File System Code

open,read, write, close, pipe, fstat, chdir, dup, mknod, link, unlink, mkdir,

Files, Inodes, Buffers

What we already know

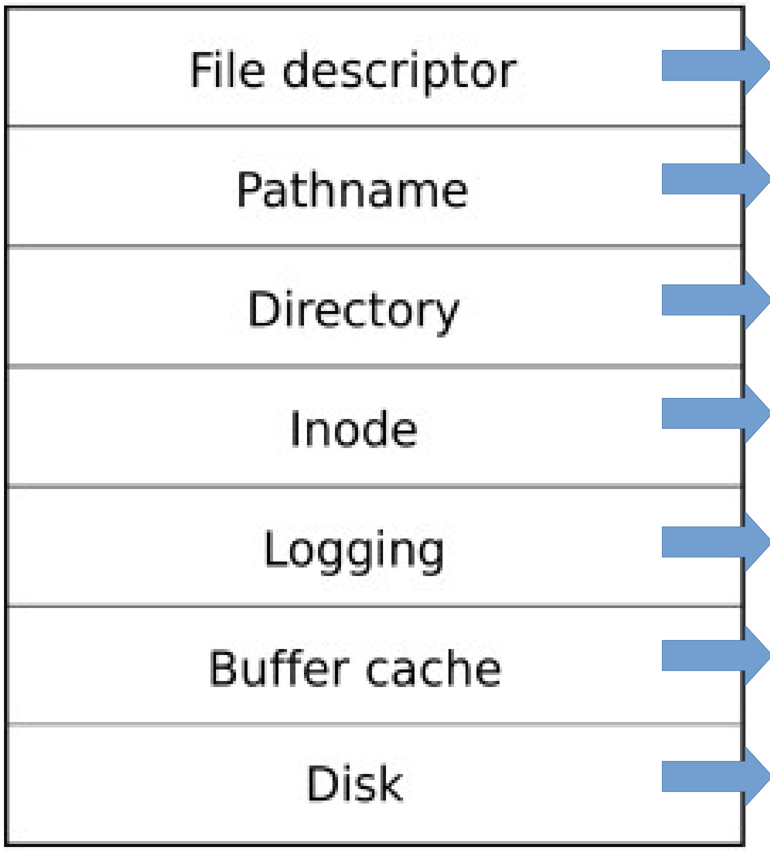
- File system related system calls
  - deal with ‘fd’ arrays (ofile in xv6). open() returns first empty index. open should ideallly locate the inode on disk and initialize some data structures
  - maintain ‘offsets’ within a ‘file’ to support sequential read/write
  - dup() like system calls duplicate pointers in fd-array
  - read/write like system calls, going through ‘ofile’ array, should locate data of file on disk
  - We need functions to read/write from disk – that is disk driver
  - cache data of files in OS data structures for performance : buffering
  - Need to handle on disk data structures as well
- Faster recovery (like journaling in ext3) is desired

xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartamentalized

Layers of xv6 file system code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,

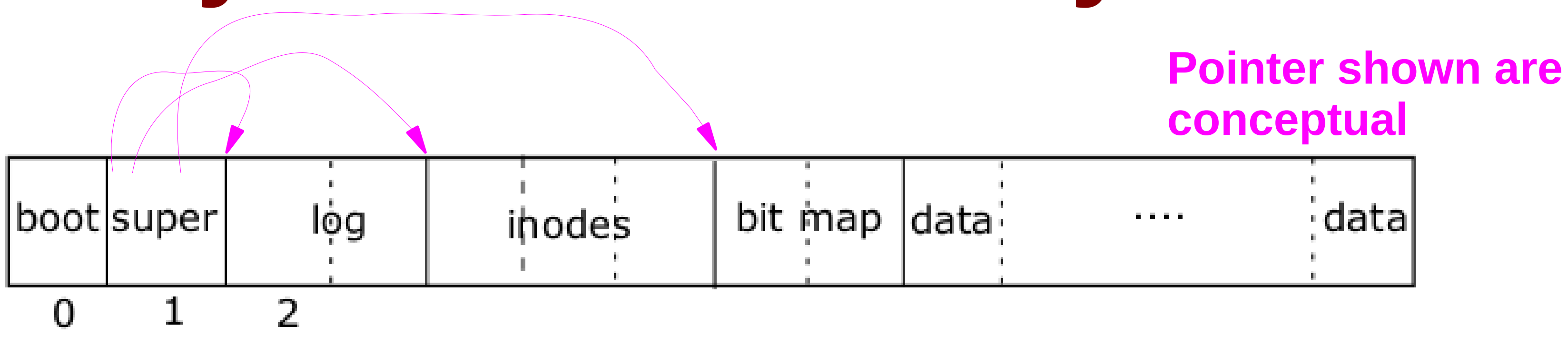


File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

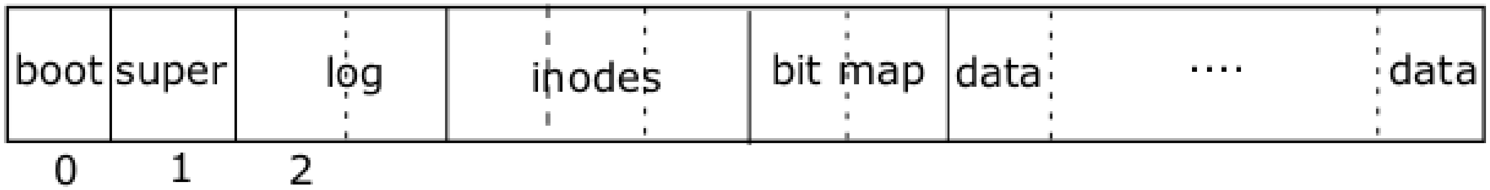
Layout of xv6 file system



May see the code of mkfs.c to get insight into the 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
};
#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size
```

Layout of xv6 file system

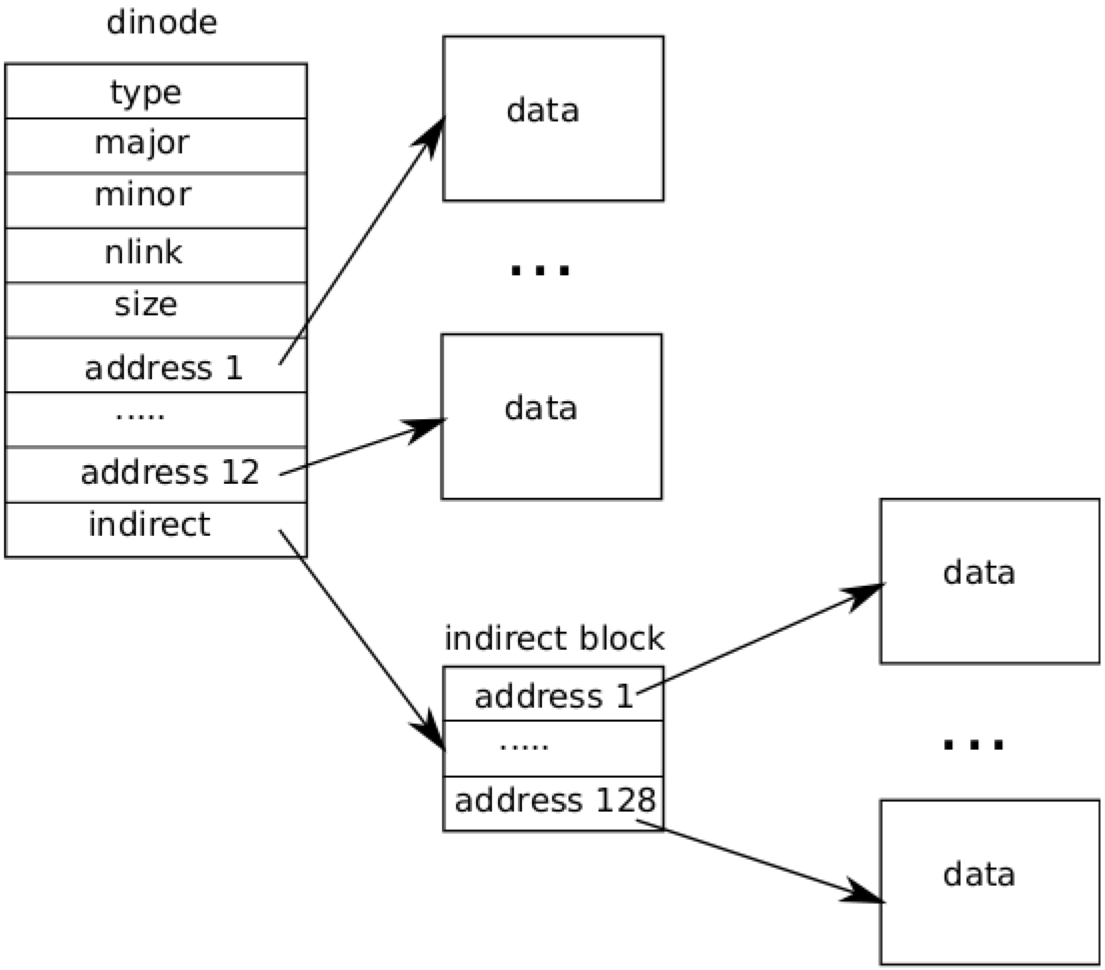



```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};

#define DIRSIZ 14

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

File on disk


dinode
type
major
minor
nlink
size
address 1
.....
address 12
indirect
data
...
data
indirect block
address 1
.....
address 128
data
...
data

Let’s discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

ide.c: idewait, ideinit, idestart, ideintr, iderw

static struct spinlock idelock;

static struct buf *idequeue;

static int havedisk1;

- ideinit
  - was called from main.c: main()
  - Initialized IDE controller by writing to certain ports
  - havedisk=1 setup
  - Initialize idelock
- idewait
  - BUSY loop waiting for IDE to be ready

ide.c: idewait, ideinit, idestart, ideintr, iderw

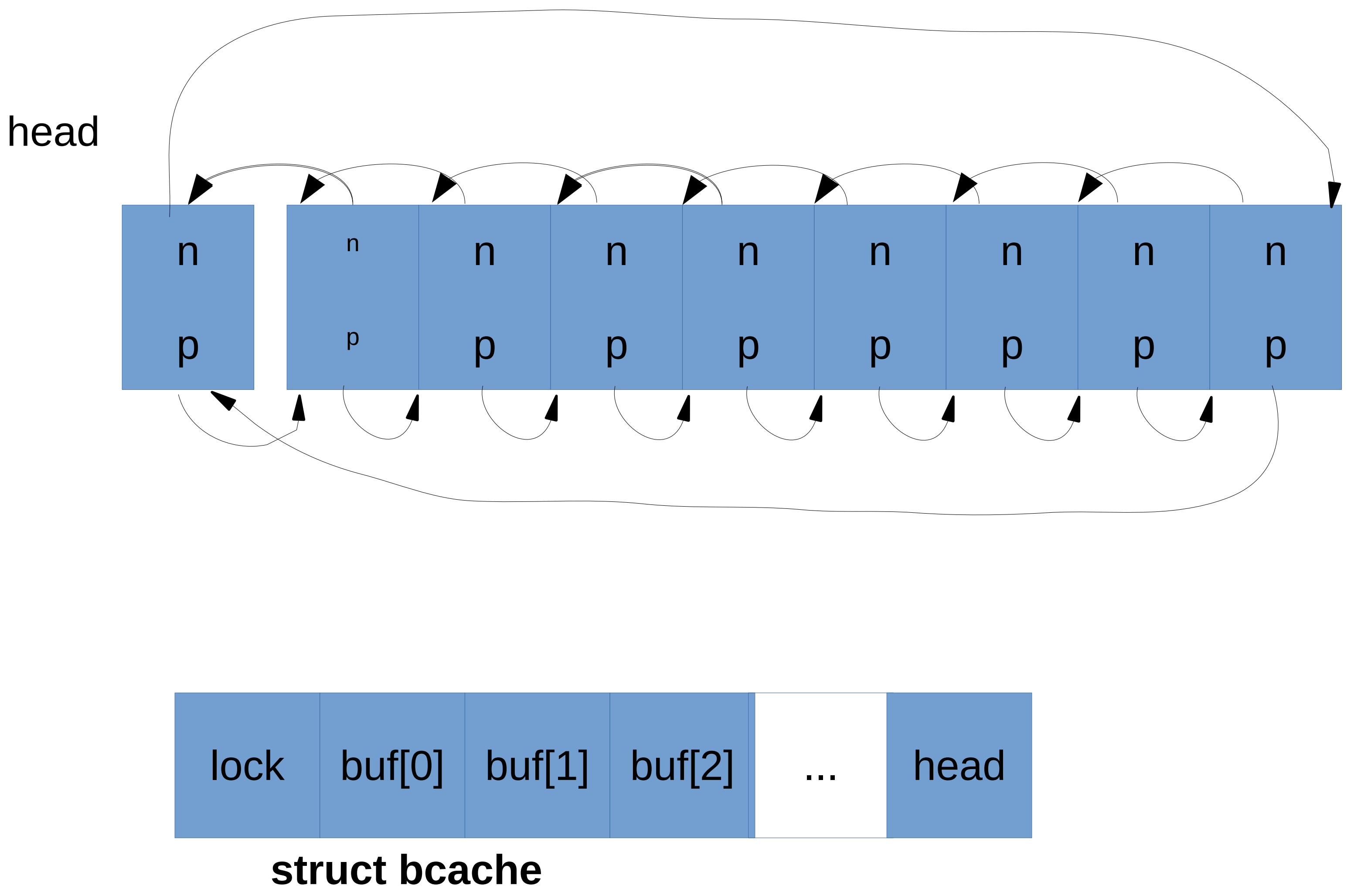
- void idestart(buf *b)
  - static void idestart(struct buf *b)
  - Calculate sector number on disk using b->blockno
  - Issue a read/write command to IDE controller.
  - (This is the first buf on idequeue)
- ideintr
  - Take idelock. Called on IDE interrupt (through alltraps()->trap())
  - Wakeup the process waiting on first buffer in buffer *idequeue;
  - call idestart(). Release idelock.
- iderw(buf *b)
  - Move buf b to end of idequeue
  - Call idestart() if not running, sleep on idelock

Let’s see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Reminder: After main()->binit()



Conceptually Linked liks this

Buffers keep moving on list, as LRU

struct buf

```
struct buf {
  int flags;  // 0 or B_VALID or B_DIRTY
  uint dev;   // device number
  uint blockno; // seq block number on device
  struct sleeplock lock; // Lock to be held by process using it
  uint refcnt; // Number of live accesses to the buf
  struct buf *prev; //  cache list
  struct buf *next; // cache list
  struct buf *qnext; // disk queue
  uchar data[BSIZE];  // data 512 bytes
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

buffer cache:
static struct buf* bget(uint dev, uint blockno)

- The bcache.head list is maintained on Most Recently Used (MRU) basis
  - head.next is the Most Recently Used (MRU) buffer
  - hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with b->blockno = blockno and b->dev = dev
  - Search the head.next list for existing buffer (MRU order)
  - Else search the head.prev list for empty buffer
  - panic() if found in-use or empty buffer
- Increment b->refcnt ; Returns buffer locked
- Does not change the list structure, just returns a buf in use

buffer cache:
struct buf* bread(uint dev, uint blockno)

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;
  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b; // locked buffer
}
```

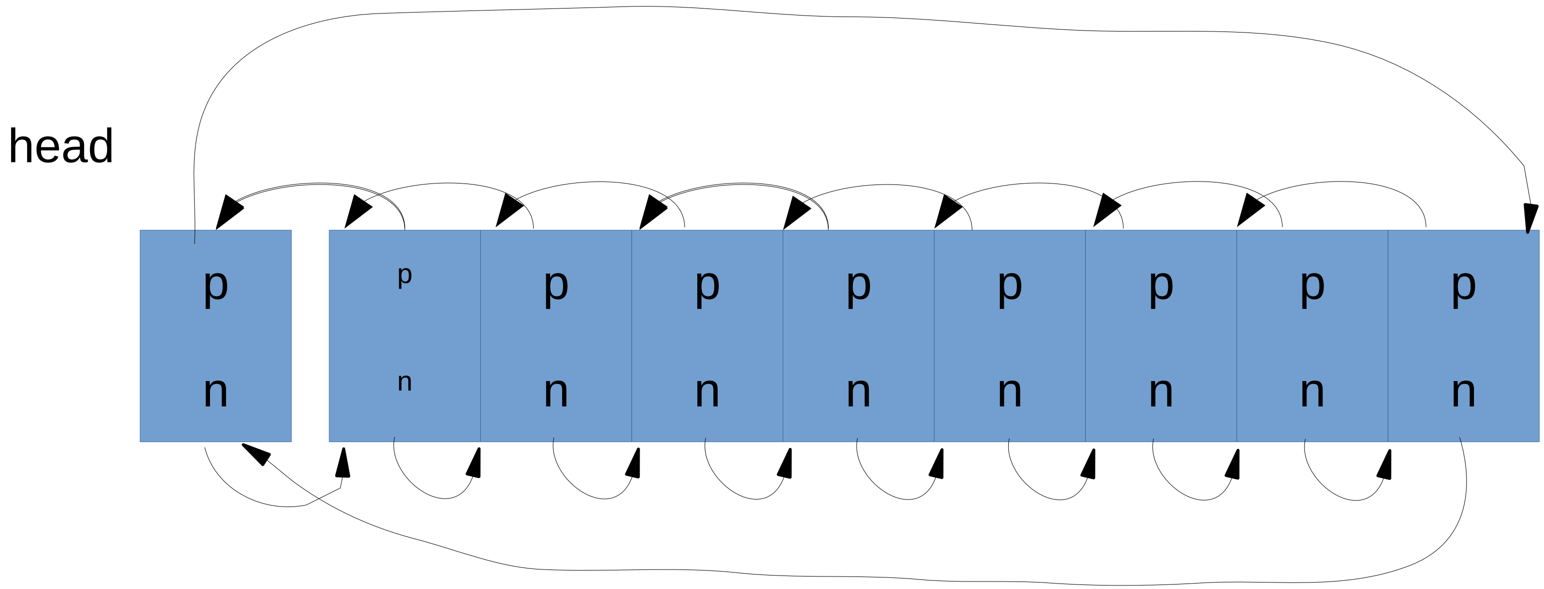
```
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Recollect: iderw moves buf to tail of idequeue, calls idestart() and sleep()

buffer cache:
void brelse(struct buf *b)

- release lock on buffer
- b->refcnt = 0
- If b->refcnt = 0
  - Means buffer will no longer be used
  - Move it to front of the front of bcache.head

Overall in this diagram



Buffers keep moving to the front of the list and around
The list always contains NBUF=30 buffers
head.next is always the MRU and head.prev is always LRU buffer

File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

data structures related to “file” layer

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // used only if it works as a pipe
  struct inode *ip;
  uint off;
};
// interesting no lock in struct file !
```

```
struct proc {
...
  struct file *ofile[NOFILE];  // Open files per process
...
}
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable; //global table from which ‘file’ is allocated to every process
```

Lock is used to protect updates to every entry in the array

Multiple processes accessing same file.

- Each will get a different ‘struct file’
  - but share the inode !
  - different offset in struct file, for each process
  - Also true, if same process opens file many times
- File can be a PIPE (more later)
  - what about STDIN, STDOUT, STDERR files ?
  - Figure out!
- ref
  - used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

file layer functions

- filealloc
  - find an empty struct file in ‘ftable’ and return it
  - set ref = 1
- filedup(file *)
  - simply ref++
- fileclose
  - --ref
  - if ref = 0
    - free struct file
    - iput() / pipeclose()
    - note – transaction if iput() called
- filestat
  - simply return fields from inode, after holding lock. on inodes for files only.

file layer functions

- filereead
  - call readi() or piperead()
  - readi() later calls device-read or inode read (using bread())
- filewrite
  - call pipewrite() or writei()
  - writei() is called in a loop, within a transaction

- Why does readi() call read on the device , why not fileread() itself call device read ?

Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

directory entry

```
#define DIRSIZ 14

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

Data of a directory file is a sequence of such entries. To find a name, just get all the data blocks and search the name

How to get the data for a directory? We already know the ans!

struct inode* dirlookup(struct inode *dp, char *name, uint *poff)

- Given a pointer to directory inode (dp), name of file to be searched
  - return the pointer to inode of that file (NULL if not found)
  - set the ‘offset’ of the entry found, inside directories data blocks, in poff
- How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?
  - During resolution of pathnames?
- Code: call readi() to get data of dp, search name in it, name comes with inode-num, iget() that inode-num

int
dirlink(struct inode *dp, char *name, uint inum)

- Create a new entry for ‘name’_’inum’ in directory given by ‘dp’
  - inode number must have been obtained before calling this. How to do that?
- Use dirlookup() to verify entry does not exist!
- Get empty slot in directory’s data block
- Make directory entry
- Update directory inode! writei()

namex

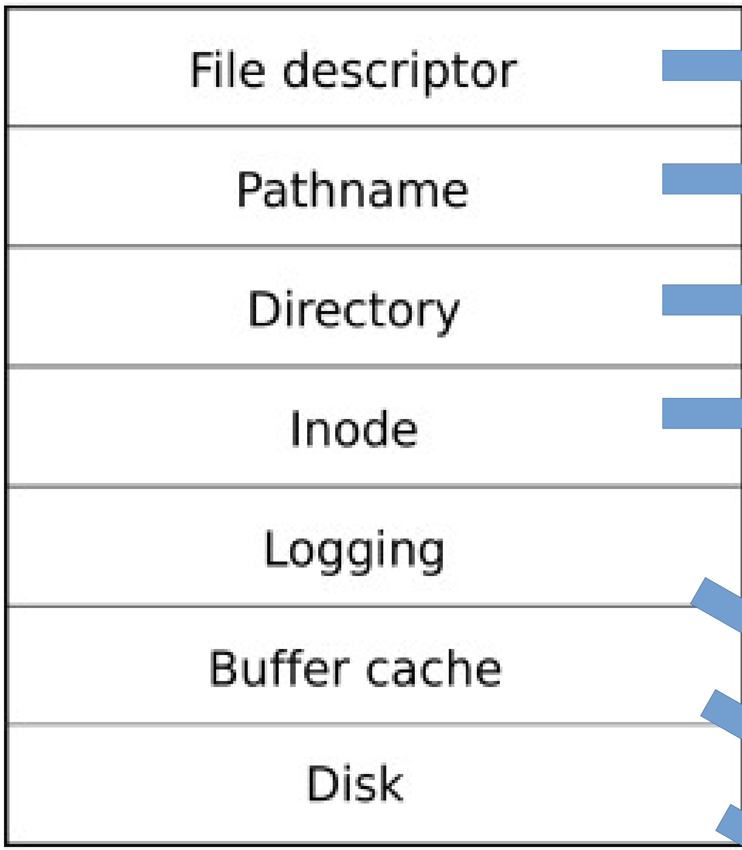
- Called by namei(), or nameiparent()
- Just iteratively split a path using “/” separator and get inode for last component
- iget() root inode, then
- Repeatedly calls
  - split on “/”, dirlookup() for next component
-

races in namex()

- Crucial. Called so many times!
- one kernel thread is looking up a pathname another kernel thread may be changing the directory by calling unlink
  - when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup() returns an inode that was obtained using iget.
- Deadlock? next points to the same inode as ip when looking up ".". Locking next before releasing the lock on ip would result in a deadlock.
  - namex unlocks the directory before obtaining a lock on next.

Let’s see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

On disk & in memory inodes

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

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

```
// 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;          //  been read from disk?

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

In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The iget and iput functions acquire and release pointers to an inode, modifying the ref count.
- See the caller graph of iget()
  - all those who call iget()
- Sleep lock in ‘inode’ protects
  - fields in inode
  - data blocks of inode

iget and iupdate

- iget
  - searches for an existing/free inode in icache and returns pointer to one
  - if found, increments ref and returns pointer to inode
  - else gets empty inode , initializes, ref=1 and return
  - No lock held after iget()
  - Code must call ilock() after iget() to get lock
  - During lookup (later), many processes can iget() an inode, but only one holds the lock

- iupdate(inode *ip)
  - read on disk block of inode
  - get on disk inode
  - modify it as specified in ‘ip’
  - modify disk block of inode
  - log_write(disk block of inode)

itrunc , iput

- iput(ip)
  - if ref is 1
    - itrunc(ip)
    - type = 0
    - iupdate(ip)
    - i->valid = 0 // free in memory
  - else
    - ref--

- itrunc(ip)
  - write all data blocks of inode to disk
    - using bfree()
  - ip->size = 0
    - Inode is freed from use
  - iupdate(ip)
  - called from iput() only when ‘ref’ becomes zero

race in iput ?

- A concurrent thread might be waiting in ilock to use this inode
  - and won't be prepared to find the in ode is not longer allocated
- This is not possible. Why?
  - no way for a syscall to get a ref to a inode with ip->ref = 1

```
void
iput(struct inode *ip)
{
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
      // inode has no links and no
other references: truncate and free.
      itrunc(ip);
```

buffer and inode cache

- to read an inode, it's block must be read in a buffer
- So the buffer always contains a copy of the on-disk dinode
  - duplicate copy in in-memory inode
- The inode cache is write-through,
  - code that modifies a cached inode must immediately write it to disk with iupdate
- Inode may still exist in the buffer cache

allocating inode

- ialloc(dev, type)
  - Loop over all disk inodes
  - read inode (from it’s block)
  - if it’s free (note inum)
  - zero on disk inode
  - write on disk inode (as zeroes)
  - return iget(dev, inum)
- panic if no free inodes

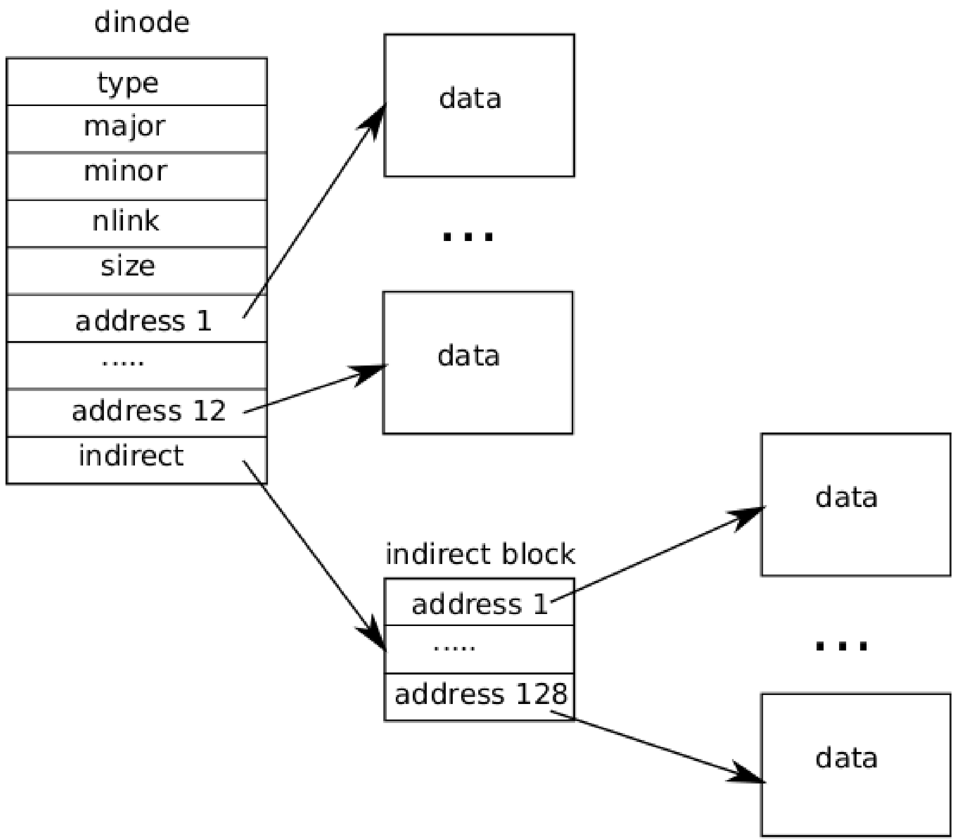
- ilock
  - code must acquire ilock before using inode’s data/fields
  - Ilock reads inode if it’s already not in memory

Trouble with iput() and crashes

- iput() doesn't truncate a file immediately when the link count for the file drops to zero, because
  - some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.
- if a crash happens before the last process closes the file descriptor for the file,
  - then the file will be marked allocated on disk but no directory entry points to it
- Unsolved problem.
- How to solve it?

Get Inode data: bmap(ip, bn)

- **Allocate ‘bn’th block for the file given by inode ‘ip’**
- **Allocate block on disk and store it in either direct entries or block of indirect entries**
  - **allocate block of indirect entries if needed using balloc()**



writing/reading data at a given offset in file

readi(struct inode *ip, char *dst, uint off, uint n)

writei(struct inode *ip, char *src, uint off, uint n)

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
- using bread(), brelse()
- Call devsw.read if inode is a device Inode.
- Writei() also updates size if required

Let’s see block allocation layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

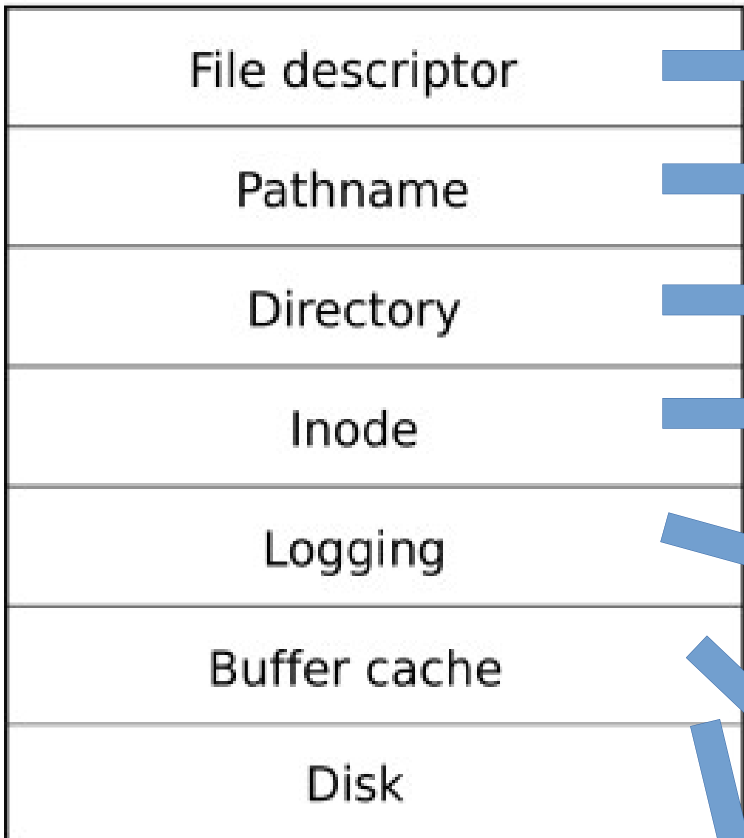
allocating & deallocating blocks on DISK

- balloc(devno)
  - looks for a block whose bitmap bit is zero, indicating that it is free.
  - On finding updates the bitmap and returns the block.
  - balloc() calls bread()->bget to get a block from disk in a buffer.
    - Race prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.
  - Calls log_write(bp);
    - Thus writes to bitmap blocks are also logged

- bfree(devno, blockno)
  - finds the right bitmap block and clears the right bit.
  - Also calls log_write()

Let’s see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Recovery

- Problem. Consider creating a file on ext2 file system.
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

Recovery

- Consistency checking (e.f. fsck command) – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Recovery

- Is a critical problem!
- Downtime is un-desired!
- A attempt at the solution: log structured / journaling file systems, e.g. ext3

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

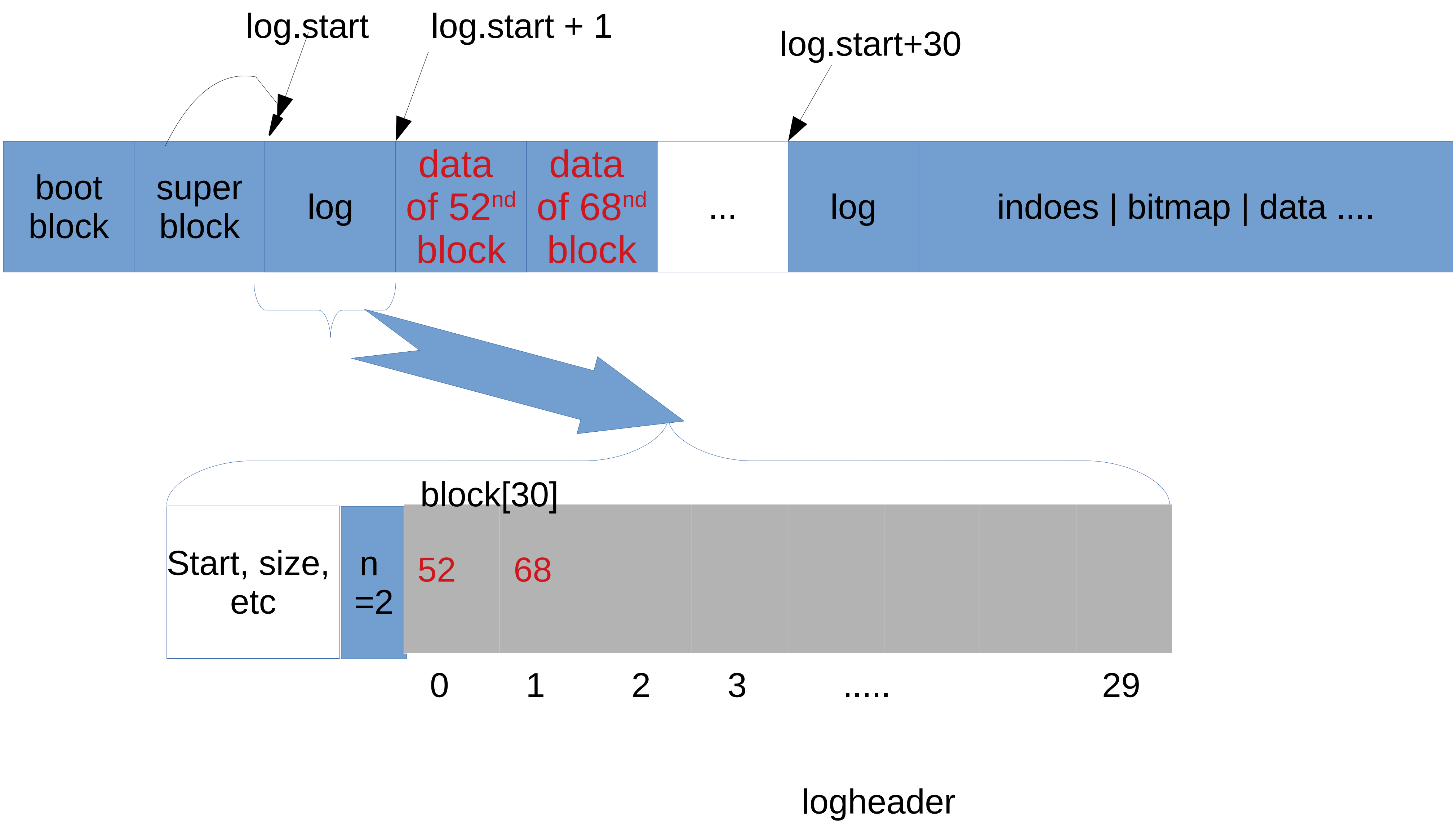
log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

log in xv6

- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin_op() at begining and end_op() at end
  - begin_op() increments log.outstanding
  - end_op() decrements log.outstanding, and if it’s 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
  - log_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in it’s actual place in FS as of now
- when finally commit() is called, all modified blocks are copied to disk in the file system

log on disk


log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

log

```
struct logheader { // ON DISK
  int n; // number of entries in use in block[] below
  int block[LOGSIZE]; // List of block numbers stored
};
struct log { // only in memory
  struct spinlock lock;
  int start;  // first log block on disk (starts with logheader)
  int size; // total number of log blocks (in use out of 30)
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev; // FS device
  struct logheader lh;  // copy of the on disk logheader
};
struct log log;
```

Typical use case of logging

```
/* In a system call code */
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

prepare for logging. Wait if logging system is not ready or ‘committing’. ++outstanding

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark B_DIRTY. Absorb multiple writes into one.

Syscall done. write log and all blocks. --outstanding.

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

```
//file_write() code
begin_op();
ilock(f->ip);
  /*loop */ r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

- each writei() in turn calls bread(), log_write() and brelse()
  - also calles iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are combined between begin_op() and end_op()

Logging functions

- Initlog()
  - Set fields in global log.xyz variables, using FS superblock
  - Recovery if needed
  - Called from first forkret()
- Following three called by FS code
- begin_op(void)
  - Increment log.outstanding
- end_op(void)
  - Decrement log.oustanding and call commit() if it's zero
- log_write(buf *)
  - Remember the specified block number in log.lh.block[] array
  - Set the block to be dirty
- write_log(void)
  - Called only from commit()
  - Use block numbers specified in log.lh.block and copy those blocks from memory to log-blocks
- commit(void)
  - Called only from end_op()
  - write_log()
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again

pipes

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;
 // number of bytes read
  uint nwrite;
 // number of bytes written
  int readopen;
  // read fd is still open
  int writeopen;
 // write fd is still open
};
```

- functions
  - pipealloc
  - pipeclose
  - piperead
  - pipewrite
-

pipes

- pipealloc
  - allocate two struct file
  - allocate pipe itself using kalloc (it’s a big structure with array)
  - init lock
  - initialize both struct file as 2 ends (r/w)
- pipewrite
  - wait if pipe full
  - write to pipe
  - wakeup processes waiting to read
- piperead
  - wait if no data
  - read from pipe
  - wakeup processes waiting to write
- Good producer consumer code !

Further to reading system call code now

- **Now we are ready to read the code of system calls on file system**
  - **sys_open, sys_write, sys_read , etc.**
- **Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.**
- **Also think of locks that need to be held.**