M[a]fs - Plan 9 userspace file systems

Mfs and Mafs wants you to be able to understand it, so you can be self-sufficient and fix a crash at two in the morning or satisfy your need for speed or a feature. This empowerment is priceless for those with skin in the game.

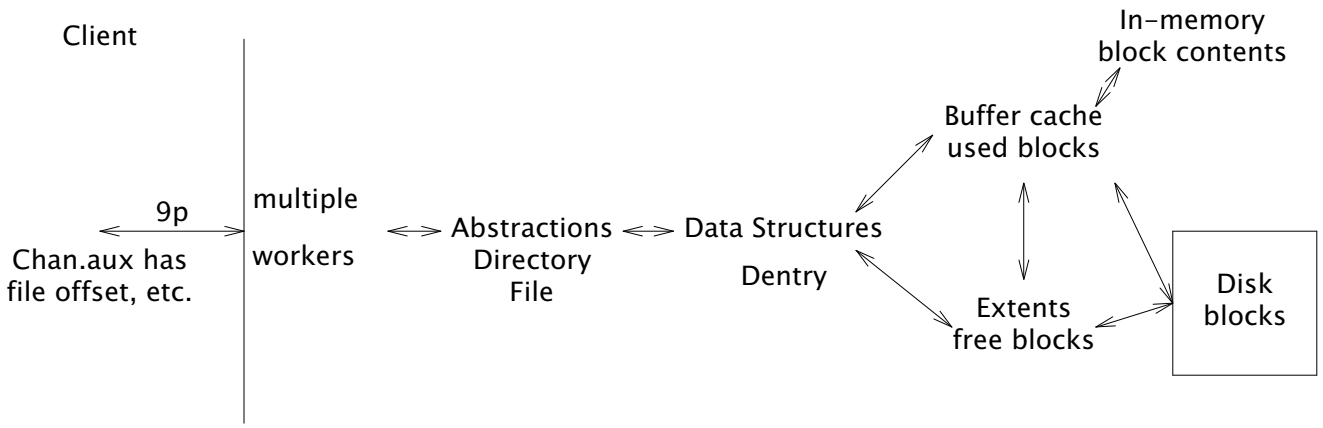
Mfs and Mafs are user space file systems to provide system stability and security. They are based on kfs.

As this document aims to also provide working knowledge, it gratuitously uses the actual commands and the relevant C data structure definitions to convey information.

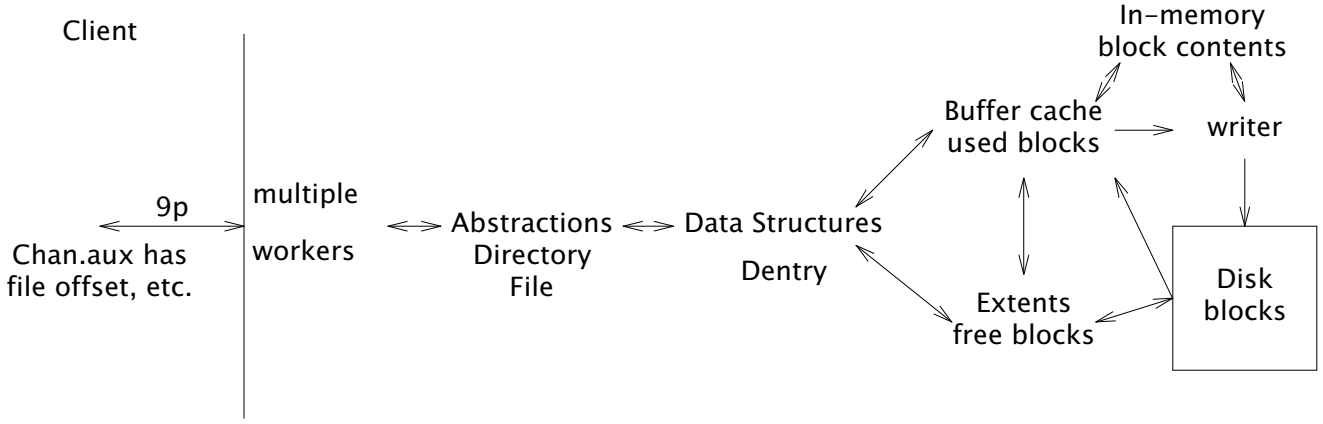
Mfs writes synchronously to the disk. It ensures that what has been said to be written has been passed along to the disk driver. Mafs writes asynchronously to the disk. It stores the writes in memory for a writer process to write to the disk at leisure.

This document uses the word M[a]fs to refer to both mfs and mafs.

Mfs Workflow



Mafs Workflow

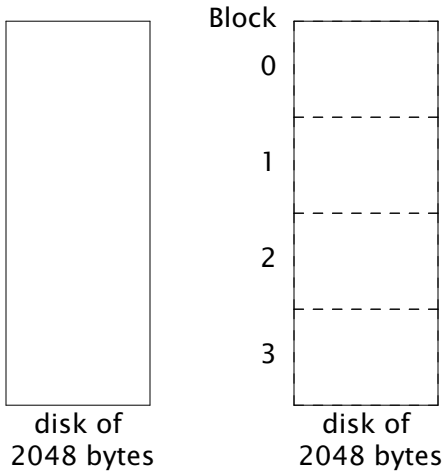


Disk Contents

M[a]fs organizes and saves content on a disk as directories and files, just like any other filesystem.

The unit of storage is a logical block (not physical sector) of data. Disk space is split into 512 byte logical blocks.

A sample disk of 2048 bytes with 4 blocks.



A block is stored to the disk with a Tag.

```
struct Tag
{
	u8 type;	/* Tfree, Tmagic, Tdentry, Tdata, Tindn */
	u64 path; /* Qid.path, unique identifier of directory or file */
};
```

Every file or directory is represented on the disk by a directory entry (Dentry). A directory entry uses a block (Tag.type = Tdentry) and is uniquely identifiable by a Qid.

A file stores its contents in blocks with a Tag.type of Tdata. A directory holds the directory entries of it's children in blocks with a Tag.type of Tdentry.

The blocks used by a file or directory entry are listed in their directory entry. As it is not possible to represent big files using the list of blocks available in the directory entry, the blocks are structured to use multiple levels of indirection as the file size increases.

A file's data blocks are identified by a tag of Tdata and that file's Qid.path. A directory's data blocks are identified by a tag of Tdentry and Qid.path of the child directory entry. (Is this quirky? Should the child's directory entry have a tag with the parent's Qid.path?)

A block number of zero represents the end of the file's contents. If a file is truncated, the data and indirect blocks are given up and the dentry.dblocks[0] = 0.

M[a]fs does not store the last access time of a file or directory.

The different types of blocks on a disk are:

```
enum
{
	Tfree  = 0,	/* free block */
	Tmagic,		/* the first (zero'th) block holds a magic word */
	Tdentry,	/* directory entry */
			/* Tind*n* are last, to allow for future increases */
	Tdata,		/* actual file contents */
	Tind0,		/* contains a list of Tdata block numbers for files
			    and Tdentry block numbers for directories.*/
	Tind1,		/* contains a list of Tind0 block numbers */
	Tind2,		/* contains a list of Tind1 block numbers */
	Tind3,		/* contains a list of Tind2 block numbers */
	Tind4,		/* contains a list of Tind3 block numbers */
	Tind5,		/* contains a list of Tind4 block numbers, maximum file size 26 TiB */
};
```

A directory entry is defined as:

```
enum {
	Rawblocksize = 512ULL,  /* real block size */
	Ndblock   = 32,		/* number of direct blocks in a Dentry */
	Niblock   = 6,		/* maximum depth of indirect blocks */
};
struct Qid9p1
{
	u32 version;
	u64 path;	/* unique identifier */
};

struct Dentry1
{
	Qid9p1	qid;
	u64 size;	/* 0 for directories. For files, size in bytes of the content */
	u64 pdblkno;	/* parent dentry absolute block number. 0 for root. */
	u64 pqpath;	/* parent qid.path */
	u64 mtime;	/* modified time in nano seconds from epoch */
	u32 mode;	/* same bits as defined in lib.h Dir.mode */
	s16 uid;
	s16 gid;
	s16 muid;
	u64 dblocks[Ndblock];	/* direct blocks. */
				/* List of Tdata block numbers for files and
				    Tdentry block numbers for directories */
				/* Tag.type = Tdentry for directories and Tdata for files */
	u64 iblocks[Niblock];	/* indirect blocks */
};

/*
 * Derived constants
 * Ndentryperblock: number of directory entries per block
 * Nindperblock: number of block pointers per block
 */
```

```
enum {
	Blocksize  = Rawblocksize - sizeof(Tag),
	Namelen  = (Blocksize-sizeof(Dentry1)),  /* maximum size of the name of a file or directory */

	Ndentryperblock	= 1,			/* Blocksize / sizeof(Dentry), */
	Nindperblock  = Blocksize / sizeof(u64),
};
struct Dentry
{
	struct Dentry1;
	char name[Namelen];
};
```

A directory entry once assigned is not given up until the parent directory is removed. It is zero'ed if the directory entry is removed. It is reused by the next directory entry created under that parent directory. This removes the need for garbage collection of directory entries on removals and also avoids zero block numbers in the middle of a directory entry's list of blocks. A zero block number while traversing a directory entry's dblocks or iblocks represents the end of directory or file contents. When a directory is removed, the parent will have a directory entry with a tag of Tdentry and Qpnone and the rest of the contents set to zero.

A directory's size is always zero.

```
	tests/6.sizes # shows the values of the above derived variables.
	Namelen 145 Ndblock 32 Niblock 6
	Blocksize 503 Nindperblock 62
	A Tind0 unit points to 1 data blocks (503 bytes)
		block points to 62 data blocks
		reli start 32	max 93
		max size 94*Blocksize = 47282 bytes
	A Tind1 unit points to 62 data blocks (31186 bytes)
		block points to 3844 data blocks
		reli start 94	max 3937
		max size 3938*Blocksize = 1980814 bytes	= 1 MiB
	A Tind2 unit points to 3844 data blocks (1933532 bytes)
		block points to 238328 data blocks
		reli start 3938  max 242265
		max size 242266*Blocksize = 121859798 bytes	= 116 MiB
	A Tind3 unit points to 238328 data blocks (119878984 bytes)
		block points to 14776336 data blocks
		reli start 242266	max 15018601
		max size 15018602*Blocksize = 7554356806 bytes	= 7 GiB
	A Tind4 unit points to 14776336 data blocks (7432497008 bytes)
		block points to 916132832 data blocks
		reli start 15018602  max 931151433
		max size 931151434*Blocksize = 468369171302 bytes = 436 GiB
	A Tind5 unit points to 916132832 data blocks (460814814496 bytes)
		block points to 56800235584 data blocks
		reli start 931151434	max 57731387017
		max size 57731387018*Blocksize = 29038887670054 bytes = 26 TiB
```

On an empty m[a]fs filesystem mounted at /n/mafs, the disk contents added by the below commands are:
mkdir /n/mafs/dir1
echo test > /n/mafs/dir1/file1

Representation of a file in a directory: /dir1/file1

Block 18 contents: /dir1 Dentry

Tdentry 64
qid.version 0
qid.path 64
size 0
pdblkno 3
pqpath 63
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct blocks
0 19
1 0
2 0
.
.
.
30 0
31 0
indirect blocks
0 0
1 0
2 0
3 0
4 0
5 0
name dir1

Block 19 contents: file1 Dentry

Tdentry 65	
qid.version 0	
qid.path 65	
size 5	
pdblkno 18	
pqpath 64	
mtime 1653302180823455071	
mode 666	
uid 10006	
gid -1	
muid 10006	
direct blocks	
0 20	content is in Block 20
1 0	
2 0	
.	
.	
.	
30 0	
31 0	
indirect blocks	
0 0	
1 0	
2 0	
3 0	
4 0	
5 0	
name file1	

Contents of block 20 are:

disk/block tests/test.1/disk 20
Tdata 65
test

Representation of two files in a directory (/dir2/file1 and /dir2/file2)

Block 21 contents: /dir2 directory entry

Tdentry 66
qid.version 0
qid.path 66
size 0
pdblkno 3
pqpath 63
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct blocks
0 22
1 24
.
.
.
31 0
indirect blocks
0 0
.
5 0
name dir2

Block 22 contents: file1 directory entry

Tdentry 67
qid.version 0
qid.path 67
size 5
pdblkno 21
pqpath 66
mtime 1653302180823455071
mode 666
uid 10006
gid -1
muid 10006
direct blocks
0 23
1 0
.
.
.
31 0
indirect blocks
0 0
.
5 0
name file1

Block 24 contents: file2 directory entry

Tdentry 68
qid.version 0
qid.path 68
size 5
pdblkno 21
pqpath 66
mtime 1653302180823455071
mode 666
uid 10006
gid -1
muid 10006
direct blocks
0 25
1 0
.
.
.
31 0
indirect blocks
0 0
.
5 0
name file2

iblocks[0] holds the block number of a Tind0 block. A Tind0 block contains a list of Tdata block numbers for files or a list of Tdentry block numbers for directories.

iblocks[1] has the block number of a Tind1 block. A Tind1 block contains a list of Tind0 block numbers.

Similarly, for other iblocks[n] entries, iblocks[n] has the block number of a Tind*n* block. A Tind*n* block contains a list of Tind*(n−1)* block numbers.

Relative index

The zero'th relative index in a directory entry is the first data block. The next relative index is the second data block of the directory entry, and so on.

tests/6.reli shows how a relative index (reli) is translated into an actual disk block number.

To find the actual block number where the first block (zero'th as zero indexed) of a file is stored:

```
tests/6.reli 0 # command, below is the output of this command
reli 0
dblock[0]
```

To find the actual block number where the second block of a file is stored:

```
tests/6.reli 1
reli 1
dblock[1]
```

And so on, for the 32nd and 33rd blocks of a file:

```
tests/6.reli 31
reli 31
dblock[31]

tests/6.reli 32
reli 32
iblock[0]
Tind0 reli 0 is at [0]
```

This is how the last block of a 26 TiB file would be stored:

```
tests/6.reli 57731387017
reli 57731387017
iblock[5]
Tind5 reli 56800235583 is at [61]
Tind4 reli 916132831 is at [61]
Tind3 reli 14776335 is at [61]
Tind2 reli 238327 is at [61]
Tind1 reli 3843 is at [61]
Tind0 reli 61 is at [61]
```

Representation of a 2 MB file (/dir3/2MB.file)

Block 27 contents

Tdentry 1 70
qid.version 0
qid.path 70
size 2056192
pdblkno 26
pqpath 69
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct blocks
0 28
1 29
2 30
.
.
indirect blocks
0 61
1 124
2 4031
3 0
name 2MB.file

Block 28 contents

Tdata 70
0 0123456789

contents of 2MB.file

Representation of a 25MB file (/dir4/25MB.file)

Block 4046 contents

Tdentry 72
qid.version 0
qid.path 72
size 26214400
pdblkno 4045
pqpath 71
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct blocks
0 4195
1 4196
2 4197
.
.
31 4226
indirect blocks
0 4228
1 4291
2 8198
3 0
name 25MB.file

Block 4195 contents

Tdata 72
0 0123456789
.
.
.

starting contents of 25MB.file

Block 4228 contents

Tind0 72
0 4227
1 4229
2 4230
.
.
61 4289

Block 4227 contents

Tdata 72
789
.
.

more content of 25MB.file

System Files			
Block	Description	Directory entry	Content
0	magic		
1	config		Y
2	super		Y
3	/	Y	
4	/adm/	Y	
5	/adm/config	Y	
6	/adm/super	Y	
7	/adm/users	Y	
8	/adm/users		Y
9	/adm/bkp/	Y	
10	/adm/bkp/config.0	Y	
11	/adm/bkp/super.0	Y	
12	/adm/bkp/root.0	Y	
13	/adm/bkp/config.1	Y	
14	/adm/bkp/super.1	Y	
15	/adm/bkp/root.1	Y	
16	/adm/ctl (virtual file, empty contents)	Y	
17	/adm/frees	Y	

The /adm/ctl file is used to halt or sync the file system. /adm/users is a r/w file that will reload users when written to it. The owner of the /adm/ctl file or any user belonging to the sys group can ream the disk.

There is no /adm/magic directory entry as the block number of the magic block is zero and zero block in a directory entry signifies the end of the directory contents.

Backup blocks

Three copies of Config, Super and Root blocks are maintained. This ensures two backups of config, Super and root blocks.

The backup block numbers on the disk are calculated during ream based on the disk size.

		Backup Blocks	
Block	Description	1	2
1	config	last block number -1	middle block number -1
2	super block (obsolete?)	last block number -2	middle block number -2
3	/	last block number -3	middle block number -3

M[a]fs needs atleast Nminblocks=17 blocks (8.5 KB). The middle block number is Nminblocks + ((nblocks - Nminblocks)/2), where nblocks = total number of blocks.

kfs and cwfs use 8192 byte blocks. Hence, they store multiple directory entries (Dentry) per block. They use slot numbers to identify a particular directory entry in a block of directory entries. M[a]fs avoids that be using 512 byte blocks thus having only one directory entry per block. This avoids locking up other sibling directory entries on access.

Buffer cache - Hash buckets with a circular linked list of Iobuf's for collisions.

An Iobuf is used to represent a block in memory. An Iobuf is unique to a block. All disk interaction, except for free block management, happens through an Iobuf. We read a block from the disk into an Iobuf. To update a block on the disk, we write to an Iobuf, which, in-turn gets written to the disk.

An Iobuf is protected by a read-write lock (RWlock). This ensures synchronization across multiple processes updating the same file.

getbuf(), putbuf(), putbufs() and putbuffree() are used to manage Iobuf's. The contents of an Iobuf is not touched unless it is locked by getbuf(). It is unlocked by putbuf(), putbufs() or putbuffree() calls. The Iobuf.dirties Ref is decremented by the mafs writer's dowrite() without a lock(). This is to avoid deadlocks between putbuf() and the writer especially when the writer queue is full.

allocblock() allocates a free block into an Iobuf. allocblocks() allocates a bunch of free blocks with their own Iobuf's.

freeblock() erases the Iobuf and returns the block to the free block management routines.

Iobuf's are organized into a list of hash buckets to speed up access.

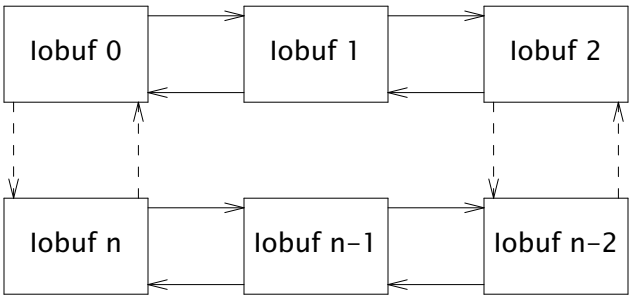
```
Hiob *hiob = nil;     /* array of nbuckets */
struct Hiob           /* Hash bucket */
{
      Iobuf* link;    /* least recently used Iobuf in the circular linked list */
      QLock;          /* controls access to this hash bucket */
};
struct Content  /* used to unmarshall the disk contents */
{
      union{
            u8 buf[Blocksize];
            u64 bufa[Nindperblock];
            Dentry d;
      };
      Tag;
};
struct Iobuf
{
      Ref;
      RWLock;              /* controls access to this Iobuf */
      u64  blkno;          /* block number on the disk, primary key */
      Iobuf      *fore;    /* for lru */
      Iobuf      *back;    /* for lru */
      union{
            u8    *xiobuf;  /* "real" buffer pointer */
            Content *io;    /* cast'able to contents */
      };
      /*
            This field is used by mafs to ensure that Iobufs are not reused
            while there are pending writes.
```

```
            dowrite() uses a Ref instead of a wlock() to mark Iobuf's
            with pending writes.
            Using a wlock() in dowrite() causes a deadlock with putwrite()
            especially when the writer queue is full.
            getbuf() guarantees that even a free'ed block cannot be
            stolen until the dirties == 0. This avoids dirty blocks
            being stolen for other block numbers.
            incref(dirties) only happens while holding a wlock() in putwrite().
        */
        Ref   dirties;     /* number of versions of this block yet to be written by the writer */
};
```

The Iobuf's are arranged into a list of hash buckets. Each bucket points a circular linked list of Iobuf's to handle collisions. If all the Iobuf's in the circular linked list are locked, new Iobuf's are added to this linked list. This circular list is ordered on a least recently used basis. Iobuf's once added to this list are not removed. When an Iobuf is not in the list, the oldest unlocked Iobuf is reused.

Hiob hiob[nbuckets] is a valid representation of the list of hash buckets. The block number is hashed to arrive at the relevant hash bucket index.

hiob[hash(block number)].link = Address of Iobuf0, where Iobuf0 is the least recently used Iobuf.



The size of the buffer cache is: number of hash buckets * collisions per hash bucket * block size. The approximate size of the buffer cache = Nbuckets * Ncollisions * Rawblocksize = 256 * 10 * 512 bytes = 1.28GiB. The -h parameter can be used to change the number of hash buckets.

If you have RAM to spare, increase Nbuckets instead of Ncollisions as the hash index lookup is faster than searching through a linked list.

Iobuf.Ref is used to avoid locking up the hash bucket when a process is waiting for a lock on an Iobuf in that hash bucket.

Iobuf.Ref ensures that an Iobuf is not stolen before another process can get to wlock()'ing it after letting go of the lock on the hash bucket. We cannot hold the lock on the hash bucket until we wlock() the iobuf as that blocks other processes from using the hash bucket. This could also result in a deadlock. For example, the directory entry is block 18, which hashes to a hash index of 7. A writer() locked the directory entry iobuf and wants to add a data block 84 to the directory entry. Block 84 hashes to the same hash index of 7. Another process wanting to access the directory entry is waiting for a lock on that io buffer. While doing so, it has locked the hash bucket. Now, this has

caused a deadlock between both these processes. The first process cannot proceed until it can lock the hash bucket holding block 84 and is still holding the lock on the directory entry in block 18. The second process cannot lock block 18 and is holding the lock on the hash bucket.

```
for locking a buffer:
        qlock(hash bucket); incref(buffer); qunlock(hash bucket);
                wlock(buffer); decref(buffer);

for stealing an unused buffer:
        qlock(hash bucket);
        find a buffer with ref == 0 and wlock()'able.
        qunlock(hash bucket);

for unlocking a buffer:
        wunlock(buffer);
```

Asynchronous writes of Mafs

The blocks to be written to a disk are stored in a linked list represented by:

```
struct Dirties
{
        QLock lck;              /* controls access to this writer queue */
        Wbuf *head, *tail;      /* linked list of dirty blocks yet to be written to the disk */
        s32 n;                  /* number of dirty blocks in this linked list */
        Rendez isfull;          /* write throttling */
        Rendez isempty;         /* writer does not have to keep polling to find work */
} drts = {0};

struct Wbuf
{
        u64  blkno;      /* block number on the disk, primary key */
        Wbuf *prev, *next;   /* writer queue */
        Iobuf *iobuf;           /* pointer to the used Iobuf in the buffer cache */
        union{
                u8    payload;  /* "real" contents */
                Content io;     /* cast'able to contents */
        };
};
```

A single writer process takes the blocks from the Dirties linked list on a FIFO (first-in-first-out) basis and writes them to the disk. putbuf() and putbufs() add blocks to the end of this linked list, the writer queue.

The dirty blocks not yet written to the disk remain in the buffer cache and cannot be stolen when a need for a new Iobuf arises.

Free'd blocks are not written to the disk to avoid writing blanks to a disk.

The writer throttles input when there are more than Npendingwrites waiting to be written. This can be adjusted with the -w parameter.

The alternative to having a single writer process is to have each worker process write to the disk, as mfs does. Synchronous writes throttle writes to disk write speed. With

asynchronous writes, memory is used to hold the data until written to the disk. This shows increased write throughput until we fill up memory. After filling up memory, writes happen at disk speed. Asynchronous writes have the side effect of a single disk write queue.

The ideal npendingwrites = ((ups time in seconds)/2) * (diskspeed in bytes/second) / Rawblocksize.

Free blocks

Free blocks are managed using Extents. The list of free blocks is stored to the disk when shutting down. If this state is not written, then the file system needs to be checked and the list of free blocks should be updated.

When shutting down, the Extents are written to free blocks. This information can be accessed from /adm/frees. Also, fsok in the super block is set to 1. M[a]fs does not start until fsok is 1. When fsok = 0, run the sanity check that the unused blocks and the free blocks in /adm/frees match up. disk/reconcile identifies any missing blocks or blocks that are marked as both used and free.

This process of fixing issues and setting fsok to 1 is manual. There is no automatic file system checker as in other file systems. This document aims to empower you with the knowledge to fix your file system issues instead of entrusting your precious data to an arbitrary decision maker such as the file system checker.

A tag of Tfree and Qpnone represent a free block. If a directory entry is removed, the parent will have a zero'ed out child directory entry (Qid.path = 0) and a tag of Tdentry and Qpnone.

Extents

Free blocks and memory are managed using Extents, an abstraction used to manage a continuous list of items.

An Extent represents a continuous list of items. An Extents is a list of such Extent's.

```
struct Extent {
	struct Extent *low, *high; /* sorted by start */
	u64 start;			/* where this extent starts from */
	u64 len;			/* how many units in this extent */

	/* circular least recently used linked list limited to Nlru items */
	struct Extent *prev, *next;
};
struct Extents {
	Extent *head;	/* find the first block in a jiffy */
	QLock lck;
	u32 n;			/* number of extents */
	Rendez isempty;	/* fully used, nothing available */

	u8 nlru;		/* number of items in the lru linked list */
	Extent *lru;	/* least recently used extent in the circular lru linked list */
```

};

To allocate n items from Extents, we find the lowest (by block number or memory address) extent that can satisfy our request. If a bigger Extent is available, slice it and take the portion we need.

If there is no available Extent to satisfy our request, panic().

allocblock() and freeblock() use balloc() and bfree() respectively. balloc() assigns blocks from an extent and bfree() adds them to an extent for next allocation.

Extents at memory location 1

lru	100
el	0
n	3

assuming that the Extent at 100 was used last

unlocked

Extent at 100

blkno	10
len	1
low	0
high	200

Extent at 200

blkno	20
len	3
low	100
high	300

Extent at 300

blkno	30
len	2
low	200
high	0

\+ freed block numbers 11,12,13,14 =

Extent at 100

blkno	10
len	5
low	0
high	200

Extent at 200

blkno	20
len	3
low	100
high	300

Extent at 300

blkno	30
len	2
low	200
high	0

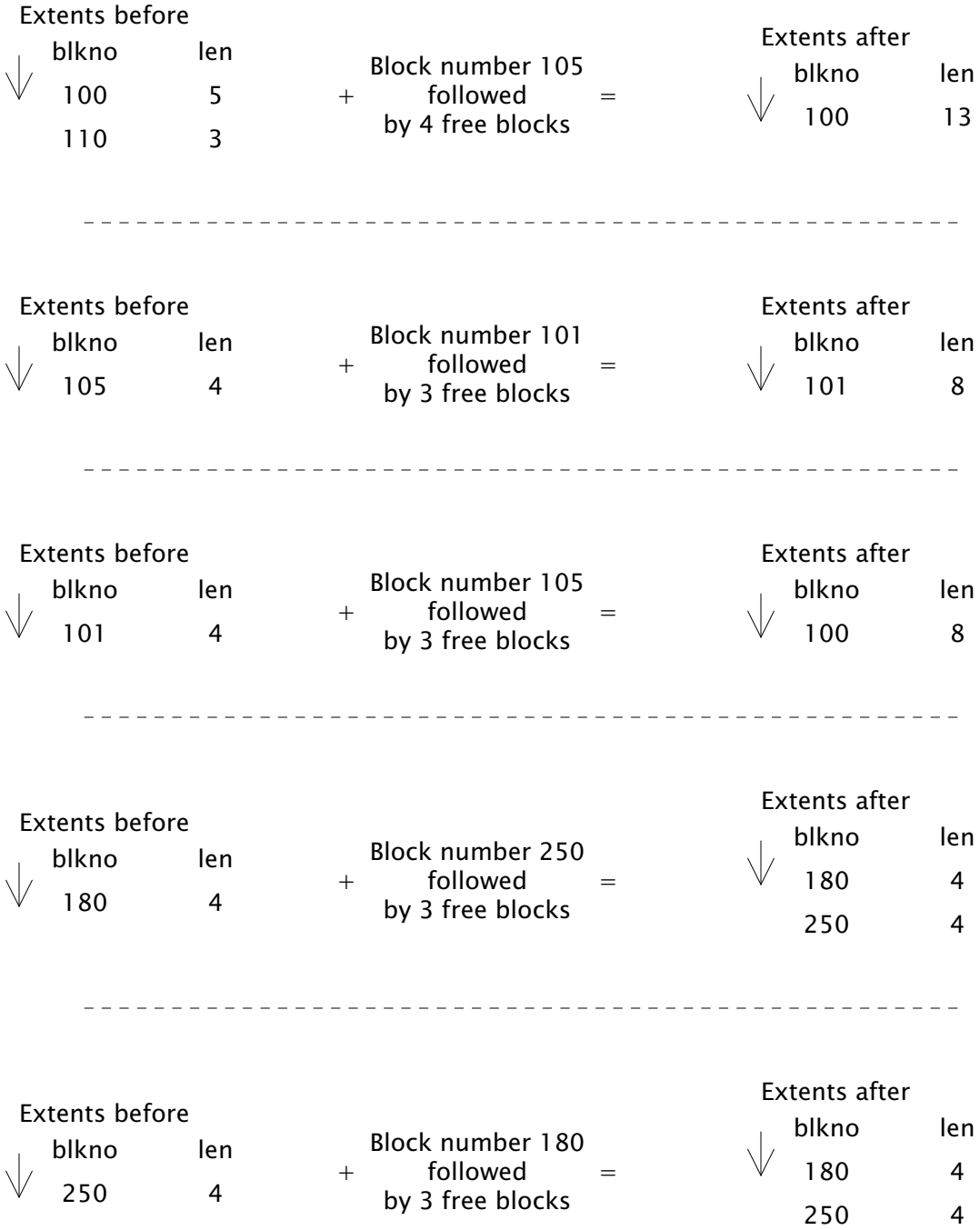
Extents before

blkno	len
20	3

\+ Block number 40 followed by 3 free blocks =

Extents after

blkno	len
20	3
40	4



Kfs stores the list of free blocks in a Tfrees block and the Superblock. Instead we use block management routines, similar to pool.h, to allocate and monitor free blocks. On shutdown(), the block management routines (extents.[ch]) store state into the free blocks. This can be read from /adm/frees. On startup, this is read back by the block management routines. On a crash, the fsck can walk the directory structure to identify the free blocks and recreate /adm/frees.

Code details

Program	Description
disk/mfs	Start mfs on a disk.
disk/mafs	Start mafs on a disk.
disk/free	List the free blocks. It reads the contents of /adm/frees.
disk/used	List the used blocks by traversing all directory entries.
disk/block	Show the contents of a block.
disk/unused	Given a list of used blocks, lists the unused blocks.
disk/updatefrees	Update the contents of /adm/frees.

File	Description	chatty9p
9p.c	9p transactions	2
sub.c	initialization and super block related routines.	2
dentry.c	encode/decode the file system abstraction into block operations.	3
iobuf.c	routines on Iobuf's. The bkp() routines operate on Iobuf's.	5
extents.[ch]	routines to manage the free blocks.	6
ctl.c	/adm/ctl operations.	
tag.c	routines to manage a relative index (reli) in a directory entry.	
blk.c	routines to show blocks.	
writer.c	disk writer routines.	
console.c	obsolete. /adm/ctl is the console.	

A Chan's state could get out of sync with the contents if another process changes the on-disk state. Ephase error occurs when that happens.

For throughput, multiple processes are used to service 9p i/o requests when the -s flag is not used.

Useful commands:

disk/mfs and disk/mafs have the same arguments. The following commands use disk/mafs to avoid duplicating them for disk/mfs.

Ream and start single process M[a]fs on a disk and also mount it for use.

```
mount -c <{disk/mafs -s -r mafs_myservice mydisk <[0=1]} /n/mafs_myservice
    -s: use stdin and stdout for communication
    -r mafs_myservice: ream the disk using mafs_myservice as the service name
     mydisk: running mafs on the disk, mydisk
```

Ream and start multiple-process mafs on a disk.

```
disk/mafs -r mafs_myservice mydisk
mount -c /srv/mafs_myservice /n/mafs_myservice
```

Ream and start mafs on a file. Also, mount thet filesystem at /n/mafs_myservice.

```
dd -if /dev/zero -of myfile -bs 512 -count 128 # 64KB file
mount -c <{disk/mafs -s -r mafs_service myfile <[0=1]} /n/mafs_myservice

# to reuse the contents of myfile later, remove -r (ream) from the above command.
mount -c <{disk/mafs -s myfile <[0=1]} /n/mafs_myservice
```

Prepare and use a disk (/dev/sdF1) for mafs.

```
disk/fdisk -bawp /dev/sdF1/data    # partition the disk
echo '
a fs 9  $-7
w
p
q' | disk/prep -b  /dev/sdF1/plan9 # add an fs plan 9 partition to the disk
disk/mafs -r mafs_sdF1 /dev/sdF1/fs     # -r to ream the disk
mount -c /srv/mafs_sdF1 /n/mafs_sdF1

# for using the mafs file system on the disk later on
disk/mafs  /dev/sdF1/fs # no -r
mount -c /srv/mafs_sdF1 /n/mafs_sdF1
```

Starting mafs on a 2MB byte file. The below commands create a disk.file to use as a disk. Mount /n/mafs_disk.file for the file system.

```
dd -if /dev/zero -of disk.file -bs 512 -count 4096;
mount -c <{disk/mafs -s -r mafs_disk.file \
        <[0=1]} /n/mafs_disk.file
```

Starting mafs on a RAM file. The below commands create a ramfs filesystem to use as a disk.

```
ramfs -m /n/mafs_ramfs
touch /n/mafs_ramfs/file
dd -if /dev/zero -of /n/mafs_ramfs/file -count 700 -bs 1m
disk/mafs -r mafs_ramfs_file /n/mafs_ramfs/file
mount -c /srv/mafs_ramfs_file /n/mafs_ramfs_file
```

Sync M[a]fs. This command does not return until all the writes are written to the disk. So, could take a long time if you have a long writer queue.

```
echo sync >> /n/mafs_myservice/adm/ctl
```

Stop M[a]fs. This command does not return until all the writes are written to the disk. So, could take a long time if you have a long writer queue.

```
echo halt >> /n/mafs_myservice/adm/ctl
```

Interpret the contents of a block based on the tag and write out a single formatted block based on the tag

```
disk/block  tests/test.0/disk 22
```

Traverse the directory heirarchy and write out all the used block numbers. disk/reconcile uses the output of this to reconcile the list of used blocks with the list of free blocks. Also, writes the invalid blocks to stderr. Starting from root, walk down each directory entry printing out the linked blocks with invalid tags. (Why not just write out the list of dirty blocks too? instead of using a different command for it?)

```
disk/used  tests/test.0/disk
```

From the contents of /adm/frees show the list of free blocks. disk/reconcile uses the output of this to reconcile the list of used blocks with the list of free blocks.

```
disk/free  tests/test.0/disk
```

Read two lists of block numbers and flag the common and missing blocks.

```
disk/reconcile -u <{disk/used tests/test.0/disk} \
      -F <{disk/free  tests/test.0/disk} 32
```

Find traverses the directory heirarchy and identifies the file that a block number belongs to.

```
disk/find tests/test.0/disk 17
```

Find the total number of blocks on a disk.

```
dd -if /dev/sdF1/fs -bs 512 -iseek 1 -count 1 -quiet 1 | awk '$1 == "nblocks" { print $2 }'

disk/block /dev/sdF1/fs 1 | awk '$1 == "nblocks" { print $2 }'
```

Build the list of free blocks. This should match the contents of /adm/frees.

```
disk/unused <{disk/used /dev/sdF1/fs} 11721040049 # 11721040049 = total number of disk blocks
disk/unused <{disk/used test.0/disk} 32 # 32 = total number of disk blocks
```

Change the contents of /adm/frees.

```
disk/updatefrees tests/test.0/disk <{disk/unused <{disk/used tests/test.0/disk} 32}
disk/updatefrees /dev/sdF1/fs <{disk/unused <{disk/used /dev/sdF1/fs} 11721040049}
```

A sanity check that the file system is not corrupt by comparing that the unused blocks and free blocks match up. $nblocks is the total number of disk blocks. $disk is the disk.

```
diff <{disk/unused -l <{disk/used tests/test.0/disk} 32}} <{disk/free tests/test.0/disk}
```

Changing the service name without a ream.

```
disk/block /dev/sdF1/fs 1 | wc
Tdata 2
size 6001172505088
nblocks 11721040049
backup config 1 to 11721040048 5860520032
backup super 2 to 11721040047 5860520031
backup root 3 to 11721040046 5860520030
service mafs_ddf_1
```

```
dd -if /dev/sdF1/fs -count 10 -skip 682 -bs 1
mafs_ddf_110+0 records in
10+0 records out

dd -if <{echo m_ddf_1; cat /dev/zero} -of /dev/sdF1/fs -count 11 -oseek 682 -bs 1
7+0 records in
7+0 records out

disk/block /dev/sdF1/fs 1
Tdata 2
size 6001172505088
nblocks 11721040049
backup config 1 to 11721040048 5860520032
backup super 2 to 11721040047 5860520031
backup root 3 to 11721040046 5860520030
service m_ddf_1
```

Changing the magic phrase in the magic block.

```
disk/block /dev/sdF1/fs 0
Tmagic 1
mafs device
512

dd -if /dev/sdF1/fs -count 16 -iseek 256 -bs 1
mafs device
512
20+0 records in
20+0 records out

dd -if <{echo m[a]fs device; echo 512; cat /dev/zero} -of /dev/sdF1/fs -count 18 -oseek 256 -bs 1
18+0 records in
18+0 records out

dd -if /dev/sdF1/fs -count 18 -iseek 256 -bs 1
m[a]fs device
512
18+0 records in
18+0 records out

disk/block /dev/sdF1/fs 0
Tmagic 1
m[a]fs device
512
```

Tests

Program	Description
tests/regress.rc	All regression tests
tests/chkextents.rc	Unit tests on extents
tests/chkreli.rc	Unit tests on relative index lookups
tests/6.offsets	Write file using different offsets to test mafswrite()
tests/6.sizes	Show the effects of the different parameters
tests/6.testextents	Test extents.[ch] state changes
tests/6.reli	Translate relative index to block number on a disk

The below disk state tests:
1. Initialize a disk for mafs.
2. Run mfs or mafs on that dsk.
3. Stop mfs or mafs.
4. Compare the contents with the expected contents (tests/test.0/blocks/*).

Disk State	
Test	Description
tests/test.0	empty disk
tests/test.1	create a file /dir1/file1 and echo test into it
tests/test.2	writes at different offsets to a file and then removes the file
tests/test.3	write, read and delete files with sizes upto 16384 blocks
tests/test.4	directory copy
tests/test.5	fcp gzipped files
tests/test.6	df
tests/test.7	multiple processes working on the filesystem simultaneously
tests/test.8	check backup blocks locations
tests/test.9	examples used by this document
tests/test.a	write, read and delete a 100MB file
tests/test.b	duplicate of test.2 but seeded with random data
tests/test.d	seed with random data and do mkdir -p a/b/c/d/e/f/g/h
tests/test.e	seed with random data and test directory and file deletions

Extents behaviour	
Test	Description
tests/extents/addabove	Figure 1 of the Extents section
tests/extents/addabove1	Figure 2 of the Extents section
tests/extents/addbelow	Figure 3 of the Extents section
tests/extents/mergeabove	Figure 4 of the Extents section
tests/extents/mergenext	Figure 5 of the Extents section
tests/extents/mergeprevious	Figure 6 of the Extents section

To run all the regression tests:

```
cd tests/; ./regress.rc
```

To loop through all the blocks of a test:

```
for(t in tests/test.2/blocks/^‘{seq 0 39}*){ echo $t; echo ’---------’; cat $t; echo }
```

Performance metrics

```
ramfs -m /n/ramfs
touch /n/ramfs/file
cat /dev/zero | tput -p > /n/ramfs/file
172.49 MB/s
174.56 MB/s
163.50 MB/s
125.00 MB/s
102.99 MB/s
87.81 MB/s
77.78 MB/s
69.50 MB/s
63.71 MB/s
58.65 MB/s
54.72 MB/s
dd -if /dev/zero -of /n/ramfs/file -count 700 -bs 1m

disk/mfs -r mfs_ramfs_file /n/ramfs/file
mount -c /srv/mfs_ramfs_file /n/mfs_ramfs_file
cat /dev/zero | tput -p > /n/mfs_ramfs_file/zeros.file
6.26 MB/s
5.99 MB/s
5.90 MB/s
echo halt >> /n/mfs_ramfs_file/adm/ctl; lc /srv
unmount /n/mfs_ramfs

disk/mafs -r mafs_ramafs_file /n/ramfs/file
mount -c /srv/mafs_ramafs_file /n/mafs_ramafs_file
cat /dev/zero | tput -p > /n/mafs_ramafs_file/zeros.file # throttles down to mfs speed
45.49 MB/s
31.52 MB/s
23.16 MB/s
24.54 MB/s
echo halt >> /n/mafs_ramafs_file/adm/ctl; lc /srv
unmount /n/ramfs
```

Limitations

As we use packed structs to store data to the disk, a disk with m[a]fs is not portable to a machine using a different endian system.

Design considerations

Why are you not using a checksum to verify the contents?
Checksums are probabilistic and can be implemented as a bespoke application instead of complicating the file system implementation.

Source

http://git.9front.org/plan9front/mafs/HEAD/info.html

References

[1] Sean Quinlan, "A Cached WORM File System," Software--Practice and Experience, Vol 21., No 12., December 1991, pp. 1289-1299
[2] Ken Thompson, Geoff Collyer, "The 64-bit Standalone Plan 9 File Server"