```
6.828 2014 Lecture 12: File System

lecture plan:
  file systems
  API -> disk layout
  caching

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, Plan 9
    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");

high-level choices visible in this API
  objects: files (vs virtual disk, DB)
  content: byte array (vs 80-byte records, BTree)
  naming: human-readable (vs object IDs)
  organization: name hierarchy
  synchronization: none (vs locking, versions)

a few implications of the API:
  fd refers to something
    that is preserved even if file name changes
    or if file is deleted while open!
  a file can have multiple links
    i.e. occur in multiple directories
    no one of those occurences is special
    so file must have info stored somewhere other than directory
  thus:
    FS records file info in an "inode" on disk
    FS refers to inode with i-number (internal version of FD)
    inode must have link count (tells us when to free)
    inode must have count of open FDs
    inode deallocation deferred until last link and FD are gone

let's talk about xv6

FS software layers
  system calls
  name ops | FD ops
  inodes
  inode cache
  log
  buffer cache
  ide driver

mechanical disks
  concentric tracks
  each track is a sequence of sectors, usually 512 bytes
  ECC on each sector
```

```
    can only read/write whole sectors
    thus: sub-sector writes are expensive (read-modify-write)

  mechanical disk performance
    rotating platter -- about 10,000 rpm = 166 rps = 6 ms per rotation
    arm moves in and out to the right track -- "seek" -- about 10 ms
    latency: time to read or write a single sector?
      13 ms to read or write a random sector
      this is an eternity for CPU -- 13 million instructions
      random I/O has throughput of only 40 kilobytes/sec -- laughable
    throughput: bytes/second for big sequential reads and writes?
      about 500 sectors per track
      thus 500*512 / 0.006 = 43 megabytes/sec
    big sequential writes have much higher throughput than random sector I/O!
    thus file systems have to pay attention to data layout
    SSDs are faster but share low random write performance

  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

  on-disk layout
    xv6 file system on 2nd IDE drive; first has just kernel
    xv6 treats IDE drive as an array of sectors, hides tracks
    0: unused
    1: super block (size, ninodes)
    2: array of inodes, packed into blocks
    X: block in-used bitmap (0=free, 1=inuse)
    Y: file/dir content blocks
    Z: log for transactions
    end of disk

  "meta-data"
    everything on disk other than file content
    super block, i-nodes, bitmap, directory content

  on-disk inode
    type (free, file, directory, device)
    nlink
    size
    addrs[12+1]

  direct and indirect blocks

  example:
    how to find file's byte 8000?
    logical block 15 = 8000 / 512
    3rd entry in the indirect block

  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

  you should view FS as an on-disk data structure
```

```
    [tree: dirs, inodes, blocks]
    with two allocation pools: inodes and blocks

  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?

  rm fs.img

  $ echo > a
  write 4 ialloc (from create sysfile.c; mark it non-free)
  write 4 iupdate (from create; initialize nlink &c)
  write 29 writei (from dirlink fs.c, from create)

  Q: what's in block 4?
     look at create() in sysfile.c

  Q: why *two* writes to block 4?

  Q: what is in block 29?

  Q: what if there are 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

  Q: how does xv6 write data to a file?

  $ echo x > a
  write 28 balloc  (from bmap, from writei)
  write 420 bzero
  write 420 writei (from filewrite file.c)
  write 4 iupdate  (from writei)
  write 420 writei
  write 4 iupdate

  Q: what's in block 28?
     look at writei call to bmap
     look at bmap call to balloc

  Q: what's in block 420?

  Q: why the iupdate?
     file length and addrs[]

  Q: why *two* writei+iupdate?

  Q: how does xv6 delete a file?

  $ rm a
  write 29 writei (from sys_unlink; directory content)
  write 4 iupdate (from sys_unlink; link count of file)
  write 28 bfree  (from itrunc, from iput)
  write 4 iupdate (from itrunc)
  write 4 iupdate (from iput)

  Q: what's in block 29?
     sys_unlink in sysfile.c

  Q: what's in block 4?

  Q: what's in block 28?
```

```
   look at iput

Q: why three iupdates?

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()?

Q: what is the block cache replacement policy?
   prev ... head ... next
   bget re-uses bcache.head.prev -- the "tail"
   brelse moves block to bcache.head.next

Q: is that the best replacement policy?

Q: what if lots of processes need to read the disk? who goes first?
   iderw appends to idequeue list
   ideintr calls idestart on head of idequeue list
   so FIFO

Q: is FIFO a good disk scheduling policy?
   priority to interactive programs?
   elevator sort?

Q: how fast can an xv6 application read big files?
   contiguous blocks?
   blow a rotation -- no prefetch?

Q: why does it make sense to have a double copy of I/O?
   disk to buffer cache
   buffer cache to user space
   can we fix it to get better performance?

Q: how much RAM should we dedicate to disk buffers?
```