Outline

Basic file system layout

Btree algorithms & data structures

Space allocation interfaces and design

Inode allocation interfaces and design

Block mapping interfaces and design

Superblock

File system is divided into a number of Allocation Groups

All but the last are of equal size

Each allocation group starts with a super block (block 0) and an allocation group header (block 1)

One copy of superblock per allocation group, but only the first one is ever updated except when file system changes size

```
typedef struct xfs_sb {
	uuid_t			sb_uuid;	/* file system unique id */
	xfs_fsblock_t		sb_dblocks;	/* number of data blocks */
	__uint32_t		sb_blocksize;/* logical block size, bytes */
	/*
	 * sb_magic is at offset 28 to be at the same location as fs_magic
	 * in an EFS filesystem, thus ensuring there is no confusion.
	 */
	__uint32_t		sb_magicnum;	/* magic number == XFS_SB_MAGIC */
	xfs_fsblock_t		sb_rblocks;	/* number of realtime blocks */
	xfs_fsblock_t		sb_rbitmap;	/* bitmap for realtime blocks */
	xfs_fsblock_t		sb_rsummary;	/* summary for xfs_rbitmap */
	xfs_ino_t		sb_rootino;	/* root inode number */
	xfs_agblock_t		sb_rextsize;	/* realtime extent size, blocks */
	xfs_agblock_t		sb_agblocks;	/* size of an allocation group */
	xfs_agnumber_t		sb_agcount;	/* number of allocation groups */
	__uint16_t		sb_versionnum;/* header version == XFS_SB_VERSION */
	__uint16_t		sb_sectsize;	/* volume sector size, bytes */
	__uint16_t		sb_inodesize;/* inode size, bytes */
	__uint16_t		sb_inopblock;/* inodes per block */
	char			sb_fname[6];	/* file system name */
	char			sb_fpack[6];	/* file system pack name */
	__uint8_t		sb_blocklog;	/* log2 of xfs_blocksize */
	__uint8_t		sb_sectlog;	/* log2 of xfs_sectsize */
	__uint8_t		sb_inodelog;	/* log2 of xfs_inodesize */
	__uint8_t		sb_inopblog;	/* log2 of xfs_inopblock */
	__uint8_t		sb_smallfiles;/* set if small files in inodes */
	/* statistics */
	/* These fields must remain contiguous. If you really
	 * want to change their layout, make sure you fix the
	 * code in xfs_trans_apply_sb_deltas().
	 */
	__uint64_t		sb_icount;	/* allocated inodes */
	__uint64_t		sb_ifree;	/* free inodes */
	__uint64_t		sb_fdblocks;	/* free data blocks */
	__uint32_t		sb_frextents;/* free realtime extents */
} xfs_sb_t;
```

Allocation Group Header

Allocation group header (xfs_aghdr_t) contains information to find everything in its allocation group

Three sections:

1. Identification
2. Freespace information
3. Inode information

May split into multiple buffers (512 byte sections) to increase parallelism

```
typedef struct xfs_aghdr {
	__uint32_t	ag_magic;	/* magic number == XFS_AGH_MAGIC */
	__uint16_t	ag_version;	/* header version == XFS_AGH_VERSION */
	xfs_agnumber_t	ag_seqno;	/* sequence # starting from 0 */
	xfs_agblock_t	ag_length;	/* size in blocks of a.g. */
	/*
	 * Freespace information
	 */
	xfs_agblock_t	ag_roots[XFS_BTNUM_MAX - 1];/* BNO, CNT */
	xfs_agblock_t	ag_freelist;	/* free blocks */
	__uint16_t	ag_levels[XFS_BTNUM_MAX - 1];/* BNO, CNT */
	xfs_extlen_t	ag_flist_count;/* #blocks */
	xfs_extlen_t	ag_freeblks;	/* total free blocks */
	xfs_extlen_t	ag_longest;	/* longest free space */
	/*
	 * Inode information
	 * Inodes are mapped by interpreting the inode number, so no
	 * mapping data is needed here.
	 */
	xfs_agino_t	ag_icount;	/* count of allocated inodes */
	xfs_agino_t	ag_ifirst;	/* first allocated inode */
	xfs_agino_t	ag_ilast;	/* last allocated inode */
	xfs_agino_t	ag_iflist;	/* first free inode */
	xfs_agino_t	ag_ifcount;	/* number of free inodes */
} xfs_aghdr_t;
```

Btree Block Format

Btrees used in the space manager:

- freespace, sorted by block number (BNO)
- freespace, sorted by block size (CNT)
- block map, sorted by file offset (BMAP)

All use a common header structure xfs_btree_block_t and some common code.

Each btree block occupies one file system block exactly, unless it is being stored in an inode (and is smaller).

Each btree block consists of a header, some records, and some pointers. The record type varies from btree to btree; the pointer is a block number within an allocation group (xfs_agblock_t). The pointers are absent in leaf blocks.

```
typedef struct xfs_btree_block
{
       __uint32_t    bb_magic;     /* magic number for block type */
       __uint16_t    bb_level;     /* 0 is a leaf */
       __uint16_t    bb_numrecs;   /* current # of data records */
       xfs_agblock_t bb_leftsib;   /* left sibling block or NULLAGBLOCK */
       xfs_agblock_t bb_rightsib;  /* right sibling block or NULLAGBLOCK */
} xfs_btree_block_t;
```

Records for BNO and CNT btree blocks have this format:

```
typedef struct xfs_alloc_rec
{
       xfs_agblock_t ar_startblock;/* starting block number */
       xfs_extlen_t  ar_blockcount;/* count of free blocks */
} xfs_alloc_rec_t;
```

Records for BMAP btree blocks have this format:

```
typedef struct xfs_bmbt_rec
{
       __uint32_t    l0, l1, l2, l3;
} xfs_bmbt_rec_t;
/* l0:0-31 and l1:9-31 are startoff.
 * l1:0-8, l2:0-31, and l3:21-31 are startblock.
 * l3:0-20 are blockcount.
 */
```

Disk Inode Format (1)

Disk inodes contain a core section (xfs_dinode_core_t) and a union, which contains type-specific information. Important to the space manager are three formats: AGINO, EXTENTS, and BTREE.

AGINO is used to link free inodes together in a list.

EXTENTS is used to keep extent information when the extents will fit in the inode.

BTREE is used when the number of extents is too large to fit in the inode.

Timestamp values are currently 64 bits (32 bits of seconds since 1970, 32 bits of nanoseconds) although only the seconds are filled in and exported to users.

Disk Inode Format (2)

```
typedef struct xfs_dinode_core {
       __uint16_t          di_magic;     /* inode magic # = XFS_DINODE_MAGIC */
       __uint16_t          di_mode;      /* mode and type of file */
       __int8_t            di_version;   /* inode version */
       __int8_t            di_format;    /* format of di_c data */
       __int16_t           di_nlink;     /* number of links to file */
       __uint16_t          di_uid;       /* owner's user id */
       __uint16_t          di_gid;       /* owner's group id */
       xfs_extnum_t        di_nextents;  /* number of extents in file */
       __int64_t           di_size;      /* number of bytes in file */
       uuid_t              di_uuid;      /* file unique id */
       /*
        * While these fields hold 64 bit values, we will only
        * be using the upper 32 bits for now. The t_nsec
        * portion of the fields should always be zero. This
        * leaves room for expansion in the future if necessary.
        */
       xfs_timestamp_t     di_atime;     /* time last accessed */
       xfs_timestamp_t     di_mtime;     /* time last modified */
       xfs_timestamp_t     di_ctime;     /* time created/inode modified */
       /*
        * Should this be 64 bits? What does nfs3.0 want?
        */
       __uint32_t          di_gen;       /* generation number */
       xfs_agino_t         di_nexti;     /* next allocated inode in ag */
} xfs_dinode_core_t;

typedef struct xfs_dinode
{
       xfs_dinode_core_t   di_core;
       union {
              xfs_agino_t         di_next;      /* next inode for freelist inodes */
              dev_t               di_dev;       /* device for IFCHR/IFBLK */
              char                di_c[1];      /* local contents */
              xfs_bmbt_rec_t      di_bmx[1];    /* extent list */
              xfs_btree_block_t   di_bmbt;      /* btree root */
              uuid_t              di_muuid;     /* mount point value */
       }             di_u;
} xfs_dinode_t;

/*
 * Values for di_format
 */
typedef enum xfs_dinode_fmt
{
       XFS_DINODE_FMT_AGINO,/* free inodes: di_next */
       XFS_DINODE_FMT_DEV,/* CHR, BLK: di_dev */
       XFS_DINODE_FMT_LOCAL,/* DIR, REG, LNK: di_c */
       XFS_DINODE_FMT_EXTENTS,/* DIR, REG, LNK: di_bmx */
       XFS_DINODE_FMT_BTREE,/* DIR, REG, LNK: di_bmbt */
       XFS_DINODE_FMT_UUID /* MNT: di_uuid */
} xfs_dinode_fmt_t;
```

Inode Numbers

Inode numbers consist of three parts:

1. allocation group number
2. block number in the allocation group
3. index of the inode in its block

The current layout has:

```
/*
 * low sb_inopblog bits - offset in block
 * next 32 - sb_blocklog bits - block number in allocation group
 * next 32 bits - allocation group number
 * high sb_blocklog - sb_inopblog bits - 0
 */
```

The layout will be changed soon to have the middle field be shorter, its width will depend on the allocation group size. This will allow more inode numbers to fit in 32 bits.

Macros are used to construct and deconstruct inode numbers. The low two fields are also referred to as the “allocation group inode number”, or xfs_agino_t.

```
#define      xfs_mask(k)  ((1 << k) - 1)
#define      xfs_ino_offset_bits(s)((s)->sb_inopblog)
#define      xfs_ino_agbno_bits(s)(32 - (s)->sb_blocklog)
#define      xfs_ino_agino_bits(s)(xfs_ino_offset_bits(s) + xfs_ino_agbno_bits(s))
#define      xfs_ino_agno_bits(s)32

#define      xfs_ino_to_agno(s,i)((xfs_agnumber_t)((i) >> xfs_ino_agino_bits(s)))
#define      xfs_ino_to_agino(s,i)((xfs_agino_t)(i) & xfs_mask(xfs_ino_agino_bits(s)))
#define      xfs_ino_to_agbno(s,i)\
     (((xfs_agblock_t)(i) >> xfs_ino_offset_bits(s)) & xfs_mask(xfs_ino_agbno_bits(s)))
#define      xfs_ino_to_offset(s,i)((int)(i) & xfs_mask(xfs_ino_offset_bits(s)))

#define      xfs_agino_to_ino(s,a,i)(((xfs_ino_t)(a) << xfs_ino_agino_bits(s)) | (i))
#define      xfs_agino_to_agbno(s,i)((i) >> xfs_ino_offset_bits(s))
#define      xfs_agino_to_offset(s,i)((i) & xfs_mask(xfs_ino_offset_bits(s)))

#define      xfs_offbno_to_agino(s,b,o) \
                        ((xfs_agino_t)(((b) << xfs_ino_offset_bits(s)) | (o)))
```

Btree Cursors

Our current working position in a btree is recorded in a structure called a “cursor” (xfs_btree_cur_t). The cursor contains identifying information (which btree, for example) that allows generic code to be useful. It also contains a buffer pointer for each level in the btree, and an index number (pointer) of a record within that btree block.

Cursors are used as the interface with all btree services. For example, to do a btree insertion, there are the following steps:

1. Allocate / initialize a cursor
2. Do a lookup to find the insertion location and set the insertion value
3. Do an insert operation
4. Delete the cursor

```
typedef struct xfs_btree_cur
{
       xfs_trans_t   *bc_tp;       /* links cursors on freelist */
       xfs_mount_t   *bc_mp;       /* mount struct */
       buf_t         *bc_agbuf;    /* ag buffer */
       xfs_agnumber_tbc_agno;      /* ag number */
       union {
              xfs_alloc_rec_t      a;     /* btree records */
              xfs_bmbt_irec_t      b;
       }             bc_rec;
       buf_t         *bc_bufs[XFS_BTREE_MAXLEVELS];    /* buffer pointers */
       int           bc_ptrs[XFS_BTREE_MAXLEVELS];     /* record/pointer numbers */
       int           bc_nlevels;   /* number of btree levels */
       xfs_btnum_t   bc_btnum;     /* btree identifier */
       int           bc_blocklog;  /* block size of file system */
       union {
              struct {
                     int                 inodesize;    /* needed for BMAP */
                     struct xfs_inode    *ip;          /* needed for BMAP */
              } b;                 /* type b private data */
       }             bc_private;   /* type-private data */
} xfs_btree_cur_t;
```

Main Btree Interfaces

Btree code is replicated for extent allocation and file space allocation. Prefix for code is xfs_alloc_ or xfs_bmbt_.

The main interfaces are:

```
int delete(xfs_btree_cur_t *cur);
int insert(xfs_btree_cur_t *cur);
int lookup_eq(xfs_btree_cur_t *cur, xfs_agblock_t bno, xfs_extlen_t len);
int lookup_ge(xfs_btree_cur_t *cur, xfs_agblock_t bno, xfs_extlen_t len);
int lookup_le(xfs_btree_cur_t *cur, xfs_agblock_t bno, xfs_extlen_t len);
int update(xfs_btree_cur_t *cur, xfs_agblock_t bno, xfs_extlen_t len);
```

The arguments to the lookup and update calls match the elements of the btree records: block number and length for the freespace btrees; file offset, block number, and length for the bmap btree.

The lookup interfaces specify where (exactly) to stop the search: at a record exactly equal to, less than or equal to, or greater than or equal to the requested record. The lookup_gt and lookup_lt interfaces are not present because none of my code needs them.

Insert inserts a record whose value was specified by the most recent lookup call, or alternatively stored in the cursor by the caller. If the latter method is used, the cursor must still be pointing to the right place in the btree after the most recent lookup.

Delete deletes the record last looked up.

There are interfaces to adjust the cursor position other than by doing a lookup, given on the next slide.

Btree Service Routines

Some of these are lower-level routines in the btree implementations, some are secondary interfaces used by the callers of the high-level btree interfaces. The xfs_btree_... routines are shared between both implementations.

```
void xfs_btree_del_cursor(xfs_btree_cur_t *cur);
xfs_btree_cur_t *xfs_btree_dup_cursor(xfs_btree_cur_t *cur);
int xfs_btree_firstrec(xfs_btree_cur_t *cur, int level);
xfs_btree_cur_t *xfs_btree_init_cursor(xfs_mount_t *mp, xfs_trans_t *tp, buf_t *agbuf,
xfs_agnumber_t agno, xfs_btnum_t btno, struct xfs_inode *ip);
int xfs_btree_lastrec(xfs_btree_cur_t *cur, int level);

int decrement(xfs_btree_cur_t *cur, int level);
int delrec(xfs_btree_cur_t *cur, int level);
int get_rec(xfs_btree_cur_t *cur, xfs_agblock_t *bnop, xfs_extlen_t *lenp);
int increment(xfs_btree_cur_t *cur, int level);
int insrec(xfs_btree_cur_t *cur, int level, xfs_agblock_t *bnop, xfs_alloc_rec_t *recp,
xfs_btree_cur_t **curp);
int lookup(xfs_btree_cur_t *cur, xfs_lookup_t mode);
int lshift(xfs_btree_cur_t *cur, int level);
int newroot(xfs_btree_cur_t *cur);
int rshift(xfs_btree_cur_t *cur, int level);
int split(xfs_btree_cur_t *cur, int level, xfs_agblock_t *bnop, xfs_alloc_rec_t *recp,
xfs_btree_cur_t **curp);
void updkey(xfs_btree_cur_t *cur, xfs_alloc_rec_t *keyp, int level);
```

Decrement and increment are used to move around in the tree, as are (less frequently) firstrec and lastrec.

Insrec and delrec do one level’s worth of insert/delete for the insert/delete routines.

Lshift and rshift do balancing (1 record at a time) for insrec and delrec.

Split is used by insrec to do block splits.

Updkey marches up the tree setting the key values if the first record in a block is changed.

Space Allocation Interfaces

These interfaces are used to allocate and free extents of disk space.

```
typedef enum xfs_alloctype
{
       XFS_ALLOCTYPE_ANY_AG,      /* Any allocation group ok */
       XFS_ALLOCTYPE_START_AG,    /* start with this bno's allocation group */
       XFS_ALLOCTYPE_THIS_AG,     /* must be somewhere in bno's allocation group */
       XFS_ALLOCTYPE_NEAR_BNO,    /* must be in bno's ag and near bno */
       XFS_ALLOCTYPE_THIS_BNO     /* must be at exactly bno */
} xfs_alloctype_t;

xfs_fsblock_t xfs_alloc_extent(xfs_trans_t *tp, xfs_fsblock_t bno, xfs_extlen_t len,
xfs_alloctype_t flag);
xfs_agblock_t xfs_alloc_next_free(xfs_mount_t *mp, xfs_trans_t *tp, buf_t *agbuf,
xfs_agblock_t bno);
xfs_fsblock_t xfs_alloc_vextent(xfs_trans_t *tp, xfs_fsblock_t bno,
xfs_extlen_t minlen, xfs_extlen_t maxlen, xfs_extlen_t *len, xfs_alloctype_t flag);
int xfs_free_extent(xfs_trans_t *tp, xfs_fsblock_t bno, xfs_extlen_t len);
```

The generic allocation routine is xfs_alloc_vextent (variable extent); xfs_alloc_extent is implemented with it.

It is given a transaction, an allocation mode, a minimum and a maximum length, and a block number. It attempts to allocate the maximum block it can, less than or equal to maxlen, greater than or equal to minlen.

The flag argument controls its behavior as given in the comments above.

The free_extent routine just frees the extent given by bno and len; the frees don't have to match the allocations.

The next_free routine is used to scan the freelist held by this code; it's really an internal routine, exposed for the debugging code to print the list with.

Space Allocation Internals

Space allocation maintains two btrees representing free space in the file system. The btrees are named BNO and CNT. BNO is sorted by block number, CNT by extent length. Each contains exactly the same data, only in a different order.

Each allocation first selects an allocation group to allocate from; this is iterative for XFS_ALLOCTYPE_ANY_AG and XFS_ALLOCTYPE_START_AG. Then the routine xfs_alloc_ag_vextent is called, with arguments as for xfs_alloc_vextent plus an allocation group number and buffer. The flag argument has been reduced to the three in-allocation group cases.

xfs_alloc_ag_vextent reduces to a switch out to routines xfs_alloc_ag_vextent_{size, near, exact}, plus bookkeeping.

These three routines implement all the allocation policy.

xfs_free_extent reduces to some address translation and a call to xfs_free_ag_extent. It checks for adjacent freespace, and merges it with the new freespace if present. The two btrees are adjusted by inserts and updates to accomplish this.

To avoid running out of space in the middle of operations, and to avoid calling itself recursively to obtain space, this code maintains a set of blocks on a list, that it can use.

xfs_alloc_ag_vextent_size (example)

```
xfs_agblock_t
xfs_alloc_ag_vextent_size(xfs_trans_t *tp, buf_t *agbuf, xfs_agnumber_t agno, xfs_ex-
tlen_t minlen, xfs_extlen_t maxlen, xfs_extlen_t *len)
{
	xfs_btree_cur_t *bno_cur;
	xfs_btree_cur_t *cnt_cur;
	xfs_agblock_t fbno;
	xfs_extlen_t flen = 0;
	int i;
	xfs_extlen_t rlen;
	xfs_mount_t *mp;

	mp = tp->t_mountp;
	cnt_cur = xfs_btree_init_cursor(mp, tp, agbuf, agno, XFS_BTNUM_CNT, 0);
	if (!xfs_alloc_lookup_ge(cnt_cur, 0, maxlen)) {
		if (xfs_alloc_decrement(cnt_cur, 0))
			xfs_alloc_get_rec(cnt_cur, &fbno, &flen);
		if (flen < minlen) {
			xfs_btree_del_cursor(cnt_cur);
			return NULLAGBLOCK;
		}
		rlen = flen;
	} else {
		xfs_alloc_get_rec(cnt_cur, &fbno, &flen);
		rlen = maxlen;
	}
	xfs_alloc_delete(cnt_cur);
	bno_cur = xfs_btree_init_cursor(mp, tp, agbuf, agno, XFS_BTNUM_BNO, 0);
	i = xfs_alloc_lookup_eq(bno_cur, fbno, flen);
	if (rlen < flen) {
		xfs_alloc_lookup_eq(cnt_cur, fbno + rlen, flen - rlen);
		xfs_alloc_insert(cnt_cur);
		xfs_alloc_update(bno_cur, fbno + rlen, flen - rlen);
	} else
		xfs_alloc_delete(bno_cur);
	*len = rlen;
	xfs_btree_del_cursor(bno_cur);
	xfs_btree_del_cursor(cnt_cur);
	return fbno;
}
```

Inode Allocation Interfaces

Disk inode allocation is done in the layer below xfs_ialloc. The routine xfs_dialloc turns a free inode into an allocated inode and returns it to the caller. It also may allocate more inodes (turn free blocks into inode blocks) if necessary.

xfs_difree is used to turn used inodes into free inodes.

We never turn free inodes into free blocks, although I suppose we could.

xfs_dilocate is used to turn an inode number into an inode location; it is very simple. The inode's location is returned in arguments bno and off.

In an earlier version of this code, btrees were used to map inodes to locations; this has all been removed in favor of storing the block numbers in the inode numbers.

```
xfs_ino_t xfs_dialloc(xfs_trans_t *tp, xfs_ino_t parent, int sameag, mode_t mode);
xfs_agino_t xfs_dialloc_next_free(xfs_mount_t *mp, xfs_trans_t *tp, buf_t *agbuf,
xfs_agino_t agino);
xfs_agino_t xfs_difree(xfs_trans_t *tp, xfs_ino_t inode);
int xfs_dilocate(xfs_mount_t *mp, xfs_trans_t *tp, xfs_ino_t ino, xfs_fsblock_t *bno,
int *off);
```

Inode Allocation Internals

This code is very simple now. There are two internal routines:

```
int                             /* success/failure */
xfs_ialloc_ag_alloc(xfs_trans_t *tp,    /* transaction pointer */
                    buf_t *agbuf);      /* alloc grp buffer */

buf_t *                         /* allocation group buffer */
xfs_ialloc_ag_select(xfs_trans_t *tp,   /* transaction pointer */
                     xfs_ino_t parent,  /* parent directory inode number */
                     int sameag,        /* =1 to force to same ag. as parent */
                     mode_t mode);      /* bits set to indicate file type */
```

Xfs_ialloc_ag_select is used to select an allocation group to do the inode allocation in; the code in xfs_dialloc iterates through other allocation groups, if the first is unsuccessful, and it isn't told not to.

Xfs_ialloc_ag_alloc allocates new inodes from freespace, if necessary to satisfy the xfs_dialloc request.

Inodes are allocated in chunks of from 1 block to 256 inodes or 16 blocks, whichever is smaller. A variable extent allocation is used, where the block number is one past the previously allocated inodes.

Block Mapping Interfaces

The main interface to block mapping is xfs_bmap. This routine does not exist yet.

The routine xfs_bmapi is an inode version of xfs_bmap. It is given an inode, a file offset and length, a read/write flag, and an array of extent descriptors (and its size). It fills in the extent descriptors, and for writes into holes (or past the end of file) it allocates space to the file as well.

```
void xfs_bmapi(xfs_mount_t *mp, xfs_trans_t *tp, struct xfs_inode *ip, xfs_fsblock_t
bno, xfs_extlen_t len, int rw, xfs_bmbt_irec_t *maps, int *nmaps);
```

In-core Inode Format

Only the portion of the inode relevant to block mapping is discussed here!

The inode points to an array of extents, which is sorted by offset in the file. This array is generated (for btree-format files) by the routine xfs_bmap_read_extents; for extent-format files the array lives in the disk inode and is just copied into the incore array.

For btree-format files, the bmap btree root is also pointed to by the incore inode. The btree code manipulates this root just as it manipulates all the other blocks in the tree, with all the macros knowing the difference.

Block Mapping Internals

Block mapping for extent-format files is simple; the extent list in core is searched (binary, eventually). New records are inserted or appended to the list.

For btree-format files, in addition, the btree must be searched (xfs_bmbt_lookup_eq) and then inserted to or updated depending on whether the new extent is adjacent to the old extents or not. The btree interfaces used are exactly the same as for freespace allocation.

xfs_bmapi has four basic sections:

1. Figure out where the first extent-list record to look at is.
2. Map (and allocate if necessary) the file space into file system blocks
3. If the file is extents-format and now has too many extents to fit in the disk inode, convert the file to btree-format.
4. Log all the changes.