devices / filesystems (start)

last time

practical LRU approximations

- second chance
- SEQ: active/inactive list
- CLOCK algorithms generally (scanning accessed bits)

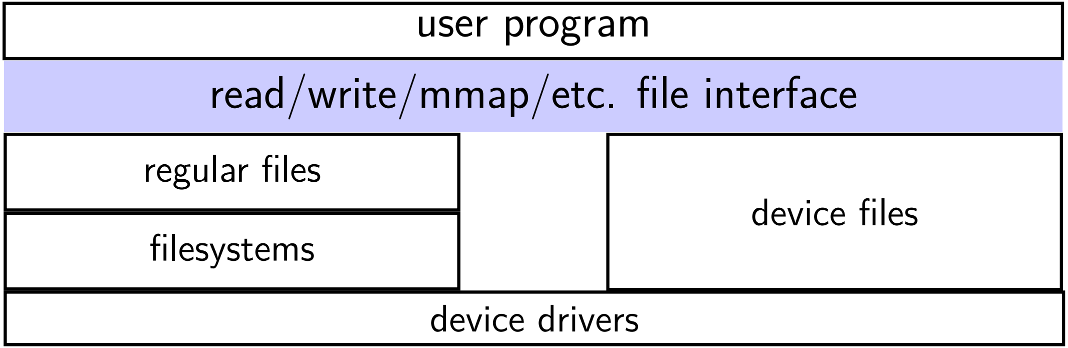
being proactive

- writeback in advance
- readahead
- maintaining little list of pre-evicted pages

recall: buffers in the kernel

device files

ways to talk to I/O devices



devices as files

talking to device? open/read/write/close

typically similar interface within the kernel

device driver implements the file interface

example device files from a Linux desktop

`/dev/snd/pcmC0D0p` — audio playback

- configure, then write audio data

`/dev/sda`, `/dev/sdb` — SATA-based SSD and hard drive

- usually access via filesystem, but can mmap/read/write directly

`/dev/input/event3`, `/dev/input/event10` — mouse and keyboard

- can read list of keypress/mouse movement/etc. events

`/dev/dri/renderD128` — builtin graphics

- DRI = direct rendering infrastructure

devices: extra operations?

read/write/mmap not enough?

- audio output device — set format of audio? headphones plugged in?
- terminal — whether to echo back what user types?
- CD/DVD — open the disk tray? is a disk present?
- …

extra POSIX file descriptor operations:

- ioctl (general I/O control) — device driver-specific interface
- tcsetattr (for terminal settings)
- fcntl
- …

also possibly extra device files for same device:

- `/dev/snd/controlC0` to configure audio settings for `/dev/snd/pcmC0D0p`, `/dev/snd/pcmC0D10p`, …

Linux example: file operations

(selected subset — table of pointers to functions)

```
struct file_operations {
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)
    ssize_t (*write) (struct file *, const char __user *,x
                      size_t, loff_t *);
    ...
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned lo
    ...
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    ...
    int (*release) (struct inode *, struct file *);
    ...
};
```

special case: block devices

devices like disks often have a different interface

unlike normal file interface, works in terms of ‘blocks’

- block size usually equal to page size

for working with page cache

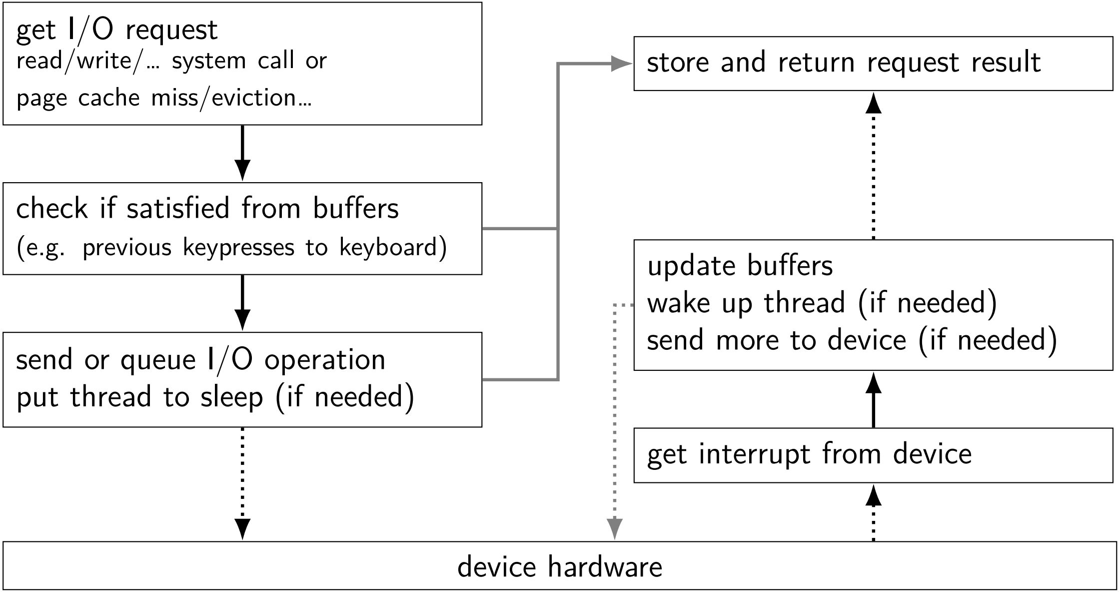
- read/write page at a time

Linux example: block device operations

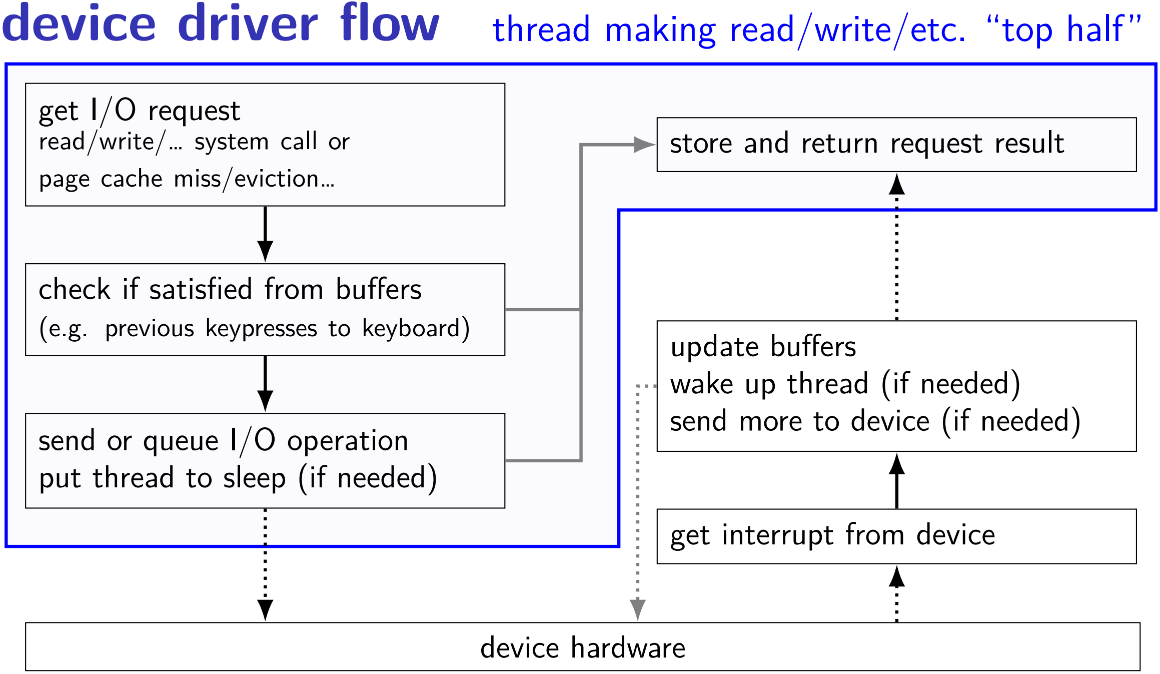
```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *,
            sector_t, struct page *, bool);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, un
    ...
};
```

read/write a page for a sector number (= block number)

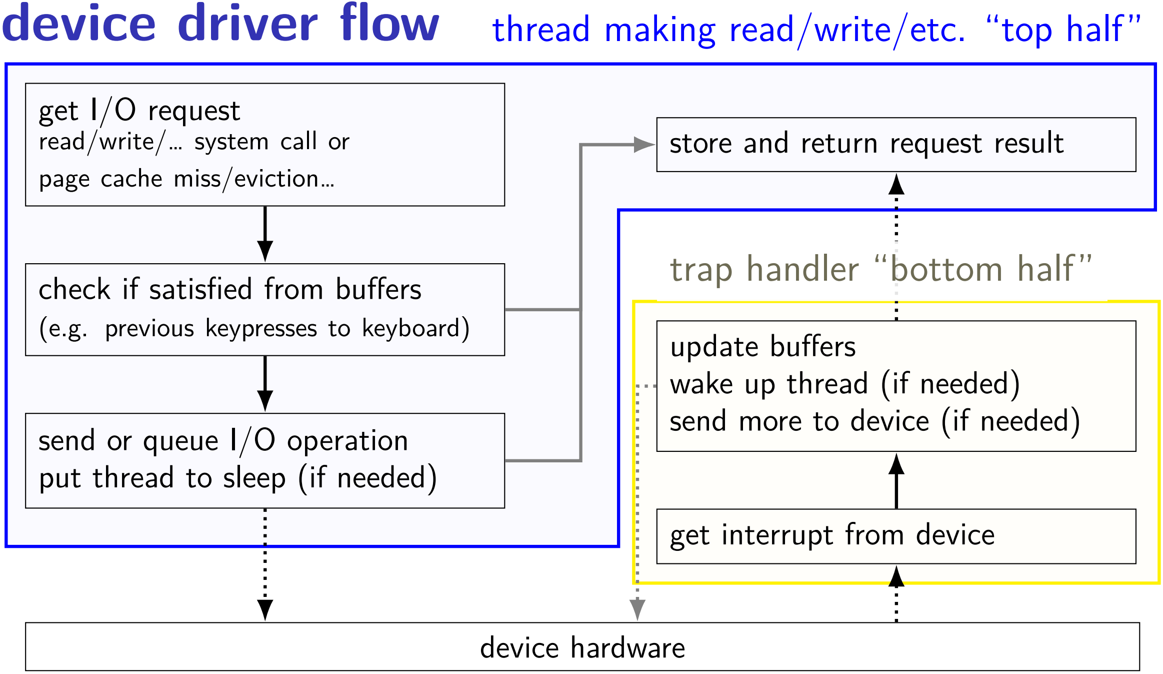
device driver flow



device driver flow



device driver flow



xv6: device files (1)

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};

extern struct devsw devsw[];
```

inode = represents file on disk

pointed to by struct file referenced by fd

xv6: device files (2)

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};
```

```
extern struct devsw devsw[];
```

array of types of devices

special type of file on disk has index into array
- "device number"
- created via `mknod()` system call

similar scheme used on real Unix/Linux
- two numbers: major + minor device number

xv6: console devsw

code run at boot:

```
devsw[CONSOLE].write = consolewrite;
devsw[CONSOLE].read = consoleread;
```

CONSOLE is the constant 1

xv6: console devsw

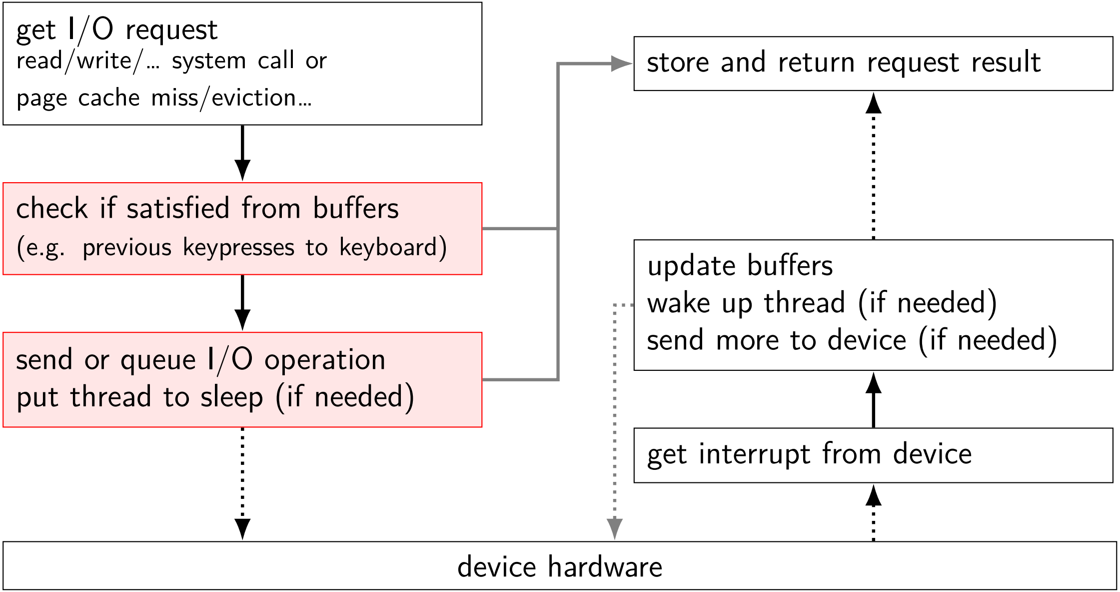
code run at boot:

```
devsw[CONSOLE].write = consolewrite;
devsw[CONSOLE].read = consoleread;
```

CONSOLE is the constant 1

consoleread/consolewrite: run when you read/write console

device driver flow



xv6: console top half (read)

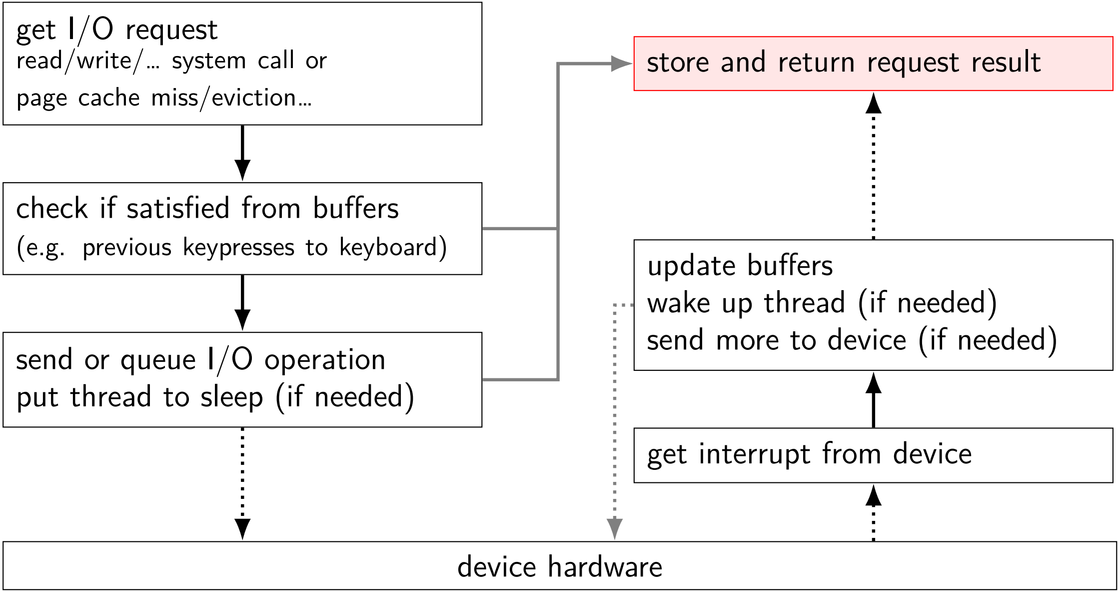
```
int
consoleread(struct inode *ip, char *dst, int n)
{
  ...
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    while(input.r == input.w){
      if(myproc()->killed){
        ...
        return -1;
      }
      sleep(&input.r, &cons.lock);
    }
    ...
  }
  release(&cons.lock)
  ...
}
```

if at end of buffer

r = reading location, w = writing location

put thread to sleep

device driver flow



xv6: console top half (read)

```
int
consoleread(struct inode *ip, char *dst, int n)
{
  ...
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    ...
    c = input.buf[input.r++ % INPUT_B
    ...
    *dst++ = c;
    --n;
    if (c == '\n')
      break;
  }
  release(&cons.lock)
  ...
  return target - n;
}
```

copy from kernel buffer to user buffer (passed to read)

xv6: console top half (read)

```
int
consoleread(struct inode *ip, char *dst, int n)
{
  ...
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    ...
    c = input.buf[input.r++ % INPUT_B
    ...
    *dst++ = c;
    --n;
    if (c == '\n')
      break;
  }
  release(&cons.lock)
  ...
  return target - n;
}
```

copy from kernel buffer
to user buffer (passed to read)

xv6: console top half

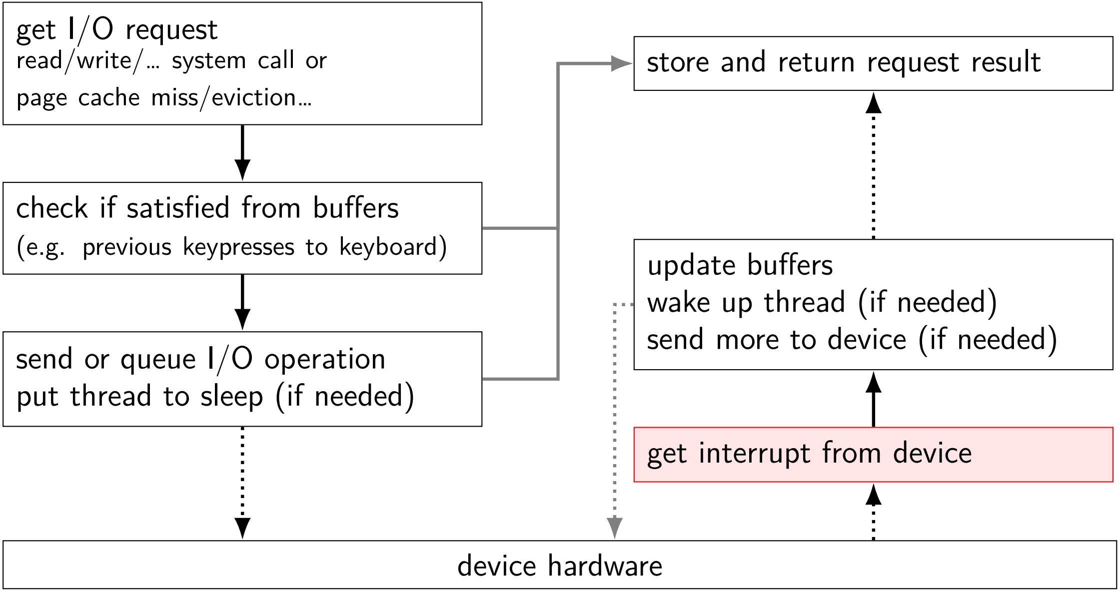
wait for buffer to fill

- no special work to request data — keyboard input always sent

copy from buffer

check if done (newline or enough chars), if not repeat

device driver flow



xv6: console interrupt (one case)

```
void
trap(struct trapframe *tf) {
  ...
  switch(tf->trapno) {
    ...
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapcieoi();
    break;
    ...
  }
  ...
}
```

kbdintr: actually read from keyboard device

lapcieoi: tell CPU "I'm done with this interrupt"

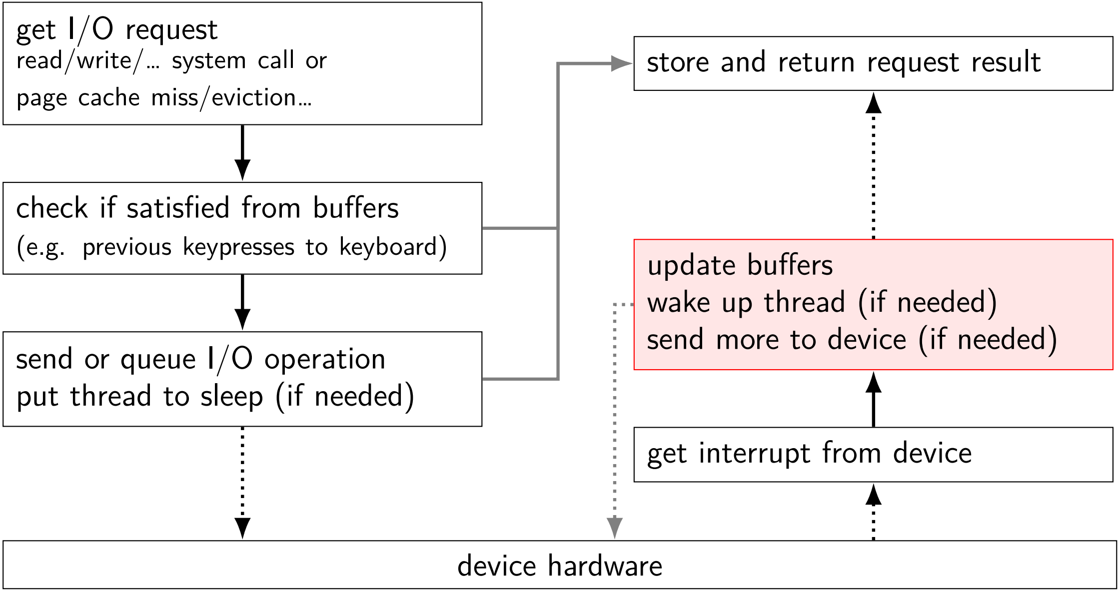
xv6: console interrupt (one case)

```
void
trap(struct trapframe *tf) {
  ...
  switch(tf->trapno) {
    ...
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapcieoi();
    break;
    ...
  }
  ...
}
```

kbdintr: actually read from keyboard device

lapcieoi: tell CPU “I’m done with this interrupt”

device driver flow



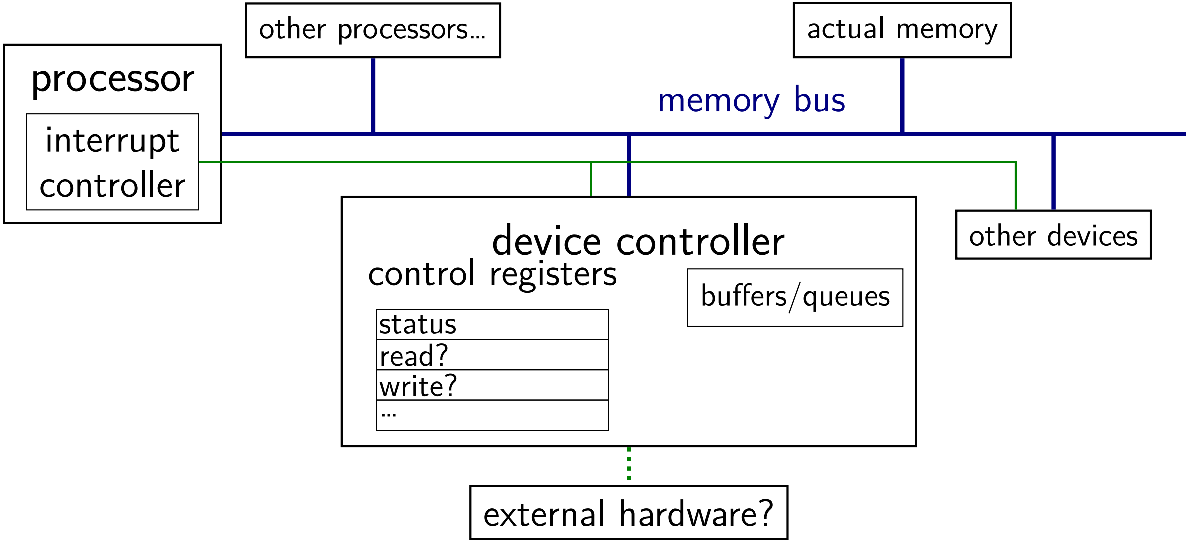
xv6: console interrupt reading

kbdintr fuction actually reads from device

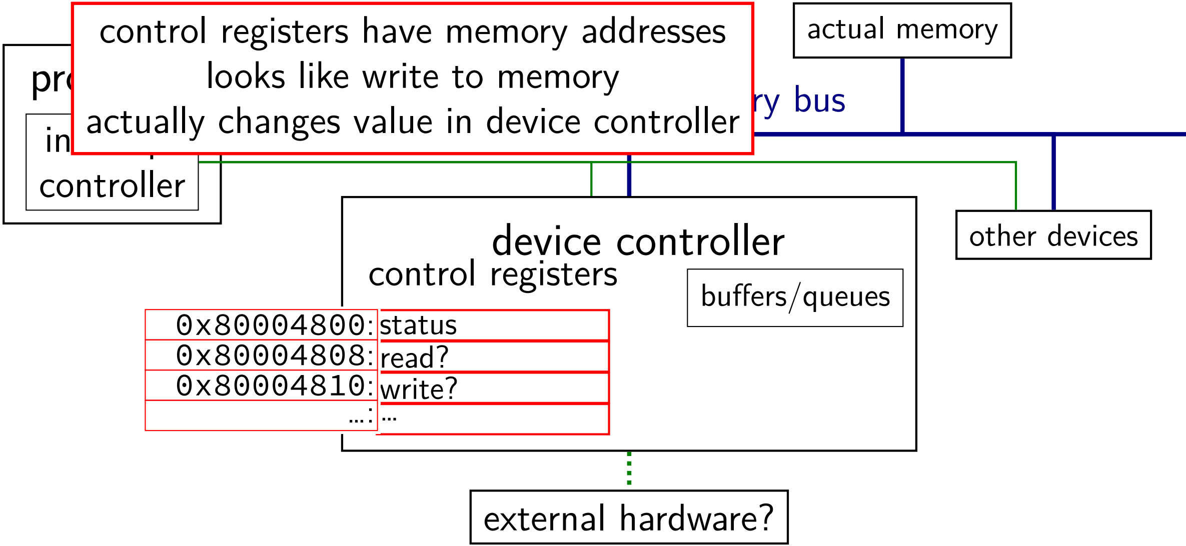
adds data to buffer (if room)

wakes up sleeping thread (if any)

connecting devices

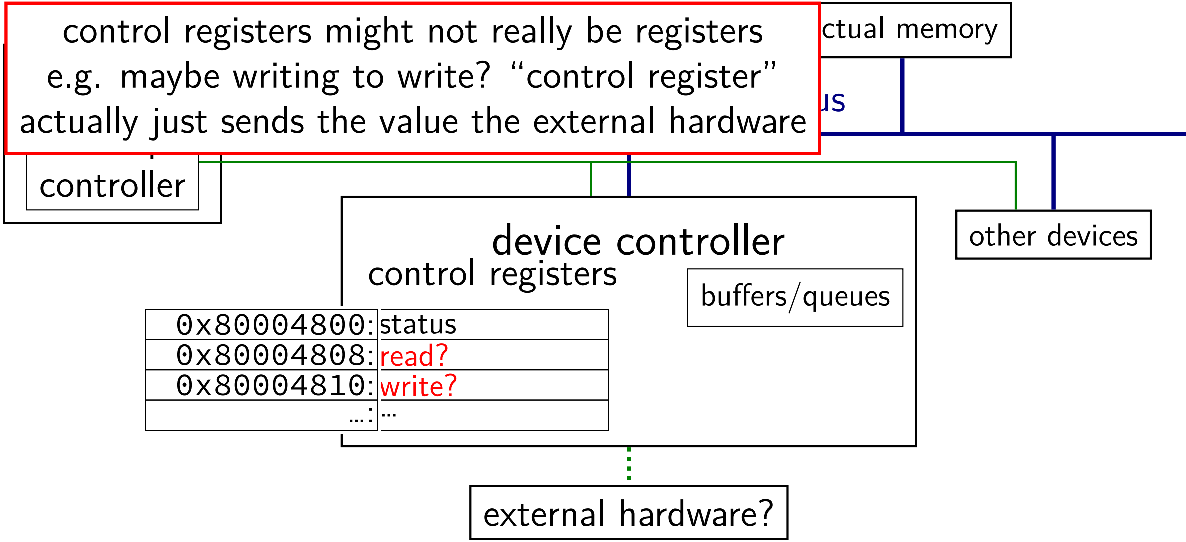


connecting devices

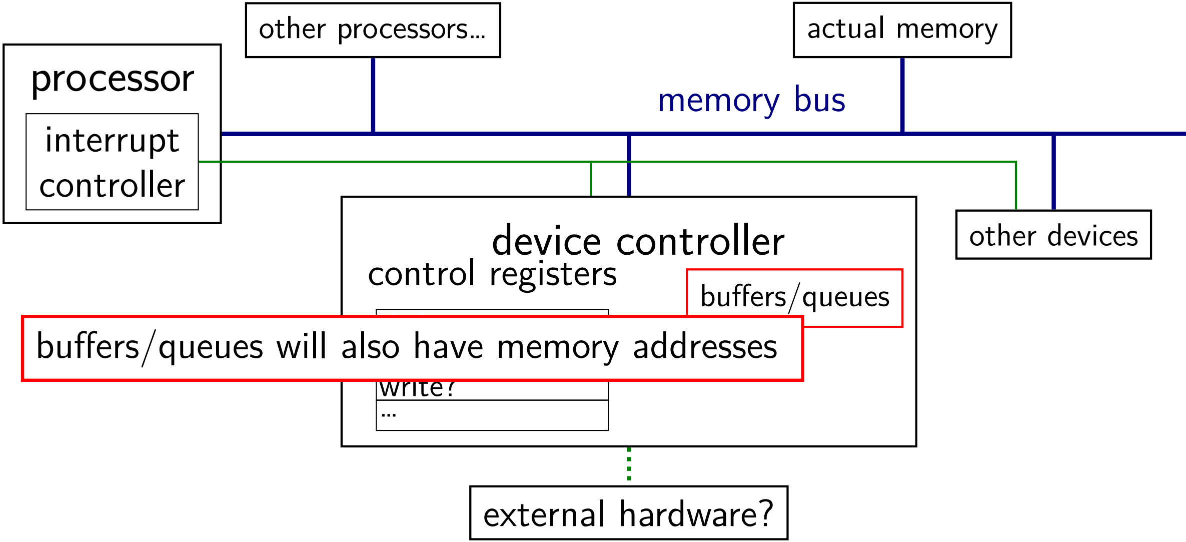


connecting devices

control registers might not really be registers
e.g. maybe writing to write? “control register” actually just sends the value the external hardware

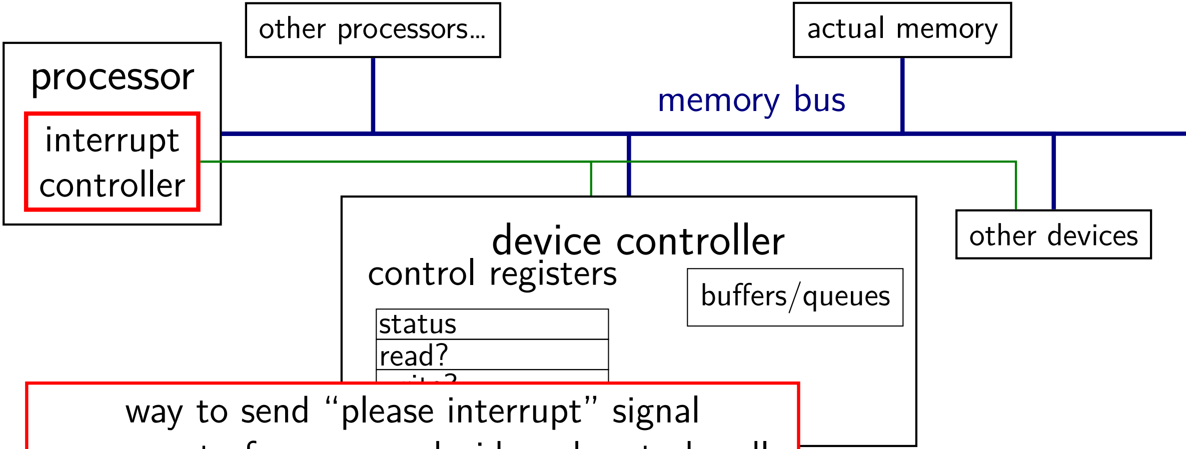


connecting devices



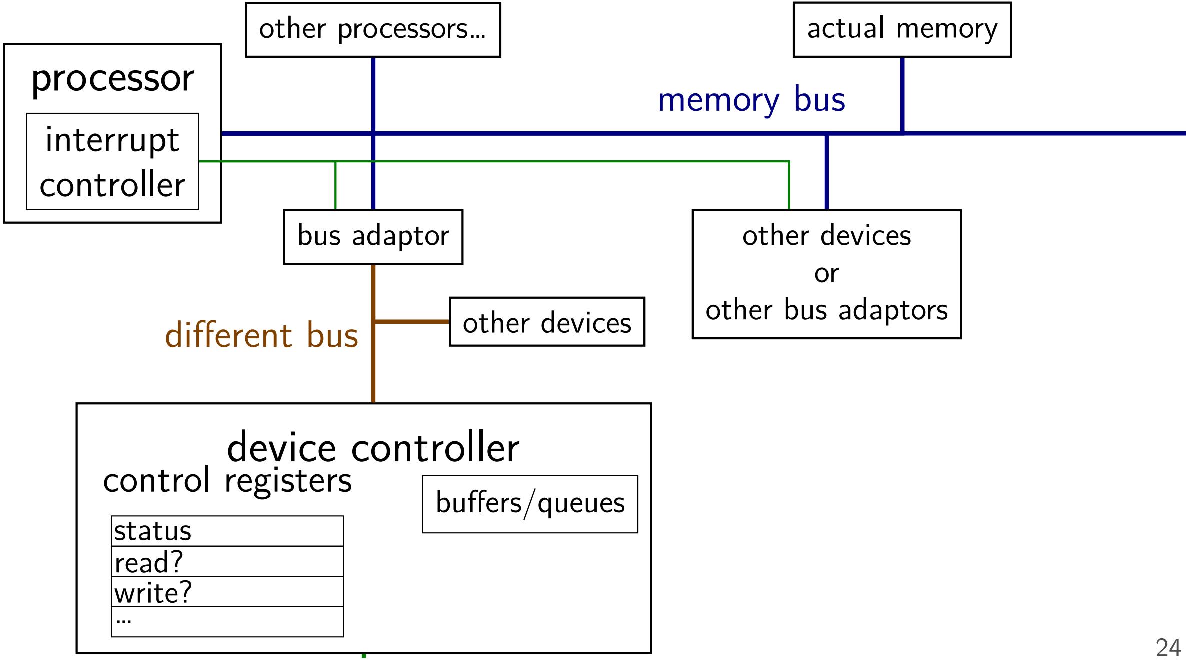
buffers/queues will also have memory addresses

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`)

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

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

programmed I/O

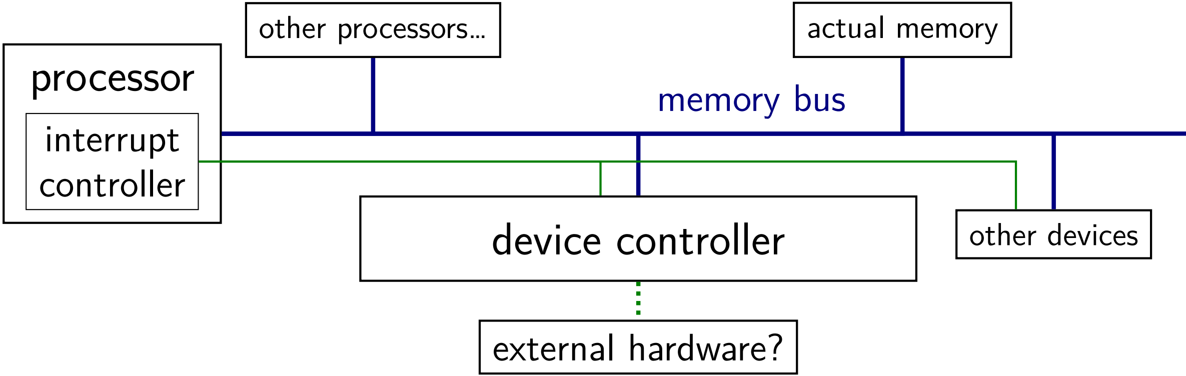
"programmed I/O": write to or read from device controller buffers directly

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?

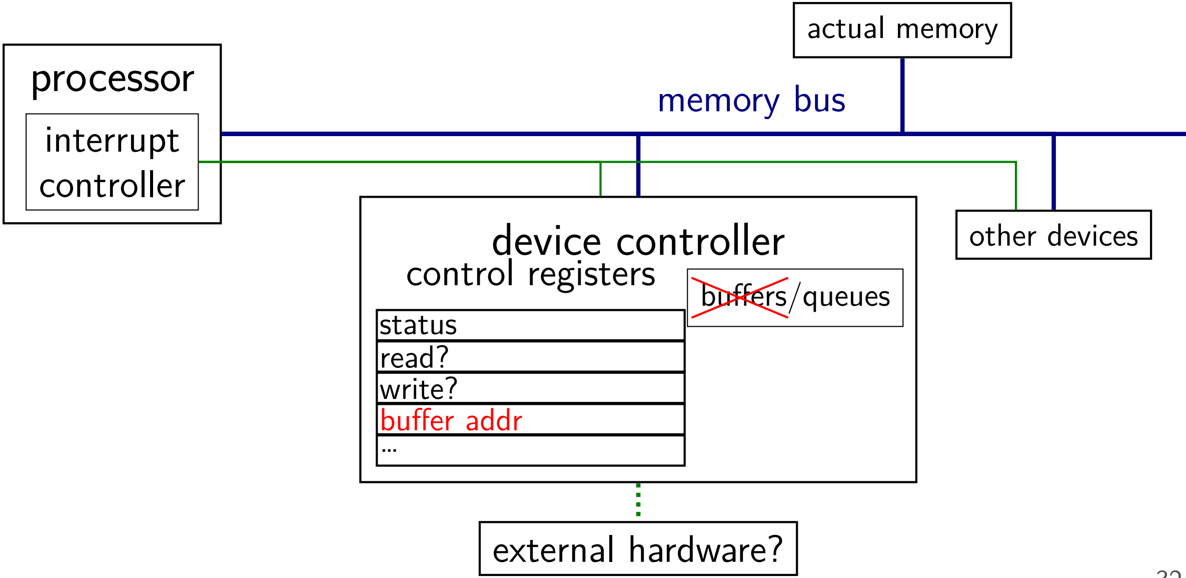
direct memory access (DMA)



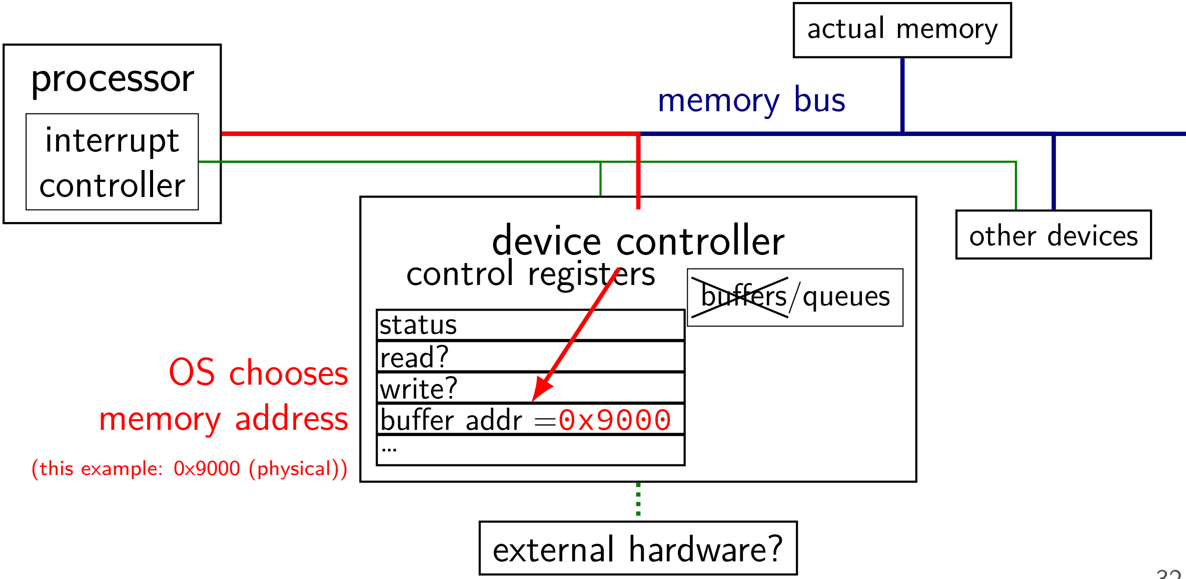
observation: devices can read/write memory

can have device copy data to/from memory

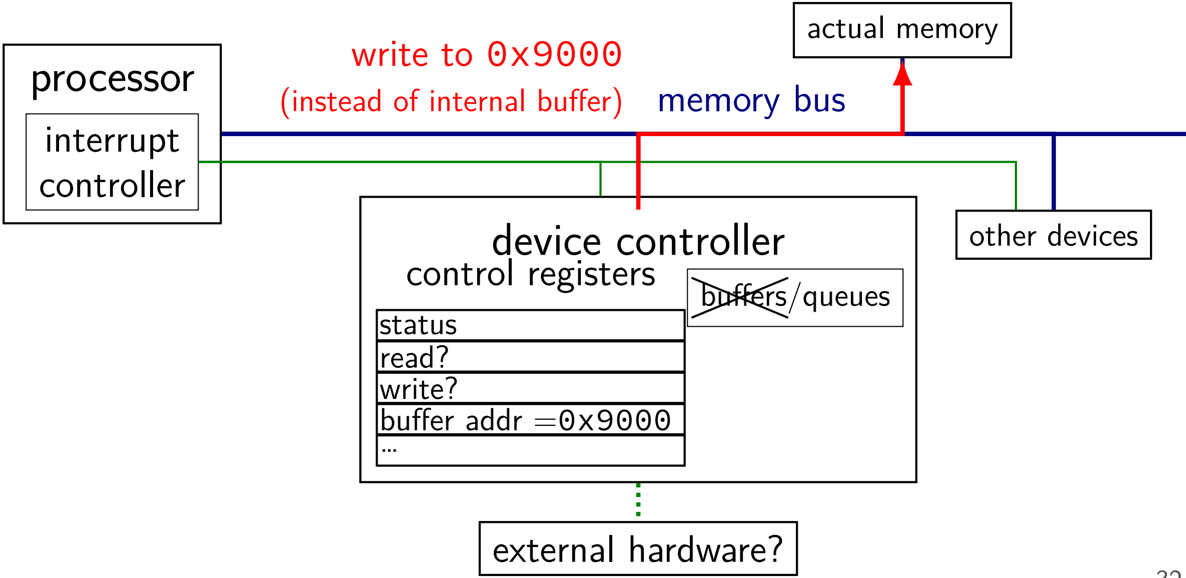
direct memory access (DMA)



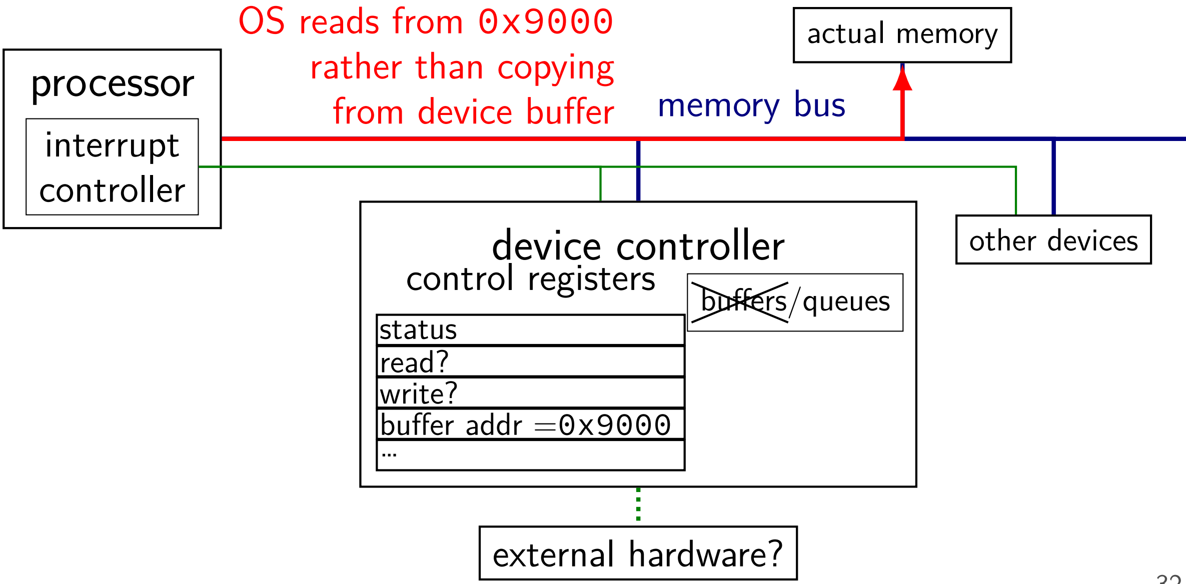
direct memory access (DMA)



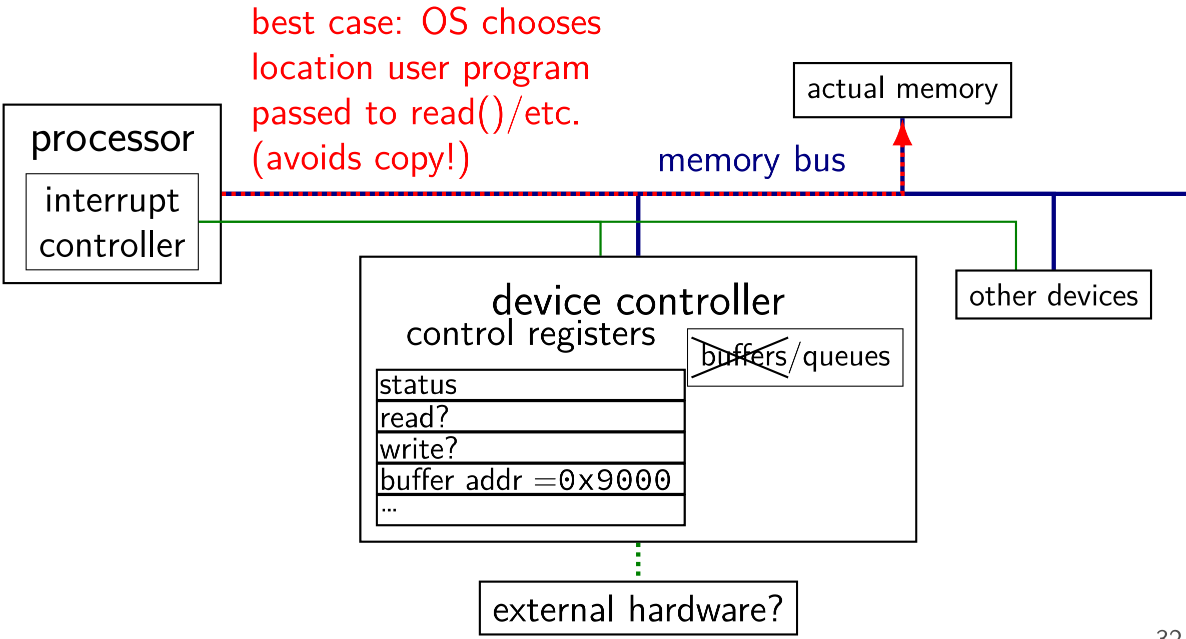
direct memory access (DMA)



direct memory access (DMA)



direct memory access (DMA)



direct memory access (DMA)

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

avoids having processor run a loop to copy data
- OS can run normal program during data transfer
- interrupt tells OS when copy finished

device uses memory as very large buffer space

device puts data where OS wants it directly (maybe)
- OS specifies physical address to use…
- instead of reading from device controller

direct memory access (DMA)

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

avoids having processor run a loop to copy data
- OS can run normal program during data transfer
- interrupt tells OS when copy finished

device uses memory as very large buffer space

device puts data where OS wants it directly (maybe)
- OS specifies physical address to use...
- instead of reading from device controller

OS puts data where it wants

so far: where it wants is the device driver's buffer

OS puts data where it wants

so far: where it wants is the device driver's buffer

seems like OS could also put it directly where application wants it?

- i.e. pointer passed to read() system call
- called "zero-copy I/O"

OS puts data where it wants

so far: where it wants is the device driver's buffer

seems like OS could also put it directly where application wants it?

- i.e. pointer passed to read() system call
- called "zero-copy I/O"

should be faster, but, in practice, very rarely done:

- if part of regular file, can't easily share with page cache
- device might expect contiguous physical addresses
- device might expect physical address is at start of physical page
- device might write data in differnt format than application expects
- device might read too much data
- need to deal with application exiting/being killed before device finishes
- …

exercise

system is running two applications

- A: reading from network
- B: doing tons of computation

timeline:

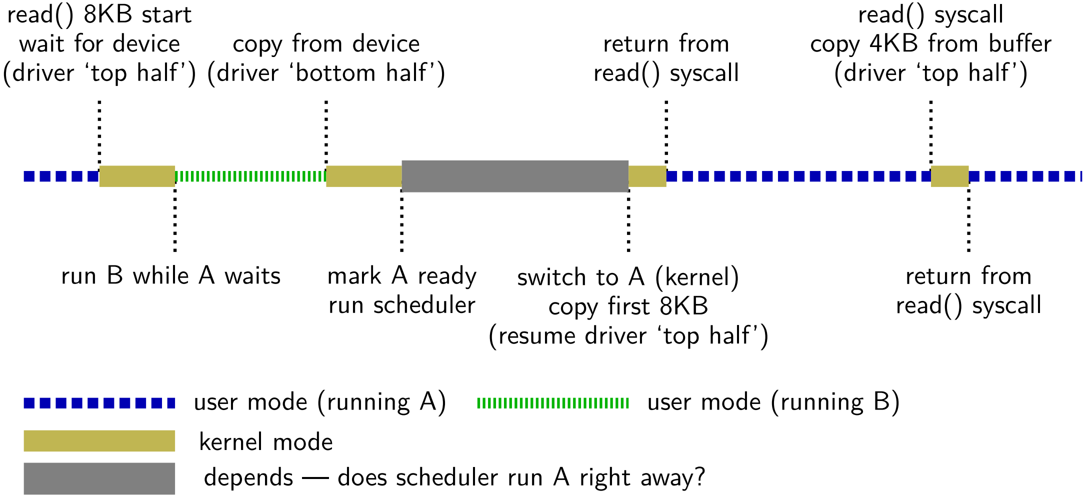
- A calls read() to 8KB of data from network
- 16KB of data comes in 10ms later
- A calls read() again to get remaining 4KB

exercise 1: how many kernel/user mode switches?

exercise 2: how many context switches?

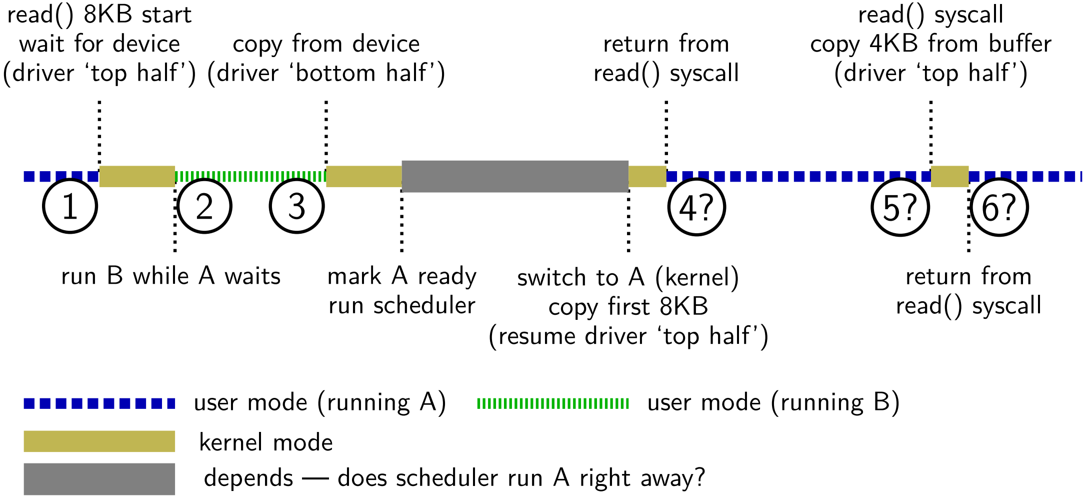
how many mode switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB



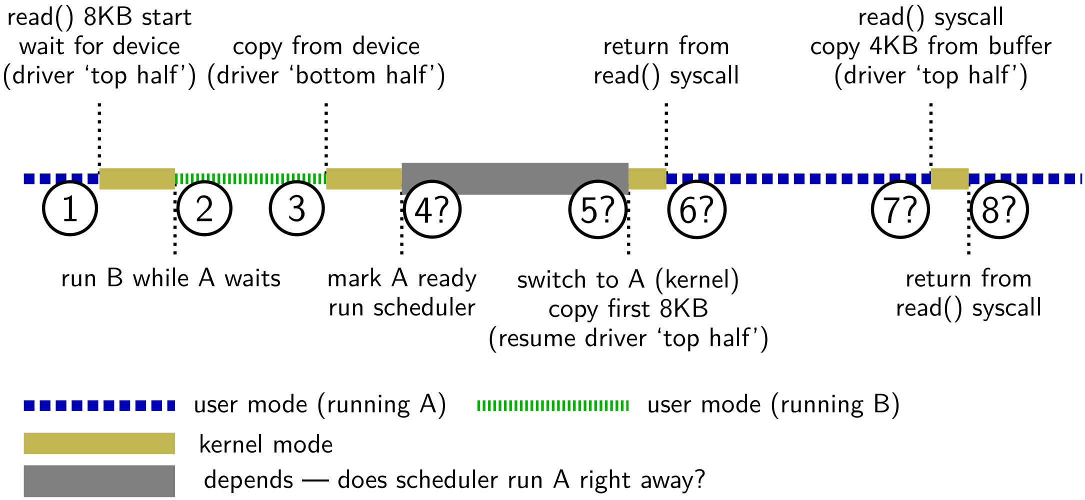
how many mode switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB



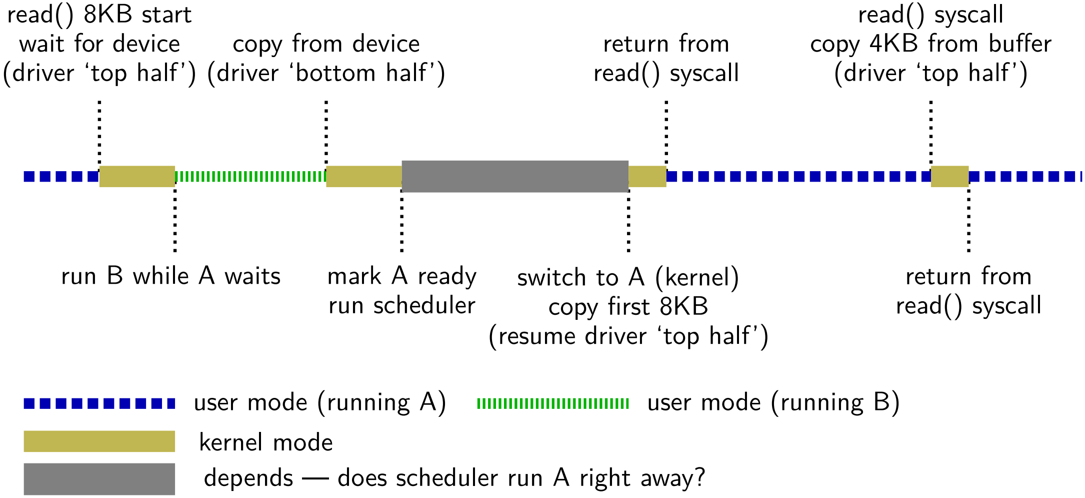
how many mode switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB



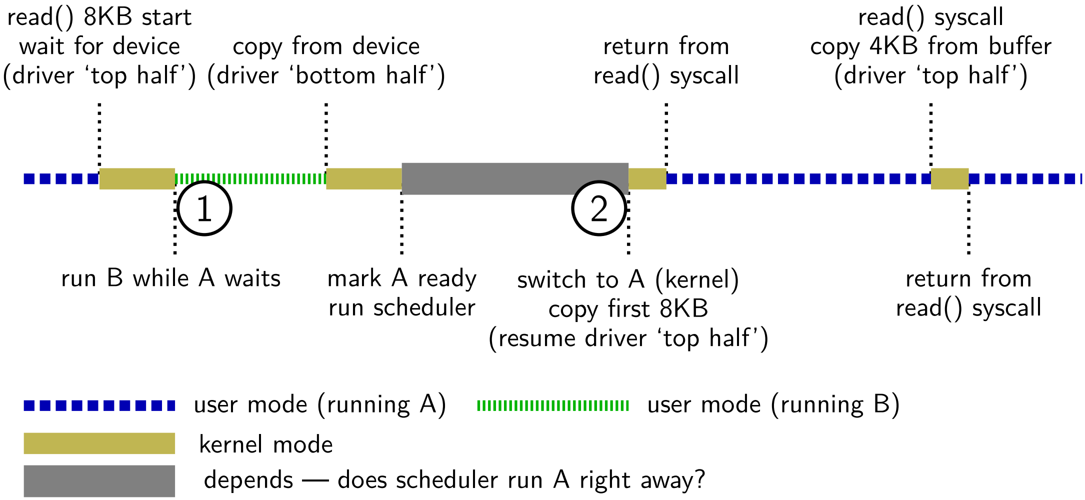
how many context switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB



how many context switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB



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

devices summary

device *controllers* connected via memory bus

- usually assigned physical memory addresses
- sometimes separate “I/O addresses” (special load/store instructions)

controller looks like “magic memory” to OS

- load/store from device controller registers like memory
- setting/reading control registers can trigger device operations

two options for data transfer

- programmed I/O: OS reads from/writes to buffer within device controller
- direct memory access (DMA): device controller reads/writes normal memory

filesystems

hard drive interfaces

hard drives and solid state disks are divided into sectors

historically 512 bytes (larger on recent disks)

disk commands:

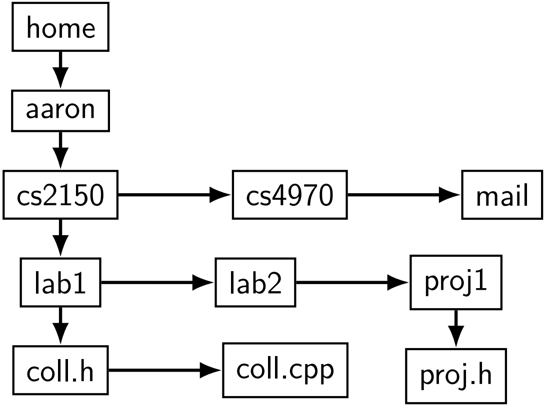
- read from sector i to sector j
- write from sector i to sector j this data

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 sectors of data



(figure adapted from Bloomfield's CS 2150 slides)

filesystem problems

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

which sectors? parts of sectors?

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

given a file/directory, where is its metadata?

owner, modification date, permissions, size, ...

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

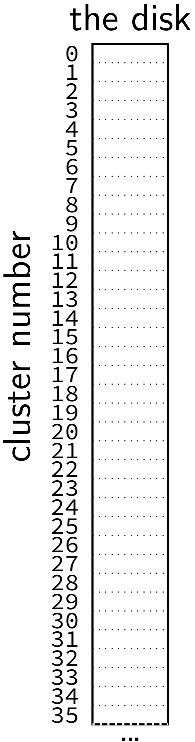
FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read

- determined by disk hardware
- historically 512 bytes, but often bigger now

cluster: typically 512 to 4096 bytes



FAT and sectors

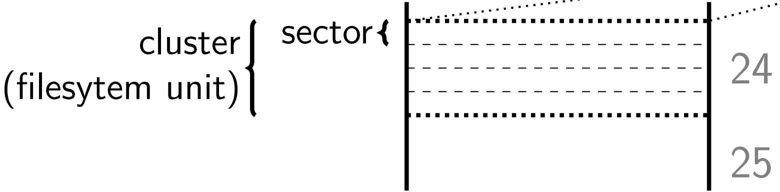
FAT divides disk into *clusters*

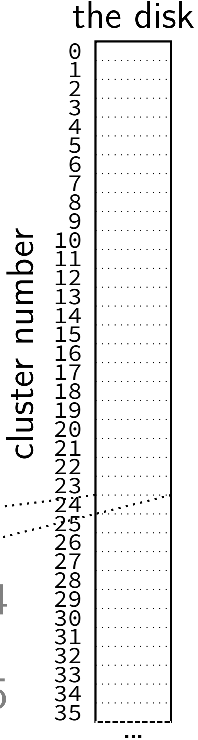
composed of one or more sectors

sector = minimum amount hardware can read

- determined by disk hardware
- historically 512 bytes, but often bigger now

cluster: typically 512 to 4096 bytes



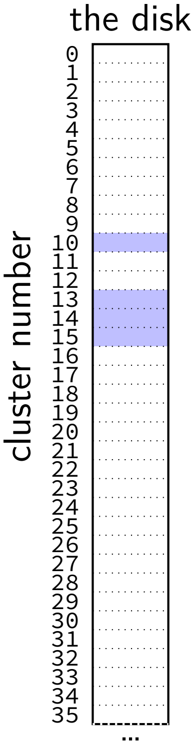


FAT: clusters and files

a file's data stored in a list of clusters

file size isn't multiple of cluster size? waste space

reading a file? need to find the list of clusters

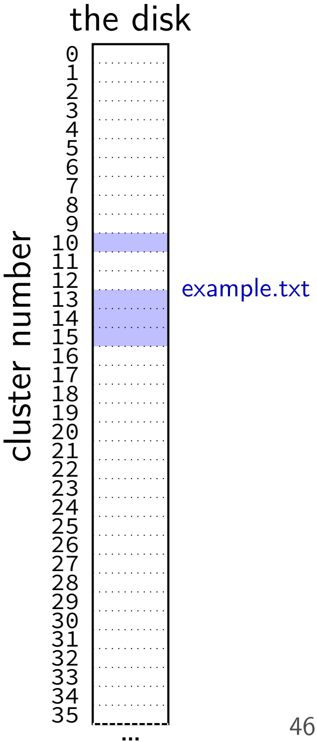


FAT: clusters and files

a file's data stored in a list of clusters

file size isn't multiple of cluster size? waste space

reading a file? need to find 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

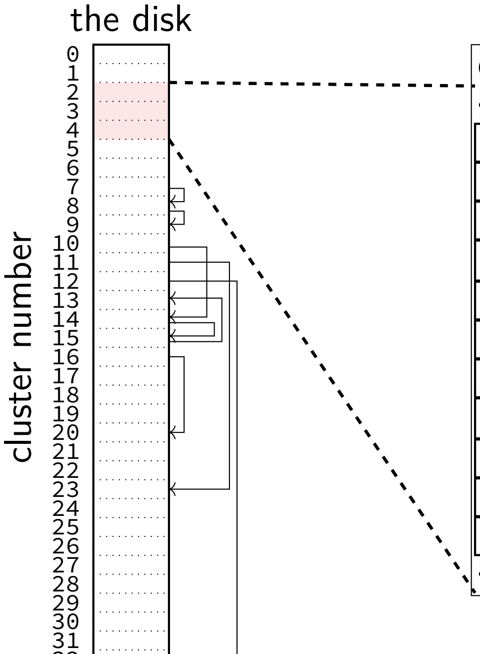
next pointers? file allocation table entry for cluster

special value for NULL (-1 in this example; maybe different in real FAT)

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 in:
cluster 10, then 14, then 15, then 13

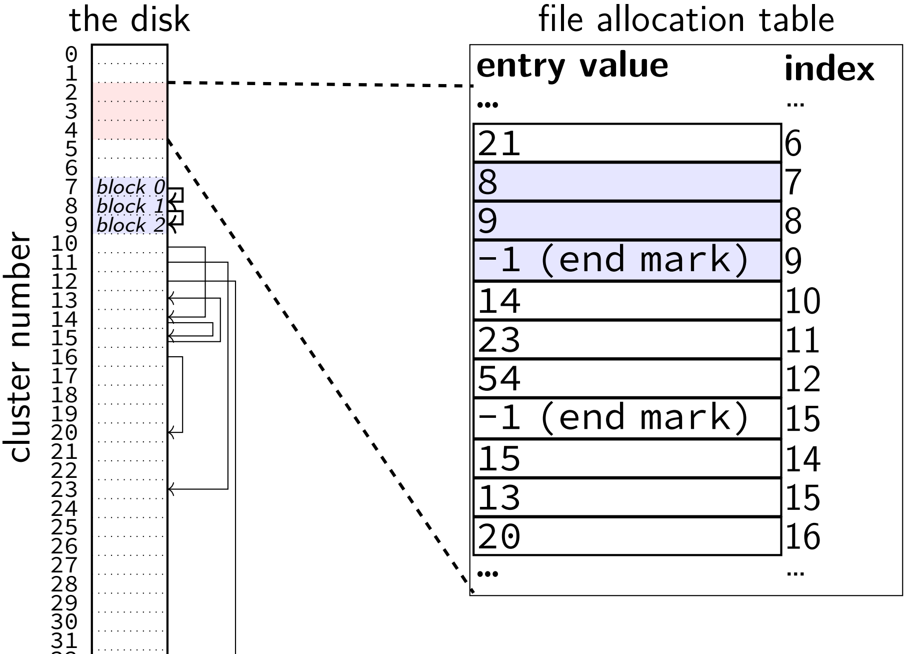
FAT: reading a file (2)



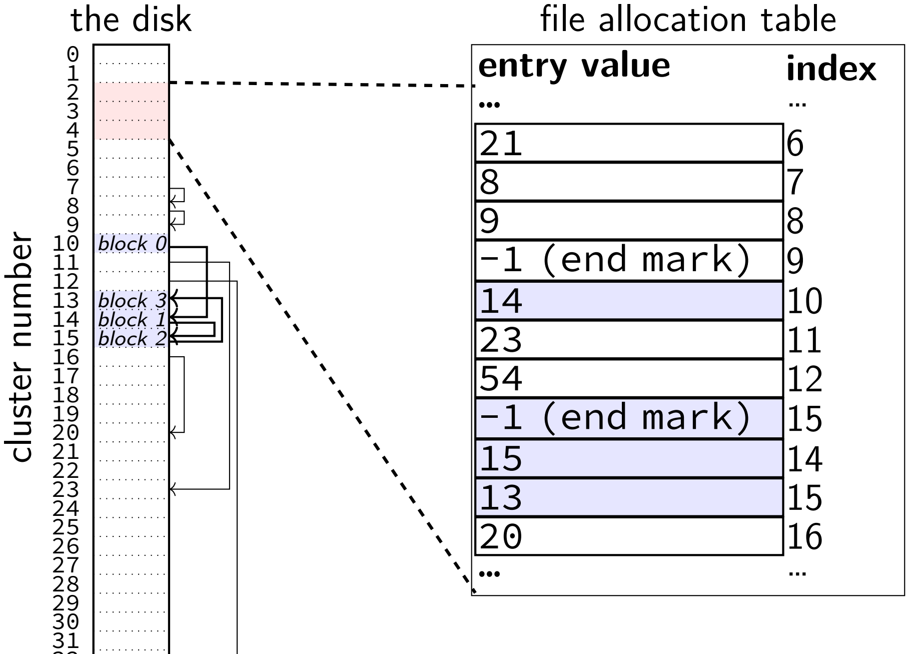
file allocation table

entry value	index
…	…
21	6
8	7
9	8
-1 (end mark)	9
14	10
23	11
54	12
-1 (end mark)	15
15	14
13	15
20	16
…	…

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?

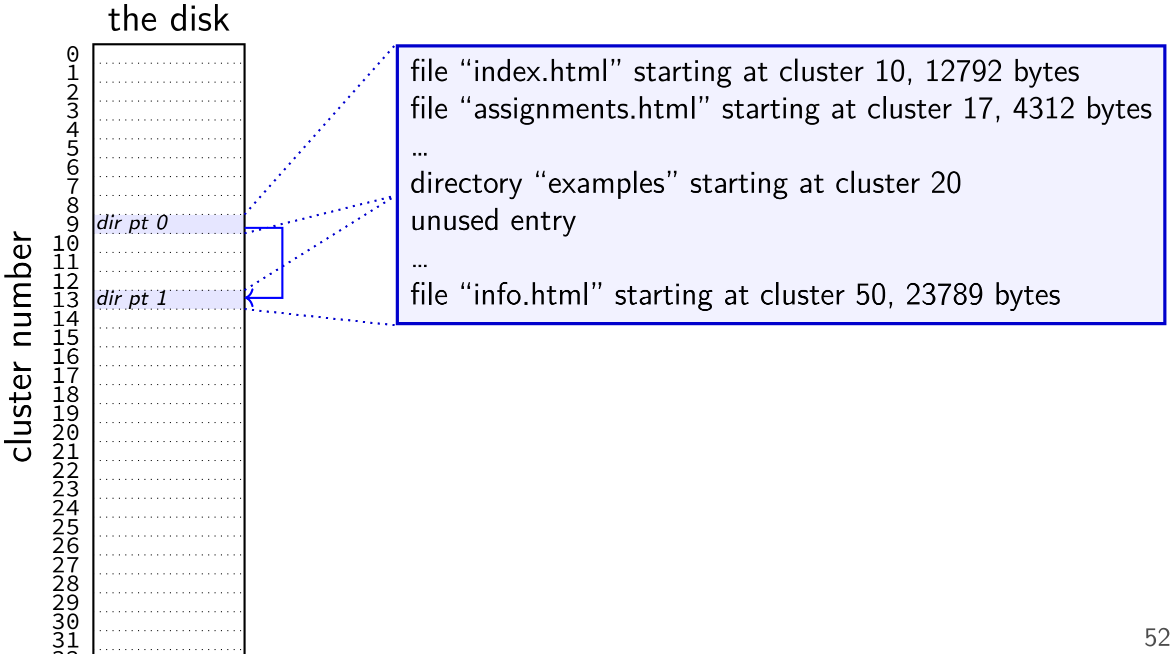
really want filenames

stored in directories!

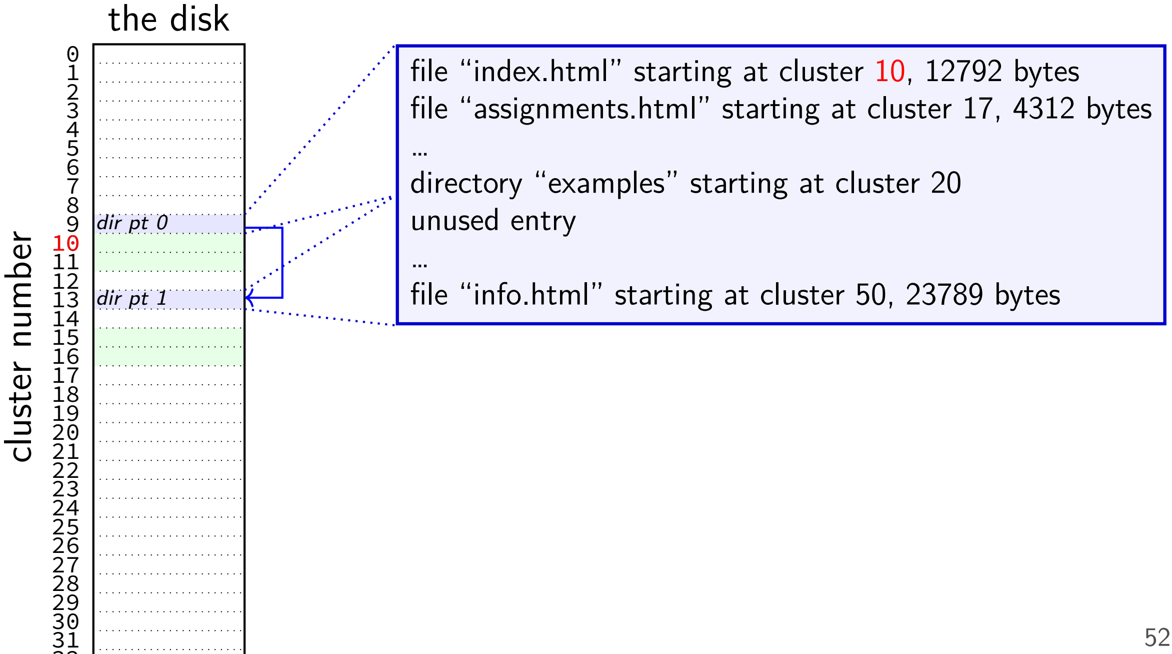
in FAT: directory is a file, but its data is list of:

(name, starting location, other data about file)

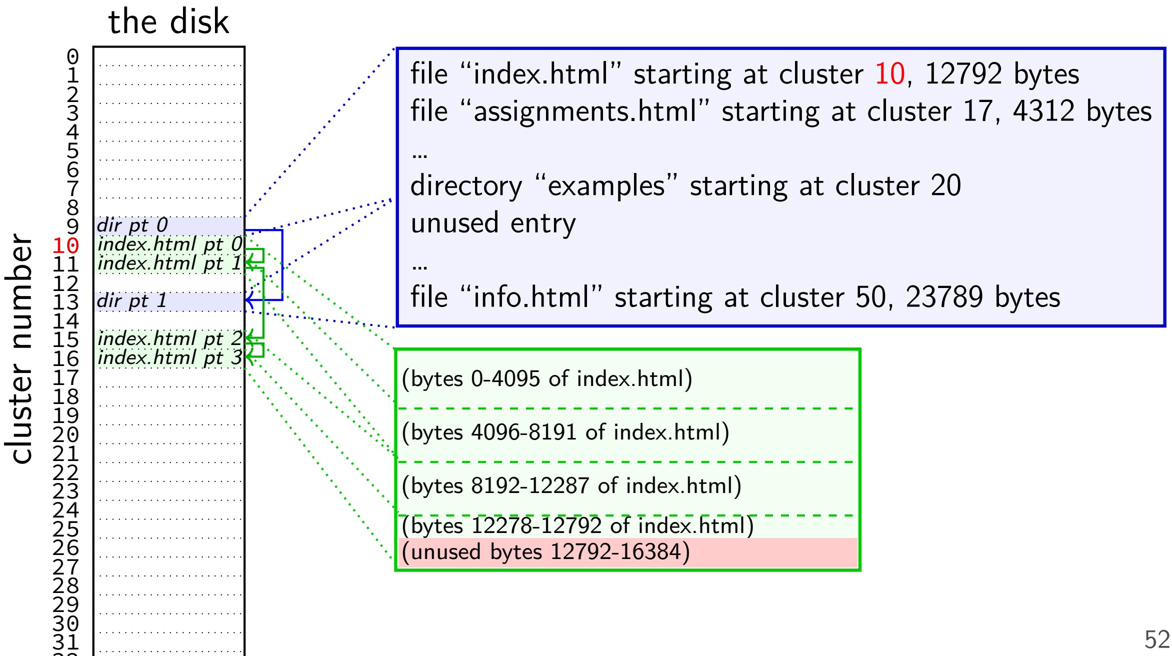
finding files with directory



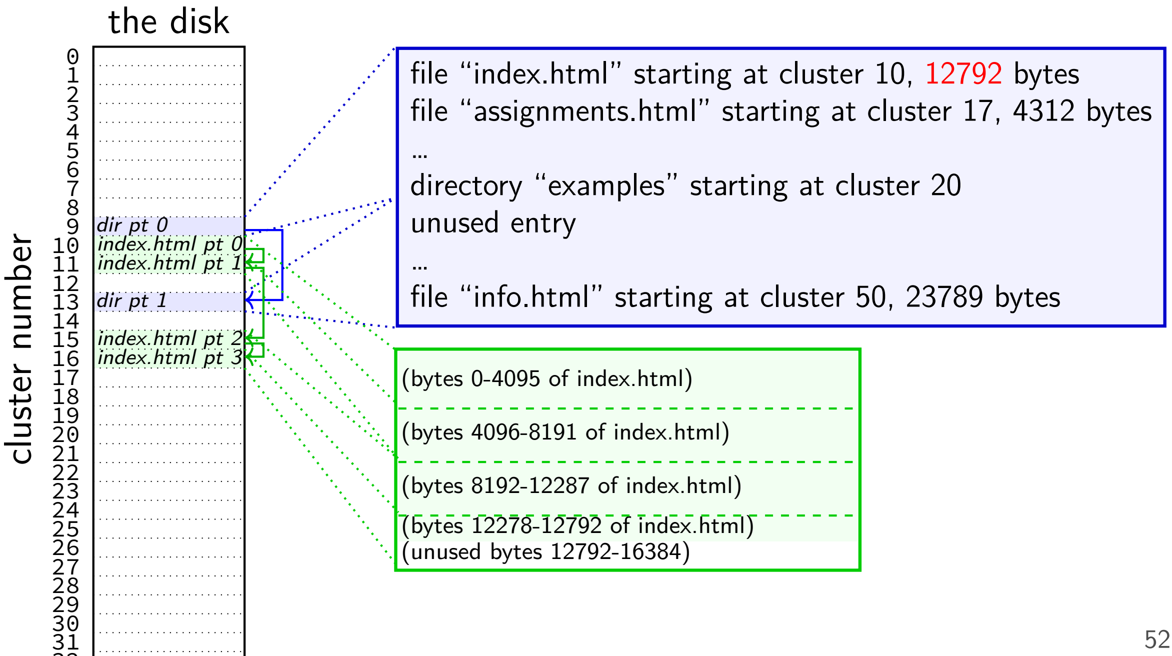
finding files with directory



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

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
…

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	…
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry…			

FAT directory entry

box = 1 byte

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

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

32-bit first cluster number split into two parts
(history: used to only be 16-bits)

FAT directory entry

box = 1 byte

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

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (`README.TXT`)											attrs

directory?
read-only?
hidden?
…

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	…	
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry…				

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

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
…

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	…
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry…			

8 character filename + 3 character extension
history: used to be all that was supported

FAT directory entry

box = 1 byte

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

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
...

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	...
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry...			

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

'R'	'E'	'A'	'D'	'M'	'E'	'␣'	'␣'	'T'	'X'	'T'	0x00
filename + extension (README.TXT)											attrs

directory?
read-only?
hidden?
…

0x9C	0xA1	0x20	0x7D	0x3C	0x7D	0x3C	0x01	0x00	0xEC	0x62	0x76
creation date + time (2010-03-29 04:05:03.56)					last access (2010-03-29)		cluster # (high bits)		last write (2010-03-22 12:23:12)		

0x3C	0xF4	0x04	0x56	0x01	0x00	0x00	'F'	'O'	'O'	…
last write con't	cluster # (low bits)		file size (0x156 bytes)				next directory entry…			

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 attribute
  uint8_t DIR_NTRes;           // set value to 0, never change t
  uint8_t DIR_CrtTimeTenth;    // millisecond timestamp for file
  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 of this entry's firs
  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's first
  uint32_t DIR_FileSize;       // file size in bytes
};
```

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Name[11];           // short name
  uint8_t DIR_Attr;               // File attribute
  uint8_t DI                                                   ge t
  uint8_t DI                                                   file
  uint16_t D
  uint16_t D
  uint16_t DIR_LstAccDate;        // last access date
  uint16_t DIR_FstClusHI;         // high word of this entry's firs
  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's first
  uint32_t DIR_FileSize;          // file size in bytes
};
```

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__((packed)) DirEntry {
  uint8_t DIR_Name[11
  uint8_t DIR_Attr;
  uint8_t DIR_NTRes;                                          ge t
  uint8_t DIR_CrtTime                                         file
  uint16_t DIR_CrtTi
  uint16_t DIR_CrtDate;        // date file was created
  uint16_t DIR_LstAccDate;     // last access date
  uint16_t DIR_FstClusHI;      // high word of this entry's firs
  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's first
  uint32_t DIR_FileSize;       // file size in bytes
};
```

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 __attribute__((packed)) DirEntry {
  uint8_t DIR_Nam
  uint8_t DIR_Att
  uint8_t DIR_NTR                                              ge t
  uint8_t DIR_CrtTimeTenth;  // millisecond timestamp for file
  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 of this entry's firs
  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's first
  uint32_t DIR_FileSize;     // file size in bytes
};
```

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

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

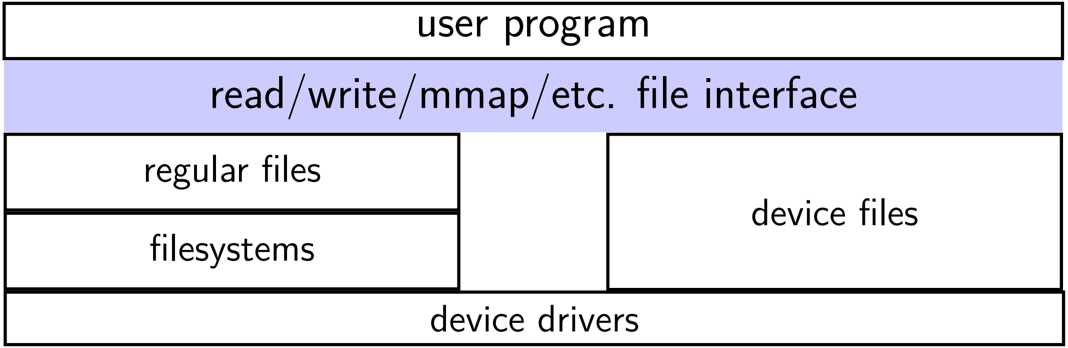
read baz's directory entries to find file.txt

the root directory?

but where is the first directory?

backup slides

ways to talk to I/O devices



devices as files

talking to device? open/read/write/close

typically similar interface within the kernel

device driver implements the file interface

example device files from a Linux desktop

`/dev/snd/pcmC0D0p` — audio playback

configure, then write audio data

`/dev/sda`, `/dev/sdb` — SATA-based SSD and hard drive

usually access via filesystem, but can mmap/read/write directly

`/dev/input/event3`, `/dev/input/event10` — mouse and keyboard

can read list of keypress/mouse movement/etc. events

`/dev/dri/renderD128` — builtin graphics

DRI = direct rendering infrastructure

devices: extra operations?

read/write/mmap not enough?

- audio output device — set format of audio? headphones plugged in?
- terminal — whether to echo back what user types?
- CD/DVD — open the disk tray? is a disk present?
- ...

extra POSIX file descriptor operations:

- ioctl (general I/O control) — device driver-specific interface
- tcsetattr (for terminal settings)
- fcntl
- ...

also possibly extra device files for same device:

- `/dev/snd/controlC0` to configure audio settings for `/dev/snd/pcmC0D0p`, `/dev/snd/pcmC0D10p`, ...