ORACLE CLUSTER FILE SYSTEM

Physical Design & Implementation

Written by

Srinivas Eeda

Preface

OCFS V1 is designed and developed by Neeraj Goyal, Suchit Kaura, Kurt Hackel, Sunil Mushran, Manish Singh, and Wim Coekaerts.

The main goal of OCFS project was to provide a replacement for raw devices. It was/is not designed as a general-purpose cluster filesystem. It was designed to store database files, with database files I mean "datafiles, redologfiles, archive logfiles, controlfiles, quorum disk file, spfile".

Using the filesystem for any other types of files is not well tested, since that was not part of the original goal. Also, this is not a filesystem that you can compare with others, like ufs or vxfs or ext2/3 etc. Things like updates of ctime/utime/mtimes are not necessarily be the same. We might have a different way of showing inode numbers; when a file moves it might end up with different sorts of timestamp updates. Why ? because the database doesn't care, OCFS do as little as possible to keep performance and as much as possible to keep Oracle RDBMS happy. Anything more is nice, but not considered a requirement.

DOCUMENT PREVIEW

This document is for folks who are interested in understanding OCFS internals. It wasn't planned to actually write a design document, but I did this out of my interest to help folks who are interested in understanding how OCFS works. I have put this information in writing while I was walking through the code (ocfs version 1.0.9-9) and understanding how OCFS is implemented.

`OCFS PHYSICAL DESIGN OVERVIEW` explains the physical structures that reside on the disk. OCFS physical structures are crusial in OCFS implementation. They contain the persistant information about the OCFS data structures, metadata and the user data. It also contains structures that are used during runtime for Inter node communication and co-ordination for accessing the shared resources.

`OCFS Functional Design` talks about how the OCFS filesystem works. It talks about how the nodes communicate and co-ordinate access to shared resources. It talks about the various components that are implemented to implement a cluster aware filesystem.

`OCFS Implementation` talks about the actual code and how OCFS is implemented. It talks about the main routines to give an idea of how OCFS works. For complete information on implementation, one needs to look at the code.

`OCFS Data Structures` talks about the implementation of disk structures and in-memory structures.

TABLE OF CONTENTS

1. OCFS PHYSICAL DESIGN OVERVIEW 7
1.1 Disk Layout 7
1.1.1 OCFS Header 8
1.1.1.1 OCFS Super Block 9
1.1.1.2 OCFS Node Config Sectors 9
1.1.1.2.1 Node Config Header 10
1.1.1.2.2 Node Config Info Sectors 10
1.1.1.2.3 New Node Config Info Sector 11
1.1.1.2.4 Node Config Header 11
1.1.1.3 Publish Sectors 12
1.1.1.4 Vote Sectors 13
1.1.1.5 Global Bitmap 13
1.1.2 Data Blocks 14
1.1.2.1 System File Headers 14
1.1.2.1.1 DirFile 14
1.1.2.1.2 DirMapFile 15
1.1.2.1.3 ExtentFile 15
1.1.2.1.4 ExtentMapFile 15
1.1.2.1.5 RecoverLogFile 15
1.1.2.1.6 CleanUpLogFile 16
1.1.2.1 System File Data 16
1.1.2.2 DirNode 16
1.1.2.3 File Entries 17
1.1.2.4 Extent group 18
1.1.2.5 Data blocks 18
1.2 Memory Structures 19
1.2.1 OCFS Global Structures 19
1.2.1.1 Global Context 20
1.2.1.2 ocfs_ipc_ctxt 20
1.2.1.3 OIN Cache 20
1.2.1.4 OFILE Cache 21
1.2.1.5 FileEntry Cache 21
1.2.1.6 LockRes cache 21
1.2.2 OCFS Volume Structures 21
1.2.2.1 Ocfs_super 22
1.2.2.2 ocfs_vol_layout 22
1.2.2.3 ocfs_inode 22
1.2.2.4 ocfs_file 22
1.2.2.5 ocfs_lock_res 23
1.2.2.6 ocfs_io_runs 23
1.2.2.7 ocfs_vol_node_map 23
2 OCFS Functional Design & Implementation 24

2.1 Node Monitoring 24
2.2 Distributed Lock Manager 24
2.3 OCFS Journaling and Recovery 25
2.3.1 Log Records 26
2.3.1.1 LOG_TYPE_DISK_ALLOC 26
2.3.1.2 LOG_CLEANUP_LOCK 26
2.3.1.3 LOG_TYPE_RECOVERY 27
2.3.1.4 LOG_FREE_BITMAP 27
2.3.1.5 LOG_UPDATE_EXTENT 28
2.3.1.6 LOG_DELETE_ENTRY 28
2.3.1.7 LOG_MARK_DELETE_ENTRY 28
2.3.1.8 LOG_DELETE_NEW_ENTRY 29
2.3.1.9 LOG_TYPE_DIR_NODE 29
2.4 OCFS Processes 29
2.4.1 OCFS Client Processes 31
2.4.2 Node Monitoring Thread 31
2.4.3 OCFS Listener Thread 32
2.4.4 Kernel Task Queues 32
3. OCFS Implementation 33
3.1 OCFS Node Monitoring Thread 33
3.1.1 ocfs_volume_thread 33
3.2 OCFS Listener Thread 34
3.2.1 ocfs_recv_thread 34
3.3 OCFS DLM Operations 35
3.3.1 ocfs_acquire_lock 35
3.3.2 ocfs_break_cache_lock 35
3.3.3 ocfs_break_cache_lock 36
3.3.4 ocfs_acquire_lockres 37
3.3.5 ocfs_release_lock 37
3.3.6 ocfs_release_lockres 37
3.4 OCFS Journaling & Recovery 37
3.4.1 ocfs_process_record 37
3.4.2 ocfs_recover_vol 38
3.4.3 ocfs_write_log 38
3.4.4 ocfs_start_trans 38
3.4.5 ocfs_commit_trans 39
3.4.6 ocfs_abort_trans 39
3.4.7 ocfs_process_log 39
3.4.8 ocfs_reset_publish 39
3.4.9 ocfs_get_system_file_size 40
3.4.10 ocfs_extend_system_file 40
3.5 OCFS Module Operations 40
3.5.1 ocfs_driver_entry 40
3.5.2 ocfs_driver_exit 40
3.6 OCFS Super Operations 40
3.6.1 ocfs_read_super 40

3.6.2 ocfs_statfs ... 41
3.6.3 ocfs_put_inode ... 42
3.6.4 ocfs_clear_inode ... 42
3.6.5 ocfs_read_inode2 ... 42
3.6.6 ocfs_put_super ... 42
3.7 OCFS File Operations ... 42
3.7.1 ocfs_file_read ... 42
3.7.2 ocfs_file_write ... 42
3.7.3 ocfs_sync_file ... 43
3.7.4 ocfs_flush ... 43
3.7.5 ocfs_file_release ... 43
3.7.6 ocfs_file_open ... 43
3.8 OCFS Directory Operations ... 43
3.8.1 ocfs_readdir ... 43
3.9 OCFS Inode Operations ... 43
3.9.1 ocfs_create ... 43
3.9.2 ocfs_lookup ... 43
3.9.3 ocfs_mkdir ... 44
3.9.4 ocfs_mknod ... 44
3.9.5 ocfs_rename ... 44
3.9.6 ocfs_setattr ... 44
3.9.7 ocfs_getattr ... 44
3.10 OCFS Address Space Operations ... 44
3.10.1 ocfs_readpage ... 44
3.10.2 ocfs_writepage ... 45
3.10.3 ocfs_prepare_write ... 45
3.10.4 ocfs_commit_write ... 45
4. OCFS DATA STRUCTURES ... 46
4.1 ocfs_vol_disk_hdr ... 46
4.2 ocfs_vol_label ... 47
4.3 ocfs_disk_lock ... 48
4.4 ocfs_bitmap_lock ... 48
4.5 ocfs_node_config_hdr ... 49
4.6 ocfs_disk_node_config_info ... 49
4.7 default port id ... 50
4.8 ocfs_ipc_config_info ... 50
4.9 ocfs_publish ... 50
4.9.1 Vote ... 51
4.9.2 Vote_types ... 51
4.10 ocfs_vote ... 52
4.11 ocfs_dir_node ... 52
4.12 dir_node_flags ??? ... 53
4.13 ocfs_file_entry ... 54
4.14 ocfs_extent_group ... 55
4.15 Ocfs_global_ctxt ... 56
4.16 Ocfs_super ... 57

4.17 ocfs_vol_state 60
4.18 ocfs_vol_layout 61
4.19 ocfs_inode 62
4.20 Structures to define entities under /proc filesystem 63
4.21 System Files 64

1. OCFS PHYSICAL DESIGN OVERVIEW

OCFS is a cluster aware filesystem designed as a replacement for raw devices. It is mainly designed to store Oracle database files, like datafiles, redologfiles, archive logfiles, controlfiles, quorum disk file, spfile. It is not designed for general-purpose files, and hence doesn't guarantee synchronized access to data. In Version 1 it only guarantee synchronized access to metadata, and limits the maximum number of nodes that can mount the filesystem to 32.

The physical design of OCFS can be further categorized as follows

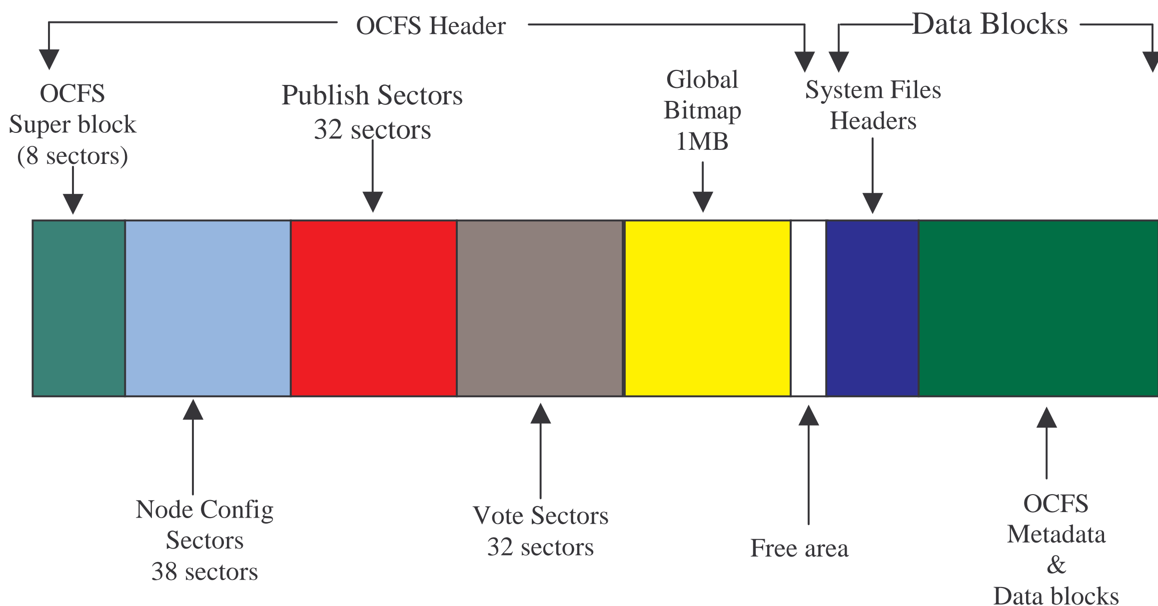
1. Disk Layout
2. In Memory Structures

OCFS divides the disk into various divisions to handle, node management and node communications. It also maintains In Memory structures to serve its functionality

1.1 Disk Layout

The below diagram depicts the disk layout of the device that is formatted to run OCFS filesystem. This layout structure is maintained per volume and hence each layout is independent of others; for e.g., if a node has more than one device/disk that has been formatted to run OCFS filesystem, then each device has this layout and is independent of the other. OCFS divides the disk into two divisions:

1. OCFS Header
2. Data Blocks



Disk Layout of OCFS

OCFS Header is of fixed size and gets allocated and initialized during the format of the device. The OCFS Header is further subdivided into various different types of blocks of fixed sizes. These blocks are further subdivided into sectors and a sector is of size 512 bytes. These sectors hold OCFS data structures that are required for node monitoring and inter node communications.

Most of the OCFS data structures are of size close to a sector size and hence each structure occupies a sector. Some of these structures contain node specific information while other contains shared information. Any node that needs to access shared sectors needs to acquire corresponding disk lock. The corresponding nodes can only write into the node specific sectors, but all other nodes can read the contents of other nodes sectors. We do not use any locks to access node specific sectors.

The first few sectors of the data blocks contain various different system file headers. These file headers are initialized when the filesystem is mounted for the very first time. We restrict the filesystem be mounted by node zero for the first time. Since any node that tries to mount first will get node 0, this is not considered to be a restriction. However if the user prefers to assign node numbers, then he need to make sure that the filesystem is mounted first by node 0 for the very first time. Once these structures are initialized any node can mount and dismount at any time.

The data blocks for system files are allocated as and when required. There are no reservations made for any other space allocations except for the OCFS Header and system file headers. There are around 8 types of system files for each node and each file header occupy 512bytes. So enough space is reserved for maximum number of nodes that can mount (in V1 it’s limited to 32 nodes).

The data blocks and metadata blocks for regular files are allocated and de-allocated as and when required with an exception for root directory. The root directory information is stored as part of the ocfs directory node structure and is immediately followed after the system file header sectors and allocated system file blocks. The OCFS Header maintains the starting disk offset to this root directory node structure on the disk and is an entry point for accessing any file or directory on this filesystem. This directory node is created and initialized when the filesystem is mounted for the first time.

1.1.1 OCFS Header

OCFS being ”clustered” filesystem maintains node specific information required for node monitoring and inter-node communication as part of the OCFS Header. It also maintains locks and bitmap structures, which govern the space allocations in the data blocks. OCFS further subdivides the header into the following blocks.

1. OCFS Super block
2. Node Config sectors
3. Publish sectors
4. Vote sectors
5. Global Bitmap
6. Free Sectors

The disk layout is created and partially initialized during the format of the device. A device must be formatted with mkfs.ocfs or ocfstool to run the OCFS filesystem. mkfs.ocfs or ocfstool are standalone programs that are aware of the OCFS disk layout and hence can be run even before OCFS module is loaded.

1.1.1.1 OCFS Super Block

OCFS Super block occupies the first 8 sectors of the OCFS volume and contains the following

1. Volume Header
2. Volume Label
3. Global Bitmap Lock
4. 5 reserved sectors (UNUSED)

Volume header is stored in the very first sector of the volume and contains information about device size, OCFS version, block or cluster size, number of clusters/blocks, maximum number of nodes that can mount this volume. It also contains starting disk offsets to node config sectors, new node config sectors, publish sectors, vote sectors, global bitmap block, system files block, and the data blocks. These values are calculated and updated in the header during the disk format.

The Volume Header also maintains the disk offset to the root directory (root_dirnode), which is calculated and updated during the very "first" mount of the volume. The OCFS volume header is defined as ocfs_vol_disk_hdr, which is defined in Common/inc/ocfsvol.h. Each OCFS volume has the signature "`OracleCFS`" stored as part of the header to identify the OCFS volume and also to identify against any corruptions.

Volume Label is stored in the second sector on the OCFS volume and contains information about the Volume Name, Volume Name Length, Volume Id, and Volume Id Length. This structure is created and initialized during the format.

OCFS Volume Label structure is defined as ocfs_vol_label in Common/inc/ocfsvol.h. There are few other values defined in this structure (cluster_name and cluster_name_length), which are currently not used.

Global Bitmap Lock is the lock structure that maintains the lock information for the global bitmap block. Any node that wants to allocate some space from the global bitmap will have to acquire this lock, update the global bitmap and release the lock. This structure also maintains a counter, which keeps track of the number of bits that are set in the Global Bitmap. Whenever any process on any node allocates some space will have to increment this counter, and vice-versa.

This structure is defined as ocfs_bitmap_lock in Common/inc/ocfsvol.h.

1.1.1.2 OCFS Node Config Sectors

OCFS Node config sectors immediately follow the OCFS Super block sectors, and there are 38 OCFS Node Config Sectors, which are grouped as follows: These sectors are cleared only during the format, and are initialized and updated as and when nodes mount.

1. Node Config Header (1 sector)
2. Free (1 sector)
3. Node Config Info (32 sectors, one per node)
4. New node config Info (1 sector)
5. Node Config Header (1 sector)
6. Free (2 sectors)

1.1.1.2.1 Node Config Header

Node config Header is stored in the very first sector of these sectors and a copy is also maintained always in the 36th sector. This copy is located adjacent to publish sectors so that it can be read by the NM thread (Node Monitor thread) when it reads the publish sectors.

This structure contains the signature "NODECFG", which is used to uniquely identify the Node Config structure. It contains Version number (in V1 it's always 2) to protect against any structure changes in future. It maintains a counter (number of nodes), which indicates how many node config slots have been used from the node config sectors. When a new node mounts the filesystem for the first time, it gets a new slot and the number of nodes counter gets incremented. If this node either unmounts or remounts using the same node number, this value remains unchanged. But if the node gets a new slot (by generating guid and not specifying reclaim-id) when it remounts, then this counter will get incremented and the old slot will remain unused (basically we lost hat slot).

The node config header maintains another counter, config sequence number, which is incremented whenever a node mounts this volume. This value is incremented by the node that mounted the filesystem to indicate the other nodes that it has mounted the volume and hence they need to refresh their node configuration. The config sequence number doesn't get incremented when a node unmounts the volume.

This structure is defined as `ocfs_node_config_hdr` in `Common/inc/ocfsvol.h`

1.1.1.2.2 Node Config Info Sectors

There are 32 Node Config Info sectors, which holds 32 node config info structures (1 in each sector), which can hold information for 32 nodes. These sectors are allocated and initialized to null when the volume is formatted. When a node mounts the volume for the first time, it uses one slot from these pre-allocated slots. Once a node gets a slot it always uses the same slot unless the node guid is modified and user didn't specify to reclaim. Once a slot is used by one node it is never used by another node.

This structure contains a disk lock structure, which is used for locking the sector when we are using. When a node tries to mount the volume, it updates this disk lock structures lock structure with 1 indicating that it is using the slot. This disk lock structure in node config structure is used only when the node is mounting the volume.

It contains the node name structure, which holds the name of the node that is used at the mount time. OCFS doesn't depend on the node name and doesn't prevent the user from re-using the same node config slot when the node name changes. It doesn't depend on the node_name directly

and only uses it for informational and sanity checking purpose only. If the node name gets changed then this value gets updated in this structure when the node re-mounts the volume.

The node config structure maintains the guid (global unique id), which is a unique value that is generated from using the NIC's mach address. The guid is 32 bytes long and consists of the mach address along with a hostid that is generated using the mach address. This guid uniquely identifies a node and OCFS uses this when the node is mounting the volume. When a node mounts the volume OCFS reads the macid, calculate the guid, and then scan all the node config sectors to identify this nodes slot. If no slot contains this guid, then this could be a new node or the NIC card changed. If the NIC card changed then the user can specify "reclaimid" option during the mount time; then OCFS looks for the matching hostname and reuses that slot. If both the node name and the mach address changed at the same time, then there is no way for OCFS to identify that this is an old node. In this case it allocates a new slot and we lose the old slot forever.

The node config structure contains the IP address and the port number on which the OCFS listener thread on this node will be listening. By default OCFS uses port 7000 if available or else the user needs to mention this in the `/etc/ocfs.conf` file. Each node stores this information in their corresponding node config structure, by which other nodes will identify the IP addresses of the rest of the nodes and the port number on which their listener threads listen.

Node config structure is defined as `ocfs_disk_node_config_info` in `Common/inc/ocfsvol.h`

1.1.1.2.3 New Node Config Info Sector

This sector holds the disk lock info and is located immediately after the node config sectors. OCFS uses this sector to synchronize from different nodes mounting the same volume at the same time. When a node wants to mount a volume it reads the new node config sector of that volume to see if any other node is mounting the volume at the same time. If not then it updates that sector with the disk lock indicating that it is now mounting the volume.

Once a node acquires this lock it spawns a kernel thread, which keeps writing this lock information at this sector for every `OCFS_VOLCFG_LOCK_ITERATE` (10 jiffies) +jiffies. If another node is trying to acquire the lock then it first reads this sector to find if any other node already acquired the lock. If yes, then it sleeps for `OCFS_VOLCFG_LOCK_TIME` (1000ms) and re-reads the sector to find if that node is still locking. If yes then it assumes that the first node might be dead and tries to break the lock by writing it's lock request. It then sleeps again for `OCFS_VOLCFG_LOCK_TIME` time and re-reads the sector to find if its lock request has been overwritten. If the request has been overwritten then the other node is still alive and hence this node re-tries, if not then since it got the lock it continues with the mount.

It uses the same node config header structure `ocfs_disk_node_config_info` (as it contains the disk lock structure `ocfs_disk_lock`) defined in `Common/inc/ocfsvol.h`

1.1.1.2.4 Node Config Header

This is the copy of the node config header that is in the first sector of the node config sectors. This is written again here in this sector because it is adjacent to the publish sectors which are read by the NM thread on every node. The NM thread reads this sector to find if any new/old mounted

the volume. If yes then NM thread re-reads the 32 node config sectors and updates its in-memory structures.

Every time any node updates the node config header structure in the first sector, it also updates the node config header in the 36th sector.

1.1.1.3 Publish Sectors

Publish sectors are used by nodes for heartbeats, and for initiating vote requests when they are using Disk DLM. Heartbeats are always done over the disk, but vote requests are first initiated over the network and will fallback to disk (for that request only). There are 32 publish sectors and each node gets a slot depending on its node number. A node can only write into its own slot, but can read other nodes publish sectors.

Any process on that node can write into the node specific publish sector to request for a vote from other nodes. To synchronize multiple processes accessing this slot at the same time, we use publish_lock lock, which is created during the volume mount.

A nodes publish sector contains a time counter which is updated by the NM thread on that node. The NM threads on one node when reads the publish sectors of the other nodes, it gets the time value of the other nodes and compares to the previous read value. If the new value of a node is same as old value then it is counted as a “heartbeat miss” and it increments the misscount counter for that node. If the max misscount value (MISS_COUNT_VALUE = 20) is reached then it decides that corresponding node is dead. If the time value changed then it resets the misscount counter to zero indicating the node is alive.

It contains a dirty field, which is marked to TRUE when a node is requesting for a vote. The vote_type filed contains the type of vote that a node is interested in. The publish structure also maintains a vote_map to indicate which nodes should vote. If a node wants another node to vote for its request, then it sets the bit (in the nodemap) corresponding to the node number, and resets the corresponding bits for the nodes that it is not interested in.

This structure maintains a sequence number to map the vote request. Whenever a node wants to request for a vote, it then checks if any other node is requesting the vote. If not then it also scans what is the largest sequence number thus far, and increments the value for one to indicate that is new request. There could be a race condition that could happen, where 2 or more nodes can read the sectors at the same time and think that they can request a vote. In this case the node with the highest node number wins the race and all other nodes will respond to that nodes vote. The node with the small node number has to retry the request.

In OCFS every resource is identified with its disk offset. For ex: if a file F1 has been created at disk offset DO1, then any node would request a lock on DO1. So the publish structure contains the filed which holds the disk offset. A node that needs a lock on a resource would update this field with the corresponding disk offset value.

Publish sector structure is defined as `ocfs_publish` in `Common/inc/ocfsdisk.h`

1.1.1.4 Vote Sectors

Vote sectors are used to respond for a vote request. These sectors are used for the requests that came over the disk. The vote requests are always initiated on the network, but for some reason if the requester didn’t get the vote responses from all nodes in time, then he will request for those nodes on disk.

There are 32 vote sectors and each node gets a slot depending on the node number. Each node can only write into its vote sector but can read other nodes vote sectors. The process that is requesting the vote reads the vote sectors to check for the vote response. The NM thread on the other nodes will write the response into these sectors.

Vote structure contains an array field for vote responses. A node updates the corresponding array column depending on the node number that it is responding. For e.g., if node 1 and node2 wanted to request for a vote, then node1 and node 2 first checks if any other node is requesting for a vote. Both of them then initiate their vote request in the public sectors. But if it so happen that node 1 has written the request and node 3 has read the vote request and node 2 has written its request. Now according to the algorithm ever other node should respond to node 2’s request and but node 3 thinks that it needs to respond for node 1. So if it just votes then node 2 might think that it might be voting for node 2. So to avoid this misinterpretation, vote response field is maintained as a structure. So only the corresponding field is set like vote [nodenum]=vote response.

Vote structure also maintains another field vote sequence number, which is the copy of the public sequence number from the vote request. This is used to indicate for which request we are responding for that node. Vote structure also contains a field for disk offset of the resource for which it is responding. This value is copied from the publish structure. The vote structure also maintains a flag to indicate if that resource is open within that node. It may happen that a node can vote yes, but wants to indicate the requester that it is having the resource open, this will indicate the requester, whom to respond after completion if it has to respond.

Vote sector structure is defined as `ocfs_vote` in `Common/inc/ocfsdisk.h`

1.1.1.5 Global Bitmap

Gobal bitmap is used to account for the disk space on the volume. Global bitmap uses a single bit to identify a single block. The bit is set to 1 if the corresponding block is used, and that bit is set to 0 when the block is unused. The size of this structure is 1MB, which is pre-allocated and set to 0 at the time of format.

This structure along with the OCFS cluster/block size dictates the maximum size a volume can be. The maximum volume size is computed as 1MB * block_size * 8(bits). All the nodes that want to allocate space will access this map. The global bitmap lock that is maintained in the OCFS header synchronizes access to this block. Any node that wants to modify this map needs to acquire the global bitmap lock.

1.1.2 Data Blocks

Data blocks are created during the format and all the blocks are of same size that the user specified during the format. This is the place where system file headers, system file data, regular files and directories are stored. The space management of these blocks is monitored in the Global Bitmap.

The following structures are stored in these blocks

1. System file headers
2. System file data
3. DirNodes (Directory headers structures)
4. File Entries (File Header Structures)
5. Extent groups
6. Data blocks

1.1.2.1 System File Headers

System file headers get created when the volume is mounted for the very first time. Only the file headers each of size 512 bytes are created for all the possible system files. System files Header structures are same as the other file header structures; only the access pattern and the type of data they store are different. These headers are stored at predefined offsets on the disk and hence can be accessed directly. There are 6 types of system files that are currently used, they are:

1. DirFile
2. DirBitMapFile
3. ExtentFile
4. ExtentBitMapFile
5. RecoverLogFile
6. CleanUpLogFile

In OCFS the metadata structures DirNode and Extent groups are of fixed sizes 128KB and 512 bytes respectively. Since OCFS allows users to create data blocks of different sizes, it uses the first 4 system files to accommodate the DirNode Structures and Extent group structures. RecoverLog File and CleanUpLog files are used for recovering the metadata incase of node failures.

Each node needs the above set of files and the headers are pre-created for the maximum possible nodes that can mount the volume. OCFS currently allocates space for 2 other system files OCFS_VOL_MD_SYSFILE and OCFS_VOL_MD_LOG_SYSFILE. Only the headers are allocated for all the nodes. Since these files are not used, this document will not talk about it.

1.1.2.1.1 DirFile

DirFile data blocks store the DirNode structures. DirNode structure is of 128KB and OCFS needed a mechanism to allocate these fixed size structures irrespective of OCFS block sizes that

the user specified during format. DirFile grows as and when the directory structures are created. Once the file is grown it cannot shrink, but the space is reused, when new directories are created.

Each node has one DirFile, and they are located at offset 2*(32+node#)*512 + data block starting offset. The actual data blocks for these files can be located anywhere.

1.1.2.1.2 DirMapFile

DirMapFile contains bitmaps to the space allocated to DirFile. It keeps track of which slots are used in the space allocated to the DirFile. When blocks are allocated to DirFile, then DirMapFile will initialize the bitmap structure, which keeps track of slots that can be allocated for DirNodes. If a particular slot is used then it sets the corresponding bit to 1 and sets to 0 when the corresponding slot is freed. When a user tries to create a directory, OCFS first scans the DirMapFile to find a free slot in the DirFile, once an unused slot is found it uses that slot and sets the bit to 1.

1.1.2.1.3 ExtentFile

ExtentFile system file is similar to DirFile, but it keeps track of Extent group structures instead of DirNode structures. The allocation and access mechanisms are same as the DirFile except the sizes allocated to ExtentMap structures are of 512bytes.

OCFS File Entry (File Header) has room to keep track of 3 extents (extents are data actual blocks). If the file grows beyond the 3 non-contiguous extents, then it creates another special structure called ExtentMap. The File Header keeps track of these extents, which in turn keeps track of the actual extents.

1.1.2.1.4 ExtentMapFile

It is similar to DirMapFile but keeps track of space allocations done within the ExtentFile.

1.1.2.1.5 RecoverLogFile

There are two system log files per node: Recover logfile and Cleanup logfile. There is no global log file. To calculate system file ids, which is really just the sector offset:
recover_file_id = (__u32) (LOG_FILE_BASE_ID + node_num);
cleanup_file_id = (__u32) (CLEANUP_FILE_BASE_ID + node_num);

The Recover Log is used to log stuff to be done in case we have to abort an operation.

Each log file consists of a series of log records (either cleanup records, or recover records). Though some of the variables are there, there is really no knowledge of what constitutes a "transaction". A single process might log several types of records in the logfiles, all related to a single action (for example, an extend) and even give each record the same transaction id, but the logging layer really doesn't care. As an aside, log files can only grow in allocated space, never shrink. There's a chicken and the egg problem associated with truncating a journal.

1.1.2.1.6 CleanUpLogFile

The Cleanup Log is used to log stuff to be done in case we want to commit an operation.

1.1.2.1 System File Data

System Files data blocks could be stored anywhere on the disk. The System file headers keep track of these blocks. These blocks are allocated to the files as and when the files grow. Once a block is allocated to a file, it is never reclaimed back to the free pool, but however, the space is reused when it can be.

1.1.2.2 DirNode

DirNode is the structure that stores the information about a directory. Each DirNode is of 128KB and can hold up to 254 File Entries. A DirNode is referrenced by its disk offset (exact physical offset where the directory is located on disk), which any node/process uses to access the corresponding DirNode. The root DirNode is the only DirNode that gets created when the volume gets mounted for the very first time. Other DirNodes are created upon user requests. The Root DirNode will be the entry point to access files or directories. The OCFS volume header contains a pointer to the Root DirNode.

The DirNode structure contains a signature "DIRNV20", which is used to identify the on disk structure. Whenever a DirNode block is read the header is checked first to validate the structure. DirNode contains the disklock structure, which any node should acquire with appropriate lock before it can access this DirNode.

A directory can end up having multiple DirNodes, as a DirNode has room to keep track of only 254 File Entries (meaning 254 files or directories combined). The use will not be able to notice these multiple DirNodes, these are the structures that are internally maintained and can be viewed by using debugocfs tool. When more files/directories get created in the directory, a new DirNode gets created and link is maintained to that DirNode. If the files get deleted, then the newly allocated DirNode still stays until the user deletes the directory. OCFS doesn't do the merging of DirNodes, as that would be very expensive.

If a single directory ends up having multiple DirNodes, a new file creation would lead us reading multiple I/Os to read all the DirNode structures. To avoid this the DirNode structure would point to the DirNode that was created last. DirNode maintains an array of indexes, which point to the offsets of the FileEntries. This index gets sorted based on the file names during the file creation time. If a file is renamed, then this index array won't get sorted, instead it is marked dirty and an offset is maintained to that slot which would imply that OCFS has to sort on next time a read happens.

DirNode also maintains a first_del pointer, which points to the slot of the File Entry of the file that got deleted recently. If another file is deleted, then the first_del would point to the recently deleted File Entry slot and that would maintain a pointer to the previously deleted FileEntry slot. This would give OCFS a direct access to the free slots, which can be used when new files get created.

DirNode maintains two counters, the max File Entries that can be created and the current number of File Entries that got created.

DirNode contains the disk offset of where the DirNode structure is stored and also the relative location within the DirFile system file,

1.1.2.3 File Entries

A File Entry is the header of the file, which keeps track of the files metadata. A FileEntry is of size 512bytes and resides in the DirNode structure. A File Entry maintains offsets to the actual data blocks where the contents of the file are stored. A FileEntry contains name, modification time (time that metadata has been modified), userid, group id, and access permissions.

OCFS doesn't worry about the data access synchronization; hence it doesn't restrict 2 nodes accessing the same data blocks. It is up to the user application to synchronize the data access. OCFS maintains disk lock for synchronizing access to metadata. This locks needs to be acquired by the node to access the metadata of the file. Once the lock is acquired the status of the lock is updated on the disk.

FileEntry structure contains the signature "FIL" which is used to identify the FileEntry structure on disk. FileEntry maintains a pointer to the DirNode in which it is residing; this is the parent directory for this file. It maintains the disk offset to its own structure.

A file consists of a file header and data blocks/extents. The file header is the FileEntry in OCFS. The data extents are stored separate from the header. When a file is extended and needs a block, it tries to allocate space from the global bitmap. Once the space is got, it checks if the new block is adjacent to any of the old blocks. Assuming this is a new block, OCFS update the FileEntry structure with the starting offset to the disk and the length of the disk. If the file gets extend again and it got a block adjacent to the old block, the OCFS only increases the extent size as knowing the starting offset and the file size will be enough to get the contents. If the new block is not adjacent to the old one, then OCFS updates the starting offset and the file size in the FileEntry (extent) structure. A FileEntry can only keep track of 3 extents.

When a new block/extent is allocated and if all the extent entries are filled, then OCFS allocates a new structure called extent group. The FileEntry then copies the current extent entries into the extent group and stores the offset to the extent group in the FileEntry. Now the File Entry is indirectly pointing to the data extents. At this time the granularity is incremented by 1 indicating that a level is increased between FileEntry and the datablocks.

Since an extent group can only store up to 18 extent entries, if a new block has to be allocated, then another level of extents is created, and the granularity is incremented by 1. Now the File is pointing to the extent groups, which are pointing to another level of extent groups, which are pointing to the data blocks (double indirection). Currently OCFS can go up to 3 level of indirection.

When the granularity is 0 or greater, then OCFS maintains the offset to the extentmap that got created recently.

This structure is defined as `ocfs_file_entry` in `Common/inc/ocfsdisk.h`

1.1.2.4 Extent group

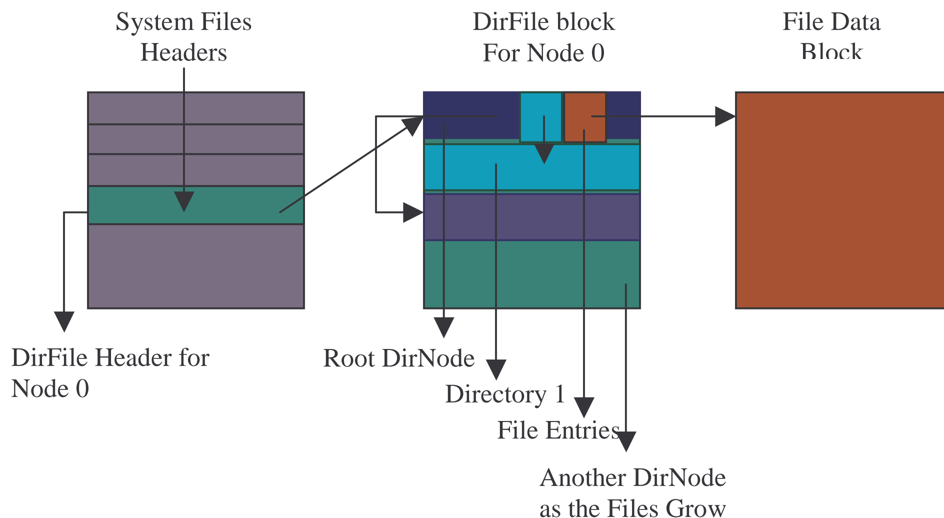
Extent groups are created to keep track of data blocks allocated to a file. Extent groups itself are stored in the ExtentFile systemfile. When a file gets extended and needs a block, OCFS first checks with ExtentMapFile systemfile, if it has room to accommodate space for the new extent group. If it finds an unallocated bit in the map file, then it creates the extent group in the ExtentFile system file and updates the corresponding file with the location of the extent group's diskoffset.

An Extent group structure contains two types of signatures "EXTDAT1" and "EXTHDR2". "EXTHDAT1" indicates that the extent group is pointing to the data blocks. "EXTHDR2" indicates that the extent group is pointing to another level of extent groups.

Extent group structure stores offsets to the actual disk offset and the relative offset in the ExtentMapFile. It also contains a pointer to the FileEntry and to the next adjacent extent. It contains an array, which holds the pointer to the data blocks. This array can hold up to 18 entries only.

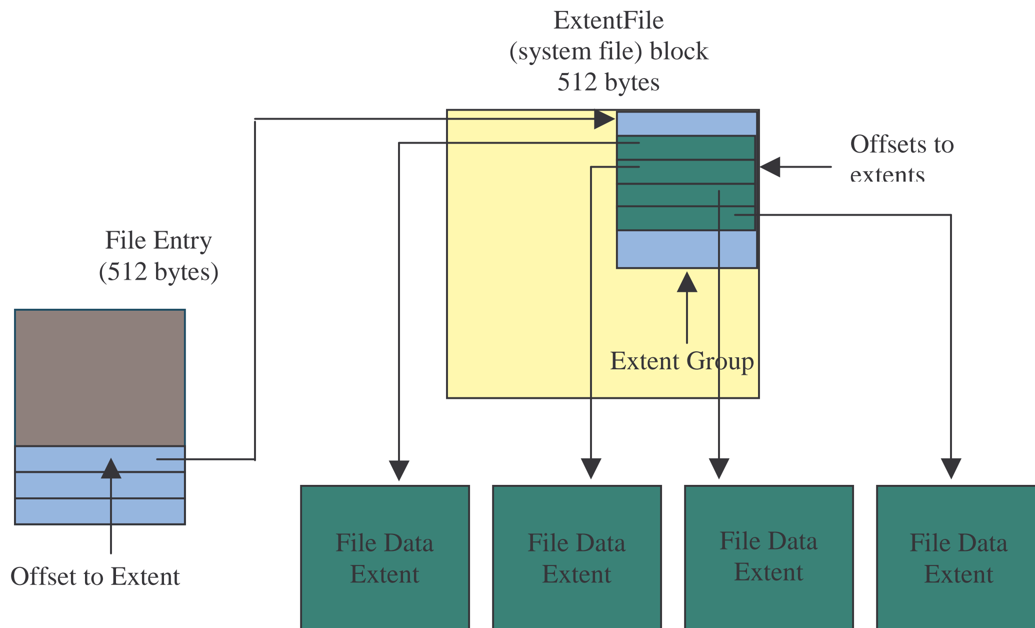
1.1.2.5 Data blocks

Data Blocks are just the physical blocks and OCFS doesn't maintain any structure to define a data block. The blocks are identified by the offset and the size.



This above diagram shows the following

1. A block that holds system file headers
2. A System file header block that holds DirNodes
3. More DirNodes for the root DirNode when more than 254 files got created
4. File Entry pointing to another dirnode when a new directory got created
5. File Entry pointing to the datablock



The above diagram depicts the relationships between FileEntry, Extent Group and Extents.

1. When a file has more than 3 non contiguous extents that's when an extent group come into picture
2. In the above diagram if there were only 3 Data Extents, then the 3 Extent pointers in the File Entry would directly point to Data Extents
3. If a file grows to more than 18 Extents, then File Entry would point to Extent Group which point another level of Extent Groups which point to the Data Extents

1.2 Memory Structures

OCFS creates and maintains some memory structures in which it stores the disk structures and also runtime information. OCFS memory structures can be categorized into the following:

1. OCFS Global Structures and
2. OCFS Volume Specific

1.2.1 OCFS Global Structures

Global structures get created when the OCFS module is loaded into the memory. These structures are created one per node and get deleted when the module is removed from the memory. OCFS creates the following global structures

1. Global Context
2. OCFS IPC Context
3. OIN Cache
4. OFILE Cache

5. FileEntry Cache
6. LOCKRES Cache

Global context is the main OCFS structure and maintains pointers to the above-specified global caches and also to the Volume specific main structure. The Caches are created so that the resource can be used if possible. These caches are created by using kmem_cache_create and destroyed by kmem_cache_release. When OCFS module gets loaded, it initializes the cache structures, and the memory is allocated and deallocated as required. These structures can be seen from /proc/slabinfo and are part of the slabcache. The slabcache does the memory management but the creation and deletion of these resources is initiated by OCFS. OCFS do not want kernel to shrink the cache.

1.2.1.1 Global Context

Global Context structure can be considered as the parent or grandparent of all the OCFS structures. Global Context maintains a linked list, which keeps track of OSB (Oracle Super Block) structures that gets created per each volume. It also maintains pointers to the OIN, OFILE, FE, and LOCKRES caches. It contains the preferred node number if the user has mentioned in the /etc/ocfs.conf. This value is stored here, because the preferred node number if valid applies to all the volumes. This structure contains the objid, which contains the type `OCFS_TYPE_GLOBAL_DATA` and size of the Global context itself.

Global context maintains node name, node IP address, and preferred port number for OCFS listener process. The flag reflects the state of this structure. When the structure gets initialized, the flag is set to `OCFS_FLAG_GLBL_CTXT_RESOURCE_INITIALIZED` indicating that it just started to initialize. The flag status `OCFS_FLAG_MEM_LISTS_INITIALIZED` indicates that this structure now has the pointers to the cache; this will allow OCFS module what to clean incase of failures. The flag status `OCFS_FLAG_SHUTDOWN_VOL_THREAD` indicates that the volume is shutting down to indicate the NM thread to exit as the NM thread loops checking for this status.

This structure is defined as `ocfs_global_ctxt` in `Common/inc/ocfsdef.h`

1.2.1.2 ocfs_ipc_ctxt

This is a global structure, which contains the information required for the ocfs clients to communicate with other nodes over the network. This structure contains pointers to the send and receives sockets that are created during the load of the OCFS module. It also contains the task structure of the listener process.

1.2.1.3 OIN Cache

OIN stands for OCFS Inode (index node), which is a wrapper to the VFS inode that represents an OCFS object. It is defined as `ocfs_inode` in `Common/inc/ocfsdef.h`. A single ocfs_inode represents a single object. This structure information including the VFS inode structure is filled by OCFS. It contains pointers to OCFS super block, file disk offset, dirnode

offset (incase it is a directory), parent dirnode offset, number of instances (processes) that opened this file, and flags that indicate the state of the inode. This structure maintains objid, which contains `OCFS_TYPE_OIN` (0x03534643) and size of the OCFS Inode structure.

OIN Cache keeps track of OCFS Inode structures. When OCFS needs memory, it requests the kernel to allocate the memory and account that memory into the OIN Cache. From the slab info we can see the size of each resource and the number of resource allocated and currently being used.

1.2.1.4 OFILE Cache

OFILE stands for Open file and is a wrapper to VFS file structure that defines an open file. This structure is defined as `ocfs_file` in `Common/inc/ocfsdef.h`. OFILE structure gets created when a file is opened, and there will one for one open, meaning if two processes open a file then we have two OFILE structures. Ocfs_file contains pointer to the OCFS inode that represents this object, VFS file structure that points to this file, pointer to the disk offset, its index within the DirNode, and a pointer to the DirNode structure. This structure also contains an objid, which contains `OCFS_TYPE_OFILE` (0x02534643) and the size of the ocfs_file structure. File Entry Cache

OFILE Cache is an in memory cache similar to OIN Cache and contains the OFILE objects.

1.2.1.5 FileEntry Cache

Same structure is used for in memory and ondisk and it is the header of the file or a directory. The directory also has the DirNode structure. The FileEntry cache is to keep track of allocations and deallocations of the memory used to FileEntries within the cache.

1.2.1.6 LockRes cache

LockResource is an in memory structure that synchronizes access to an OCFS object. Any process that needs to obtain a lock on a resource, should acquire a lock on the lockresource that represent the resource. These lockresources govern locks within a node; for locks across nodes OCFS tries to acquire disklock also. For ex: if a process needs to acquire a lock on a file, it first creates a lockresource if it doesn't exist already and acquires the lock on the lockresource, and then it tries to acquire a lock from other nodes if that node is not owning that lock.

LockRes cache is the slab cache similar to OFILE and OIN and is created and destroyed

1.2.2 OCFS Volume Structures

Each Volume has some volume specific global structures, which are used within the volume. These structures get created when a volume is mounted and gets deleted during the dismount. There could be multiple OCFS volumes mounted on a node and each volume specific structures are independent of others.

The following are the structures that are maintained in memory, apart from these OCFS also defines other structures to hold the ondisk structure information.

1. ocfs_super
2. ocfs_vol_layout
3. ocfs_inode
4. ocfs_file
5. ocfs_lock_res
6. ocfs_io_runs
7. ocfs_vol_node_map

1.2.2.1 Ocfs_super

OCFS super is the superblock structure for OCFS FileSystem which is a wrapper for VFS superblock. It contains OCFS volume specific information. When the volume is mounted, this structure is created and a pointer to this structure is linked to Global Context. This structure contains information like, number of open files, number of nodes configured, trans_id, offset to the root directory, pointer to the root directory inode, and status of the volume. It maintains a pointer to the NM thread to signal the thread when the volume is dismounting.

This VFS super block in this structure contains pointers to various OCFS methods like ocfs_statfs, ocfs_put_inode, ocfs_clear_inode, ocfs_read_inode, ocfs_read_inode2, ocfs_put_super. These are OCFS volume specific methods implemented by OCFS and are invoked by VFS appropriately.

1.2.2.2 ocfs_vol_layout

OCFS volume layout contains the physical volume specific information. Much of this information is populated from the volume header and contains disk offsets to various disk structures. It's an in memory structure that holds the volume header information and resides in ocfs_super.

1.2.2.3 ocfs_inode

ocfs_inode is the wrapper to VFS inode and an instance of it is created whenever VFS requests. VFS inode structure contains pointers to inode methods implemented by OCFS like, ocfs_create, ocfs_lookup, ocfs_link, ocfs_unlink, ocfs_symlink, ocfs_mkdir, ocfs_mknod, ocfs_rename, ocfs_setattr, and ocfs_getattr. These methods are invoked by VFS as to serve the user land requests.

1.2.2.4 ocfs_file

ocfs_file is the wrapper to the VFS file structure and is created for every open operation. VFS file structure contains pointers to OCFS implemented file methods like, ocfs_file_read, ocfs_file_write, generic_file_mmap, ocfs_sync_file, ocfs_flush, ocfs_file_release,

ocfs_file_open, and ocfs_ioctl. These methods are invoked by VFS directly to serve the user land requests.

1.2.2.5 ocfs_lock_res

ocfs_lock_res is the OCFS in memory lock structure. This is created for every object, file or directory that needs to be accessed. This structure contains, sector_num (physical diskoffset), which uniquely distinguishes one instance of this structure to the other. A HASHTABLE is maintained which holds the instances of these structures. Whenever a file/directory lock needs to acquire, the hashbucket is searched with the offset of the file/directory. If lock resource is not found then one is created and inserted into this hashtable.

When a process wants to acquire a lock on lock resource, it checks if the lock is in use and what lock does it hold. If the lock requesting is compatible, then inuse counter is incremented and the lockfield is set appropriately. It also maintains a field voted_event_voken on a which a process waits to be woken up by other process.

It also contains two fields request vote map and got vote map, which is used to tally if the node got the votes. Request vote map is initialized with the map of nodes that the node is requesting the votes. For every vote received the appropriate bit is set in the got vote map and finally they are compared to see if it got the votes. This is only used when the voting is happening over the network.

1.2.2.6 ocfs_io_runs

This structure holds three values, diskoffset, read length, and the offset within the buffer. When a user initiates and IO of certain length, then the whole data may not be contiguous on disk. OCFS may have to do multiple IO calls to service this requests. Ocfs_io_runs basically contains this information; this structure gets filled depending on how spread the data of that file is.

1.2.2.7 ocfs_vol_node_map

This structure is used to keep track of the heartbeat of the other nodes. It stores heartbeat time, interval, and misscount, mount and dismount status of each node.

2 OCFS Functional Design

2.1 Node Monitoring

Each node does node monitoring per each volume that gets mounted. The primary function is to check the health of the other nodes that have mounted the same volume (Heart beating). OCFS maintains a publish sector area in the OCFS Header portion into which each node writes its heartbeat counter value every 500 ms. After writing it also reads the publish sectors of all the other nodes sectors and process them. It checks each nodes heartbeat counter value and compares with the previous heartbeat time of that node.

A misscount counter is maintained for each node and for each volume, which gets incremented every time the previous and the current heartbeat match for that node. The counter is reset to 0 every time there is a difference in previous and current heartbeat counter. If the misscount counter has reached the max allowed misscount, which is 20, then the node is marked as dead. Each node maintains a map in which each bit indicates whether a node is alive or dead. If a misscount of a node has reached the max limit then the node map is updated on all the other nodes marking that the node is down.

In OCFS V1, node monitoring is done on the disk.

2.2 Distributed Lock Manager

OCFS has implemented DLM to synchronize access to the shared resources across the nodes. A resource can have any of the following locks:

1. OCFS_DLM_NO_LOCK
2. OCFS_DLM_SHARED_LOCK
3. OCFS_DLM_EXCLUSIVE_LOCK
4. OCFS_DLM_ENABLE_CACHE_LOCK

In OCFS DLM locks are reflect on the disk. A DLM lock is associated with the current owner of that resource. The current owner doesn't mean that if any one else needs the lock will request this node; it is used to find out if you are the owner or if some other node is the owner and if there needs any recovery in case the owner is dead

OCFS uses two mechanisms for lock managing, 1) lock resource and 2) disk lock. A lock resource is an in memory lock structure for a particular resource. Processes within the node will try to acquire this lock before they try to acquire the disk lock. The process trying to acquire will wait if some other process has marked the resource as in use. The acquiring process will either try until the resource is marked unused or the timeout. Once the lockresource is marked not in use, the acquiring process will mark it inuse and update the structure with its processid.

OCFS_DLM_EXCLUSIVE_LOCK and OCFS_DLM_ENABLE_CACHE_LOCK will always be reflected on the disk. OCFS_DLM_EXCLUSIVE_LOCK lock means that this resource is acquired in exclusive mode. OCFS_DLM_ENABLE_CACLE_LOCK means that the resource is cached on the owning node and hence needs to be asked to flush before that resource can be used. OCFS_DLM_SHARED_LOCK means that the resource is in shared mode.

If a node (process on that node) wants to acquire an OCFS_DLM_SHARED_LOCK, then it will first check if any other process on that node is using that resource. If yes, it will wait till timeout or that process release the lockresource. Once the lockresource is unused, it will mark it as in use and reflects the lock status from the disk. If the lock type on that disk resource is either OCFS_DLM_NO_LOCK or OCFS_DLM_SHARED_LOCK then it will just acquire the lock, as they are compatible. The shared lock acquisition is not communicated to other nodes as any exclusive request will anyhow be communicated to all nodes. If the lock type is OCFS_DLM_ENABLE_CACHE_LOCK, then the process sends a message to the owning node to flush its cache and then acquires the shared lock. If there is a OCFS_DLM_EXCLUSIVE_LOCK then it will retry.

If a node wants to acquire an OCFS_DLM_ENABLE_CACHE_LOCK or OCFS_DLM_EXCLUSIVE_LOCK then the reason for this lock should be specified. The reason could be any of create, delete, extend, rename, update, create directory, update ocfs inode, release master, change master, or release cache. If a node already has a lock, or if there is no lock on that resource it will ask for other nodes to vote, if it is either deleting or renaming the resource. Otherwise it will just acquire the lock. If another node owns a shared lock, and the reason is to delete or rename, it will then send request all the nodes to vote. If the reason something else, it will send the message to owner to change the owner of the resource.

If another node has the OCFS_DLM_ENABLE_CACHE_LOCK then it will send a message to break the lock or else it will wait until that node releases the lock. A node will hold the lock in exclusive mode only till it needs it, after that it will release the lock.

2.3 OCFS Journaling and Recovery

Most of the time, this is how a process logs something, almost always through ocfs_create_modify_file:

1. call ocfs_start_trans
2. do the action required of us, optionally logging stuff while we do it.
3. call ocfs_commit_trans if we succeeded, ocfs_abort_trans otherwise.

commit_trans and abort_trans actually replay the logfiles, so the process won't return from them until the logfile has been read off disk and all records have been processed.

Here is the general algorithm for how the two logs get processed: Commit Transaction (example, call ocfs_commit_trans):

1. Play back the cleanup log.
2. Truncate the recover log to zero.

Abort Transaction (example, call ocfs_abort_trans):

1. Play back the recover log.
2. Truncate the cleanup log to zero.

Node Recovery (example, call ocfs_recover_vol):

1. If both logs are empty we're done, otherwise:
2. If the recover log is empty
    a. Play back the cleanup log.
3. Else the recover log is NOT empty

a. Play back the recover log.
b. Truncate the cleanup log to zero.

2.3.1 Log Records

The two log record structs are in ocfstrans.h. They're both structs with a log_id (the transaction id), and a log_type (the specific record type), which helps them to determine which field in a union to deal with. The union (called 'rec') is where the two diverge. Each has their own different sets of structs in the union (some are the same between the two).

Recover logfile records are all sector sized (usually from osb->sect_size) and cleanup logfile records are like 7k or something (sizeof(ocfs_cleanup_record)) and are aligned (with OCFS_ALIGN) to the sector size.

```
recover_log_rec_size = osb->sect_size;
cleanup_log_rec_size = (__u32) OCFS_ALIGN(sizeof(ocfs_cleanup_record),
                                osb->sect_size);
```

These seem to be the sorts of things that we do in fact, log. They are also the only values that (ocfs_log_record/ocfs_cleanup_record)->log_type can take:

Below I will enumerate each possible log_type. I will note if it is unused by any other function (as in, it'll be marked unused if nobody even logs that type). If we don't even trap for that type in ocfs_process_record than I'll mark that too. Otherwise if it's unused
we might actually have code to handle it though it's anybody’s guess whether that stuff works. For reference, below each log_type I put the struct, which matches it on disk (if any). I have removed padding bytes from the structs. Any associated constants are also left there for convenience.

* Note that the descriptions below are essentially a description of what the code in ocfs_process_record does for each log_type.

2.3.1.1 LOG_TYPE_DISK_ALLOC

```
#define  DISK_ALLOC_DIR_NODE      1
#define  DISK_ALLOC_EXTENT_NODE   2
#define  DISK_ALLOC_VOLUME        3
typedef struct _ocfs_alloc_log
{
        __u64 length;
        __u64 file_off;
        __u32 type;
        __u32 node_num;
}
ocfs_alloc_log;
```

2.3.1.2 LOG_CLEANUP_LOCK

```
typedef struct _ocfs_lock_update
{
	__u64 orig_off;
	__u64 new_off;
}
ocfs_lock_update;
#define  LOCK_UPDATE_LOG_SIZE     450
typedef struct _ocfs_lock_log
{
	__u32 num_lock_upds;
	ocfs_lock_update lock_upd[LOCK_UPDATE_LOG_SIZE];
}
ocfs_lock_log;
```

2.3.1.3 LOG_TYPE_RECOVERY

ocfs_recover_vol writes one of these guys when it's recovering a node. It puts the nodes number in node_num. This way if we die during ocfs_recover_vol we can recover that node again when we come back up, or more likely, whoever recovers *our* logs will also recover that nodes logs.

```
typedef struct _ocfs_recovery_log
{
	__u64 node_num;
}
ocfs_recovery_log;
```

1. Save a copy of osb->node_recovering to a temporary variable
2. call ocfs_recover_vol on node_num
3. Put osb->node_recovering back to what it was.

2.3.1.4 LOG_FREE_BITMAP

If you need to unset any bit(s) in any of the bitmaps (global, node specifics), you'll wanna use this type and look at ocfs_free_disk_bitmap.

```
#define  DISK_ALLOC_DIR_NODE      1
#define  DISK_ALLOC_EXTENT_NODE   2
#define  DISK_ALLOC_VOLUME        3
typedef struct _ocfs_free_bitmap
{
	__u64 length;
	__u64 file_off;
	__u32 type;
	__u32 node_num;
}
ocfs_free_bitmap;
#define  FREE_LOG_SIZE            150
typedef struct _ocfs_free_log
{
	__u32 num_free_upds; /* Number of free updates */
	ocfs_free_bitmap free_bitmap[FREE_LOG_SIZE];
```

```
}
ocfs_free_log;
```

call ocfs_free_disk_bitmap on our record. This has the effect of actually freeing those bits from the proper bitmap.

2.3.1.5 LOG_UPDATE_EXTENT

Looks like if you were allocating a new extent for a file entry, you would stick one of these in your recover log so that if you die, or the operation fails, it can abort it (by clearing out the actual extents file_off, disk_off, num_bytes triplet). You will need to stick in some other records to free allocated space and stuff though.

```
typedef struct _ocfs_free_extent_log
{
    __u32 index;
    __u64 disk_off;
}
ocfs_free_extent_log;
```

1. read the extent group at disk_off
2. zero out file_off, num_bytes and disk_off of the extent at location index.
3. write the extent group back to disk.

2.3.1.6 LOG_DELETE_ENTRY

Deletes a file entry. Looks like you'd put this in your cleanup log if you were gonna do a delete on this file entry, the commit_trans will finish the work, or if you die, the recovery should...

```
typedef struct _ocfs_delete_log
{
    __u64 node_num;
    __u64 ent_del;
    __u64 parent_dirnode_off;
    __u32 flags;
}
ocfs_delete_log;
```

1. get the fe at ent_del
2. get the fe at parent_dirnode_off (call it lock_node)
3. look at the node_num, apparently for no reason.
4. call ocfs_del_file_entry on that fe, passing it the parent dir fe as well.

2.3.1.7 LOG_MARK_DELETE_ENTRY

Conditionally deletes a file entry? Does different things' depending on the flags parameter below.

```
typedef struct _ocfs_delete_log
{
    __u64 node_num;
    __u64 ent_del;
    __u64 parent_dirnode_off;
```

```
        __u32 flags;
}
ocfs_delete_log;
```

1. get the fe at ent_del
2. if flags has FLAG_RESET_VALID set, then set OCFS_SYNC_FLAG_VALID in the fe->sync_flags, write that fe back out to disk and break.
3. if a flag has OCFS_SYNC_FLAG_VALID set, then break.
4. call ocfs_delete_file_entry, passing it the fe, the parent_dirnode_off and the node_num.

2.3.1.8 LOG_DELETE_NEW_ENTRY

Deletes the specified entry. Looks like you'd put one of these records in your recover log if you were creating a new file entry and wanted it removed in case of an abort.

```
typedef struct _ocfs_delete_log
{
        __u64 node_num;
        __u64 ent_del;
        __u64 parent_dirnode_off;
        __u32 flags;
}
ocfs_delete_log;
```

1. get the fe at ent_del
2. get the fe at parent_dirnode_off (call it lock_node)
3. look at the node_num, apparently for no reason.
4. call ocfs_del_file_entry on that fe, passing it the parent dir fe as well.

2.3.1.9 LOG_TYPE_DIR_NODE

unused, though a tiny bit of code is in place:

```
typedef struct _ocfs_dir_log
{
        __u64 orig_off;
        __u64 saved_off;
        __u64 length;
}
ocfs_dir_log;
```

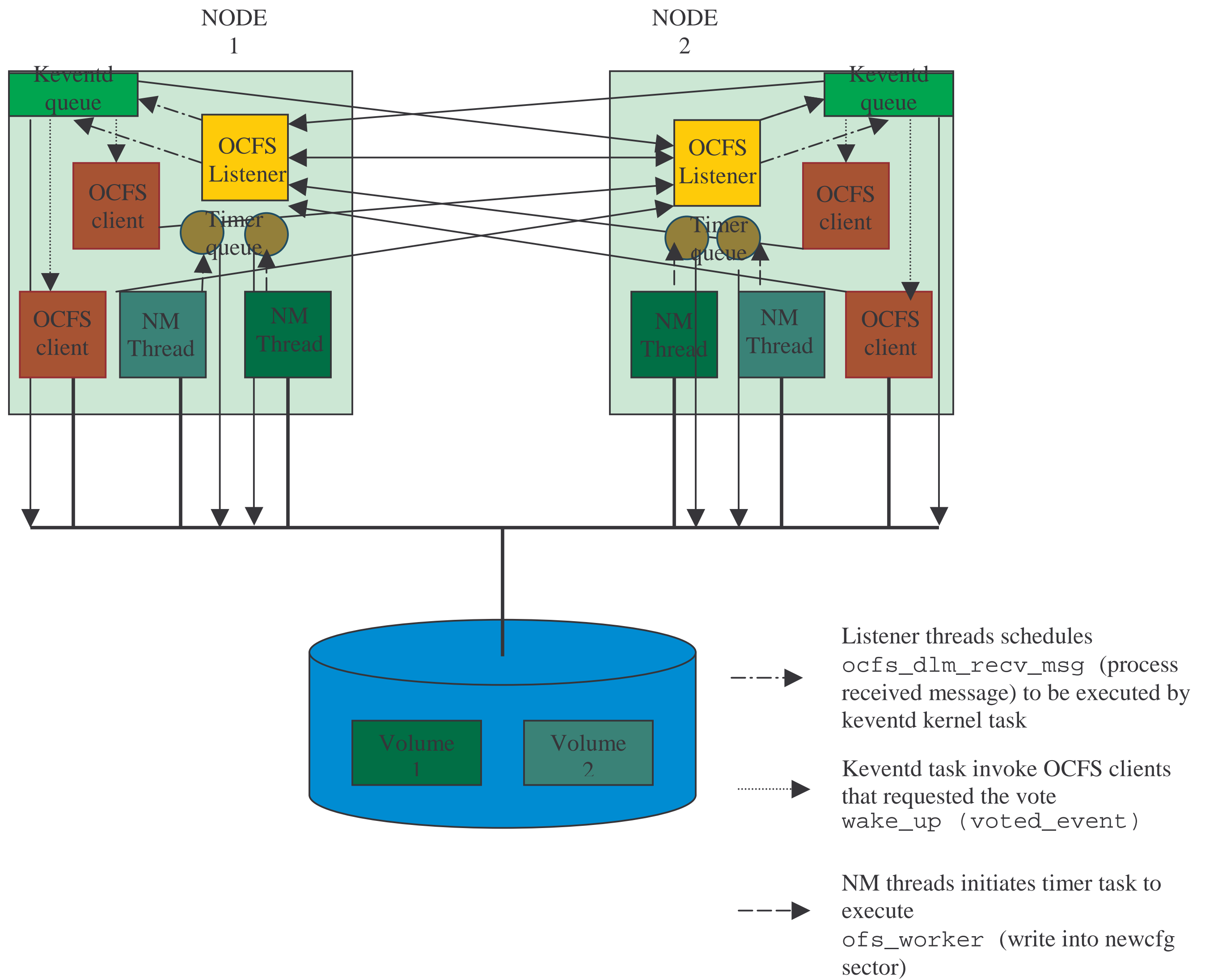
1. call ocfs_recover_dir_node, passing orig_off and saved_off. This does nothing usefull as ocfs_recover_dir_node is an empty function.

2.4 OCFS Processes

OCFS Filesystem has implemented node communication and node monitoring services along with the regular filesystem services. The following diagram gives an overview of the processes involved in the node communications and gives an overview of how they interact with each other in a clustered environment. OCFS has the following processes

1. OCFS client process

2. OCFS Node Monitoring Thread
3. OCFS Listener Thread
4. Kernel Task Queues



2.4.1 OCFS Client Processes

OCFS Client Processes are the user land processes that are accessing objects to or from OCFS file system. A user land processes executes a system call, which gets translated into OCFS call through the VFS layer. When the volume is mounted on a directory, OCFS will create an OCFS superblock and provides that to VFS. Superblock contains the inode of the root directory of that volume and also contains pointers to "super" methods that OCFS has implemented.

When the client process tries to access a file or directory on OCFS volume, VFS first builds the complete path to that file or directory and resolves each directory at a time. It basically invokes the OCFS lookup method and passes it the inode of the parent directory and dentry of the file or directory it is looking for. OCFS then locates the file on the disk and builds an inode structure and passes it back to the VFS layer. The inode structure will now contain pointers to inode methods implemented by the OCFS. The procedure is repeated for each and every directory along the way to the actual file or directory.

OCFS has implemented methods that are defined by the VFS, which are required for accessing OCFS filesystem. Any processes once entered into the kernel executing the OCFS code will stay here until OCFS checks if there are any signals for this process and process them or done with the requested job.

A process once entered into OCFS layer will have access to 2 OCFS global structures, Global context and global ipc context structures that are created and initialized at the OCFS load time. Apart from this these process will have access to the OCFS inode, OCFS superblock structures that are provided by VFS.

The entry point for the OCFS clients into the OCFS is code is through the OCFS implemented methods.

2.4.2 Node Monitoring Thread

OCFS does node monitoring to monitor the nodes that mount a shared volume. It keeps track of if any nodes mount, unmount, hung or crash. When a new node mounts the volume, cfg_seq_num counter is incremented. The other nodes that mounted the same volume will then realize that a new node has joined the cluster and updates their in memory structures. Once a node is joined it is then monitored if it is alive or dead. As long as the heartbeat counter is incremented, the corresponding node is considered alive. If 20 heartbeats are missed then that node is considered to be dead and the publish map is updated to reflect this.

If the network dlm is used, then a dismount message is broadcasted to all the live nodes other wise each node has to find the dismount, hung or crash on their own. Nodes won't communicate with each other to reconfig, they find this on their own. In version 1, OCFS does the node monitoring over the disk and is done by NM thread.

2.4.3 OCFS Listener Thread

OCFS listener is started one per node when the first volume is mounted on that node. It exits during the last volume's dismount. The primary function of listener thread is to facilitate the vote requests over the network. During the first mount, the mount thread will create and initialize send and receive sockets and binds the receive socket to a user defined port. If for some reason the listener thread couldn't be started, the volume will still get mounted.

The job of the listener thread is just to listen on the socket for any new messages and then queue the request in the task queue, which is executed by the keventd one at a time. It doesn't send any acknowledgements or initiate any vote requests. The protocol used is UDP.

The OCFS Listener code is defined by `ocfs_recv_thread` in `ocfs2/Linux/ocfsipc.c`, which is started the mount process that is mounting the first OCFS volume on that node

2.4.4 Kernel Task Queues

OCFS uses two kernel task queues; the timer queue to execute `ocfs_assert_lock_owned` and the schedule queue to execute `ocfs_dlm_recv_msg` function. The process that is mounting the volume initiates triggers the timer queue to execute `ocfs_assert_lock_owned` to continuously write to the newconfig sector. The schedule function is executed once for every 10 jiffies for about 1000ms. This function is scheduled to execute only one time during the startup of the mount.

OCFS listener thread listens on the sockets for any incoming vote requests. Once the requests arrives it schedules `ocfs_dlm_recv_msg` function to be executed with the message that it received as the input. The keventd thread later executes this function. This function services the incoming requests and then replies to the requesting nodes listener thread on the sending socket that is stored in the global OcfsIpcCtxt structure.

3. OCFS Implementation

In Linux VFS layer is not cluster aware and hence OCFS had to design and implement additional functionalities to implement a cluster ware filesystem. VFS defines certain methods, which it calls as and when required. OCFS to be a cluster aware filesystem has implemented functions required for clustering and also functions defined by VFS. OCFS has implemented the following functionalities

1. OCFS Node Monitoring Thread
2. OCFS Listener Thread
3. OCFS DLM Methods
4. OCFS Journaling & Recovery
5. OCFS Module Operations
6. OCFS Super Operations
7. OCFS File Operations
8. OCFS Directory Operations
9. OCFS Inode Operations
10. OCFS Address Space Operations

OCFS Node Monitoring, Listener Thread, DLM methods are defined by OCFS and the rest are defined by VFS. Journaling & Recovery functions are not defined by OCFS, but are needed for any filesystem that wants to recover from system crashes. OCFS has implemented only metadata recovery.

3.1 OCFS Node Monitoring Thread

In OCFS, Node Monitoring Thread does the node monitoring. There is one NM thread per volume per node. Multiple NM threads on a single node are independent of each other. Each NM thread on a node monitors its own volume. It keeps track of any other node that mounted the volume from being hung, crashed or unmounted. The NM threads doesn't inform each other when they have unmounted, each NM thread has to identify this on their own. However if network DLM is used, unmount message is sent.

3.1.1 ocfs_volume_thread

ocfs_volume_thread is the main routine for the NM thread, which does the heart beating, and monitor the health of other nodes. The NM thread is a kernel thread and is started by the mount process by calling kernel_thread as

```
kernel_thread (ocfs_volume_thread, osb,
               CLONE_FS|CLONE_FILES|CLONE_SIGHAND)
```

The NM thread initializes to run as a daemon with `init` as its parent to make sure that things get cleaned-up when the thread exits. It reads the publish sectors of all the nodes and updates it's own publish sector with the heartbeat time and node information. It then waits for OCFS_HEARTBEAT_INIT iterations (10) and wakes the mount process that is waiting for NM thread to signal upon initializing.

Mount thread once started the NM thread, goes ahead and does some other initialization before it waits for NM thread to complete initialization. Now this leaves room for NM thread to initialize and then signal mount thread before mount thread actually waited for NM thread. Not knowing this mount thread can wait indefinitely, so to avoid this mount thread checks the osb->nm_init value and see if NM thread has increased this counter to OCFS_HEARTBEAT_INIT and only waits if its less.

NM thread loops indefinitely until the volume state is set to OCFS_FLAG_SHUTDOWN_VOL_THREAD or OCFS_OSB_FLAGS_BEING_DISMOUNTED, which indicate that the volume is being unmounted. NM thread sleeps for OCFS_NM_HEARTBEAT_TIME (500ms) and reads the publish sectors of all the other nodes. It first writes its heartbeat time and compares the other nodes heartbeat times. If the times are same, misscount counter for that node is incremented and if they are different, then new heartbeat time is saved and misscount counter is reset indicating the node is alive. Once the misscount counter is reached to MISS_COUNT_VALUE (20), then the node is marked DEAD in it's publish map. It basically resets the corresponding bit to zero in osb->publ_map.

3.2 OCFS Listener Thread

OCFS Listener Thread listens on the predefined port to listen for incoming lock requests. This thread is created when the first OCFS volume is mounted, and stays until the last OCFS volume dismounts. There is only one listener thread per volume. Ocfs_recv_thread is the main routine for the listener thread

3.2.1 ocfs_recv_thread

ocfs_recv_thread is the main routine for the OCFS listener thread. This is a kernel thread that gets created by the mount thread. The mount thread first creates a send socket and a listener socket. It binds the listener socket to the predefined port (default 7000) and then creates the listener thread as

```
kernel_thread (ocfs_recv_thread, NULL,
               CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
```

The mount thread then initializes a completion structure (`OcfsIpcCtxt.complete`) on which the unmount thread waits for this thread to exit. The unmount thread first sends a SIGINT interrupt before it waits for the thread to exit.

The ocfs_recv_thread code is simple, so that it could keep listening to the incoming requests without any delay. It first daemonizes itself and makes `init` thread as its parent, to get a proper cleanup after exit. It then enters an infinite loop, where it waits on the incoming socket. Once a request is received, it calls `schedule_task` to queue a request for `keventd` thread to execute `ocfs_dlm_recv_msg`. Once the request is scheduled it then goes back and listens on the socket. If an interrupt is received before a message is received, listener thread will exit.

At the time of exit, this thread will release the send and receive sockets, and waits for all the schedule requests to complete (calls `flush_scheduled_tasks`) and then signals the unmount thread that is waiting on `OcfsIpcCtxt.complete`.

3.3 OCFS DLM Operations

3.3.1 ocfs_acquire_lock

ocfs_acquire_lock is the main lock routine that is called to acquire the lock on the disk. The requesting lock type could be OCFS_DLM_SHARED_LOCK or OCFS_DLM_EXCLUSIVE_LOCK or OCFS_DLM_ENABLE_CACHE_LOCK. It is called before a resource is being accessed. The resource id on which the lock is requested is specified as the offset on the disk.

This routine first creates a in memory File Entry structure to store the file header information. It then calls ocfs_find_update_res, which will try and locate if there is an existing lock resource structure in the memory.

OCFS stores the lockresource structures in LockRes slab cache and also maintains a hashtable for a quick search. This routine first looks in the hashtable; if found it marks the resource that it is interested in that resource (increments lr_ref_cnt). If there is not one in the memory, then it a lockres structure needs to be created. ocfs_allocate_lockres is called to allocate memory from LockRes slab and then ocfs_init_lockres is called to basic initialization of this structure. ocfs_disk_update_resource is called to read the resource from the disk and update the newly created lockres structure. Once the structure is updated, it will try to insert this lockres in the HashTable. ocfs_insert_sector_node, first calculates a hashvalue and then checks hangs to the end of the appropriate bucket list.

Once the in memory structure is refreshed from the disk for that resource, now the lock request will be initiated. If the lock request is OCFS_DLM_SHARED_LOCK (this locktype is requested from only one place, to find files on the disk) and there is no lock (OCFS_NO_DLM_LOCK), or this node owns the resource, then it is simply taken by setting locktype to OCFS_DLM_SHARED_LOCK. Otherwise if the lock on this resource is OCFS_DLM_ENABLE_CACHE_LOCK and is owned by some other node, then ocfs_break_cache_lock is called to break the lock. Otherwise the locktype on the resource is updated, but the lr_share_cnt is incremented to indicate the lockres is being used (dirty read). The OCFS_DLM_SHARED_LOCK is not reflected on the disk.

If the vote request is for OCFS_DLM_EXCLUSIVE_LOCK or OCFS_DLM_ENABLE_CACHE_LOCK, then ocfs_try_exclusive_lock is called to get the exclusive lock request on the disk.

3.3.2 ocfs_break_cache_lock

If an node is trying to acquire a disk lock on a resource, and there is already a OCFS_DLM_SHARED_LOCK lock acquired on that resource by other node, then this routine is called to break that lock.

It first tries to acquire a lock on the in memory locresource for that disk resource and prepares a vote map to which it needs to send this message (basically only one node that currently mastered it). It then calls ocfs_send_dlm_request_msg to send the vote request (reason = FLAG_FILE_RELEASE_CACHE) over the network. Once the message is sent, it waits for that node to get back. If it timed out it returns an error or if the response is to try AGAIN, then it sleeps for 500ms and then retry the request.

If the network voting timed out (network problems) then the request will be sent over the disk. It calls ocfs_request_vote, which first reads (ocfs_read_disk_ex) all nodes publish sectors and see if any node is requesting a vote. If so it waits and rechecks (Only ONE vote request can happen on disk at any given time). If not it will check the current highest sequence number and then increments it by one (to indicate it's the latest request). It then sets the vote to FLAG_VOTE_NODE, and vote_type to the reason (FLAG_FILE_RELEASE_CACHE) the vote is being requested. Once the vote request is written to the disk (nodes publish sectors), the node calls ocfs_wait_for_vote to check for the vote responses. ocfs_wait_for_vote sleeps for WAIT_FOR_VOTE_INCREMENT time and calls ocfs_get_vote_on_disk, which reads all the nodes vote sectors and traverse the response. If the node we are requesting is dead then the lock needs to be recovered and retried.

If the node is alive, and has not yet voted, it will be rechecked. If the node has voted, then the largest seq# is checked to make sure if the vote response is for the request that the node just initiated. If the vote response is FLAG_VOTE_NODE then it got the vote and the vote on disk will be OCFS_DLM_NO_LOCK. If the vote response is FLAG_VOTE_UPDATE_RETRY, then it didn't get the vote and needs to retry the vote.

3.3.3 ocfs_break_cache_lock

ocfs_try_exclusive_lock is called to acquire either a OCFS_DLM_CACHE_LOCK or OCFS_DLM_EXCLUSIVE_LOCK on disk. This routine will get the lock type, file header, file disk offset, and the reason for this lock (FLAG_FILE_CREATE, FLAG_FILE_EXTEND ...) from the caller.

It will first acquire the lock on the lockres (in memory, basically acquiring a lock on the same node to prevent other processes accessing the same resource). Then the lockres structure is refreshed with the disk contents (basically the current master, locktype). If the current node is the master of this lockresource, and if the reason for the lock is not FLAG_FILE_DELETE or FLAG_FILE_RENAME then the current node takes the lock and updates the lock on the disk. If the reason is either delete or rename, then it needs to inform other nodes that it is going to delete/rename the file. So it calls ocfs_get_x_for_del to send a message to all the nodes.

In ocfs_get_x_for_del it'll make a list of live node to whom it should inform, and then calls ocfs_send_dlm_request_msg to send the message and get the response. The keventd thread actually processes the responses (discussed below) and then invokes this thread. If for some reason the network has a problem, then ocfs_request_vote is called to initiate vote request on disk. ocfs_request_vote will make sure that no other node is requesting the vote and initiates the vote request on the disk.

ocfs_get_x_for_del calls the ocfs_wait_for_vote to read and process the responses. In ocfs_wait_for_vote, it sleeps for WAIT_FOR_VOTE_INCREMENT and then reads calls ocfs_get_vote_on_disk to read and process the vote response of all the nodes. In ocfs_get_vote_on_disk, it first reads all the nodes vote sectors, and process one by one. If the response is FLAG_VOTE_NODE then it got the vote. If the response is FLAG_VOTE_OIN_ALREADY_INUSE, then it didn't get the vote and hence needs to retry the vote request

If the reason was not delete or rename and the current node is the master, then the locktype is update and written on to the disk.

If this node is not the master of this resource, then check if the master of this resource is alive. If not alive, then it calls ocfs_recover_vol to recover the lock. If the master node is alive, then the current node should be made the master. So, it prepares the vote map and sends the vote request to all the live nodes. It then waits and process the vote responses. If any node voted FLAG_VOTE_NODE, and the reason for the request is FLAG_FILE_EXTEND or FLAG_FILE_UPDATE, then it is checked if the responding node has the file open. If open then oin_open_map is updated to keep track of nodes that have this file open. If the response is FLAG_VOTE_OIN_ALREADY_INUSE or FLAG_VOTE_UPDATE_RETRY then request needs to be retried. If the response is FLAG_VOTE_OIN_UPDATED then the node got the vote. If the response is FLAG_VOTE_FILE_DEL, then the file no longer exist and hence an error is returned.

If the node got "OK" votes from all the nodes, the lock type along with the current node as master is reflected in the header of the file entry. If any node responded to RETRY, the vote request is reinitiated

3.3.4 ocfs_acquire_lockres

This function will try to acquire the lock on the lock resource. It tries to increment in_use field if it’s not already in_use. If it’s in_use it will wait until it is marked unused or timeout

3.3.5 ocfs_release_lock

Once the lock is acquired and the work is done, then ocfs_release_lock is called to release the lock. If the lock acquired was OCFS_DLM_SHARED_LOCK lock, then no changes were made on the disk. It just resets lock_type to OCFS_DLM_NO_LOCK and decrements lr_share_cnt to indicate it is no longer using the resource.

If the lock acquired was OCFS_DLM_EXCLUSIVE_LOCK, and the reason was FLAG_FILE_UPDATE_OIN or FLAG_FILE_DELETE, then a message (locktype=OCFS_DLM_NO_LOCK) is sent to all the nodes to refresh their cache and then the lock on the disk is reset to OCFS_DLM_NO_LOCK.

3.3.6 ocfs_release_lockres

This is called to release the lock on the in memory lock resource. It decrements in_use by 1 and resets the thread_id (current process id) to zero if it’s no longer using (in_use = 0) the lock resource.

3.4 OCFS Journaling & Recovery

3.4.1 ocfs_process_record

This function actually processes the record and commits the transaction associated with it. Buffer is a pointer to a record type. It can be called on a record of either type (`ocfs_log_record` or `ocfs_cleanup_record`) as it figures out which one you're talking about by looking at the 'log_type' field in the records. Other than some variable declarations, this function is really just a

huge switch statement around the 'log_type' type field. The description of each log record in section II is what you want to look at, it describes what ocfs_process_record does in each case.

3.4.2 ocfs_recover_vol

1. make sure we're not already recovering this node and get the recovery lock
2. reset the publish sector (using ocfs_reset_publish)
3. get the file_size of the nodes cleanup file and log file
4. if they're both 0, quit
5. set osb->node_recovering to node_num and our volume state (osb->vol_state) to VOLUME_IN_RECOVERY. This way if we get called while recovering, we will quit with error (see #1). There's a potential for this to happen because we call ocfs_process_log, which in turn calls ocfs_process_record, which may call us.
6. Grab the lock on the recover log file for the node which needs recovery (this ensures nobody else in the cluster process the recovery)
7. ok, here's an interesting one. Basically we check to see if we're recovering someone elses log (if node_num != osb->node_num) and if so, we note that in OUR own recover log so that if we die, the next guy who recovers our log knows that he needs to recover the other log too. specifically we log the current transaction id, the node we were recovering, and that the operation was in fact, a LOG_TYPE_RECOVERY.
8. actually call into process_log() for the recover log now. It seems that this will abort any transactions that were in the process of being done.
9. If process_log() returned LOG_CLEANUP in our log_type, call into process_log for the cleanup log.
10. cleanup. Complete the recovery of that node set and enables our volume. (osb->vol_state = VOL_ENABLED). Drop our recovery lock, and drop the lock on the recover log file.

3.4.3 ocfs_write_log

seems to be a function where the rubber meets the road if you know
what i mean. "type" is what type of log record to write (either LOG_RECOVER or LOG_CLEANUP) "log_rec" seems to be the actual log record with all the necessary fields filled. Writes the log_rec to the appropriate log.

1 grab the nodes log lock (ocfs_down_sem(&(osb->log_lock), true))
2 calculate the size of the record (log_rec_size) and the fileid (log_file_id)
3 acquire a disk-lock on the log file
4 if the total allocated space for this file is less than the current size plus the size of the new record, allocate another megabyte for the file.
5 write the system file out to disk (ocfs_write_system_file)
6 extend the file by log_rec_size [why do we do this?]
7 drop both locks and cleanup

3.4.4 ocfs_start_trans

looks like it just sets the osb->trans_in_progess flag and puts a new number in osb->curr_trans_id (it gets this from osb->vol_node_map.largest_seq_num)

3.4.5 ocfs_commit_trans

We get passed the osb, and the transaction id, which is passed in to ocfs_process_log.
Truncate this nodes recover log to zero.

1. Process all the records in this nodes cleanup log (call ocfs_process_log on it)
2. Truncate this nodes cleanup log to zero. Shouldn't it already be zero, considering we just processed the records in it? In fact, isn't this racy as we no longer have a lock on the cleanup log? ugh...
3. Cleanup. set osb->current_trans_id = -1, and set osb->trans_in_progress to false.

3.4.6 ocfs_abort_trans

We get passed the osb, and the transaction id, which is passed in to ocfs_process_log.

1. Process all the records in the recover log (call ocfs_process_log on it)
2. Truncate this nodes recover log to zero. Same raciness as we see in commit_trans?
3. Truncate this nodes cleanup log to zero.
4. Cleanup. set osb->current_trans_id = -1, and set osb->trans_in_progress to false.

3.4.7 ocfs_process_log

"type" is the type of log (LOG_RECOVER or LOG_CLEANUP) and under certain conditions will be set to LOG_CLEANUP. trans_id is unused. really, it's not passed to anything or set or checked. Essentially this function processes every record in the logfile. If you give it a LOG_RECOVER type, it may set that to LOG_CLEANUP if the recover log is empty, probably indicating that you need to do a recovery on that log too. ocfs_recover_vol is the only function that actually checks the return val of 'type'

1. calculate the offset of the log file (log_file_id) and the size of each individual record (log_rec_size). This is actually quite straightforward and easy.
2. the resultant log_rec_size is then aligned to PAGE_SIZE. Why this isn't all just done in one step is beyond me...
3. one or mallocing our own. At this point it looks like we're overloading the ocfs_log_record variable and it could actually be an ocfs_cleanup_record. I can only hope that the prealloced one (if used) is always big enough...
4. Get the allocated size (alloc_size) and used size (file_size) of our system file (ocfs_get_system_file_size())
5. Here's some interesting stuff. If the file size is zero, then do two things:
   if *type == LOG_RECOVER, set type = *type = LOG_CLEANUP.
   quit (we're done).
6. Otherwise, the log file size is not zero and we continue.
7. If it's a recover log we're dealing with (log_type == LOG_RECOVER) then we truncate the size of the cleanup log file to zero, essentially clearing it.
8. Starting at the end of the file, take off the last record, call ocfs_process_record on it, and set the file size minus that records size. do this until the file size is zero (we've processed all our records, yay!)
9. We're done, do cleanup and return.

3.4.8 ocfs_reset_publish

read a nodes publish sector off the disk, set publish->dirty = publish->vote = publish->vote_type = 0, and write it back.

3.4.9 ocfs_get_system_file_size

given the system file id (see above to calculate this), gets the file entry associated with it and returns to you the file size and the allocated size (fe->file_size and fe->alloc_size respectively).

3.4.10 ocfs_extend_system_file

Quite simply extends the system file with id FileId to new size FileSize. If you don't have an ocfs_file_entry to pass, just give it NULL and it'll allocate and free it's own local one.
Once all parameters are checked and a proper file entry is determined, the algorithm is like so:

1. If the current allocated size is big enough to hold our new size (if FileSize <= fe->alloc_size) then just set the file entry to the new size (fe->file_size = FileSize)
2. otherwise, we find at least (FileSize - fe->alloc_size) bytes from our bitmap with ocfs_find_contiguous_space_from_bitmap (really it'll just grab an extent), allocate an extent from that area (ocfs_allocate_extent) and update the file size and alloc size (fe->file_size, fe->alloc_size respectively).
3. write the new file entry to disk. there is some bWriteThru stuff here, which I don't understand.

3.5 OCFS Module Operations

3.5.1 ocfs_driver_entry

ocfs_driver_entry is the entry point for loading ocfs module. It calls register_sysctl_table to create three sysctl variables (debug_context debug_exclude debug_level) under the /proc subsystem. These are used for controlling OCFS debugging. It calls kmem_cache_create to initialize memory slabs (oin_cache, ofile_cache, lockres_cache, fileentry_cache) under /proc/slabcache. It then creates read-only entries (version, nodename, globalctxt) under the /proc/ocfs. It calls register_filesystem and passes the ocfs_read_super structure to register OCFS filesystem.

3.5.2 ocfs_driver_exit

ocfs_driver_exit is the exit point for unloading OCFS module. It cleans up the memory structures created and initialized, cleans up the entries created under /proc subsystem and finally unregisters the filesystem

3.6 OCFS Super Operations

3.6.1 ocfs_read_super

ocfs_read_super is called during the mount of a OCFS volume. It reads the volume header structure and checks for the volume signature and other information to identify the ocfs volume. It calls ocfs_mount_volume to do more initialization and mount the volume. ocfs_mount_volume checks if any other node has already mounted this volume in Exclusive mode, if so it will error out. It calls ocfs_initialize_osb to read the publish sector and updates the timestamp. In

ocfs_initialize_osb, it creates various semaphores initializes osb structures, volume layout structure and then calls ocfs_get_config to get the config sectors. ocfs_get_config calls ocfs_chk_update_config to read the config header sectors and also the disk nodes config sectors and check if there is an available slot that can be used. If this node has been mounted before and has a slot, then that will be taken.

ocfs_get_config later calls ocfs_add_node_to_config as since it found a slot, this could be happening on another node too. So ocfs_add_node_to_config it will update the disk node config sector and calls ocfs_config_with_disk_lock. ocfs_config_with_disk_lock will first read the new node config sector and checks if any other node is owning the lock. If no other node is owning the lock then it will just try to acquire the lock. So it writes the lock into the new config structure, sleeps for a while and re-reads. If its contents are not overwritten, then it got the lock, so it calls add_timer, which starts a timer thread, which executes ocfs_assert_lock_owned to write the lock continuously to the disk until the OCFS_VOLCFG_LOCK_ITERATE jiffies. If the lock was already taken then it sleeps for a while and retries until it gets the lock.

Now that it got a node number, ocfs_mount_volume checks if the volume has been mounted before, if not then it complains if the node number is not zero. It then starts the NM thread and also starts the OCFS Listener thread if this is the first OCFS volume to be mounted on this node. It calls ocfs_vol_member_reconfig to join or form the cluster.

In ocfs_vol_member_reconfig it calls ocfs_nm_heart_beat to write the heartbeat into it's publish sectors. It then calls ocfs_create_root_oin to either create the root directory (if this is the first time the volume is mounted) and/or create the ocfs inode for this root directory. ocfs_wait_for_disk_lock_release checks if any other node has acquired the lock on the volume, if so it waits until the lock is released or time out (10000ms). Once the lock is released it calls ocfs_acquire_lock to acquire OCFS_DLM_EXCLUSIVE_LOCK lock on offset OCFS_VOLUME_LOCK_OFFSET (volume header) for FLAG_FILE_CREATE (reason). Once the lock is acquired, it reads the volume disk header and releases the lock. It then calls ocfs_create_root_dir_node to create the root directory.

In ocfs_create_root_dir_node it calls ocfs_acquire_lock to acquire OCFS_DLM_EXCLUSIVE_LOCK on volume header. It then calls ocfs_find_contiguous_space_from_bitmap to find the contiguous space of size ONE_MEGA_BYTE or cluster_size in the global bitmap. Once the space is found, it calls ocfs_allocate_file_entry to create a File Entry for this root directory. It then calls ocfs_init_system_file (8 times) to create 8 system file headers. It then calls ocfs_alloc_node_block to allocate the root directory node.

In ocfs_alloc_node_block it acquires the lock on the DirAlloc system file for this node. It then allocates some space for this file and allocates a block (root dirnode size) for the root directory. It then initializes the root dirnode structure and calls ocfs_write_dir_node to write the root directory node. It then updates the volume header with the offset to this root directory node.

Once the root directory is created and the volume header is updated, it re-reads the publish sector and calls ocfs_recover_vol if the publish sector is marked dirty during the last mount

3.6.2 ocfs_statfs

ocfs_statfs is called when the VFS needs to get filesystem statistics. This is called with the kernel lock held. It reads the ocfs bitmap lock sector, which has the number of blocks, used. It then fills the statfs structure with appropriate values and returns.

3.6.3 ocfs_put_inode

ocfs_put_inode is called when the VFS inode is removed from the inode cache. It gets the ocfs inode from the VFS inode and destroys the extent map.

3.6.4 ocfs_clear_inode

ocfs_clear_inode is called when the VFS clears the inode. If the inode is the root inode, then it calls the ocfs_dismount_volume to dismount the volume. It checks for the ofile (created for every open file) and calls ocfs_release_ofile to release the memory allocated for ofile. It then calls ocfs_extent_map_destroy to clear the extent map and clears the ocfs inode and removes the memory from oin cache.

3.6.5 ocfs_read_inode2

ocfs_read_inode2 method is called to read a specific inode from the mounted filesystem. The "i_ino" member in the "struct inode" will be initialized by the VFS to indicate which inode to read. It initializes the rest of the inode structure. It updates the inode with the pointers to the OCFS file operation methods, inode operation methods, directory operation methods depending on whether the inode is for file or directory or a link.

3.6.6 ocfs_put_super

ocfs_put_super is called when the VFS wishes to free the superblock. It just calls fsync_no_super to sync all the buffers for this device.

3.7 OCFS File Operations

3.7.1 ocfs_file_read

It calls ocfs_verify_update_oin to search the oin cache for a volume for a given filename. It reads the file header, validates the inode and updates the inode structure. If the I/O is normal (not O_DIRECT) then the vfs generic_file_write is called to write the I/Os through the page cache. It the I/O submitted is direct I/O then it calls ocfs_rw_direct, which calls ocfs_get_block2, which looks up the existing mapping of VBO to LBO for a file. The information it queries is either stored in the extent map field of the oin or is stored in the allocation file and needs to be retrieved, decoded and updated in the extent map. It traverses the extentmaps of a file, for the list of blocks that has to be read and calls brw_kiovec to do the IO. It calls multiple brw_kiovec depending on how distributed the blocks of the file are.

3.7.2 ocfs_file_write

ocfs_file_write is called to write into a file. The write could be to replace the existing contents or to append the file. If the contents need to be added, the OCFS first calls ocfs_create_modify_file

to extend the file. It then calls ocfs_rw_direct to direct IO write to the file or calls generic_file_write to do IOs through pagecache.

3.7.3 ocfs_sync_file

ocfs_sync_file calls the generic inode sync method (fsync_inode_buffers)

3.7.4 ocfs_flush

ocfs_flush calls the generic fsync_inode_buffers method

3.7.5 ocfs_file_release

ocfs_file_release is called to close the file. If it's a directory it just calls ocfs_release_ofile to release the cache allocated for ofile. If it's a file then it calls ocfs_release_ofile to release the ofile, decrements the file_open_cnt. It calls ocfs_release_cached_oin to release the oin if there are no ofiles for this file.

3.7.6 ocfs_file_open

ocfs_file_open calls ocfs_create_or_open_file which gets the inode of the parent, allocates memory to hold file header and calls ocfs_find_files_on_disk to find the given file in the dirnode structure. Once the file is found, it calls ocfs_create_oin_from_entry to allocate oin structure and update the oin.

If the file is already open then it will check if the open options conflict with the existing options (like if already opened in O_DIRECT). It will then call ocfs_allocate_ofile to allocate a new ofile structure and updates that structure.

3.8 OCFS Directory Operations

3.8.1 ocfs_readdir

ocfs_readdir is called by VFS when it needs to read the directory contents. It calls ocfs_allocate_ofile to allocate ofile structure and calls ocfs_find_files_on_disk to find the directory's fileentry in the parent dirnode structure and the updates the dirent.

3.9 OCFS Inode Operations

3.9.1 ocfs_create

called by the open(2) and creat(2) system calls. The dentry we get should not have an inode (i.e. it should be a negative dentry). It calls ocfs_create_or_open_file to open the file. It then calls new_inode to allocate VFS inode and calls ocfs_populate_reads the file header and update the inode structure.

3.9.2 ocfs_lookup

ocfs_lookup is called when the VFS needs to lookup an inode in a parent directory. It first gets the inode offset of the parent directory, and calls ocfs_find_files_on_disk to look for the filename in the dirnode structure. If the file is found then it calls ocfs_find_inode to get the inode and calls ocfs_populate_inode

3.9.3 ocfs_mkdir

calls ocfs_mknod

3.9.4 ocfs_mknod

ocfs_mknod is called by the mknod(2) system call to create a device (char, block) inode or a named pipe (FIFO) or socket. ocfs_mknod calls ocfs_create_or_open_file to create the file. ocfs_create_or_open_file it creates the file header, initializes it and calls ocfs_create_modify_file to create the direntry. It then calls ocfs_find_files_on_disk to create the dentry and calls ocfs_create_new_oin to create the inode. The inode gets initialized in ocfs_initialize_oin and later updated.

3.9.5 ocfs_rename

ocfs_rename calls ocfs_set_rename_information, which refreshes the inode of the old file checks for the file name on the disk. If the source and the target directories are different, then if the new file is found in the target directory then it is deleted (ocfs_del_file). If the new file is found then the oin is deleted (ocfs_release_cached_oin). Old FE is read (ocfs_read_file_entry) and is deleted (ocfs_del_file) and the new file entry is created (ocfs_create_file). This delete and create is done as a single transaction so that it can be recovered incase of failure.

If the source and the target directories are same, then ocfs_rename_file is called which reads the fileentry and renames the filename and writes back to the disk.

3.9.6 ocfs_setattr

ocfs_setattr is called to set the attributes (file size, create time, modify time, uid change, gid change …) of the file or directory. The inode is built if it already doesn’t exist in the memory.

3.9.7 ocfs_getattr

This is called to read the file attributes. ocfs_verify_update_oin is called to refresh the inode contents of the file or directory.

3.10 OCFS Address Space Operations

3.10.1 ocfs_readpage

ocfs_readpage calls the generic block_read_full_page method

3.10.2 ocfs_writepage

ocfs_writepage calls the generic block_write_full_page method

3.10.3 ocfs_prepare_write

ocfs_prepare_write calls the generic block_prepare_write method

3.10.4 ocfs_commit_write

ocfs_commit_write calls the generic generic_commit_write method

4. OCFS DATA STRUCTURES

4.1 ocfs_vol_disk_hdr

This is structure of the volume header that is stored in the first sector of the volume. This strucure is partially initialized during the format and partially during the first mount of the volume.

```
typedef struct _ocfs_vol_disk_hdr
{
        __u32 minor_version;
        __u32 major_version;
        __u8  signature[MAX_VOL_SIGNATURE_LEN];
        __u8  mount_point[MAX_MOUNT_POINT_LEN];
        __u64 serial_num;;
        __u64 device_size;
        __u64 start_off;
        __u64 bitmap_off;
        __u64 publ_off;
        __u64 vote_off;
        __u64 root_bitmap_off;
        __u64 data_start_off;
        __u64 root_bitmap_size;
        __u64 root_off;
        __u64 root_size;
        __u64 cluster_size
        __u64 num_nodes;
        __u64 num_clusters;
        __u64 dir_node_size;
        __u64 file_node_size;
        __u64 internal_off
        __u64 node_cfg_off
        __u64 node_cfg_size
        __u64 new_cfg_off;
        __u32 prot_bits;
        __u32 uid;
        __u32 gid;
        __s32 excl_mount
}
ocfs_vol_disk_hdr;
```

`minor_version` – minor version (2)
`major_version` – major version (1)
`signature` - "OracleCFS"
`mount_point` – this stores the mountpoint specified during the format. It can be ignored as it doesn't restrict the user from specifiying a different mountpoint. Currently this value is only used by ocfstool to get the

default mountpoint if the user didn't specify a mountpoint. This is NOT USED by mount command

serial_num – UNUSED (always 0)
device_size – size of the device/volume in bytes
start_off – UNUSED (always 0)
bitmap_off – disk offset to the Global bitmap
publ_off – disk offset to the first publish sector
vote_off – disk offset to the first vote sector
root_bitmap_off – UNUSED (always 0)
data_start_off – disk offset to the starting of the data segment
root_bitmap_size – UNUSED (always 0)
root_off – disk offset to the ocfs root dirnode (root directory)
root_size – UNUSED (always 0)
cluster_size – cluster size SELECT (4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576)
num_nodes – max # of nodes that can mount this volume, i.e., 32
num_clusters – total number of blocks or clusters on the device (device_size – data_start_off)/cluster_size
dir_node_size – UNUSED (always 0)
file_node_size – UNUSED (always 0)
internal_off – disk offset to the data segment (same as data_start_off)
node_cfg_off – offset to the first node config sector
node_cfg_size – size of the node config sector
new_cfg_off – this is the pointer to the node_cfg_hdr that is stored just before the publish sectors
prot_bits – Protection bits for user/group/others
uid – user id specified during the format of the volume
gid – group id specfied during the format of the volume
excl_mount – either contains –1 (not exclusively mounted by anyone) or the number of the node indicating that that node has mounted this volume exclusively. Once a node has mounted it exclusively, then othernodes trying to mount it would fail

4.2 ocfs_vol_label

This is the structure that is stored in the second sector of the volume. It contians the volume label information

```
typedef struct _ocfs_vol_label
{
        ocfs_disk_lock disk_lock;
        __u8 label[MAX_VOL_LABEL_LEN];
        __u16 label_len;
        __u8 vol_id[MAX_VOL_ID_LENGTH];
        __u16 vol_id_len;
        __u8 cluster_name[MAX_CLUSTER_NAME_LEN];
        __u16 cluster_name_len;
}
ocfs_vol_label;
```

disk_lock – disk structure used to synchronize access to the volume label disk structure
label – Volume label specified during the format
label_len – length of the volume
vol_id – randomly generated id during the format of the volume
vol_id_len – length of the vol_id
cluster_name – UNUSED
cluster_name_len – UNUSED

4.3 ocfs_disk_lock

This is the lock structure that is used to aqcuire a lock on any disk strcuture. The protocol is UDP

```
typedef struct _ocfs_disk_lock
{
        __u32 curr_master;
        __u8 file_lock;
        __u64 last_write_time;
        __u64 last_read_time;
        __u32 writer_node_num;
        __u32 reader_node_num;
        __u64 oin_node_map;
        __u64 dlock_seq_num;
}
ocfs_disk_lock;
```

curr_master – current node# that is owning the lock
file_lock – lock types (refer below)
last_write_time – Last time this lock is written
last_read_time – Last time this lock is read
writer_node_num – The node that has update the lock
reader_node_num – The node that has read the lock
oin_node_map – This is a 64bit number, but in this version only 32 bits are used. Each bit stands for node and that bit would be set if this node has some interest on this lock. This is how any other node knows if any other node was using this resource and needs some (recovery)
dlock_seq_num – UNUSED

Lock Types

```
#define  OCFS_DLM_NO_LOCK                (0x0)
#define  OCFS_DLM_SHARED_LOCK            (0x1)
#define  OCFS_DLM_EXCLUSIVE_LOCK         (0x2)
#define  OCFS_DLM_ENABLE_CACHE_LOCK      (0x8)
```

4.4 ocfs_bitmap_lock

This is the lock structure that is stored in 3 sector in the OCFS Header block. This structure is used to synchronize access to the Global bitmap. Any node that wants to allocate space from Global bitmap needs to acquire this lock.

```
typedef struct _ocfs_bitmap_lock
{
    ocfs_disk_lock disk_lock;
    __u32 used_bits;
}
OCFS_GCC_ATTR_PACKALGN
ocfs_bitmap_lock;
```

`disk_lock` – lock that should be acquired before modifying the global bitmap
`used_bits` – This keeps track of number of bits that are under use in Global Bitmap. Everytime a bit is used this value is incremented and everytime a bit is released, this value is decremented.

4.5 ocfs_node_config_hdr

This disk structure stores the header information of the node configuration. This is stored in the first sector of node config sectors and also a copy at the endof the node config sectors. The sector that is stored at the end of the nodeconfig sector (which is also just before the publish sector) is read by each nodes NM thread whenever it reads the publish sectors.

```
typedef struct _ocfs_node_config_hdr
{
        ocfs_disk_lock disk_lock;
        char signature[NODE_CONFIG_SIGN_LEN];
        __u32 version;
        __u32 num_nodes;
        __u32 last_node;
        __u64 cfg_seq_num;
}
OCFS_GCC_ATTR_PACKALGN
ocfs_node_config_hdr;
```

`disk_lock` – disk lock to synchronize access to this block
`signature` - "NODECFG"
`version` – This value is used to identify if any changes are made to the structure in future
`num_nodes` – number of nodes that are currently configured. This value gets incremented when a new node joins the cluster. Currently there is no way of know if a node has permanently left the cluster, so this value is not decremented.
`last_node` - UNUSED
`cfg_seq_num` – An increase in seq# indicates that there is a change in the node configuration

4.6 ocfs_disk_node_config_info

This is the disk structure that contains the node specific node configuration information. Each node is allocated a slot/sector from the node config sectors. The node will write the above structure into it's own sector

```
typedef struct _ocfs_disk_node_config_info
{
        ocfs_disk_lock disk_lock;
        char node_name[MAX_NODE_NAME_LENGTH + 1];
        ocfs_guid guid;
        ocfs_ipc_config_info ipc_config;
}
ocfs_disk_node_config_info;
```

`disk_lock` – (why do we need this)
`node_name` – node_name
`ocfs_guid guid` – guid
`ipc_config` – ipc configuration used for communicating over network

4.7 default port id

```
#define OCFS_IPC_DEFAULT_PORT   7000
```

4.8 ocfs_ipc_config_info

```
typedef struct _ocfs_ipc_config_info
{
        __u8 type;
        char ip_addr[MAX_IP_ADDR_LEN + 1];
        __u32 ip_port;
        char ip_mask[MAX_IP_ADDR_LEN + 1];
}
ocfs_ipc_config_info;
```

4.9 ocfs_publish

This is the structure that is stored in the publish sectors on disk. Each node owns a publish sector at the startup time and only the processes and nm thread (for every 500ms) on this node can write into it's sector. But the nm thread on each node can/will read other nodes publish sectors to determine the health of the other nodes as well as if they are requesting any vote in case of Disk DLM.

Access to this structure is synchronized by `osb->publish_lock` lock semaphore within the node.

```
typedef struct _ocfs_publish
{
        __u64 time;
        __s32 vote;
        bool dirty;
        __u32 vote_type;
```

```
        __u64 vote_map;
        __u64 publ_seq_num;
        __u64 dir_ent;
        __u8 hbm[OCFS_MAXIMUM_NODES];
        __u64 comm_seq_num;
}
OCFS_GCC_ATTR_PACKALGN
ocfs_publish;
```

`time` – is the counter which is written by the nm thread. This needs to be changing as long as the node is alive.
`vote` – this is the vote response
`dirty` – Set to true when we are requesting the vote
`vote_type` – This is the type of vote that we are interested in
`vote_map` – this is used to reflect the nodes that we are interested in to vote.
`publ_seq_num` – This is incremented whenever there is an interest in vote.
`dir_ent` – this contains the diskoffset (lock_id) of the resource that we are interested in for the vote.
`hbm` – This is where we store the heartbeats of the nodes
`comm_seq_num` –

4.9.1 Vote

```
#define  FLAG_VOTE_NODE                   0x1
#define  FLAG_VOTE_OIN_UPDATED            0x2
#define  FLAG_VOTE_OIN_ALREADY_INUSE      0x4
#define  FLAG_VOTE_UPDATE_RETRY           0x8
#define  FLAG_VOTE_FILE_DEL               0x10
```

4.9.2 Vote_types

```
#define  FLAG_FILE_CREATE           0x1
#define  FLAG_FILE_EXTEND           0x2
#define  FLAG_FILE_DELETE           0x4
#define  FLAG_FILE_RENAME           0x8
#define  FLAG_FILE_UPDATE           0x10
#define  FLAG_FILE_CREATE_DIR       0x40
#define  FLAG_FILE_UPDATE_OIN       0x80
#define  FLAG_FILE_RELEASE_MASTER   0x100
#define  FLAG_CHANGE_MASTER         0x400
#define  FLAG_ADD_OIN_MAP           0x800
#define  FLAG_DIR                   0x1000
#define  FLAG_DEL_NAME              0x20000
#define  FLAG_RESET_VALID           0x40000
#define  FLAG_FILE_RELEASE_CACHE    0x400000
#define  FLAG_FILE_CREATE_CDSL      0x800000  - UNUSED
#define  FLAG_FILE_DELETE_CDSL      0x1000000 - UNUSED
#define  FLAG_FILE_CHANGE_TO_CDSL   0x4000000 - UNUSED
#define  FLAG_FILE_TRUNCATE         0x8000000
```

4.10 ocfs_vote

This is the structure that is written to the vote sectors. Each node owns a vote sector on the disk and only it can write into that sector. This is written always in response for a vote request. The node that requested the vote will read all the other nodes vote sectors and analyze their vote responses.

```
typedef struct _ocfs_vote
{
        __u8 vote[OCFS_MAXIMUM_NODES];
        __u64 vote_seq_num;
        __u64 dir_ent;
        __u8 open_handle;
}
OCFS_GCC_ATTR_PACKALGN
ocfs_vote;
```

`vote` – used to put our vote
`vote_seq_num` – this is the publ_seq_num for which we are responding
`dir_ent` – this is the offset (lock_id) of the resource for which we are voting.
`open_handle` – to indicate if we have the inode open

4.11 ocfs_dir_node

ocfs_dir_node is the structure to hold the information about the directories. Each directory has one corresponding ocfs_dir_node structure created and is stored in it's parents ocfs_dir_node structure. The size of this structure is 128Kb and can hold upto 254 entries (file or dir entries). When the directory grows beyond 254 entries then another ocfs_dir_node is created and link is created between these two.

Any node that is modifying the metadata info needs to acquire a lock on this ocfs_dir_node offset.

```
typedef struct _ocfs_dir_node
{
        ocfs_disk_lock disk_lock;
        __u8 signature[8];
        __u64 alloc_file_off;
        __u32 alloc_node;
        __u64 free_node_ptr;
        __u64 node_disk_off;
        __s64 next_node_ptr;
        __s64 indx_node_ptr;
        __s64 next_del_ent_node;
        __s64 head_del_ent_node;
        __u8 first_del;
        __u8 num_del;
        __u8 num_ents;
        __u8 depth;
        __u8 num_ent_used;
```

```
        __u8 dir_node_flags;
        __u8 sync_flags;
        __u8 index[256];
        __u8 index_dirty;
        __u8 bad_off;
        __u8 reserved[127];
        __u8 file_ent[1];
}
OCFS_GCC_ATTR_PACKALGN
ocfs_dir_node;
```

`disk_lock` – holds the information about the node that is locking
`signature` – used for sanity checking. Stores "DIRNV20" in it.
`alloc_file_off` – contains the offset value within the directory system file.
`alloc_node` – node that created the directory
`free_node_ptr` – contains the offset of the last ocfs_dir_node
`node_disk_off` – disk offset for the dirnode
`next_node_ptr` – pointer to next ocfs_dir_entry for this directory (incase we had created more than 254 files/directories)
`indx_node_ptr` – UNUSED
`next_del_ent_node` – Currently UNUSED
`head_del_ent_node` – points to the
`first_del` – pointer to the index that was recently deleted. This is how we track the deleted FEs. When a FE is deleted, first_del will point to the deleted FE, now if another FE is deleted then we point to the newly deleted FE and that FE will point to the previously deleted FE.
`num_del` – gets incremented when a file is deleted and decrements when that FE slot is reused
`num_ents` – Total #of entries(file/directory) that this structure can accommodate.
`Depth` – UNUSED
`num_ent_used` – # of entries that are currently being used
`dir_node_flags` – (this is always set DIR_NODE_FLAG_ROOT)
`sync_flags` – UNUSED
`index[256]` – stores sorted offsets to the files or directories within this directory
`index_dirty` – When a new file is added then the above index is sorted based on the name. But if a file is renamed, then we simply set this to TRUE indicating that we need to sort the index
`bad_off` – contains the offset to the file that got renamed
`reserved` –
`file_ent` –

4.12 dir_node_flags

```
#define  DIR_NODE_FLAG_ROOT          0x1
```

4.13 ocfs_file_entry

ocfs_file_entry is created for each file that is created. This strcucture contains the information about the file and is stored in ocfs_dir_node structure of the directory under which this file has got created. Any node that wants to access a file should get a lock on that file(should a node get a lock for accessing the data).

```
typedef struct _ocfs_file_entry
{
        ocfs_disk_lock disk_lock;
        __u8 signature[8];
        bool local_ext;
        __u8 next_free_ext;
        __s8 next_del;
        __s32 granularity;
        __u8 filename[OCFS_MAX_FILENAME_LENGTH];
        __u16 filename_len;
        __u64 file_size;
        __u64 alloc_size;
        __u64 create_time;
        __u64 modify_time;
        ocfs_alloc_ext extents[OCFS_MAX_FILE_ENTRY_EXTENTS]
        __u64 dir_node_ptr;
        __u64 this_sector;
        __u64 last_ext_ptr;
        __u32 sync_flags;
        __u32 link_cnt;
        __u32 attribs;
        __u32 prot_bits;
        __u32 uid;
        __u32 gid;
        __u16 dev_major;
        __u16 dev_minor;
        /* 32-bit: sizeof(fe) = 484 bytes */
        /* 64-bit: sizeof(fe) = 488 bytes */
}
ocfs_file_entry;
```

disk_lock	– If a file is locked, then this is where the lock info is stored
signature[8]	– It stores "FIL" to identify that this is a FE
local_ext	– When this is true we are still using local_extents
next_free_ext	– This is used in combination with granularity and local_ext to find which would be the next ext that should be created when extending the file.
next_del	– This value is set when a file is deleted. If this is the first file in the ocfs_dir_node to be deleted then this will be set to –1 if not it will point to the slot of the previous FE of the file that was deleted before us.
granularity	– this is to indicate the level of extents
file_size	– This contains the actual size of the file in bytes
alloc_size	– space allocation is always in terms of extents, so this will indicate how much space is reserved/used for this file

```
create_time    – currently this value can be ignored
modify_time    – time when the last modification happened to the metadata of the file
extents        – It contians the pointers to either the data extents or to the extent groups
                 depeneding on the granularity
dir_node_ptr   – If points to the dirnode that is holding us (parent directory)
this_sector    – It's the physical offset at which this FE is stored
last_ext_ptr   – this is the pointer to the last extent
sync_flags     – indicates the status of the file
link_cnt       – UNUSED
attribs        – to indicate the filetype (see below)
prot_bits      – user/group/other permissions on the file
uid            – user id owning the file
gid            – group id owning the file
dev_major      – major # of the device on which the file is created
dev_minor      – minor # of the device on which the file is created
```

```
sync_flags
/*
** File Entry contains this information
*/
#define  OCFS_SYNC_FLAG_DELETED              (0)
#define  OCFS_SYNC_FLAG_VALID                (0x1)
#define  OCFS_SYNC_FLAG_CHANGE               (0x2)
#define  OCFS_SYNC_FLAG_MARK_FOR_DELETION    (0x4)
#define  OCFS_SYNC_FLAG_NAME_DELETED         (0x8)

attribs
#define  OCFS_ATTRIB_DIRECTORY               (0x1)
#define  OCFS_ATTRIB_FILE_CDSL               (0x8)
#define  OCFS_ATTRIB_CHAR                    (0x10)
#define  OCFS_ATTRIB_BLOCK                   (0x20)
#define  OCFS_ATTRIB_REG                     (0x40)
#define  OCFS_ATTRIB_FIFO                    (0x80)
#define  OCFS_ATTRIB_SYMLINK                 (0x100)
#define  OCFS_ATTRIB_SOCKET                  (0x200)
```

4.14 ocfs_extent_group

When a file needs additional space an extent is allocated. If the allocated new extent is not adjacent to the existing extents, then we need to add an entry into the FE. But currently FE can hold only upto 3 extents, so if we are trying allocate more than that then ocfs_extent_group comes into picture.

This structure can hold pointers upto to 18 extents and the extent entries in the FE will point to this extent group. The space for ocfs_extent_group is accounted in a special system files called OCFS_FILE_FILE_ALLOC and OCFS_FILE_FILE_ALLOC_BITMAP. The size of the ocfs_extent_group would be ~512bytes

```
typedef struct _ocfs_extent_group
```

```
{
        __u8 signature[8];
        __s32 next_free_ext;
        __u32 curr_sect;
        __u32 max_sects;
        __u32 type;
        __s32 granularity;
        __u32 alloc_node;
        __u64 this_ext;
        __u64 next_data_ext;
        __u64 alloc_file_off;
        __u64 last_ext_ptr;
        __u64 up_hdr_node_ptr;
        ocfs_alloc_ext extents[OCFS_MAX_DATA_EXTENTS];
}
ocfs_extent_group;
```

`Signature` – contains "EXTHDR2"
`next_free_ext` – # of the next free extent
curr_sect – (logica/physical)offset to our extent group structure
`max_sects` -
`type` – This is to indicate if the extent is pointing to the data blocks or to the next level extents.
`granularity` -
`alloc_node` – node that allocated the extent group, this way we can track back which system file is accounting the space allocated for storing the ocfs_extent_group structure.
`this_ext` – disk offset to this extent block
`next_data_ext` – disk offset of another extent map at the same level allocated for the same file.
`alloc_file_off` – disk offset into the system file from where we got the space. Used in combination with alloc_node
`last_ext_ptr` – UNUSED
`up_hdr_node_ptr` – offset to the parent Extent block or FE
`ocfs_alloc_ext extents` – contains info about the extents

4.15 Ocfs_global_ctxt

Ocfs Global context is an inmemory structure created during the insmod of the ocfs module. This is a global structure and there is only one structure per node. This structure contains pointers to the global ocfs caches (oin, ofile, fe, and lockres). It also keeps track of all the OCFS superblocks that are created for each ocfs volume that is mounted.

```
typedef struct _ocfs_global_ctxt
{
        ocfs_obj_id obj_id;
        ocfs_sem res;
        struct list_head osb_next;
        kmem_cache_t *oin_cache;
```

```
        kmem_cache_t *ofile_cache;
        kmem_cache_t *fe_cache;
        kmem_cache_t *lockres_cache;
        __u32 flags;
        __u32 pref_node_num;
        ocfs_guid guid;
        char *node_name;
        char *cluster_name;
        ocfs_comm_info comm_info;
        bool comm_info_read;
        wait_queue_head_t flush_event;
        __u8 hbm;
        spinlock_t comm_seq_lock;
        __u64 comm_seq_num;
#ifdef OCFS_LINUX_MEM_DEBUG
        struct list_head item_list;
#endif
        atomic_t cnt_lockres
}
ocfs_global_ctxt;
```

obj_id – unique id
res – used to synchronize access to this structure
osb_next – list that keeps track of osbs
oin_cache – kernel cache that is created to store oins
ofile_cache – kernel cache for storing the files (for every file that is open we create a ofile structure)
fe_cache – cache for allocate FEs
lockres_cache – cache for allocating lockres
flags – flags to indicate the state of Global structure initialization
pref_node_num – prefered node number (specified in the ocfs.conf)
node_name – node name
cluster_name – UNUSED
comm_info – used to store ipc info for this node
comm_info_read - UNUSED
flush_event – UNUSED
hbm – UNUSED
comm_seq_lock – Synchronizes access to comm_seq_num
comm_seq_num – This is incremented everytime a dlm message is sent
cnt_lockres – counter that keeps track of lockresources

4.16 Ocfs_super

A mounted volume is represented using this structure. This structure is created one of each volume mounted on that node. Ocfs_global_context structure will maintain a linked list which contains these structures. If a resource on a particular volume is accessed then it will be through this structure.

```
struct _ocfs_super
```

```
{
        ocfs_obj_id obj_id;
        ocfs_sem osb_res;
        struct list_head osb_next;
        __u32 osb_id;
        struct completion complete;
        struct task_struct *dlm_task;
        __u32 osb_flags;
        __s64 file_open_cnt;
        __u64 publ_map;
        HASHTABLE root_sect_node;
        struct list_head cache_lock_list;
        struct super_block *sb;
        ocfs_inode *oin_root_dir;
        ocfs_vol_layout vol_layout;
        ocfs_vol_node_map vol_node_map;
        ocfs_node_config_info
                      *node_cfg_info[OCFS_MAXIMUM_NODES];
        __u64 cfg_seq_num;
        bool cfg_initialized;
        __u32 num_cfg_nodes;
        __u32 node_num;
        bool reclaim_id;
        __u8 hbm;
        __u32 hbt;
        __u64 log_disk_off;
        __u64 log_meta_disk_off;
        __u64 log_file_size;
        __u32 sect_size;
        bool needs_flush;
        bool commit_cache_exec;
        ocfs_sem map_lock;
        ocfs_extent_map metadata_map;
        ocfs_extent_map trans_map;
        ocfs_alloc_bm cluster_bitmap;
        __u32 max_dir_node_ent;
        ocfs_vol_state vol_state;
        __s64 curr_trans_id;
        bool trans_in_progress;
        ocfs_sem log_lock;
        ocfs_sem recovery_lock;
        __u32 node_recovering;
#ifdef PARANOID_LOCKS
        ocfs_sem dir_alloc_lock;
        ocfs_sem file_alloc_lock;
#endif
        ocfs_sem vol_alloc_lock;
        struct timer_list lock_timer;
        atomic_t lock_stop;
        wait_queue_head_t lock_event;
        atomic_t lock_event_woken;
        struct semaphore comm_lock;
```

```
        atomic_t nm_init;
        wait_queue_head_t nm_init_event;
        bool cache_fs;
        __u32 prealloc_lock;
        ocfs_io_runs *data_prealloc;
        ocfs_io_runs *md_prealloc;
        __u8 *cfg_prealloc;
        __u32 cfg_len;
        __u8 *log_prealloc;
        struct semaphore publish_lock;
        atomic_t node_req_vote;
        struct semaphore trans_lock;
};
```

`obj_id` – 8byte structure stores unique id `(0x05534643)`
`osb_res` – resource to synchronize access to ocfs_super
`osb_next` – list of ocfs_super(s)
`osb_id` – This is our value under which we exist under /proc/ocfs (e.g /proc/ocfs/0). There is a global variable that keeps track of the next value that can be used by the next ocfs volume that is getting mounted.
`Complete` – Used to ensure that NM thread is initalized/killed
`dlm_task` – this is used to store the task_struct of the nm thread. This is used to make sure that the nm thread is exited before we say we succesfully dismounted
`osb_flags` – this is used to track the progress of OSB
`file_open_cnt` – num of open files/dirs.
`publ_map` – each bit represents state of node (0 dead, 1 alive)
`root_sect_node` – This is the hashtable for lockres
`cache_lock_list` – UNUSED
`super_block *sb` – pointer to the super block
`oin_root_dir` – ptr to the root dir ocfs_inode
`vol_layout` – contains the inmemory volume header
`vol_node_map` – Used for heart beat
`node_cfg_info` – pointer to the node config structures
`cfg_seq_num` – stores the config seq number used
`cfg_initialized` – used to check if the node config structure is initialized
`num_cfg_nodes` – number of nodes that have been configured
`node_num` – our nodenumber for this volume
`reclaim_id` – did the user specified to reclaim his original node number. This is used only at the mount time
`hbm` – UNUSED
`Hbt` – UNUSED
`log_disk_off` – UNUSED
`log_meta_disk_off` – UNUSED
`log_file_size` – UNUSED
`sect_size` – sector size (512 bytes)
`needs_flush` – used to indicate if the data needs to be flushed
`commit_cache_exec` –

```
map_lock           -  UNUSED
metadata_map       -  UNUSED
trans_map          -  UNUSED
cluster_bitmap     –  Global bitmap initalized during mount
max_dir_node_ent   –  maximum dirnode entries (254)
vol_state          –  used to indicate current state of the volume (refer below)
curr_trans_id      –  current transaction id
trans_in_progress  –  set to true if a transaction is initiated
log_lock           –  used to synchronize access to recovery and cleanup log file
recovery_lock      –  used to ensure that only one process is doing recovery
node_recovering    –  used to indicate which node is being recovered
#ifdef PARANOID_LOCKS
        ocfs_sem dir_alloc_lock; - To syncronize access to dirbitmap
        ocfs_sem file_alloc_lock;- To syncronize access to extent bitmap
#endif
vol_alloc_lock     –  To synchronize access to Global Bitmap
lock_timer         -  Event Timer -  Used for node configuration
lock_stop          -  When to stop -  Used for node configuration
lock_event         –  Event (Used for node configuration)
lock_event_woken   -
comm_lock          –  protects ocfs_comm_process_vote_reply
nm_init            –  this is on which the mount thread waits for the nm thread to get
                      initialized before we could continue
nm_init_event      –  this is the event on which the mount thread waits
cache_fs           –  UNUSED
prealloc_lock      –  this is where we specifiy if we have preallocated memory for data,
                      md,and lock blocks
data_prealloc      –  memory preallocated to use for storing data io runs
md_prealloc        –  memory preallocated for reuse for storing metadata io runs
cfg_prealloc       –  memory preallocated for reading cfg info by NM thread
log_prealloc       –  memory preallocated for log records
publish_lock       –  protects r/w to publish sector
node_req_vote      –  set when node's vote req pending
trans_lock         –  serializes transactions
```

OSB STATUS FLAGS

```
#define  OCFS_OSB_FLAGS_BEING_DISMOUNTED  (0x00000004)
#define  OCFS_OSB_FLAGS_SHUTDOWN          (0x00000008)
#define  OCFS_OSB_FLAGS_OSB_INITIALIZED   (0x00000020)
```

4.17 ocfs_vol_state

The above states indicates a state that a volume can be.

```
typedef enum _ocfs_vol_state
{
        VOLUME_DISABLED,
        VOLUME_INIT,
```

```
        VOLUME_ENABLED,
        VOLUME_LOCKED,
        VOLUME_IN_RECOVERY,
        VOLUME_MOUNTED,
        VOLUME_BEING_DISMOUNTED,
        VOLUME_DISMOUNTED
}
ocfs_vol_state;
```

4.18 ocfs_vol_layout

This is the in-memory structure of the volume header and is stored in OCFS super block. It contains the volume header information and pointers (offsets) to other disk structures like node config block, bitmap block, … this structure is initialized during the mount of the volume, and is one per volume per node.

```
typedef struct _ocfs_vol_layout
{
        __u64 start_off;
        __u32 num_nodes;
        __u32 cluster_size;
        __u8 mount_point[MAX_MOUNT_POINT_LEN];
        __u8 vol_id[MAX_VOL_ID_LENGTH];
        __u8 label[MAX_VOL_LABEL_LEN];
        __u32 label_len;
        __u64 size;
        __u64 root_start_off;
        __u64 serial_num;
        __u64 root_size;
        __u64 publ_sect_off;
        __u64 vote_sect_off;
        __u64 root_bitmap_off;
        __u64 root_bitmap_size;
        __u64 data_start_off;
        __u64 num_clusters;
        __u64 root_int_off;
        __u64 dir_node_size;
        __u64 file_node_size;
        __u64 bitmap_off;
        __u64 node_cfg_off;
        __u64 node_cfg_size;
        __u64 new_cfg_off;
        __u32 prot_bits;
        __u32 uid;
        __u32 gid;
}
ocfs_vol_layout;
```

`start_off` – UNUSED

`num_nodes` – max number of nodes that can mount this volume (32)

cluster_size – size of the cluster (or block).
mount_point – this is mount point specified at the time of format (used by ocfs tool only)
vol_id – Some random id (This is unique for each mounted volume)
label – volume label that is specified during format
label_len – length of the volume
size – of the volume/device
root_start_off – Disk offset to the root director node
serial_num – UNUSED
root_size – UNUSED
publ_sect_off – offset to the public sector
vote_sect_off – offset to the vote sector
root_bitmap_off – UNUSED
root_bitmap_size – UNUSED
data_start_off – offset to the sector after free sectors
num_clusters – total number of clusters/blocks
root_int_off – offset to the data start off
dir_node_size – DIRECTORY NODE SIZE (128K)
file_node_size – FILE ENTRY SIZE (512 BYTES)
bitmap_off – offset to the bitmap block
node_cfg_off – offset to the nodeconfig block
node_cfg_size – size of the nodeconfig block
new_cfg_off – Pointer to the 35th sector in the node config sectors
prot_bits – Protection bits (User/group/other priveliges)
uid – userid specified during the format
gid – groupid specified during the format

4.19 ocfs_inode

ocfs_inode is the inmemory structure that keeps track of a file or a directory. An inode structure is created for each of the file/directory while it is first referenced. The memory for this structure is allocated from the oin_cache.

```
struct _ocfs_inode
{
        ocfs_obj_id obj_id;
        __s64 alloc_size;
        struct inode *inode;
        ocfs_sem main_res;
        ocfs_sem paging_io_res;
        ocfs_lock_res *lock_res;
        __u64 file_disk_off;
        __u64 dir_disk_off;
        __u64 chng_seq_num;
        __u64 parent_dirnode_off;
        ocfs_extent_map map;
        struct _ocfs_super *osb;
        __u32 oin_flags;
```

```
        struct list_head next_ofile;
        __u32 open_hndl_cnt;
        bool needs_verification;
        bool cache_enabled;
};
```

obj_id – that uniquely identifies this inode
alloc_size – File Entry allocation size
inode – pointer to the Linux inode
main_res – Protects the resource
paging_io_res – UNUSED
lock_res – pointer to the inmemory lock resource
file_disk_off – disk offset to the file that this inode is representing
dir_disk_off – disk offset to the directory(or parent) that this inode is representing
chng_seq_num – UNUSED
parent_dirnode_off – disk offset to the parent of this directory
map – points to the structures that holds the extent maps
osb – pointer to the ocfs super block
oin_flags – indicates the state of the inode
next_ofile – pointer to the next ofile entry
open_hndl_cnt – # of handles opened for this resource
needs_verification – UNUSED
cache_enabled – UNUSED

INODE STATUS

```
#define  OCFS_OIN_IN_TEARDOWN                (0x00000002)
#define  OCFS_OIN_DIRECTORY                  (0x00000008)
#define  OCFS_OIN_ROOT_DIRECTORY             (0x00000010)
#define  OCFS_OIN_CACHE_UPDATE               (0x00000100)
#define  OCFS_OIN_DELETE_ON_CLOSE            (0x00000200)
#define  OCFS_OIN_NEEDS_DELETION             (0x00000400)
#define  OCFS_INITIALIZED_MAIN_RESOURCE      (0x00002000)
#define  OCFS_INITIALIZED_PAGING_IO_RESOURCE (0x00004000)
#define  OCFS_OIN_INVALID                    (0x00008000)
#define  OCFS_OIN_IN_USE                     (0x00020000)
#define  OCFS_OIN_OPEN_FOR_DIRECTIO          (0x00100000)
#define  OCFS_OIN_OPEN_FOR_WRITE             (0x00200000)
```

4.20 Structures to define entities under /proc filesystem

```
static ctl_table ocfs_dbg_table[] = {
        {1, "debug_level", &debug_level, sizeof (__u32), 0644, NULL,
                  &proc_dointvec, &sysctl_intvec, NULL, NULL, NULL},
        {2, "debug_context", &debug_context, sizeof (__u32), 0644,
            NULL, &proc_dointvec, &sysctl_intvec, NULL, NULL, NULL},
        {3, "debug_exclude", &debug_exclude, sizeof (__u32), 0644,
            NULL, &proc_dointvec, &sysctl_intvec, NULL, NULL, NULL},
```

```
        {0}
};

static ctl_table ocfs_kern_table[] = {
        {KERN_OCFS, "ocfs", NULL, 0, 0555, ocfs_dbg_table},
        {0}
};
static ctl_table ocfs_root_table[] = {
                {CTL_KERN, "kernel", NULL, 0, 0555, ocfs_kern_table},
                {0}
};
static struct ctl_table_header *ocfs_table_header = NULL;
```

4.21 System Files

System files are special purpose files (created and managed similar to normal files). Each files has a special purpose as defined below.

```
#define  OCFS_DIR_FILENAME          "DirFile"
#define  OCFS_DIR_BITMAP_FILENAME   "DirBitMapFile"
#define  OCFS_FILE_EXTENT_FILENAME "ExtentFile"
#define  OCFS_FILE_EXTENT_BITMAP_FILENAME
                                             "ExtentBitMapFile"
#define  OCFS_RECOVER_LOG_FILENAME "RecoverLogFile"
#define  OCFS_CLEANUP_LOG_FILENAME "CleanUpLogFile"
```

DirFile – stores the offsets to the blocks (clusters) that are allocated for creating directories. Any ocfs_dir_node structure has to reside in these blocks only
DirBitMapFile – This file is used to keep track of space that is allocate and free withing the space allocated for ocfs_dir_nodes.
ExtentFile – This file keeps tracks of offsets that got allocated for extent maps.
ExtentBitMapFile – This keeps track of allocated and free space in the above ExtentFile blocks.
RecoverLogFile – keeps track of recovery stuff
CleanUpLogFile – keeps track of cleanup stuff/roleforward stuff

OCFS - Oracle Clustered Filesystem for Linux
Physical Design & Implementation.

December 2003

Author: Srinivas Eeda
Contributing Authors: Wim Coekaerts, Kurt Hackel, Sunil Mushran and Mark Fasheh.

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

For information about the Oracle Open Source, visit us at http://oss.oracle.com

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Copyright © 2003 Oracle Corporation
All rights reserved