

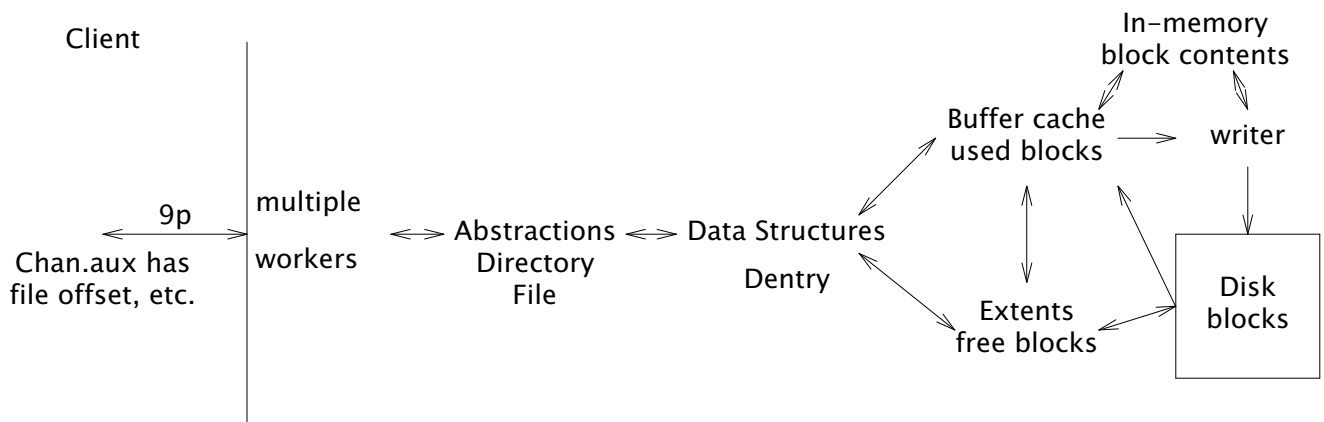
Mafs – Plan 9 userspace file system

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 desire for speed or a feature. This empowerment is priceless for those with skin in the game.

Mafs is a user space file system to provide system stability and security. It is 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.

Workflow

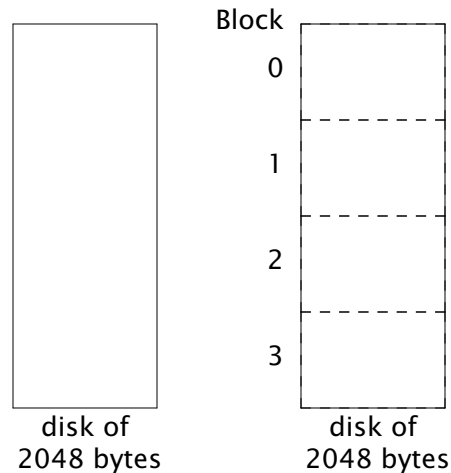


Disk Contents

Mafs 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, Tindir */
    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 its 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 a list of blocks, the blocks are structured to use multiple levels of indirection as 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 of 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.

Mafs 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= 512, /* 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 pdblkn0;   /* parent dentry absolute block number. 0 for root. */
    u64 pqpah;     /* parent qid.path */
    u64 mtime;     /* modified time 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
 * Ndentriperblock: 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. 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 144 Ndblock 32 Niblock 6

Blocksize 502 Nindperblock 62

A Tind0 unit points to 1 data blocks (502 bytes)

block points to 62 data blocks

reli start 32 max 93

max size 94*Blocksize = 47188 bytes

A Tind1 unit points to 62 data blocks (31124 bytes)

block points to 3844 data blocks

reli start 94 max 3937

max size 3938*Blocksize = 1976876 bytes = 1 MiB

A Tind2 unit points to 3844 data blocks (1929688 bytes)

block points to 238328 data blocks

reli start 3938 max 242265

max size 242266*Blocksize = 121617532 bytes = 115 MiB

A Tind3 unit points to 238328 data blocks (119640656 bytes)

block points to 14776336 data blocks

reli start 242266 max 15018601

max size 15018602*Blocksize = 7539338204 bytes = 7 GiB

A Tind4 unit points to 14776336 data blocks (7417720672 bytes)

block points to 916132832 data blocks

reli start 15018602 max 931151433

max size 931151434*Blocksize = 467438019868 bytes = 435 GiB

A Tind5 unit points to 916132832 data blocks (459898681664 bytes)

block points to 56800235584 data blocks

reli start 931151434 max 57731387017

max size 57731387018*Blocksize = 28981156283036 bytes = 26 TiB

On an empty mafs 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
pdblknno 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
pdblknno 18
pqpath 64
mtime 1653302180823455071
mode 666
uid 10006
gid -1
muid 10006
direct blocks
  0 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
```

content is in Block 20

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
pdblkn0 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
pdblkn0 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
pdblkn0 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] has 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
pdblkn0 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
pdblkn0 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 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 back-ups of config, Super and root blocks.

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

Block	Description	Backup Blocks	
		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

Mafs needs atleast $N_{minblocks}=17$ blocks (8.5 KB). The middle block number is $N_{minblocks} + ((nblocks - N_{minblocks})/2)$, 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. Mafs avoids that by 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 lobuf's for collisions.

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

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

getbuf(), putbuf() and putbuffree() are used to manage lobuf's. The contents of an lobuf is not touched unless it is locked between getbuf(), putbuf() and putbuffree() calls. The lobuf.dirties Ref is decremented by the writer's dwrite() 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 lobuf.

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

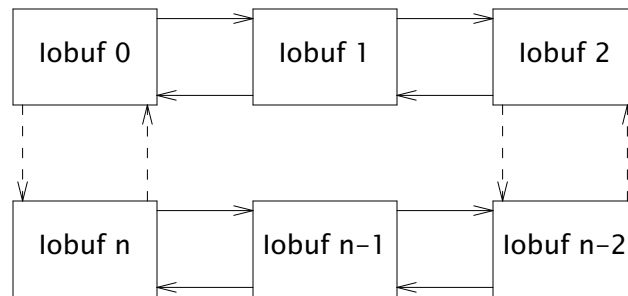
lobuf's are organized into a list of hash buckets to speed up access.

```
struct Hiob          /* Hash bucket */
{
    lobuf* link;      /* least recently used lobuf in the circular linked list */
    QLock;            /* controls access to this hash bucket */
};
struct lobuf
{
    Ref;
    RWLock;           /* controls access to this lobuf */
    u64 blkno;         /* block number on the disk, primary key */
    lobuf *fore;       /* for lru */
    lobuf *back;       /* for lru */
    union{
        u8 *xiobuf;    /* "real" buffer pointer */
        Content *io;    /* cast'able to contents */
    };
    Ref dirties;       /* number of versions of this block yet to be written by the writer */
};
```

The lobuf's are arranged into a list of hash buckets. Each bucket points a circular linked list of lobuf's to handle collisions. If all the lobuf's in the circular linked list are locked, new lobuf's are added to this linked list. This circular list is ordered on a least recently used basis. lobuf's once added to this list are not removed. When an lobuf is not in the list, the oldest unlocked lobuf 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 lobuf0, where lobuf0 is the least recently used lobuf.



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 * Raw-blocksize = 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.

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

lobuf.Ref ensures that an lobuf 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 lobuf 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 lobuf 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

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

```
struct Dirties
{
    QLock lck;           /* controls access to this queue */
    Wbuf *head, *tail;   /* linked list of dirty blocks yet to be written to the disk */
    s32 n;
    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;
    lobuf *lobuf;        /* pointer to the used lobuf in the buffer cache */
    union{
        u8  payload;    /* "real" contents */
        Content io;      /* cast'able to contents */
    };
};
```

A 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() adds blocks to the end of this linked list.

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

Free blocks – Extents

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 is written to /adm/frees. Also, fsok in the super block is set to 1. When fsok = 0, run an fsck (filesystem checker) to correct any inconsistencies on the disk.

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.

Algorithm to allocate blocks from Extents:

1. Of all the Extents with the length we need, pick the Extent with the lowest block number (blkno).
2. If no Extent of the length we need is available, then break up the smallest extent.

```

struct Extent {
    struct Extent *low, *high; /* sorted by start */
    u64 start;                 /* where this extent starts from */
    u64 len;                   /* how many units in this extent */
};
struct Extents {
    Extent *lru;               /* least recently used extent */
    Extent *head;              /* find the first block in a jiffy */
    QLock lck;
    u32 n;                     /* number of extents */
    Rendez isempty;            /* fully used, nothing available */
};

```

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
<hr/>	
low	0
high	200

Extent at 200

blkno	20
len	3
<hr/>	
low	100
high	300

Extent at 300

blkno	30
len	2
<hr/>	
low	200
high	0

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

Extent at 100

blkno	10
len	5
<hr/>	
low	0
high	200

Extent at 200

blkno	20
len	3
<hr/>	
low	100
high	300

Extent at 300

blkno	30
len	2
<hr/>	
low	200
high	0

Extents before				Extents after		
↓	blkno	len		↓	blkno	len
	20	3	+		20	3
					40	4

Block number 40 followed by 3 free blocks =

Extents before				Extents after		
↓	blkno	len		↓	blkno	len
	100	5	+		100	13
	110	3				

Block number 105 followed by 4 free blocks =

Extents before				Extents after		
↓	blkno	len		↓	blkno	len
	105	4	+		101	8

Block number 101 followed by 3 free blocks =

Extents before				Extents after		
↓	blkno	len		↓	blkno	len
	101	4	+		100	8

Block number 105 followed by 3 free blocks =

Extents before				Extents after		
↓	blkno	len		↓	blkno	len
	180	4	+		180	4
					250	4

Block number 250 followed by 3 free blocks =

Extents before				Extents after		
↓	blkno	len		↓	blkno	len
	250	4	+		180	4
					250	4

Block number 180 followed by 3 free blocks =

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/mafs	Start mafs on a disk
disk/free	List the free blocks
disk/used	List the used blocks
disk/block	Show the contents of a block

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 lobuf's. The bkp() routines operate on lobuf'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.

Useful commands:

Ream and start single process Mafs on a disk and also mount it for use.

```
mount -c <{disk/mafs -s -r mafs_myservice -h 10 mydisk <[0=1]} /n/mafs_myservice
-s: use stdin and stdout for communication
-r myservice: ream the disk using mafs_myservice as the service name
-h 10: use 10 hash buckets
mydisk: running Mafs on the mydisk
```

Ream and start multiple-process mafs on a disk.

```
disk/mafs -r mafs_myservice -h 10 mydisk
mount -c /srv/mafs_myservice /n/mafs_myservice
```


Ream and start mafs on a file. Also, mount that 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 -h 10 myfile <[0=1]} /n/mafs_myservice

# for reusing the contents of myfile later, remove -r (ream).
mount -c <{disk/mafs -s -h 10 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 sdF1 # 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 -m 1 -n 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 Mafs. 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 Mafs. 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 heirarchy and identifies the file that a block number belongs to.

```
disk/find disk.file blocknumber
```

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. Initializes a disk for mafs.
2. Run mafs on that dsk.
3. Stop mafs.
4. Compares 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	fcg 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

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 loop through all the blocks of a test:

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

Limitations

As we use packed structs to store data to the disk, a disk with mafs 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 filesystem.

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"