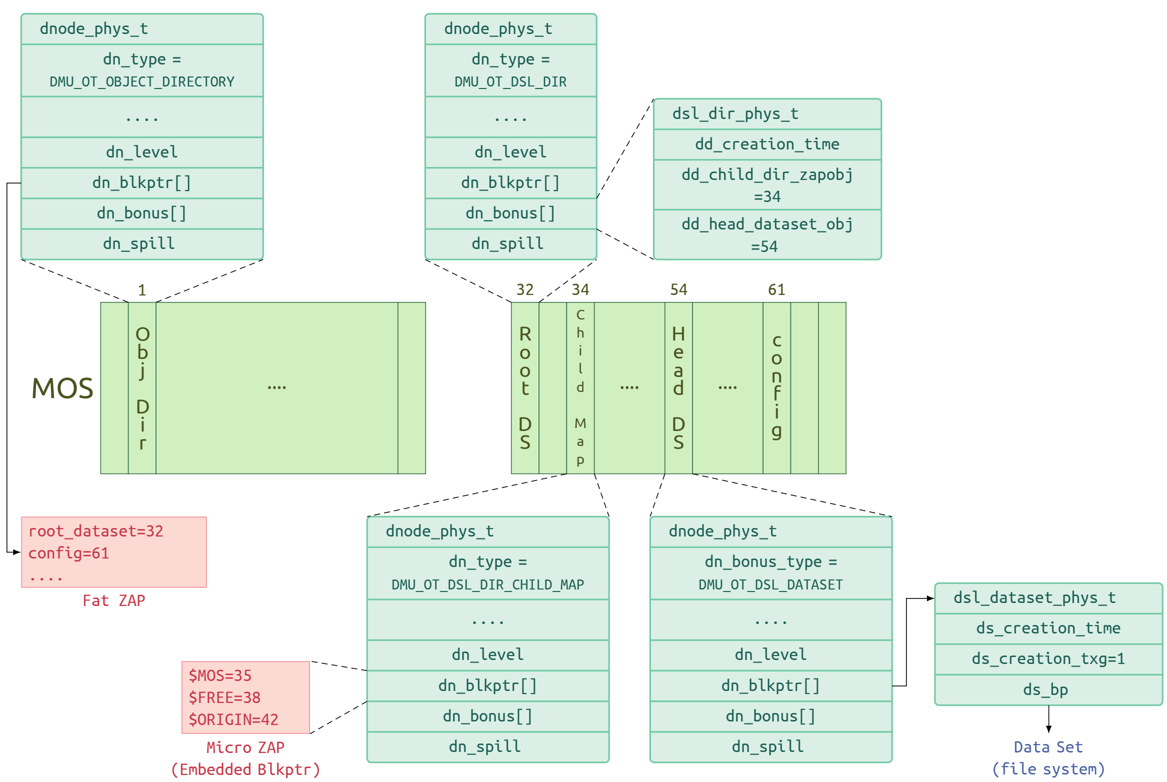

Illustration 1.7: MOS

Figure. 1.8 illustrates the path from a data set object to user data (contents of the sample file `/sbin/zdump`).

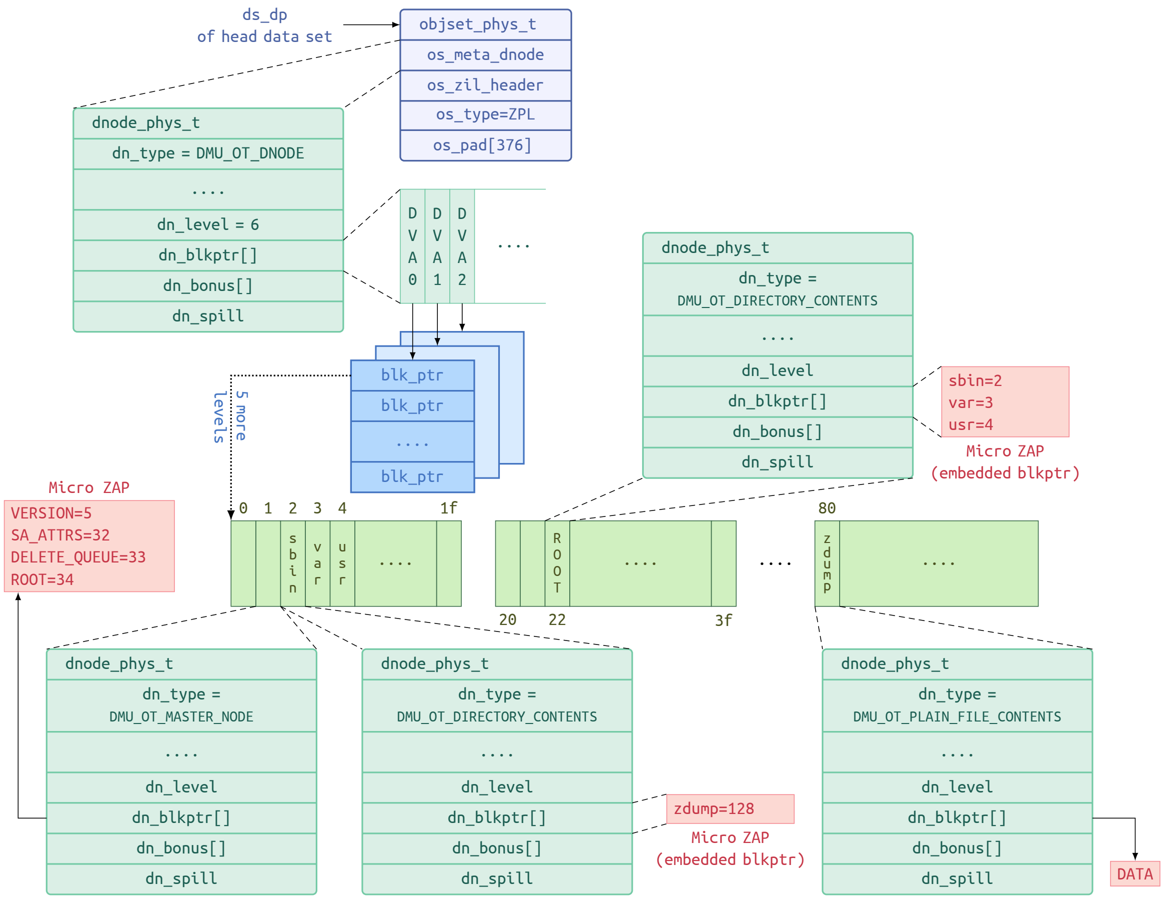

Illustration 1.8: From the data set to user data

1.7 ZAP

The *ZFS Attributes Processors*, or *ZAPs* are objects used to store attributes in the form of name-value pairs. ZAPs come in two forms: *micro ZAP* for small number of attributes and *fat ZAP* for large number of attributes. Both of them are arranged based on hash of the attribute's name. Directories in ZFS are implemented as ZAP objects.

Chapter 2

On-Disk Data Walk (Or: Where's My Data)

This part (title and content) was inspired by Max Bruning's great demostration – ZFS On-Disk Data Walk (Or: Where's My Data) , and even better, his training material. He used the modified[1] `mdb`, `zdb`, and `dd` to read and dump ZFS data structure from physical devices to illustrate the ZFS on-disk layout, from vdev label to content of a file.

There is not `mdb` equivalent on Linux, and I don't want to switch among tools from time to time, so that I wrote a simple tool to do all the things, reading (via `open` and `read` system calls), decompressing (by calling *liblz4* functions) data from the physical vdev, and dumping the ZFS physical data structures as JSON format. It doesn't call any function of the ZFS libraries. A few helpers are still used because I was too lazy to write my own, perhaps I will remove all of them in the future. However the core functions such as `spa_xxxx`, `dmu_xxxx`, `dsl_xxxx`, `zio_xxxx`, are avoided.

2.1 Environment

```
$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.2 LTS (Focal Fossa)"
.... output omitted ....
$ sudo apt install libzpool2linux libzfs2linux libzfslinux-dev
$ sudo apt install libnvpair1linux libjson-c-dev liblz4-dev
```

2.2 Preparation

1. Clone and build `zdump` tool.

2. Create a new file system, with a very simple hierarchy. Note that, as the `zdump` tool only supports lz4, the default compression algorithm of ZFS on Linux, don't set the `compression` property for creating ZFS.

   ```
   $ sudo zfs create -V 4G dpool/zvol0
   $ sudo fdisk -l /dev/zd0 | head -1
   Disk /dev/zd0: 4 GiB, 4294967296 bytes, 8388608 sectors
   $ sudo dmsetup create zdisk0 --table '0 8388607 linear /dev/zd0 0'
   $ sudo mkdir /mnt/zwalk
   ```

[1] by himself 👍

```
$ sudo zpool create -f -m /mnt/zwalk zwalk /dev/mapper/zdisk0
$ sudo zfs list
NAME         USED  AVAIL     REFER  MOUNTPOINT
zwalk        840K  3.62G      192K  /mnt/zwalk
$ sudo mkdir /mnt/zwalk/{sbin,var,usr}
$ sudo su -c 'printf "#!/bin/bash\n\necho Hello ZFS\n" >/mnt/zwalk/sbin/zdump'
$ cat /mnt/zwalk/sbin/zdump
#!/bin/bash

echo Hello ZFS
```

To clean up after the walk:

```
$ sudo zpool destroy zwalk
$ sudo dmsetup remove zdisk0
$ sudo zfs destroy dpool/zvol0
```

3. Add the user into `disk` group so that `sudo` is not needed to read the block device file.

```
$ sudo usermod -aG disk $(whoami)
```

2.3 Walk the data.

1. The first step is dumping the label and active uberblock. Note that the `offset`, `psize`, and `lsize` are hexadecimal.

```
$ zdump --label /dev/mapper/zdisk0:0:40000/40000
{
  "Vdev Label":{
    "name":"zwalk",
    "version":5000,
    "uberblock":{
      "magic":"0x0000000000bab10",
      "version":5000,
      "txg":10,
      "rootbp":{
        "vdev":"0",
        "offset":"3801e000",
        "asize":"2000",
        "psize":"1000",
        "lsize":"1000",
        "compressed":"uncompressed"
      }
    }
  }
}
```

2. Dump dnode of the MOS, using the `offset` (3801e000), `psize` (1000), and `lsize` (1000) we got from the previous step.

```
$ zdump --mos /dev/mapper/zdisk0:"3801e000":1000/1000
{
  "MOS":{
    "os_type":"META",
    "dnonde":{
```

```
      "dn_type":"DMU dnode",
      "dn_bonustype":0,
      "dn_indblkshift":17,
      "dn_nlevels":2,
      "dn_nblkptr":3,
      "dn_blkptr":[
        {
          "vdev":"0",
          "offset":"4e000",
          "asize":"2000",
          "psize":"2000",
          "lsize":"20000",
          "compressed":"lz4"
        },
        .... output omitted ....
      ]
    }
  }
}
```

3. From output of the previous step, we notice that the MOS uses 2-levels indirect addressing (`dn_nlevels` was 2[2]), so we need to find the 0^{th} level block pointer to access to the data block. The `lsize` of each block pointer is 16K, that can contain 32 dnodes.

```
$ zdump --indirect-blkptr /dev/mapper/zdisk0:"4e000":20000/2000:2
{
  "[L0]":[
    {
      "vdev":"0",
      "offset":"28008000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"4000",
      "compressed":"lz4"
    },
    {
      "vdev":"0",
      "offset":"2802c000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"4000",
      "compressed":"lz4"
    },
    {
      "vdev":"0",
      "offset":"0",
      "asize":"0",
      "psize":"200",
      "lsize":"200",
      "compressed":"inherit"
    },
    {
      "vdev":"0",
      "offset":"0",
      "asize":"0",
      "psize":"200",
```

[2]If we create the ZVOL with `-s` option, there will only one level of block pointer.

```
      "lsize":"200",
      "compressed":"inherit"
    },
    {
      "vdev":"0",
      "offset":"28006000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"4000",
      "compressed":"lz4"
    },
    .... output omitted ....
  ]
}
```


4. Dump the MOS object directory, which is the 1^{st} object (the 0^{th} is not used) in the dnode array. The MOS is a fat ZAP object, whose entries will be dumped as well as its dnode. We will use the `root_dataset` object to move forward.

```
$ zdump --mos-objdir /dev/mapper/zdisk0:"28008000":4000/2000
{
  "dnode":{
    "dn_type":"object directory",
    "dn_bonustype":0,
    "dn_indblkshift":17,
    "dn_nlevels":1,
    "dn_nblkptr":3,
    "dn_blkptr":[
      {
        "vdev":"0",
        "offset":"10000",
        "asize":"2000",
        "psize":"2000",
        "lsize":"4000",
        "compressed":"lz4"
      },
      {
        "vdev":"0",
        "offset":"12000",
        "asize":"2000",
        "psize":"2000",
        "lsize":"4000",
        "compressed":"lz4"
      },
      {
        "vdev":"0",
        "offset":"0",
        "asize":"0",
        "psize":"200",
        "lsize":"200",
        "compressed":"inherit"
      }
    ]
  },
  "FZAP":{
    "zap_block_type":"ZBT_HEADER",
    "zap_magic":"0x00000002f52ab2a",
    "zap_num_entries":13,
```

```
    "zap_table_phys":{
      "zt_blk":0
    }
  },
  "FZAP leaf":{
    "entries":[
      {
        "name":"root_dataset",
        "value":32
      },
      .... output omitted ....
      {
        "name":"config",
        "value":61
      },
      .... output omitted ....
    ]
  }
}
```

5. Dump the root data set. From the previous step, we knew that it's the 32^{nd} item in the dnode array, therefore, we seek it in the 1^{st} block (with the offset $2802c000$). `dsl_dir_phys_t` is stored in the `dn_bonus` field of dnode of the root data set object. We can see that the head data set object is the 54^{th} object, located in the same block as the root data set's.

```
$ zdump --mos-rootds /dev/mapper/zdisk0:"2802c000":4000/2000:32
{
  "dnode":{
    "dn_type":"DSL directory",
    "dn_bonustype":12,
    "dn_indblkshift":17,
    "dn_nlevels":1,
    "dn_nblkptr":1,
    "dn_blkptr":[
      {
        "vdev":"0",
        "offset":"0",
        "asize":"0",
        "psize":"200",
        "lsize":"200",
        "compressed":"inherit"
      }
    ]
  },
  "DSL":{
    "dd_creation_time":"....",
    "dd_child_dir_zapobj":34,
    "dd_head_dataset_obj":54
  }
}
```

6. Dump the childmap and head data set. The later will be used to move forward to ZPL.

```
$ zdump --mos-childmap /dev/mapper/zdisk0:"2802c000":4000/2000:34
{
  "Embedded Block Pointer":{
    "type":0,
```

```
    "psize":78,
    "lsize":512,
    "compressed":"lz4",
    "Micro ZAP":[
      {
        "mze_name":"$MOS",
        "mze_value":35
      },
      {
        "mze_name":"$FREE",
        "mze_value":38
      },
      {
        "mze_name":"$ORIGIN",
        "mze_value":42
      }
    ]
  }
}
$ zdump --mos-headds /dev/mapper/zdisk0:"2802c000":4000/2000:54
{
  "Head data set":{
    "ds_dir_obj":32,
    "ds_creation_time":"Tue May 18 08:51:28",
    "ds_create_txg":1,
    "ds_bp":{
      "vdev":"0",
      "offset":"8028000",
      "asize":"2000",
      "psize":"1000",
      "lsize":"1000",
      "compressed":"uncompressed"
    }
  },
  "Object Set":{
    "os_type":"ZPL",
    "dnonde":{
      "dn_type":"DMU dnode",
      "dn_bonustype":0,
      "dn_indblkshift":17,
      "dn_nlevels":6,
      "dn_nblkptr":3,
      "dn_blkptr":[
        {
          "vdev":"0",
          "offset":"8024000",
          "asize":"2000",
          "psize":"2000",
          "lsize":"20000",
          "compressed":"lz4"
        },
        .... output omitted ....
      ]
    }
  }
}
```

7. Note that 6-levels indirect block pointer is used, we need to walk down to the L0 block pointer

first.

```
$ zdump --indirect-blkptr /dev/mapper/zdisk0:"8024000":20000/2000:6
{
  "[L4]":[
    {
      "vdev":"0",
      "offset":"3801c000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"20000",
      "compressed":"lz4"
    },
    .... output omitted ....
  ],
  "[L3]":[
    {
      "vdev":"0",
      "offset":"2802a000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"20000",
      "compressed":"lz4"
    },
    .... output omitted ....
  ],
  "[L2]":[
    {
      "vdev":"0",
      "offset":"8022000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"20000",
      "compressed":"lz4"
    },
    .... output omitted ....
  ],
  "[L1]":[
    {
      "vdev":"0",
      "offset":"4c000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"20000",
      "compressed":"lz4"
    },
    .... output omitted ....
  ],
  "[L0]":[
    {
      "vdev":"0",
      "offset":"46000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"4000",
      "compressed":"lz4"
    },
    {
      "vdev":"0",
```

```
      "offset":"4a000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"4000",
      "compressed":"lz4"
    },
    {
      "vdev":"0",
      "offset":"0",
      "asize":"0",
      "psize":"200",
      "lsize":"200",
      "compressed":"inherit"
    },
    {
      "vdev":"0",
      "offset":"0",
      "asize":"0",
      "psize":"200",
      "lsize":"200",
      "compressed":"inherit"
    },
    {
      "vdev":"0",
      "offset":"48000",
      "asize":"2000",
      "psize":"2000",
      "lsize":"4000",
      "compressed":"lz4"
    },
    .... output omitted ....
  ]
}
```

8. Dump the master node, which is a micro ZAP object and fixed in the 1^{st} dnode in the array.

```
$ zdump --headds-masternode /dev/mapper/zdisk0:"46000":4000/2000
{
  "dnode":{
    "dn_type":"ZFS master node",
    "dn_bonustype":0,
    "dn_indblkshift":17,
    "dn_nlevels":1,
    "dn_nblkptr":3,
    "dn_blkptr":[
      {
        "vdev":"0",
        "offset":"2000",
        "asize":"2000",
        "psize":"200",
        "lsize":"200",
        "compressed":"uncompressed"
      },
      .... output omitted ....
    ]
  },
  "Micro ZAP":[
    .... output omitted ....
```

```
    {
      "mze_name":"ROOT",
      "mze_value":34
    }
  ]
}
```

9. The ROOT object ("/") is the 34^{th} object, located in the 1^{st} block (offset $4a000$). In this simplest case, the ROOT's dnode contains *embedded block pointer*, it is a micro ZAP object.

```
$ zdump --root-dir /dev/mapper/zdisk0:"4a000":4000/2000:34
{
  "Micro ZAP":[
    {
      "mze_name":"sbin",
      "mze_value":2
    },
    {
      "mze_name":"var",
      "mze_value":3
    },
    {
      "mze_name":"usr",
      "mze_value":4
    }
  ]
}
```

10. Dump the `sbin` directory, the 2^{nd} object in the 0^{th} block, from the output of indirect block pointer dumping, we knew that the block is located at 46000, the object contains embedded block pointer again.

```
$ zdump --dir /dev/mapper/zdisk0:"46000":4000/2000:2
{
  "Micro ZAP":[
    {
      "mze_name":"zdump",
      "mze_value":128
    },
    {
      "mze_name":"",
      "mze_value":0
    },
    {
      "mze_name":"",
      "mze_value":0
    }
  ]
}
```

11. The `/sbin/zdump` file is the128^{th} object, located in the 4^{th} block (offset 48000), let's dump it.

```
$ zdump --text /dev/mapper/zdisk0:"48000":4000/2000:128
#!/bin/bash

echo Hello ZFS
```

Chapter 3

Recovering removed file

TBD