```
6.S081 2020 Lecture 14: File System

why are file systems useful?
  durability across restarts
  naming and organization
  sharing among programs and users

why interesting?
  crash recovery
  performance/concurrency
  sharing
  security
  abstraction is useful: pipes, devices, /proc, /afs, Plan 9
    so FS-oriented apps work with many kinds of objects
  topic of in two labs

API example -- UNIX/Posix/Linux/xv6/&c:
  fd = open("x/y", -);
  write(fd, "abc", 3);
  link("x/y", "x/z");
  unlink("x/y");
  write(fd, "def", 3);
  close(fd);
  // file y/z contains abcdef

high-level choices visible in the UNIX FS 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)
    link()/unlink() can change name hierarchy concurrently with an open()
  there are other file system APIs, sometimes quite different!

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
  virtio_disk driver

data stored on a persistent medium
  data stays on disk without power
  common storage medium:
    hard disk drives (big but slow, inexpensive)
```

```
    solid state drives (smaller, but fast, and more expensive)
  historically, disks were read/write usually in 512-byte units, called sectors

hard disk drives (HDD)
  concentric tracks
  each track is a sequence of sectors
  head must seek, disk must rotate
    random access is slow (5 or 10ms per access)
    sequential access is much faster (100 MB/second)
  ECC on each sector
  can only read/write whole sectors
  thus: sub-sector writes are expensive (read-modify-write)

solid state drives (SSD)
  non-volatile "flash" memory
  random access: 100 microseconds
  sequential: 500 MB/second
  internally complex -- hidden except sometimes performance
    flash must be erased before it's re-written
    limit to the number of times a flash block can be written
    SSD copes with a level of indirection -- remapped blocks

for both HDD and SSD:
  sequential access is much faster than random
  big reads/writes are faster than small ones
  both of these influence FS design and 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 2-sector blocks

on-disk layout
  xv6 treats disk as an array of sectors (ignoring physical properties of disk)
  0: unused
  1: super block (size, ninodes)
  2: log for transactions
  32: array of inodes, packed into blocks
  45: block in-use bitmap (0=free, 1=used)
  46: file/dir content blocks
  end of disk

xv6's mkfs program generates this layout for an empty file system
  the layout is static for the file system's lifetime
  see output of mkfs

"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 7 = 8000 / BSIZE (=1024)
  7th entry in addrs

each i-node has an i-number
  easy to turn i-number into inode
```

```
    inode is 64 bytes long
    byte address on disk: 32*BSIZE + 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 & make qemu

  $ echo hi > x
    // create
    bwrite: block 33 by ialloc   // allocate inode in inode block 33
    bwrite: block 33 by iupdate  // update inode (e.g., set nlink)
    bwrite: block 46 by writei   // write directory entry, adding "x" by dirlink()
    bwrite: block 32 by iupdate  // update directory inode, because inode may have changed
    bwrite: block 33 by iupdate  // itrunc new inode (even though nothing changed)
    // write
    bwrite: block 45 by balloc   // allocate a block in bitmap block 45
    bwrite: block 524 by bzero   // zero the allocated block (block 524)
    bwrite: block 524 by writei  // write to it (hi)
    bwrite: block 33 by iupdate  // update inode
    // write
    bwrite: block 524 by writei  // write to it (\n)
    bwrite: block 33 by iupdate  // update inode

  call graph:
    sys_open        sysfile.c
      create        sysfile.c
        ialloc      fs.c
        iupdate     fs.c
        dirlink     fs.c
          writei    fs.c
            iupdate fs.c
      itrunc        sysfile.c
        iupdate     fs.c

  Q: what's in block 33?
     look at create() in sysfile.c

  Q: why *two* writes to block 33?

  Q: what is in block 32?

  Q: how does xv6 write data to a file?  (see write part above)

  call graph:
    sys_write       sysfile.c
      filewrite     file.c
        writei      fs.c
          bmap
```

```
          balloc
            bzero
        iupdate

Q: what's in block 45?
   look at writei call to bmap
   look at bmap call to balloc

Q: what's in block 524?

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

Q: how does xv6 delete a file?

$ rm x
bwrite: block 46 by writei    // from sys_unlink; directory content
bwrite: block 32 by iupdate   // from writei of directory content
bwrite: block 33 by iupdate   // from sys_unlink; link count of file
bwrite: block 45 by bfree     // from itrunc, from iput
bwrite: block 33 by iupdate   // from itrunc; zeroed length
bwrite: block 33 by iupdate   // from iput; marked free

call graph:
  sys_unlink
    writei
    iupdate
    iunlockput
      iput
        itrunc
          bfree
          iupdate
        iupdate

Q: what's in block 46?
   sys_unlink in sysfile.c

Q: what's in block 33?

Q: what's in block 45?
   look at iput

Q: why four iupdates?

Concurrency in file system
  xv6 has modest goals
    parallel read/write of different files
    parallel pathname lookup
  But, even those poses interesting correctness challenges

E.g., 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

Let's look at the block cache in bio.c
   block cache holds just a few recently-used blocks
   bcache at start of bio.c

FS calls bread, which calls bget
   bget looks to see if block already cached
   if present, lock (may wait) and return the block
     may wait in sleeplock until current using processes releases
     sleep lock puts caller to sleep, if already locked
```

```
    if not present, re-use an existing buffer
    b->refcnt++ prevents buf from being recycled while we're waiting
    invariant: one copy of a disk block in memory

 Two levels of locking here
   bcache.lock protects the description of what's in the cache
   b->lock protects just the one buffer

 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: 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?

 Pathname lookup
   Traverse a pathname element at the time
   Potentially many blocks involved:
     inode of top directory
     data of top directory
     inode of next-level down
     .. and so on ..
   Each one of them might result in a cache miss
     disk access are expensive
   => Allow parallel pathname lookup
     If one process blocks on disk, another process may proceed with lookup
     Challenging: unlink may happen concurrent with lookup

 Let's look at namex() (kernel/fs.c)
   ilock(): locks current directory
   find next directory inode
   then unlock current directory
   another process may unlink the next inode
     but inode won't be deleted, because inode's refcnt > 0
   risk: next points to same inode as current (lookup of ".")
     unlock current before getting lock on next
     key idea: getting a reference separately from locking
```