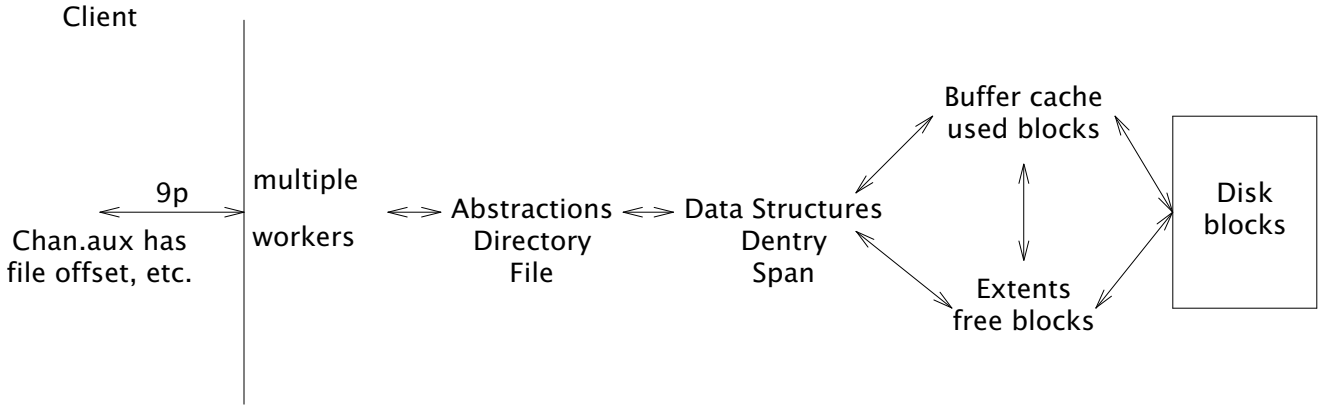
Mafs - Plan 9 userspace file system

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 as software literacy rises.

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

Workflow



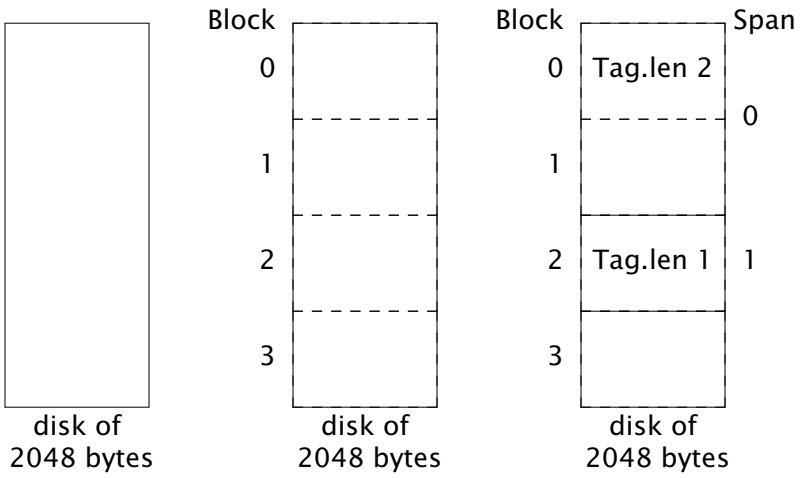
Disk Contents

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. For optimum throughput, file data blocks are logically grouped into Spans.

A Tag identifies a Span written to the disk. A continuous space of Tag.len blocks makes up each Span. To efficiently manage system memory across numerous users and files, the size of each Span is constrained.

A sample disk of 2048 bytes with 4 blocks and 2 Spans



The different types of possible Span's on a disk are:

```
enum
{
	Tfree  = 0,	/* free blocks */
	Tmagic,		/* the first (zero'th) block holds a magic word */
	Tdentry,	/* directory entry */
			/* Tind n are last, to allow for future increases */
	Tdata,		/* file contents */
	Tind0,		/* list of Tdata Spans for files or Tdentry Spans for directories.*/
	Tind1,		/* list of Tind0 blocks */
	Tind2,		/* list of Tind1 blocks */
	Tind3,		/* list of Tind2 blocks. we can have a 11 TB file */
};
```

A Span is stored to the disk with a Tag.

```
struct Tag
{
	u64 path; /* Qid.path, unique identifier of directory or file */
	u8 type;   /* Tfree, Tmagic, Tdentry, Tdata, Tind n */
	u8 dirty;   /* is 1, when being written to.
			Identifies dirty data on a crash.
			This byte position is denoted by the enum Nthdirty. */
	u16 len;   /* number of blocks in this Span */
};
```

Every file or directory is represented on the disk by a directory entry (Dentry). Every directory entry uses a 1-block Span (Tag.type = Tdentry) and is uniquely identifiable by a Qid.

Mafs does not store the last access time of a file or directory.

A Dentry is defined as:

```
enum {
	Rawblocksize= 512, /* real block size */
	Ndspanid = 24,	/* number of direct Span identifiers in a Dentry */
	Niblock    = 4,	/* max depth of indirect blocks */
};
```

```
struct Qid9p1
{
	u32 version;
	u64 path;	/* unique identifier */
};
struct Spanid		/* Span identifier */
{
	u64 blkno;	/* starting block number */
	u16 len;	/* number of blocks */
};
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 nano seconds from epoch */
	u32 mode;	/* same bits as defined in lib.h Dir.mode */
	s16 uid;
	s16 gid;
	s16 muid;
	Spanid dspans[Ndspanid];/* direct Span identifiers */
				/* Tag.type = Tdentry for directories and Tdata for files */
	u64 iblocks[Niblock];	/* indirect blocks */
};

/*
 * derived constants
 * Ndentriesperblock: number of Dentry's per block
 * Nindperblock: number of block pointers per block
 * Nspanidperblock: number of Span identifiers per a Tind0 block
 */
enum {
	Blocksize  = Rawblocksize - sizeof(Tag),
	Namelen  = (Blocksize-sizeof(Dentry1)),		/* max size of file name components */
	Maxspanlen	= MB/Rawblocksize,		/* in blocks */
	Maxspansize	= (Maxspanlen*Rawblocksize)-sizeof(Tag), /* in bytes */

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

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

Block 18 contents: /dir1 Dentry

Tdentry 1 64
qid.version 0
qid.path 64
size 0
pdblkno 3
pqpath 63
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct spans
0 19 1
1 0 0
2 0 0
3 0 0
4 0 0
5 0 0
6 0 0
7 0 0
8 0 0
9 0 0
10 0 0
11 0 0
12 0 0
13 0 0
14 0 0
15 0 0
16 0 0
17 0 0
18 0 0
19 0 0
20 0 0
21 0 0
22 0 0
23 0 0
indirect blocks
0 0
1 0
2 0
3 0
name dir1

Block 19 contents: file1 Dentry

Tdentry 1 65
qid.version 0
qid.path 65
size 5
pdblkno 18
pqpath 64
mtime 1653302180823455071
mode 666
uid 10006
gid -1
muid 10006
direct spans
0 20 1
1 0 0
2 0 0
3 0 0
4 0 0
5 0 0
6 0 0
7 0 0
8 0 0
9 0 0
10 0 0
11 0 0
12 0 0
13 0 0
14 0 0
15 0 0
16 0 0
17 0 0
18 0 0
19 0 0
20 0 0
21 0 0
22 0 0
23 0 0
indirect blocks
0 0
1 0
2 0
3 0
name file1

content is in Block 20

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

Block 21 contents: /dir2 Dentry

```
Tdentry 1 66
qid.version 0
qid.path 66
size 0
pdblkno 3
pqpath 63
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct spans
0 22 1
1 24 1
.
.
.
23 0 0
indirect blocks
0 0
.
3 0
name dir2
```

Block 22 contents: file1 Dentry

```
Tdentry 1 67
qid.version 0
qid.path 67
size 5
pdblkno 21
pqpath 66
mtime 1653302180823455071
mode 666
uid 10006
gid -1
muid 10006
direct spans
0 23 1
1 0 0
.
.
.
23 0 0
indirect blocks
0 0
.
3 0
name file1
```

Block 24 contents: file2 Dentry

```
Tdentry 1 68
qid.version 0
qid.path 68
size 5
pdblkno 21
pqpath 66
mtime 1653302180823455071
mode 666
uid 10006
gid -1
muid 10006
direct spans
0 25 1
1 0 0
.
.
.
23 0 0
indirect blocks
0 0
.
3 0
name file2
```

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 spans
    0 28 2048
    1 2076 1969
    2 0 0
    .
    .
name 2MB.file
```

Block 28 contents

```
Tdata 2048 70
0 0123456789
```

contents of 2MB.file

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

Block 4046 contents

```
Tdentry 1 72
qid.version 0
qid.path 72
size 26214400
pdblkno 4045
pqpath 71
mtime 1653302180819962729
mode 20000000777
uid 10006
gid -1
muid 10006
direct spans
  0 8123 2048
  1 10171 2048
  2 12219 2048
  .
  .
  23 57249 2048
indirect blocks
  0 22434
  1 0
  2 0
  3 0
name 25MB.file
```

Block 8123 contents

```
Tdata 2048 72
0 0123456789
.
.
.
```

starting contents of 25MB.file

Block 22434 contents

```
Tind0 1 72
0 59297 2048
  1 18363 1
  2 0 0
  .
  .
  49 0 0
```

Block 59297 contents

```
Tdata 2048 72
6789
.
.
```

more content of 25MB.file

kfs and cwfs do not use Spans. They use blocks but with a size of 8192 bytes to avoid dealing with small data blocks (less system calls, context switches, etc). 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. Instead, we use variable length

Span's for Tdata. All other types of data use 1 block Span's. This makes us use bigger sized content for data blocks (lesser system calls, context switches, etc.).

iblocks[0] contains the block number of a Tind0 Span(1 block). A Tind0 Span is a list of Span identifiers with the block numbers of Tdata Span's for files and Tdentry Span's for directories.

iblocks[1] contains the block number of a Tind1 Span(1 block). A Tind1 Span is a list of block numbers of Tind0 blocks.

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

The Tag.dirty flag is set while a Span is being written. This helps identify dirty Span's after a crash.

To increase read and write throughput, all Tdata allocations will be Span's (Block number + len). The maximum Span length is 1MB (Maxspanlen blocks). Only the last span can be less than 1MB size.

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's dspanids 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.

A file's data blocks are identified by a tag of Tdata and 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.dspanids[0] = (Spanid){0,0}.

The directory entry is locked with a read-write lock (RWlock) for any file operations. This ensures synchronization across multiple processes updating the same file.

Why is the Span's len stored in the directory entry when the same information can be obtained from the block's Tag.len?

1. This avoids an extra read call for the Tag.len before getting the contents from the disk. Instead, we can read the contents with one read call as we know the number of blocks to read. Also, it works as a cross-checking mechanism if the Tag gets overwritten.
2. Code is simpler when we store the length of the Span in the directory entry.

Disk State after ream - 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. /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 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

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

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

An Iobuf is used to represent a Span in memory. An Iobuf is unique to a Span. All disk interaction, except for free block management, happens through an Iobuf. We read

Span's from the disk into an Iobuf. To update Span's on the disk, we write to an Iobuf, which, in-turn gets written to the disk.

getbuf() and putbuf() are used to manage Iobuf's. The contents of an Iobuf is not touched unless it is locked between getbuf() and putbuf() calls.

allocblocks() allocates free blocks into an Iobuf.

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

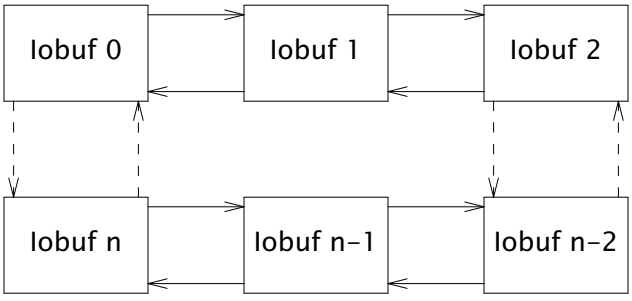
Iobuf's are organized into a list of hash buckets to speed up access.

```
struct Hiob             /* Hash bucket */
{
        Iobuf* link;    /* least recently used Iobuf in the circular linked list */
        QLock;
};
struct Iobuf
{
        Ref;
        RWLock;
        u64  blkno;     /* block number on the disk, primary key */
        u16  len;       /* number of blocks of data xiobuf points to */
        Iobuf *fore;    /* for lru */
        Iobuf *back;    /* for lru */
        union{
                u8    *xiobuf;    /* "real" buffer pointer */
                Content *io;      /* cast'able to contents */
        }
        int    flags;
};
```

The Iobuf's are arranged into a hash structure of Nbuckets. Each bucket has a circular linked list of Ncollisions' 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 of the Span 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 approximately: number of hash buckets * collisions per hash bucket * Maximum Span size. By default, the approximate size of the buffer cache = Nbuckets * Ncollisions * Maxspansize = 256 * 10 * 1MB = 2.56GB. The -h parameter can be used to change the number of hash buckets.

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 the issue.

A tag of Tfree and Qpnone represents 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 blkno */
	struct Extent *small, *big;/* sorted by len */
	u64 blkno;			/* block number */
	u64 len;			/* in blocks */
};
struct Extents {
	Extent *lru;	/* least recently used extent */
	QLock el;
	u32 n;		/* number of extents */
};
```

allocblocks() and freeblocks() use balloc() and bfree(). balloc() assigns blocks and bfree() holds them for next allocation.

Extents at memory location 1

lru	100	assuming that the Extent at 100 was used last
el	0	unlocked
n	3	

Extent at 100

blkno	10
len	1
low	0
high	200
small	0
big	300

Extent at 200

blkno	20
len	3
low	100
high	300
small	300
big	0

Extent at 300

blkno	30
len	2
low	200
high	0
small	100
big	200

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

Extent at 100

blkno	10
len	5
low	0
high	200
small	200
big	0

Extent at 200

blkno	20
len	3
low	100
high	300
small	300
big	100

Extent at 300

blkno	30
len	2
low	200
high	0
small	200
big	100

Extents before

blkno	len
20	3

\+ Block number 40 followed by 3 free blocks =

Extents after

blkno	len
20	3
40	4

Extents after

blkno	len
20	3
40	4

Extents before

blkno	len
100	5
110	3

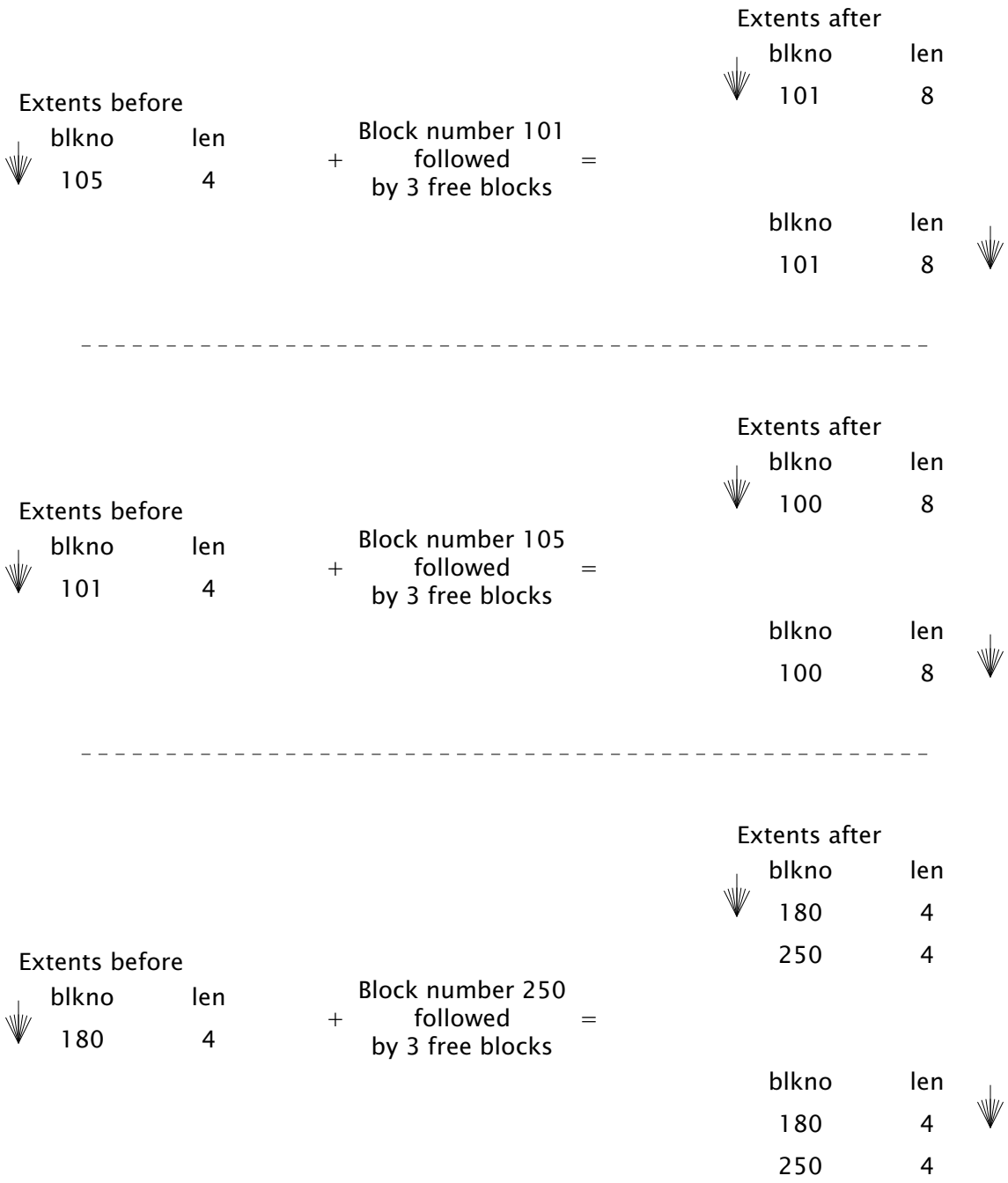
\+ Block number 105 followed by 4 free blocks =

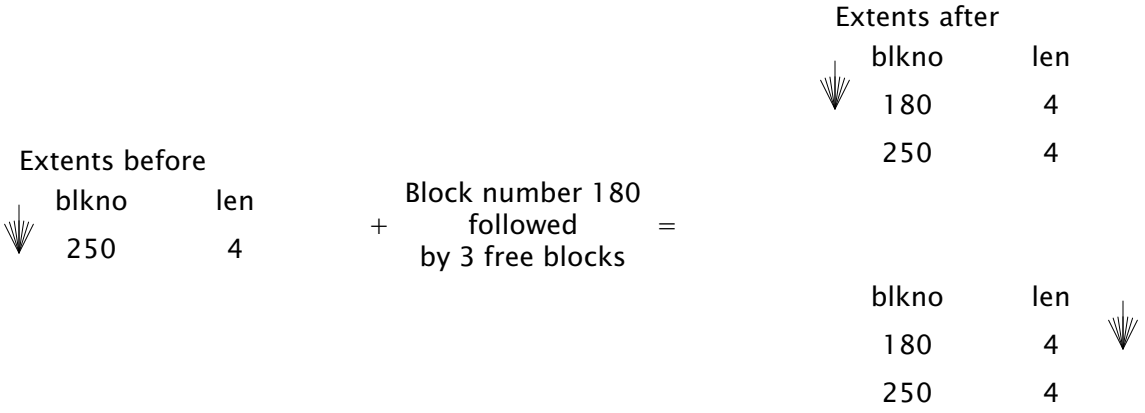
Extents after

blkno	len
100	13

Extents after

blkno	len
100	13


Extents before
blkno
len
105
4
+
Block number 101
followed
by 3 free blocks
=
Extents after
blkno
len
101
8
blkno
len
101
8
Extents before
blkno
len
101
4
+
Block number 105
followed
by 3 free blocks
=
Extents after
blkno
len
100
8
blkno
len
100
8
Extents before
blkno
len
180
4
+
Block number 250
followed
by 3 free blocks
=
Extents after
blkno
len
180
4
250
4
blkno
len
180
4
250
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

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 convert from a relative index in a directory entry to a tag.	
blk.c	routines to show blocks.	
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 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 -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
```

Stop Mafs.

```
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 the block number belongs to.

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

Starting mafs on a 2MB byte file. The above commands will 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
```

tests/sizes.c shows the maximum file size representable by a Dentry.

```
: tests ; ./6.sizes
Namelen 174 Ndspanid 24 Niblock 4
Blocksize 500 Nspanidperblock 50 Nindperblock 62
Maxspanlen 2048 Maxspansize 1048564
A Tind0 unit points to 1 data spans (1048564 bytes)
        block points to 50 data spans
A Tind1 unit points to 50 data spans (52428200 bytes)
        block points to 3100 data spans
A Tind2 unit points to 3100 data spans (3250548400 bytes)
        block points to 192200 data spans
A Tind3 unit points to 192200 data spans (201534000800 bytes)
        block points to 11916400 data spans
sizeof(Dentry1) 326 Namelen 174
maxsize direct spans max 24
maxsize Tind0 50 max 74
maxsize Tind1 3100 max 3174
maxsize Tind2 192200 max 195374
maxsize Tind3 11916400 max 12111774
maximum possible spans 12111774
        (12111774*Maxspansize = 12699970192536 bytes)
        (12111774*Maxspansize = 12699970192536 bytes = 11 TB)
```

Tests

tests/regress.rc: Regression tests.

tests/6.offsets: Write file using different offsets to test mafswrite().

tests/6.sizes: Show the effects of the different parameters.

tests/chkextents.rc: Unit tests on extents.

tests/6.testextents: Test extents.[ch] state changes.

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 removing the file
tests/test.3	write, read and deletion of 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

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 be portable to a machine using a different endian system.

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"