OCFS2 Support Guide - On-Disk Format

This is the third in the series of support guides for OCFS2. In this we will walk the file system layout on disk using the tool, debugfs.ocfs2.

debugfs.ocfs2, or debugfs for short, should not be confused with debugfs, the kernel module. While both are used in file system debugging, debugfs, the tool, reads the on-disk structures, while debugfs, the kernel module, assists the filesystem to relay debugging information from the kernelspace to the userspace. In this writeup, our discussion is limited to the userspace debugfs.ocfs2 tool.

To start debugfs.ocfs2, do:

```
# debugfs.ocfs2 /dev/sdg1
debugfs.ocfs2 1.2.1
debugfs:
```

debugfs, by default, is an interactive tool. To run scripts, refer to the debugfs.ocfs2(8) manpage.

Super Block

We start with the super block. That's the block that contains the block and cluster sizes of the filesystem. The information is stored in number of bits. At 12 bits, the block size is 4K (2^12) and at 15 bits, the cluster size is 32K (2^15). The other information gleaned is the block numbers for the root and the system directories.

```
debugfs: stats -h
        Revision: 0.90
        Mount Count: 0   Max Mount Count: 20
        State: 0   Errors: 0
        Check Interval: 0   Last Check: Thu Mar  9 16:01:36 2006
        Creator OS: 0
        Feature Compat: 0   Incompat: 0   RO Compat: 0
        Root Blknum: 17   System Dir Blknum: 18
        First Cluster Group Blknum: 8
        Block Size Bits: 12   Cluster Size Bits: 15
        Max Node Slots: 4
        Label: sunil060309a
        UUID: 1639290507B347F683B06A5FFCB531D7
```

OCFS2 views the disk in terms of blocks. All on-disk structures occupy at least 1 block. In other words, no two structures can share a block. This is important as the filesystem identifies on-disk resources across the cluster by its block number. Also, as the block size can range from 512 bytes to 4K, the structures are designed to make full use of the entire block. We will explore this further when we look at an inode.

System Directory

System directory is like any directory. It has files and sub-directories. Only difference is that it is hidden and can only be browsed using this tool. The system

directory holds the system files which manages the filesystem metadata.

To list all entries in the system directory, do:

```
debugfs: ls -l //
18   drwxr-xr-x   6  0  0        4096 9-Mar-2006 16:01 .
18   drwxr-xr-x   6  0  0        4096 9-Mar-2006 16:01 ..
19   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 bad_blocks
20   -rw-r--r--   1  0  0      851968 9-Mar-2006 16:01 global_inode_alloc
21   -rw-r--r--   1  0  0       32768 9-Mar-2006 16:01 slot_map
22   -rw-r--r--   1  0  0     1048576 9-Mar-2006 16:01 heartbeat
23   -rw-r--r--   1  0  0  2000650240 9-Mar-2006 16:01 global_bitmap
24   drwxr-xr-x   2  0  0        4096 9-Mar-2006 16:01 orphan_dir:0000
25   drwxr-xr-x   2  0  0       12288 30-Mar-2006 18:15 orphan_dir:0001
26   drwxr-xr-x   2  0  0       12288 30-Mar-2006 18:15 orphan_dir:0002
27   drwxr-xr-x   2  0  0        4096 9-Mar-2006 16:01 orphan_dir:0003
28   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 extent_alloc:0000
29   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 extent_alloc:0001
30   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 extent_alloc:0002
31   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 extent_alloc:0003
32   -rw-r--r--   1  0  0   100663296 9-Mar-2006 16:01 inode_alloc:0000
33   -rw-r--r--   1  0  0    20971520 9-Mar-2006 16:01 inode_alloc:0001
34   -rw-r--r--   1  0  0    83886080 9-Mar-2006 16:01 inode_alloc:0002
35   -rw-r--r--   1  0  0   104857600 9-Mar-2006 16:01 inode_alloc:0003
36   -rw-r--r--   1  0  0    67108864 9-Mar-2006 16:01 journal:0000
37   -rw-r--r--   1  0  0    67108864 9-Mar-2006 16:01 journal:0001
38   -rw-r--r--   1  0  0    67108864 9-Mar-2006 16:01 journal:0002
39   -rw-r--r--   1  0  0    67108864 9-Mar-2006 16:01 journal:0003
40   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 local_alloc:0000
41   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 local_alloc:0001
42   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 local_alloc:0002
43   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 local_alloc:0003
44   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 truncate_log:0000
45   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 truncate_log:0001
46   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 truncate_log:0002
47   -rw-r--r--   1  0  0           0 9-Mar-2006 16:01 truncate_log:0003
```

In debugfs, we use the "//" notation to denote the system directory.

The listing format above is similar to "ls -il" in shell. The first column lists the block number. Notice the file names. The ones with trailing numbers are local system files and are for each slot. Ones without the numbers are referred to as global system files. More on this later.

Root Directory

To view the root dir, do:

```
debugfs: ls -l /
      17      drwxr-xr-x   8  0  0    4096 11-May-2006 15:02 .
      17      drwxr-xr-x   8  0  0    4096 11-May-2006 15:02 ..
      158378  drwxrwxrwx  19  0  0    4096  5-Mar-2006 11:07 linux-2.6.15.6
      227650  drwxr-xr-x   2  0  0    4096 15-Mar-2006 13:16 tmp
      66073   -rw-r--r--   1  0  0  688128 11-May-2006 15:06 file1
      278929  -rw-r--r--   1  0  0  688128 11-May-2006 15:06 file2
```

```
387082  drwxr-xr-x   3  0  0    4096 15-Mar-2006 12:09 ca-test92
160065  drwxr-xr-x   3  0  0    4096 15-Mar-2006 12:09 ca-test95
158018  drwxr-xr-x   3  0  0    4096 15-Mar-2006 12:09 ca-test96
383523  drwxr-xr-x   3  0  0    4096 15-Mar-2006 12:09 ca-test89
```

Like in listing the system directory, the first column lists the block number of that object. The block number is also used as the inode number in the fs. The remaining columns in order are permissions, links count, user id, group id, size, modify date/time and object name.

Inode

To view an inode, do:

```
debugfs: stat /linux-2.6.15.6/README
        Inode: 267577   Mode: 0644   Generation: 1394565908 (0x531f6314)
        FS Generation: 3764567147 (0xe062bc6b)
        Type: Regular   Flags: Valid
        User: 0 (root)   Group: 0 (root)   Size: 15070
        Links: 1   Clusters: 1
        ctime: 0x442ddf68 -- Fri Mar 31 18:03:20 2006
        atime: 0x442ddf66 -- Fri Mar 31 18:03:18 2006
        mtime: 0x440b370a -- Sun Mar  5 11:07:54 2006
        dtime: 0x0 -- Wed Dec 31 16:00:00 1969
        ctime_nsec: 0x32204592 -- 840975762
        atime_nsec: 0x00000000 -- 0
        mtime_nsec: 0x00000000 -- 0
        Last Extblk: 0
        Sub Alloc Slot: 2   Sub Alloc Bit: 233
        Tree Depth: 0   Count: 243   Next Free Rec: 1
        ## Offset        Clusters       Block#
        0  0             1              272392
```

One can also view the inode by just the inode/block# like, "stat <267577>". Notice the "Tree Depth". 0 means that the inode is directly pointing to the data blocks. In this example, the inode is pointing to a cluster starting at block 272392. To extract the file using dd, do:

```
# dd if=/dev/sdg1 of=/tmp/readme bs=4096 skip=272392 count=$[$[15070/4096]+1]
# head -c 15070 /tmp/readme >/tmp/README
```

This file has only one extent. Now let's view the inode of a file with more extents.

```
debugfs: stat /file1
        Inode: 66073   Mode: 0644   Generation: 1728531012 (0x67074a44)
        FS Generation: 3764567147 (0xe062bc6b)
        Type: Regular   Flags: Valid
        User: 0 (root)   Group: 0 (root)   Size: 688128
        Links: 1   Clusters: 21
        ctime: 0x4463b56b -- Thu May 11 15:06:35 2006
        atime: 0x4463b4c0 -- Thu May 11 15:03:44 2006
        mtime: 0x4463b56b -- Thu May 11 15:06:35 2006
        dtime: 0x0 -- Wed Dec 31 16:00:00 1969
        ctime_nsec: 0x2417c7e3 -- 605538275
```

```
        atime_nsec: 0x2a6d6e9c -- 711814812
        mtime_nsec: 0x2417c7e3 -- 605538275
        Last Extblk: 0
        Sub Alloc Slot: 0   Sub Alloc Bit: 1
        Tree Depth: 0   Count: 243   Next Free Rec: 20
        ## Offset        Clusters       Block#
        0  0             2              78384
        1  2             1              78408
        2  3             1              78424
        3  4             1              78440
        4  5             1              78456
        5  6             1              78472
        6  7             1              78488
        7  8             1              78504
        8  9             1              78520
        9  10            1              78536
        10 11            1              78552
        11 12            1              78568
        12 13            1              78584
        13 14            1              78600
        14 15            1              78616
        15 16            1              78632
        16 17            1              78648
        17 18            1              78664
        18 19            1              78680
        19 20            1              78696
```

Notice the "Count" in the inode. That's the number of direct extents this inode can point to. With 4K block sized inodes, one can have 243 direct extents. The number reduces for smaller block sizes.

After extending the file such that it grows to 600 extents, the inode looks like:

```
    debugfs: stat /file1
        Inode: 66073   Mode: 0644   Generation: 1728531012 (0x67074a44)
        FS Generation: 3764567147 (0xe062bc6b)
        Type: Regular   Flags: Valid
        User: 0 (root)   Group: 0 (root)   Size: 19693568
        Links: 1   Clusters: 601
        ctime: 0x4463b772 -- Thu May 11 15:15:14 2006
        atime: 0x4463b4c0 -- Thu May 11 15:03:44 2006
        mtime: 0x4463b772 -- Thu May 11 15:15:14 2006
        dtime: 0x0 -- Wed Dec 31 16:00:00 1969
        ctime_nsec: 0x30179516 -- 806851862
        atime_nsec: 0x2a6d6e9c -- 711814812
        mtime_nsec: 0x30179516 -- 806851862
        Last Extblk: 129574
        Sub Alloc Slot: 0   Sub Alloc Bit: 1
        Tree Depth: 1   Count: 243   Next Free Rec: 3
        ## Offset        Clusters       Block#
        0  0             253            129570
        1  253           252            129572
        2  505           96             129574
        SubAlloc Bit: 2   SubAlloc Slot: 0
        Blknum: 129570   Next Leaf: 129572
        Tree Depth: 0   Count: 252   Next Free Rec: 252
```

```
## Offset        Clusters       Block#
0  0             2              78384
1  2             1              78408
2  3             1              78424
....
250 251          1              130712
251 252          1              130728
SubAlloc Bit: 4   SubAlloc Slot: 0
Blknum: 129572   Next Leaf: 129574
Tree Depth: 0   Count: 252   Next Free Rec: 252
## Offset        Clusters       Block#
0  253           1              130744
1  254           1              130760
2  255           1              130776
....
250 503          1              471272
251 504          1              471288
SubAlloc Bit: 6   SubAlloc Slot: 0
Blknum: 129574   Next Leaf: 0
Tree Depth: 0   Count: 252   Next Free Rec: 96
## Offset        Clusters       Block#
0  505           1              471304
1  506           1              471320
...
93 598           1              114104
94 599           1              114120
95 600           1              114136
```

The output has been compressed for readability. Here we see the inode at "Tree Depth" 1. Means it is pointing to extent blocks which are pointing to the actual data blocks. This gets us to the "Extent" block type. Theoretically, the tree depth could grow without bound, allowing one to have unlimited number of extents. However, for performance reasons, that is not desired.

Now with that many extents, if one were to dd the file, one would have to figure out all the starting block numbers and length for each extent in order. Instead, one can also do:

```
debugfs: dump /file1 /tmp/file1
```

Please note that all the ios are performed o_direct. If the filesystem is mounted, while debugfs will work, the results may be not what one expects. The "dump" command has been provided mainly to extract information from a corrupted / unmountable filesystem. To recursively dump an entire directory tree, use "rdump".

Journal File

Like ext3, OCFS2 uses JBD for journaling. The journaling is block-based, as in, it keeps track of the latest image of the block rather than the change vectors. Also the transactions can only be rolled forward and not backwards. If a system crashes, the next time the volume is mounted, the filesystem walks thru the journal file and rolls forward all committed transactions. fsck is also capable of doing the same.

To illustrate this process, let us review a dump of a dirty journal file.

To dump a journal for a particular slot, do:

```
# echo "logdump 0" | debugfs.ocfs2 -n /dev/sdX >/tmp/log0
```

While one can view the logdump interactively, it is easier to view it in your favorite editor.

The first block, the journal super block, looks as follows:

```
Block 0: Journal Superblock
Seq: 0   Type: 4 (JFS_SUPERBLOCK_V2)
Blocksize: 4096   Total Blocks: 16384   First Block: 1
First Commit ID: 4553   Start Log Blknum: 85
Error: 0
Features Compat: 0x0   Incompat: 0x0   RO Compat: 0x0
Journal UUID: 1639290507B347F683B06A5FFCB531D7
FS Share Cnt: 1   Dynamic Superblk Blknum: 0
Per Txn Block Limit    Journal: 0    Data: 0
```

Notice the "First Commit ID" and the "Start Log Blknum". When the journal is replayed, it will look for "Seq: 4553" starting at "Block: 85" in that journal file.

```
Block 85: Journal Descriptor
Seq: 4553   Type: 1 (JFS_DESCRIPTOR_BLOCK)
No. Blocknum        Flags
 0. 278929          none
UUID: 00000000000000000000000000000000
 1. 40              JFS_FLAG_SAME_UUID
 2. 66073           JFS_FLAG_SAME_UUID
 3. 258048          JFS_FLAG_SAME_UUID
 4. 23              JFS_FLAG_SAME_UUID
 5. 8               JFS_FLAG_SAME_UUID
 6. 129568          JFS_FLAG_SAME_UUID
 7. 28              JFS_FLAG_SAME_UUID
 8. 129569          JFS_FLAG_SAME_UUID
 9. 129570          JFS_FLAG_SAME_UUID JFS_FLAG_LAST_TAG
```

The journal descriptor keeps track of all the blocks in that transaction. The JFS_FLAG_LAST_TAG indicates the last block. The blocks listed will follow the descriptor in order, followed by the commit block.

```
Block 96: Journal Commit Block
Seq: 4553   Type: 2 (JFS_COMMIT_BLOCK)
```

If all of the said blocks exist, the transaction will be rolled forward. If not, it will be ignored. The process is repeated for the next commit id until no more are found. Once all the transactions are rolled forward, the journal super block is initialized and the dirty flag on the journal inode is reset.

Slot Map

This holds the mapping between the slot and the node numbers. The slot# denotes the set of local system files in use by that node. The slot to node mappings can change on each mount. To view the mappings, do:

```
debugfs: slotmap
      Slot#   Node#
          0      92
          1      89
          2      96
          3      95
```

Heartbeat

This points to the area on disk used in disk hearbeating. This file needs to be contiguous and is created as such in mkfs. To view this file, do:

```
debugfs: hb
        node: node              seq       generation checksum
          89:   89 00000000445e7660 0000000000000000 38bddb52
          92:   92 0000000044483342 0000000000000000 411bcad4
          95:   95 00000000445e7430 0000000000000000 81e06d2e
          96:   96 00000000445e763c 0000000000000000 12fbac31
```

When a node mounts a volume, the o2hb kernel thread starts heartbeating in the heartbeat file in the same block number as its node number. Please do not confuse the slot number with the node number The node always heartbeats based on its node number as specified in cluster.conf.

Orphan Directory

When an object is deleted, the fs unlinks the directory entry from the existing directory and links it in that node's orphan_dir. When that object is no longer in use across the cluster, the fs frees the inode and its associated space. Use "ls -l //orphan_dir:0000" to view the contents of the directory.

Local Alloc

On mount, the fs allocates a chunk of bits to that node's local alloc bitmap to be used for small allocations. This improves performace as the node does not need to take cluster lock on the global bitmap for each small allocation. The local alloc bitmap is replenished as required. On umount, the unused bits are flushed back to the global bitmap.

To view the local_alloc for a slot, do:

```
debugfs: stat //local_alloc:0000
      Inode: 40   Mode: 0644   Generation: 3764567147 (0xe062bc6b)
      FS Generation: 3764567147 (0xe062bc6b)
      Type: Regular   Flags: Valid System Localalloc Allocbitmap
      User: 0 (root)   Group: 0 (root)   Size: 0
      Links: 1   Clusters: 0
      ctime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
      atime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
      mtime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
      dtime: 0x0 -- Wed Dec 31 16:00:00 1969
      ctime_nsec: 0x00000000 -- 0
```

```
        atime_nsec: 0x00000000 -- 0
        mtime_nsec: 0x00000000 -- 0
        Last Extblk: 0
        Sub Alloc Slot: Global   Sub Alloc Bit: 24
        Bitmap Total: 256   Used: 204   Free: 52
        Local Bitmap Offset: 38728   Size: 3888
```

Truncate Log

Truncate logs help to improve the delete performance. This system file allows the fs to collect freed bits and flush it to the global bitmap in chunks. What that also means is that space could be temporarily "lost" from the fs. As in, the space freed by deleting a large file may not show up immediately. One can view the orphan_dirs and the truncate_logs to account for such "lost" space.

To view a truncate log, do:

```
    debugfs: stat //truncate_log:0000
        Inode: 44   Mode: 0644   Generation: 3764567147 (0xe062bc6b)
        FS Generation: 3764567147 (0xe062bc6b)
        Type: Regular   Flags: Valid System Dealloc
        User: 0 (root)   Group: 0 (root)   Size: 0
        Links: 1   Clusters: 0
        ctime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
        atime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
        mtime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
        dtime: 0x0 -- Wed Dec 31 16:00:00 1969
        ctime_nsec: 0x00000000 -- 0
        atime_nsec: 0x00000000 -- 0
        mtime_nsec: 0x00000000 -- 0
        Last Extblk: 0
        Sub Alloc Slot: Global   Sub Alloc Bit: 28
        Total Records: 487   Used: 0
        ##   Start Cluster   Num Clusters
        0    50              2
        1    2394            5
```

The truncate_log keeps records of start cluster# and number of clusters. The max number of such records "Total Records" depend on the block size. The above is for a 4K block size.

Global Bitmap

mkfs divides the volume in cluster groups. Barring the last group, all cluster groups have the same number of clusters. The first block of each group contains the bitmap for that the group. The global bitmap can directly point to a fixed max number of groups. When the number of groups exceed that max, the newer groups are chained to the older groups. This allows the filesystem to have an unlimited number of cluster groups.

```
    debugfs: stat //global_bitmap
        Inode: 23   Mode: 0644   Generation: 3764567147 (0xe062bc6b)
        FS Generation: 3764567147 (0xe062bc6b)
        Type: Regular   Flags: Valid System Allocbitmap Chain
```

```
User: 0 (root)   Group: 0 (root)   Size: 2000650240
Links: 1   Clusters: 61055
ctime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
atime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
mtime: 0x4410c1e0 -- Thu Mar  9 16:01:36 2006
dtime: 0x0 -- Wed Dec 31 16:00:00 1969
ctime_nsec: 0x00000000 -- 0
atime_nsec: 0x00000000 -- 0
mtime_nsec: 0x00000000 -- 0
Last Extblk: 0
Sub Alloc Slot: Global   Sub Alloc Bit: 7
Bitmap Total: 61055   Used: 35166   Free: 25889
Clusters per Group: 32256   Bits per Cluster: 1
Count: 243   Next Free Rec: 2
##   Total        Free         Used         Block#
0    32256        10788        21468        8
1    28799        15101        13698        258048

Group Chain: 0   Parent Inode: 23  Generation: 3764567147
##   Block#            Total    Free     Used     Size
0    8                 32256    10788    21468    4032

Group Chain: 1   Parent Inode: 23  Generation: 3764567147
##   Block#            Total    Free     Used     Size
0    258048            28799    15101    13698    4032
```

In the above example, the global bitmap has only 2 block groups. For it to chain groups, it would need to have more than 243 block groups. The fs maintains usage summary at the global and at the block group. Also, as each bitmap is no larger than a block, the bitmap operation to locate free contiguous bits is fast.

Inode and Extent Alloc

Unlike ext3, OCFS2 does not preallocate space for inodes. This space is dynamically allocated. As the global bitmap allocates space in cluster sized chunks, we use a sub-allocator to break up larger allocations in block-sized chunks. Inode and Extent allocs are used to allocate the inode and extent blocks respectively. Like the global bitmap, both these sub-allocators also maintain space as same sized cluster groups. The difference being, these sub-allocators grow with usage. The global bitmap, meanwhile, can only grow when the volume is explicitly resized.

Global Inode Alloc

This is a special sub-allocator for inodes and in only used by user space tools like tunefs and fsck. When one uses tunefs to increase the number of slots, it has to create inodes for local system files for each new slot. Global Inode Alloc is used as the sub-allocator for these new inodes.