Lecture 32: File system implementation in xv6

Mythili Vutukuru
IIT Bombay
https://www.cse.iitb.ac.in/~mythili/os/

Disk layout

- Disk in xv6 is formatted to contain superblock, log (for crash recovery), inode blocks (multiple inodes packed per block), bitmap (indicating which data blocks are free), actual data blocks
- Disk inode contains block numbers of direct blocks and one indirect block
- Directory is special file: data blocks contain directory entries, mapping file names of files in the directory to corresponding inode numbers
- Link count of inode = number of directory entries pointing to a file inode

```
// On-disk file system format.
// Both the kernel and user programs use this header file.


#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size

// Disk layout:
// [ boot block | super block | log | inode blocks |
//                                          free bit map | data blocks]
//
// mkfs computes the super block and builds an initial file system. The
// super block describes the disk layout:
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
```

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

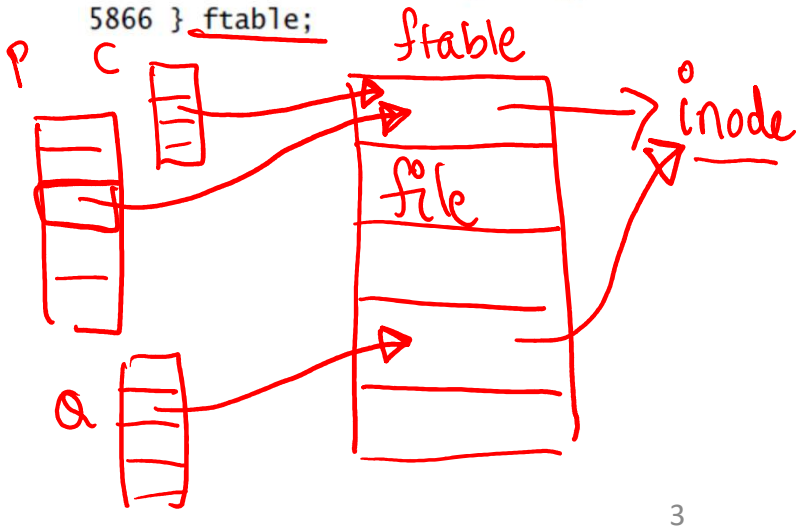
In-memory data structures (1)

- Every open file has a struct file associated with it
  - Pointer to inode or pipe structure
- All struct files stored in fixed size array called file table (ftable)
- File descriptor array of a process contains pointers to struct files in the file table
- Two processes P and Q open same file, will use two struct file entries in file table
  - Points to same inode
  - Read and write independently at different offsets
- P forks child C, both file descriptors will point to same struct file (ref is increased)
  - Offsets are shared
- Reference count of struct file is number of file descriptors that point to it

```
struct file *ofile[NOFILE];  // Open files
```

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

```
struct devsw devsw[NDEV];
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```



In-memory data structures (2)

- Struct file points to in-memory inode structure of an open file (pipe structure for pipes)
- In-memory inode is almost copy of disk inode, stored in memory for open files
- All in-memory inodes stored in fixed size array called inode cache (icache)
- Reference count of in-memory inode is number of pointers from file table entries, current working directory of process etc.
  - Different from link count
  - A file is cleaned up on disk only when both ref count and link count are zero

inode

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;
```

Inode functions (1)

- Function ialloc() allocates free inode from disk by looking over disk inodes and finding a free one for a file
- Function iget() returns a reference counted pointer to in-memory inode in icache, to use in struct file etc
  - Non-exclusive pointer, information inside inode structure may not be up to date
  - Pointer released by iput()
- Function ilock() locks the inode for use by a process, and updates its information from disk if needed
  - Unlocked by iunlock()
- Function iupdate() propagates changes from in-memory inode to on-disk inode

ialloc
iget
ilock
iupdate

Inode functions (2)

- Inode has pointers to file datablocks
- Function bmap returns the address of n-th block of file
  - If direct block, read from inode
  - If indirect block, read indirect block first and then return block number from it
- Function can allocate data blocks too: if n-th data block of file not present, allocates new block on disk, writes it to inode, and returns address
- Functions readi/writei are used to read/write file data at given offset, call bmap to find corresponding data block

```
// Inode content
//
// The content (data) associated with each inode is stored
// in blocks on the disk. The first NDIRECT block numbers
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in block ip->addrs[NDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}
```

Directory functions

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

- Directory lookup: read directory entries from the data blocks of directory. If file name matches, return pointer to inode from icache
- Linking a file to a directory: check file with the same name does not exist, and add the mapping from file name to inode number to directory

```
// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```

```
// Write a new directory entry (name, inum) into the directory dp.
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }

  // Look for an empty dirent.
  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;
  }

  strncpy(de.name, name, DIRSIZ);
  de.inum = inum;
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("dirlink");

  return 0;
}
```

Creating a file (if it doesn't exist)

- Locate the inode of parent directory by walking the filepath from root (lookup root inode, find inode number of next element of pathname in inode data blocks, and repeat)
- Lookup filename in parent directory. If file already exists, return its inode
- If file doesn't exist, allocate a new inode for it, lock it, initialize it
- If new file is a directory, add entries for "." and ".."
- If new file is a regular file, link it to its parent directory

/a/b/foo.txt

```
static struct inode*
create(char *path, short type, short major, short minor)
{
  uint off;
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);

  if((ip = dirlookup(dp, name, &off)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    iunlockput(ip);
    return 0;
  }

  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);

  if(type == T_DIR){  // Create . and .. entries.
    dp->nlink++;  // for ".."
    iupdate(dp);
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      panic("create dots");
  }

  if(dirlink(dp, name, ip->inum) < 0)
    panic("create: dirlink");

  iunlockput(dp);

  return ip;
}
```

System call: open

- Get arguments: filename, mode
- Create file (if specified) and get a pointer to its inode
- Allocate new struct file in ftable, and new file descriptor entry in struct proc of process pointing to the struct file in ftable
- Return index of new entry in file descriptor array of process
- Note the begin_op and end_op for transactions

```
int
sys_open(void)
{
  char *path;
  int fd, omode;
  struct file *f;
  struct inode *ip;

  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
  return fd;
}
```

fd → ftable → inode

System call: link

ln old new

- Link an existing file from another directory with a new name (hard linking)
- Get pointer to file inode by walking the old filename
- Update link count in inode
- Get pointer to inode of new directory, and link old inode from parent directory in new name

```
// Create the path new as a link to the same inode as old.
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;

  begin_op();
  if((ip = namei(old)) == 0){
    end_op();
    return -1;
  }

  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    end_op();
    return -1;
  }

  ip->nlink++;
  iupdate(ip);
  iunlock(ip);

  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);

  end_op();

  return 0;

bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return -1;
}
```

new dir

old inode

System call: file read

- Other system calls follow same pattern
- For example, file read:
  - Get arguments (file descriptor number, buffer to read into, number of bytes to read)
  - Fetch inode pointer from struct file and perform read on inode (or pipe if file descriptor pointed to pipe)
  - Function readi uses the function "bmap" to get the block corresponding to n-th byte and reads from it
  - Offset in struct file updated

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

```
// Read from file f.
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

Summary

- On disk: inodes, data blocks, free bitmap (and log)
- In-memory: file descriptor array (points to) struct file in file table array (points to) in-memory inode in inode cache
- Directory is a special file, where data blocks contain directory entries (filenames and corresponding inode numbers)
- System calls related to files extract arguments, perform various operations on in-memory and on-disk data structures
- Updates to disk happen via the buffer cache
  - Changes to all blocks in a system call are wrapped in a transaction and logged for atomicity