15

The UFS File System

Updated by Frank Batschulat, Shawn Debnath, Sarah Jelinek, Dworkin Muller, and Karen Rochford

The UFS file system is the general-purpose, disk-based file system that is shipped with Solaris today and has been the default file system since early versions of SunOS 4.x. For over 20 years, UFS has undergone extensive changes to keep pace with the performance, security, and reliability requirements of today's modern enterprise applications.

15.1 UFS Development History

The original version of UFS is derived from the Berkeley Fast File System (FFS) work from BSD UNIX, architected by Marshall Kirk McKusick and Bill Joy in the mid 1980s. The Berkeley FFS was the second major file system available for UNIX and was a leap forward from the original System V file system. The System V file system was lightweight and simple but had significant shortcomings: poor performance, unreliability, and lack of functionality.

During the development of Sun OS 2.0, a file-system-independent interface was introduced to support concurrent, different file systems within an operating system instance. This interface, today known as the `vnode`/`vfs` interface, is the mechanism that all file systems use to interface with the file-related system calls. (The `vnode`/`vfs` architecture is discussed further in Section 14.6.) UFS was modified so that it could be used within this new `vnode`/`vfs` framework and since has been the focus of much of the file system development effort in Solaris.

A second major overhaul for UFS came about at the time of SunOS 4.0, when the virtual memory (VM) system was redeveloped to use the `vnode` as the core of virtual memory operations. The new VM system implemented the concept of virtual file caching—a departure from the traditional physical file cache (known as the "buffer cache" in previous versions of UNIX). The old buffer cache was layered under the file systems and was responsible for caching physical blocks from the file system to the storage device. The new model is layered above the file systems and allows the VM system to act as a cache for files rather than blocks. The new system caches page-sized pieces of files, whereby the file and a particular offset are cached as pages of memory. From this point forward, the buffer cache was used only for file system metadata, and the VM system implemented the file system caching. The introduction of the virtual file caching affected file systems in many ways and required significant changes to the `vnode` interface. At that point, UFS was substantially modified to support the new `vnode` and VM interfaces.

The third major change to UFS came about in Solaris 2.4 in the year 1994 with the introduction of file system metadata logging in an effort to provide better reliability and faster reboot times after a system crash or outage. The first versions of logging were introduced with the unbundled Online: DiskSuite 3.0 software package, the precursor to Solstice DiskSuite (SDS) product and the Solaris Volume Manager (SVM) as it is known today. Solaris 7 saw the integration of logging into UFS, and after six years of development, Solaris 10 shipped with logging turned on by default. Table 15.1 summarizes the major UFS development milestones.

Table 15.1 UNIX File System Evolution

1984	SunOS 1.0	FFS from 4.2 BSD.
1985	SunOS 2.0	UFS rearchitected to support vnodes/`vfs`.
1988	SunOS 4.0	UFS integrated with new VM virtual file cache.
1991	SunOS 4.1	I/O clustering added to allow extentlike performance.
1992	SunOS 4.1	1TB file system and ability to grow UFS file systems with Online: Disk Suite 1.0.
1992	Solaris 2.0	1TB file system support included in base Solaris.
1994	Solaris 2.4	Metadata logging option with Online: DiskSuite 3.0.
1995	Solaris 2.5	Access Control Lists.
1995	Solaris 2.6	Large file support allows 1TB files. Direct I/O uncached access added.
1998	Solaris 7	Metadata logging integrated into base Solaris UFS.
2002	Solaris 9	File System Snapshots Extended Attributes
2003	Solaris 9 Update 4	Multi-terabyte UFS support was added.
2004	Solaris 10 and Solaris 9 Update 7	Logging on by default in UFS.

15.2 UFS On-Disk Format

UFS is built around the concept of a disk's geometry, which is described as the number of sectors in a track, the location of the head, and the number of tracks. UFS uses a hybrid block allocation strategy that allocates full blocks or smaller parts of the block called fragments. A block is a set of contigous fragments starting on a particular boundary. This boundary is determined by the size of a fragment and the number of fragments that constitute a block. For example, fragment 32 and block 32 both relate to the same physical location on disk. Although the next fragment on disk is 33 followed by 34, 35, 36, 37 and so on, the next block is at 40, which begins on fragment 40. This is true in the case of 8-Kbyte block size and 1-Kbyte fragment size, where 8 fragments constitutes a file system block.

15.2.1 On-Disk UFS Inodes

In UFS, all information pertaining to a file is stored in a special file index node called the inode (except for the name of the file, which is stored in the directory). There are two types of inodes: in-core and on-disk. The on-disk inodes, as the name implies, reside on disk, whereas the in-core inode is created only when a particular file is opened for reading or writing.

The on-disk inode is represented by `struct icommon`. It occupies exactly 128 bytes on disk and can also be found embedded in the in-core inode structure, as shown in Figure 15.1.

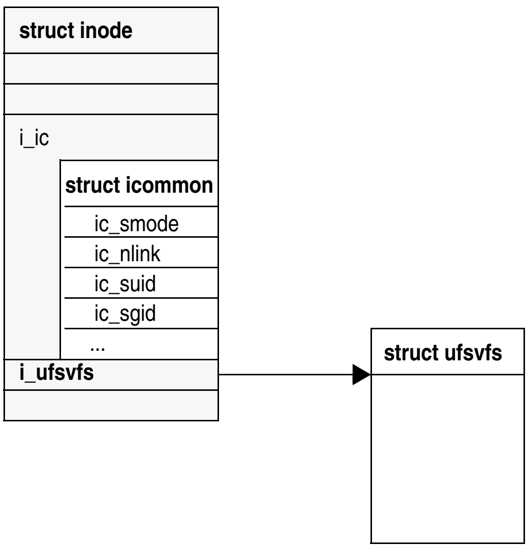

Figure 15.1 Embedded On-Disk in In-Core Inode

The structure of `icommon` looks like this.

```
struct  icommon {
        o_mode_t ic_smode;      /*  0: mode and type of file */
        short   ic_nlink;       /*  2: number of links to file */
        o_uid_t ic_suid;        /*  4: owner's user id */
        o_gid_t ic_sgid;        /*  6: owner's group id */
        u_offset_t ic_lsize;    /*  8: number of bytes in file */
#ifdef _KERNEL
        struct timeval32 ic_atime;      /* 16: time last accessed */
        struct timeval32 ic_mtime;      /* 24: time last modified */
        struct timeval32 ic_ctime;      /* 32: last time inode changed */
#else
        time32_t ic_atime;      /* 16: time last accessed */
        int32_t ic_atspare;
        time32_t ic_mtime;      /* 24: time last modified */
        int32_t ic_mtspare;
        time32_t ic_ctime;      /* 32: last time inode changed */
        int32_t ic_ctspare;
#endif
        daddr32_t       ic_db[NDADDR];  /* 40: disk block addresses */
        daddr32_t       ic_ib[NIADDR];  /* 88: indirect blocks */
        int32_t ic_flags;       /* 100: cflags */
        int32_t ic_blocks;      /* 104: 512 byte blocks actually held */
        int32_t ic_gen;         /* 108: generation number */
        int32_t ic_shadow;      /* 112: shadow inode */
        uid_t   ic_uid;         /* 116: long EFT version of uid */
        gid_t   ic_gid;         /* 120: long EFT version of gid */
        uint32_t ic_oeftflag;   /* 124: extended attr directory ino, 0 = none */
};
```

See usr/src/uts/common/sys/fs/ufs_inode.h

Most of the fields are self-explaining, but a couple of them need a bit of help:

- **`ic_smode`.** Indicates the type of inode. There are primarily four main types of inode: zero, special node (`IFCHR`, `IFBLK`, `IFIFO`, `IFSOCK`), symbolic link (`IFLNK`), a directory (`IFDIR`), a file (`IFREG`), or an extended metadata inode (IFSHAD, IFATTRDIR). Type zero indicates that the inode is not in use and `ic_nlink` should be zero, unless logging's `reclaim_needed` flag is set. With the special nodes, no data blocks are associated. They are used for character and block devices, pipes and sockets. The type file indicates where this inode is a directory, a regular file, a shadow inode, or an extended attribute directory.
- **`ic_nlink`.** Refers to the number of links to a file, that is, the number of names in the namespace that correspond to a specific file identifier. A regular file will have link count of 1 because only one name in the namespace corresponds to that particular file identifier. A directory link count has the value

2 by default: one is the name of the directory itself, and the other is the "." entry within the directory. Any subdirectory within a directory causes the link count to be incremented by 1 because of the ".." entry. The limit is 32,767 and hence, the limit for the number of subdirectories is 32,765 and also the total number of links. The ".." entry counts against the parent directory only.

- **`ic_db`.** Is an array that holds 12 pointers to data blocks. These are called the direct blocks. On a system with block size of 8192 bytes or 8 Kbytes, these can accommodate up to 98,304 bytes or 96 Kbytes. If the file consists entirely of direct blocks, then the last block for the file (not the last `ic_db` entry) may contain fragments. Note that if the file size exceeds the capacity of the `ic_db` array, then the block list for the file must consist entirely of full-sized file system blocks.
- **`ic_ib`.** Is a small array of only three pointers but allows a file to be up to one terabyte. How does this work? Well, the first entry in `ic_ib` points to a block that stores 2048 block addresses. A file with a single indirect block can accommodate up to 8192 * (12 + 2048) bytes or 16 Mbytes. If more storage is required, another level of indirection is added and the second indirect block is used.

  The second entry in `ic_ib` points to 2048 block addresses, and each of those 2048 entries points to another block containing 2048 entries that finally point to the data blocks. With two levels of indirection, a file can accommodate up to 8192 * 12 + 2048 + (2048 * 2048) bytes, or 32 Gbytes. A third level of indirection permits the file to be 8192 * 12 + 2048 + (2048 * 2048) + (2048 * 2048 * 2048) = 70,403,120,791,552 bytes long or—yes, you guessed it—64 Tbytes! However, since all addresses must be addressable as fragments, that is, a 31-bit count, the maximum is 2TB (2^31 * 1KB). Multi-terrabyte UFS (MTBUFS) enables 16TB filesystem sizes by enforcing the minimum frag size to be 8K, which gives you 2^31 * 2^10 * 8k, or 16 TB.

  Figure 15.2 illustrates the layout.
- **`ic_shadow`.** If non-zero, contains the number of an inode providing shadow metadata (usually, this data would be ACLs).
- **`ic_oeftflag`.** If non-zero, contains the number of an inode of type `IFATTRDIR`, which is a directory containing extended attribute files.

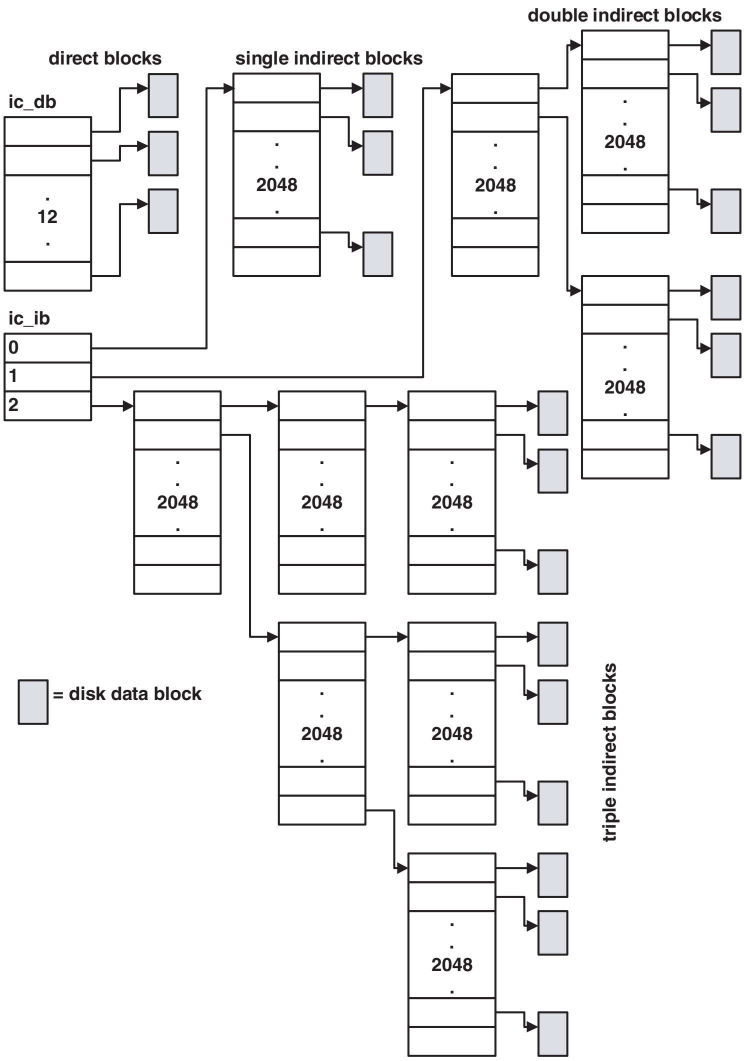

Figure 15.2 UFS Block Layout

15.2.2 UFS Directories

The file name information and hierarchy information that constitute the directory structure of UFS are stored in directories. Each directory stores a list of file names and the inode number for each file; this information (stored in `struct direct`) allows the directory structure to relate file names to real disk files.

The directory itself is stored in a file as a series of chunks, which are groups of the directory entries. Earlier file systems like the System V file system had a fixed directory record length, which meant that a lot of space would be wasted if provision was made for long file names. In the UFS, each directory entry can be of variable length, thus providing a mechanism for long file names without a lot of wasted space. UFS file names can be up to 255 characters long.

The group of directory chunks that constitute a directory is stored as a special type of file. The notion of a directory as a type of file allows UFS to implement a hierarchical directory structure: Directories can contain files that are directories. For example, the root directory has a name, "/", and an inode number, 2, which holds a chunk of directory entries holding a number of files and directories. One of these directory entries, named `etc`, is another directory containing more files and directories. For traversal up and down the file system, the `chdir` system call opens the directory file in question and then sets the current working directory to point to the new directory file. Figure 15.3 illustrates the directory hierarchy.

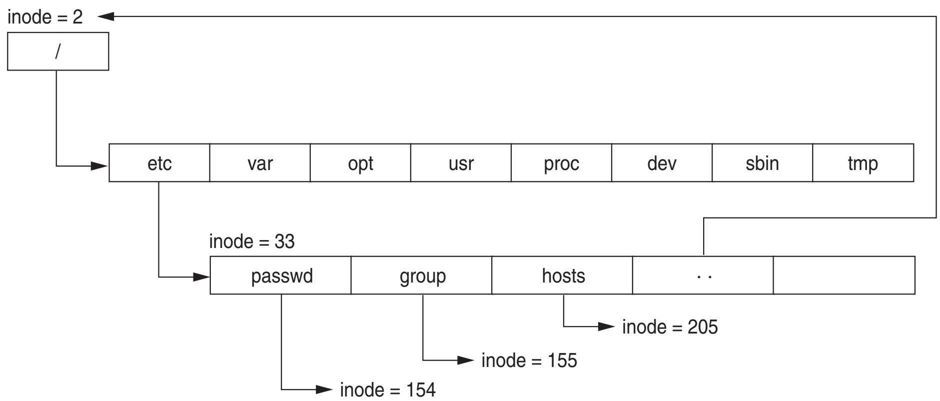

Figure 15.3 UNIX Directory Hierarchy

Each directory contains two special files. The file named "." is a link to the directory itself; the file named ". ." is a link to the parent directory. Thus, a change of directory to `..` leads to the parent directory.

Now let's switch gears and see what the on-disk structures for directories look like.

The contents of a directory are broken up into `DIRBLKSIZ` chunks, also known as dirblks. Each of these contains one or more direct structures. `DIRBLKSIZ` was chosen to be the same as the size of a disk sector so that modifications to directory entries could be done atomically on the assumption that a sector write either completes successfully or fails (which can no longer be guaranteed with the advancement of cached hard drives).

Each directory entry is stored in a structure called `direct` that contains the inode number (`d_ino`), the length of the entry (`d_reclen`), the length of the name (`d_namelen`), and a null-terminated string for the name itself (`d_name`).

```
#define DIRBLKSIZ       DEV_BSIZE
#define MAXNAMLEN       255

struct  direct {
        uint32_t        d_ino;          /* inode number of entry */
        ushort_t        d_reclen;       /* length of this record */
        ushort_t        d_namlen;       /* length of string in d_name */
        char    d_name[MAXNAMLEN + 1];  /* name must be no longer than this */
};

                                      See usr/src/uts/common/sys/fs/ufs_fsdir.h
```

`d_reclen` includes the space consumed by all the fields in a directory entry, including `d_name`'s trailing null character. This facilitates directory entry deletion because when an entry is deleted, if it is not the first entry in the current directory, the entry before it is grown to include the deleted one, that is, `d_reclen` is incremented to account for the size of the next entry. The procedure is relatively inexpensive and helps keep internal fragmentation down. Figure 15.4 illustrates the concept of directory deletion.

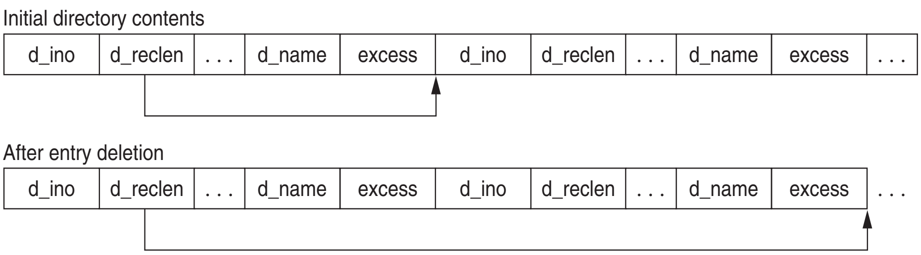

Figure 15.4 Deletion of a Directory Entry

15.2.3 UFS Hard Links

There is one inode for each file on disk; however, with hard links, each file can have multiple file names. With hard links, file names in multiple directories point to the same on-disk inode. The inode reference count field reflects the number of hard links to the inode. Figure 15.5 illustrates inode 1423 describing a file; two separate directory entries with different names both point to the same inode number. Note that the reference count, `refcnt`, has been incremented to 2.

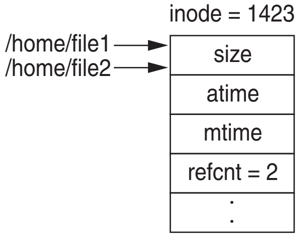

Figure 15.5 UFS Links

15.2.4 Shadow Inodes

UFS allows storage of additional per-inode data through the use of shadow inodes. The implementation of a shadow inode is generic enough to permit storage of any arbitrary data. All that is needed are a tag to identify the data and functions to convert the appropriate data structures from on-disk to in-core, and vice versa. As of this writing (2005), only two data types are defined: `FSD_ACL` for identification of ACLs and `FSD_DFACL` for default ACLs. Only one shadow inode is permitted per inode today, and as a result both ACLs and default ACLs are stored in the same shadow inode.

```
typedef struct ufs_fsd {
        int     fsd_type;               /* type of data */
        int     fsd_size;               /* size in bytes of ufs_fsd and data */
        char    fsd_data[1];            /* data */
} ufs_fsd_t;
                                            See usr/src/uts/common/sys/fs/ufs_acl.h
```

The way a shadow inode is laid out on disk is quite simple (see Figure 15.6). All the entries for the shadow inode contain one header that includes the type of data and the length of the whole record, data + header. Entries are then simply concatenated and stored to disk as a separate inode with the inode's `ic_smode` set to `ISHAD`. The parent's `ic_shadow` is then updated to point to this shadow inode.

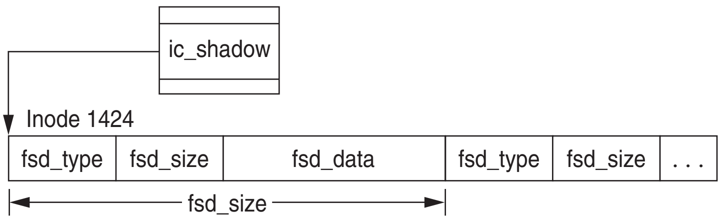

Figure 15.6 On-Disk Shadow Inode Layout

15.2.5 The Boot Block

Figure 15.7 illustrates the UFS layout discussed in this section. At the start of the file system is the boot block. This is a spare sector reserved for the boot program when UFS is used as a root file system. At boot time, the boot firmware loads the first sector from the boot device and then starts executing code residing in that block. The firmware boot is file system independent, which means that the boot firmware has no knowledge about the file system. We rely on code in the file system boot block to mount the root file system. When the system starts, the UFS boot block is loaded and executed, which, in turn, mounts the UFS root file system. The boot program then passes control to a larger kernel loader, in `/platform/sun4[mud]/ufsboot`, to load the UNIX kernel.

The boot program is loaded onto the first sector of the file system at install time with the `installboot(1M)` command. The 512-byte install boot image resides in `/usr/platform/sun4[mud]/lib/fs/ufs/bootblk` in the platform-dependent directories.

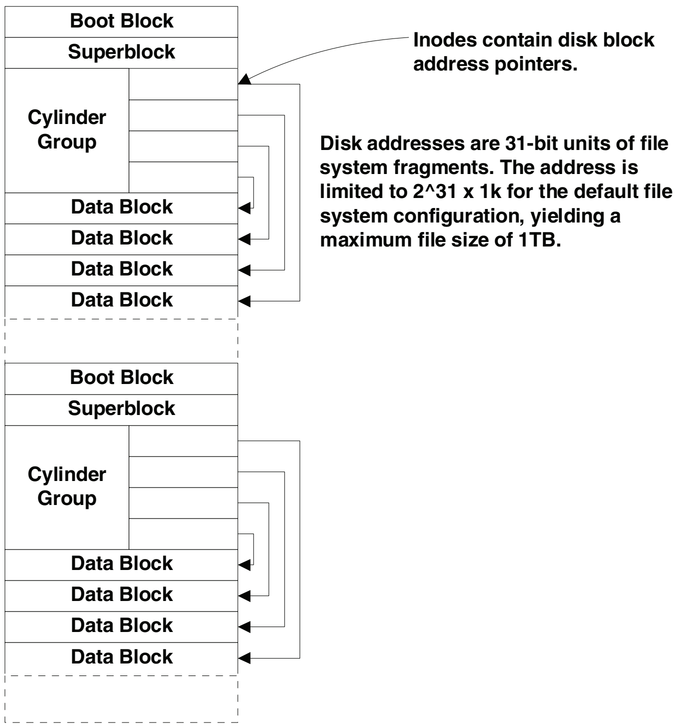

Figure 15.7 UFS Layout

15.2.6 The Superblock

The superblock contains all the information about the geometry and layout of the file system and is critical to the file system state. As a safety precaution, the superblock is replicated across the file system with each cylinder group so that the file system is not crippled if the superblock becomes corrupted. It is initially created by `mkfs` and updated by `tunefs` and `mkfs` (in case a file system is grown). The primary superblock starts at an offset of 8192 bytes into the partition slice and occupies one file system block (usually 8192 bytes, but can be 4096 bytes on x86 architectures). The superblock contains a variety of information, including the location of each cylinder group and a summary list of available free blocks. The major information in the superblock that identifies the file system geometry is listed below.

- **`fs_sblkno`.** Address of superblock in file system; defaults to block number 16.
- **`fs_cblkno`.** Offset of the first cylinder block in the file system.
- **`fs_iblkno`.** Offset of the first inode blocks in the file system.
- **`fs_dblkno`.** Offset of the first data blocks after the first cylinder group.
- **`fs_cgoffset`.** Cylinder group offset in the cylinder.
- **`fs_cgmask`.** Mask to obtain physical starting fragment number of the cylinder group.
- **`fs_time`.** Last time written.
- **`fs_size`.** Number of blocks in the file system.
- **`fs_dsize`.** Number of data blocks the in file system.
- **`fs_ncg`.** Number of cylinder groups.
- **`fs_cpg`.** Number of cylinders in a cylinder group.
- **`fs_ipg`.** Number of inodes in a cylinder group.
- **`fs_fpg`.** Number of fragments (including metadata) in a cylinder group.
- **`fs_bsize`.** Size of basic blocks in the file system.
- **`fs_fsize`.** Size of fragmented blocks in the file system.
- **`fs_frag`.** Number of fragments in a block in the file system.
- **`fs_magic`.** A magic number to validate the superblock.

The file system configuration parameters also reside in the superblock. The file system parameters include some of the following, which are configured at the time the file system is constructed. You can tune the parameters later with the `tunefs` command.

- **`fs_minfree`.** Minimum percentage of free blocks.

- **fs_rotdelay.** Number of milliseconds of rotational delay between sequential blocks. The rotational delay was used to implement block interleaving when the operating system could not keep up with reading contiguous blocks. Since this is no longer an issue, fs_rotdelay defaults to zero.
- **fs_rps.** Disk revolutions per second.
- **fs_maxcontig.** Maximum number of contiguous blocks, controls the number of read-ahead blocks.
- **fs_maxbpg.** Maximum number of data blocks per cylinder group.
- **fs_optim.** Optimization preference, space, or time.

And here are the significant logging related fields in the superblock:

- **fs_rolled.** Determines whether any data in the log still needs to be rolled back to the file system.
- **fs_si.** Indicates whether logging summary information is up to date or whether it needs to be recalculated from cylinder groups.
- **fs_clean.** Is set to FS_LOG for logging file system.
- **fs_logbno.** Is the disk block number of logging metadata.
- **fs_reclaim:** Is set to indicate if the reclaim thread is running or needs to be run.

See struct fs in usr/src/uts/common/sys/fs/ufs_fs.h for the complete superblock structure definition

15.2.7 The Cylinder Group

The cylinder group is made up of several logically distinct parts. At logical offset zero into the cylinder group is a backup copy of the file system's superblock. Following that, we have the cylinder group structure, the blktot array (indicating how many full blocks are available), the blks array (representing the full-sized blocks that are free in each rotational position), inode bitmap (marking which inodes are in use), and finally, the bitmap of which fragments are free. Next in the layout is the array of inodes whose size varies according to the number of inodes in a cylinder group (on-disk inode size is restricted to 128 bytes). And finally, the rest of the cylinder group is filled by the data blocks.

Figure 15.8 illustrates the layout.

fs superblock	struct cg + bitmaps	inodes	data blocks ...

Figure 15.8 Logical Layout of a Cylinder Group

The last cylinder group in a file system may be incomplete because the number of cylinders in a disk drive is usually not exactly rounded up to the cylinder groups. In this case, we simply reduce the number of data blocks available in the last cylinder group; however, the metadata portion of the cylinder group stays the same throughout the file system. The `cg_ncyl` and `cg_nblk` fields of the cylinder group structure guide us to the size so that we don't accidentally go out of bounds.

```
/*
 * Cylinder group block for a file system.
 *
 * Writable fields in the cylinder group are protected by the associated
 * super block lock fs->fs_lock.
 */
#define CG_MAGIC        0x090255
struct  cg {
        uint32_t cg_link;               /* NOT USED linked list of cyl groups */
        int32_t cg_magic;               /* magic number */
        time32_t cg_time;               /* time last written */
        int32_t cg_cgx;                 /* we are the cgx'th cylinder group */
        short   cg_ncyl;                /* number of cyl's this cg */
        short   cg_niblk;               /* number of inode blocks this cg */
        int32_t cg_ndblk;               /* number of data blocks this cg */
        struct  csum cg_cs;             /* cylinder summary information */
        int32_t cg_rotor;               /* position of last used block */
        int32_t cg_frotor;              /* position of last used frag */
        int32_t cg_irotor;              /* position of last used inode */
        int32_t cg_frsum[MAXFRAG];      /* counts of available frags */
        int32_t cg_btotoff;             /* (int32_t)block totals per cylinder */
        int32_t cg_boff;                /* (short) free block positions */
        int32_t cg_iusedoff;            /* (char) used inode map */
        int32_t cg_freeoff;             /* (uchar_t) free block map */
        int32_t cg_nextfreeoff;         /* (uchar_t) next available space */
        int32_t cg_sparecon[16];        /* reserved for future use */
        uchar_t cg_space[1];            /* space for cylinder group maps */
/* actually longer */
};

                                      See usr/src/uts/common/sys/fs/ufs_fs.h
```

15.2.8 Summary of UFS Architecture

Figure 15.9 puts it all together.

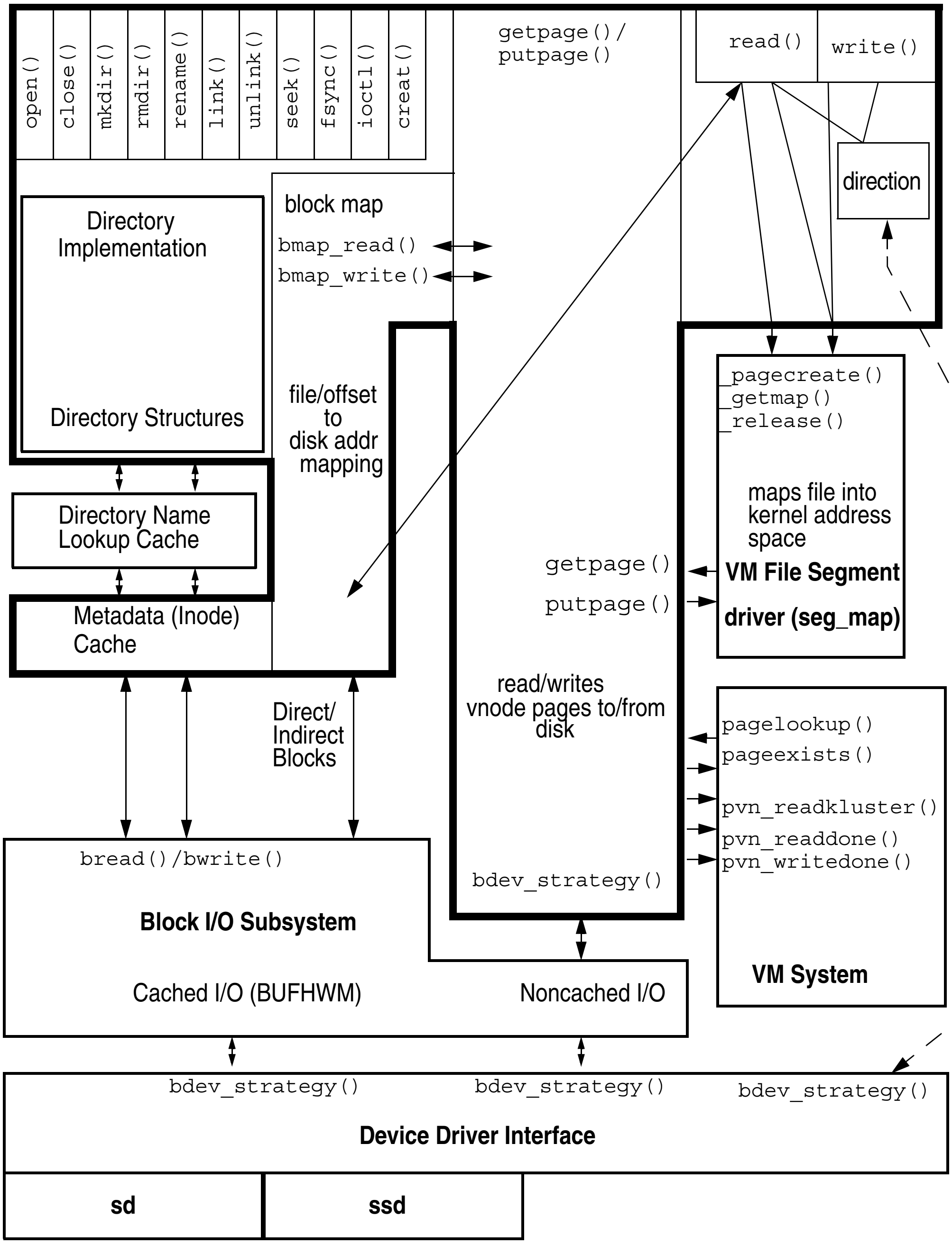

Figure 15.9 The UFS File System

15.3 The UFS Inode

The *inode* (Index Node) is UFS's internal descriptor for a file. Each file system has two forms of an inode: the on-disk inode and the in-core (in-memory) inode. The on-disk inode resides on the physical medium and represents the on-disk format and layout of the file.

15.3.1 In-Core UFS Inodes

The in-core inode, as you may have guessed, resides in memory and contains the file-system-dependent information, free-list pointers, hash anchors, kernel locks (covered in UFS locking below), and inode state.

```
typedef struct inode {
        struct  inode *i_chain[2];      /* must be first */
        struct inode *i_freef;  /* free list forward - must be before i_ic */
        struct inode *i_freeb;  /* free list back - must be before i_ic */
        struct  icommon i_ic;   /* Must be here */
        struct  vnode *i_vnode; /* vnode associated with this inode */
        struct  vnode *i_devvp; /* vnode for block I/O */
        dev_t   i_dev;          /* device where inode resides */
        ino_t   i_number;       /* i number, 1-to-1 with device address */
        off_t   i_diroff;       /* offset in dir, where we found last entry */
                                /* just a hint - no locking needed */
        struct ufsvfs *i_ufsvfs; /* incore fs associated with inode */
        struct  dquot *i_dquot; /* quota structure controlling this file */
        krwlock_t i_rwlock;     /* serializes write/setattr requests */
        krwlock_t i_contents;   /* protects (most of) inode contents */
        kmutex_t i_tlock;       /* protects time fields, i_flag */
        offset_t i_nextr;       /*                                      */
                                /* next byte read offset (read-ahead)   */
                                /*   No lock required                   */
                                /*                                      */
        uint_t  i_flag;         /* inode flags */
        uint_t  i_seq;          /* modification sequence number */
        boolean_t i_cachedir;   /* Cache this directory on next lookup */
                                /* - no locking needed  */
        long    i_mapcnt;       /* mappings to file pages */
        int     *i_map;         /* block list for the corresponding file */
        dev_t   i_rdev;         /* INCORE rdev from i_oldrdev by ufs_iget */
        size_t  i_delaylen;     /* delayed writes, units=bytes */
        offset_t i_delayoff;    /* where we started delaying */
        offset_t i_nextrio;     /* where to start the next clust */
        long    i_writes;       /* number of outstanding bytes in write q */
        kcondvar_t i_wrcv;      /* sleep/wakeup for write throttle */
        offset_t i_doff;        /* dinode byte offset in file system */
        si_t *i_ufs_acl;        /* pointer to acl entry */
        dcanchor_t i_danchor;   /* directory cache anchor */
        kthread_t *i_writer;    /* thread which is in window in wrip() */
} inode_t;
```

See usr/src/uts/common/sys/fs/ufs_inode.h

New with Solaris 10, an inode sequence number was added to the in-core `inode` structure to support NFSv3 and NFSv4 detection of atomic changes to the inode. Two caveats with this new value: `i_seq` must be updated if `i_ctime` and `i_mtime` are changed; the value of `i_seq` is only guaranteed to be persistent while the inode is active.

15.3.2 Inode Cache

When the last reference to a `vnode` is released, the `vop_inactive()` routine for the file system is called. (See `vnode` reference counts in Section 14.6.8.) UFS uses `vop_inactive()` to free the inode when it is no longer required. If we were to destroy each `vnode` when the last reference to a `vnode` is relinquished, we would throw away all the data relating to that `vnode`, including all the file pages cached in the page cache. This practice could mean that if a file is closed and then reopened, none of the file data that was cached would be available after the second open and would need to be reread from disk. To remedy the situation, UFS caches all unused inodes in its global cache.

The UFS inode cache contains an entry for every open inode in the system. It also attempts to keep as many closed inodes as possible so that inactive inodes/vnodes and associated pages are around in memory for possible reuse. This is a global cache and not a per-file system cache, and that unfortunately leads to several performance issues.

The inode cache consists of several disconnected queues or chains, and each queue is linked with the inode's `i_forw` and `i_backw` pointers (see Figure 15.10). Starting with Solaris 10, hashing of inode entries is done with the inode number (because of recent `devfs` changes) rather than with the inode number and the device number (Solaris 9 and earlier). These queues are managed according to least recently used (LRU) scheme.

An inode free list is also maintained within the cache which is built upon the `i_freef` and `i_freeb` pointers. These enable the free list to span several hash chains. If an inode is not on the free list, then the `i_freef` and `i_freeb` values point back to the inode itself.

Inodes on the free list can be part of two separate queues:

- **Idle queue.** Holds the idle or unreferenced inodes (where the `v_count` equals 1, t and the `i_nlink` is greater than 0). This queue is managed by the global file system idle thread, which frees entries, starting at the head. When new entries are added, `ufs_inactive()` adds an inode to the head if the inode has no associated pages; otherwise, the inode is added to the tail. This ensures that pages are retained longer in memory for possible reuse—the frees are done starting at the head.

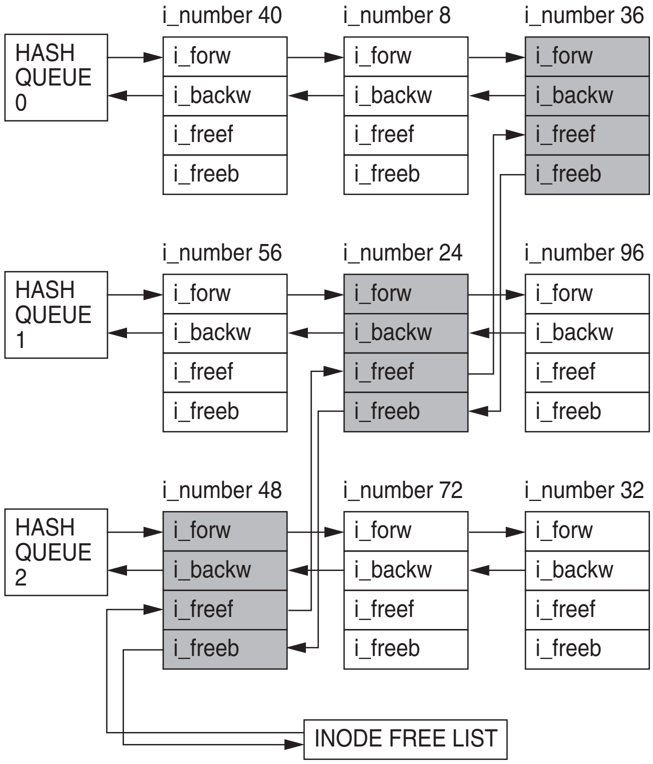

Figure 15.10 UFS Inode Hash Queues

Starting with Solaris 10, the idle queue architecture was reorganized into two separate hash queues: `ufs_useful_iq` and `ufs_junk_iq`. If an inode has pages associated with it (`vn_has_cached_data(vnode)`) or is a fast symbolic link (`i_flag` and `IFASTSYMLNK`), then it is attached to the useful idle queue. All other inodes are attached to the junk idle queue instead. These queues are not used for searching but only for grouping geographically local inodes for faster updates and fewer disk seeks upon reuse. Entries from the junk idle queue are destroyed first when `ufs_idle_free()` is invoked by the UFS idle thread so that cached pages pertaining to entries in the `ufs_useful_iq` idle queue stay in memory longer.

The idle thread is adjusted to run when there are 25% of `ufs_ninode` entries on the idle queue. When it runs, it gives back half of the idle queue until the queue falls below the low water mark of `ufs_q->uq_lowat`. Inodes on the junk queue get destroyed first. Figure 15.11 illustrates the process.

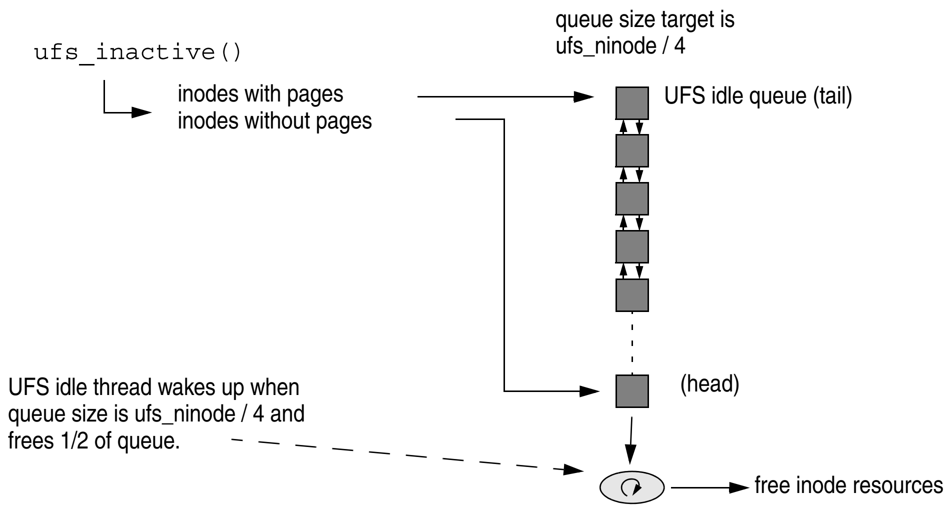

Figure 15.11 UFS Idle Queue

- **Delete queue.** Is active if UFS logging is enabled and consists of inodes that are unlinked or deleted (`v_count` equals 1 and `i_nlink` is less than or equal to 0). This queue is a performance enhancer for file systems with logging turned on and observing heavy deletion activity. The delete queue is handled by the per-file system delete thread, which queues the inodes to be deleted by `the ufs_delete()` thread. This significantly boosts response times for removal of large amounts of data. If logging is not enabled, `ufs_delete()` is called immediately. `ufs_delete()` calls `VN_RELE()` after it has finished processing, which causes the inode to once again be processed by `ufs_inactive`, which this time puts it on the idle queue. While on the delete queue, the `inode`'s `i_freef` and `i_freeb` point to the inode itself since the inodes are not free yet.

15.3.3 Block Allocation

The UFS file system is block based where each file is represented by a set of fixed sized units of disk space, indexed by a tree of *physical-meta-data* blocks.

15.3.3.1 Layout Policy

UFS uses block sizes of 4 and 8 Kbytes, which provides significantly higher performance than the 512-byte blocks used in the System V file system. The downside of larger blocks was that when partially allocated blocks were created, several kilobytes of disk space for each partly filled file system block was wasted. To overcome this disadvantage, UFS uses the notion of file system fragments. Fragments allow a single block to be broken up into 2, 4, or 8 fragments when necessary (4 Kbytes, 2 Kbytes or 1 Kbyte, respectively).

UFS block allocation tries to prevent excessive disk seeking by attempting to co-locate inodes within a directory and by attempting to co-locate a file's inode and its data blocks. When possible, all the inodes in a directory are allocated in the same cylinder group. This scheme helps reduce disk seeking when directories are traversed; for example, executing a simple `ls -l` of a directory will access all the inodes in that directory. If all the inodes reside in the same cylinder group, most of the data are cached after the first few files are accessed. A directory is placed in a cylinder group different from that of its parent.

Blocks are allocated to a file sequentially, starting with the first 96 Kbytes (the first 12 direct blocks), skipping to the next cylinder group and allocating blocks up to the limit set by the file system parameter `maxbpg` (maximum-blocks-per-cylinder-group). After that, blocks are allocated from the next available cylinder group.

By default, on a file system greater than 1 Gbyte, the algorithm allocates 96 Kbytes in the first cylinder group, 16 Mbytes in the next available cylinder group, 16 Mbytes from the next, and so on. The maximum cylinder group size is 54 Mbytes, and the allocation algorithm allows only one-third of that space to be allocated to each section of a single file when it is extended. The `maxbpg` parameter is set to 2,048 8-Kbyte blocks by default at the time the file system is created. It is also tunable but can only be tuned downward since the maximum cylinder group size is 16-Mybte allocation per cylinder group.

Selection of a new cylinder group for the next segment of a file is governed by a rotor and free-space algorithm. A per-file-system allocation rotor points to one of the cylinder groups; each time new disk space is allocated, it starts with the cylinder group pointed to by the rotor. If the cylinder group has less than average free space, then it is skipped and the next cylinder group is tried. This algorithm makes the file system attempt to balance the allocation across the cylinder groups.

Figure 15.12 shows the default allocation that is used if a file is created on a large UFS. The first 96 Kbytes of file 1 are allocated from the first cylinder group. Then, allocation skips to the second cylinder group and another 16 Mbytes of file 1 are allocated, and so on. When another file is created, we can see that it consumes the holes in the allocated blocks alongside file 1. There is room for a third file to do the same.

The actual on-disk layout will not be quite as simple as the example shown but does reflect the allocation policies discussed. We can use an add-on tool, `filestat`, to view the on-disk layout of a file, as shown below.

```
sol8# /usr/local/bin/filestat testfile
Inodes per cyl group:   128
Inodes per block:       64
Cylinder Group no:      0
Cylinder Group blk:     64
File System Block Size: 8192
Block Size:             512
Number of 512b Blocks:  262288

Start Block    End Block    Length (512 byte Blocks)
-----------    -----------  -----------------------
        144 -> 335          192
        400 -> 33167        32768
     110800 -> 143567       32768
     221264 -> 221343       80
     221216 -> 221263       48
     221456 -> 254095       32640
     331856 -> 331999       144
     331808 -> 331855       48
     332112 -> 364687       32576
     442448 -> 442655       208
     442400 -> 442447       48
     442768 -> 475279       32512
```

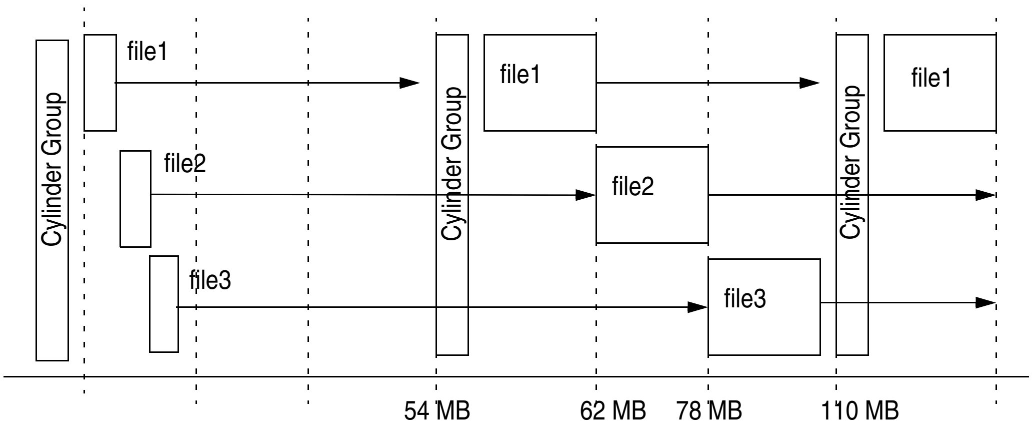

Figure 15.12 Default File Allocation in 16-Mbyte Groups

The `filestat` output shows that the first segment of the file occupies 192 (512-byte) blocks, followed by the next 16 Mbytes, which start in a different cylinder group. This particular file system was not empty when the file was created, which is why the next cylinder group chosen is a long way from the first.

We can observe the file system parameters of an existing file system with the `fstyp` command. The `fstyp` command simply dumps the superblock information for the file, revealing all the cylinder group and allocation information. The following example shows the output for a 4-Gbyte file system with default parameters.

We can see that the file system has 8,247,421 blocks and has 167 cylinder groups spaced evenly at 6,272 (51-Mbyte) intervals. The maximum blocks to allocate for each group is set to the default of 2,048 8-Kbyte, 16 Mbytes.

```
sol8# fstyp -v /dev/vx/dsk/homevol |more
ufs
magic   11954   format  dynamic time    Sat Mar  6 18:19:59 1999
sblkno  16      cblkno  24      iblkno  32      dblkno  800
sbsize  2048    cgsize  8192    cgoffset 32     cgmask  0xffffffe0
ncg     167     size    8378368 blocks  8247421
bsize   8192    shift   13      mask    0xffffe000
fsize   1024    shift   10      mask    0xfffffc00
frag    8       shift   3       fsbtodb 1
minfree 1%      maxbpg  2048    optim   time
maxcontig 32    rotdelay 0ms    rps     120
csaddr  800     cssize  3072    shift   9       mask    0xfffffe00
ntrak   32      nsect   64      spc     2048    ncyl    8182
cpg     49      bpg     6272    fpg     50176   ipg     6144
nindir  2048    inopb   64      nspf    2
nbfree  176719  ndir    10241   nifree  956753  nffree  21495
cgrotor 152     fmod    0       ronly   0       logbno  0
```

The UFS-specific version of the `fstyp` command dumps the superblock of a UFS file system, as shown below.

```
sol8# fstyp -v /dev/vx/dsk/homevol |more
ufs
magic   11954   format  dynamic time    Sat Mar  6 18:19:59 1999
sblkno  16      cblkno  24      iblkno  32      dblkno  800
sbsize  2048    cgsize  8192    cgoffset 32     cgmask  0xffffffe0
ncg     167     size    8378368 blocks  8247421
bsize   8192    shift   13      mask    0xffffe000
fsize   1024    shift   10      mask    0xfffffc00
frag    8       shift   3       fsbtodb 1
minfree 1%      maxbpg  2048    optim   time
maxcontig 32    rotdelay 0ms    rps     120
csaddr  800     cssize  3072    shift   9       mask    0xfffffe00
ntrak   32      nsect   64      spc     2048    ncyl    8182
cpg     49      bpg     6272    fpg     50176   ipg     6144
nindir  2048    inopb   64      nspf    2
nbfree  176719  ndir    10241   nifree  956753  nffree  21495
cgrotor 152     fmod    0       ronly   0       logbno  0
fs_reclaim is not set
file system state is valid, fsclean is 0
blocks available in each rotational position
cylinder number 0:
   position 0:      0    4    8   12   16   20   24   28   32   36   40   44
                   48   52   56   60   64   68   72   76   80   84   88   92
                   96  100  104  108  112  116  120  124
   position 2:      1    5    9   13   17   21   25   29   33   37   41   45
                   49   53   57   61   65   69   73   77   81   85   89   93
                   97  101  105  109  113  117  121  125
   position 4:      2    6   10   14   18   22   26   30   34   38   42   46
                   50   54   58   62   66   70   74   78   82   86   90   94
                   98  102  106  110  114  118  122  126
   position 6:      3    7   11   15   19   23   27   31   35   39   43   47
                   51   55   59   63   67   71   75   79   83   87   91   95
                   99  103  107  111  115  119  123  127
```

continues

```
cs[].cs_(nbfree,ndir,nifree,nffree):
        (23,26,5708,102) (142,26,5724,244) (87,20,5725,132) (390,69,5737,80)
        (72,87,5815,148) (3,87,5761,110) (267,87,5784,4) (0,66,5434,4)
        (217,46,5606,94) (537,87,5789,70) (0,87,5901,68) (0,87,5752,20)
.
.
cylinders in last group 48
blocks in last group 6144


cg 0:
magic   90255   tell    6000    time    Sat Feb 27 22:53:11 1999
cgx     0       ncyl    49      niblk   6144    ndblk   50176
nbfree  23      ndir    26      nifree  5708    nffree  102
rotor   1224    irotor  144     frotor  1224
frsum   7       7       3       1       1       0       9
sum of frsum: 102
iused:  0-143, 145-436
free:   1224-1295, 1304-1311, 1328-1343, 4054-4055, 4126-4127, 4446-4447, 4455, 4637-
4638,
```

15.3.3.2 Mapping Files to Disk Blocks

At the heart of a disk-based file system are the block map algorithms, which implement the on-disk file system format. These algorithms map UFS file and offsets pairs into disk addresses on the underlying storage. For UFS, two main functions—`bmap_read()` and `bmap_write()`—implement the on-disk format. Calling these functions has the following results:

- `bmap_read()` *queries* the file system as to which physical disk sector a file block resides on; that is, requests a lookup of the direct/indirect blocks that contain the disk address(es) of the required blocks.
- `bmap_write()` *allocates*, with the aid of helper functions, new disk blocks when extending or allocating blocks for a file.

The `bmap_read()` function reads file system block addresses. It accepts an inode and offset as input arguments, and a pointer to a disk address and contiguity length as output arguments.

```
int
bmap_read(struct inode *ip, u_offset_t off, daddr_t *dap, int *lenp)
```

See usr/src/uts/common/fs/ufs/ufs_bmap.c

The file system uses the `bmap_read()` algorithm to locate the physical blocks for the file being read. The `bmap_read()` function searches through the direct, indirect, and double-indirect blocks of the inode to locate the disk address of the disk blocks that map to the supplied offset. The function also searches forward from the offset, looking for disk blocks that continue to map contiguous portions of the inode, and returns the length of the contiguous segment (in blocks) in the length pointer argument. The length and the file system block clustering parameters are used within the file system as bounds for clustering contiguous blocks to provide better performance by reading larger parts of a file from disk at a time. See `ufs_getpage_ra()`, defined in `usr/src/uts/common/fs/ufs_vnops.c`, for more information on read-aheads.

```
int
bmap_write(struct inode *ip, u_offset_t off, int size,
           int alloc_only, struct cred *cr);
```

See usr/src/uts/common/fs/ufs/ufs_bmap.c

The `bmap_write()` function allocates file space in the file system when a file is extended or a file with holes has blocks written for the first time and is responsible for storing the allocated block information in the inode. `bmap_write()` traverses the block free lists, using the rotor algorithm (discussed in Section 15.3.3), and updates the local, direct, and indirect blocks in the inode for the file being extended. `bmap_write` calls several helper functions to facilitate the allocation of blocks.

daddr_t blkpref(struct inode *ip, daddr_t lbn, int indx, daddr32_t *bap)

Guides bmap_write in selecting the next desired block in the file. Sets the policy as described in Section 15.3.3.1.

int realloccg(struct inode *ip, daddr_t bprev, daddr_t bpref, int osize, int nsize, daddr_t *bnp, cred_t *cr)

Re-allocates a fragment to a bigger size. The number and size of the old block size is specified and the allocator attempts to extend the original block. Failing that, the regular block allocator is called to obtain an appropriate block.

int alloc(struct inode *ip, daddr_t bpref, int size, daddr_t *bnp, cred_t *cr)

Allocates a block in the file system. The size of the block is specified which is a multiple of (fs_fsize <= fs_bsize). If a preference (usually obtained from blkpref()) is specified, the allocator will try to allocate the requested block. If that fails, a rotationally optimal block in the same cylinder is found. Failing that a block in the same cylinder group is searched for. And in case that fails, the allocator quadratically rehashes into other cylinder groups (see hashalloc() in uts/common/fs/ufs/ufs_alloc.c) to locate an available block. If no preference is given, a block in the same cylinder is found, and failing that the allocator quadratically searches other cylinder groups for one.

See uts/common/fs/ufs/ufs_alloc.c

```
static void ufs_undo_allocation(inode_t *ip,
                                int block_count,
                                struct ufs_allocated_block table[],
                                int inode_sector_adjust)
```

In the case of an error, bmap_write() will call ufs_undo_allocation to free any blocks which were used during the allocation process.

See uts/common/fs/ufs/ufs_bmap.c

15.3.3.3 Reading and Writing UFS Blocks

A file system read calls `bmap_read()` to find the location of the underlying physical blocks for the file being read. UFS then calls the device driver's strategy routine for the device containing the file system to initiate the read operation by calling `bdev_strategy()`.

A file system write operation that extends a file first calls `bmap_write()` to allocate the new blocks and then calls `bmap_read()` to obtain the block location for the write. UFS then calls the device driver's strategy routine, by means of `bdev_strategy()`, to initiate the file write.

15.3.3.4 Buffering Block Metadata

The block map functions access metadata (single, double and triple indirect blocks) on the device media through the buffer cache, using the `bread_common()` and `bwrite_common()` buffered block I/O kernel functions. The block I/O functions read and write device blocks in 512-byte chunks, and they cache physical disk blocks in the block buffer cache (note: this cache is different from the page cache, used for file data). The UFS file system requires 1 Mbyte of metadata for every 2 Gbytes of file space. This relationship can be used as a rule to calculate the size of the block buffer cache, set by the `bufhwm` kernel parameter.

15.3.4 Methods to Read and Write UFS Files

Files can be read or written in two ways: by the `read()` or `write()` system calls or by mapped file I/O. The `read()` and `write()` system calls call the file system's `ufs_read()` and `ufs_write()` method. These methods map files into the kernel's address space and then use the file system's `ufs_getpage()` and `ufs_putpage()` methods to transfer data to and from the physical media.

15.3.4.1 `ufs_read()`

An example of the steps taken by a UFS read system call is shown in Figure 15.13. A read system call invokes the file-system-dependent read function, which turns the read request into a series of `vop_getpage()` calls by mapping the file into the kernel's address space with the `seg_kpm` driver (through the `seg_map` driver), as described in Section 14.7.

The `ufs_read` method calls into the `seg_map` driver to locate a virtual address in the kernel address space for the file and offset requested with the `segmap_getmapflt()` function. The `seg_map` driver determines whether it already has a mapping for the requested offset by looking into its hashed list of mapping slots. Once a slot is located or created, an address for the page is located. `segmap` then calls back into the file system with `ufs_getpage()` to soft-initiate a page fault to

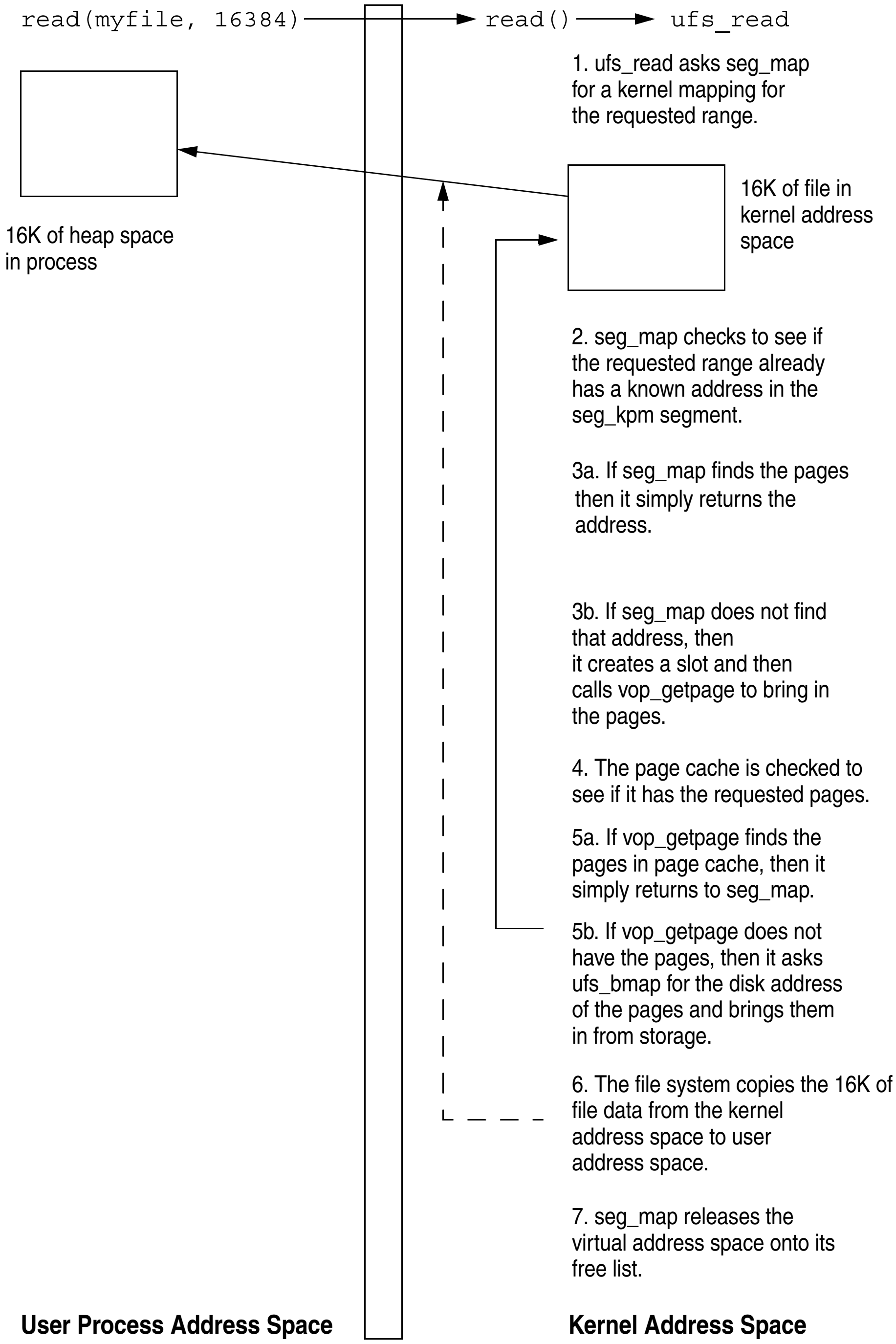

Figure 15.13 `ufs_read()`

read in the page at the virtual address of the `seg_map` slot. The page fault is initiated while we are still in the `segmap_getmap()` routine, by a call to `segmap_fault()`. That function in turn calls back into the file system with `ufs_getpage()`, which calls out file system's `_getpage()`. If not, then a slot is created and `ufs_getpage()` is called to read in the pages.

The `ufs_getpage()` routine brings the requested range of the file (`vnode`, offset, and length) from disk into the virtual address, and the length is passed into the `ufs_getpage()` function. The `ufs_getpage()` function locates the file's blocks (through the block map functions discussed in Section 15.3.3.2) and reads them by calling the underlying device's strategy routine.

Once the page is read by the file system, the requested range is copied back to the user by the `uiomove()` function. The file system then releases the slot associated with that block of the file by using the `segmap_release()` function. At this point, the slot is not removed from the segment, because we may need the same file and offset later (effectively caching the virtual address location); instead, it is added to a `seg_map` free list so that it can be reclaimed or reused later.

15.3.4.2 `ufs_write()`

Writing to the file system is performed similarly, although it is more complex because of some of the file system write performance enhancements, such as delayed writes and write clustering. Writing to the file system follows the steps shown in Figure 15.14.

The write system call calls the file-system-independent write, which in our example calls `ufs_write()`. UFS breaks the write into 8-Kbyte chunks and then processes each chunk. For each 8-Kbyte chunk, the following steps are performed.

1. UFS asks the `segmap` driver for an 8-Kbyte mapping of the file in the kernel's virtual address space. The page for the file and offset is mapped here so that the data can be copied in and then written out with paged I/O.
2. If the file is being extended or a new page is being created within a hole of a file, then a call is made to the `segmap_pagecreate` function to create and lock the new pages. Next, a call is made `segmap_pageunlock()` to unlock the pages that were locked during the `page_create`.
3. If the write is to a whole file system block, then a new zeroed page is created with `segmap_pagecreate()`. In the case of a partial block write, the block must first be read in so that the partial block contents can be replaced.
4. The new page is returned, locked, to UFS. The buffer that is passed into the write system call is copied from user address space into kernel address space.
5. The `ufs_write` throttle first checks to see if too many bytes are outstanding for this file as a result of previous delayed writes. If more than the kernel

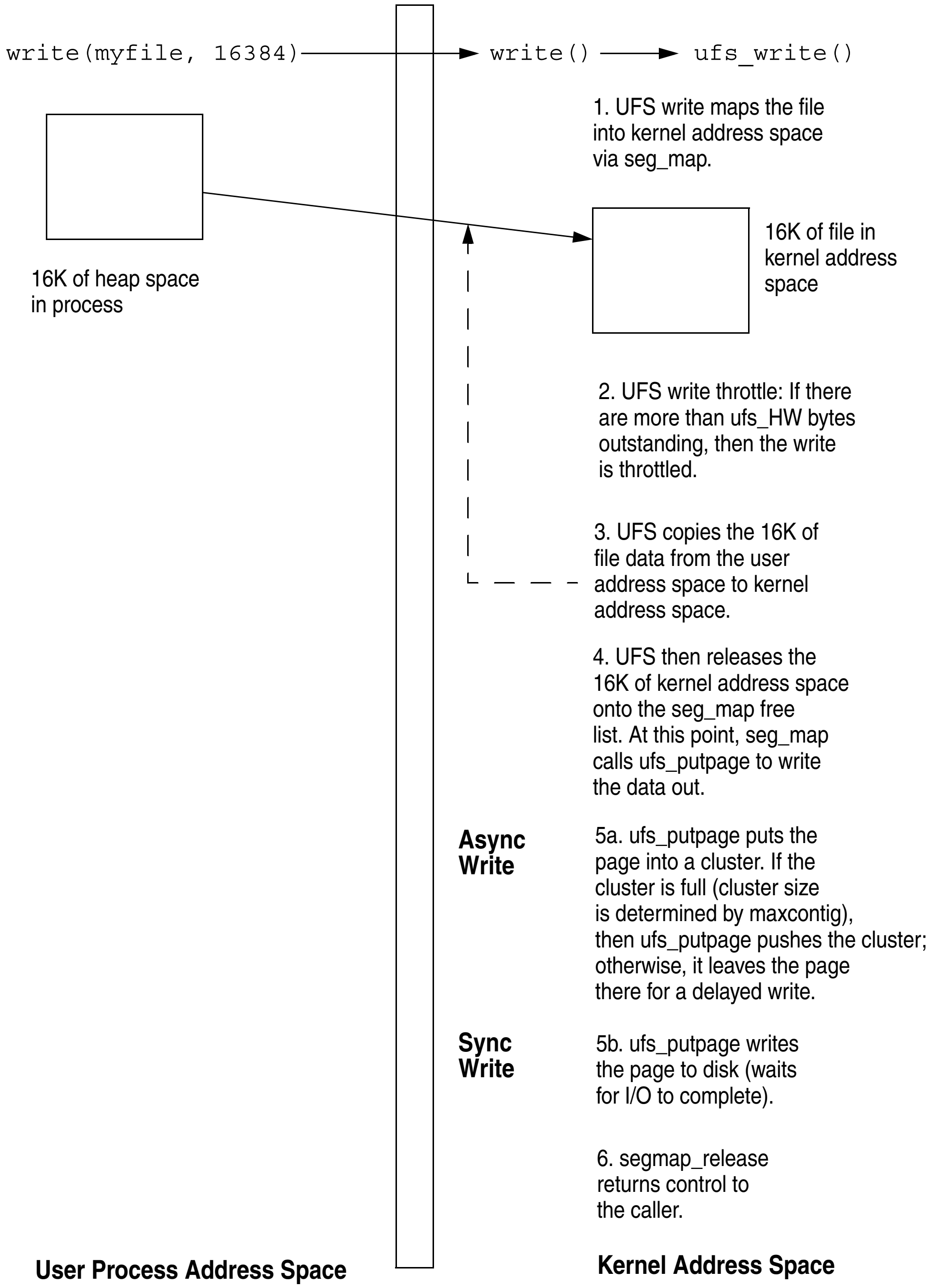

Figure 15.14 `ufs_write()`

parameter ufs_HW bytes are outstanding, the write is put to sleep until the amount of outstanding bytes drops below the kernel parameter ufs_LW.

The file system calls the seg_map driver to map in the portion of the file we are going to write. The data is copied from the process's user address space into the kernel address space allocated by seg_map, and seg_map is then called to release the address space containing the dirty pages to be written. This is when the real work of write starts, because seg_map calls ufs_putpage() when it realizes there are dirty pages in the address space it is releasing.

15.4 Access Control in UFS

The traditional UNIX File System provides a simple file access scheme based on users, groups, and world, whereby each file is assigned an owner and a UNIX group, and then is assigned a bitmap of permissions for user, group, and world, as illustrated in Figure 15.15.

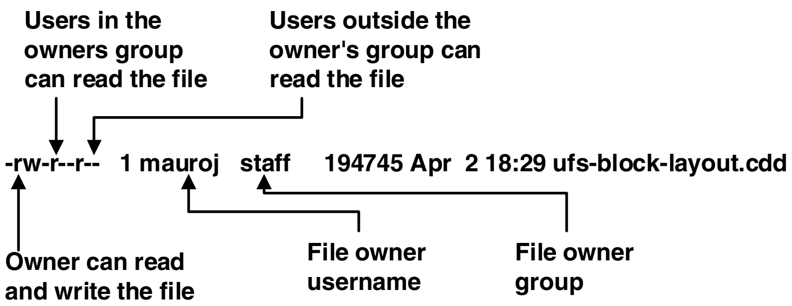

Figure 15.15 Traditional File Access Scheme

This scheme is flexible when file access permissions align with users and groups of users, but it does not provide a mechanism to assign access to lists of users that do not coincide with a UNIX group. For example, if we want to give read access to file 1 to Mark and Chuck, and then read access to file 2 to Chuck and Barb, then we would need to create two UNIX groups, and Chuck would need to switch groups with the chgrp command to gain access to either file.

To overcome this drawback, some operating systems use an access control list (ACL), whereby lists of users with different permissions can be assigned to a file. Solaris introduced the notion of access control lists in the B1 secure version, known as Trusted Solaris, in 1993. Trusted Solaris ACLs were later integrated with the commercial Solaris version in 1995 with Solaris 2.5.

With Solaris ACLs, administrators can assign a list of UNIX user IDs and groups to a file by using the `setfacl` command and can review the ACLs by using the `getfacl` command, as shown below.

```
# setfacl -m user:jon:rw- memtool.c
# getfacl memtool.c

# file: memtool.c
# owner: rmc
# group: staff
user::r--
user:jon:rw-            #effective:r--
group::r--              #effective:r--
mask:r--
other:r--

# ls -l memtool.c
-r--r--r--+  1 rmc      staff        638 Mar 30 11:32 memtool.c
```

For example, we can assign access to a file for a specific user by using the `setfacl` command. Note that the UNIX permissions on the file now contain a +, signifying that an access control list is assigned to this file.

Multiple users and groups can be assigned to a file, offering a flexible mechanism for assigning access rights. ACLs can be assigned to directories as well. Note that unlike the case with some other operating systems, access control lists are not inherited from a parent, so a new directory created under a directory with an ACL will not have an ACL assigned by default.

ACLs are divided into three parts: on-disk, in-core, and user level. On-disk format is used to represent the ACL data that is stored in the file's shadow inode, in-core structure is used by UFS internally, and the user-level format is used by the system to present data to the requester.

The `ufs_acl` structure defines an ACL entry that is encapsulated in the `ufs_fsd` structure and then stored on disk in a shadow inode. Refer to Section 15.2.4 for more information on shadow inode storage.

```
/*
 * On-disk UFS ACL structure
 */
typedef struct ufs_acl {
        union {
                uint32_t        acl_next;       /* Pad for old structure */
                ushort_t        acl_tag;        /* Entry type */
        } acl_un;
        o_mode_t        acl_perm;               /* Permission bits */
        uid_t           acl_who;                /* User or group ID */
} ufs_acl_t;
                                               See usr/src/uts/common/sys/fs/ufs_acl.h
```

The in-core format consists of the `ufs_ic_acl` structure and the in-core ACL mask (`ufs_aclmask`) structure.

```
/*
 * In-core UFS ACL structure
 */
typedef struct ufs_ic_acl {
        struct ufs_ic_acl       *acl_ic_next;   /* Next ACL for this inode */
        o_mode_t                acl_ic_perm;    /* Permission bits */
        uid_t                   acl_ic_who;     /* User or group ID */
} ufs_ic_acl_t;

/*
 * In-core ACL mask
 */
typedef struct ufs_aclmask {
        short           acl_ismask;     /* Is mask defined? */
        o_mode_t        acl_maskbits;   /* Permission mask */
} ufs_aclmask_t;
                                        See usr/src/uts/common/sys/fs/ufs_acl.h
```

When ACL data is exchanged to and from the application, a `struct acl` relays the permission bits, user or group ID, and the type of ACL.

```
typedef struct acl {
        int             a_type;         /* the type of ACL entry */
        uid_t           a_id;           /* the entry in -uid or gid */
        o_mode_t        a_perm;         /* the permission field */
} aclent_t;
                                        See usr/src/uuts/common/sys/acl.h
```

The following routines are available in UFS to manipulate ACLs.

```
static int
ufs_setsecattr(struct vnode *vp, vsecattr_t *vsap, int flag, struct cred *cr)

Used primarily for updates to ACLs. The structure vsecattr is converted to ufs_acl for
in-core storage of ACLs. All file mode changes are updated via this routine.

static int
ufs_getsecattr(struct vnode *vp, vsecattr_t *vsap, int flag,struct cred *cr)

If ACL data is present, it is converted to vsecattr. Otherwise a new entry is created
from the mode bits and returned.

int
ufs_acl_access(struct inode *ip, int mode, cred_t *cr)

Checks the inode's ACLs to see if access of type mode is allowed.

int
ufs_acl_get(struct inode *ip, vsecattr_t *vsap, int flag, cred_t *cr)

Called by ufs_getsecattr() to obtain ACL information.
```

continues

```
int
ufs_acl_set(struct inode *ip, vsecattr_t *vsap, int flag, cred_t *cr)

Called by ufs_setsecattr() to set the inode's ACL information.

si_t *
ufs_acl_cp(si_t *sp)

Copies ACL information from one shadow inode into a new created shadow inode.

int
ufs_acl_setattr(struct inode *ip, struct vattr *vap, cred_t *cr)

Sets the inode's ACL attributes.

usr/src/uuts/common/fs/ufs/ufs_acl.c
```

15.5 Extended Attributes in UFS

In Solaris 9, a new interface was added to UFS for the storage of attributes. Rather than ACLs, which added a shadow inode to each file for permission storage; extended attributes adds a directory inode to each file (see `struct icommon`). This directory is not part of the regular file system name space, rather it is in its own dimension and is attached to ours via a worm-hole of function calls, such as `openat(2)` and `attropen(3C)`.

An excellent discussion of extended attributes can be found in `fsattr(5)`. This interface exists to support any extra attributes desired for files - this may be to support files from other file systems that require the storing of non-UFS attributes. Other uses will be discovered over time.

The following demonstration should get to the point quickly. Here we create an innocuous file, `tardis.txt`, and copy (yes, copy) several other files into its extended attribute name space, purely as a demonstration.

```
$ date > tardis.txt
$ ls -l tardis.txt
-rw-r--r--   1 user1    other         29 Apr  3 10:46 tardis.txt

$ runat tardis.txt cp /etc/motd /etc/group /usr/bin/ksh .
$ runat tardis.txt ls -l
total 352
-rw-r--r--   1 user1    other        286 Apr  3 10:47 group
-r-xr-xr-x   1 user1    other     171396 Apr  3 10:47 ksh
-rw-r--r--   1 user1    other         55 Apr  3 10:47 motd

$ ls -l tardis.txt
-rw-r--r--   1 user1    other         29 Apr  3 10:46 tardis.txt
$ ls -@ tardis.txt
-rw-r--r--@  1 user1    other         29 Apr  3 10:46 tardis.txt
$
$ du -ks tardis.txt
184     tardis.txt
```

The `runat tardis.txt ls -l` command is listing the contents of the extended attribute name space associated with tardis.txt, which now contains a copy of three files. Note that the final `ls -l tardis.txt` doesn't show any difference unless the `-@` option is used (displaying "@" in the same place where files with ACLs display "+"). The `-@` option is new to `ls(1)`, `cp(1)`, `tar(1)` and `cpio(1)`. The `find(1)` command has a `-xattr` option to find files that have extended attributes. The demonstration also shows that `du` is extended attribute aware.

Copying the `ksh` file was deliberate, as it allows us to journey to another world:

```
$ runat tardis.txt ./ksh
cannot access parent directories
$ ls -la
total 33136
drwxr-xr-x   2 user1    other        180 Apr  3 10:47 .
-rw-r--r--   1 user1    other   16777245 Apr  3 10:52 ..
-rw-r--r--   1 user1    other        286 Apr  3 10:47 group
-r-xr-xr-x   1 user1    other     171396 Apr  3 10:47 ksh
-rw-r--r--   1 user1    other         55 Apr  3 10:47 motd
$ pwd
cannot access parent directories
$ cd ..
./ksh: ..: not a directory
$ exit
```

Those security minded readers may imagine many entertaining abuses of extended attributes at this point. The can be turned off if needed, in Solaris 10 a `-noxattr` UFS mount option was added.

15.6 Locking in UFS

UFS uses two basic types of locks: `kmutex_t` and `krwlock_t`. The workings of these synchronization primitives is covered in Chapter 17. UFS locks can be divided into eight categories:

- Inode locks
- Queue locks
- ACL locks
- `VNODE` locks
- `VFS` locks
- `VOP_RWLOCK`
- `ufs_iuniqtime_lock`
- Logging locks

15.6.1 UFS Lock Descriptions

Tables 15.2 through 15.9 describe the UFS locks in more detail.

Table 15.2 Inode Locks

Name	Type	Description
`i_rwlock`	krwlock_t	• Serializes write requests. Allows reads to proceed in parallel. Serializes directory reads and updates. • Does not protect inode fields. • Indirectly protects block lists since it serializes allocations/deallocations in UFS. • Must be taken before starting UFS logging transactions if operating on a file; otherwise, taken after starting logging transaction.
`i_contents`	krwlock_t	• Protects most fields in the inode. • When held as a writer, protects all the fields protected by the `i_tlock`.
`i_tlock`	kmutex_t	• When held with the `i_contents` reader lock, protects the following inode fields: `i_utime`, `i_ctime`, `i_mtime`, `i_flag`, `i_delayoff`, `i_delaylen`, `i_nextrio`, `i_writes`, `i_writer`, `i_mapcnt`. • Also used as mutex for write throttling in UFS. • `i_contents` and `i_tlock` held together allows parallelism in updates.
`i_hlock`	kmutex_t	• Inode hash lock.

Table 15.3 Inode Queue Locks

Name	Type	Description
`ufs_scan_lock`	kmutex_t	• Synchronizes `ufs_scan_inodes` threads • `ufs_update()`, `ufs_sync()`, `ufs_scan_inodes()`. • Needed because of global inode list.
`ufs_q->uq_mutex`	krwlock_t	• Protects the two inode idle queues `ufs_junk_iq` and `ufs_useful_iq`.

continues

Table 15.3 Inode Queue Locks (*continued*)

Name	Type	Description
`ufs_hlock`	kmutex_t	• Used by the hlock thread. For more information, see man `lockfs(1M)`, hardlock section.
`ih_lock`	kmutex_t	• Protects the inode hash. The inode hash is global, per system, not per file system.

Table 15.4 Quota Queue Locks

Name	Type	Description
`dq_cachelock`	kmutex_t	• Protects the quota cache list. Prerequisite before taking the `dquot.dq_lock`.
`dq_freelock`	kmutex_t	• Protects the free quota list.
`dq_rwlock`	krwlock_t	• Protects the entire quota subsystem. • Taken as writer when the quota subsystem is initialized. Taken as reader when we do not want entire quota subsystem to be quiesced. • As writer, allows updates to quota-related fields in the `ufsvfs` structure. Also protects the `dquot` file as writer to allow quota updates. • As reader, allows reads from the quota-related fields in the `ufsvfs` structure.
`dqout.dq_lock`	kmutex_t	• Gives exclusive access to `dquot` struct.

Table 15.5 VNODE Locks

Name	Type	Description
`v_lock`	kmutex_t	• Protects the `vnode` fields. Also used by `VN_HOLD/VN_RELE`.

Table 15.6 ACL Locks

Name	Type	Description
`s_lock`	krwlock_t	• Protects the in-core shadow inode structure.

Table 15.7 VFS Locks

Name	Type	Description
vfs_lock	kmutex_t	• Locks contents of file system and cylinder groups. Also protects fields of the vfs_dio.
vfs_dqrwlock	krwlock_t	• Manages quota subsystem quiescence. • If held as writer, UFS quota subsystem may be experiencing changes in quotas, enabling/disabling of quotas, setting new quota limits. • Protects d_quot structure. This structure keeps track of all the enabled quotas per file system. • **Important note:** UFS shadow inodes that are used to hold ACL data and extended attribute directories are not counted against user quotas. Thus, this lock is not held for updates to these. • Reader held for this lock indicates to quota subsystem that major changes should not be occurring during that time. • Held when the i_contents writer lock is held, as described above, signifying that changes are occurring that affect user quotas. • Since UFS quotas can be enabled/disabled on the fly, this lock must be taken in all appropriate situations. It is not sufficient to check if the UFS quota subsystem is enabled before taking the lock.
ufsvfs_mutex	kmutex_t	• Protects access to the list that links all UFS file system instances. • Updates lists as a part of the mount operation. • Allows synchronization of all UFS file systems.

Table 15.8 `VOP_RWLOCK` or `ufs_rwlock`

Name	Type	Description
`ufs_rwlock()`	function	• Prevents concurrent reads and writes to a file. • Used by NFS when calling a `VOP_READDIR`, to prevent directory contents from changing. • NFS uses this lock to get attributes before and after a read or write to disable another operation from modifying the file.

Table 15.9 Logging Locks

Name	Type	Description
`mtm_lock`	kmutex_t	• Protects `mtm_taskq_sync_count` (keeps track of the number of pending `top_issue_sync` requests) field in `mt_map_t`.
`mtm_mutex`	kmutex_t	• Protects all the fields in the `mt_map_t` structure except `mtm_mapext` and `mtm_refcnt`.
`mtm_rwlock`	krwlock_t	• Protects `agenext_mapentry` field.
`un_log_mutex`	kmutex_t	• Allows one write to the log at a time. Part of `ml_unit_t` structure (in-core log data structure).
`un_state_mutex`	kmutex_t	• Allows one log state update at a time.

15.6.2 Inode Lock Ordering

Now that we are all familiar with the several different types of locks available in UFS, let us put them in order as if we were to work on an inode. Lock ordering is critical, and any mistake will more than likely cause the system to deadlock, and may end up panicking it!

Figure 15.16 give us a quick overview of lock ordering specific to the inode.

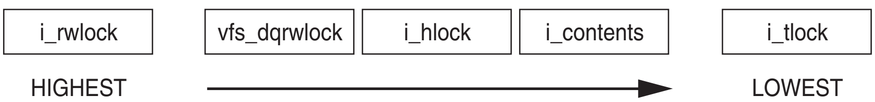

Figure 15.16 Inode Lock Ordering Precedence

15.6.3 UFS Lockfs Protocol

Along with basic inode locking, UFS also provides a mechanism to quiesce a file system for file system locking and for the forced unmounting of a file system. All VOPs (vnode operations) in UFS are required to follow the UFS lock protocol with `ufs_lockfs_begin()` and `ufs_lockfs_end()`, although the following functions purposely do not adhere to the tradition:

- `ufs_close`
- `ufs_putpage`
- `ufs_inactive`
- `ufs_addmap`
- `ufs_delmap`
- `ufs_rwlock`
- `ufs_rwunlock`
- `ufs_poll`

The basic principle here is that UFS supports various file system lock states (see list below) and each vnode operation must initiate the protocol by calling `ufs_lockfs_begin()` with an appropriate lock mask (a lock that this operation might grab while it is being processed) and end the protocol by calling `ufs_lockfs_end` before it returns. This way, UFS knows exactly how many vnode operations are in progress for the given file system by incrementing and decrementing the `ul_vnops_cnt` variable in the file-system-dependent `ulockfs` structure. If the file system is hard-locked, the thread gets an `EIO` error. If the file system is error-locked, then the thread is blocked.

Here are the file system locks and their actions.

- **Write lock.** Suspends writes that would modify the file system. Access times are not kept while a file system is write-locked.
- **Name lock.** Suspends accesses that could change or remove existing directories entries.
- **Delete lock.** Suspends access that could remove directory entries.
- **Hard lock.** Returns an error upon every access to the locked file system and cannot be unlocked. Hard-locked file systems can be unmounted. Hard lock supports forcible unmount.
- **Error lock.** Blocks all local access to the file system and returns `EWOULDBLOCK` on all remote access. File systems are error-locked by UFS upon detection of

internal inconsistency. They can only be unlocked after successful repair by `fsck`, which is usually done automatically. Error-locked file systems can be unmounted. Once the file system becomes clean, it can be upgraded to a hard lock.

- **Soft lock.** Quiesces a file system.
- **Unlock.** Awakens suspended accesses, releases existing locks, and flushes the file system.

While a vnode operation is being executed in UFS, a call can be made to another vnode function on the same UFS or a different UFS. This is called recursive VOP. The per-file system `vnode` operation counter is not incremented or decremented during recursive calls.

Here is the basic ordering to initiate and complete the lock protocol when operating on an inode in UFS.

```
1) Acquire i_rwlock (from the vnode layer in most cases).
2) Begin the UFS lock protocol by calling ufs_lockfs_begin().
3) Open UFS logging transactions if necessary now.
4) Acquire inode and quota locks (vfs_dqrwlock, i_contents, i_tlock, ...).
5) [work on inode]
6) Drop inode and quota locks (i_tlock, i_contents, vfs_dqrwlock, ...).
7) Close logging transactions.
8) End the UFS lock protocol by calling ufs_lockfs_end().
9) Release i_rwlock.
```

When working with directories, you need to make one minor change. `i_rwlock` is acquired after the logging transaction is initialized, and `i_rwlock` is released before the transaction is ended. Here are the steps.

```
1) Begin the UFS lock protocol by calling ufs_lockfs_begin().
2) Open UFS logging transactions if necessary now.
3) Acquire i_rwlock.
4) Acquire inode and quota locks (vfs_dqrwlock, i_contents, i_tlock, ...).
5) [work on inode]
6) Drop inode and quota locks (i_tlock, i_contents, vfs_dqrwlock, ...).
7) Release i_rwlock.
8) Close logging transactions.
9) End the UFS lock protocol by calling ufs_lockfs_end().
```

15.7 Logging

Important criteria for commercial systems are reliability and availability, both of which may be compromised if the file system does not provide the required level of robustness. We have become familiar with the term *journaling* to mean just one thing, but, in fact, file system logging can be implemented in several ways. The three most common forms of journaling are

- **Metadata logging.** Logs only file system structure changes
- **File and metadata logging.** Logs all changes to the file system
- **Log-structured file system.** Is an entire file system implemented as a log

The most common form of file system logging is metadata logging, and this is what UFS implements. When a file system makes changes to its on-disk structure, it uses several disconnected synchronous writes to make the changes. If an outage occurs halfway through an operation, the state of the file system is unknown, and the whole file system must be checked for consistency. For example, if the file is being extended the free block bitmap must be updated to mark the newly allocated block as no longer free. The inode block list must also be updated to indicate that the allocated block is owned by the file. If an outage occurs after the block is allocated, but before the inode is updated, file system inconsistency occurs.

A metadata logging file system such as UFS has an on-disk, cyclic, append-only log area that it can use to record the state of each disk transaction. Before any on-disk structures are changed, an intent-to-change record is written to the log. The directory structure is then updated, and when complete, the log entry is marked complete. Since every change to the file system structure is in the log, we can check the consistency of the file system by looking in the log, and we need not do a full file system scan. At mount time, if an intent-to-change entry is found but not marked complete the changes will not be applied to the file system. Figure 15.17 illustrates how metadata logging works.

Logging was first introduced in UFS in Solaris 2.4; it has come a long way since then, to being turned on by default in Solaris 10. Enabling logging turns the file system into a transaction-based file system. Either the entire transaction is applied or it is completely discarded. Logging is on by default in Solaris 10; however, it can be manually turned on by `mount(1M) -o` logging (using the `_FIOLOGENABLE` ioctl). Logging is not compatible with Solaris Logical Volume Manager (SVM) translogging, and attempt to turn on logging on a UFS file system that resides on an SVM will fail.

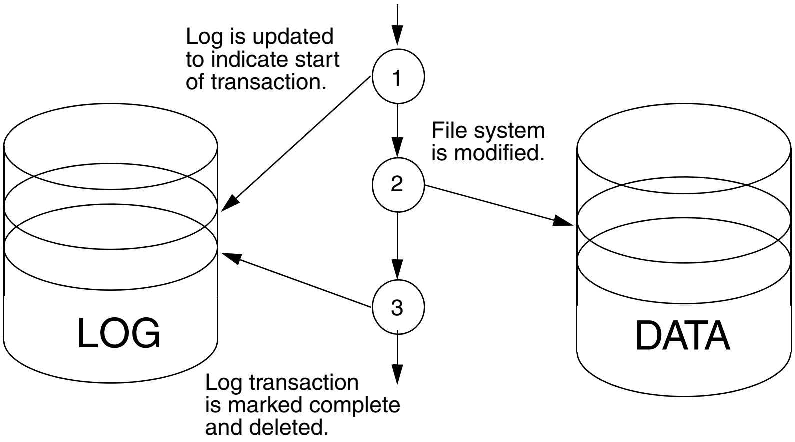

Figure 15.17 File System Metadata Logging

15.7.1 On-Disk Log Data Structures

The on-disk log is allocated from contiguous blocks where possible, and are only allocated as full sized file system blocks, no fragments are allowed. The initial pool of blocks is allocated when logging is first enabled on a file system, and blocks are not freed until logging is disabled. UFS uses these blocks for its own metadata and for times when it needs to store file system changes that have not yet been applied to the file system. This space on the file system is known as the “on disk” log, or log for short. It requires approximately 1 Mbyte per 1 Gbyte of file system space. The default minimum size for the log is 1 Mbyte, and the default maximum log size is 64 Mybtes. Figure 15.18 illustrates the on-disk log layout.

The file system superblock contains the block number where the main on-disk logging structure (`extent_block_t`) resides. This is defined by the `extent_block` structure. Note that the `extent_block` structure and all the accompanying `extent` structures fit within a file system block.

```
typedef struct extent_block {
        uint32_t        type;           /* Set to LUFS_EXTENTS to identify */
                                        /*   structure on disk. */
        int32_t         chksum;         /* Checksum over entire block. */
        uint32_t        nextents;       /* Size of extents array. */
        uint32_t        nbytes;         /* # bytes mapped by extent_block. */
        uint32_t        nextbno;        /* blkno of next extent_block. */
        extent_t        extents[1];
} extent_block_t;

                                See usr/src/uts/common/sys/fs/ufs_log.h
```

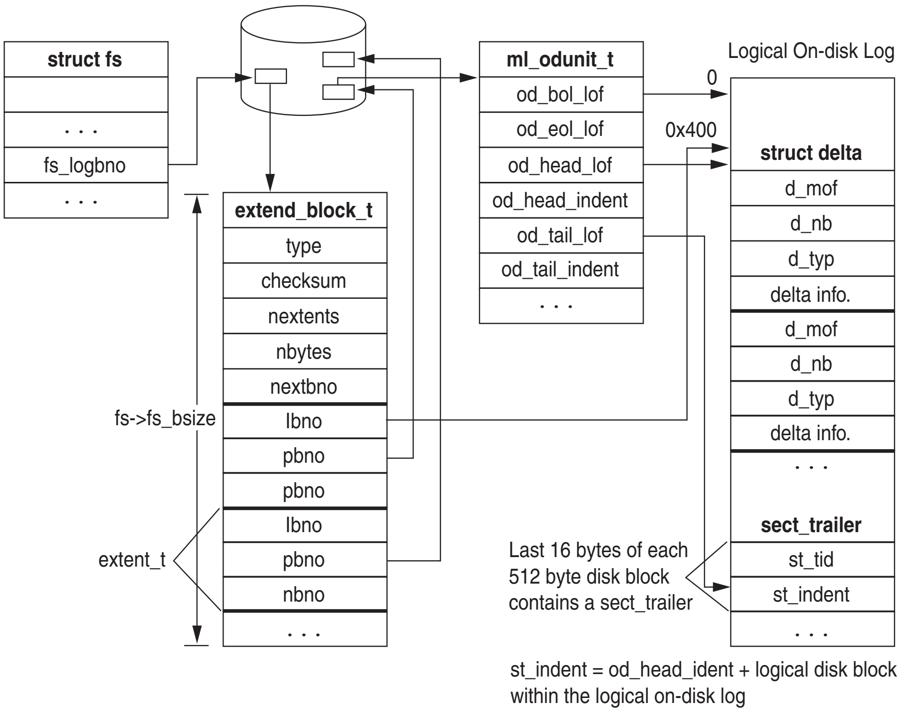

Figure 15.18 On-Disk Log Data Structure Layout

The `extent_block` structure describes logging metadata and is the main data structure used to find the on-disk log. It is followed by a series of extents that contain the physical block number for on-disk logging segments. The number of extents present for the file system is described by the `nextents` field in the `extent_block` structure.

```
typedef struct extent {
        uint32_t        lbno;   /* Logical block # within the space */
        uint32_t        pbno;   /* Physical block number of extent. */
                                /* in disk blocks for non-MTB ufs */
                                /* in frags for MTB ufs */
        uint32_t        nbno;   /* # blocks in this extent */
} extent_t;

                                        See usr/src/uts/common/sys/fs/ufs_log.h
```

Only the first extent structure is allowed to contain a `ml_odunit` structure (simplified: metadata logging on-disk unit structure).

```
typedef struct ml_odunit {
        uint32_t        od_version;     /* version number */
        uint32_t        od_badlog;      /* is the log okay? */
        uint32_t        od_unused1;

        /*
         * Important constants
         */
        uint32_t        od_maxtransfer; /* max transfer in bytes */
        uint32_t        od_devbsize;    /* device bsize */
        int32_t         od_bol_lof;     /* byte offset to begin of log */
        int32_t         od_eol_lof;     /* byte offset to end of log */

        /*
         * The disk space is split into state and circular log
         */
        uint32_t        od_requestsize; /* size requested by user */
        uint32_t        od_statesize;   /* size of state area in bytes */
        uint32_t        od_logsize;     /* size of log area in bytes */
        int32_t         od_statebno;    /* first block of state area */
        int32_t         od_unused2;

        /*
         * Head and tail of log
         */
        int32_t         od_head_lof;    /* byte offset of head */
        uint32_t        od_head_ident;  /* head sector id # */
        int32_t         od_tail_lof;    /* byte offset of tail */
        uint32_t        od_tail_ident;  /* tail sector id # */
        uint32_t        od_chksum;      /* checksum to verify ondisk contents */

        /*
         * Used for error recovery
         */
        uint32_t        od_head_tid;    /* used for logscan; set at sethead */

        /*
         * Debug bits
         */
        int32_t         od_debug;

        /*
         * Misc
         */
        struct timeval  od_timestamp;   /* time of last state change */
} ml_odunit_t;
```

See usr/src/uts/common/sys/fs/ufs_log.h

The values in the `ml_odunit_t` structure represent the location, usage and state of the on-disk log. The contents in the on-disk log consist of delta structures, which define the changes, followed by the actual changes themselves. Each 512 byte disk block of the on-disk log will contain a `sect_trailer` at the end of the block. This `sect_trailer` is used to identify the disk block as containing valid deltas. The `*_lof` fields reference the byte offset in the logical on-disk layout and not the physical on-the-disk contents.

```
struct delta {
        int64_t         d_mof;  /* byte offset on device to start writing */
                                /*   delta */
        int32_t         d_nb;   /* # bytes in the delta */
        delta_t         d_typ;  /* Type of delta.  Defined in ufs_trans.h */
};
```

See usr/src/uts/common/sys/fs/ufs_log.h

```
typedef struct sect_trailer {
        uint32_t        st_tid;         /* transaction id */
        uint32_t        st_ident;       /* unique sector id */
} sect_trailer_t;
```

See usr/src/uts/common/sys/fs/ufs_log.h

15.7.2 In-Core Log Data Structures

Figure 15.19 illustrates the data structures for in-core logging.

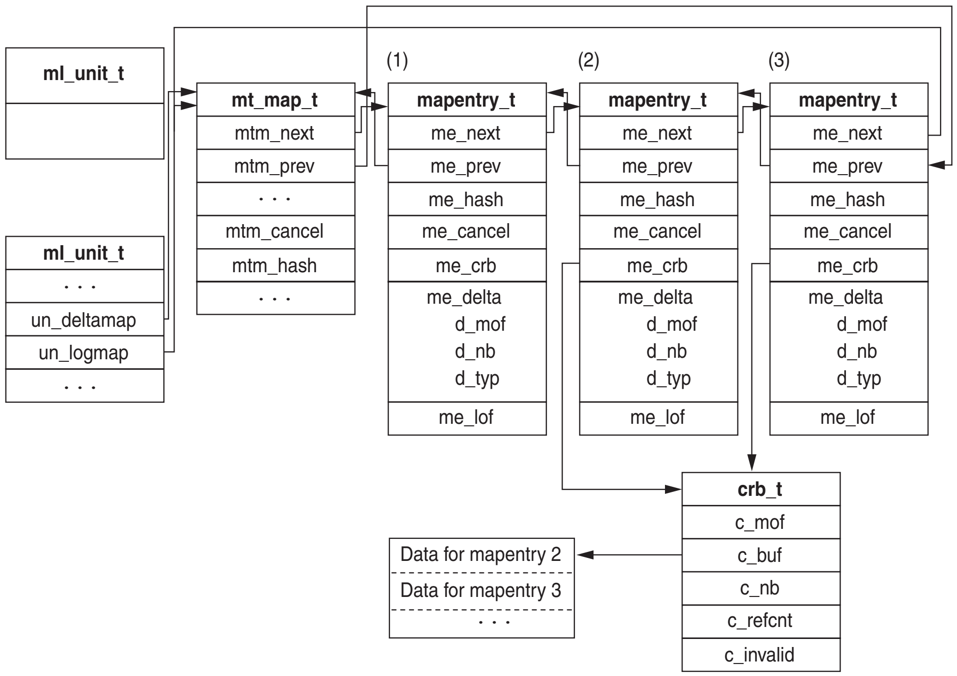

Figure 15.19 In-Core Log Data Structure Layout

ml_unit_t is the main in-core logging structure. There is only one per file system, and it contains all logging information or pointers to all logging data structures for the file system. The un_ondisk field contains an in-memory replica of the on-disk ml_odunit structure.

```
typedef struct ml_unit {
        struct ml_unit  *un_next;       /* next incore log */
        int             un_flags;       /* Incore state */
        buf_t           *un_bp;         /* contains memory for un_ondisk */
        struct ufsvfs   *un_ufsvfs;     /* backpointer to ufsvfs */
        dev_t           un_dev;         /* for convenience */
        ic_extent_block_t *un_ebp;      /* block of extents */
        size_t          un_nbeb;        /* # bytes used by *un_ebp */
        struct mt_map   *un_deltamap;   /* deltamap */
        struct mt_map   *un_logmap;     /* logmap includes moby trans stuff */
        struct mt_map   *un_matamap;    /* optional - matamap */

        /*
         * Used for managing transactions
         */
        uint32_t        un_maxresv;     /* maximum reservable space */
        uint32_t        un_resv;        /* reserved byte count for this trans */
        uint32_t        un_resv_wantin; /* reserved byte count for next trans */

        /*
         * Used during logscan
         */
        uint32_t        un_tid;

        /*
         * Read/Write Buffers
         */
        cirbuf_t        un_rdbuf;       /* read buffer space */
        cirbuf_t        un_wrbuf;       /* write buffer space */

        /*
         * Ondisk state
         */
        ml_odunit_t     un_ondisk;      /* ondisk log information */

        /*
         * locks
         */
        kmutex_t        un_log_mutex;   /* allows one log write at a time */
        kmutex_t        un_state_mutex; /* only 1 state update at a time */
} ml_unit_t;
```

See usr/src/uts/common/sys/fs/ufs_log.h

mt_map_t tracks all the deltas for the file system. At least three mt_map_t structures are defined:

- **deltamap.** Tracks all deltas for currently active transactions. When a file system transaction completes, all deltas from the delta map are written to the log map and all the entries are then removed from the delta map.

- **logmap.** Tracks all committed deltas from completed transactions, not yet applied to the file system.
- **matamap.** Is the debug map for delta verification.

See usr/src/uts/common/sys/fs/ufs_log.h for the definition of mt_map structure

```
 struct mapentry {
        /*
         * doubly linked list of all mapentries in map -- MUST BE FIRST
         */
        mapentry_t      *me_next;
        mapentry_t      *me_prev;

        mapentry_t      *me_hash;
        mapentry_t      *me_agenext;
        mapentry_t      *me_cancel;
        crb_t           *me_crb;
        int             (*me_func)();
        ulong_t         me_arg;
        ulong_t         me_age;
        struct delta    me_delta;
        uint32_t        me_tid;
        off_t           me_lof;
        ushort_t        me_flags;
};
```

See usr/src/uts/common/sys/fs/ufs_log.h

The mapentry structure defines changes to filesystem metadata. All existing mapentries for a given mt_map are linked into the mt_amp at the mtm_next and mtm_prev fields. The mtm_hash field of the mt_map is a hash list of all the mapentries, hashed according to the master byte offset of the delta on the file system and the MAPBLOCKSIZE. For example, the MTM_HASH macro determines the hash list in which a mapentry for the offset mof (where mtm_nhash is the total number of hash lists for the map). The default size used for MAPBLOCKSIZE is 8192 bytes, the hash size for the delta map is 512 bytes, and the hash size for the log map is 2048 bytes.

```
#define MAP_INDEX(mof, mtm) \
        (((mof) >> MAPBLOCKSHIFT) & (mtm->mtm_nhash-1))
#define MAP_HASH(mof, mtm) \
        ((mtm)->mtm_hash + MAP_INDEX((mof), (mtm)))
```

See usr/src/uts/common/sys/fs/ufs_log.h

A canceled mapentry with the ME_CANCEL bit set in the me_flags field is a special type of mapentry. This type of mapentry is basically a place holder for free blocks and fragments. It can also represent an old mapentry that is no longer valid due to a new mapentry for the same offset. Freed blocks and fragments are not eligible for reallocation until all deltas have been written to the on-disk log. Any attempt to allocate a block or fragment in which a corresponding canceled mapentry exists in the logmap, results in the allocation of a different block or fragment.

```
typedef struct crb {
        int64_t         c_mof;          /* master file offset of buffer */
        caddr_t         c_buf;          /* pointer to cached roll buffer */
        uint32_t        c_nb;           /* size of buffer */
        ushort_t        c_refcnt;       /* reference count on crb */
        uchar_t         c_invalid;      /* crb should not be used */
} crb_t;

                                                    See sys/fs/ufs_log.h
```

The crb_t, or cache roll buffer, caches blocks that exist within the same disk-block. It is merely a performance enhancement when information is rolled back to the file system. It helps reduce reads and writes that can occur while writing completed transactions deltas to the file system. It also acts as a performance enhancement on read hits of deltas.

UFS logging maintains private buf_t structures used for reading and writing of the on-disk log. These buf_t structures are managed through cirbuf_t structures. Each file system will have 2 cirbuf_t structures. One is used to manage log reads, and one to manage log writes.

```
typedef struct cirbuf {
        buf_t           *cb_bp;         /* buf's with space in circular buf */
        buf_t           *cb_dirty;      /* filling this buffer for log write */
        buf_t           *cb_free;       /* free bufs list */
        caddr_t         cb_va;          /* address of circular buffer */
        size_t          cb_nb;          /* size of circular buffer */
        krwlock_t       cb_rwlock;      /* r/w lock to protect list mgmt. */
} cirbuf_t;

                                                    See sys/fs/ufs_log.h
```

15.7.3 Summary Information

Summary information is critical to maintaining the state of the file system. Summary information includes counts of directories, free blocks, free fragments, and free inodes. These bits of information exist in each cylinder group and are valid only for that respective cylinder group. All cylinder group summary information is totaled; these numbers are kept in the `fs_cstotal` field of the superblock. A copy of all the cylinder group's summary information is also kept in a buffer pointed to from the file system superblock's `fs_csp` field. Also kept on disk for redundancy is a copy of the `fs_csp` buffer, whose block address is stored in the `fs_csaddr` field of the file system superblock.

All cylinder group information can be determined from reading the cylinder groups, as opposed to reading them from `fs_csaddr` blocks on disk. Hence, updates to `fs_csaddr` are logged only for large file systems (in which the total number of cylinder groups exceeds `ufs_ncg_log`, which defaults to 10,000). If a file system isn't logging deltas to the `fs_csaddr` area, then the `ufsvfs->vfs_nolog_si` is set to 1 and instead marks the `fs_csaddr` area as bad by setting the superblock's `fs_si` field to `FS_SI_BAD`. However, these changes are brought up to date when an unmount or a log roll takes place.

15.7.4 Transactions

A transaction is defined as a file system operation that modifies file system metatdata. A group of these file system transactions is known as a moby transaction.

Logging transactions are divided into two types:

- **Synchronous file system transactions** are those that are committed and written to the log as soon as the file system transaction ends.
- **Asynchronous file system transactions** are those for which the file system transactions are committed and written to the on-disk log after closure of the moby transaction. In this case the file system transaction may complete, but the metadata that it modified is not written to the log and not considered commited until the moby transaction has been completed.

So what exactly are committed transactions? Well, they are transactions whose deltas (unit changes to the file system) have been moved from the delta map to the log map and written to the on-disk log.

There are four steps involved in logging metadata changes of a file system transaction:

1. Reserve space in the log.
2. Begin a file system transaction.
3. Enter deltas in the delta map for all the metadata changes.
4. End the file system transaction.

15.7.4.1 Reserving Space in the Log

A file system transaction that is to log metadata changes should first reserve space in the log. This prevents hangs if the on-disk log is full. A file system transaction that is part of the current moby transaction can not complete if there isn't enough log space to log the deltas. Log space can not be reclaimed until the current moby transation completes and is committed. And the current moby transaction can't complete until all file system transaction in the current moby transaction complete. Thus reserving space in the log must be done by the file system transaction when it enters the current moby transation. If there is not enough log space available, the file system transaction will wait until sufficient log space becomes available, before entereing the the current moby transaction.

The amount of space reserved in the log for write and truncation vary, depending on the size of the operation. The macro TRANS_WRITE_RESV estimates how much log space is needed for the operation.

```
#define TRANS_WRITE_RESV(ip, uiop, ulp, resvp, residp)  \
        if ((TRANS_ISTRANS(ip->i_ufsvfs) != NULL) && (ulp != NULL)) \
                ufs_trans_write_resv(ip, uiop, resvp, residp);
                                                     See sys/fs/ufs_trans.h
```

All other file system transactions have a constant transaction size, and UFS has predefined macros for these operations:

```
/*
 * size calculations
 */
#define TOP_CREATE_SIZE(IP)     \
        (ACLSIZE(IP) + SIZECG(IP) + DIRSIZE(IP) + INODESIZE)
#define TOP_REMOVE_SIZE(IP)     \
        DIRSIZE(IP)  + SIZECG(IP) + INODESIZE + SIZESB
#define TOP_LINK_SIZE(IP)       \
        DIRSIZE(IP) + INODESIZE
#define TOP_RENAME_SIZE(IP)     \
        DIRSIZE(IP) + DIRSIZE(IP) + SIZECG(IP)
#define TOP_MKDIR_SIZE(IP)      \
        DIRSIZE(IP) + INODESIZE + DIRSIZE(IP) + INODESIZE + FRAGSIZE(IP) + \
            SIZECG(IP) + ACLSIZE(IP)
#define TOP_SYMLINK_SIZE(IP)    \
        DIRSIZE((IP)) + INODESIZE + INODESIZE + SIZECG(IP)
#define TOP_GETPAGE_SIZE(IP)    \
        ALLOCSIZE + ALLOCSIZE + ALLOCSIZE + INODESIZE + SIZECG(IP)
#define TOP_SYNCIP_SIZE         INODESIZE
#define TOP_READ_SIZE           INODESIZE
#define TOP_RMDIR_SIZE          (SIZESB + (INODESIZE * 2) + SIZEDIR)
#define TOP_SETQUOTA_SIZE(FS)   ((FS)->fs_bsize << 2)
#define TOP_QUOTA_SIZE          (QUOTASIZE)
#define TOP_SETSECATTR_SIZE(IP) (MAXACLSIZE)
#define TOP_IUPDAT_SIZE(IP)     INODESIZE + SIZECG(IP)
#define TOP_SBUPDATE_SIZE       (SIZESB)
```

continues

```
#define TOP_SBWRITE_SIZE        (SIZESB)
#define TOP_PUTPAGE_SIZE(IP)    (INODESIZE + SIZECG(IP))
#define TOP_SETATTR_SIZE(IP)    (SIZECG(IP) + INODESIZE + QUOTASIZE + \
                ACLSIZE(IP))
#define TOP_IFREE_SIZE(IP)      (SIZECG(IP) + INODESIZE + QUOTASIZE)
#define TOP_MOUNT_SIZE          (SIZESB)
#define TOP_COMMIT_SIZE         (0)
sys/fs/ufs_trans.h
```

15.7.4.2 Starting Transactions

Starting a transaction simply means that the transaction has successfully entered the current moby transaction. As a result, once started, the moby will not end until all active file system transactions have completed. A moby transaction can accommodate both synchronous and asynchronous transactions. Most file system transactions in UFS are asynchronous; however, a synchronous transaction occurs if any of the following are true:

- If the file system is mounted `syncdir`
- If a `fsync()` system call is executed
- If `DSYNC` or `O_SYNC` open modes are set on reads and writes
- If `RSYNC` is set on reads
- During an unmount of a file system

A transaction can be started with one of the following macros:

- `TRANS_BEGIN_ASYNC`—Enters a file system transaction into the current moby transaction. Once the file system transaction ends, the moby transaction may still be active and hence the changes the file system transaction has made have not yet been committed.

```
#define TRANS_BEGIN_ASYNC(ufsvfsp, vid, vsize)\
{\
        if (TRANS_ISTRANS(ufsvfsp))\
                (void) top_begin_async(ufsvfsp, vid, vsize, 0); \
}

                                        See sys/fs/ufs_trans.h
```

- **`TRANS_BEGIN_SYNC`.** Enters a file system transaction into the current moby transaction with the requirement that the completion of the file system transaction forces a completion and commitment of the moby transaction. All file system transactions that have occurred within the moby transaction are also considered as committed.

```
#define TRANS_BEGIN_SYNC(ufsvfsp, vid, vsize, error)\
{\
        if (TRANS_ISTRANS(ufsvfsp)) { \
                error = 0; \
                top_begin_sync(ufsvfsp, vid, vsize, &error); \
        } \
}
                                                          See sys/fs/ufs_trans.h
```

- **TRANS_BEGIN_CSYNC.** Does a TRANS_BEGIN_SYNC if the mount option syncdir is set; otherwise, does a TRANS_BEGIN_ASYNC.
- **TRANS_TRY_BEGIN_ASYNC and TRANS_TRY_BEGIN_CSYNC.** Try to enter the file system transaction into the moby transaction. If the result would cause the thread to block, then do not block and return EWOULDBLOCK instead. This macro is used in cases where the calling thread must not block.

```
#define TRANS_TRY_BEGIN_ASYNC(ufsvfsp, vid, vsize, err)\
{\
        if (TRANS_ISTRANS(ufsvfsp))\
                err = top_begin_async(ufsvfsp, vid, vsize, 1); \
        else\
                err = 0; \
}

#define TRANS_TRY_BEGIN_CSYNC(ufsvfsp, issync, vid, vsize, error)\
{\
        if (TRANS_ISTRANS(ufsvfsp)) {\
                if (ufsvfsp->vfs_syncdir) {\
                        ASSERT(vsize); \
                        top_begin_sync(ufsvfsp, vid, vsize, &error); \
                        ASSERT(error == 0); \
                        issync = 1; \
                } else {\
                        error = top_begin_async(ufsvfsp, vid, vsize, 1); \
                        issync = 0; \
                }\
        }\
}
                                    See usr/src/uts/common/sys/fs/ufs_trans.h
```

15.7.4.3 Ending the Transaction

Once all metadata changes have been completed, the transaction must be ended. This is accomplished by calling one of the following macros:

- **TRANS_END_CSYNC.** Calls TRANS_END_ASYNC or TRANS_END_SYNC, depending on which type of file system transaction was initially started.
- **TRANS_END_ASYNC.** Ends an asynchronous file system transaction. If, at this point, the log is getting full, (the number of mapentries in the logmap is greater than the global variable logmap_maxnme_async) committed deltas

in the log will be applied to the file system and removed from the log. This is known as "rolling the log" and is done in by a seperate thread.

```
#define TRANS_END_ASYNC(ufsvfsp, vid, vsize)\
{\
        if (TRANS_ISTRANS(ufsvfsp))\
                top_end_async(ufsvfsp, vid, vsize); \
}
                                        See usr/src/uts/common/sys/fs/ufs_trans.h
```

- **TRANS_END_SYNC.** Closes and commits the current moby transaction, and writes all deltas to the on-disk log. A new moby transaction is then started.

```
#define TRANS_END_SYNC(ufsvfsp, error, vid, vsize)\
{\
        if (TRANS_ISTRANS(ufsvfsp))\
                top_end_sync(ufsvfsp, &error, vid, vsize); \
}
                                        See usr/src/uts/common/sys/fs/ufs_trans.h
```

15.7.5 Rolling the Log

Occasionally, the data in the log needs to be written back to the file system, a procedure called log rolling. Log rolling occurs for the following reasons:

- To update the on-disk file system with committed metadata deltas
- To free space in the log for new deltas
- To roll the entire log to disk at unmount
- To partially roll the on-disk log when it is getting full
- To completely roll the log with the `_FIOFFS` ioctl (file system flush)
- To partially roll the log every 5 seconds when no new deltas exist in the log
- To roll some deltas when the log map is getting full (that is, when `logmap` has more than `logmap_maxnme` mapentries, by default, 1536)

The actual rolling of the log is handled by the log roll thread, which executes the `trans_roll()` function found in `usr/src/uts/common/fs/lufs_thread.c`. The `trans_roll()` function preallocates a number of `rollbuf_t` structures (based on `LUFS_DEFAULT_NUM_ROLL_BUF` = 16, `LUFS_DEFAULT_MIN_ROLL_BUFS` = 4, `LUFS_DEFAULT_MAX_ROLL_BUFS` = 64) to handle rolling deltas from the log to the file system.

```
typedef uint16_t rbsecmap_t;
typedef struct rollbuf {
        buf_t rb_bh;            /* roll buffer header */
        struct rollbuf *rb_next; /* link for mof ordered roll bufs */
        crb_t *rb_crb;          /* cached roll buffer to roll */
        mapentry_t *rb_age;     /* age list */
        rbsecmap_t rb_secmap;   /* sector map */
} rollbuf_t;
                                        See usr/src/uts/common/sys/fs/ufs_log.h
```

Along with allocating memory for the `rollbuf_t` structures, `trans_roll` also allocates `MAPBLOCKSIZE` * `lufs_num_roll_bufs` bytes to be used by `rollbuf_t`'s `buf_t` structure stored in `rb_bh`. These `rollbuf_t`'s are populated according to information found in the rollable mapentries of the logmap. All rollable mapentries will be rolled starting from the logmap's `un_head_lof` offset, and continuing until an unrollable mapentry is found. Once a rollable mapentry is found, all other rollable mapentries within the same `MAPBLOCKSIZE` segment on the file system device are located and mapped by the same rollbuf structure.

If all mapentries mapped by a rollbuf have the same cache roll buffer (`crb`), then this `crb` maps the on-disk block and buffer containing the deltas for the rollbuf's `buf_t`. Otherwise, the rollbuf's `buf_t` uses `MAPBLOCKSIZE` bytes of kernel memory allocated by the `trans_roll` thread to do the transfer. The `buf_t` reads the `MAPBLOCKSIZE` bytes on the file system device into the `rollbuf` buffer. The deltas defined by each mapentry overlap the old data read into the `rollbuf` buffer. This buffer is then writen to the file system device.

If the rollbufs contain holes, these rollbufs may have to issue more than one write to disk to complete writing the deltas. To asynchronously write these deltas, the rollbuf's `buf_t` structure is cloned for each additional write required for the given rollbuf. These cloned `buf_t` structures are linked into the rollbuf's `buf_t` structure at the `b_list` field. All writes defined by the rollbuf's `buf_t` structures and any clone `buf_t` structures are issued asynchronously.

The `trans_roll()` thread waits for all these writes to complete. If any fail, a warning is printed to the console and the log is marked as `LDL_ERROR` in the `logmap->un_flags` field. If the roll completes successfully, all corresponding mapentries are completely removed from the log map. The head of the log map is then adjusted to reflect this change, as illustrated in Figure 15.20.

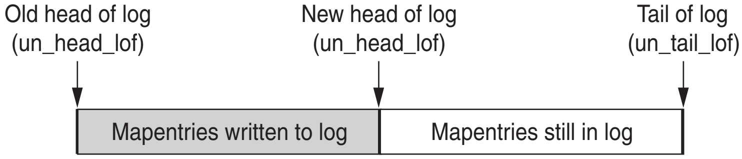

Figure 15.20 Adjustment of Head of Log Map

15.7.6 Redirecting Reads and Writes to the Log

When the UFS module is loaded, the global variable bio_lufs_strategy is set to point to the lufs_strategy() function. As a result, bread_common() and bwrite_common() functions redirect reads and writes to the bio_lufs_strategy (if it exists and if logging is enabled). lufs_strategy() then determines if the I/O request is a read or a write and dispatches to either lufs_read_strategy() or lufs_write_strategy(). These functions are responsible for resolving the read/write request from and to the log. In some instances in UFS, the functions lufs_read_strategy() and lufs_write_strategy() are called directly, bypassing the bio_lufs_strategy() code path.

15.7.6.1 `lufs_read_strategy()` Behavior

The lufs_read_strategy() function is called for reading metadata in the log. Mapentries already in the log map that correspond to the requested byte range are linked in the me_agenext list and have the ME_AGE bit set to indicate that they are in use. If the bytes being read are not defined in a logmap mapentry, the data is read from the file system as normal. Otherwise, lufs_read_strategy() then calls ldl_read() to read the data from the log.

The function ldl_read() can get the requested data from a variety of sources:

- A cache roll buffer
- The write buffer originally used to write this data to the log (mlunit->un_wrbuf)
- The buffer previously used to read this data from the log (mlunit->un_rdbuf)
- The on-disk log itself

15.7.6.2 `lufs_write_strategy()` Behavior

The lufs_write_strategy() function writes deltas defined by mapentries from the delta map to the log map if any exist. It does so by calling logmap_add() or logmap_add_buf(). logmap_add_buf() is used when crb buffers are being used, otherwise logmap_add() is used. These function in turn call ldl_write() to actually write the data to log.

The function ldl_write() always writes data into the the memory buffer of the buf_t contained in the write cirbuf_t structure. Hence, requested writes may or may not always actually be written to the physical on-disk log. Writes to the physical on-disk log occur when the log rolls the tail around back to the head, the write buf_t buffer is full, or a commit record is written.

15.7.7 Failure Recovery

An important aspect of file system logging is the ability to recover gracefully after an abnormal operating system halt. When the operating system is restarted and the file system remounted, the logging implementation will complete any outstanding operations by replaying the commited log transactions. The on-disk log is read and any commited deltas found are populated into the logmap as committed logmap mapentries. The roll thread will then write these to the file system and remove the mapentries from the logmap. All uncommitted deltas found in the ondisk log will be discarded.

15.7.7.1 Reclaim Thread

A system panic can leave inodes in a partially deleted state. This panic can be caused by an interrupted delete thread (refer to Section 15.3.2 for more information on the delete thread) in which `ufs_delete()` never finished processing the inode. The sole purpose of the UFS reclaim thread (`ufs_thread_reclaim()` in `usr/src/uts/common/fs/ufs/ufs_thread.c`) is to clean up the inodes left in this state. This thread is started if the superblock's `fs_reclaim` field has either `FS_RECLAIM` or `FS_RECLAIMING` flags set, indicating that freed inodes exist or that the reclaim thread was previously running.

The reclaim thread reads each on-disk inode from the file system device, checking for inodes whose `i_nlink` is zero and `i_mode` isn't zero. This situation signifies that `ufs_delete()` never finished processing these inodes. The thread simply calls `VN_RELE()` for every inode in the file system. If the node was partially deleted, the `VN_RELE()` forces the inode to go through `ufs_inactive()`, which in turn queues the inode in the `vfs_delete` queue to be processed later by the delete thread.

15.8 MDB Reference

Table 15.10 UFS MDB Reference

dcmd or walker	**Description**
dcmd acl	Given an inode, display its in core acl's
dcmd cg	Display a summarized cylinder group structure
dcmd inode	Display summarized inode_t

continues

Table 15.10 UFS MDB Reference (*continued*)

dcmd or walker	**Description**
dcmd inode_cache	Search/display inodes from inode cache
dcmd mapentry	Dumps ufslog mapentry
dcmd mapstats	Dumps ufslog stats
walk acl	Given an inode, walk chains of in core acl's
walk cg	Walk cg's in bio buffer cache
walk inode_cache	Walk inode cache
walk_ufslogmap	Walk the log map
walk ufs_inode_cache	Walk the ufs_inode_cache cache