Linux系统分析

第8讲 文件系统

张雷
zhangl@nju.edu.cn
南京大学计算机科学与技术系

主要内容

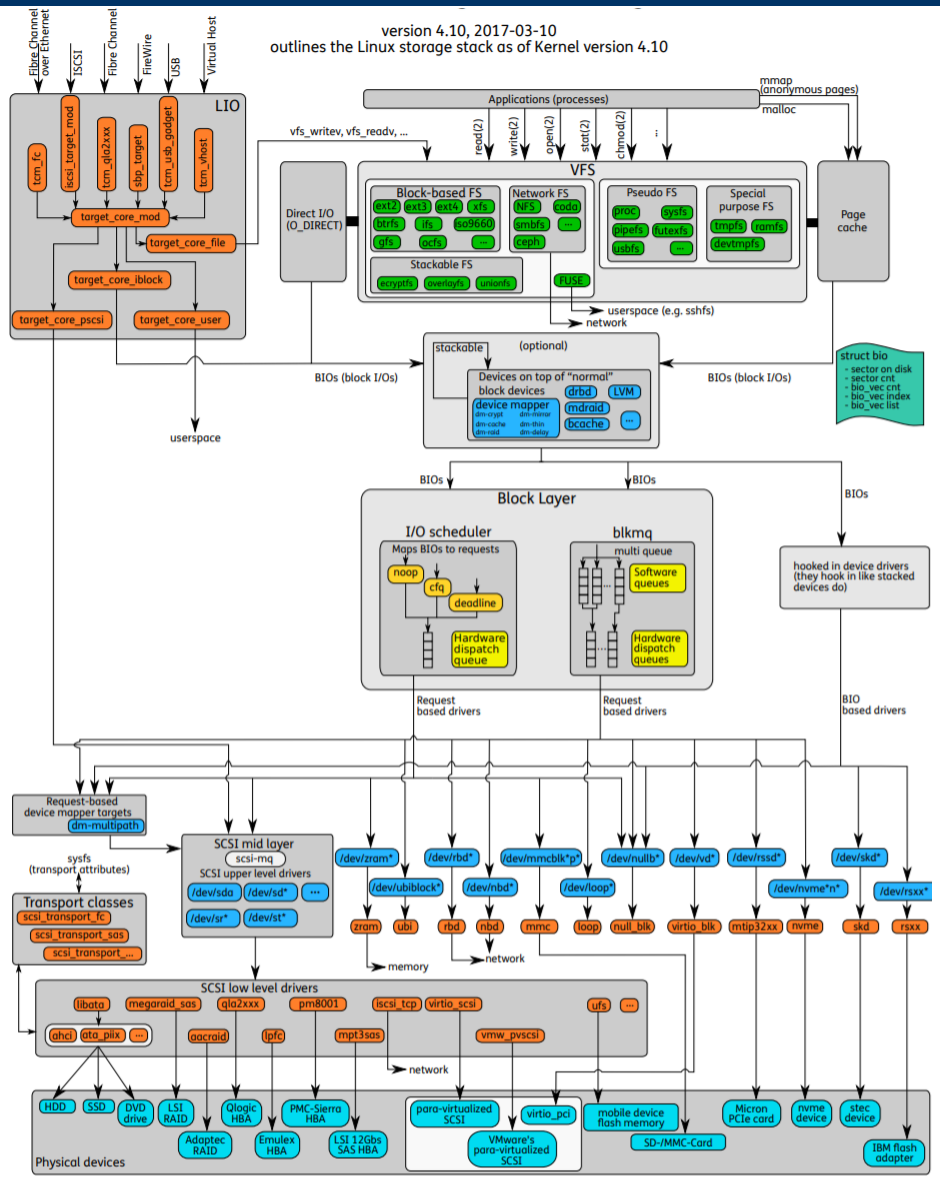
- 文件系统
  - 文件与目录
  - /proc文件系统
  - 文件系统接口及设计
- 如何实现文件系统
  - Unix文件系统
  - Fast File System
  - Log-structured File System

虚拟化

- 虚拟化CPU：进程
- 虚拟化内存：地址空间
- 虚拟化存储：文件与目录

- File systems: traditionally hardest part of OS
  - More papers on FSes than any other single topic

Linux Storage Stack



Why disks are different

❖ Disk vs. Memory

	Disk	MLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	> 1 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	> 1 GB/s
Cost	$0.03/GB	$0.35/GB	$6/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*: Flash write performance degrades over time

Why disks are different

- Disk reads/writes in terms of sectors, not bytes
  - Read/write single sector or adjacent groups
- How to write a single byte? "Read-modify-write"
  - Read in sector containing the byte
  - Modify that byte
  - Write entire sector back to disk
  - Key: if cached, don't need to read in

- Sector = unit of atomicity.
  - Sector write done completely, even if crash in middle (disk saves up enough momentum to complete)

Some Useful Trends

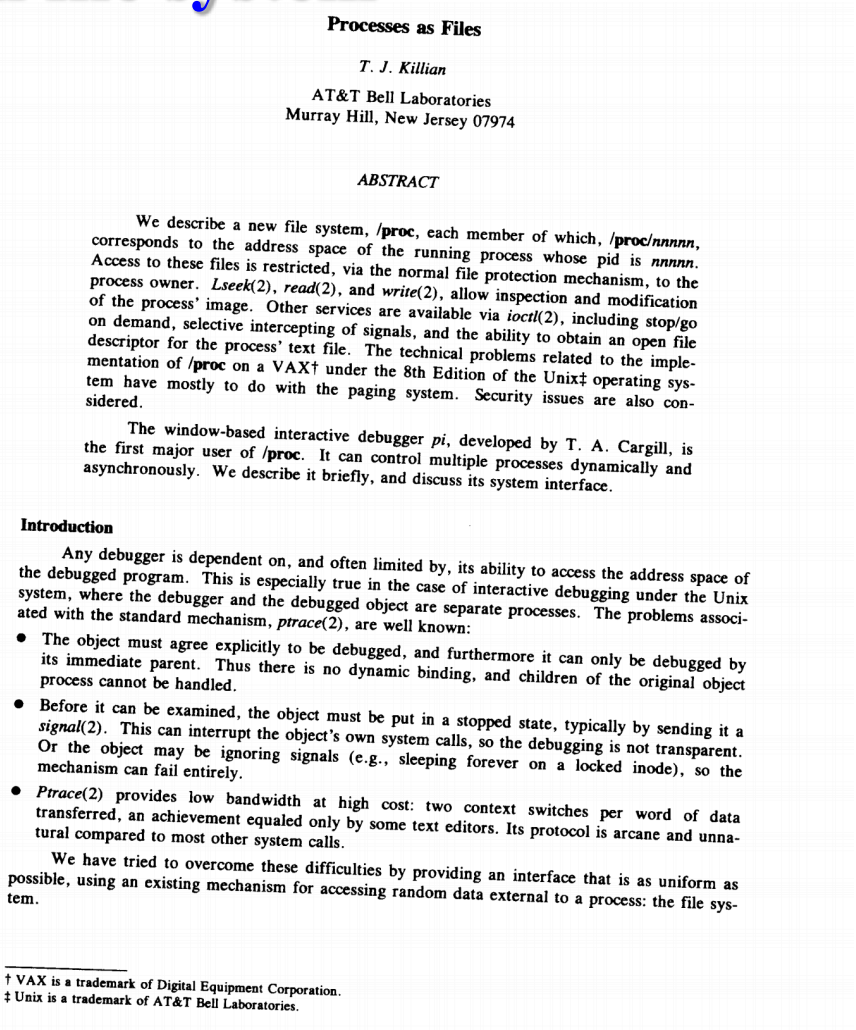
- Disk bandwidth and cost/bit improving exponentially
  - Similar to CPU speed, memory size, etc.
- Seek time and rotational delay improving very slowly
  - Why? require moving physical object (disk arm)
- Disk accesses a huge system bottleneck & getting worse
  - Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
  - Trade bandwidth for latency if you can get lots of related stuff.
- Desktop memory size increasing faster than typical workloads
  - More and more of workload fits in file cache
  - Disk traffic changes: mostly writes and new data •
- Memory and CPU resources increasing
  - Use memory and CPU to make better decisions
  - Complex prefetching to support more IO patterns
  - Delay data placement decisions reduce random IO

文件

- File: named bytes on disk
  - data with some properties
  - contents, size, owner, last read/write time, protection, etc
- A file can also have a type
  - Understood by the file system
    - ✓ Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or runtime libraries
    - ✓ Executable, dll, source, object, text, etc
- A file's type can be encoded in its name or contents
  - Windows encodes type in name
    - ✓ .com, .exe, .bat, .dll, .jpg, etc.
  - Unix encodes type in contents
    - ✓ Magic numbers, initial characters (#! for shell scripts)

文件

- Everything named through file system
  - Files, devices, pipes,
- Process as Files
  - File system /proc
  - Tom Killian, USENIX1984



Processes as Files

T. J. Killian

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe a new file system, **/proc**, each member of which, **/proc/***nnnnn*, corresponds to the address space of the running process whose pid is *nnnnn*. Access to these files is restricted, via the normal file protection mechanism, to the process owner. *Lseek*(2), *read*(2), and *write*(2), allow inspection and modification of the process' image. Other services are available via *ioctl*(2), including stop/go on demand, selective intercepting of signals, and the ability to obtain an open file descriptor for the process' text file. The technical problems related to the implementation of **/proc** on a VAX† under the 8th Edition of the Unix‡ operating system have mostly to do with the paging system. Security issues are also considered.

The window-based interactive debugger *pi*, developed by T. A. Cargill, is the first major user of **/proc**. It can control multiple processes dynamically and asynchronously. We describe it briefly, and discuss its system interface.

Introduction

Any debugger is dependent on, and often limited by, its ability to access the address space of the debugged program. This is especially true in the case of interactive debugging under the Unix system, where the debugger and the debugged object are separate processes. The problems associated with the standard mechanism, *ptrace*(2), are well known:

- The object must agree explicitly to be debugged, and furthermore it can only be debugged by its immediate parent. Thus there is no dynamic binding, and children of the original object process cannot be handled.
- Before it can be examined, the object must be put in a stopped state, typically by sending it a *signal*(2). This can interrupt the object's own system calls, so the debugging is not transparent. Or the object may be ignoring signals (e.g., sleeping forever on a locked inode), so the mechanism can fail entirely.
- *Ptrace*(2) provides low bandwidth at high cost: two context switches per word of data transferred, an achievement equaled only by some text editors. Its protocol is arcane and unnatural compared to most other system calls.

We have tried to overcome these difficulties by providing an interface that is as uniform as possible, using an existing mechanism for accessing random data external to a process: the file system.

† VAX is a trademark of Digital Equipment Corporation.
‡ Unix is a trademark of AT&T Bell Laboratories.

/proc文件系统

❖ Obtaining Information About a Process: /proc/*PID*

```
$ cat /proc/1/status
Name:   init                                    Name of command run by this process
State:  S (sleeping)                            State of this process
Tgid:   1                                       Thread group ID (traditional PID, getpid())
Pid:    1                                       Actually, thread ID (gettid())
PPid:   0                                       Parent process ID
TracerPid:      0                               PID of tracing process (0 if not traced)
Uid:    0       0       0       0               Real, effective, saved set, and FS UIDs
Gid:    0       0       0       0               Real, effective, saved set, and FS GIDs
FDSize: 256                                     # of file descriptor slots currently allocated
Groups:                                         Supplementary group IDs
VmPeak:       852 kB                            Peak virtual memory size
VmSize:       724 kB                            Current virtual memory size
VmLck:          0 kB                            Locked memory
VmHWM:        288 kB                            Peak resident set size
VmRSS:        288 kB                            Current resident set size
VmData:       148 kB                            Data segment size
VmStk:         88 kB                            Stack size
VmExe:        484 kB                            Text (executable code) size
VmLib:          0 kB                            Shared library code size
VmPTE:         12 kB                            Size of page table (since 2.6.10)
Threads:        1                               # of threads in this thread's thread group
SigQ:   0/3067                                  Current/max. queued signals (since 2.6.12)
SigPnd: 0000000000000000                        Signals pending for thread
ShdPnd: 0000000000000000                        Signals pending for process (since 2.6)
SigBlk: 0000000000000000                        Blocked signals
SigIgn: fffffffe5770d8fc                        Ignored signals
SigCgt: 00000000280b2603                        Caught signals
CapInh: 0000000000000000                        Inheritable capabilities
CapPrm: 00000000ffffffff                        Permitted capabilities
```

/proc文件系统

- Selected files in each /proc/*PID* directory

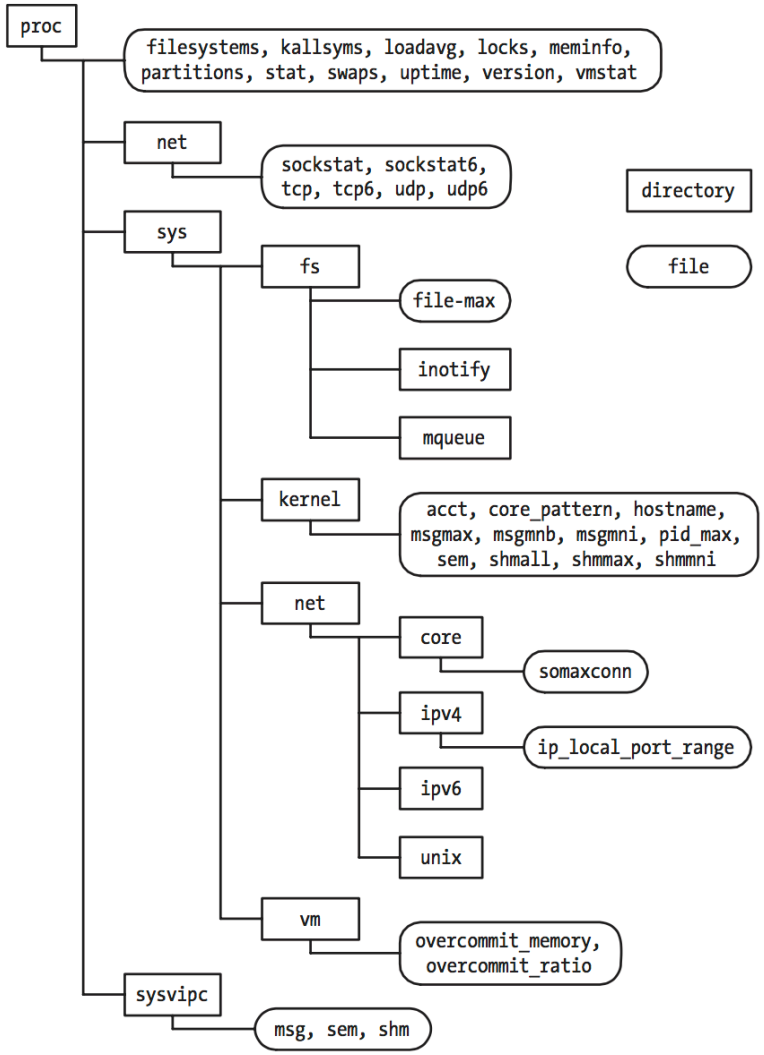
File	Description (process attribute)
`cmdline`	Command-line arguments delimited by `\0`
`cwd`	Symbolic link to current working directory
`environ`	Environment list *NAME=value* pairs, delimited by `\0`
`exe`	Symbolic link to file being executed
`fd`	Directory containing symbolic links to files opened by this process
`maps`	Memory mappings
`mem`	Process virtual memory (must *lseek()* to valid offset before I/O)
`mounts`	Mount points for this process
`root`	Symbolic link to root directory
`status`	Various information (e.g., process IDs, credentials, memory usage, signals)
`task`	Contains one subdirectory for each thread in process (Linux 2.6)

- /proc/1968/fd/1 is a symbolic link to the standard output of process 1968.

/proc文件系统

❖ echo 100000 > /proc/sys/kernel/pid_max

Directory	Information exposed by files in this directory
/proc	Various system information
/proc/net	Status information about networking and sockets
/proc/sys/fs	Settings related to file systems
/proc/sys/kernel	Various general kernel settings
/proc/sys/net	Networking and sockets settings
/proc/sys/vm	Memory-management settings
/proc/sysvipc	Information about System V IPC objects



文件

- Three types of names
  - Unique id: inode numbers, see inodes via “ls –i”
  - Path
  - File descriptor

- What does “i” stand for?
  - “In truth, I don‘t know either. It was just a term that we started to use. ‘Index’ is my best guess, because of the slightly unusual file system structure that stored the access information of files as a flat array on the disk…”
    ~ Dennis Ritchie

- 目录：<filename, inode>对

文件系统接口

❖创建文件

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

Returns file descriptor on success, or −1 on error

❖ **int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);**

创建文件

Table 4-3: Values for the *flags* argument of *open()*

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if *pathname* is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)	
O_NOCTTY	Don't let *pathname* become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

读写文件

- #echo hello >foo
- #cat foo

  hello
- strace 工具

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
prompt>
```

读写文件

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or −1 on error

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or −1 on error

```
#include <unistd.h>

int close(int fd);
```

Returns 0 on success, or −1 on error

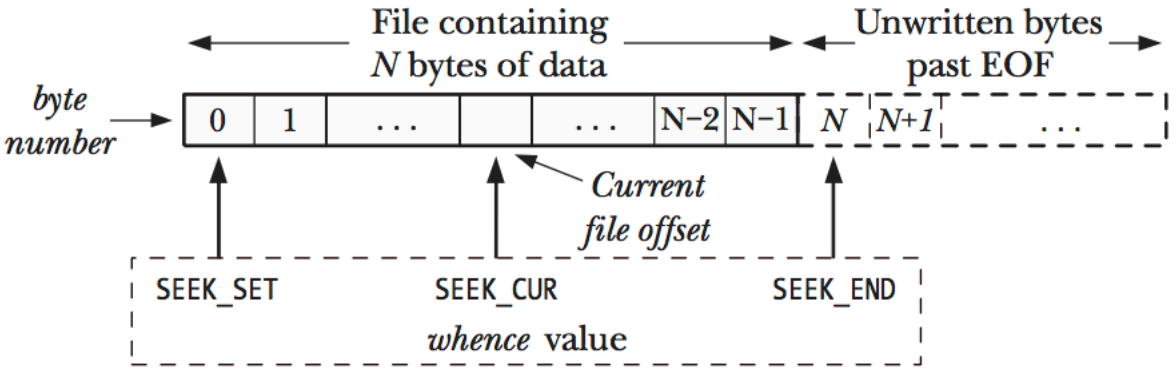
读写文件

- 随机，非顺序读写

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
                                    Returns new file offset if successful, or −1 on error
```

- *whence*: SEEK_SET, SEEK_CUR, SEEK_END



读写文件

❖立即写 fsync

```
#include <unistd.h>

int fsync(int fd);
```

Returns 0 on success, or −1 on error

❖文件改名 mv foo bar

- strace显示调用rename(char *old, char *new)

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

文件IO

❖删除文件 rm

```
prompt> strace rm foo
...
unlink("foo")                           = 0
...
```

❖创建目录

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                      = 0
...
prompt>
```

文件IO

❖硬连接ln

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

文件IO

❖软连接

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

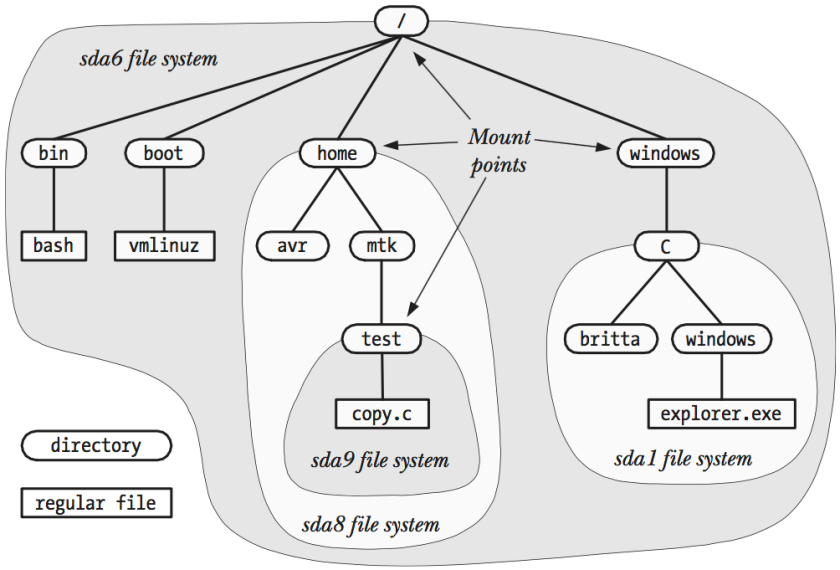
❖file2 4 bytes

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 -> alongerfilename
```

创建挂载文件系统

- mkfs输入：设备/dev/sda1，文件系统类型 ext3
- mount *device directory*

```
$ mount
/dev/sda6 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
/dev/sda8 on /home type ext3 (rw,acl,user_xattr)
/dev/sda1 on /windows/C type vfat (rw,noexec,nosuid,nodev)
/dev/sda9 on /home/mtk/test type reiserfs (rw)
```



Rethinking file io

```
int fd = open(char *path, int flag, mode_t mode)
read(int fd, void *buf, size_t nbyte)
write(int fd, void *buf, size_t nbyte)
close(int fd)
```

❖ Why not use inode

```
read(int inode, void *buf, size_t nbyte)
write(int inode, void *buf, size_t nbyte)
seek(int inode, off_t offset)
```

❖ Why not use path

```
pread(char *path, void *buf,
      off_t offset, size_t nbyte)

pwrite(char *path, void *buf,
       off_t offset, size_t nbyte)
```

Why not inode

- names hard to remember
- no organization or meaning to inode numbers
- semantics of offset across multiple processes?

- File system still interacts with inode numbers
- Store file-to-inode mappings for each directory

Why not path

- Let's say you want to open "/one/two/three"
  - fd = open("/one/two/three", O_RDWR);
- What goes on inside the file system?
  - open directory "/" (well known, can always find)
  - search the directory for "one", get location of "one"
  - open directory "one", search for "two", get location of "two"
  - open directory "two", search for "three", get loc. of "three"
  - open file "three"
  - (of course, permissions are checked at each step)
- FS spends lots of time walking down directory paths
  - this is why open is separate from read/write (session state)
  - OS will cache prefix lookups to enhance performance
    - ✓ /a/b, /a/bb, /a/bbb all share the "/a" prefix

Rethinking file io

- Do expensive traversal once (open file)
- Store inode in descriptor object (kept in memory)
- Do reads/writes via descriptor, which tracks offset

- Each process: File-descriptor table contains pointers to open file descriptors
  - Integers used for file I/O are indexes into this table stdin: 0, stdout: 1, stderr: 2

Designing file structure

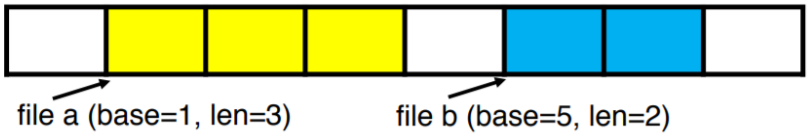
- [A+07] Most files are small.

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

- [A+07] Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch A Five-Year Study of File-System Metadata FAST '07, pages 31–45, February 2007, San Jose, CA

File structure

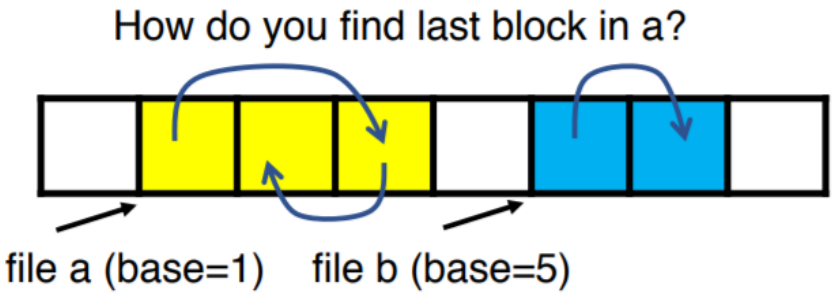
- **Contiguous Allocation**
  - Example: IBM OS/360
  - Inode contents: location and size



- **Pros?**
  - Simple, fast access, both sequential and random
- **Cons?**
  - External fragmentation

File structure

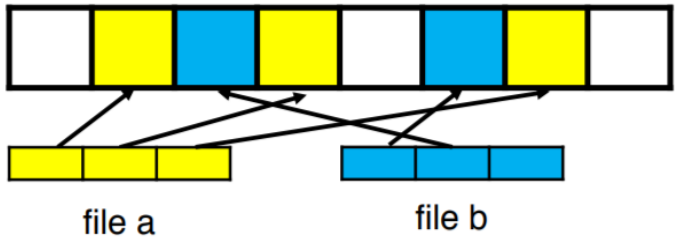
- **Linked Files**
  - Example: DOS FAT
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next one



- **Pros?**
  - Easy dynamic growth & sequential access, no fragmentation
- **Cons?**
  - Linked lists on disk a bad idea because of access times
  - Random very slow

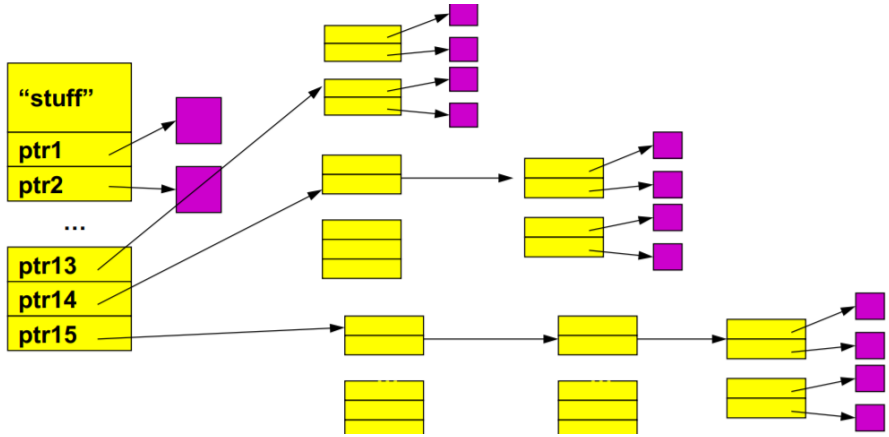
File structure

❖ Indexed Files

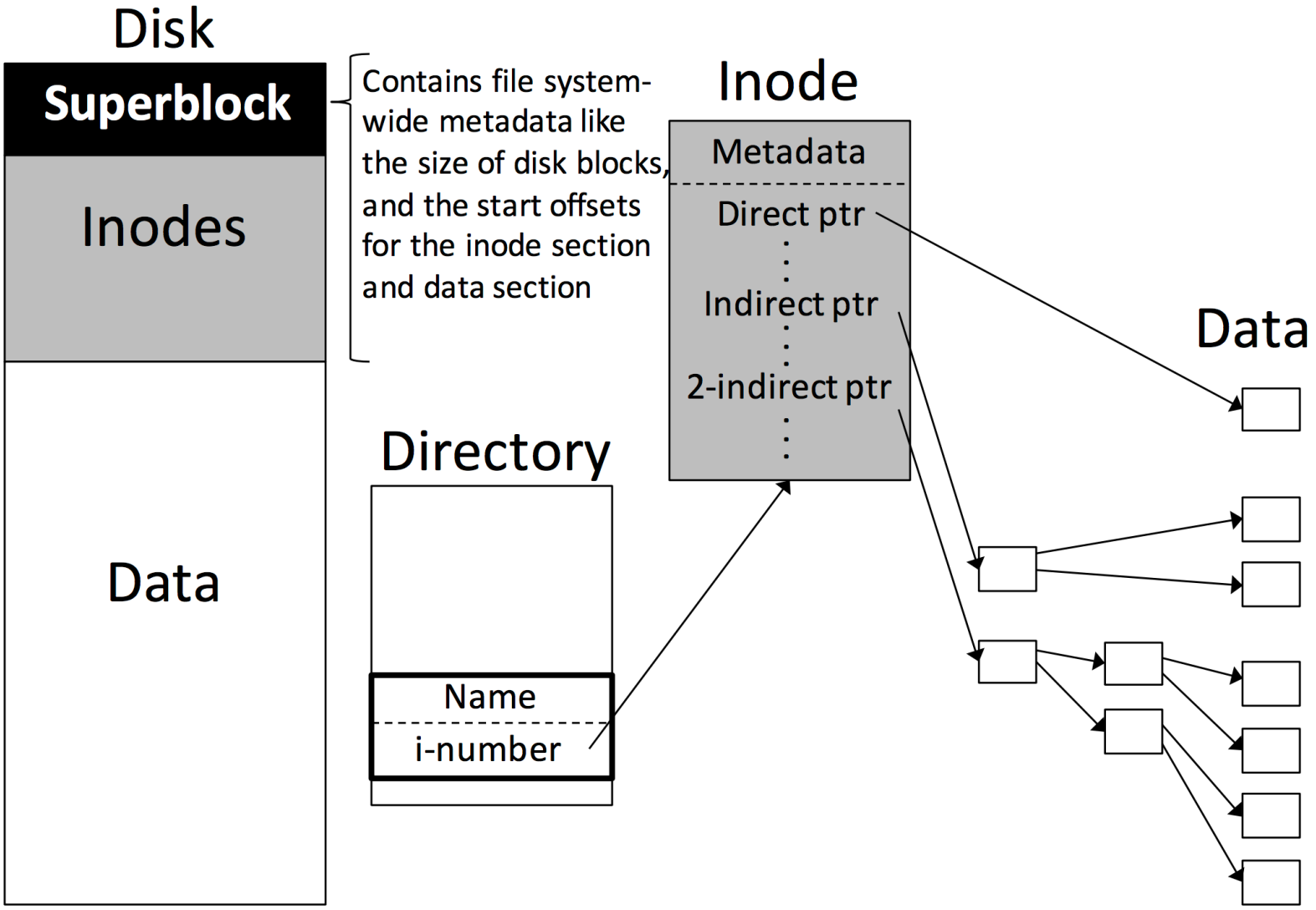


❖ inode = 15 block pointers + "stuff"

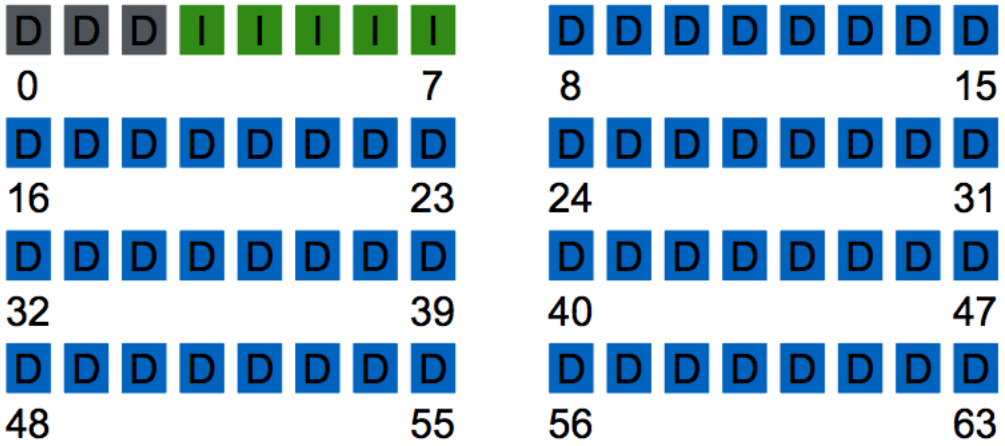
- first 12 are direct blocks: solve problem of first blocks access slow
- then single, double, and triple indirect block



The Original, Not-Fast Unix Filesystem



Inodes


D D D I I I I I
0 7
D D D D D D D D
8 15
D D D D D D D D
16 23
D D D D D D D D
24 31
D D D D D D D D
32 39
D D D D D D D D
40 47
D D D D D D D D
48 55
D D D D D D D D
56 63

open /foo/bar

- open("/foo/bar", O RDONLY)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			**read**			**read**				
				read			**read**			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

read /foo/bar

- assume opened

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
							read
				write			

write to /foo/bar

- assume file exists and has been opened

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		read write	read	read	read write	read	read			
				write			write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

nothing to do on disk!

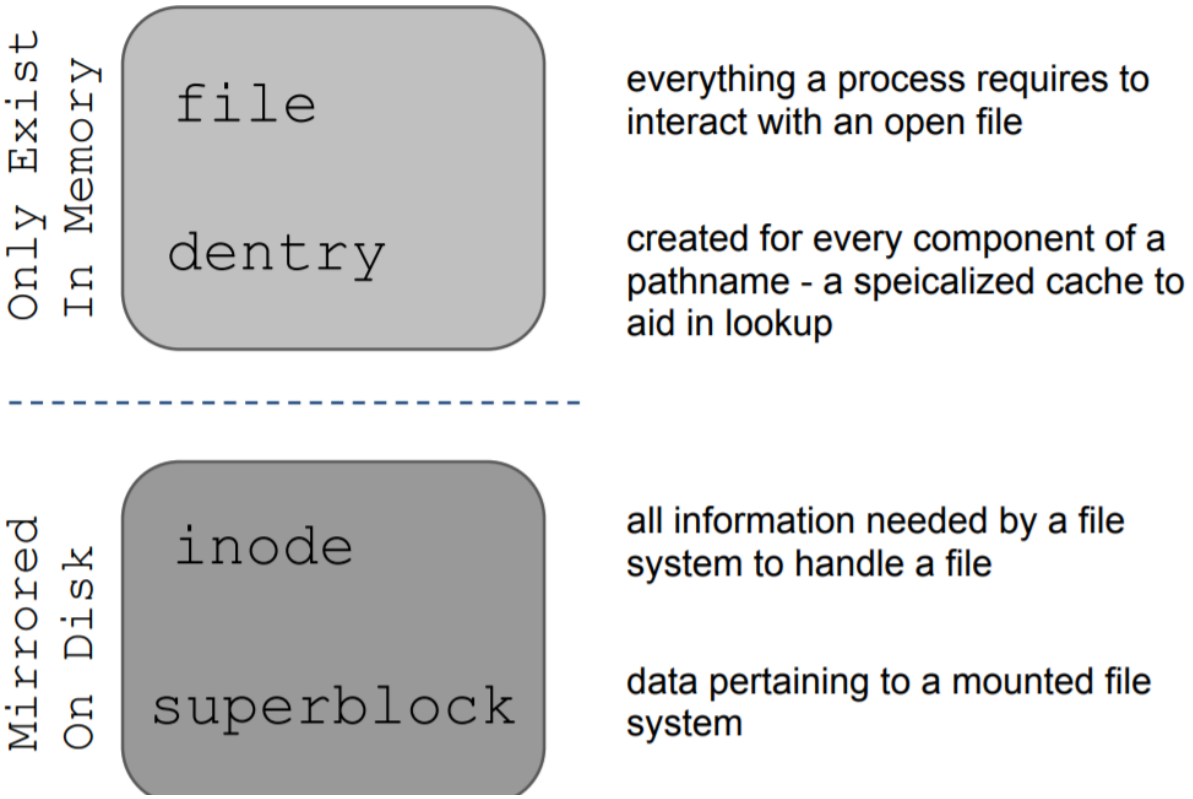
Efficiency

- How can we avoid this excessive I/O for basic ops?
- Cache for:
  - reads
  - write buffering

Write Buffering

- OSes typically do write back caching
  - Maintain a queue of uncommitted blocks
  - Periodically flush the queue to disk (30 second threshold)
  - If blocks changed many times in 30 secs, only need one I/O
  - If blocks deleted before 30 secs (e.g., /tmp), no I/Os needed
- Unreliable, but practical
  - On a crash, all writes within last 30 secs are lost
  - Modern OSes do this by default; too slow otherwise
  - System calls (Unix: fsync) enable apps to force data to disk

Core Data Structures



everything a process requires to interact with an open file

created for every component of a pathname - a speicalized cache to aid in lookup

all information needed by a file system to handle a file

data pertaining to a mounted file system

File Operations

```
struct file_operations {
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **);
};
```

adjust file position

read/write data

map to memory

sync to disk

Inode Operations

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        int (*permission) (struct inode *, int);
        int (*check_acl)(struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
        void (*truncate) (struct inode *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
        long (*fallocate)(struct inode *inode, int mode, loff_t offset,
                          loff_t len);
        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                      u64 len);
}
```

read from disk

manage links

check permissions

manage blocks

Super Operations

```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*dirty_inode) (struct inode *);
        int (*write_inode) (struct inode *, struct writeback_control *wbc);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        int (*freeze_fs) (struct super_block *);
        int (*unfreeze_fs) (struct super_block *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*statfs) (struct dentry *, struct kstatfs *);
        int (*show_options)(struct seq_file *, struct vfsmount *);
        int (*show_stats)(struct seq_file *, struct vfsmount *);
#ifdef CONFIG_QUOTA
        ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
        ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
#endif
        int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

create/destroy/commit inodes

synchronize file system state

summarize file system state

Dentry Operations

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, struct nameidata *);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
        char *(*d_dname)(struct dentry *, char *, int);
};
```

generally use generic hash function

hooks called before deleting/freeing

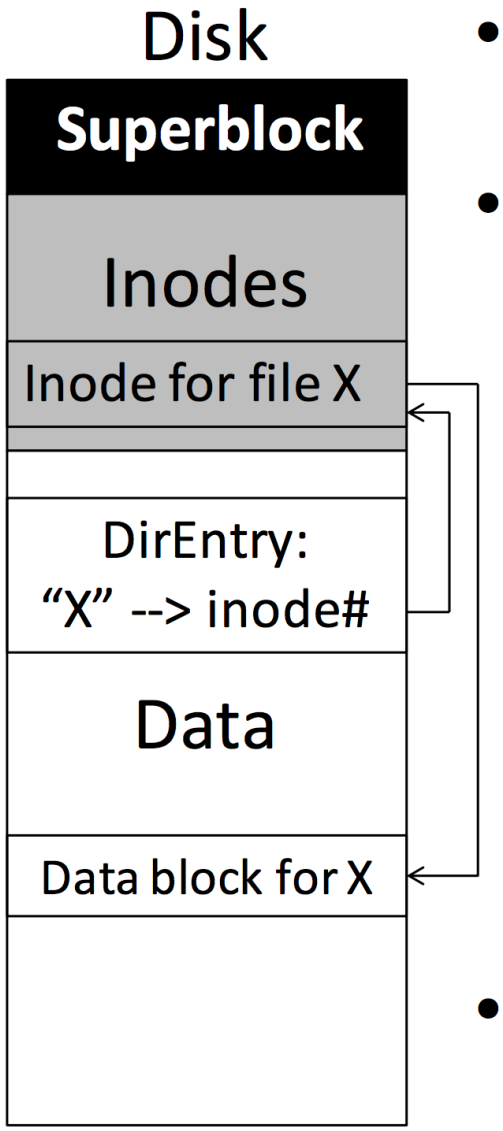
inode and dentry caches coupled

"**dcache**" is just a hashtable

File Systems Examples

- **BSD Fast File System (FFS)**
  - What were the problems with the original Unix FS?
  - How did FFS solve these problems?
- **Log-Structured File system (LFS)**
  - What was the motivation of LFS?
  - How did LFS work?

The Original, Not-Fast Unix Filesystem



- Design: Disk is treated like a linear array of bytes
- Problem: Data access incurs mechanical delays!
  - Accessing a file's inode and then a data block requires two seeks (or more if the block is indirectly-pointed-to)
  - Block allocation wasn't clever, so files in the same directory were often far apart
  - Block size was 512 bytes, increasing penalty for poor block allocation (more disk seeks!)
- Result: File system provided only 4% of the sequential disk bandwidth!

Why So Slow? 20Kb/sec

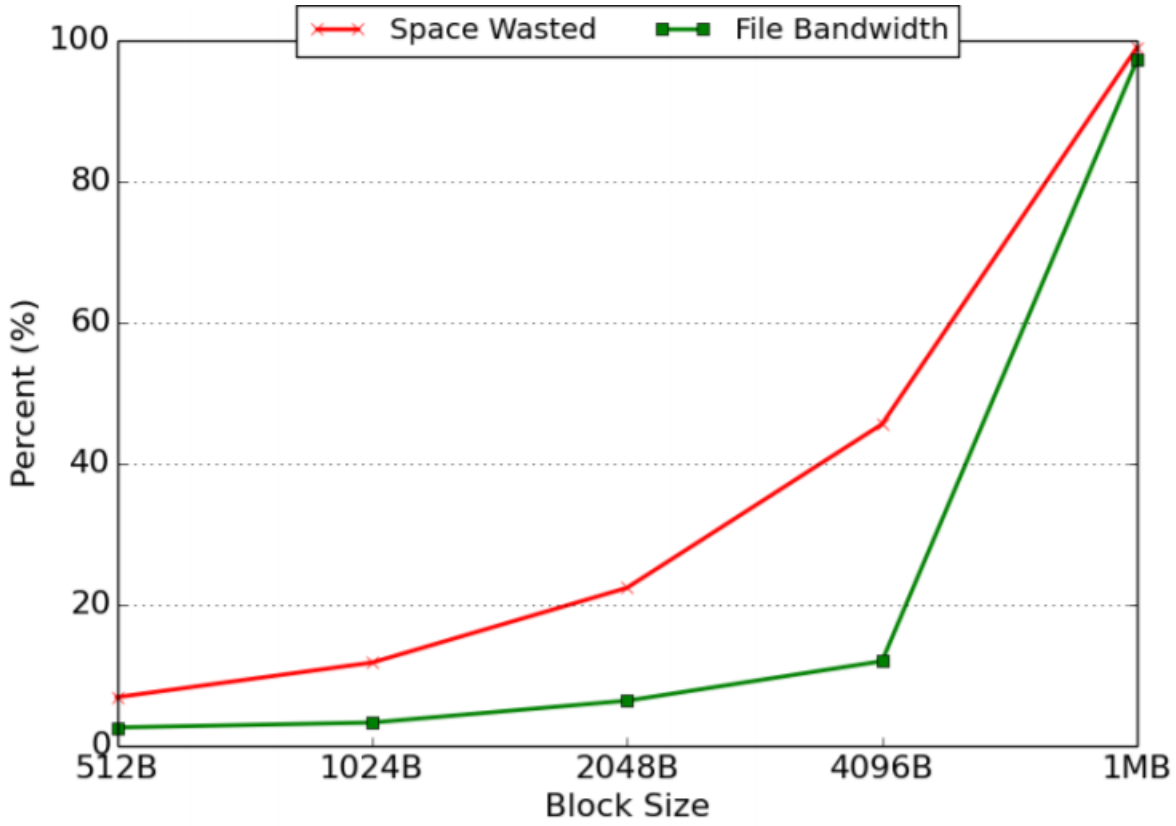
- Problem 1: blocks too small (512 bytes)
  - File index too large
  - Require more indirect blocks
  - Transfer rate low (get one block at time)
- Problem 2: unorganized freelist
  - Consecutive file blocks not close together
    - ✓ Pay seek cost for even sequential acces
  - Aging: becomes fragmented over time
- Problem 3: poor locality
  - inodes far from data blocks
  - inodes for directory not close together
    - ✓ poor enumeration performance: e.g., “ls”, “grep foo *.c”

FFS: The Fast File System

- **Designed by a Berkeley research group for the BSD UNIX**
  - **A classic file systems paper to read:** Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst.* 2, 3 (August 1984), 181-197.
- **Approach:**
  - **measure an state of the art systems**
  - **identify and understand the fundamental problems**
    - ✓ **The original FS treats disks like random-access memory!**
  - **get an idea and build a better systems**
- **Idea: design FS structures and allocation polices to be "disk aware"**

Problem : Blocks Too Small

Measurement:



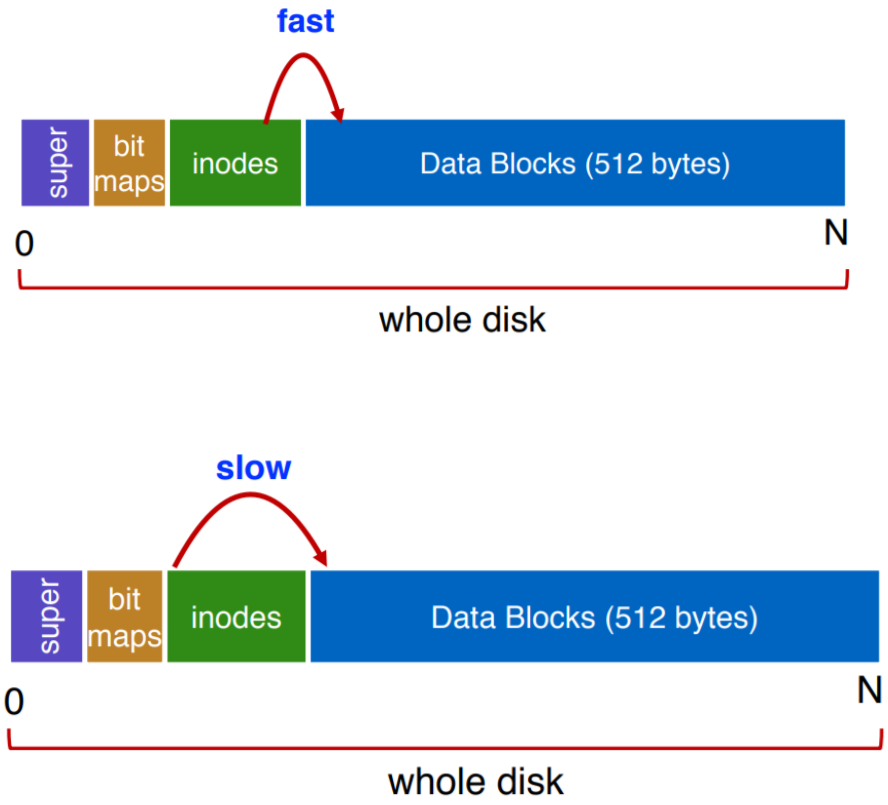
❖ **Increase block size from 512 bytes to 4096 bytes**

Problem : unorganized freelist

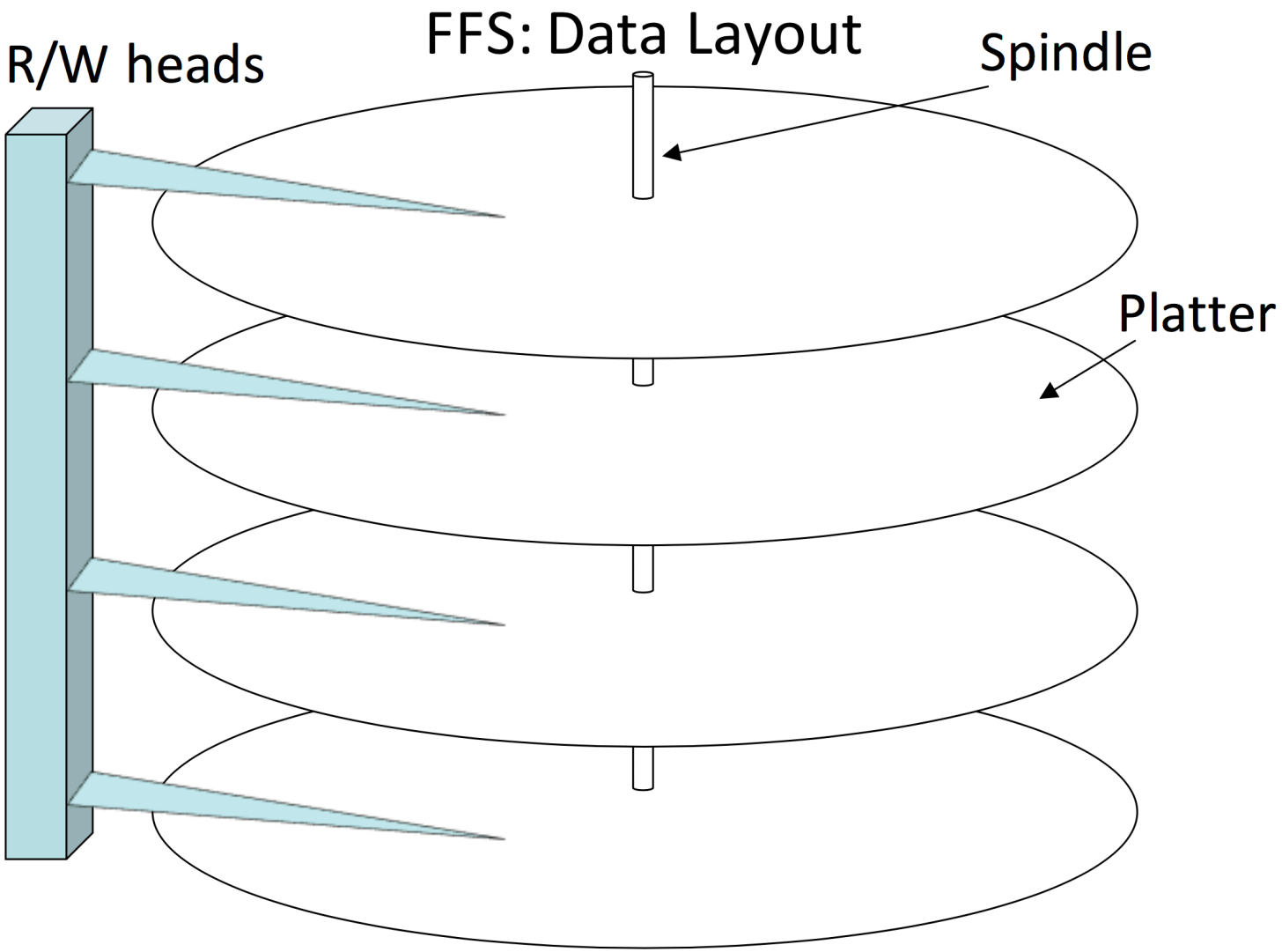
- Using a Bitmap

- Trade space for time (search time, file access time)

Problem: Poor Locality

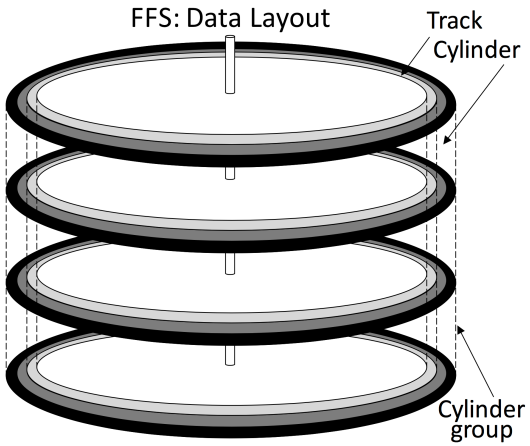


FFS: The Fast File System

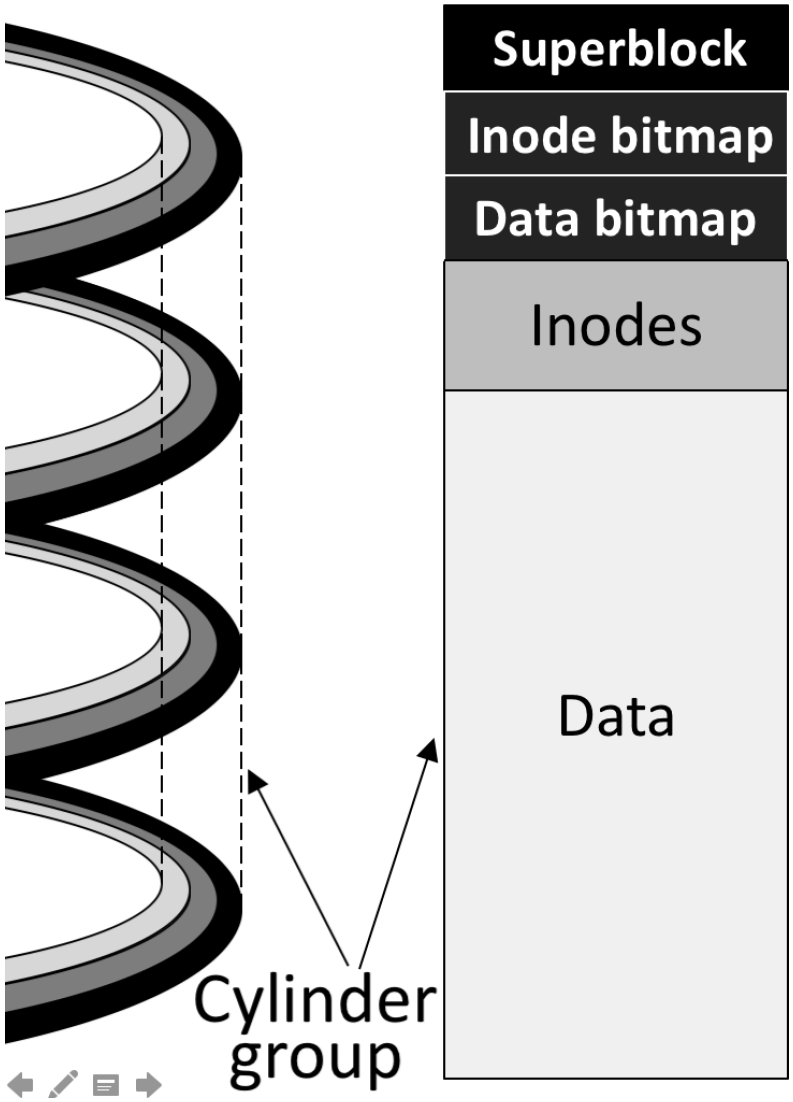


FFS

- Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
- Tries to put everything related in same cylinder group
- Tries to put everything not related in different group



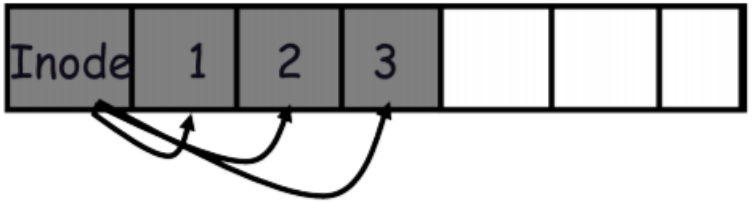
FFS Data Layout



- Directory allocation: Use a cylinder group with few allocated directories and many free inodes
- File allocation: Allocate file inodes in cylinder group of parent directory; allocate file data blocks in cylinder group of file inode
- Allocation policies driven by expectation of temporal locality
  - Files in the same directory will be accessed together (e.g., source code compilation, a browser's web cache)
  - Providing spatial locality for data with temporal locality decreases disk seeks!

FFS

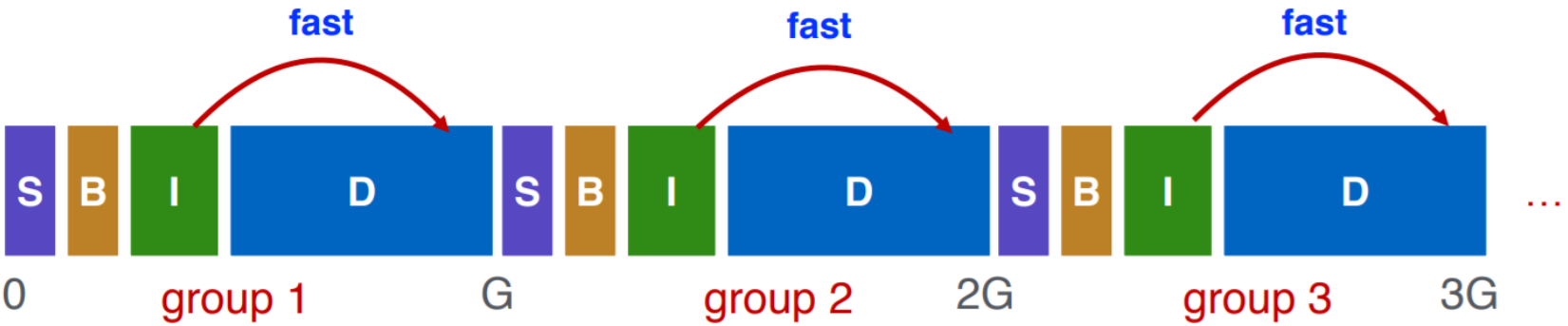
- Tries to put sequential blocks in adjacent
  - (Access one block, probably access next)
- Tries to keep inode in same cylinder as file data
  - (If you look at inode, most likely will look at data too)



- Tries to keep all inodes in a dir in same cylinder group
  - - Access one name, frequently access many, e.g., “ls -l”

What Does Disk Layout Look Like Now?

- Each cylinder group basically a mini-Unix file system
- Is it useful to have multiple super blocks?
  - Yes, if some (but not all) fail



FFS Results

- **Performance improvements:**
  - Able to get 20-40% of disk bandwidth for large files
  - 10-20x original Unix file system!
  - Stable over FS lifetime
  - Better small file performance (why?)
- **Other enhancements**
  - Long file names
  - Parameterization
  - Free space reserve (10%) that only admin can allocate blocks from

LFS: Log-structured File System

- **Motivation**
  - Faster CPUs: I/O becomes more and more of a bottleneck
  - More memory: file cache is effective for reads
  - Implication: writes compose most of disk traffic
- **Problems with previous FS**
  - Perform many small writes
    - ✓ Good performance on large, sequential writes, but many writes are still small, random
  - Synchronous operation to avoid data loss
  - Depends upon knowledge of disk geometry
- **An influential work designed by Mendel Rosenblum (VMWare co-founder) and John Ousterhout**

LFS Idea

- Insight: treat disk like a tape-drive
  - Best performance from disk for sequential access
- File system buffers writes in main memory until "enough" data
  - How much is enough?
  - Enough to get good sequential bandwidth from disk (MB)
  - Unit called a "segment"
- •Write buffered data to new segment on disk in a sequential log
  - Transfer all updates into a series of sequential writes
  - Do not overwrite old data on disk: old copies left behind
  - Write both data and metadata in one operation

Pros And Cons

- **Pros**
  - Always large sequential writes > good performance
  - No knowledge of disk geometry
    - ✓ Assume sequential better than random
- **Potential problems**
  - How do you find data to read?
  - What happens to metadata during write?
  - What happens when you fill up the disk?

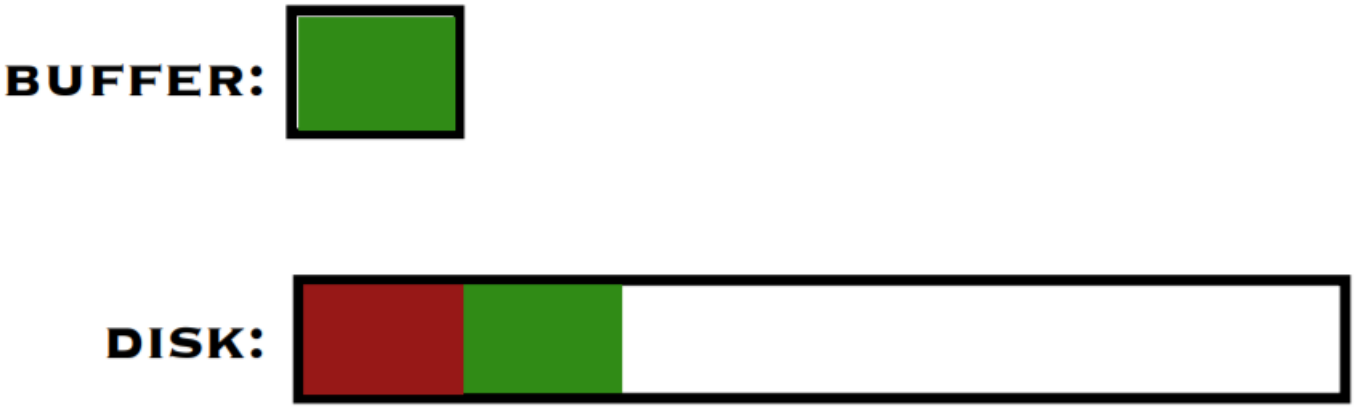
Read in LFS

- Same basic structures as Unix
  - Directories, inodes, indirect blocks, data blocks
  - Reading data block implies finding the file's inode
    - ✓ Unix FS: inodes kept in array
    - ✓ LFS: inodes spread around on disk
- Solution: inode map indicates where each inode is stored
  - Can keep cached copy in memory
  - inode map written to log with everything else
  - Periodically written to known checkpoint location on disk for crash recovery

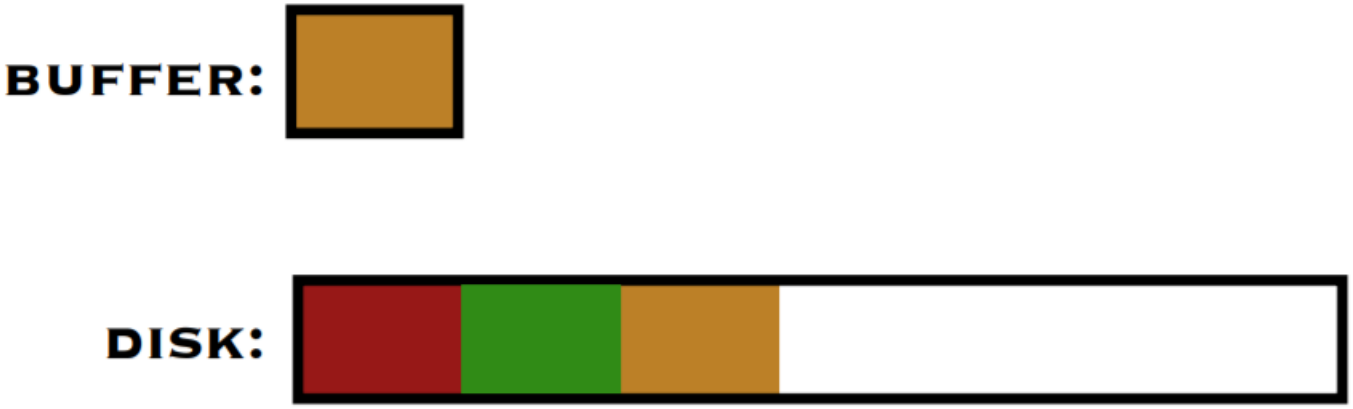
Write in LFS

- **Why do we buffer the write?**
  - Sequential write alone is not enough
  - Disk is constantly rotating!
  - Must issue a large number of contiguous writes

Write in LFS

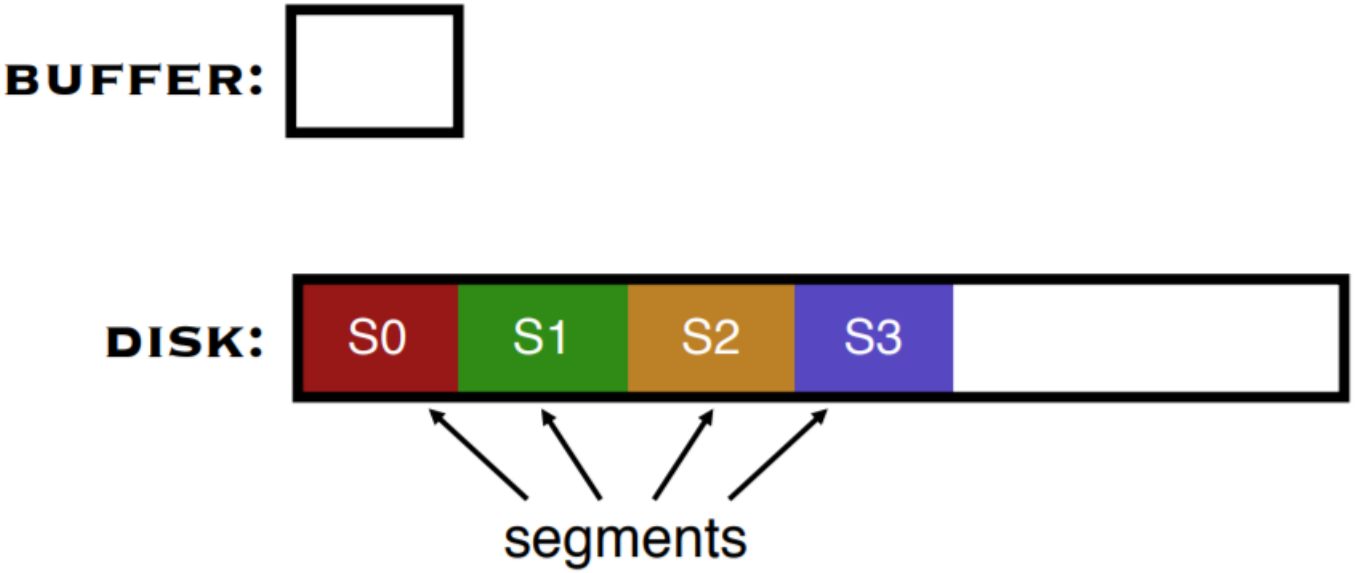


Write in LFS



Write in LFS

Write in LFS

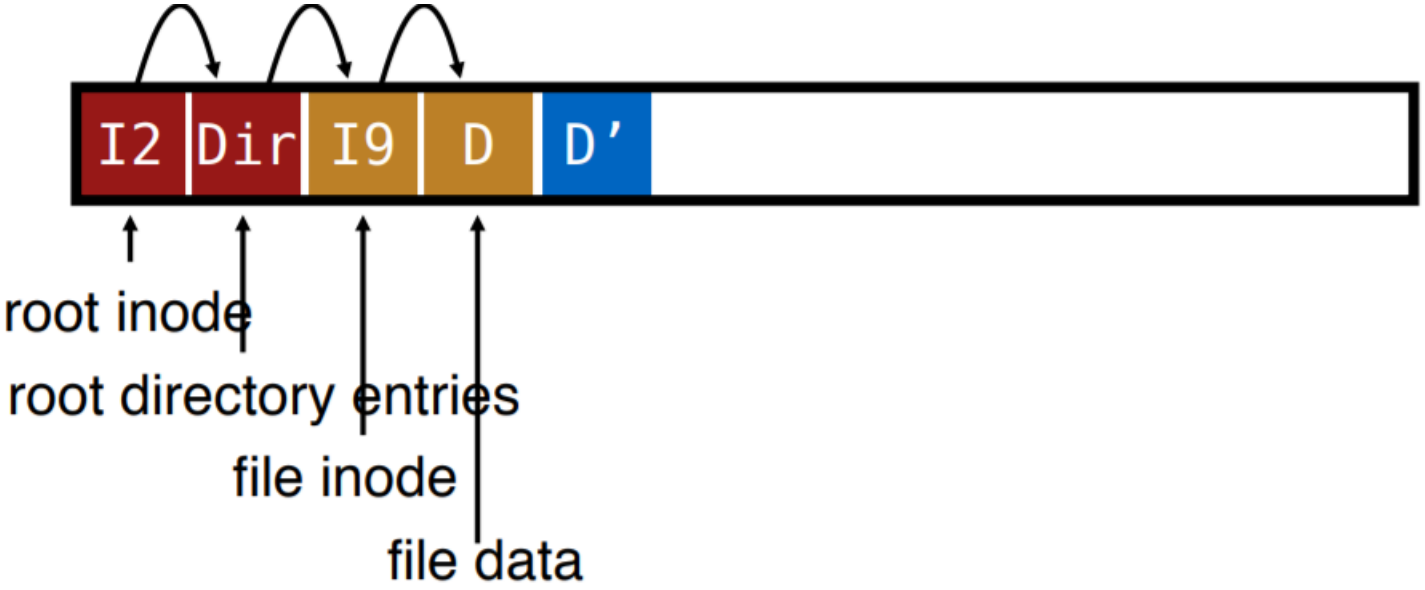


Data Structures for LFS (attempt 1)

- What data structures from FFS can LFS remove?
  - allocation structs: data + inode bitmaps
- What type of name is much more complicated?
  - Inodes are no longer at fixed offset
  - Use current offset on disk instead of table index for name
  - Note: when update inode, inode number changes!!

Overwrite Data in LFS – Attempt 1

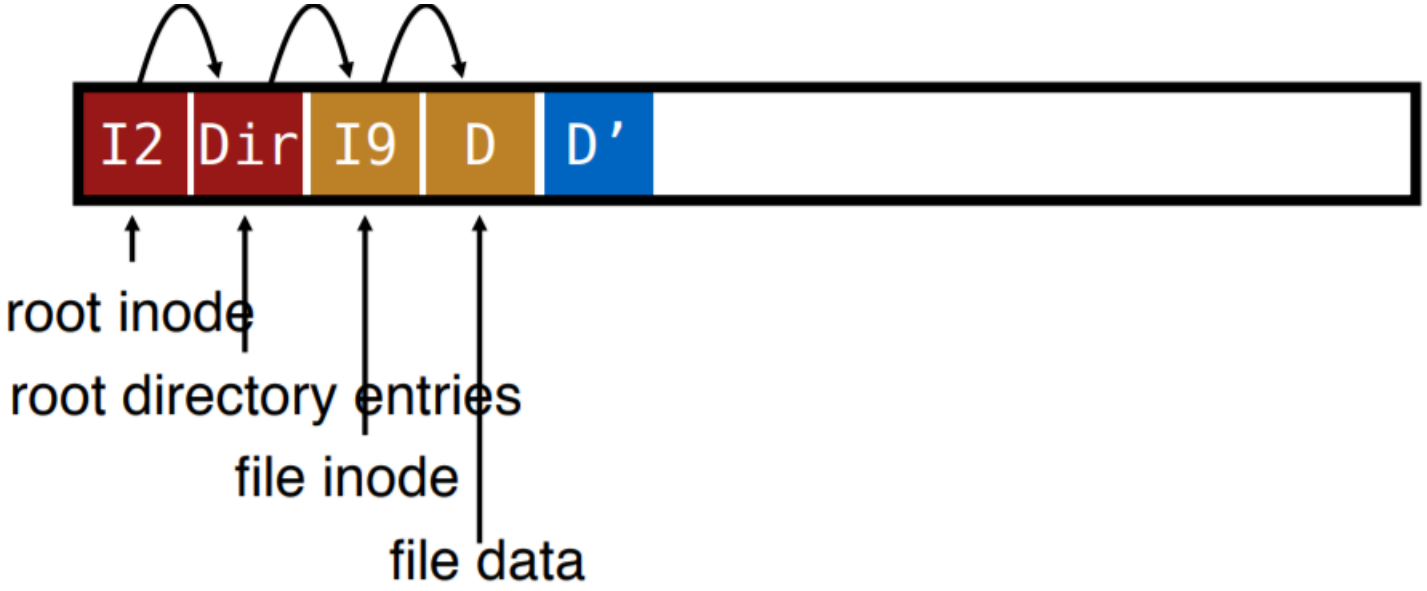
- Overwrite data in /file.txt



- How to update Inode 9 to point to new D' ???

Overwrite Data in LFS – Attempt 1

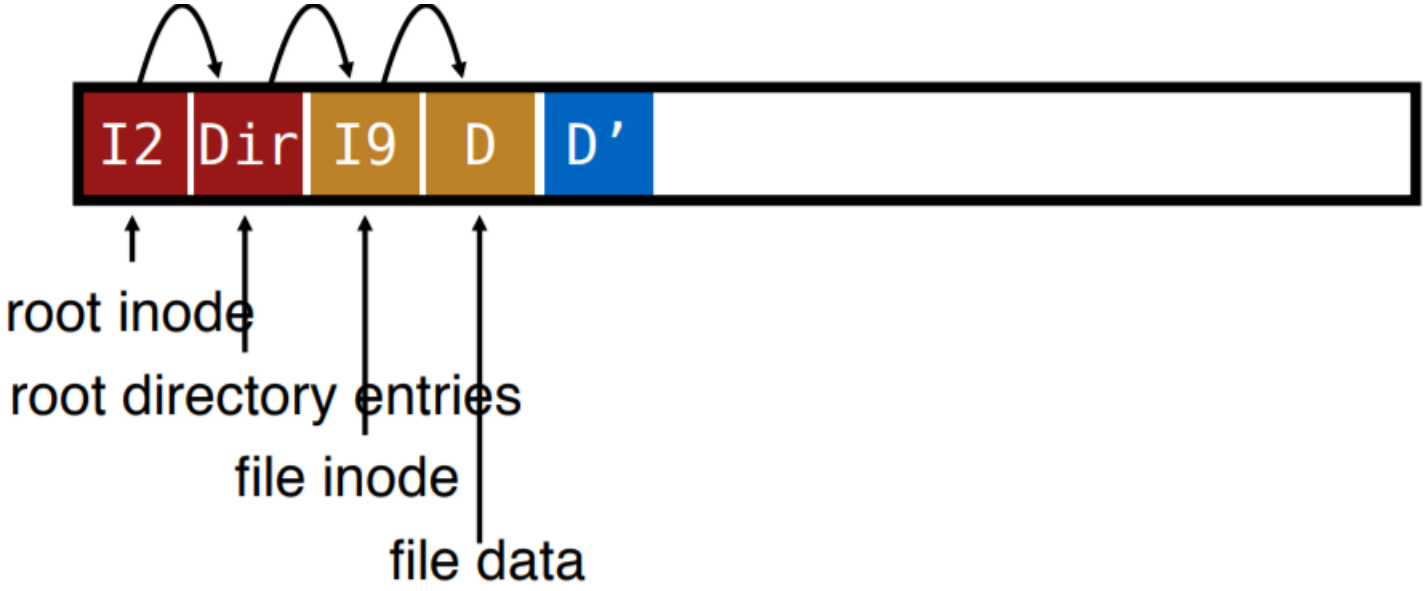
- Overwrite data in /file.txt



- Can LFS update Inode 9 to point to new D'?
  - NO! This would be a random write

Overwrite Data in LFS – Attempt 1

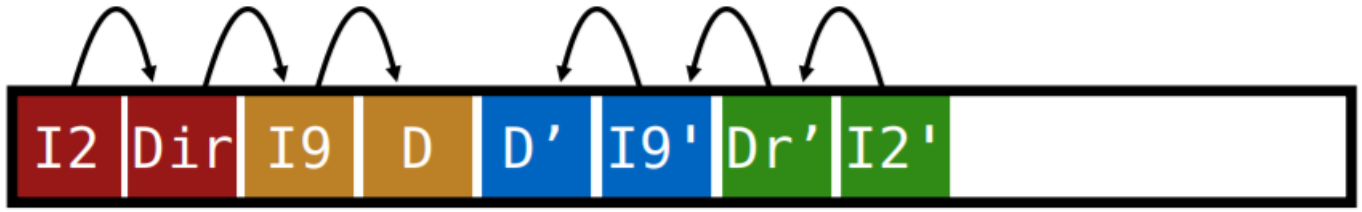
- Overwrite data in /file.txt



- Must update all structures in sequential order to log

Attempt 1: Problem w/ Inode Numbers

- **Problem:**
  - For every data update, must propagate updates all the way up directory tree to root
- **Why?**
  - When inode copied, its location (inode number) changes
- **Solution:**
  - Keep inode numbers constant; don't base name on offset
- **FFS found inodes with math. How now?**

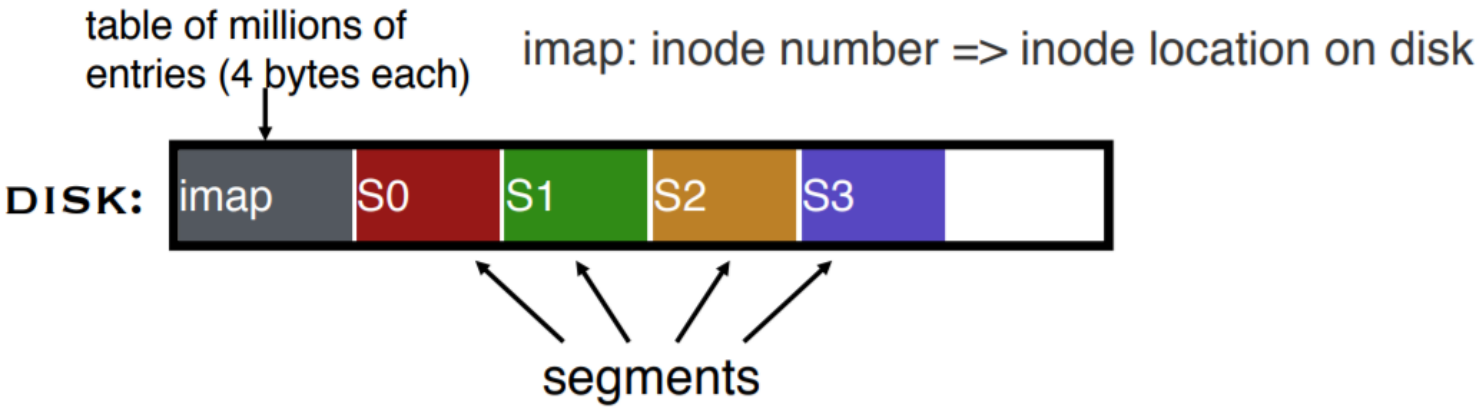


Data Structures for LFS (attempt 2)

- What data structures from FFS can LFS remove?
  - allocation structs: data + inode bitmaps
- What type of name is much more complicated?
  - Inodes are no longer at fixed offset
  - Use imap structure to map:
    - ✓ inode number => most recent inode location on disk

Where to keep Imap?

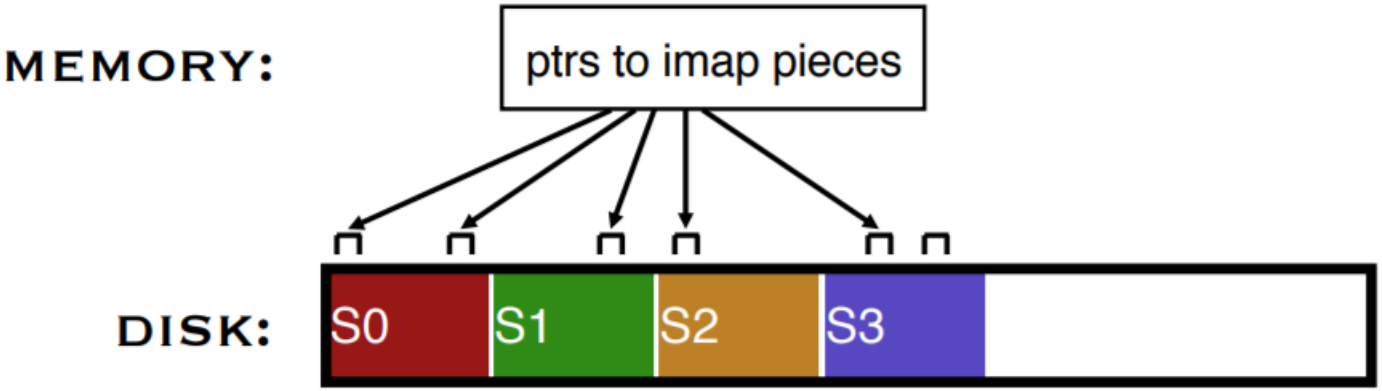
- **Where can imap be stored? Dilemma:**
  - 1. imap too large to keep in memory
  - 2. don't want to perform random writes for imap
- **Solution: Write imap in segments**
  - Keep pointers to pieces of imap in memory



Solution: Imap in Segments

❖ Solution:

- Write imap in segments
- Keep pointers to pieces of imap in memory
- Keep recent accesses to imap cached in memory



Disk Cleaning

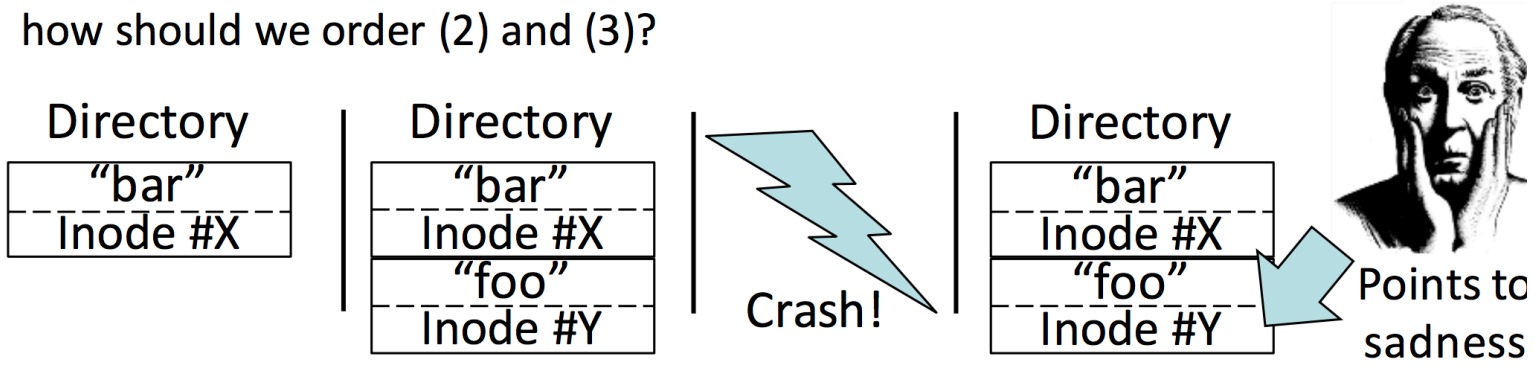
- When disk runs low on free space
  - Run a disk cleaning process
  - Compacts live information to contiguous blocks of disk
- Problem: long-lived data repeatedly copied over time
  - Solution: partition disk in to segments
  - Group older files into same segment
    - ✓ Do not clean segments with old files
- Try to run cleaner when disk is not being used

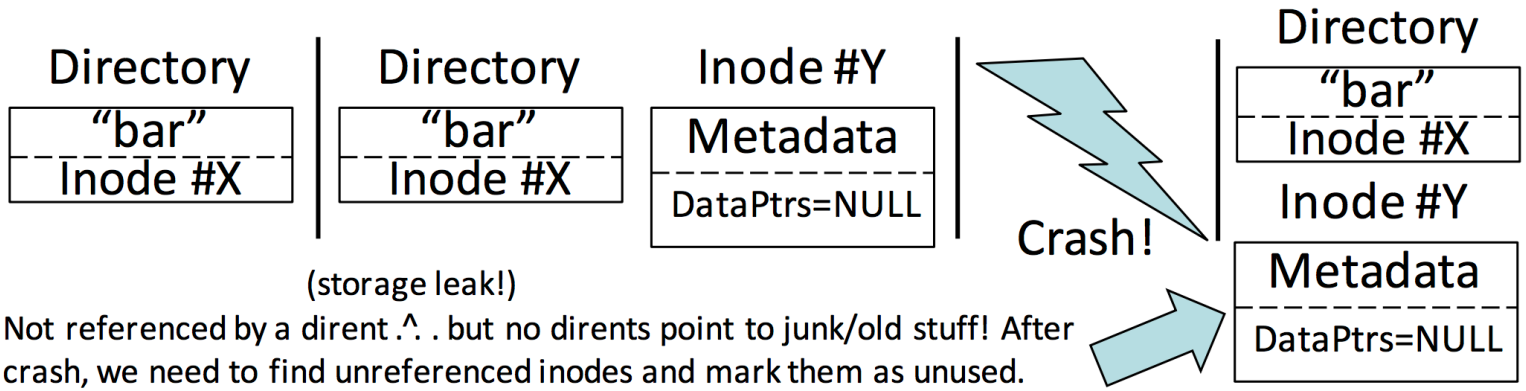
File System Crash Consistency

- Atomically update file system from one consistent state to another, which may require modifying several sectors, despite that the disk only provides atomic write of one sector at a time

Ensuring Consistency After Crashes

- To create a new file "foo", you need to write three things to disk:
  1. Update on-disk inode bitmap to allocate a new inode
  2. Write the new inode for "foo" to disk
  3. Write updated version of the directory that points to the new inode
- The order of the writes makes a difference! Suppose (1) has completed . . . how should we order (2) and (3)?





(storage leak!)

Not referenced by a dirent .^. . but no dirents point to junk/old stuff! After crash, we need to find unreferenced inodes and mark them as unused.

Why Is Crash Consistency Hard?

- We want to atomically move the file system from one state to another, but . . .
- A single, high-level file system operation like "create a new file" often corresponds to multiple disk writes, but the machine may crash without completing all writes
- The file system may issue writes to disk asynchronously, to allow processes to continue execution immediately after issuing IOs
  - Ex: FFS would only flush data once every 30 seconds unless explicitly asked via a sync() request
- To improve performance, the hard disk may complete writes in a different order than they were generated!
  - Ex: If the hard disk receives w0 (whose destination is far from the current position of the disk head), and w1 (whose destination is nearby), the disk may write w1 first

fsck

- In FFS, the fsck ("file system checker") tool must be run after a crash to restore the file system to a consistent state
  - "Consistent" is defined as "all metadata is in agreement (e.g., no block that is marked as unallocated is referenced by an inode)"
  - This definition of "consistent" is NOT the same as "all writes which made it to the disk pre-crash must be reflected in the post-crash file system"—in other words, fsck may discard some write data
  - fsck is run on the file system before the file system is mounted, i.e., before the file system can be accessed by other applications

fsck

- Ex: Appending to a file via a direct pointer requires three disk writes
  - (1) inode (to update data pointer, file size, last update time)
  - (2) data block bitmap (to indicate that a new data block has been allocated)
  - (3) the data block itself
- Suppose that writes (1) and (3) complete, but (2) doesn't
  - Inconsistency: inode points to a valid data block, but the block bitmap wasn't updated, so the new data block isn't marked as in-use
  - fsck resolves block bitmap inconsistencies by treating inodes as ground truth
    - ✓ When fsck starts, it initializes the bitmap to "all unallocated"
    - ✓ Start from root directory, fsck recursively scans all inodes, and marks all reachable data blocks as allocated in the bitmap
  - At the end of a fsck, metadata is always consistent, but file *data* may contain junk!
    - ✓ Ex: Writes (1) and (2) complete, but not (3)

fsck

- **Besides fixing the data block bitmaps, fsck does a bunch of other checks**
  - Ex: An inode's link count refers to the number of directory entries that refer to inode (the number may be higher than 1 due to hard links)
  - fsck traverses the entire file system, starting from the root directory, updating each inode's link count
- **Disadvantages of fsck**
  - Implementing fsck requires deep knowledge of the file system: the code is complicated and hard to get right
  - fsck is very slow, because it requires multiple traversals of the entire file system!
    - ✓ Ideally, the recovery effort should be proportional to the number of outstanding writes at the time of the crash
    - ✓ Journaling achieves this goal!

THX!