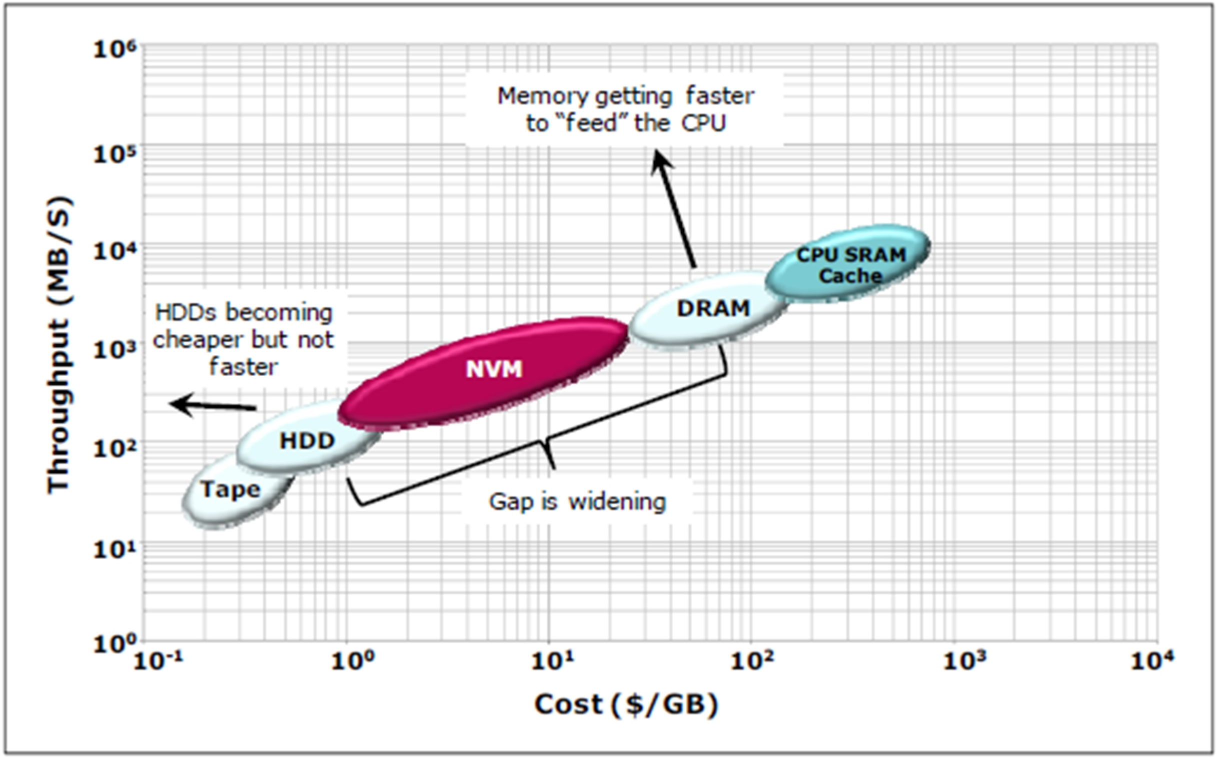
CS3210: File Systems

Tim Andersen

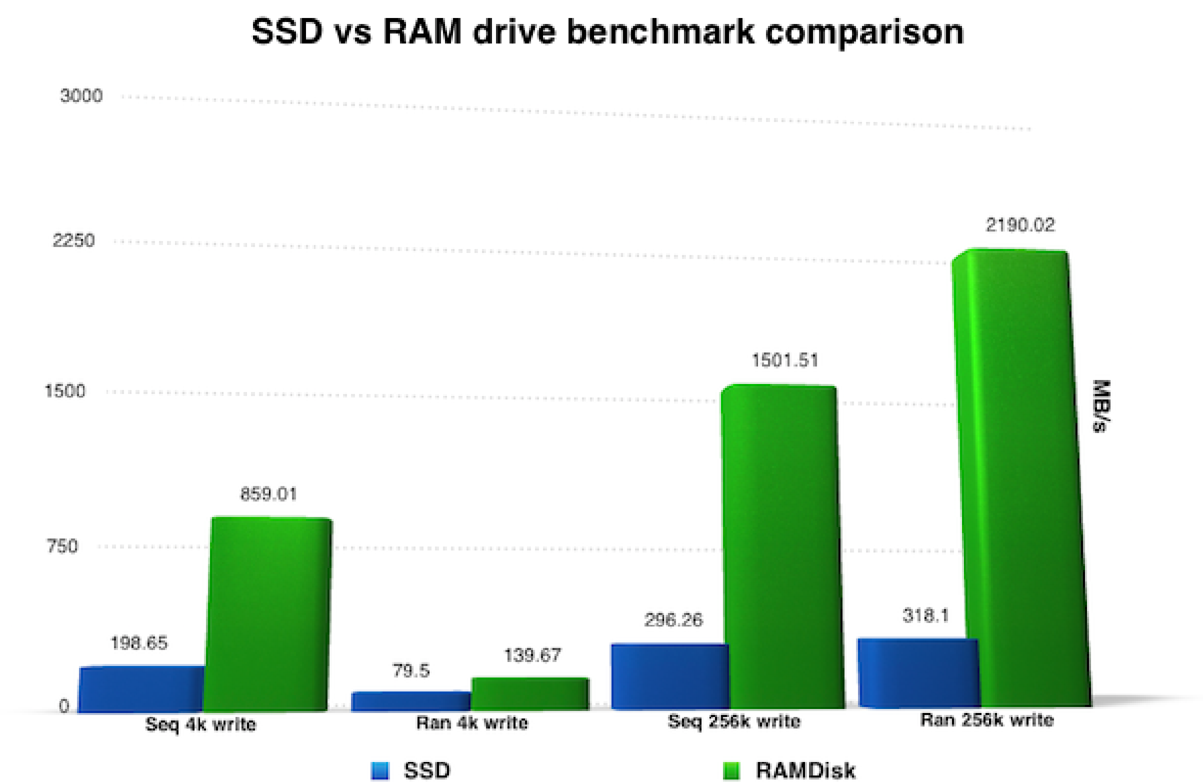
Lecture plan:

- File systems
- API -> disk layout
    - dumpfs
- Buffer cache
- xv6 in action - code walk

Storage trend



Do SSDs solve the problem?



- http://www.makeuseof.com/tag/ram-drives-faster-ssds-5-things-must-know/

High speed storage in NVM is approaching RAM

- High performance data recorders can approach RAM speeds, e.g., 2.5 GB/sec
- These solutions, however, are far more expensive than DRAM
- Used in applications where reliable persistent storage is required such as real-time sensor (radar, imagery, etc.) data recording.

- None of this is useful, however, without an efficient file system.

Why are file systems useful?

- Durability across restarts
- Naming and organization
- Sharing among programs and users

Why interesting?

- Crash recovery
- Performance
- API design for sharing
- Security for sharing
- Abstraction is useful: pipes, devices, /proc, /afs, etc.
    - so FS-oriented apps work with many kinds of objects
- You will implement one for JOS!

API example -- UNIX/Posix/Linux/xv6/&c:

- fd = open("x/y", -);
- write(fd, "abc", 3);
- link("x/y", "x/z");
- unlink("x/y");
- Plan 9 OS (Bell labs)
  - Attempts to structure entire OS as a filesystem
  - http://plan9.bell-labs.com/plan9/

High-level API choices

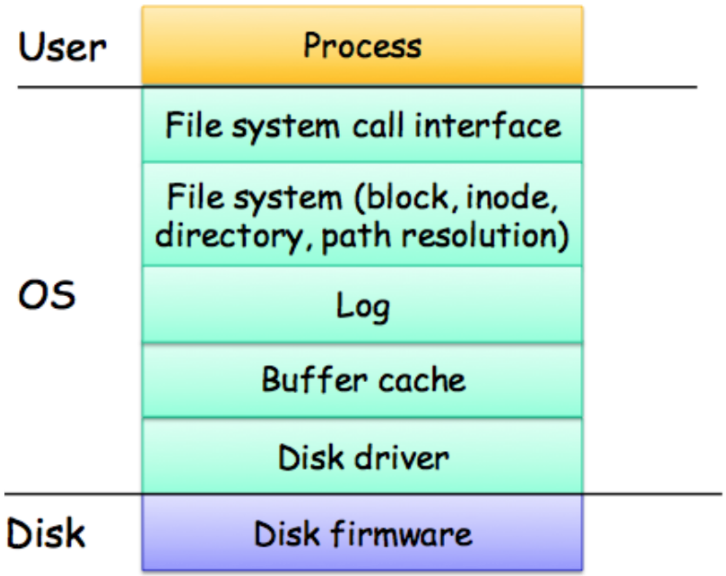
- Granularity
  - files, virtual disks, databases
- File content
  - byte array, records, b-tree (or key-value stores)
- Organization:
  - name hierarchy vs flat names (object IDs)
- Synchronization
  - None vs locks, transaction rollbacks

API implications:

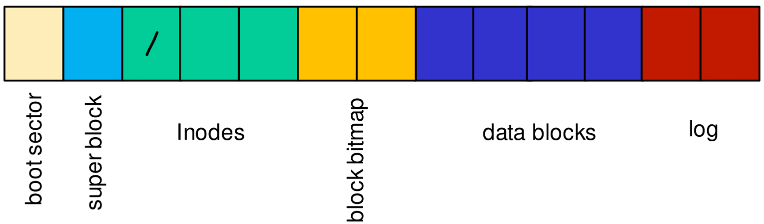
- File descriptor (fd) refers to something
    - preserved even if file name changes or deleted
- File can have multiple links i.e., multiple directories
    - file info should be stored somewhere other than directory
- Thus a file is independent of its names
    - it is called an "inode"
    - inode must keep link count (tells us when to free)
    - inode must have count of open fds'
    - inode deallocation deferred until last link, fd removed

Let us talk about xv6

FS software layers



On-disk layout

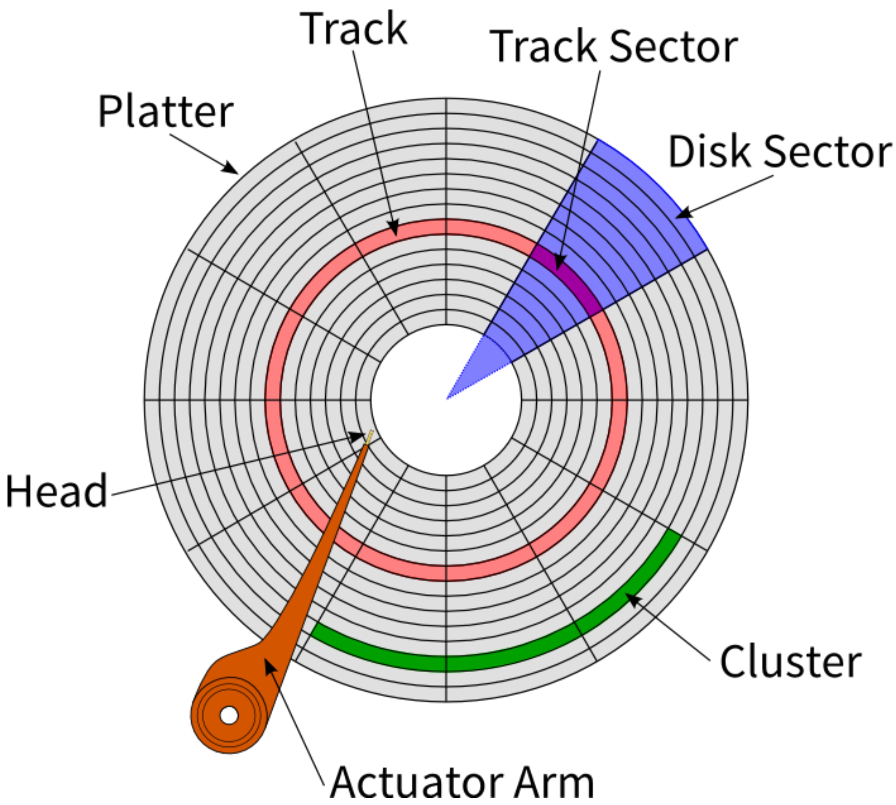


- fs.h, fs.c, mkfs.c
- *struct superblock*

7

- Let's discuss each layer

Hard disk


Track
Track Sector
Platter
Disk Sector
Head
Cluster
Actuator Arm

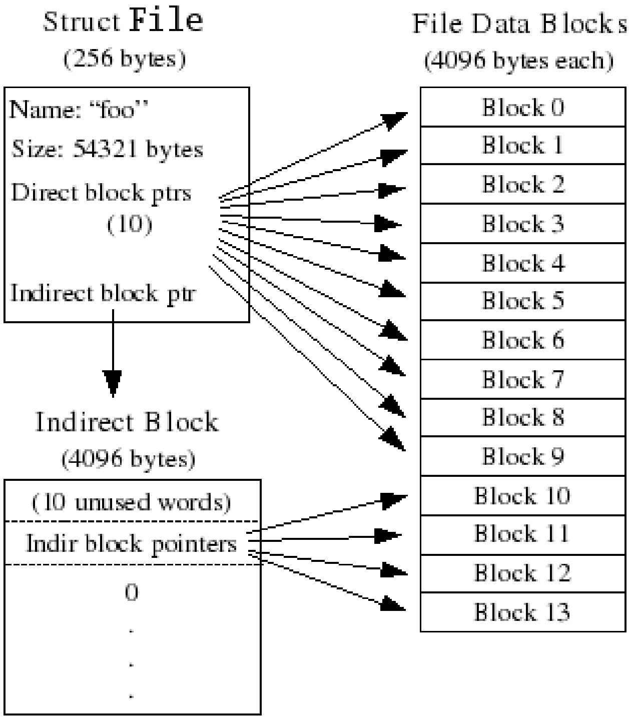
Disk blocks

- Most o/s use blocks of multiple sectors
  - e.g. 4 KB blocks = 8 sectors
  - to reduce book-keeping and seek overheads
- xv6 uses single-sector blocks for simplicity
- "meta-data"
  - everything on disk other than file content
  - super block, i-nodes, bitmap, directory content

Inode

- On-disk
  - type (free, file, directory, device)
  - nlink
  - size
  - addrs[12+1]
- Q: Why 12+1 ?

Direct and indirect blocks



Direct and indirect blocks

- How to find file's byte 8000?
    - logical block 15 = 8000 / BLOCK_SIZE
    - 3rd entry in the indirect block
- i-node structure
    - each i-node has an i-number
    - easy to turn i-number into inode
    - inode is 64 bytes long
    - byte address on disk: 2*512 + 64*inum

Directory contents

- Directory much like a file
    - but user can't directly write
- Content is array of dirents
- Dirent:
    - inum
    - 14-byte file name
    - dirent is free if inum is zero

Inode operations

- kernel keeps inode in-memory until reference != 0
- ialloc() - allocate inode
- ilock() - and iunlock sync access to inode
- iget() - returns the inode struct and inc ref count
- iput() - dec the ref count and frees is ref = 0
- iupdate() - copy modified inode to the disk

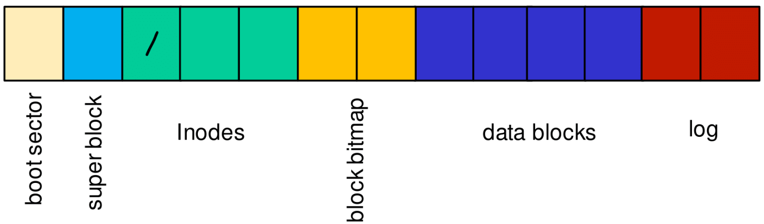
Inode xv6 usage

```
ip = iget(dev, inum)
ilock(ip)
... examine and modify ip->xxx ...
iunlock(ip)
iput(ip)
~~~~
```

Concurrent calls to ialloc?

- Will they get the same inode?
  - note bread / write / brelse in ialloc
  - bread locks the block, perhaps waiting, and reads from disk
  - brelse unlocks the block
- Why do we use iget even after finding an inode?
  - Let's see the iget method
- Q: Why iget does not hold ilock?

Free block bitmap



- fs.h, fs.c, mkfs.c
- *struct superblock*

7

- xv6 maintain free bitmap on disk – one bit per block (sb->bmapstart)
  - 0 means block is free, 1 means block in use
- Checking if a block is free if you know block number
  - buf[blockNum/8] & (0x1 << (blockNum % 8))

Block allocation sequence

- balloc() - allocates new disk block
- readsb() - into to sb struct in memory
- Iterate over the bitmap blocks for free block
- If block found, update corresponding bit
- bfree() - clear the relevant bit

Buffer cache layer

- A double-linked list of buf structures
- Holding cached copies of disk block contents
- Two jobs:
  - synchronize access to disk blocks
  - one block on disk – one block in memory
  - one kernel thread at the time use same block
- Cache popular blocks in fixed buffers

Buffer cache layer

- Flags:
    - B_BUSY – buffer locked
    - B_VALID – buffer has been read from disk
    - B_DIRTY – buffer was modified and should be written to disk
- Interface:
    - binit() - called by main
    - bread() - to read buffer from block on disk
    - bwrite()- to write buf to disk
    - brelse()- to release buf when done and move it to the head

Buffer cache layer

- Let's look at the block cache in bio.c
    - block cache holds just a few recently-used blocks
- FS calls bread, which calls bget
    - bget looks to see if block already cached
    - if present and not B_BUSY, return the block
    - if present and B_BUSY, wait
    - if not present, re-use an existing buffer
- Q: why goto loop after sleep()?

Replacement policy

- xv6 implements LRU for buffer cache replacement.
- Maintain the buffers in a doubly-linked list.
- When done accessing a buffer (at the time of clearing the busy bit),
  - move the buffer to the front of the buffer cache list
  - start replacement at the last entry of the list.
- Let's discuss buffer cache and disk driver interaction

Disk driver

- Let's look into ide.c
- ideinit() initializes the IDE
    - Q. What does this line mean ioapicenable(IRQ_IDE, ncpu - 1)?
    - Q. Why do we check if disk 1 is present?

Disk driver

- ide_rw() - read or write a block from/to the disk
  - Q: How to handle multiple ide_rw() calss?
- Notice just one lock (ide_lock) for enforcing multiple invariants
- iderw and ideintr share the request queue using idelock
- Q: What if we enable interrupts with single processor?

Now, let's look at xv6 in action

- Focus on disk writes
- Illustrate on-disk data structures via how updated

Q: How does xv6 create a file?

```
$ echo > a
write 34 ialloc (from create sysfile.c; mark it non-free)
write 34 iupdate (from create; initialize nlink &c)
write 59 writei (from dirlink fs.c, from create)
```

- xv6 supports logging which we will discuss next class
  - log_write replaces bwrite()
- Q: what's in block 34?
  - look at create() in sysfile.c
- Q: why *two* writes to block 34?
- Q: what is in block 59?

xv6 Write data to a file

```
$ echo x > a
write 58 balloc- (from bmap, from writei)
write 613 bzero
write 613 writei (from filewrite file.c)
write 34 iupdate- (from writei)
write 613 writei
write 34 iupdate
```

- Q: what's in block 58, block 613?
  - look at writei call to bmap
  - look at bmap call to balloc

Delete a file

```
$ rm a
write 59 writei (from sys_unlink; directory content)
write 34 iupdate (from sys_unlink; link count of file)
write 58 bfree- (from itrunc, from iput)
write 34 iupdate (from itrunc)
write 34 iupdate (from iput)
```

Q: How fast xv6 apps. can read big files?

- First reads data from disk to buffer cache
- Then, from buffer cache to user space
- What happens if we pass user buffer to the disk device driver?
- Q: How much RAM should we dedicate to disk buffers?