last time

page replacement (= page allocation)

maximizing hit rate: Belady's MIN
- assuming entirely reactive

referenced (or accessed) bits

dirty bits

locality assumption: least recently used

approximating least recently used
- second chance
- Linux's SEQ policy — inactive list (scan for access), active list (don't)

when LRU fails

being proactive

previous assumption: load on demand

why is something loaded?

- page fault
- maybe because application starts

can we do better?

readahead

program accesses page 4 of a file, page 5, page 6. What's next?

readahead

program accesses page 4 of a file, page 5, page 6. What's next?

page 7 — idea: guess this

  on page fault, does it look like contiguous accesses?

called readahead

being less lazy elsewhere

showed OS: proactively reading in pages

can also proactively free pages (faster replacement)

and proactively write out pages ‘dirty’ pages

- save time writing later
- avoid data loss on power failure

page cache/replacement summary

program memory + files — swapped to disk, cached in memory

mostly, assume temporal locality

- least recently used variants

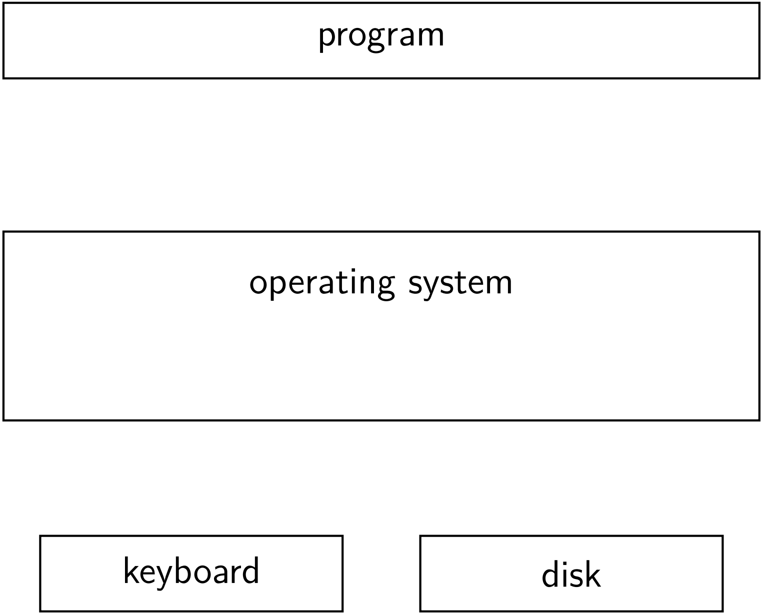
special cases for non-LRU-friendly patterns (e.g. scans)

- maybe more we haven't discussed?

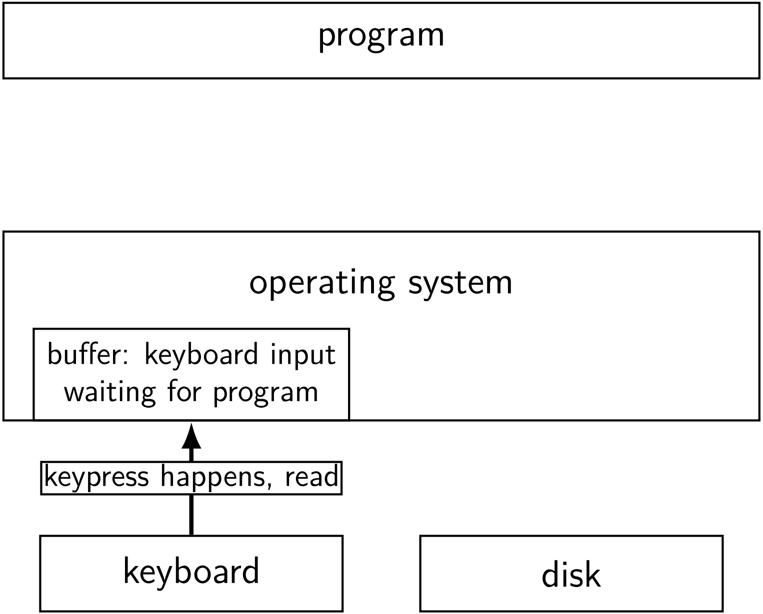
being proactive (writeback early, readahead, pre-evicted pages)

missing: handling non-miss-rate goals?

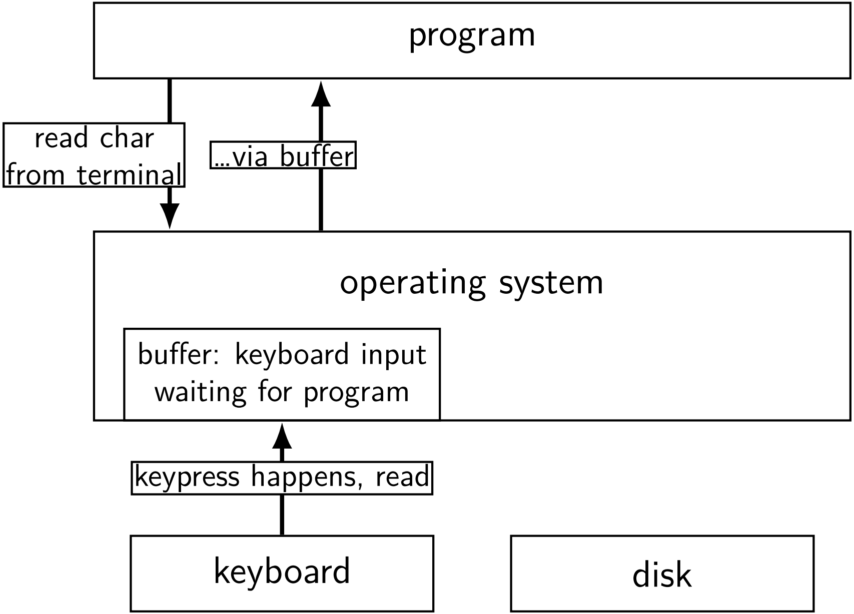
kernel buffering (reads)



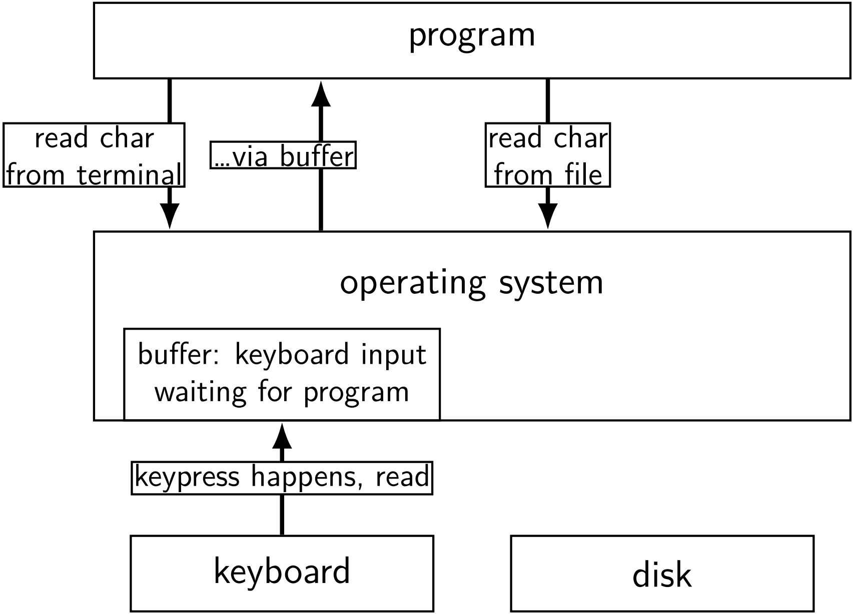
kernel buffering (reads)



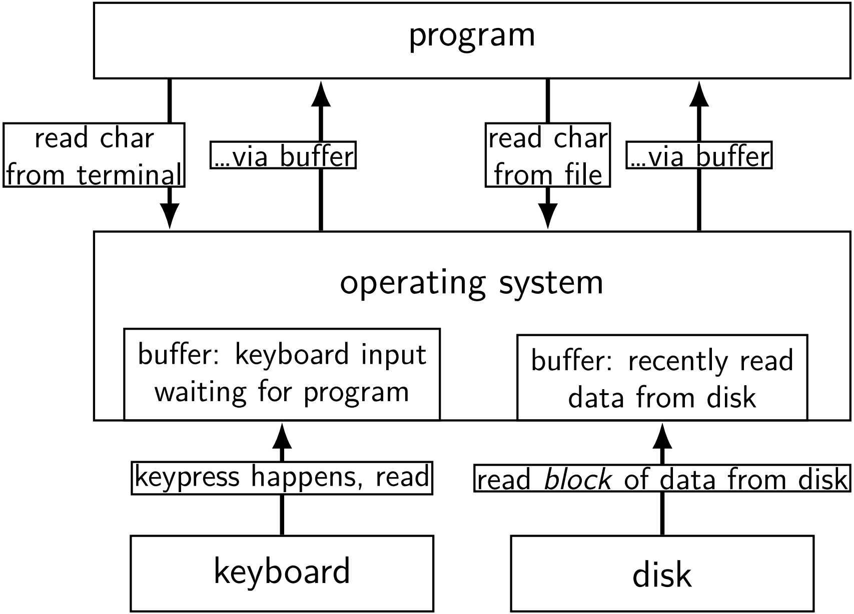
kernel buffering (reads)



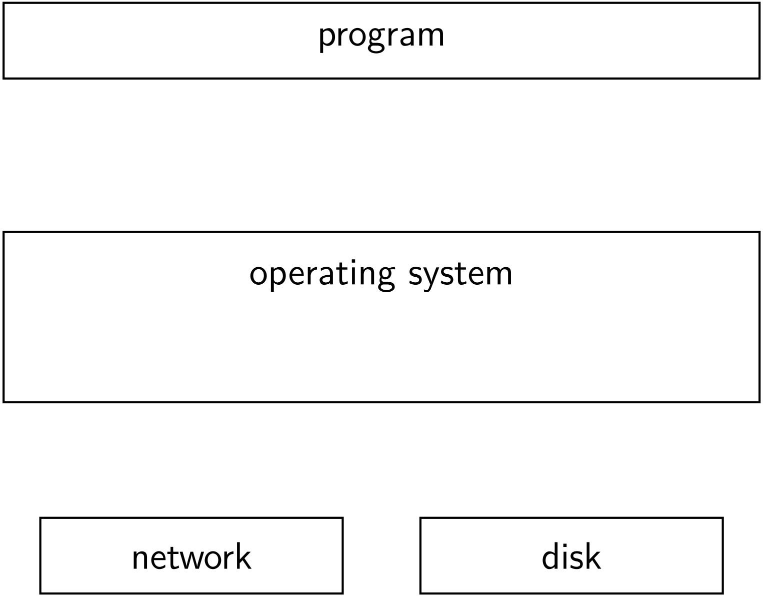
kernel buffering (reads)



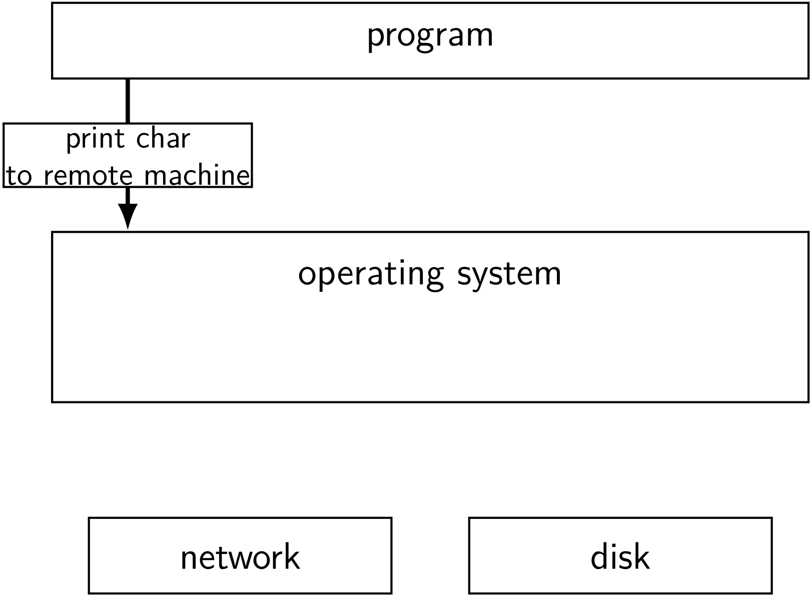
kernel buffering (reads)



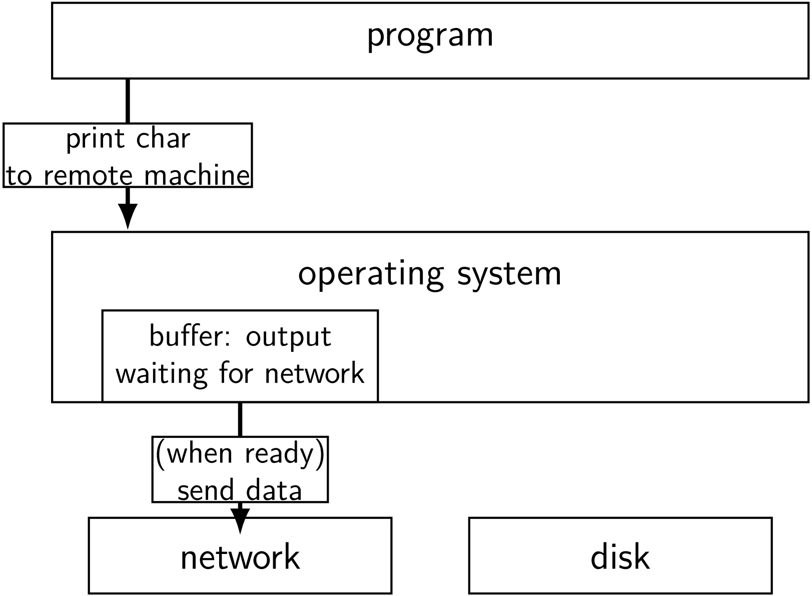
kernel buffering (writes)



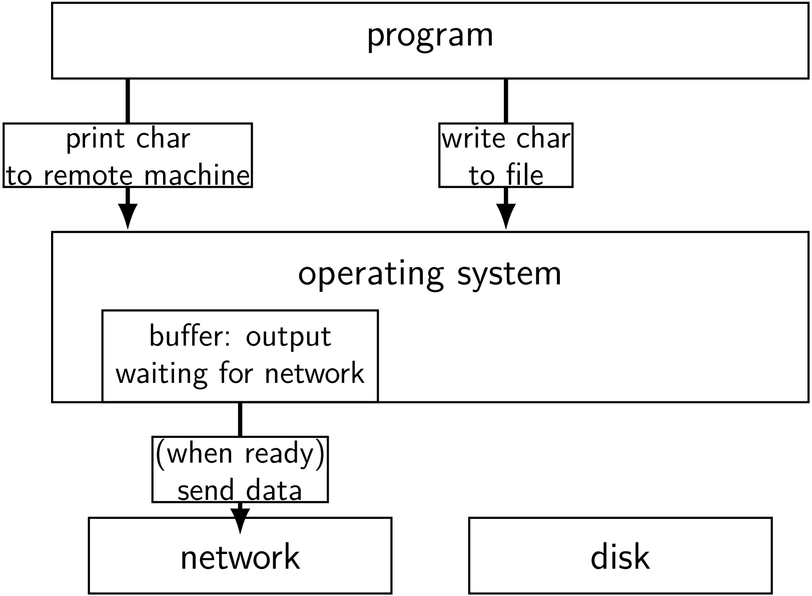
kernel buffering (writes)



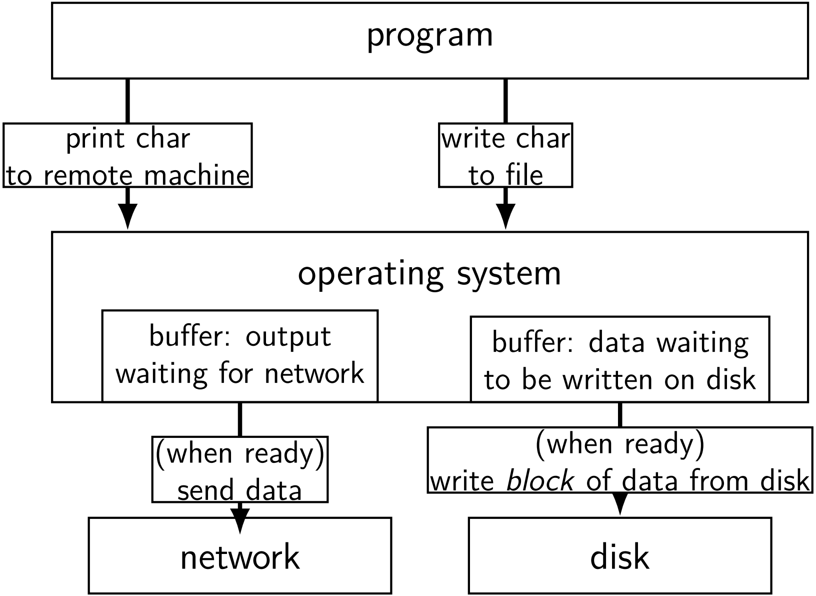
kernel buffering (writes)



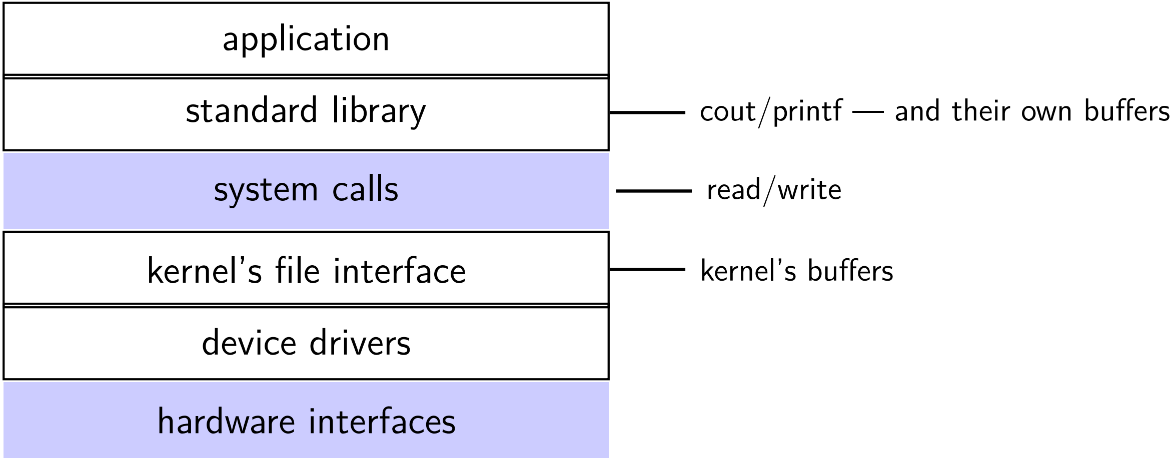
kernel buffering (writes)



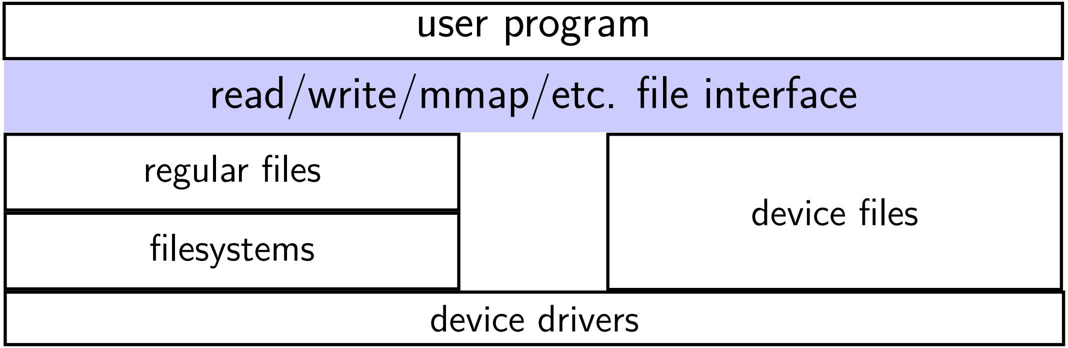
kernel buffering (writes)



layering


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

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

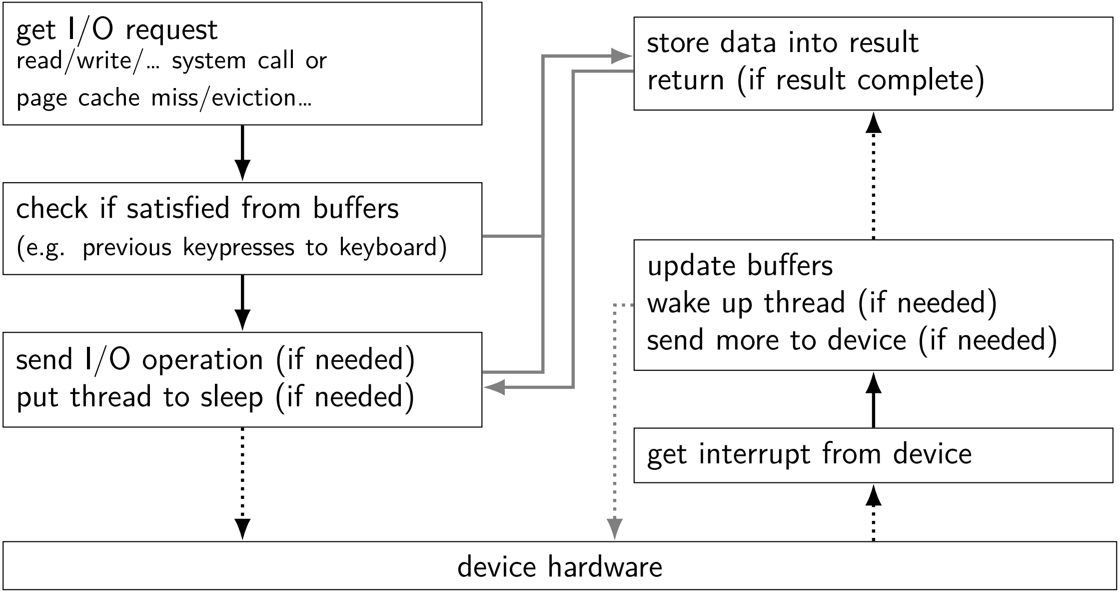
read/write page at a time

Linux example: block device operations

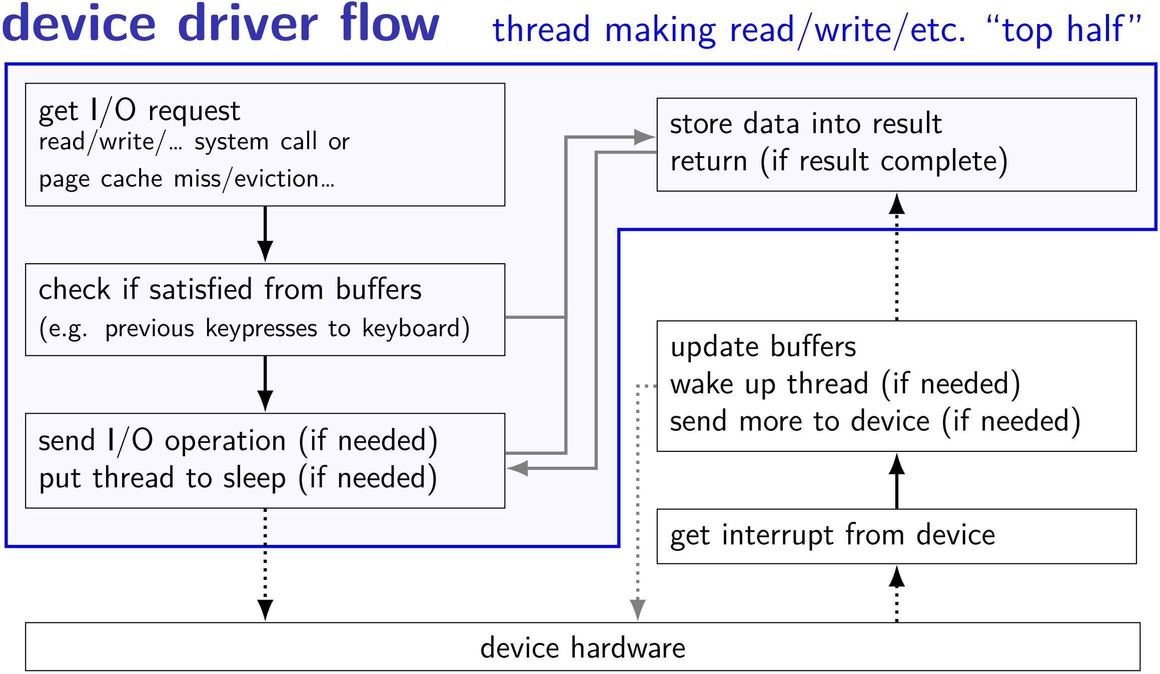
```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *,
            sector_t, struct page *, bool);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, un
    ...
};
```

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

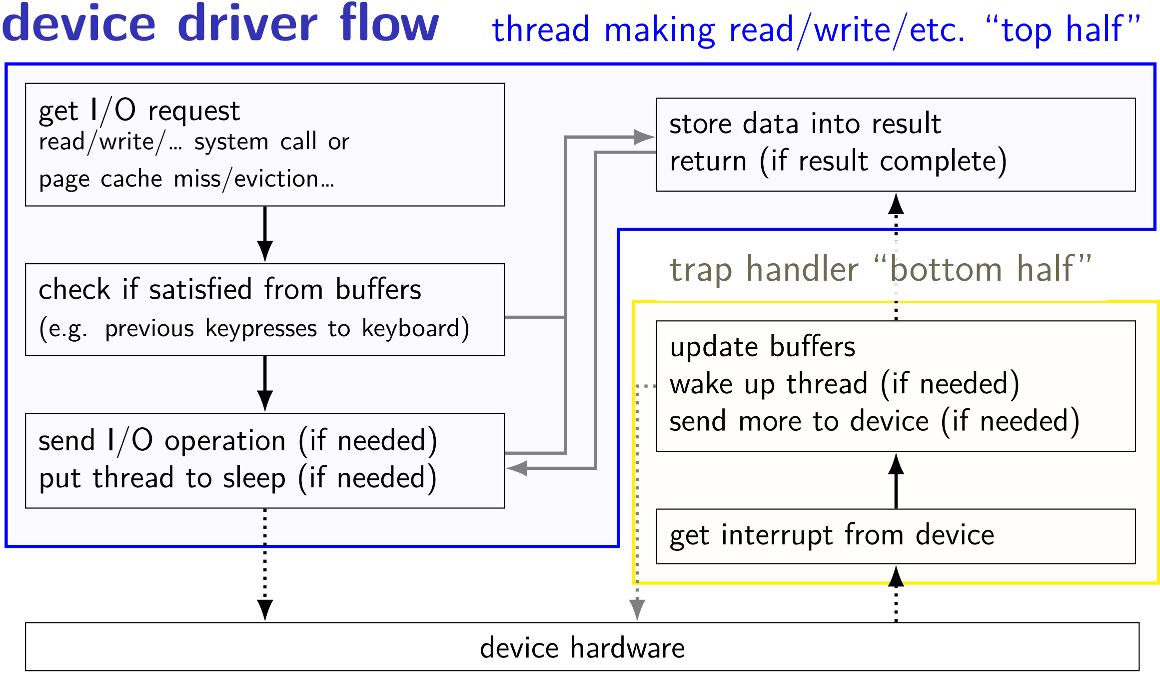
device driver flow



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

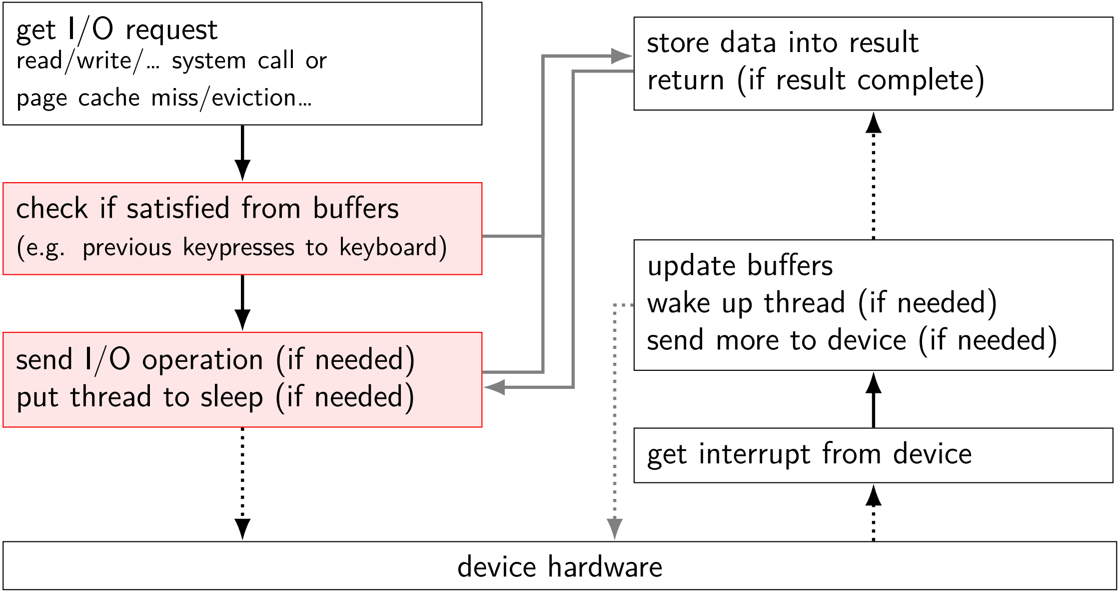
code run at boot:

```
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)

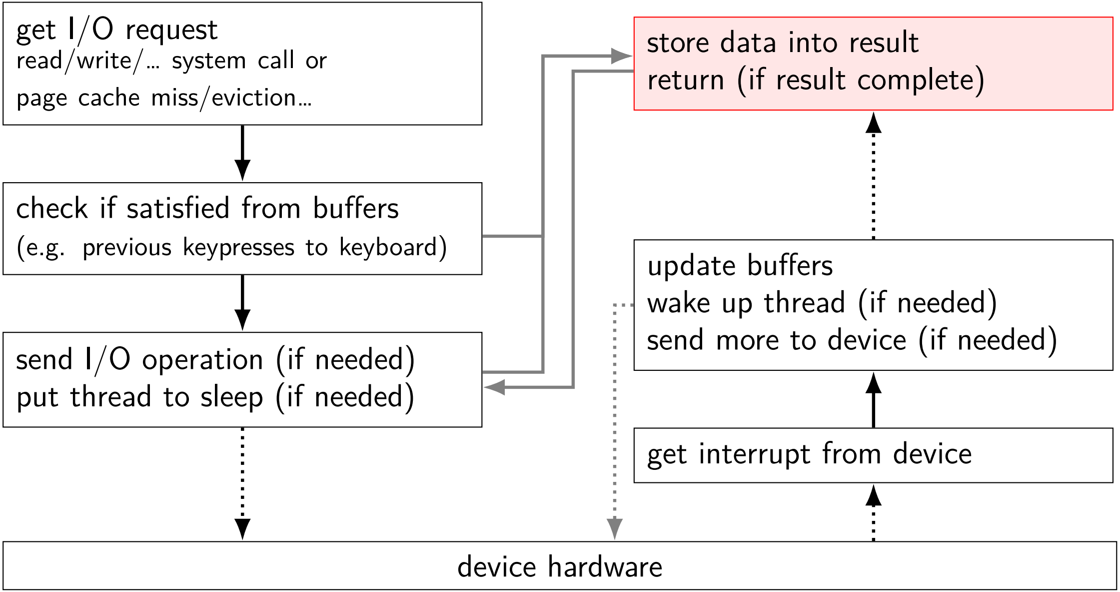
```
int
consoleread(struct inode *ip, char *dst, int n)
{
   ...
   target = n;
   acquire(&cons.lock);
   while(n > 0){
      while(input.r == input.w){
         if(myproc()->killed){
            ...
            return -1;
         }
         sleep(&input.r, &cons.lock);
      }
      ...
   }
   release(&cons.lock)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

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

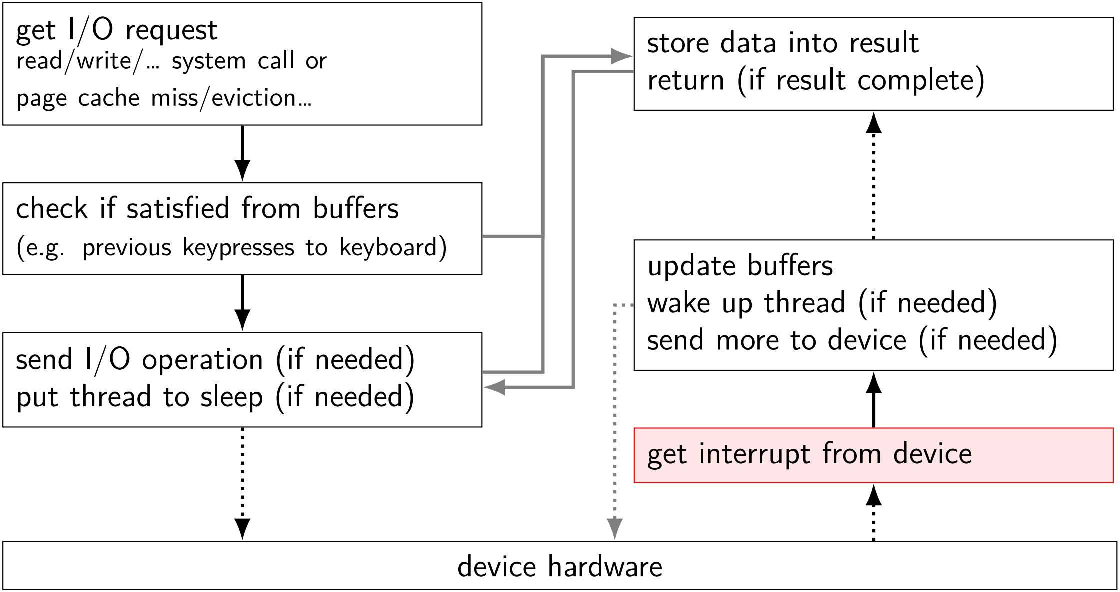
wait for buffer to fill

- 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”

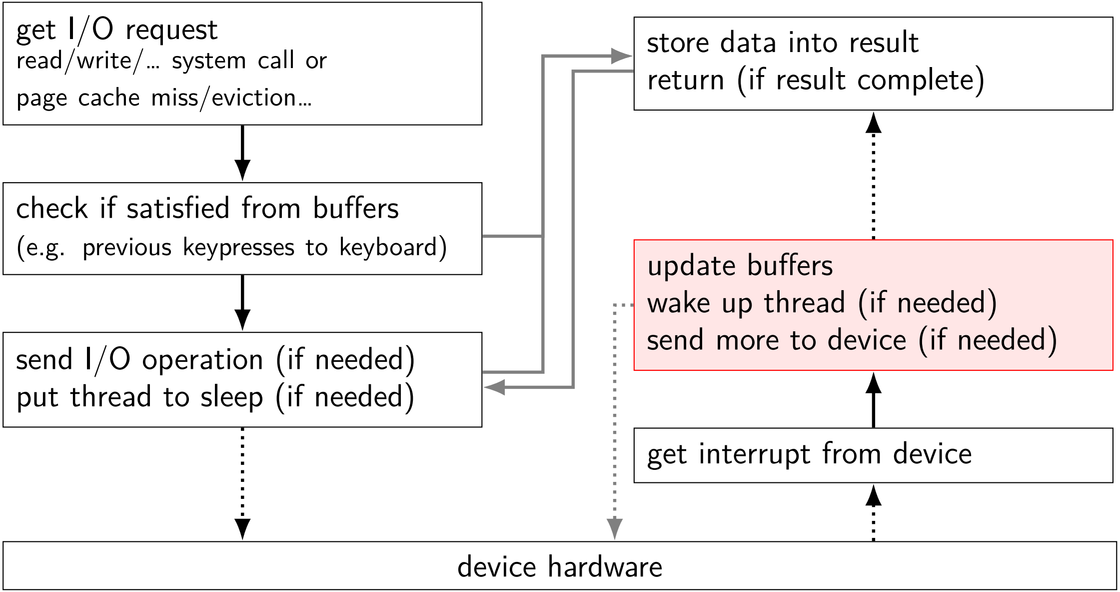
xv6: console interrupt (one case)

```
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



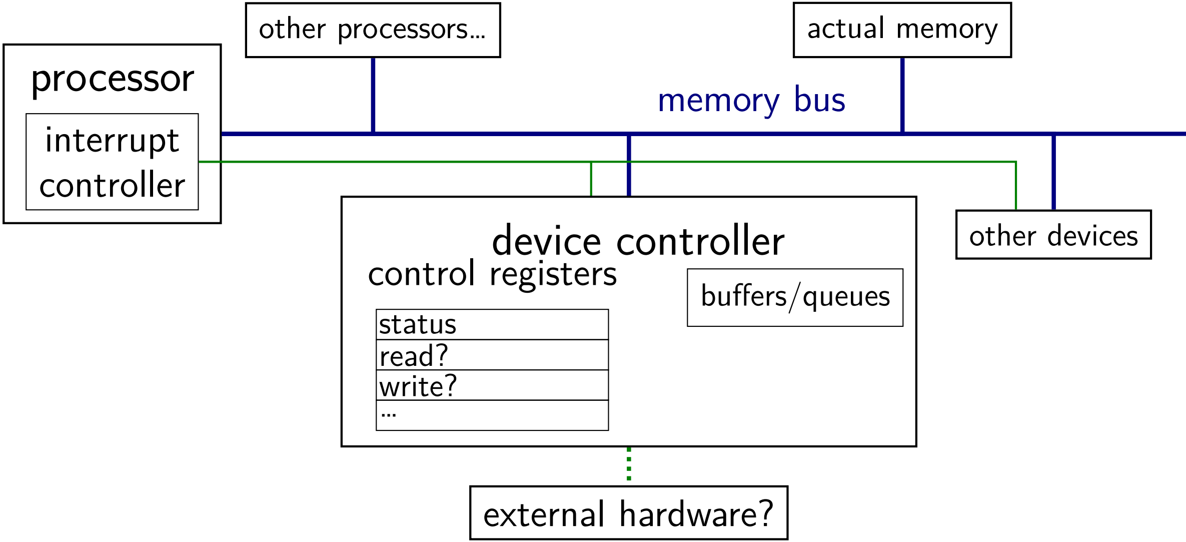
xv6: console interrupt reading

kbdintr fuction actually reads from device

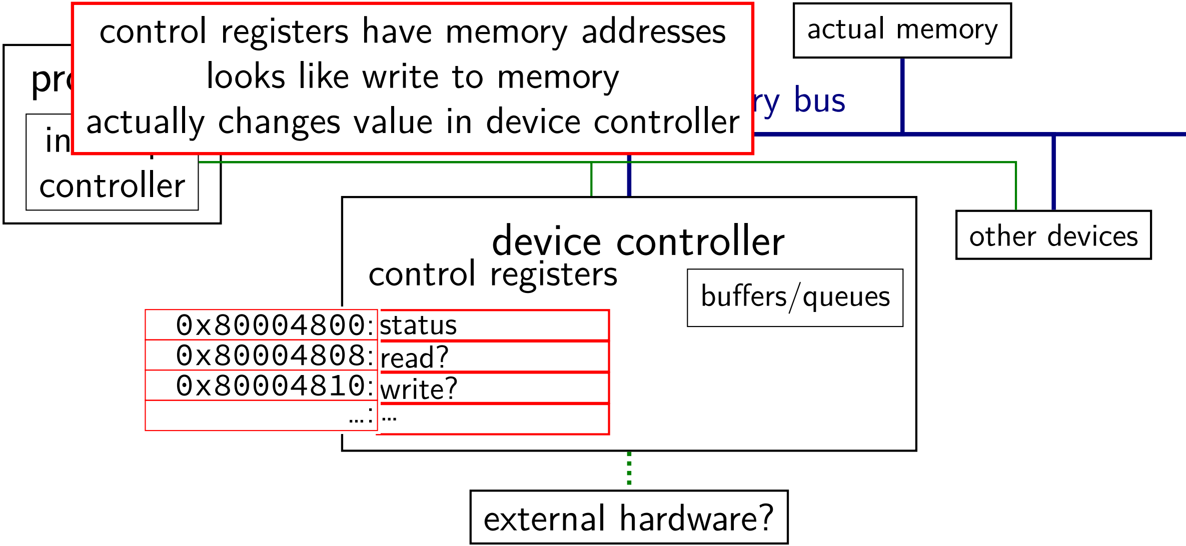
adds data to buffer (if room)

wakes up sleeping thread (if any)

connecting devices

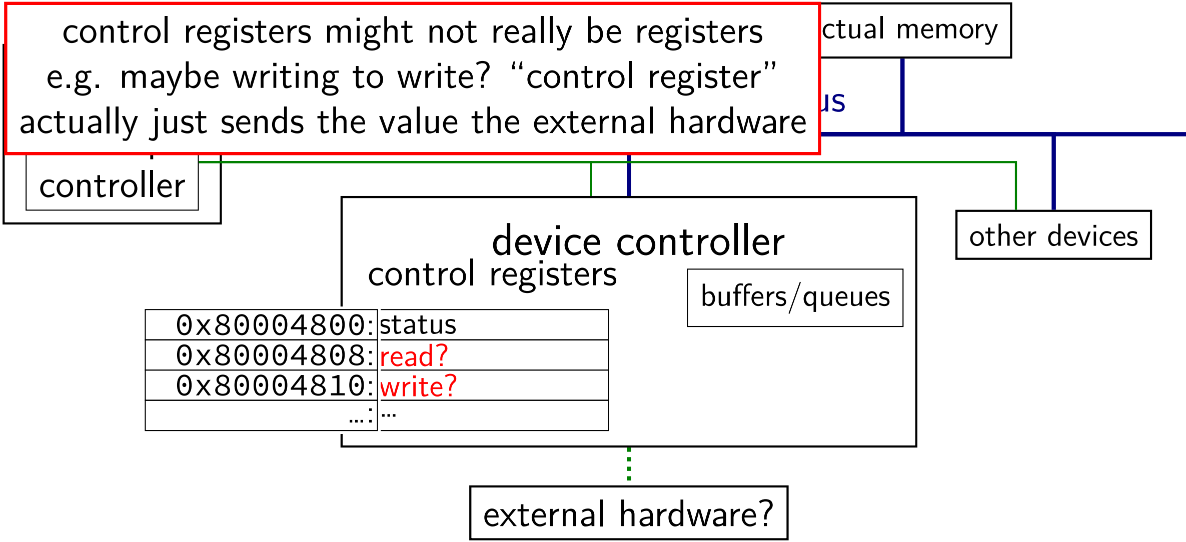


connecting devices

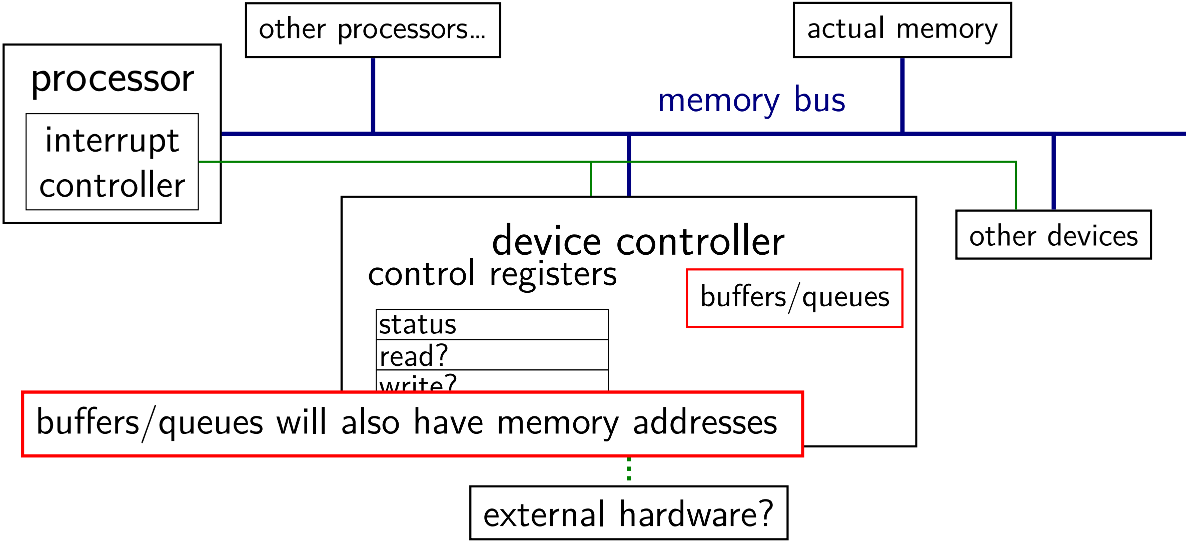


connecting devices

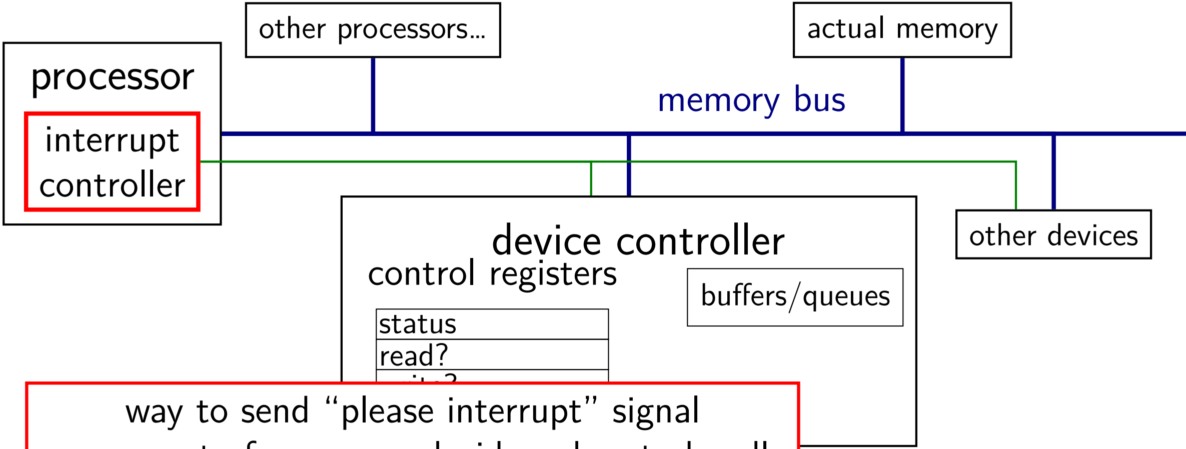
control registers might not really be registers
e.g. maybe writing to write? "control register"
actually just sends the value the external hardware



connecting devices

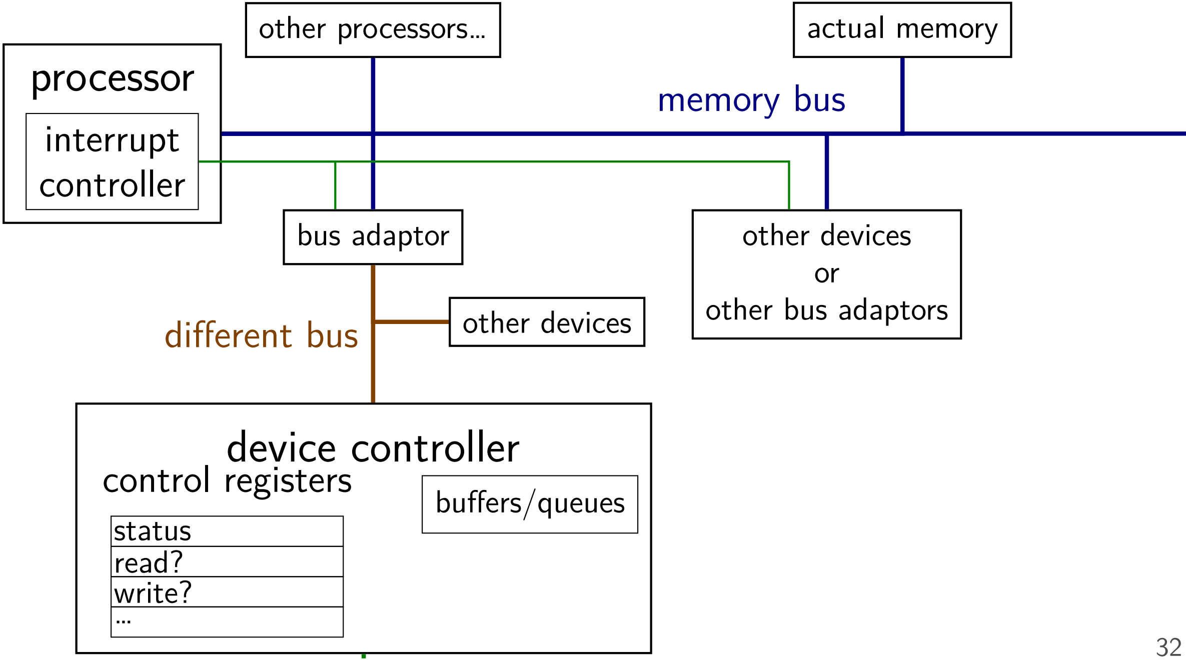


connecting devices



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 */
```

exercise

system is running two applications

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

timeline:

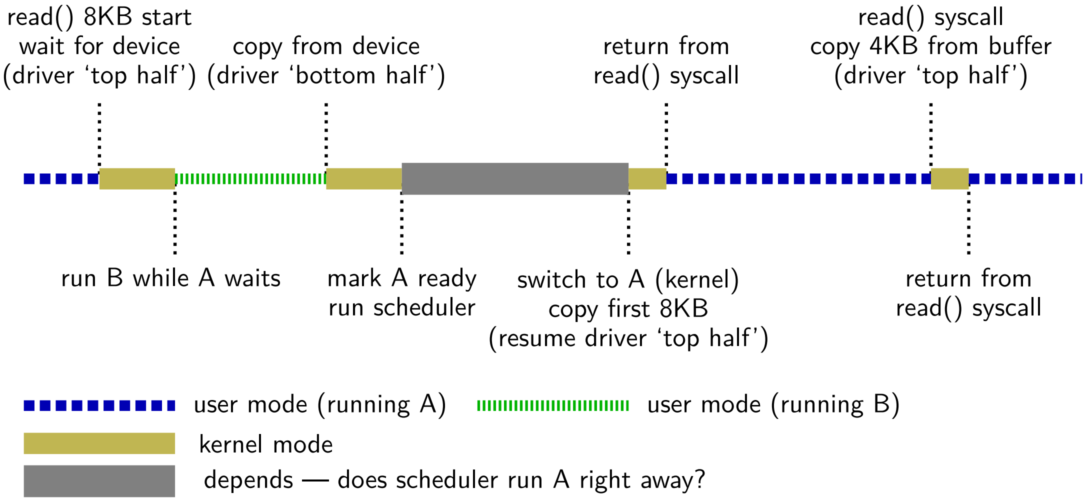
- A calls read() to 8KB of data from network
  - not immediately available
- 16KB of data comes in 10ms later
- A calls read() again to get 4KB more

exercise 1: how many kernel/user mode switches?

exercise 2: how many context switches?

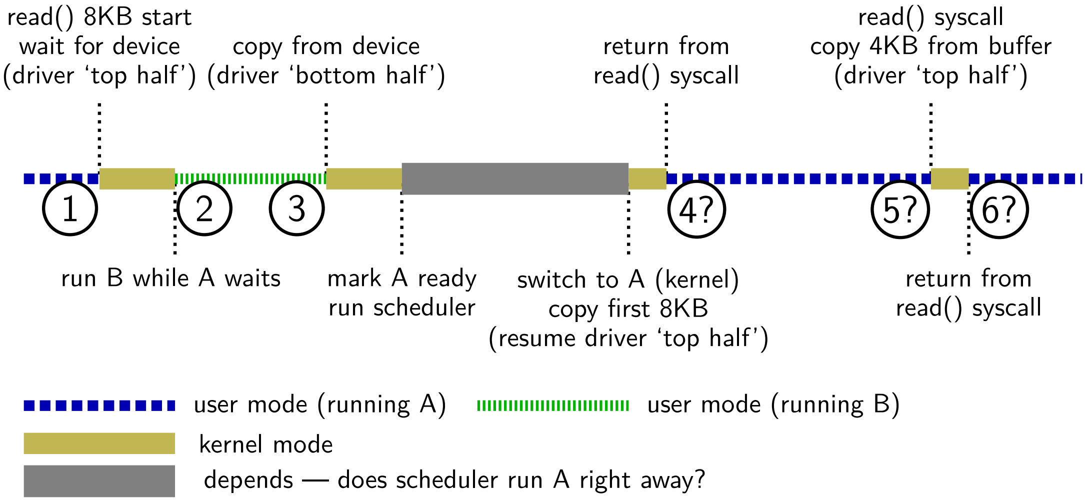
how many mode switches?

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



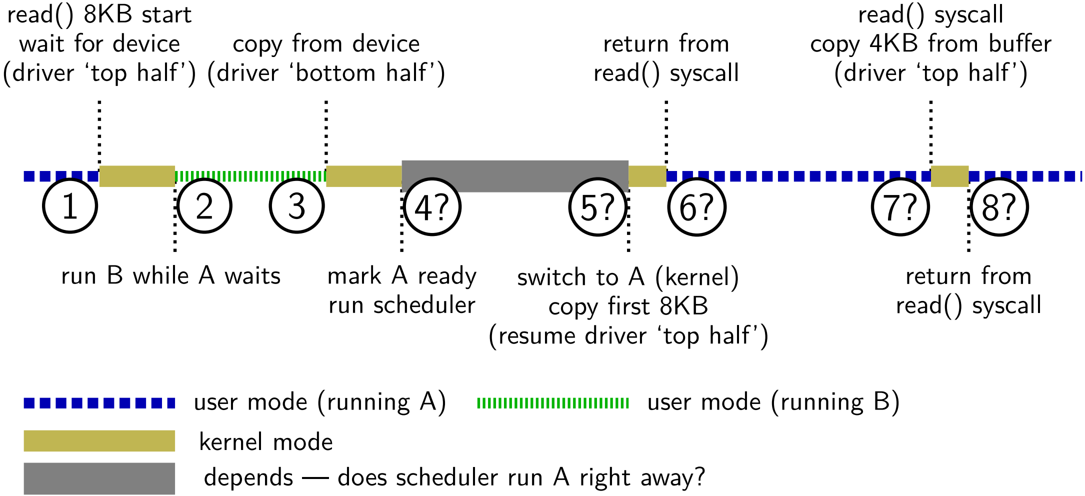
how many mode switches?

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



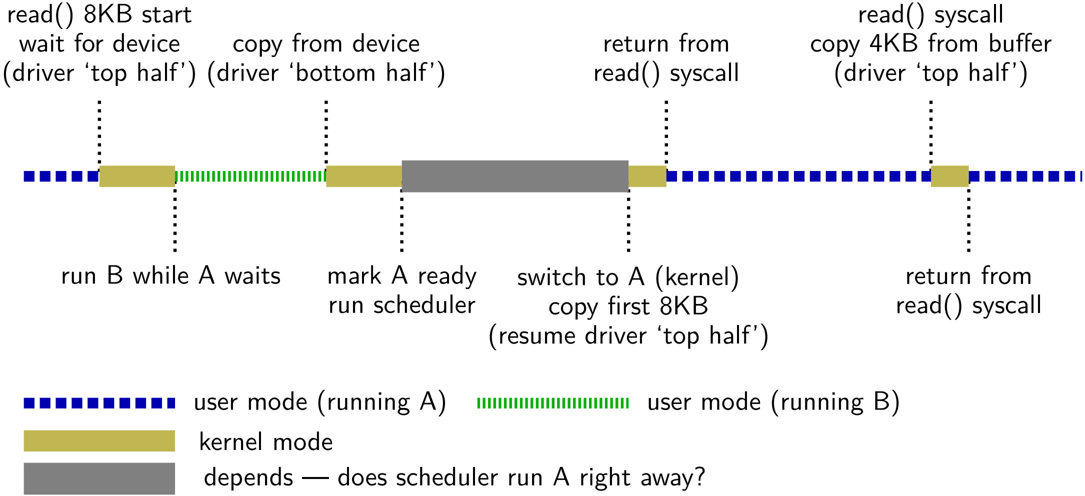
how many mode switches?

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



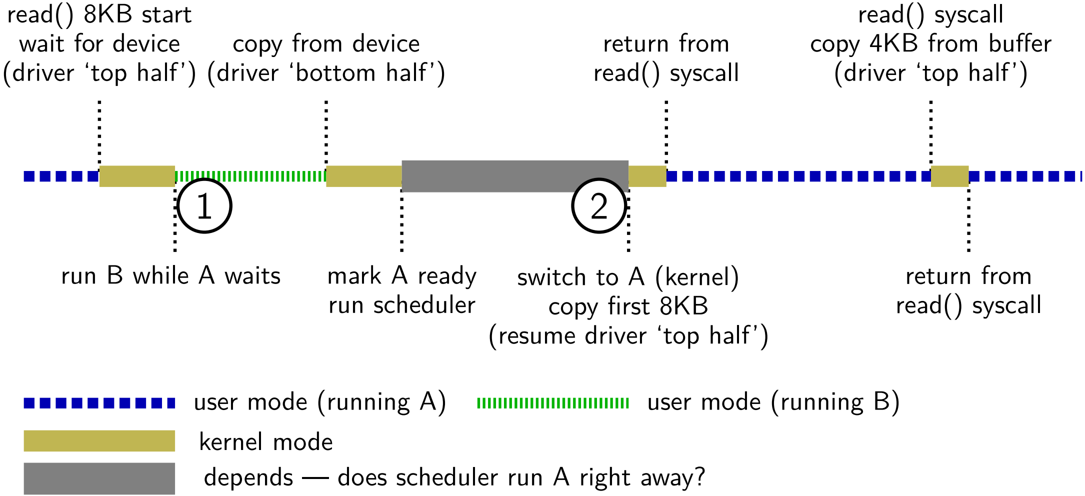
how many context switches?

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



programmed I/O

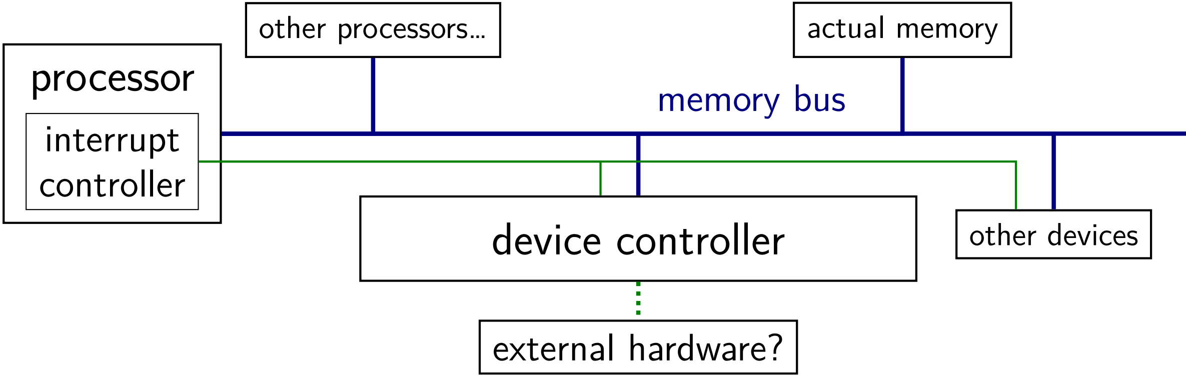
"programmed I/O": write to or read from device controller buffers directly

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?

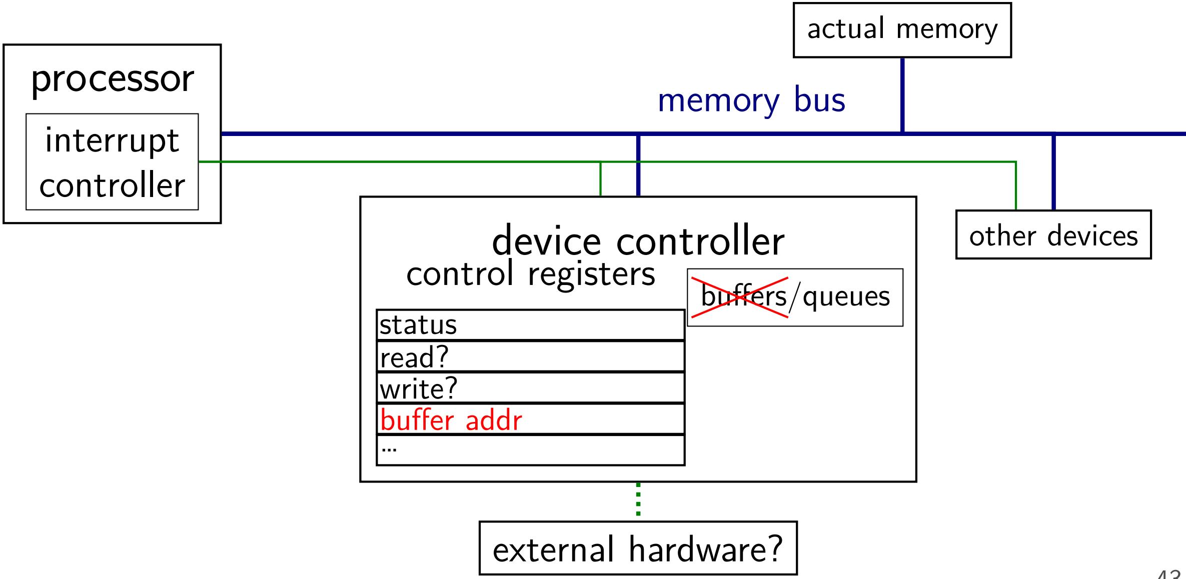
direct memory access (DMA)



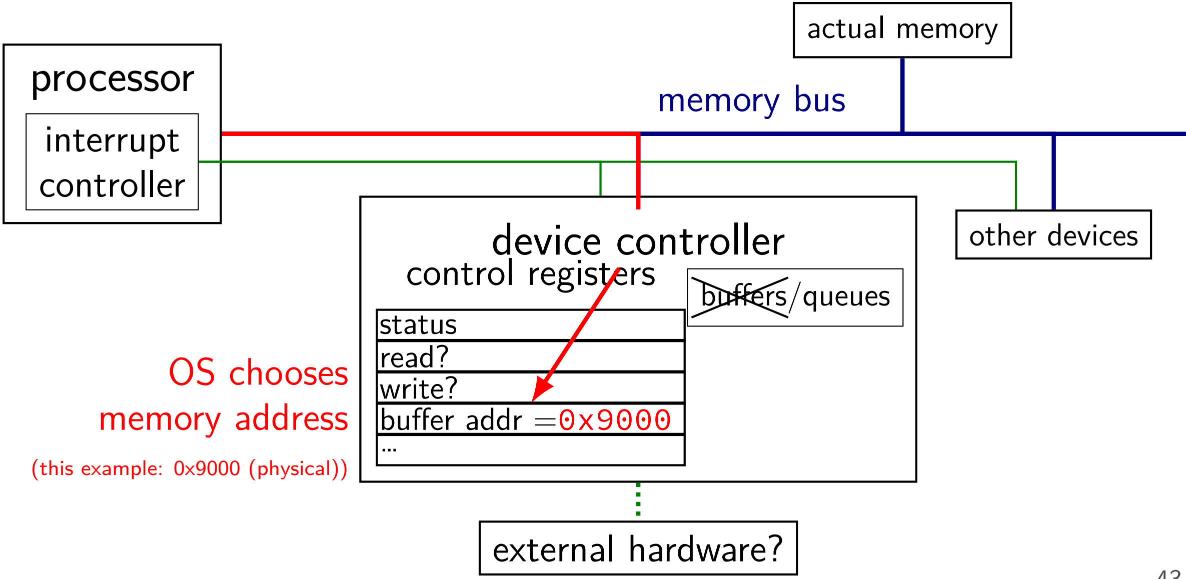
observation: devices can read/write memory

can have device copy data to/from memory

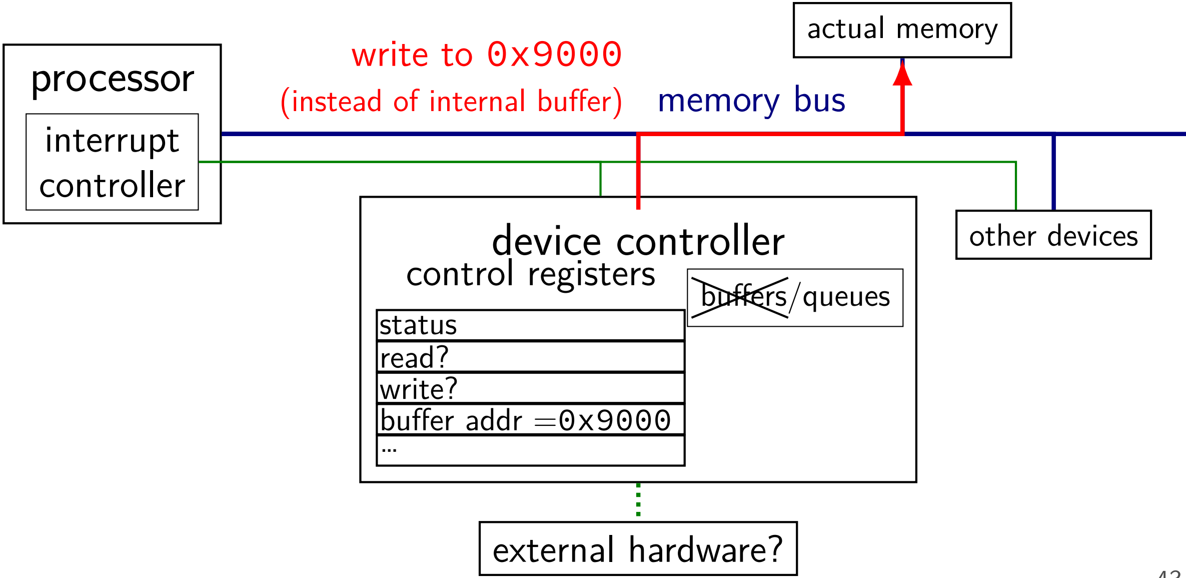
direct memory access (DMA)



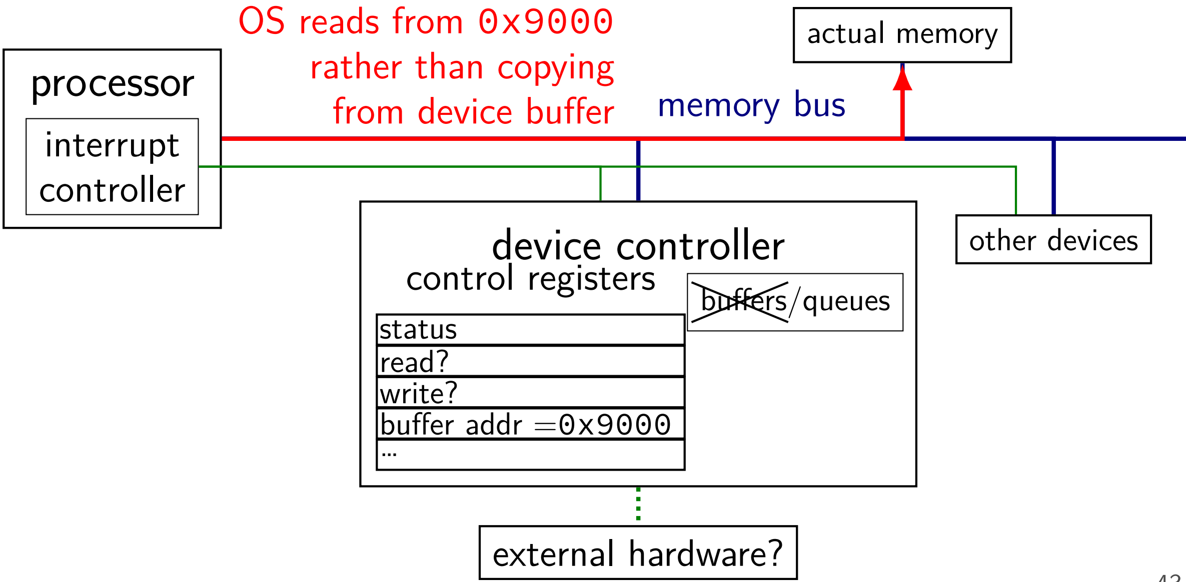
direct memory access (DMA)



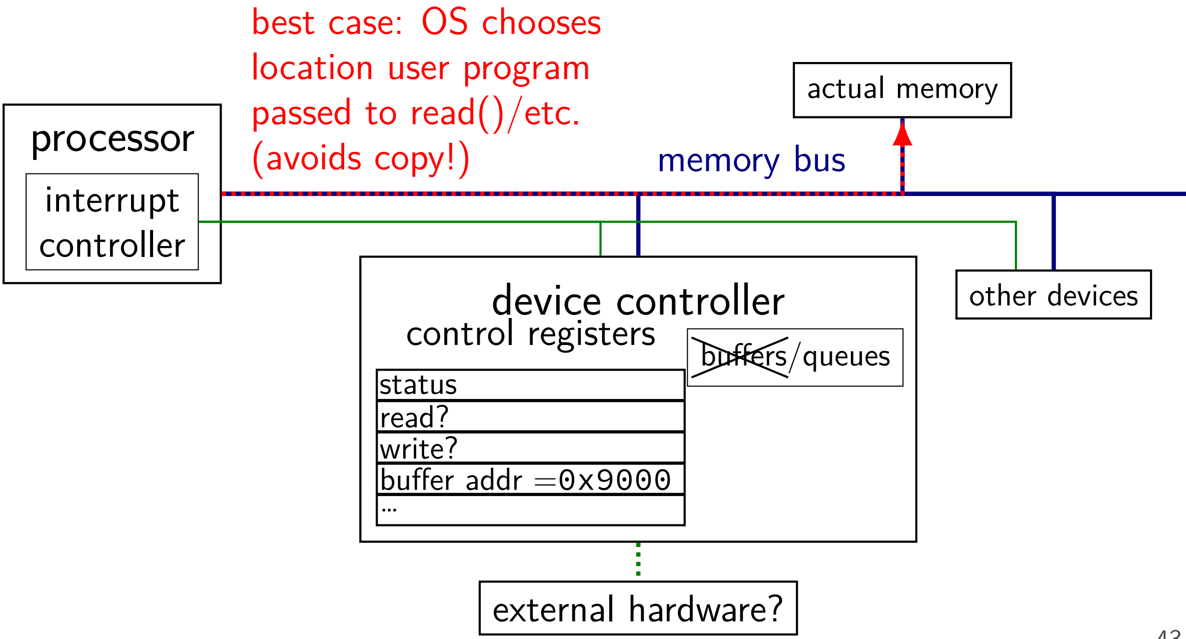
direct memory access (DMA)



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
- ...

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