

15-213
"The course that gives CMU its Zip!"

Files Oct. 28, 2008

Topics

- Mapping file offsets to disk blocks
- File system buffering and you
- The directory hierarchy

lecture-18.ppt

Announcements

Exam Thursday

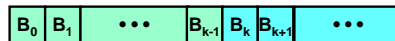
- style like exam #1: in class, open book/notes, no electronics
- class website has details and old exams

2

15-213, F'08

Reminder: Unix I/O

- **Key Features**
 - Elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.
 - Important idea: All input and output is handled in a consistent and uniform way.
- **Basic Unix I/O operations (system calls):**
 - **Opening and closing files**
 - `open()` and `close()`
 - **Reading and writing a file**
 - `read()` and `write()`
 - **Changing the *current file position* (seek)**
 - indicates next offset into file to read or write
 - `Lseek()`



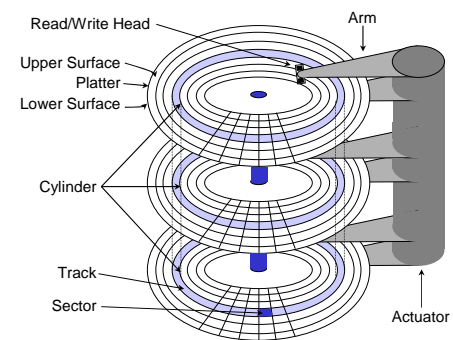
Current File Position = k

3

From lecture-13.ppt

15-213, F'08

Reminder: Disk Structure



4

From lecture-17.ppt

15-213, F'08

Reminder: Disk storage as array of blocks



OS's view of storage device
(as exposed by SCSI or IDE/ATA protocols)

- **Common “logical block” size: 512 bytes**
- **Number of blocks: device capacity / block size**
- **Common OS-to-storage requests defined by few fields**
 - R/W, block #, # of blocks, memory source/dest

5 From lecture-17.ppt

15-213, F'08

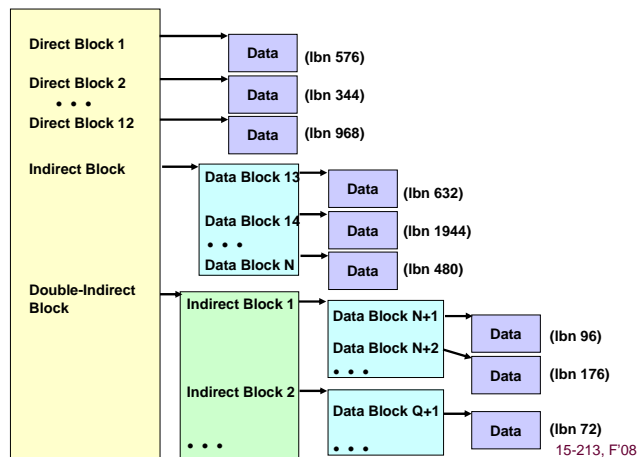
Mapping file offsets to disk LBNs

- **Issue in question**
 - need to keep track of which LBNs hold which file data
- **Most trivial mapping: just remember start location**
 - then keep entire file in contiguous LBNs
 - what happens when it grows?
 - alternately, include a “next pointer” in each “block”
 - how does one find location of a particular offset?
- **Most common approach: block lists**
 - an array with one LBN per block in the file
 - **Note:** file block size can exceed one logical (disk) block
 - so, groups of logical blocks get treated as a unit by file system
 - e.g., 8KB = 16 disk blocks (of 512 bytes each)

6

15-213, F'08

A common approach to recording a block list



7

15-213, F'08

Other per-file information must also be stored somewhere

- **Examples**
 - length of file
 - owner
 - access permissions
 - last modification time
 - ...

8

15-213, F'08

Reminder: File Metadata

- **Metadata** is data about data, in this case file data
- **Per-file metadata maintained by kernel**
 - accessed by users with the `stat` and `fstat` functions

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t      st_dev;      /* device */
    ino_t      st_ino;     /* inode */
    mode_t     st_mode;    /* protection and file type */
    nlink_t    st_nlink;   /* number of hard links */
    uid_t      st_uid;     /* user ID of owner */
    gid_t      st_gid;     /* group ID of owner */
    dev_t      st_rdev;    /* device type (if inode device) */
    off_t      st_size;    /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;   /* time of last access */
    time_t     st_mtime;   /* time of last modification */
    time_t     st_ctime;   /* time of last change */
};
```

From lecture-13.ppt

Other per-file information must also be stored somewhere

- **Examples**
 - length of file
 - owner
 - access permissions
 - last modification time
 - ...
- **Usually kept together with the block list**
 - In a structure called an **“inode”**

10

15-213, F'08

File block allocation

- **Two issues**
 - Keep track of which space is available
 - When a new block is needed, pick one of the free ones
- **Malloc-like solution – free list**
 - maintain a linked list of free blocks
 - using space in unused blocks to store the pointers
 - grab block from this list when a new block is needed
 - usually, the list is used as a stack
 - while simple, this approach rarely yields good performance
 - why?

11

15-213, F'08

File block allocation (cont.)

- **Most common approach – a bitmap**
 - Use a large array of bits, with one per allocatable unit
 - one value says “free” and the other says “in use”
 - Scan the array for a “free” setting, when we need a block
 - note: we don't have to just take first “free” block in array
 - we can look in particular regions or for particular patterns
- **In choosing an allocation, try to provide locality**
 - e.g., second block should be right after first
 - e.g., first block should be near inode

12

15-213, F'08

Reminder: Reading Files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
 - Return type `ssize_t` is signed integer
 - `nbytes < 0` indicates that an error occurred
 - short counts** (`nbytes < sizeof(buf)`) are possible and are not errors!

13

15-213, F'08

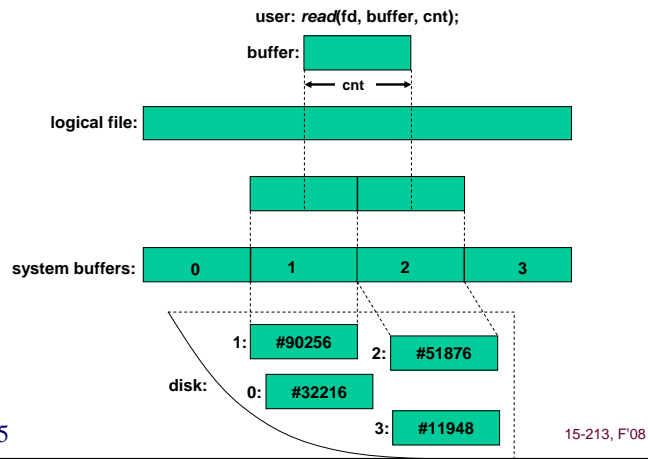
Managing file data in-kernel: buffers

- Staging area between disk and processes

14

15-213, F'08

Block-based file buffer management



15

15-213, F'08

Note: large I/Os are more efficient

- Recall disk performance is location dependent
 - milliseconds to position read/write head
 - microseconds to read next sector (usually = next LBN)
- Small `read()`s/`write()`s sometimes perform very poorly
 - Process 1 `read()`s 4KB from file #1 and waits for disk I/O
 - Process 2 `read()`s 4KB from file #2 and waits for disk I/O
 - Process 1 continues and `read()`s next 4KB from file #1
 - Process 2 continues and `read()`s next 4KB from file #2
 - ...
 - Result: random-like performance instead of sequential
 - bandwidth achieved would double with 8KB reads

16

15-213, F'08

Naturally, OS keeps a buffer cache

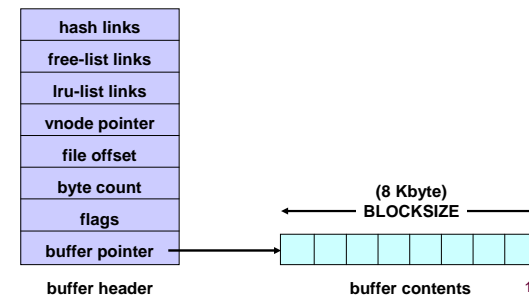
- **Disk I/O costs milliseconds**
 - as compared to microseconds for in-memory access
 - so, cache in-kernel buffers from previous read(s)
- **Each non-free buffer often kept on a number of lists**
 - overflow list associated with hash index
 - so that it can be found during read()
 - Least-Recently-Used list (or other importance tracking lists)
 - so that good choices can be made for replacement
 - vnode list
 - so that all buffers associated with a file can be found quickly
 - dirty block list
 - so that dirty buffers can be propagated to disk, when desired

17

15-213, F'08

Managing file data in the kernel: buffers

- **Staging area between disk and processes**
- **Two parts of each “buffer”**
 - header describing controls and buffer containing data



18

15-213, F'08

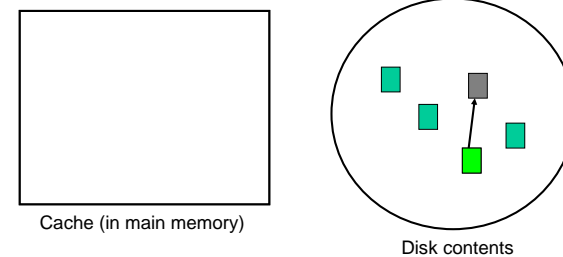
Write-back caching of file data

- **We talked about buffered Unix I/O**
 - wherein your printf(s) might not show up right away
- **This is different, but similar**
 - that was in your application (library); this is in-kernel
- **Most file systems use write-back caching**
 - buffers in memory are updated on write()
 - so, contents handed off
 - will be sent to disk at some later point
 - e.g., “30 second sync”
 - or, when OS runs low on memory space
 - if system crashes before the disk writes...
 - the file updates disappear

19

15-213, F'08

Volatile main memory and caching



20

15-213, F'08

You can force the disk writes

- The `fsync()` operation
 - directs file system to write the specified file to disk
 - includes everything associated with that file
 - directory entries, inode/attributes, indirect blocks, and data

21

15-213, F'08

Reminder: Opening Files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd; /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

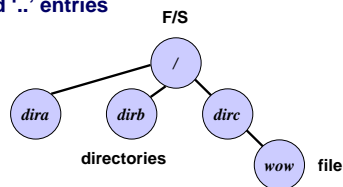
- Returns a small identifying integer *file descriptor*
 - `fd == -1` indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associated with a terminal:
 - 0: standard input
 - 1: standard output
 - 2: standard error

22

15-213, F'08

Common approach to naming: directory hierarchy

- Hierarchies are a good way to deal with complexity
 - ... and data organization is a complex problem
- It works pretty well for moderate-sized data sets
 - easy to identify coarse breakdowns
 - whenever gets too big, split it and refine namespace
- Traversing the directory hierarchy
 - the `.` and `..` entries



23

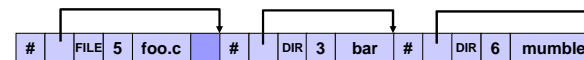
15-213, F'08

What's in a directory

- Directories to translate file names to inode IDs
 - just a special file with an array of formatted entries

4 bytes	2 bytes	2 bytes	variable length	
Inode number	Record length	Length of name	File Name (max. 255 characters)	N U L L

- often, sets of entries organized in sector-sized chunks

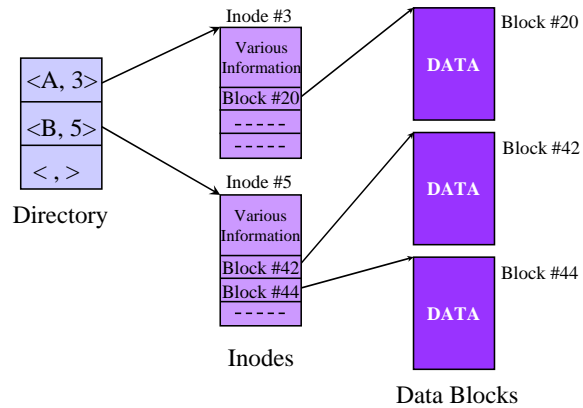


A directory block with three entries

24

15-213, F'08

A directory and two files



25

15-213, F'08

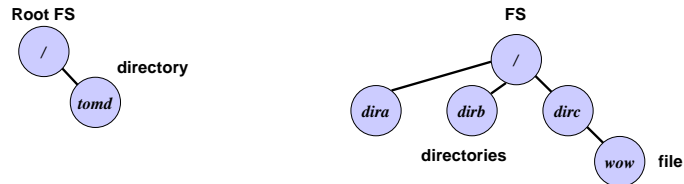
Managing namespace: mount/unmount

- One can have many FSs on many devices
 - ... but only one namespace
- So, one must combine the FSs into one namespace
 - starts with a "root file system"
 - the one that has to be there when the system boots
 - "mount" operation attaches one FS into the namespace
 - at a specific point in the overall namespace
 - "unmount" detaches a previously-attached file system

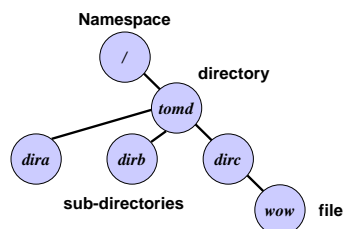
26

15-213, F'08

VIEW BEFORE MOUNTING



VIEW AFTER MOUNTING



27

15-213, F'08

Finally: walking thru an `open()`

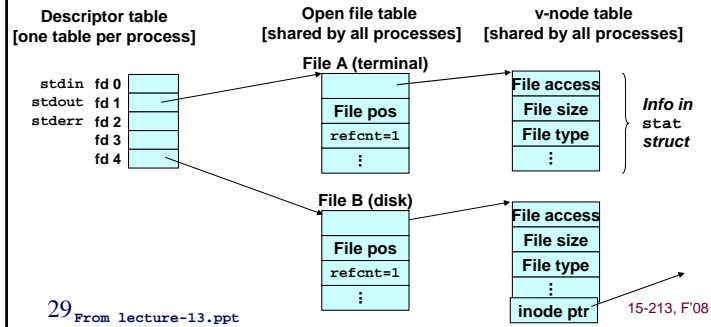
- `int fd = open("/foo/bar", RO);`
- Steps:
 - translate file name to inode identifier
 - lookup "foo" in root directory
 - read directory "foo" contents
 - lookup "bar" in directory "foo"
 - use directory lookup cache first for each lookup step
 - create a vnode structure for inode
 - lookup inode in inode cache; fetch from disk if necessary
 - initialize vnode structure appropriately
 - create open file structure
 - initialize, pointing to new vnode
 - fill in fd table entry
 - pick unused entry in table; have it point to new open file structure
- return corresponding index into fd table

28

15-213, F'08

Reminder: How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



Finally: walking thru a *read()*

- `int retcode = read(fd, buffer, size);`
- Steps:
 - index into fd table to get open file object
 - call `vnode_op_read(vnode, offset, buffer, size)`
 - calls into specific file system with associated inode (part of vnode)
 - index into block list at `offset/blocksize` to find data's LBN
 - may involve reading indirect blocks
 - grab ownership of buffer containing corresponding data
 - check buffer cache first
 - read from disk if not there
 - Ask device driver to read it, which creates CDB and so forth
 - copy data from cache buffer to caller's buffer
 - repeat last three steps until *size* reached
 - return to application
 - update open file object's offset on the way

30

15-213, F'08