I/O 2 / Filesystems 1

Changelog

Changes made in this version not seen in first lecture:

13 November: Correct cluster number on FAT directory entry slide.

last time

page replacement modifications for scanning
- Linux: guess file pages used once until multiple references
  (but non-file pages do actual LRU approximation)

readahead: proactive replacement
- detect sequential access patterns
- try to keep slightly ahead of program scanning a file

device drivers
- file (or block) interface — top half
- interrupt handling — bottom half

devices as magic memory
- connected to same bus as memory
- often via bus adaptors — or chains of them

on the homework (1)

yes, debugging more challenging than I expected

what I did? — lots of cprintfs

...including (virtual and physical) *addresses* and *process IDs* involved

should be able to track intended state of page tables/physical pages

try to make really simple test cases

- minimize number of pages active

could also potentially use GDB

on the homework (2)

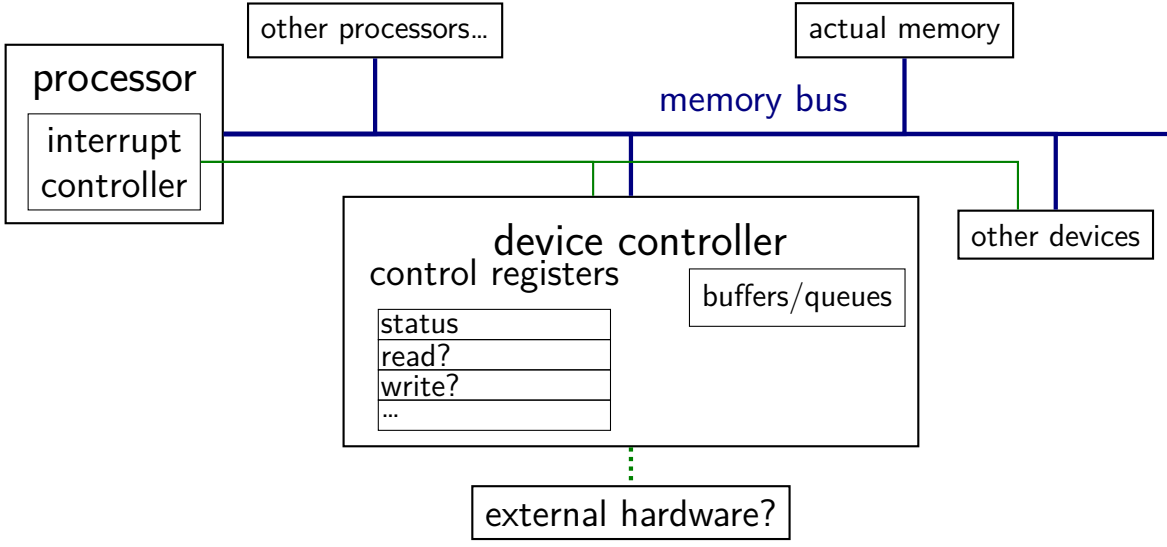
anonymous feedback:

> "This homework is the most frustrating homework we've had this semester. It's like everything can go wrong and once it goes wrong it's almost impossible to figure out what went wrong without knowing every possible interaction. This along with the number of things that can go wrong just makes it very frustrating. At least with previous homeworks, we were able to learn from most of our mistakes whereas here it's like everything is guess and check. I think if we had a checkpoint where we implemented only allocate on demand and another checkpoint where we implemented copy on write this process could have been better."

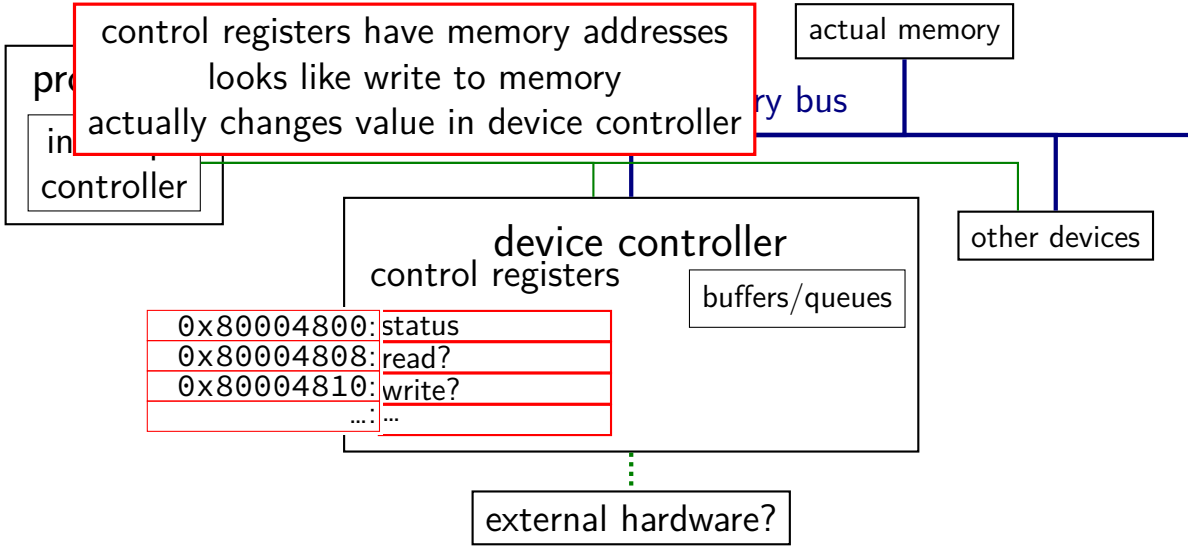
next time — will split into checkpoint (but too late now)

on the homework (3)

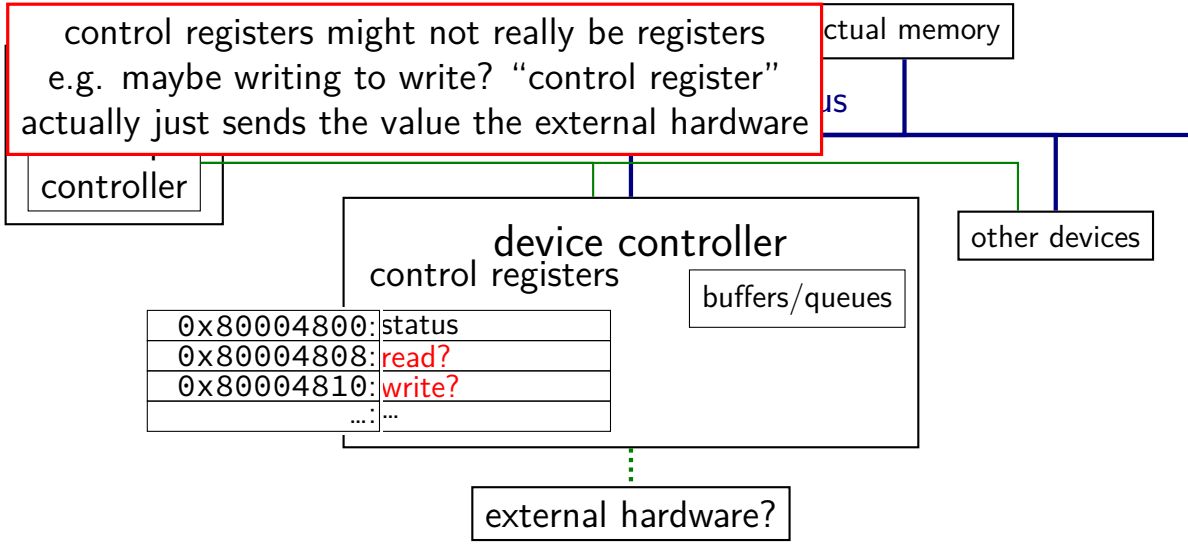
connecting devices



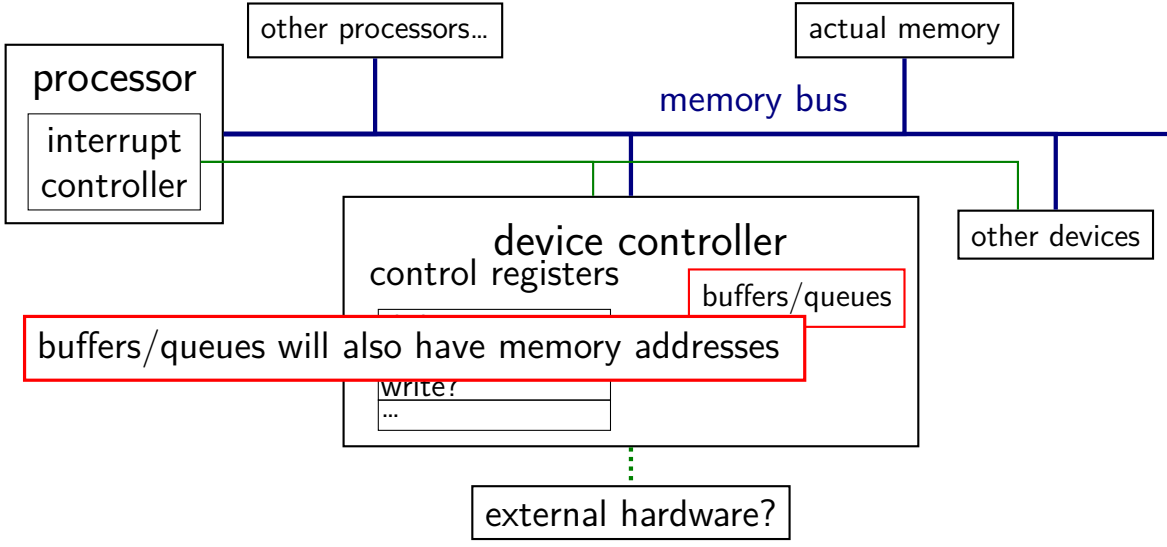
connecting devices



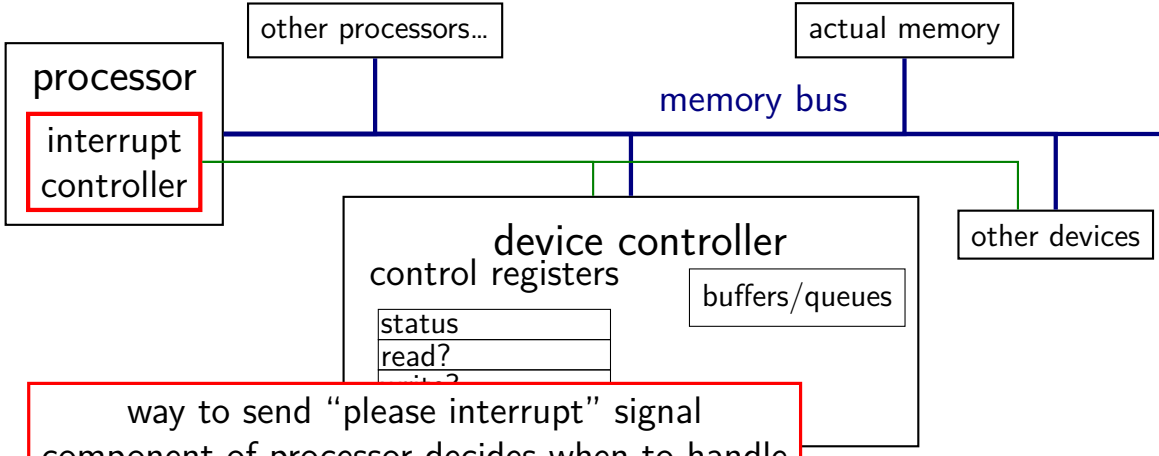
connecting devices



connecting devices

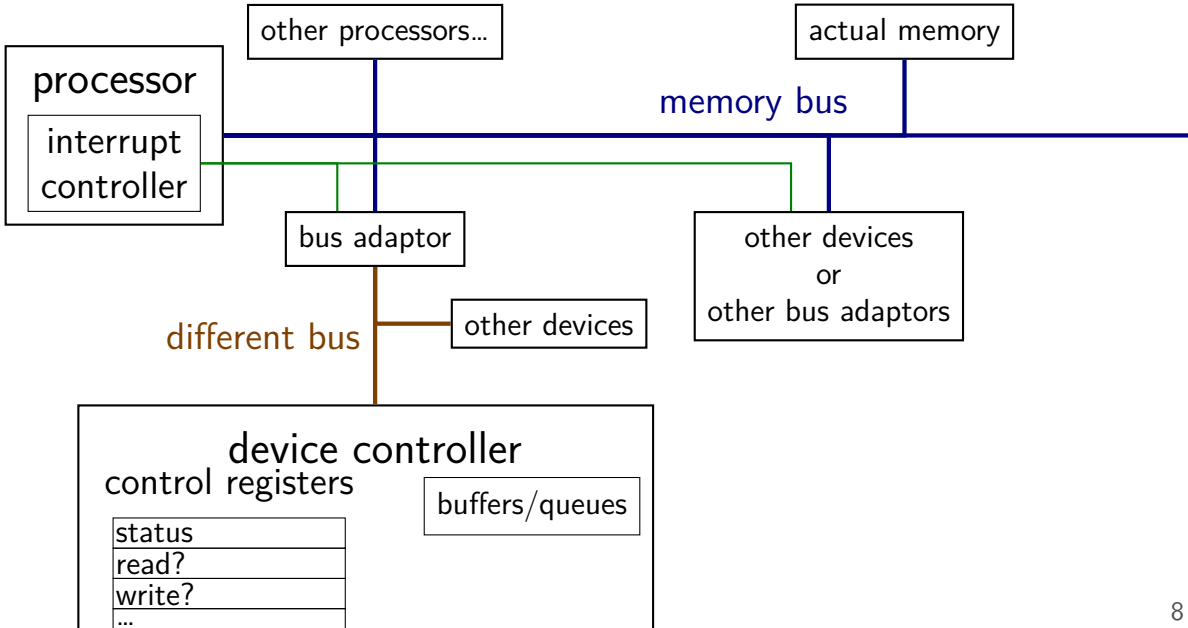


connecting devices



way to send "please interrupt" signal
component of processor decides when to handle
(deals with ordering, interrupt disabling,
which of several processors handles it, ..., etc.)

bus adaptors



devices as magic memory (1)

devices expose memory locations to read/write

use read/write instructions to manipulate device

example: keyboard controller

read from magic memory location — get last keypress/release

reading location clears buffer for next keypress/release

get interrupt whenever new keypress/release you haven't read

devices as magic memory (1)

devices expose memory locations to read/write

use read/write instructions to manipulate device

example: keyboard controller

read from magic memory location — get last keypress/release

reading location clears buffer for next keypress/release

get interrupt whenever new keypress/release you haven't read

devices as magic memory (1)

devices expose memory locations to read/write

use read/write instructions to manipulate device

example: keyboard controller

read from magic memory location — get last keypress/release

reading location clears buffer for next keypress/release

get interrupt whenever new keypress/release you haven't read

device as magic memory (2)

example: display controller

write to pixels to magic memory location — displayed on screen

other memory locations control format/screen size

example: network interface

write to buffers

write "send now" signal to magic memory location — send data

read from "status" location, buffers to receive

what about caching?

caching “last keypress/release”?

I press ‘h’, OS reads ‘h’, does that get cached?

what about caching?

caching “last keypress/release”?

I press ‘h’, OS reads ‘h’, does that get cached?

...I press ‘e’, OS reads what?

what about caching?

caching “last keypress/release”?

I press ‘h’, OS reads ‘h’, does that get cached?

...I press ‘e’, OS reads what?

solution: OS can mark memory uncachable

x86: bit in page table entry can say “no caching”

aside: I/O space

x86 has a "I/O addresses"

like memory addresses, but accessed with different instruction

- `in` and `out` instructions

historically — and sometimes still: separate I/O bus

more recent processors/devices usually use memory addresses

- no need for more instructions, buses
- always have layers of bus adaptors to handle compatibility issues
- other reasons to have devices and memory close (later)

xv6 keyboard access

two control registers:

KBSTATP: status register (I/O address `0x64`)
KBDATAP: data buffer (I/O address `0x60`)

```
st = inb(KBSTATP);  // in instruction: read from I/O address
if ((st & KBS_DIB) == 0)  // bit KBS_DIB indicates data in b
  return -1;
data = inb(KBDATAP);  // read from data --- *clears* buffer

/* interpret data to learn what kind of keypress/release */
```

programmed I/O

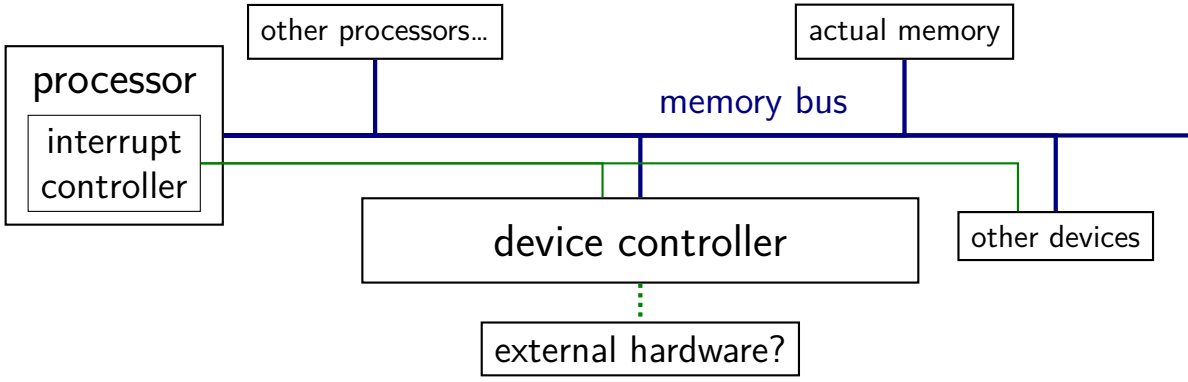
"programmed I/O": write to or read from device controller buffers directly

OS runs loop to transfer data to or from device controller

might still be triggered by interrupt

- new data in buffer to read?
- device processed data previously written to buffer?

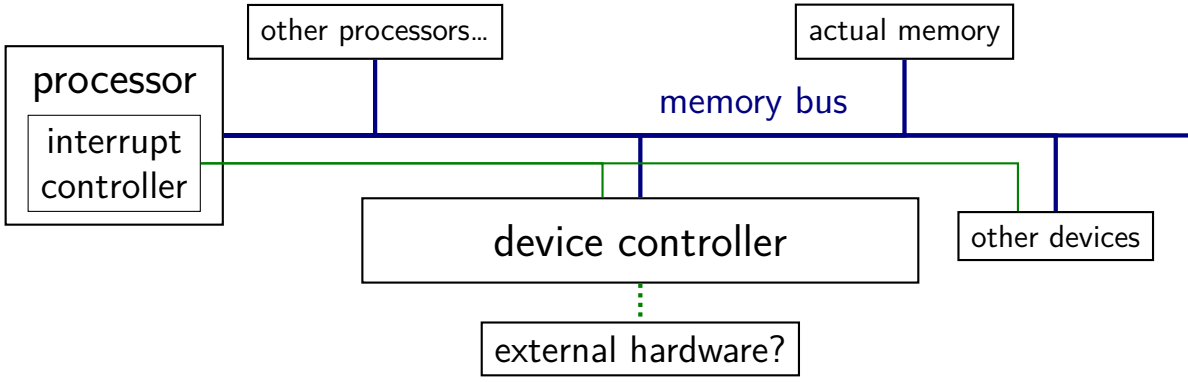
direct memory access (DMA)



observation: devices can read/write memory

can have device copy data to/from memory

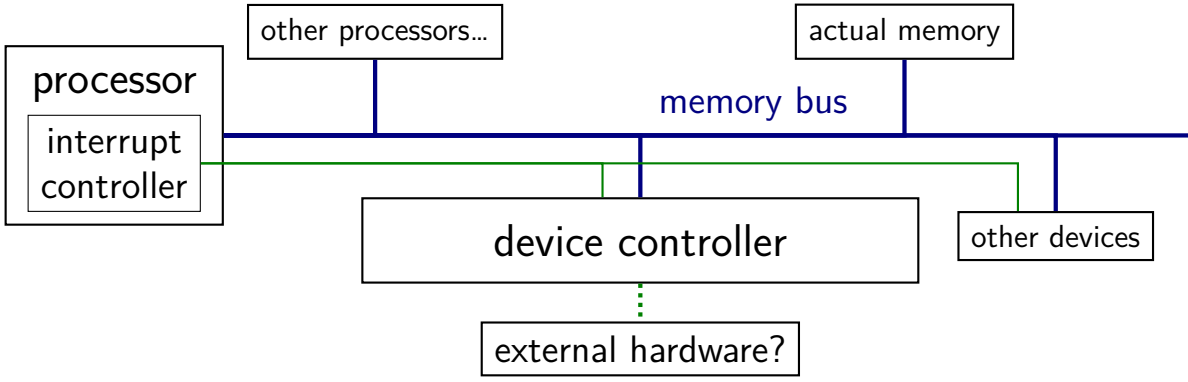
direct memory access (DMA)



observation: devices can read/write memory

can have device copy data to/from memory

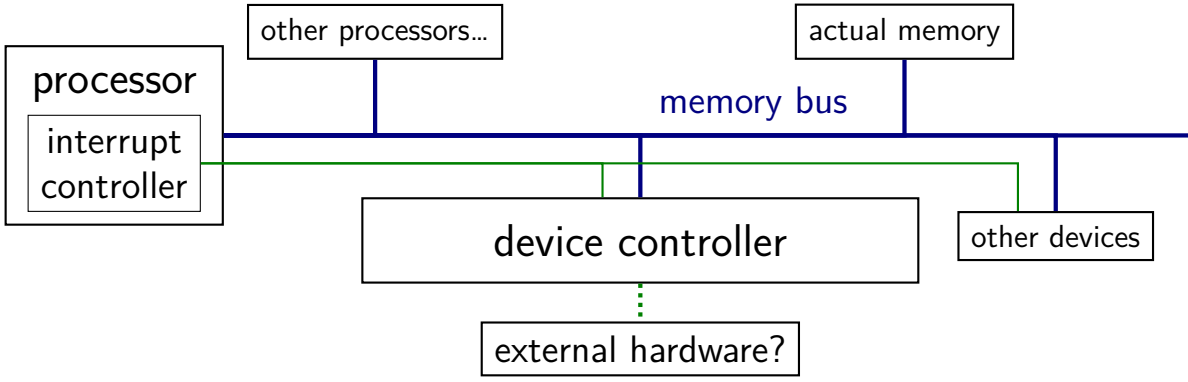
direct memory access (DMA)



observation: devices can read/write memory

can have device copy data to/from memory

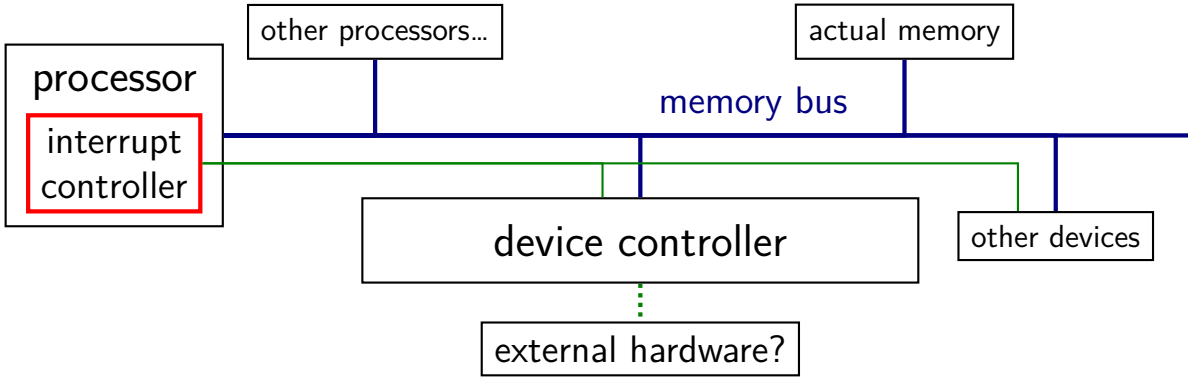
direct memory access (DMA)



observation: devices can read/write memory

can have device copy data to/from memory

direct memory access (DMA)



observation: devices can read/write memory

can have device copy data to/from memory

direct memoyr access (DMA)

observation: devices can read/write memory

can have device copy data to/from memory

much faster, e.g., for disk or network I/O

avoids having processor run a loop

allows device to use memory as very large buffer space

allows device to read/write data as it needs/gets it

direct memory access protocol

store address of buffer *in memory*

OS needs to keep buffer around until device indicates it's done

end of transfer indicated via interrupt + control registers

IOMMUs

typically, direct memory access requires using physical addresses

- devices don't have page tables
- need contiguous physical addresses (multiple pages if buffer >page size)
- devices that messes up can overwrite arbitrary memory

recent systems have an IO Memory Management Unit

- pagetables for devices
- allows non-contiguous buffers
- enforces protection — broken device can't write wrong memory location
- helpful for virtual machines

hard drive interfaces

hard drives and solid state disks are divided into sectors

historically 512 bytes (larger on recent disks)

disk commands:

read from sector i to sector j
write from sector i to sector j this data

typically want to read/write more than sector— 4K+ at a time

filesystems

filesystems: store hierarchy of directories on disk

disk is a flat list of blocks of data

given a file (identified how?), where is its data?

- which sectors? parts of sectors?

given a directory (identified how?), what files are in it?

metadata: names, owner, permissions, size, ...of file

making a new file: where to put it?

making a file/directory bigger: where does new data go?

the FAT filesystem

FAT: File Allocation Table

probably simplest widely used filesystem (family)

named for important data structure: *file allocation table*

FAT and sectors

FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read

cluster: typically 512 to 4096 bytes

a file's data is stored in clusters

reading a file: determine the list of clusters

FAT: the file allocation table

big array on disk, one entry per cluster

each entry contains a number — usually “next cluster”

cluster num.	entry value
0	4
1	7
2	5
3	1434
…	…
1000	4503
1001	1523
…	…

FAT: reading a file (1)

get (from elsewhere) first cluster of data

linked list of cluster numbers

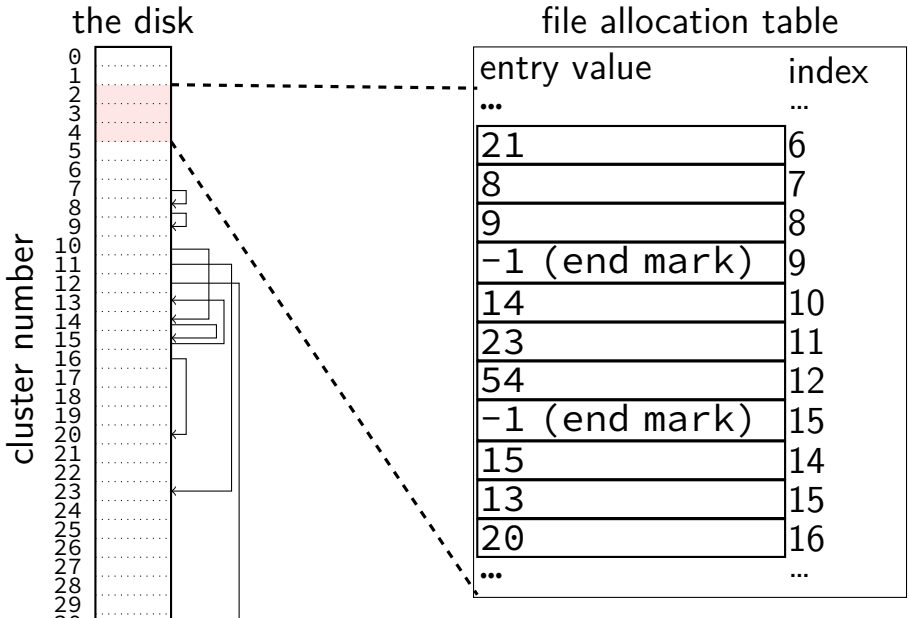
next pointers? file allocation table entry for cluster

special value for NULL

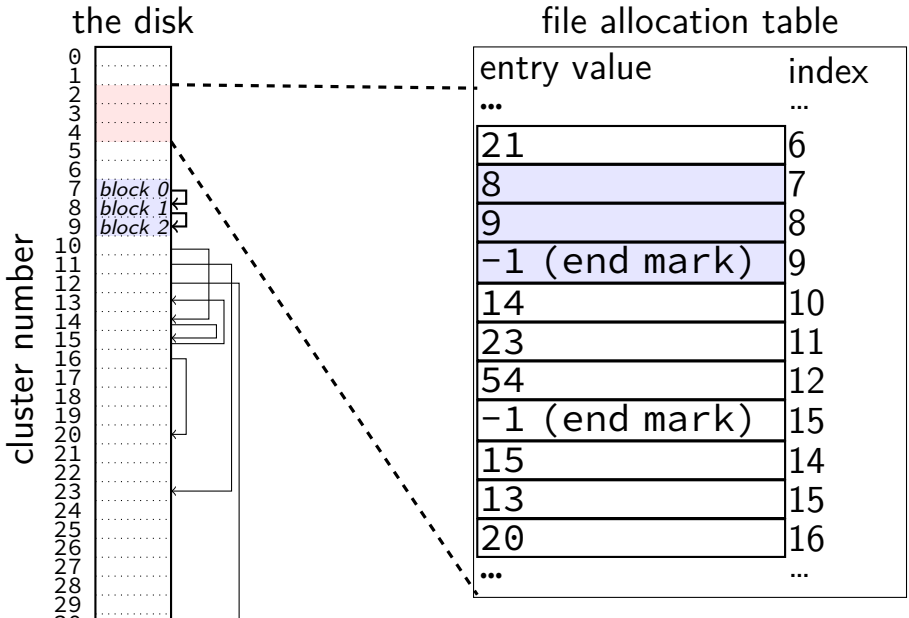
cluster num.	entry value
...	...
10	14
11	23
12	54
13	-1 (end mark)
14	15
15	13
...	...

file starting at cluster 10 contains data i
cluster 10, then 14, then 15, then 13

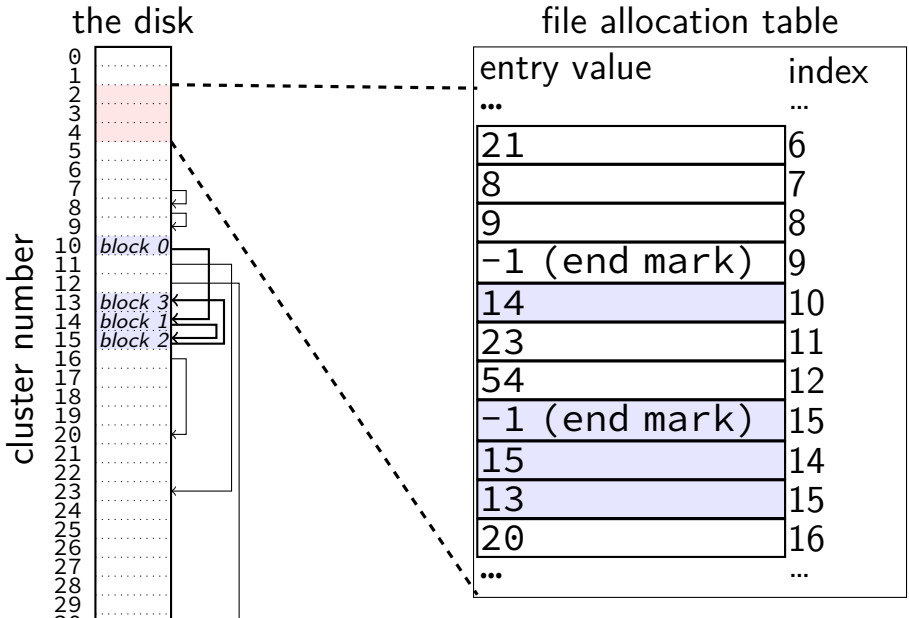
FAT: reading a file (2)



FAT: reading a file (2)



FAT: reading a file (2)



FAT: reading files

to read a file given it's start location

read the starting cluster X

get the next cluster Y from FAT entry X

read the next cluster

get the next cluster from FAT entry Y

...

until you see an end marker

start locations?

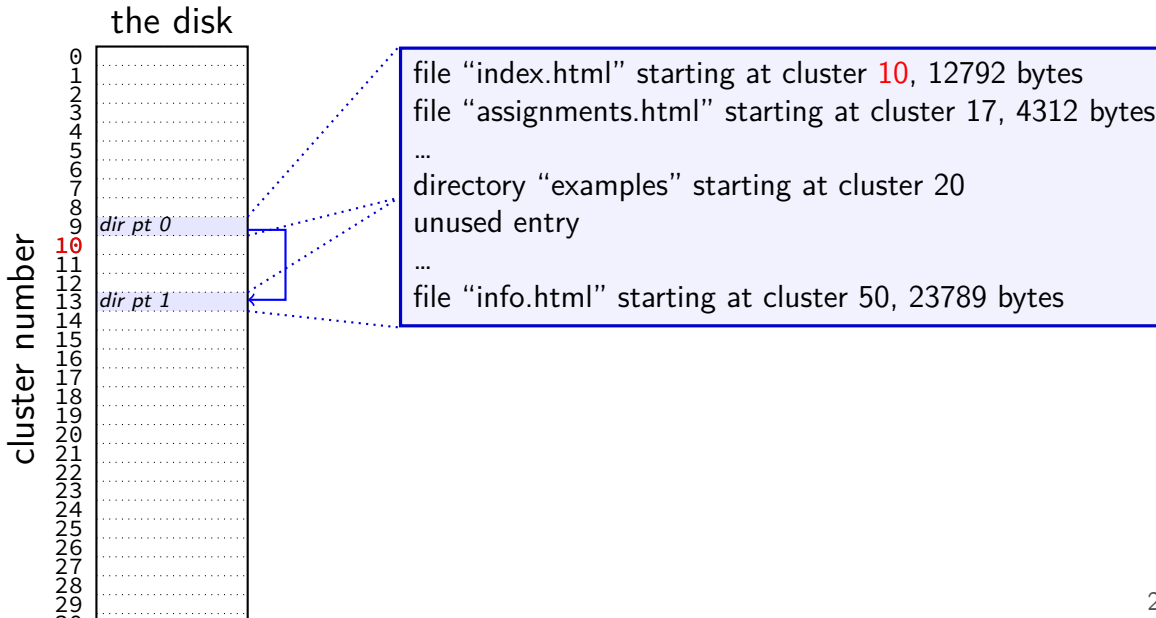
really want filenames

stored in directories!

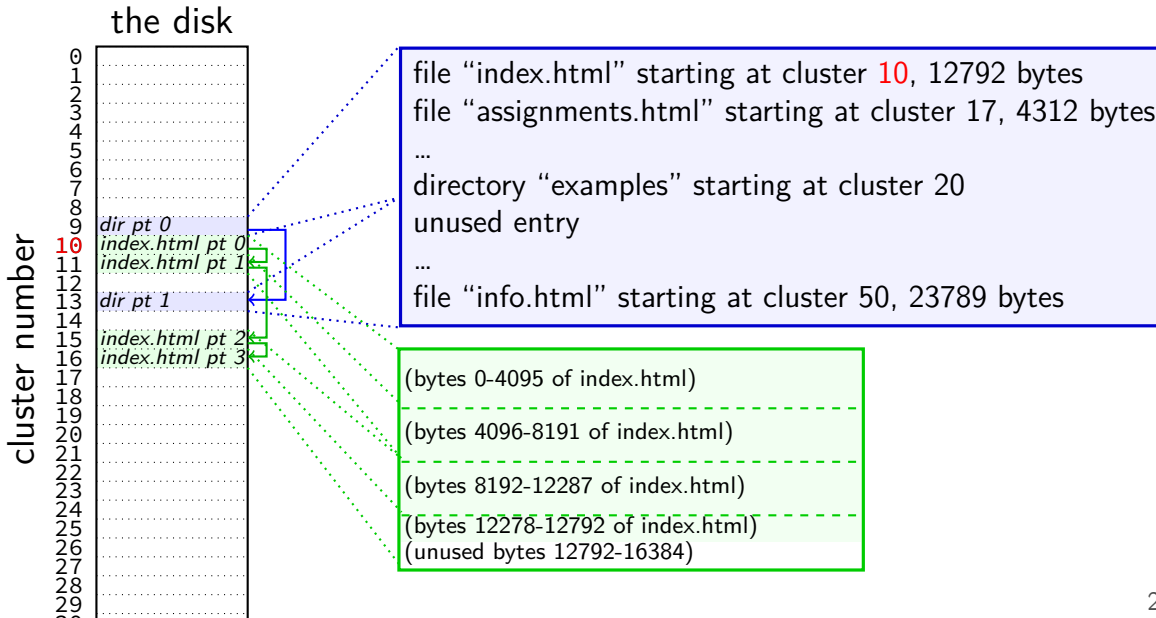
in FAT: directory is a list of:

(name, starting location, other data about file)

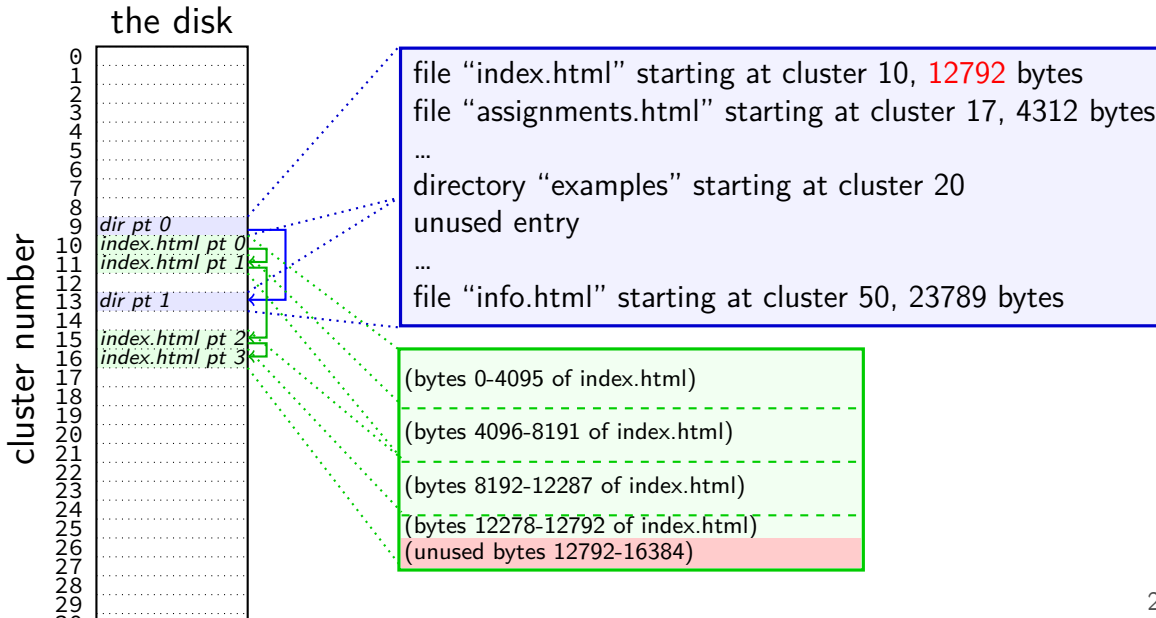
finding files with directory



finding files with directory



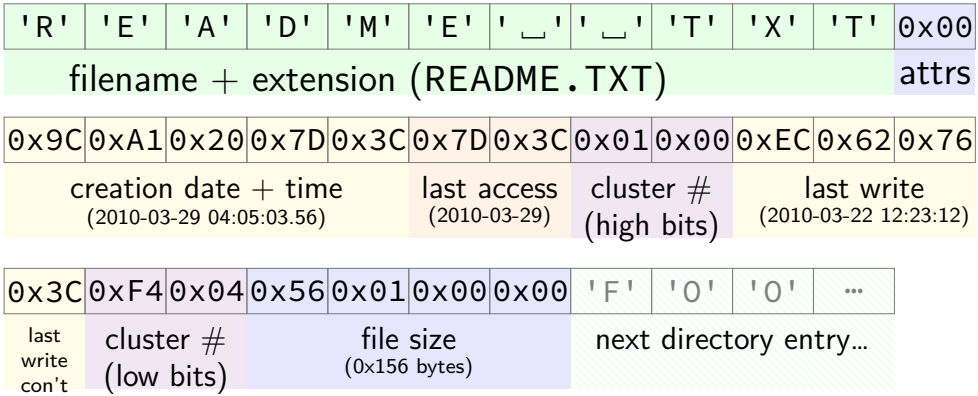
finding files with directory



FAT directory entry

box = 1 byte

entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4

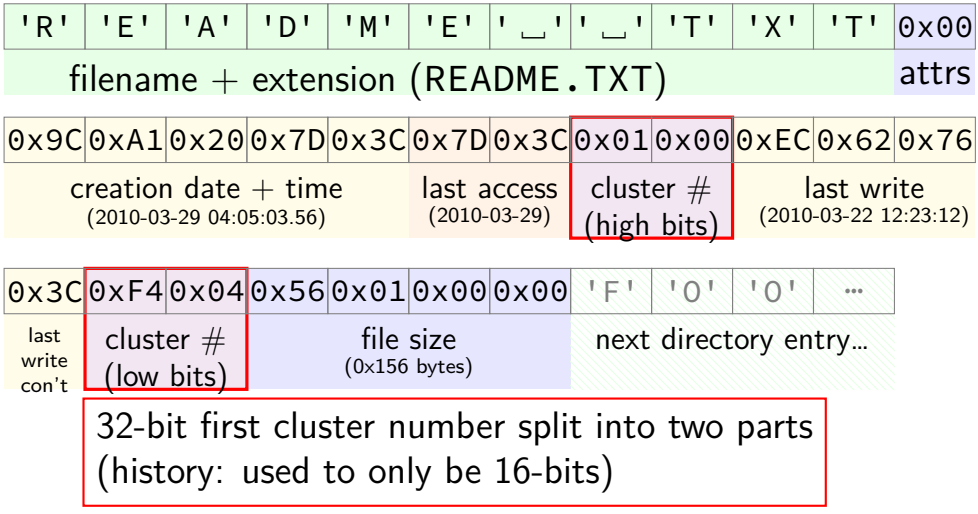


directory?
read-only?
hidden?
...

FAT directory entry

box = 1 byte

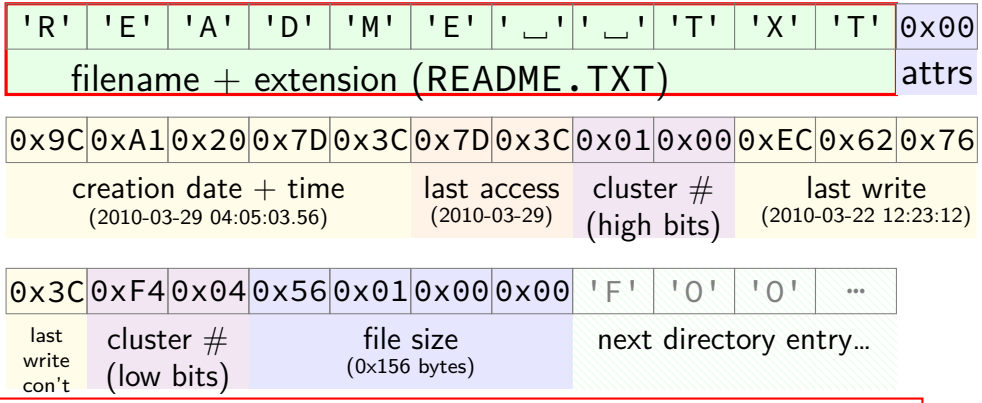
entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4



FAT directory entry

box = 1 byte

entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4

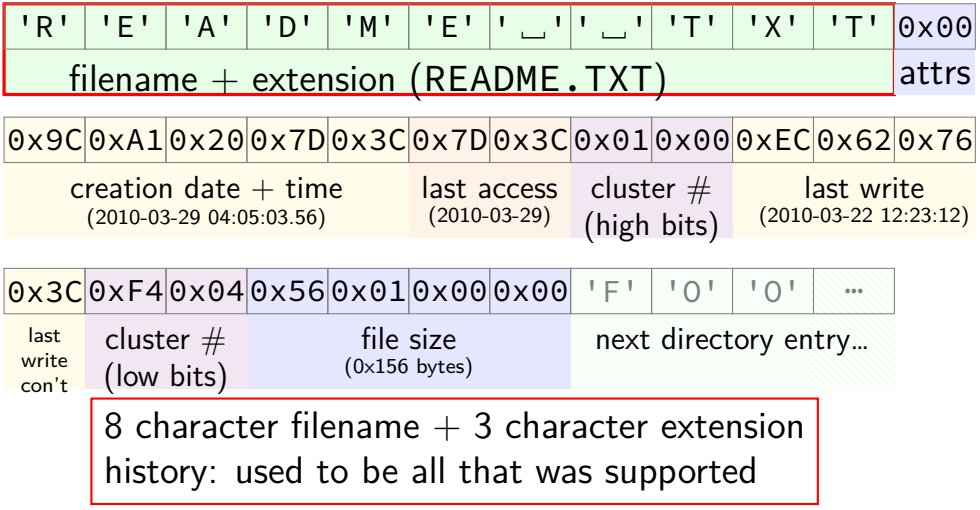


8 character filename + 3 character extension
longer filenames? encoded using extra directory entries
(special `attrs` values to distinguish from normal entries)

FAT directory entry

box = 1 byte

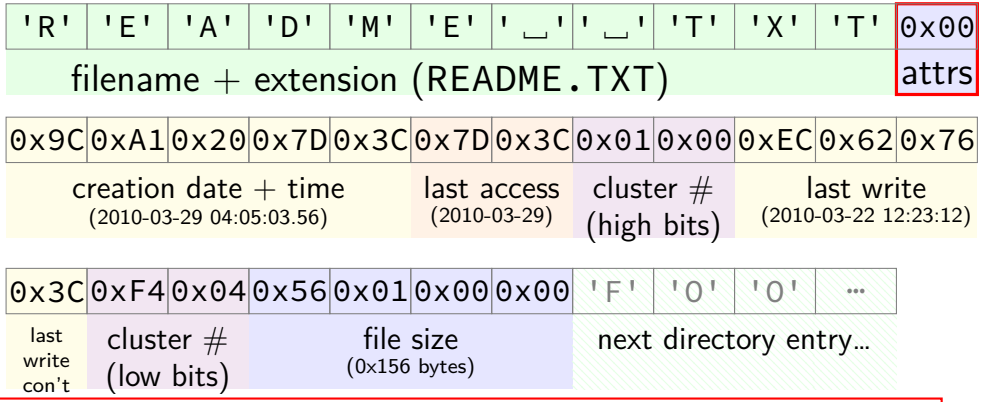
entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4



FAT directory entry

box = 1 byte

entry for `README.TXT`, 342 byte file, starting at cluster 0x104F4

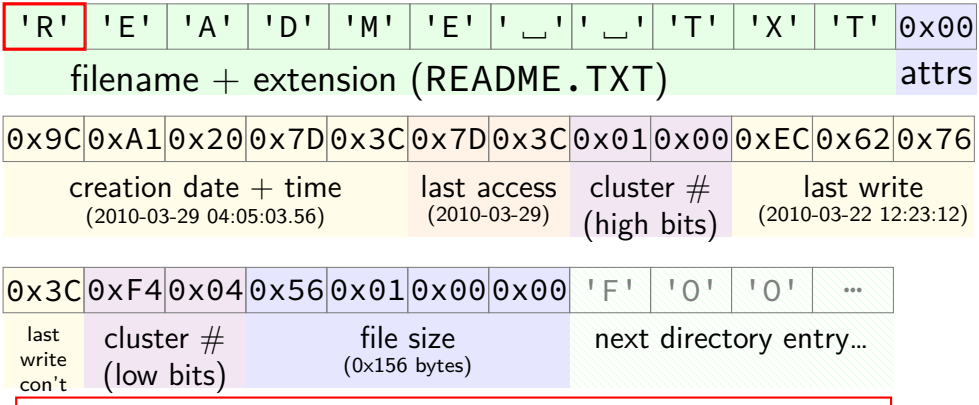


attributes: is a subdirectory, read-only, ...
also marks directory entries used to hold extra filename data

FAT directory entry

box = 1 byte

entry for README.TXT, 342 byte file, starting at cluster 0x104F4



convention: if first character is 0x0 or 0xE5 — unused

0x00: for filling empty space at end of directory

0xE5: 'hole' — e.g. from file deletion

aside: FAT date encoding

seperate date and time fields (16 bits, little-endian integers)

bits 0-4: seconds (divided by 2), 5-10: minute, 11-15: hour

bits 0-4: day, 5-8: month, 9-15: year (minus 1980)

sometimes extra field for 100s(?) of a second

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
    uint8_t DIR_Name[11];             // short name
    uint8_t DIR_Attr;                 // File sttribute
    uint8_t DIR_NTRes;                // Set value to 0, never c
    uint8_t DIR_CrtTimeTenth;         // millisecond timestamp f
    uint16_t DIR_CrtTime;             // time file was created
    uint16_t DIR_CrtDate;             // date file was created
    uint16_t DIR_LstAccDate;          // last access date
    uint16_t DIR_FstClusHI;           // high word fo this entry
    uint16_t DIR_WrtTime;             // time of last write
    uint16_t DIR_WrtDate;             // dat eof last write
    uint16_t DIR_FstClusLO;           // low word of this entry'
    uint32_t DIR_FileSize;            // 32-bit DWORD hoding thi
};
```

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
    uint8_t DIR_Name[11];             // short name
    uint8_t
    uint8_t                                          er c
    uint8_t                                          mp f
    uint16_t                                         ed
    uint16_t DIR_CrtDate;             // date file was created
    uint16_t DIR_LstAccDate;          // last access date
    uint16_t DIR_FstClusHI;           // high word fo this entry
    uint16_t DIR_WrtTime;             // time of last write
    uint16_t DIR_WrtDate;             // dat eof last write
    uint16_t DIR_FstClusLO;           // low word of this entry'
    uint32_t DIR_FileSize;            // 32-bit DWORD hoding thi
};
```

GCC/Clang extension to disable padding

normally compilers add padding to structs

(to avoid splitting values across cache blocks or pages)

FAT directory entries (from C)

```
struct __attribute__(
    uint8_t DIR_Name[
    uint8_t DIR_Attr;
    uint8_t DIR_NTRes
    uint8_t DIR_CrtTi
    uint16_t DIR_CrtTime;          // time file was created
    uint16_t DIR_CrtDate;          // date file was created
    uint16_t DIR_LstAccDate;       // last access date
    uint16_t DIR_FstClusHI;        // high word fo this entry
    uint16_t DIR_WrtTime;          // time of last write
    uint16_t DIR_WrtDate;          // dat eof last write
    uint16_t DIR_FstClusLO;        // low word of this entry'
    uint32_t DIR_FileSize;         // 32-bit DWORD hoding thi
};
```

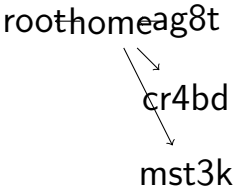
8/16/32-bit unsigned integer
use exact size that's on disk
just copy byte-by-byte from disk to memory
(and everything happens to be little-endian)

FAT directory entries (from C)

```
struct __attribut
    uint8_t DIR_N
    uint8_t DIR_A
    uint8_t DIR_NTRes;                  // Set value to 0, never c
    uint8_t DIR_CrtTimeTenth;           // millisecond timestamp f
    uint16_t DIR_CrtTime;               // time file was created
    uint16_t DIR_CrtDate;               // date file was created
    uint16_t DIR_LstAccDate;            // last access date
    uint16_t DIR_FstClusHI;             // high word fo this entry
    uint16_t DIR_WrtTime;               // time of last write
    uint16_t DIR_WrtDate;               // dat eof last write
    uint16_t DIR_FstClusLO;             // low word of this entry'
    uint32_t DIR_FileSize;              // 32-bit DWORD hoding thi
};
```

why are the names so bad ("FstClusHI", etc.)?
comes from Microsoft's documentation this way

trees of directories



nested directories

foo/bar/baz/file.txt

read root directory entries to find foo

read foo's directory entries to find bar

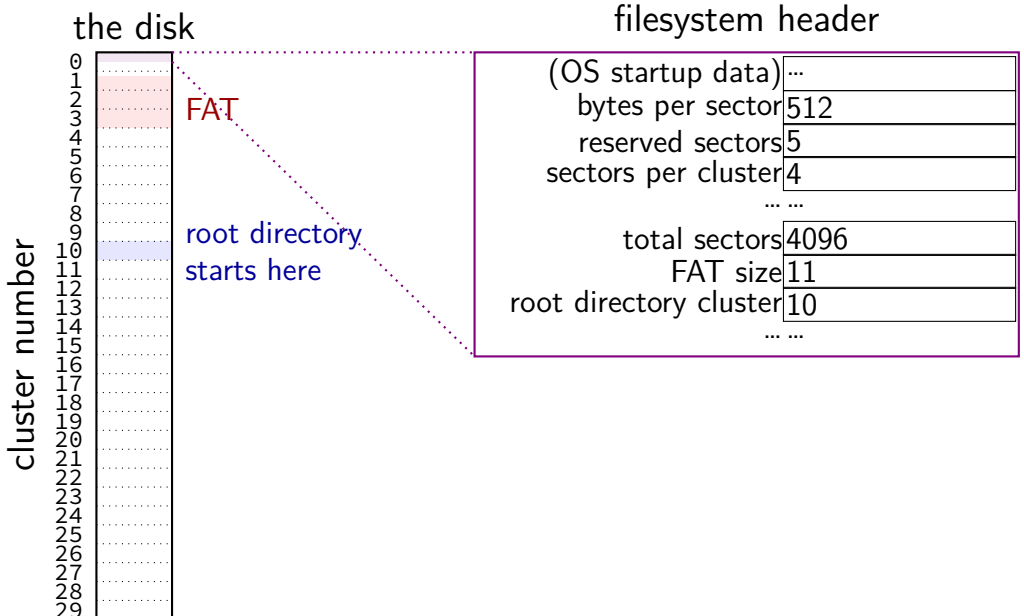
read bar's directory entries to find baz

read baz's directory entries to find file.txt

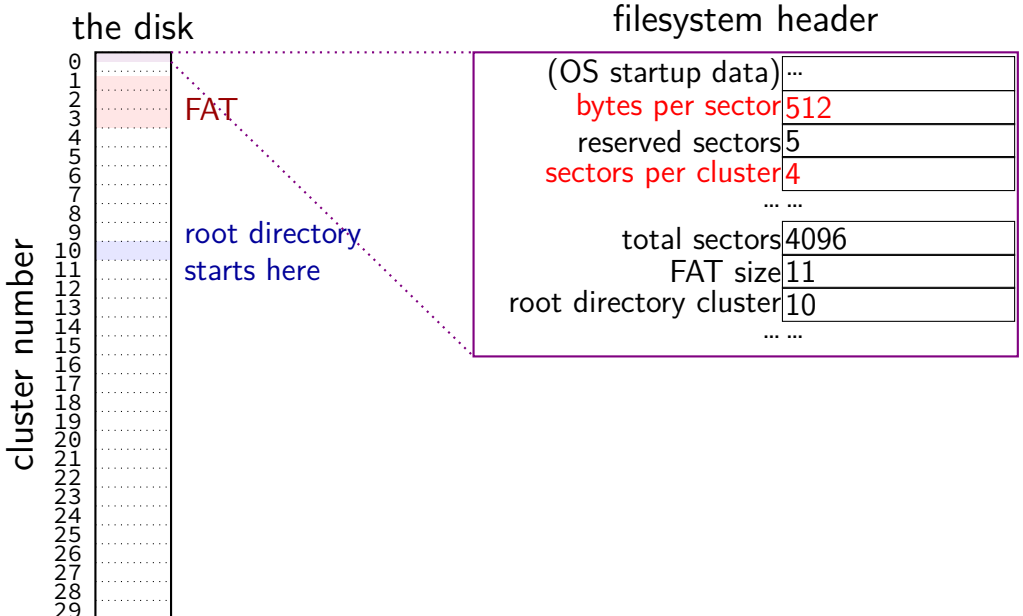
the root directory?

but where is the first directory?

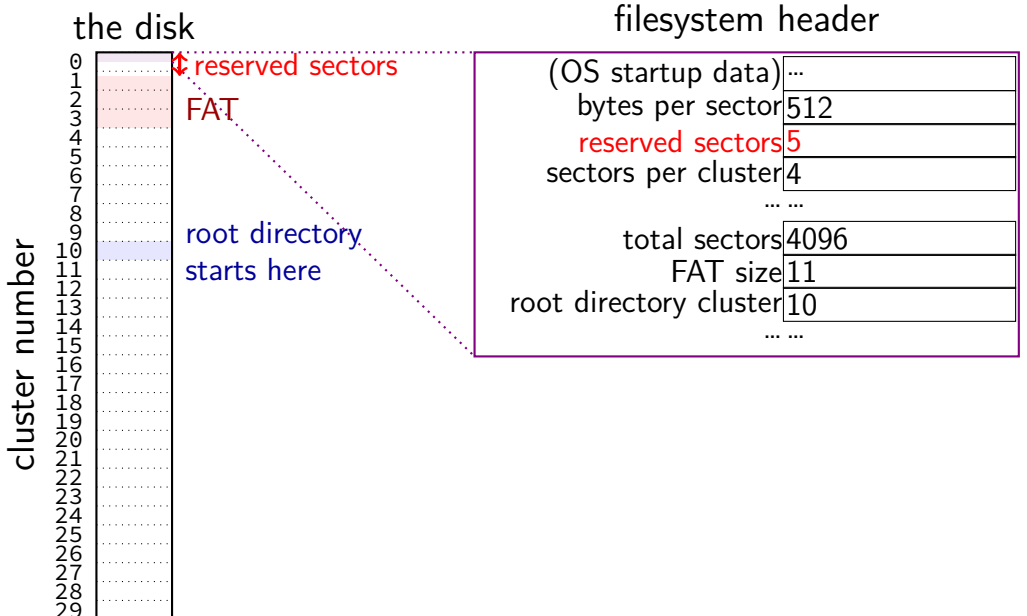
FAT disk header



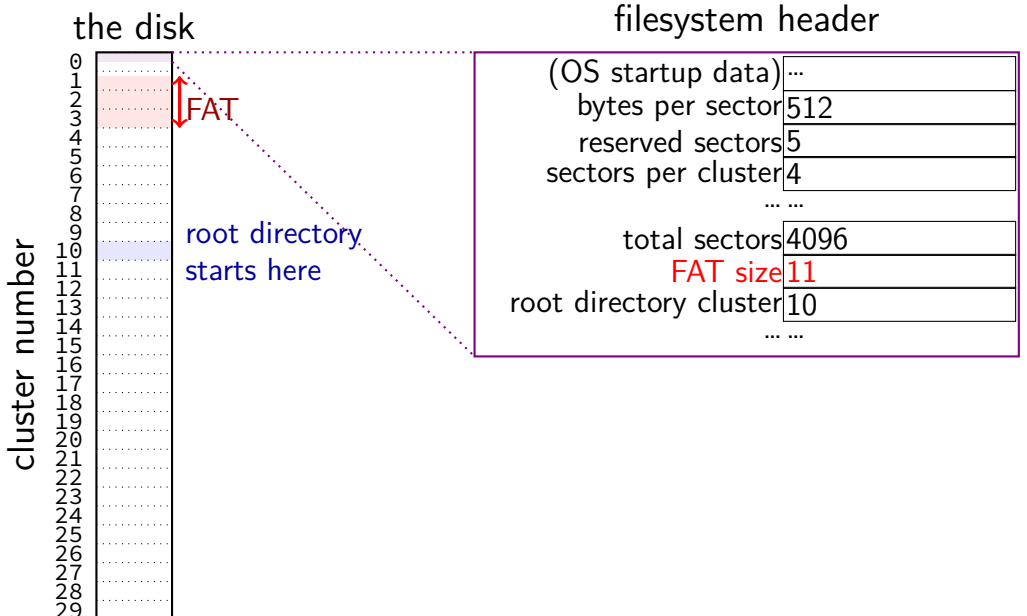
FAT disk header



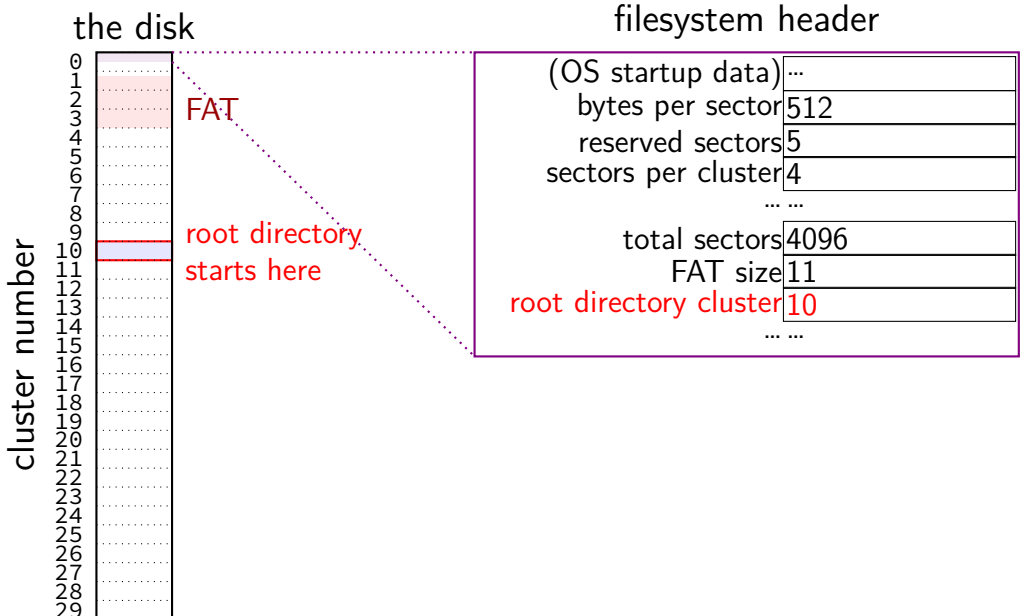
FAT disk header



FAT disk header



FAT disk header



filesystem header

fixed location near beginning of disk

determines size of clusters, etc.

tells where to find FAT, root directory, etc.

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
        uint8_t BS_jmpBoot[3];          // jmp instr to boot code
        uint8_t BS_oemName[8];          // indicates what system for
        uint16_t BPB_BytsPerSec;       // Count of bytes per sector
        uint8_t BPB_SecPerClus;         // no.of sectors per allocat
        uint16_t BPB_RsvdSecCnt;        // no.of reserved sectors in
        uint8_t BPB_NumFATs;            // The count of FAT datastru
        uint16_t BPB_rootEntCnt;        // Count of 32-byte entries
        uint16_t BPB_totSec16;          // total sectors on the volu
        uint8_t BPB_media;              // value of fixed media
....
```

FAT: creating a file

add a directory entry

choose clusters to store file data (how???)

update FAT to link clusters together

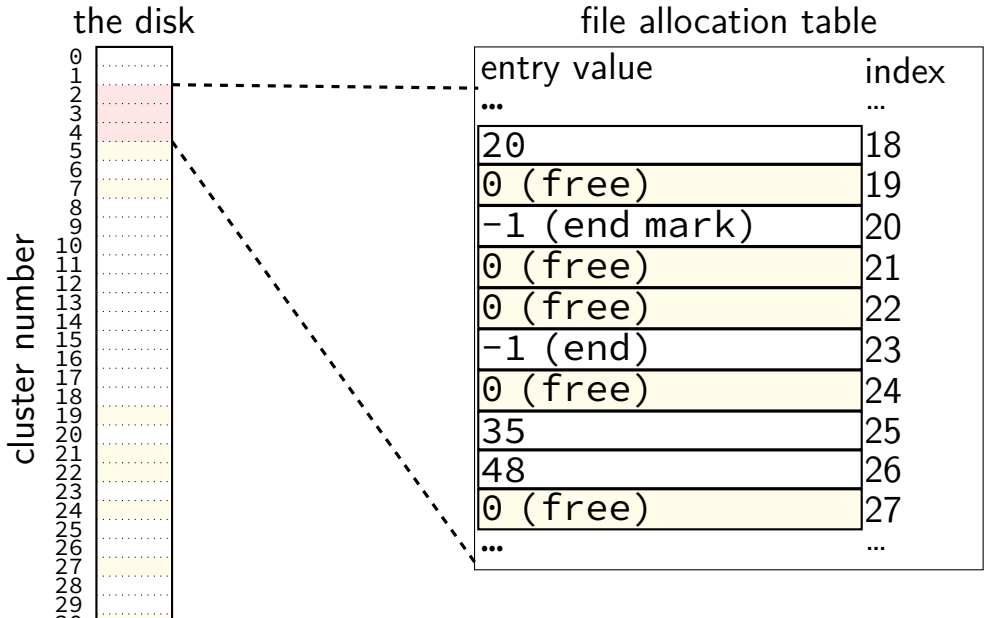
FAT: creating a file

add a directory entry

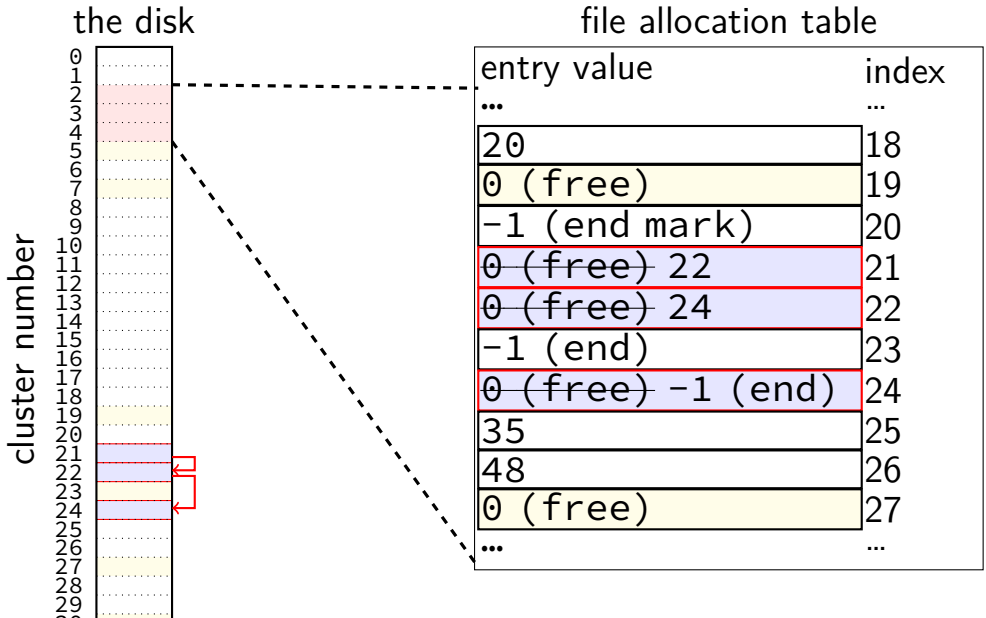
choose clusters to store file data (how???)

update FAT to link clusters together

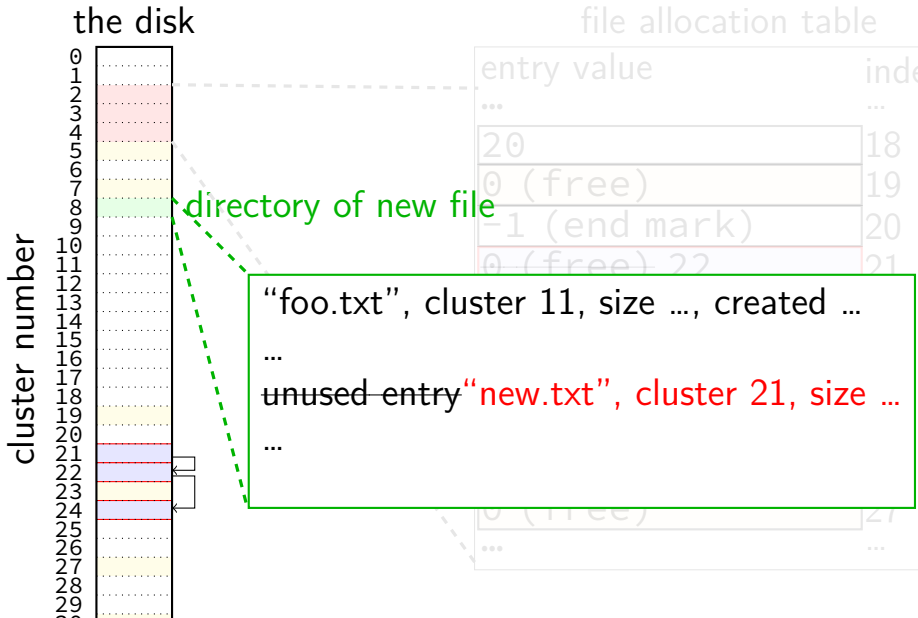
FAT: free clusters



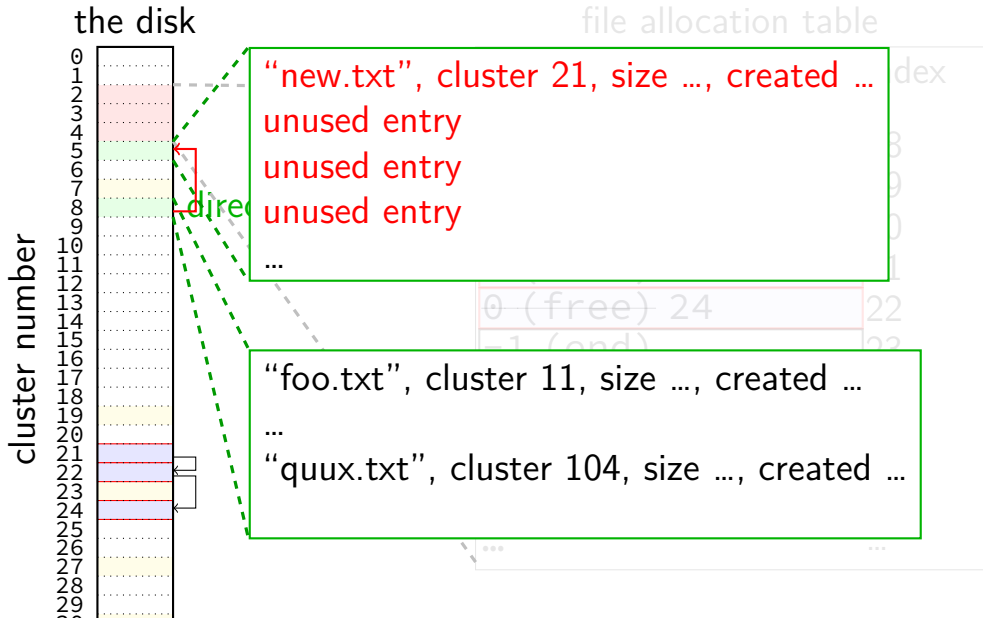
FAT: writing file data



FAT: replacing unused directory entry



FAT: extending directory



FAT: deleting files

reset FAT entries for file clusters to free (0)

write “unused” character in filename for directory entry

- maybe rewrite directory if that’ll save space?

FAT pros and cons?

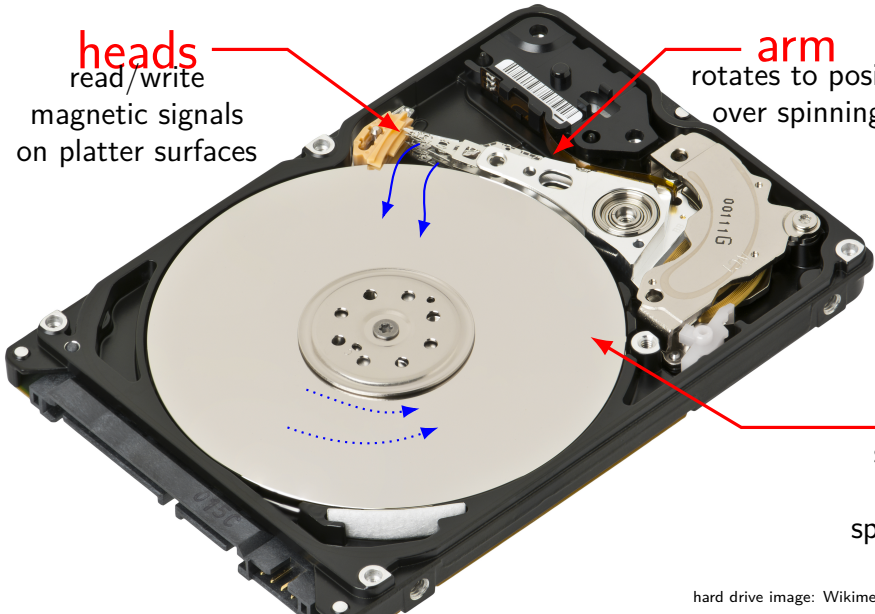
why hard drives?

what filesystems were designed for

currently most cost-effective way to have a lot of online storage

solid state drives (SSDs) imitate hard drive interfaces

hard drives



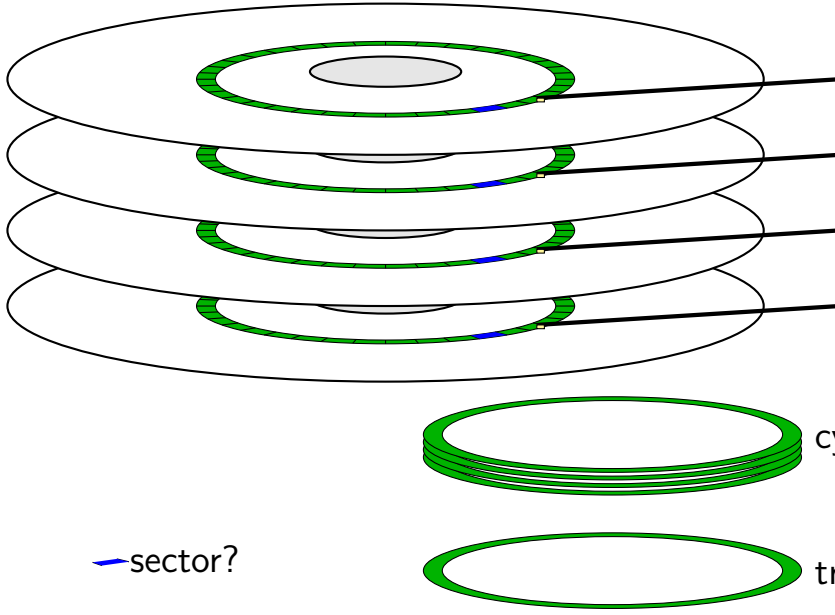
heads
read/write
magnetic signals
on platter surfaces

arm
rotates to position heads
over spinning platters

platters
stack of flat discs
(only top visible)
spins when operating

hard drive image: Wikimedia Commons / Evan-Amos

sectors/cylinders/etc.

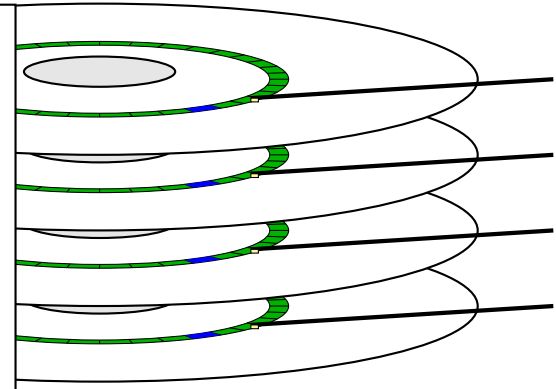


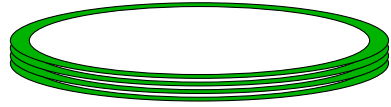
sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

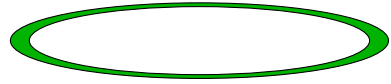
rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data





sector?

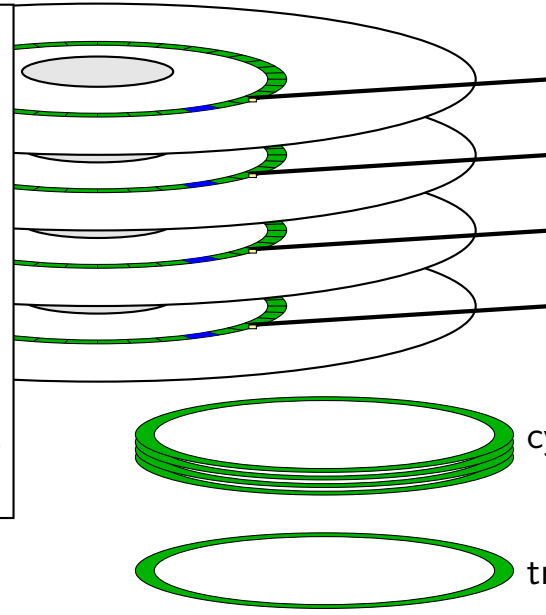


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



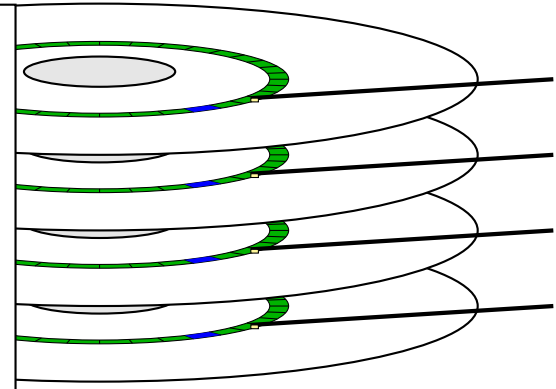
sector?

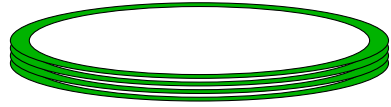
sectors/cylinders/etc.

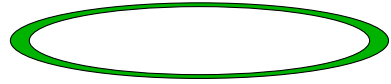
seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data







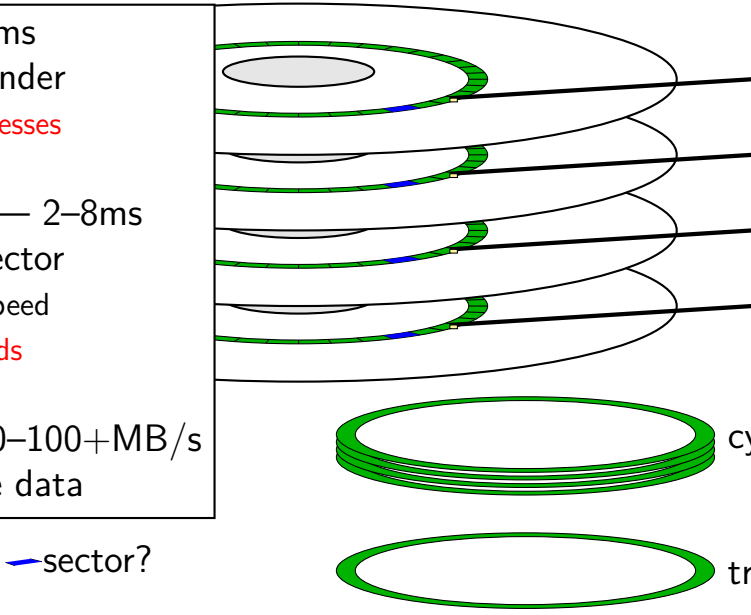
sector?

sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



POSIX: everything is a file

the file: one interface for

- devices (terminals, printers, ...)
- regular files on disk
- networking (sockets)
- local interprocess communication (pipes, sockets)

basic operations: open(), read(), write(), close()

the file interface

open before use

- setup, access control happens here

byte-oriented

- real device isn't? operating system needs to hide that

explicit close

the file interface

open before use

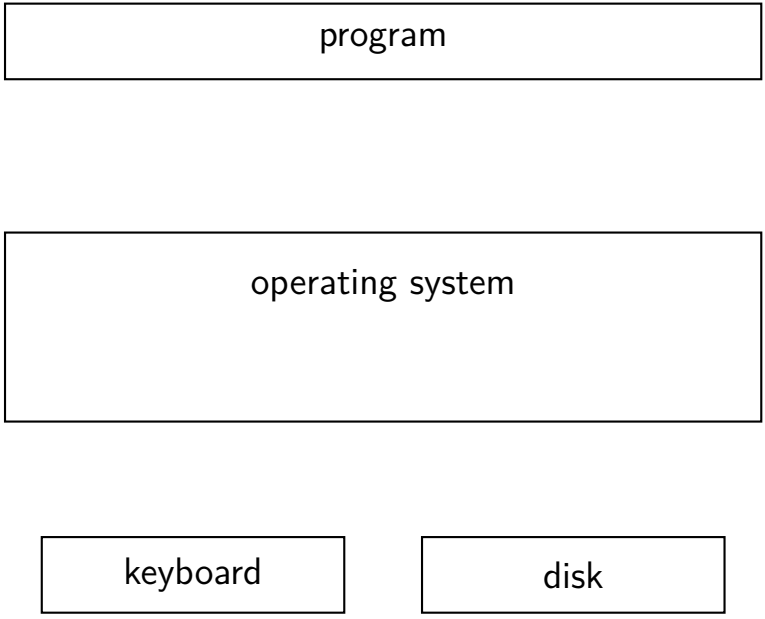
- setup, access control happens here

byte-oriented

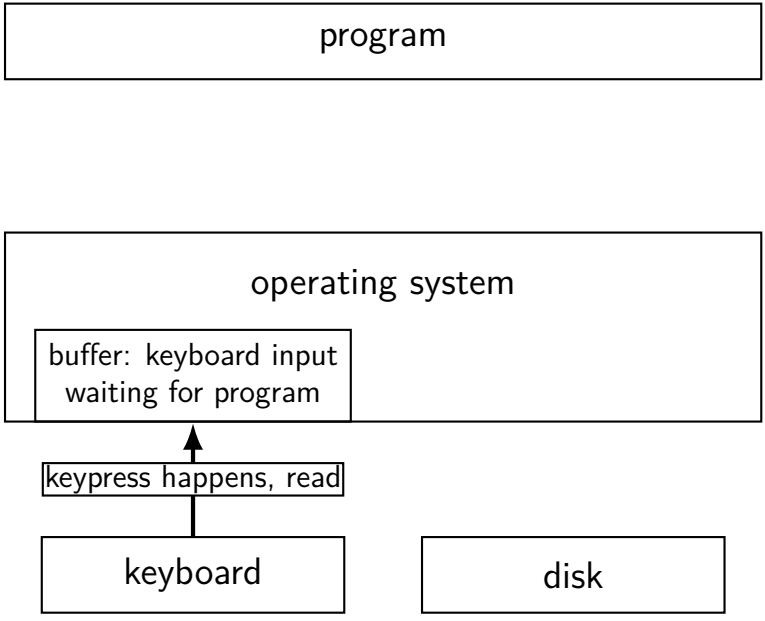
- real device isn't? operating system needs to hide that

explicit close

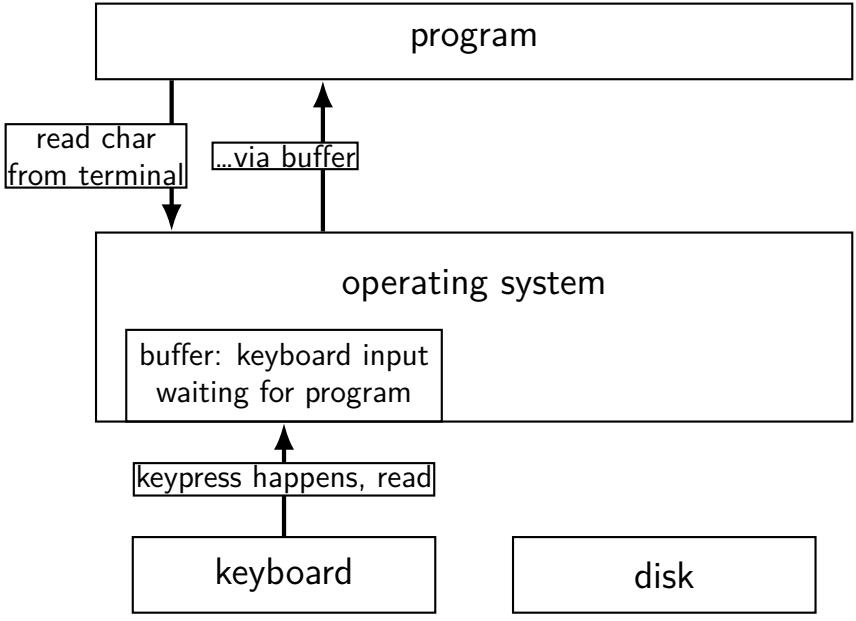
kernel buffering (reads)



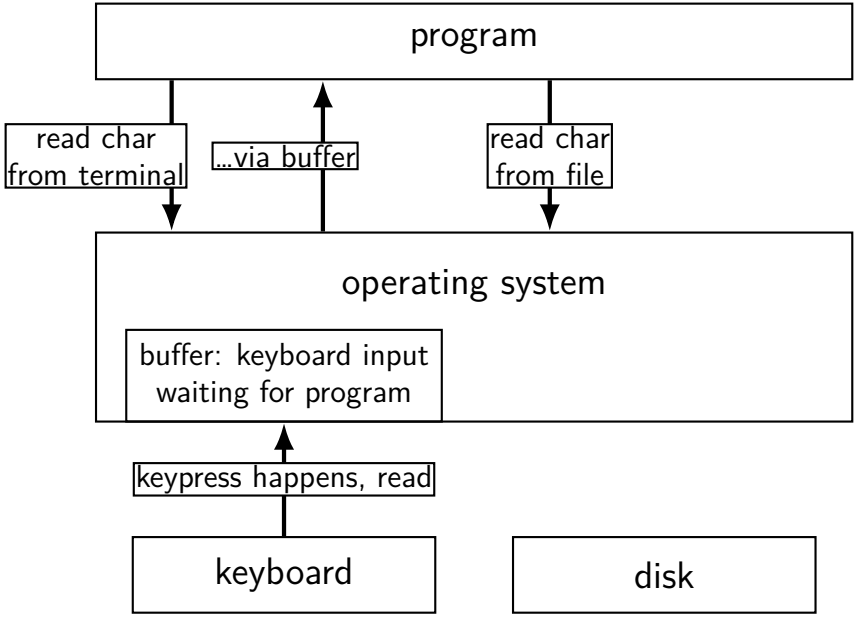
kernel buffering (reads)



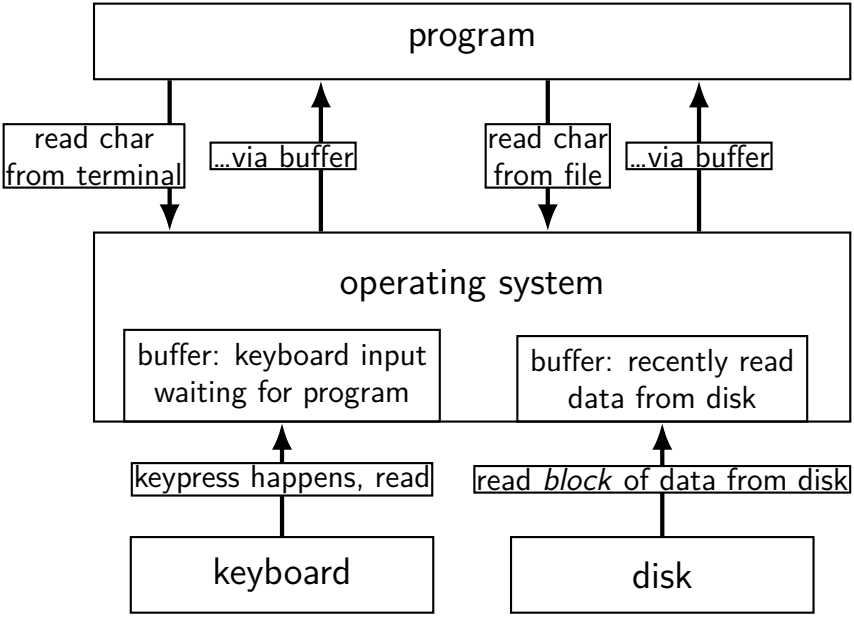
kernel buffering (reads)



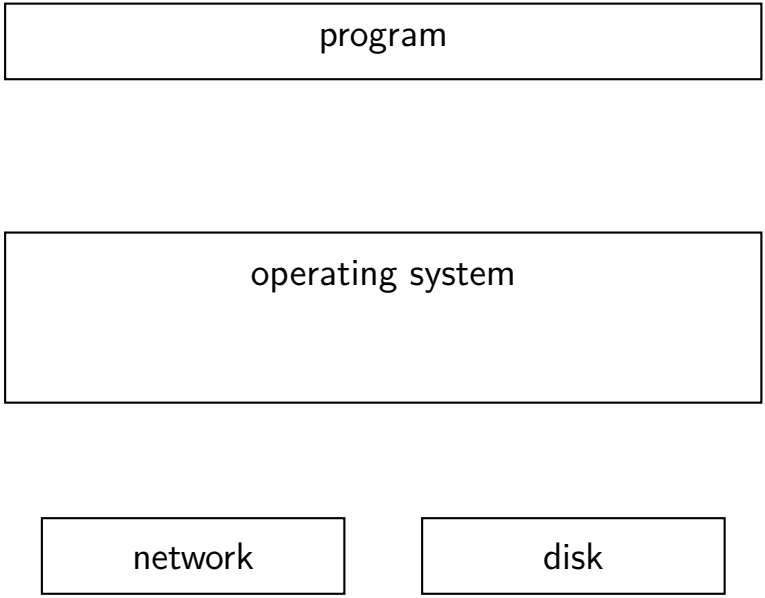
kernel buffering (reads)



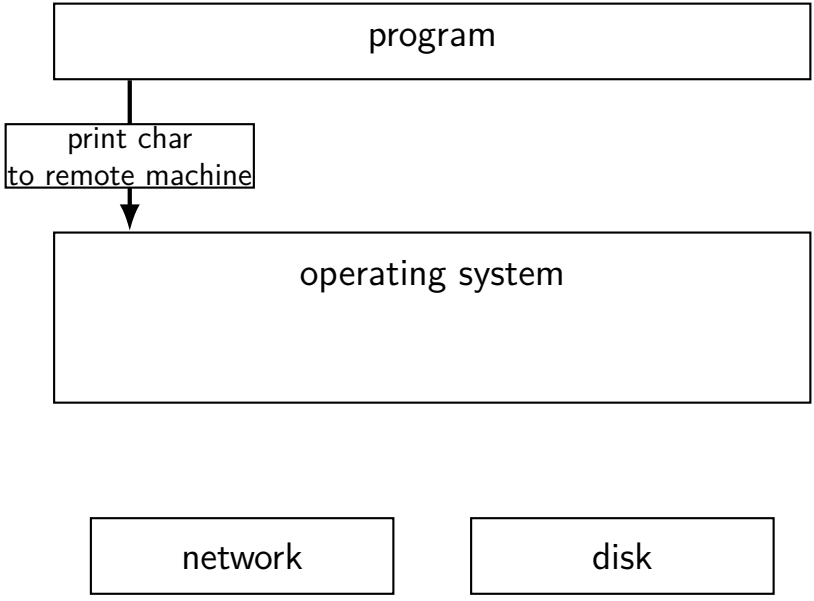
kernel buffering (reads)



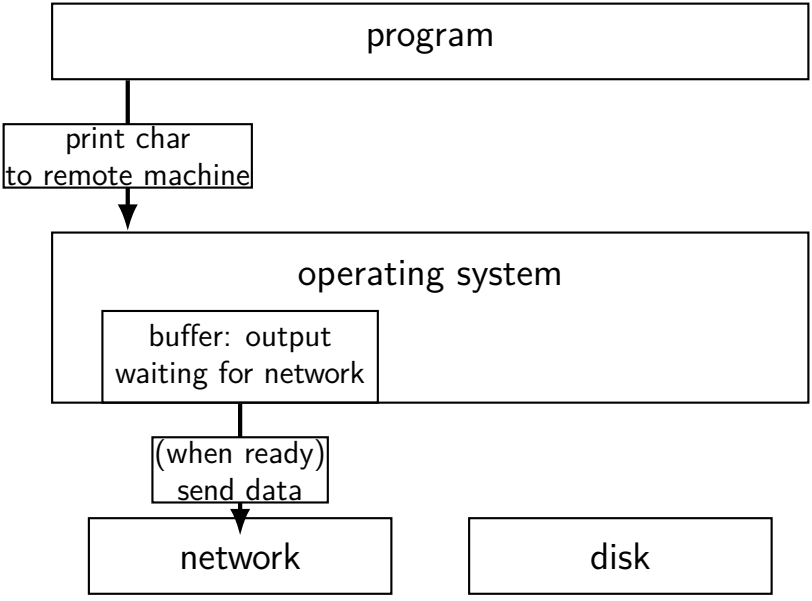
kernel buffering (writes)



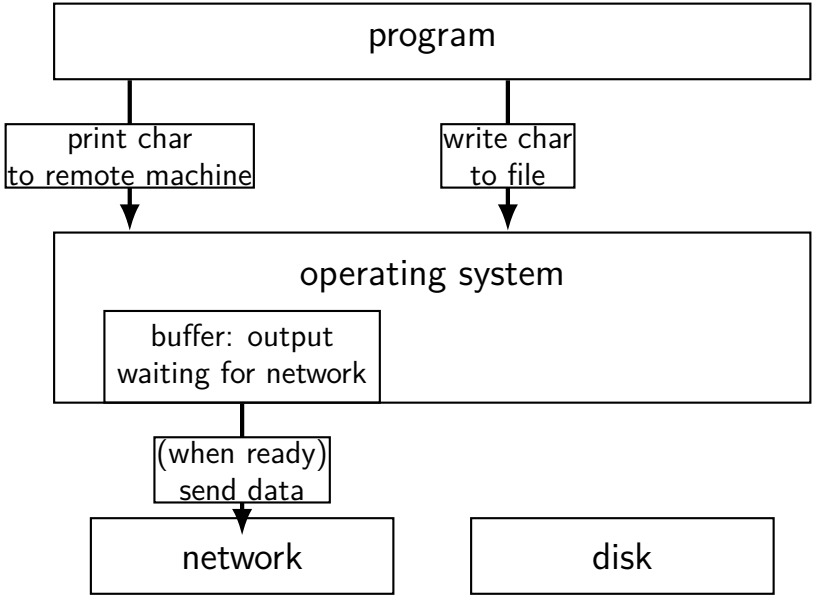
kernel buffering (writes)



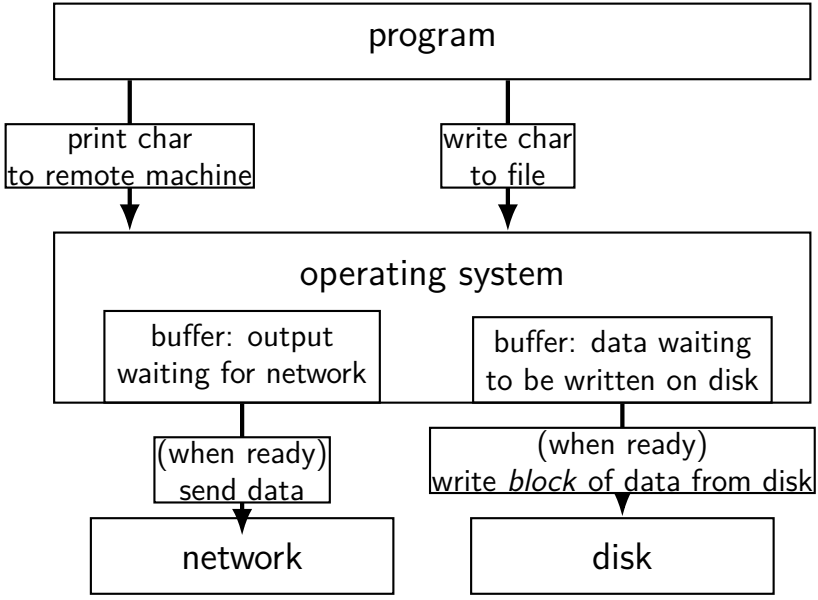
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

read/write: move data into/out of buffer

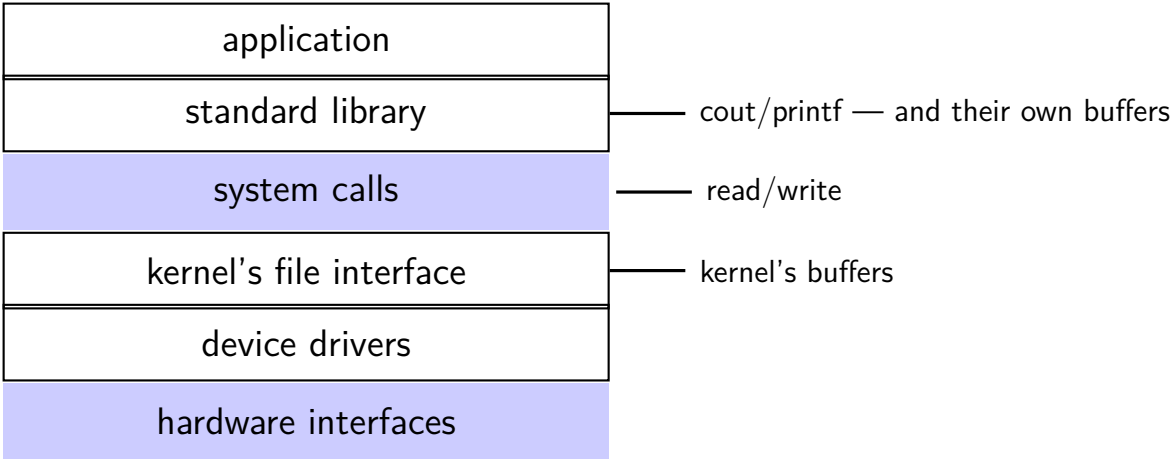
block (make process wait) if buffer is empty (read)/full (write)

(default behavior, possibly changeable)

actual I/O operations — wait for device to be ready

trigger process to stop waiting if needed

layering


application
standard library
cout/printf — and their own buffers
system calls
read/write
kernel's file interface
kernel's buffers
device drivers
hardware interfaces