virtual memory 5 / devices

last time

page replacement metrics
- optimizing hit rate
- really care about throughput
- other possibilities (like processor scheduling)

Belady's MIN: ideal hit rate policy
- replace what is accessed furthest in future

working set model: subset of memory in use

LRU policy: possible approximation of Belady's MIN
- ...assuming working set model/temporal locality

practical approx of LRU: second chance, SEQ
- key idea: check if accessed in time window

lazy replacement?

so far: don't do anything special until memory is full

only then is there a reason to writeback pages or evict pages

lazy replacement?

so far: don't do anything special until memory is full

only then is there a reason to writeback pages or evict pages

but real OSes are more proactive

non-lazy writeback

what happens when a computer loses power

how much data can you lose?

if we never run out of memory...all of it?

- no changed data written back

solution: track or scan for dirty pages and writeback

example goals:

- lose no more than 90 seconds of data
- force writeback at file close
- ...

non-lazy eviction

so far — allocating memory involves evicting pages

hopefully pages that haven't been used a long time anyways

non-lazy eviction

so far — allocating memory involves evicting pages

hopefully pages that haven't been used a long time anyways

alternative: evict earlier "in the background"

- "free": probably have some idle processor time anyways

allocation = remove already evicted page from linked list

- (instead of changing page tables, file cache info, etc.)

problems with LRU

question: when does LRU perform poorly?

exercise: which of these is LRU bad for?

code in a text editor for handling out-of-disk-space errors

initial values of the shell's global variales

on a desktop, long movies that are too big to fit in memory and played from beginning to end

on web server, long movies that are too big to fit in memory and frequently downloaded by clients

files that are parsed when loaded and overwritten when saved

on web server, frequently requested HTML files

problems with LRU

question: when does LRU perform poorly?

only reading things once

repeated scans of large amounts of data

problems with LRU

question: when does LRU perform poorly?

only reading things once

repeated scans of large amounts of data

both common access patterns for files

CLOCK-Pro: special casing for one-use pages

by default, Linux tries to handle scanning of files

- one read of file data — e.g. play a video, load file into memory

basic idea: delay considering pages active until second access

- second access = second scan of accessed bits/etc.

single scans of file won't "pollute" cache

without this change: reading large files slows down other programs

- recently read part of large file steals space from active programs

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

readahead implementation ideas?

which of these is probably best?

(a) when there's a page fault requring reading page X of a file from disk, read pages X and $X + 1$

(b) when there's a page fault requring reading page $X > 200$ of a file from disk, read the rest of the file

(c) when page fault occurs for page X of a file, read pages X through $X + 200$ and proactively add all to the current program's page table

(d) when page fault occurs for page X of a file, read pages X through $X + 200$ but don't place pages $X + 1$ through $X + 200$ in the page table yet

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

when to start reads?

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

- takes some time to read in data — well before needed

how much to readahead?

readahead heuristics

exercise: devise an algorithm to detect to do readahead.

how to detect the reading pattern?

- need to record subset of accesses to see sequential pattern
- not enough to look at misses!
- want to check when readahead pages are used — keep up with program

when to start reads?

- takes some time to read in data — well before needed

how much to readahead?

- if too much: evict other stuff programs need
- if too little: won't keep up with program
- if too little: won't make efficient use of HDD/SSD/etc.

page cache/replacement summary

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

mostly, assume working set model

- keep (hopefully) small active set in memory
- least recently used variants

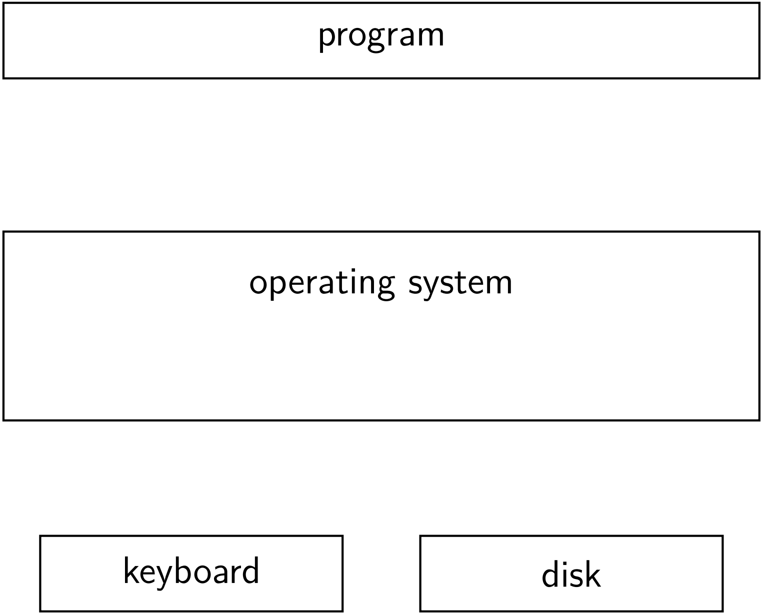
special cases for non-LRU-friendly patterns (e.g. scans)

- maybe more we haven't discussed?

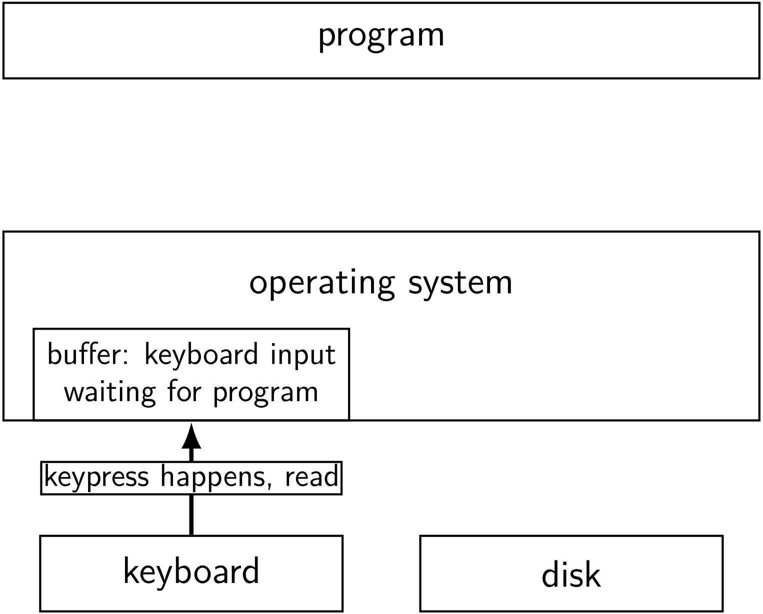
being proactive (writeback early, readahead, pre-evicted pages)

missing: handling non-miss-rate goals?

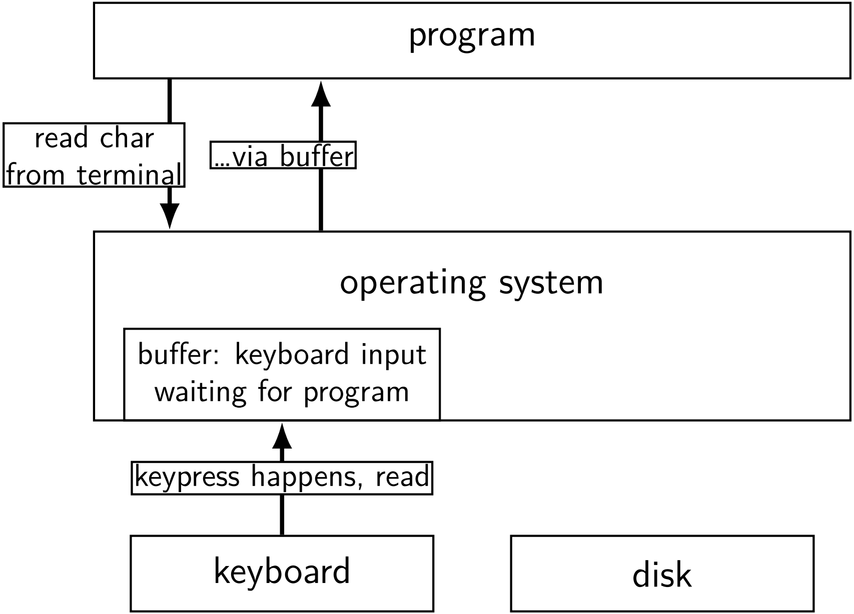
recall: kernel buffering (reads)



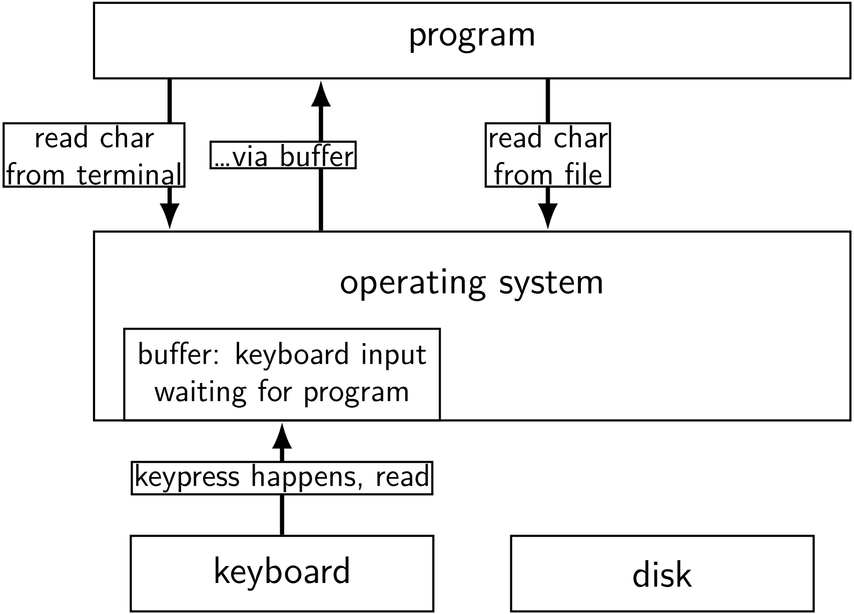
recall: kernel buffering (reads)



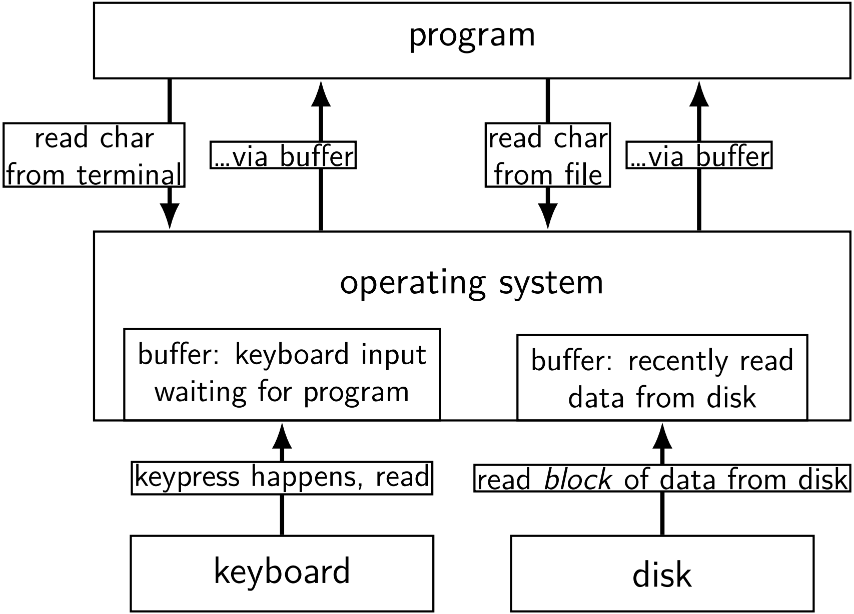
recall: kernel buffering (reads)



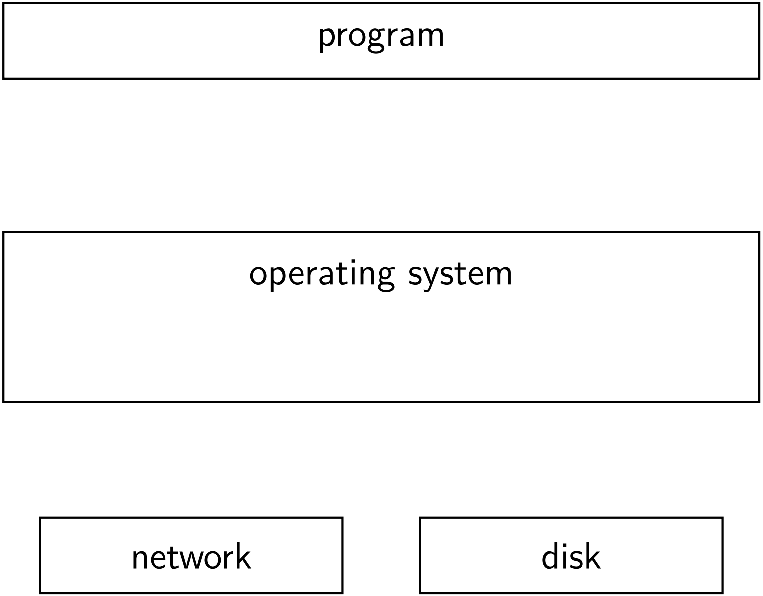
recall: kernel buffering (reads)



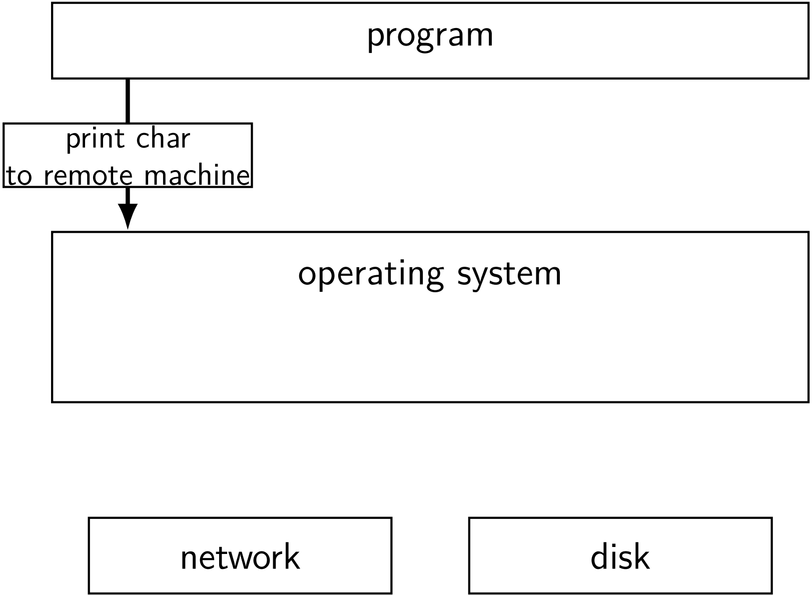
recall: kernel buffering (reads)



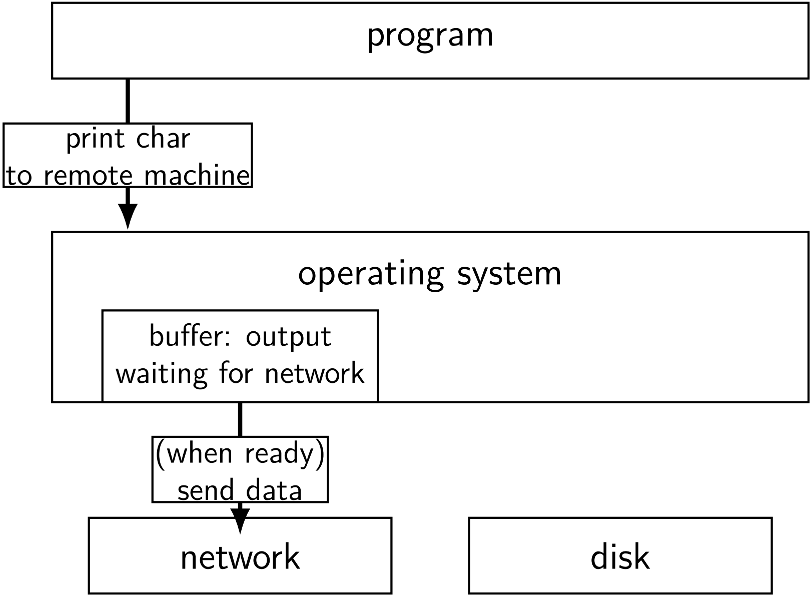
recall: kernel buffering (writes)



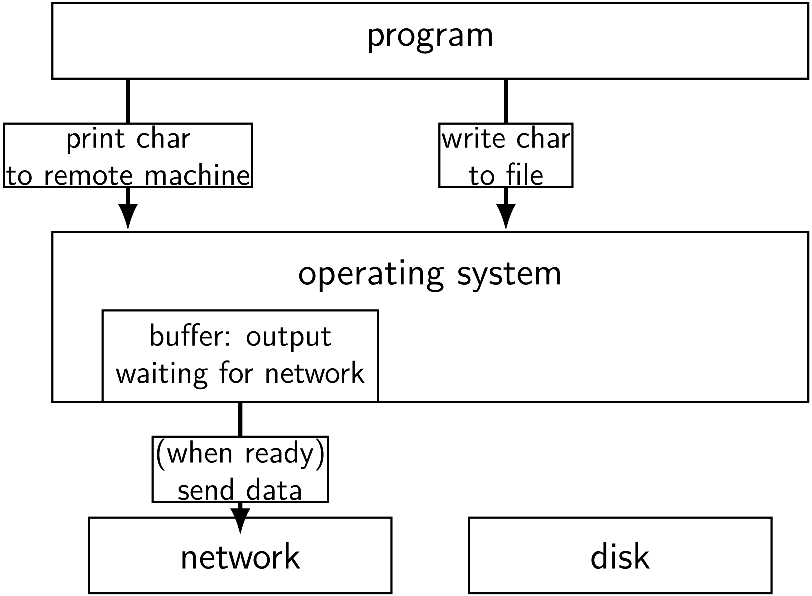
recall: kernel buffering (writes)



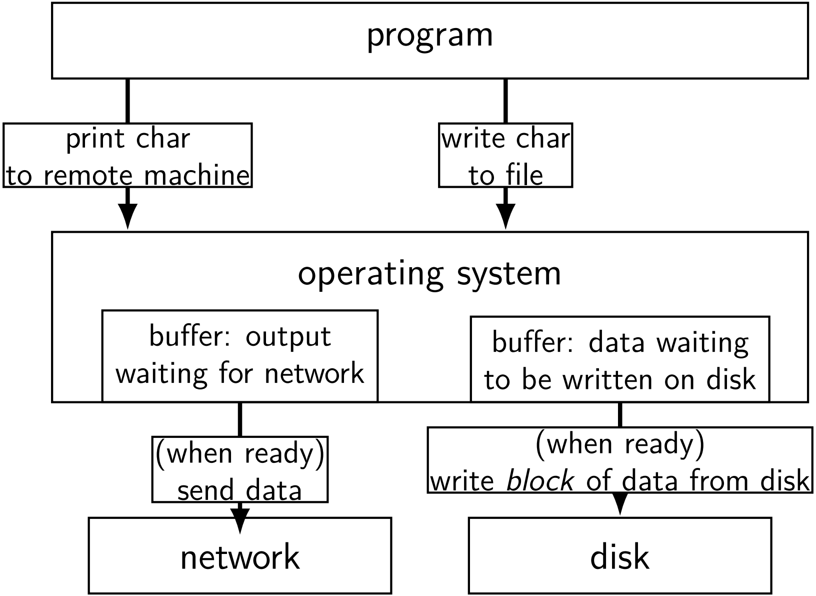
recall: kernel buffering (writes)



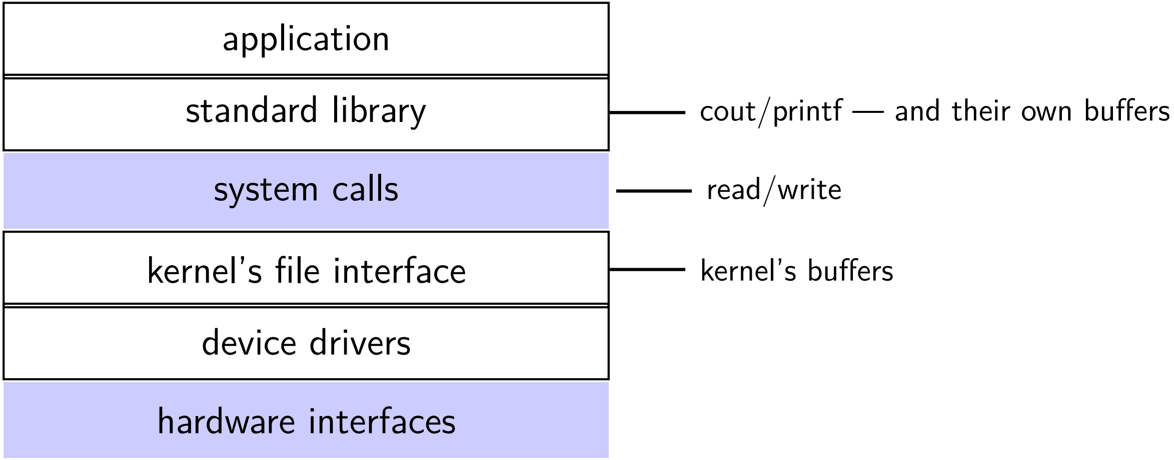
recall: kernel buffering (writes)



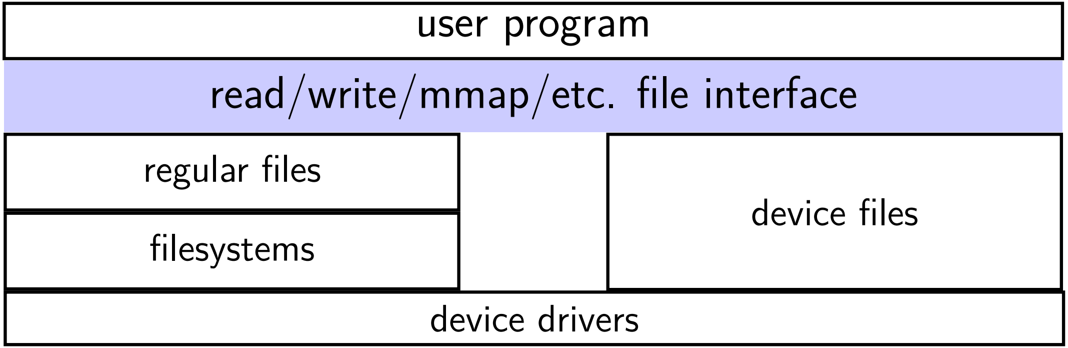
recall: kernel buffering (writes)



recall: layering



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

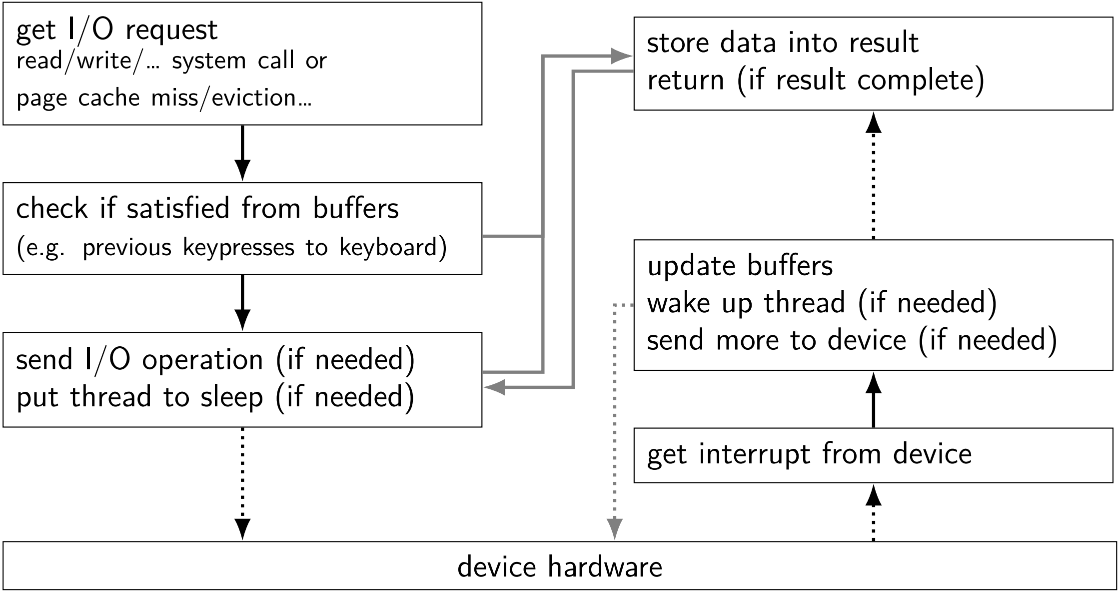
- read/write page at a time

Linux example: block device operations

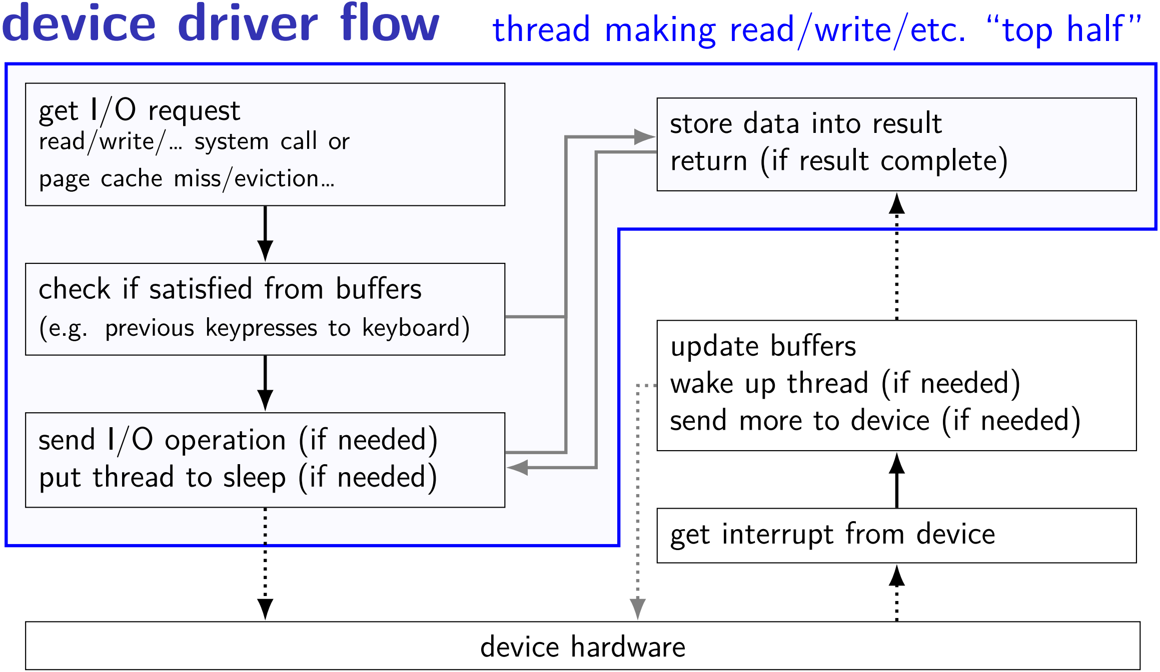
```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *,
            sector_t, struct page *, bool);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, un
    ...
};
```

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

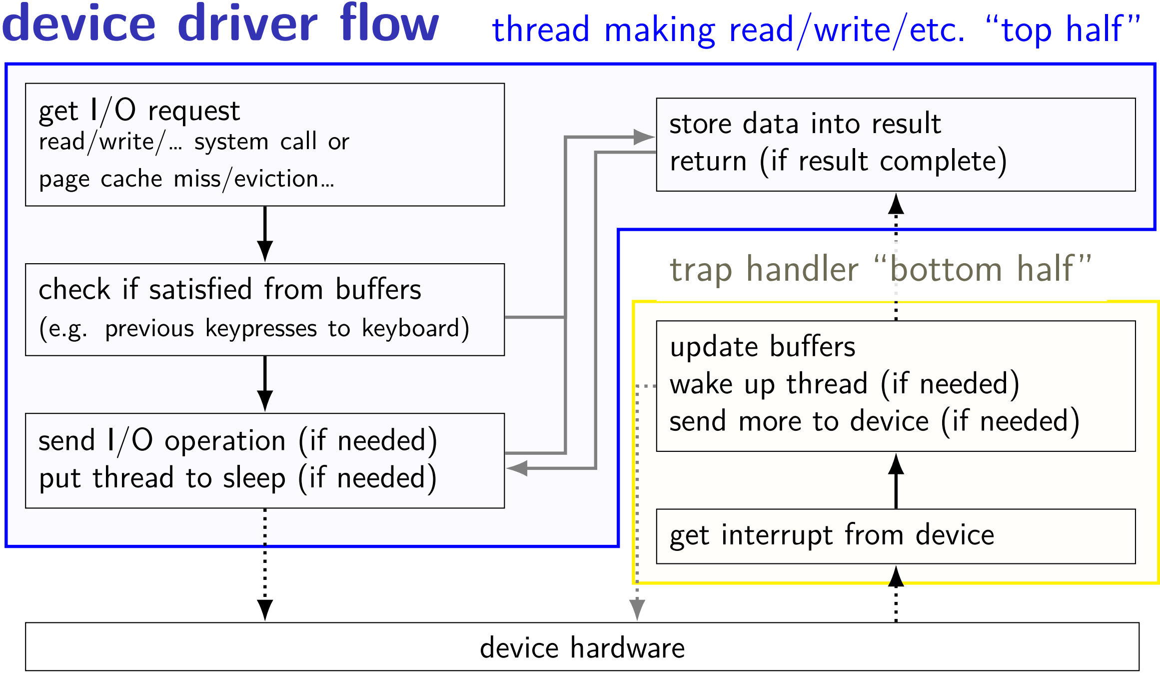
device driver flow



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

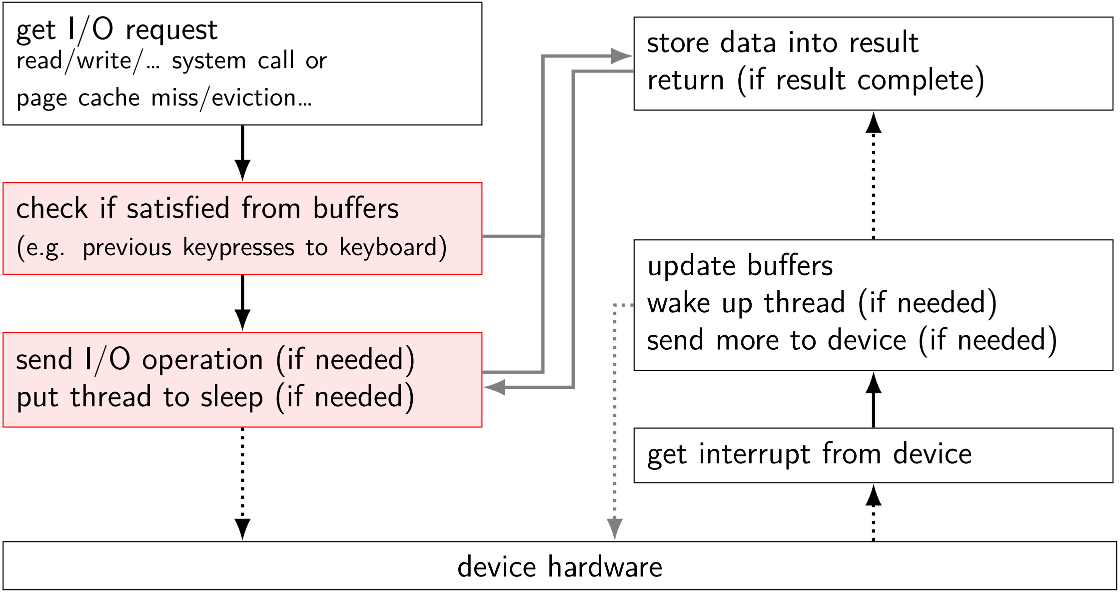
code run at boot:

```
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)

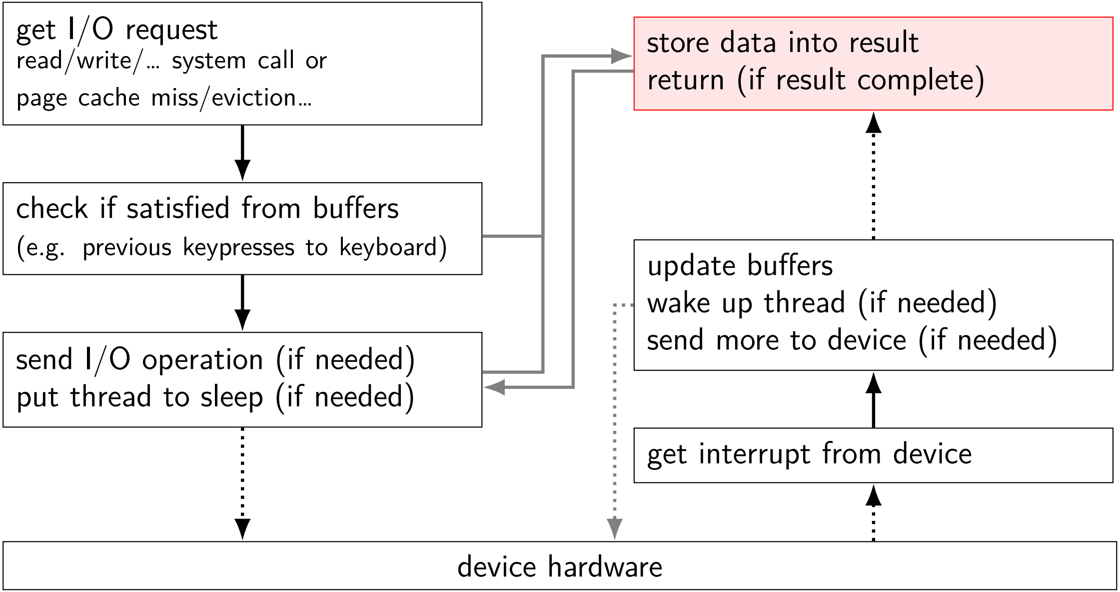
```
int
consoleread(struct inode *ip, char *dst, int n)
{
   ...
   target = n;
   acquire(&cons.lock);
   while(n > 0){
     while(input.r == input.w){
       if(myproc()->killed){
         ...
         return -1;
       }
       sleep(&input.r, &cons.lock);
     }
     ...
   }
   release(&cons.lock)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

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

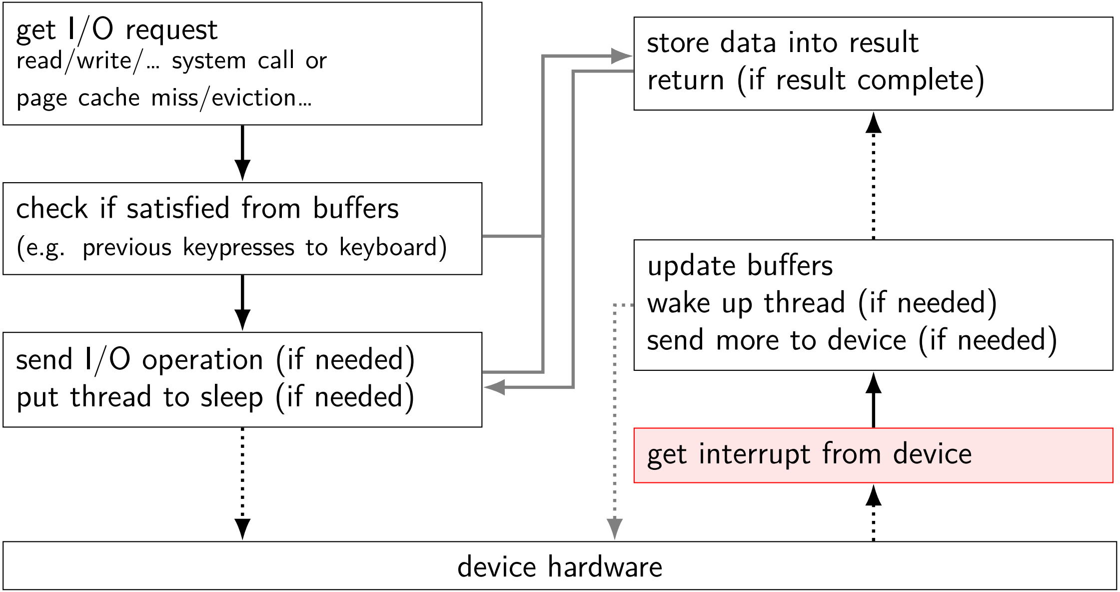
wait for buffer to fill

- 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”

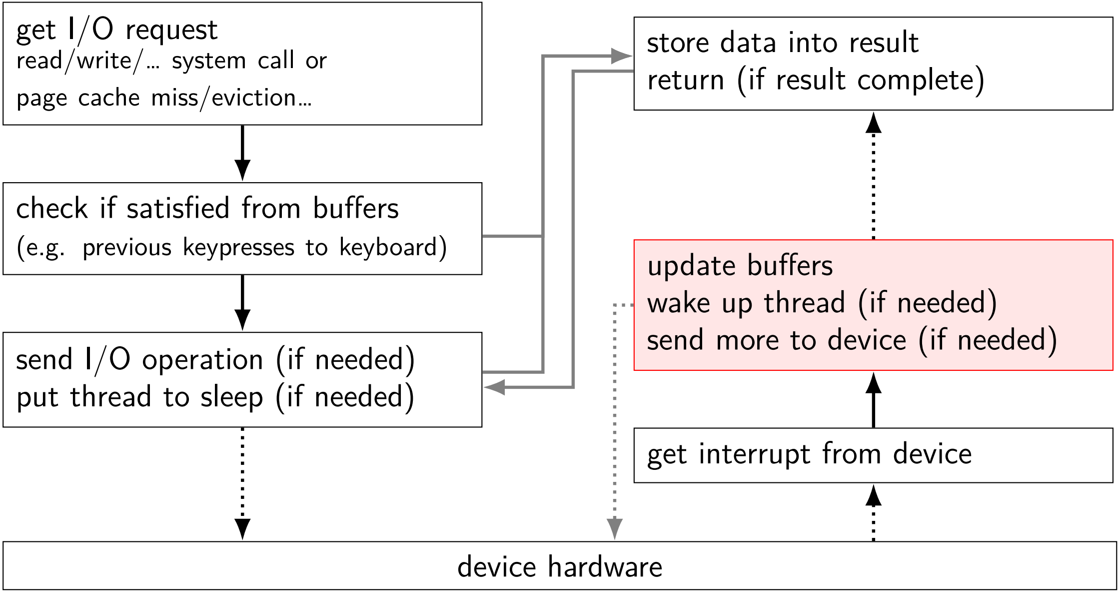
xv6: console interrupt (one case)

```
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



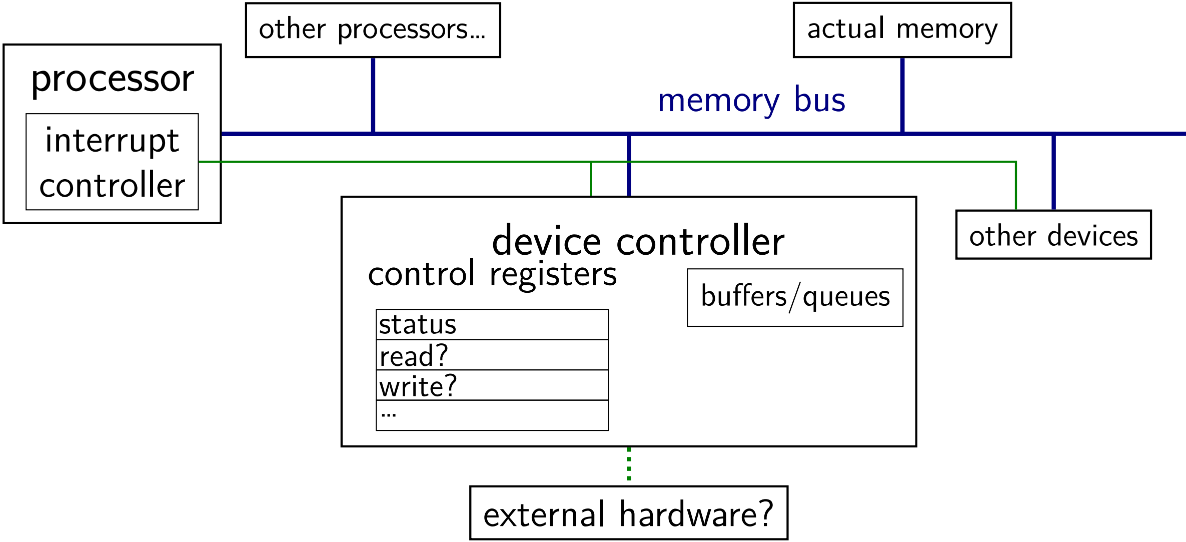
xv6: console interrupt reading

kbdintr fuction actually reads from device

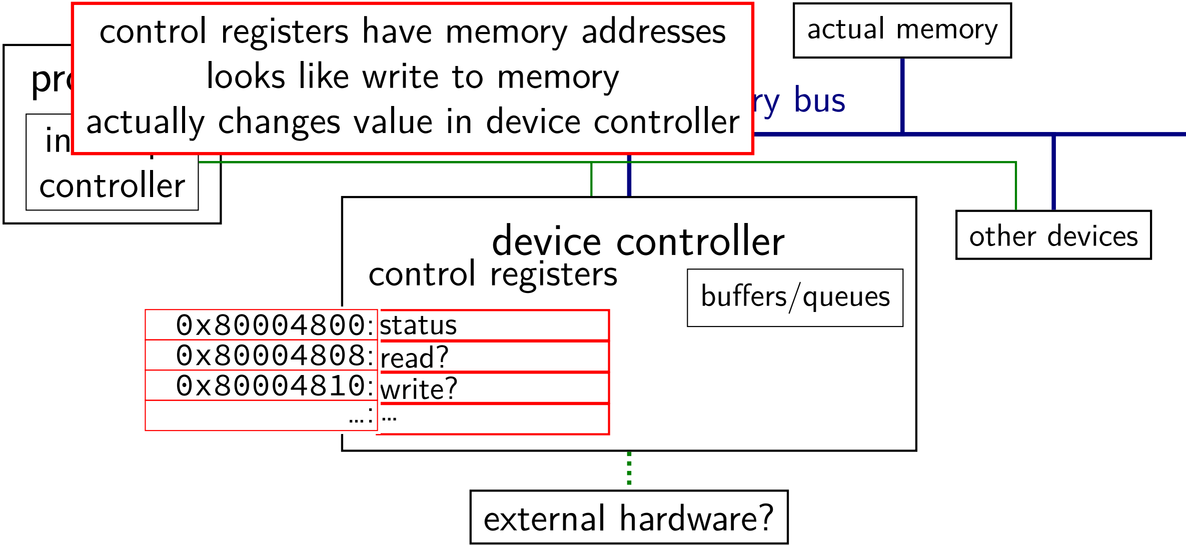
adds data to buffer (if room)

wakes up sleeping thread (if any)

connecting devices

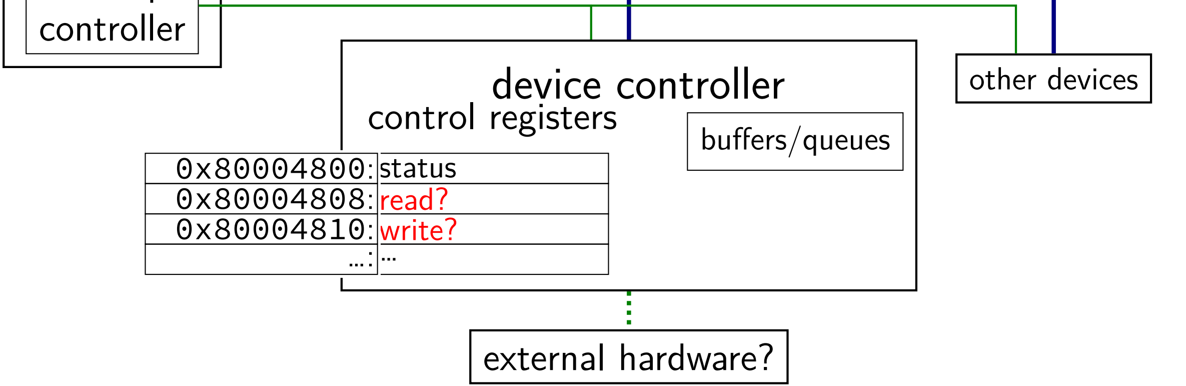


connecting devices

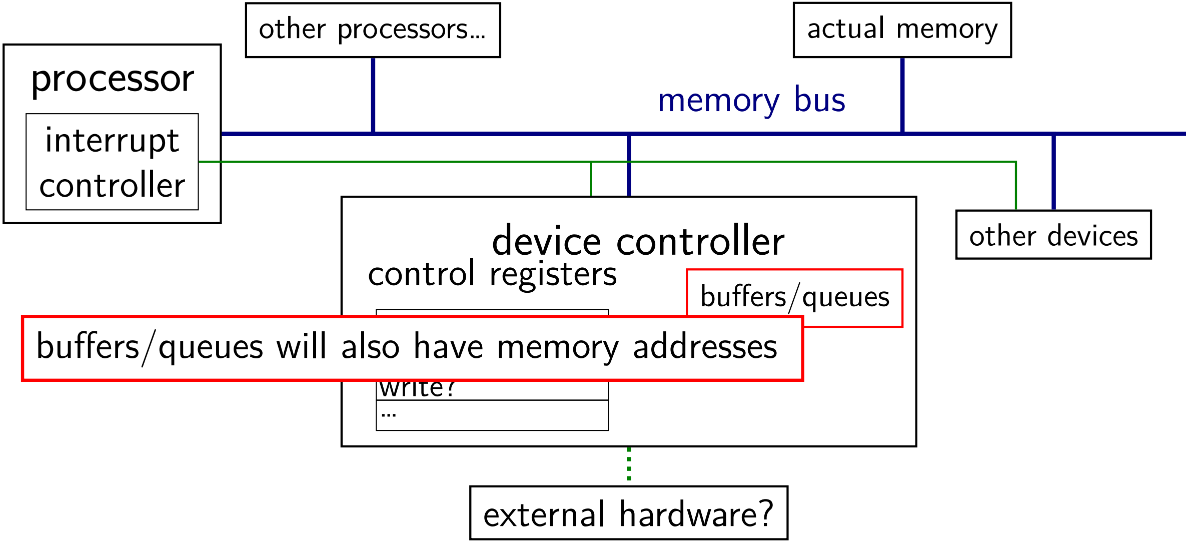


connecting devices

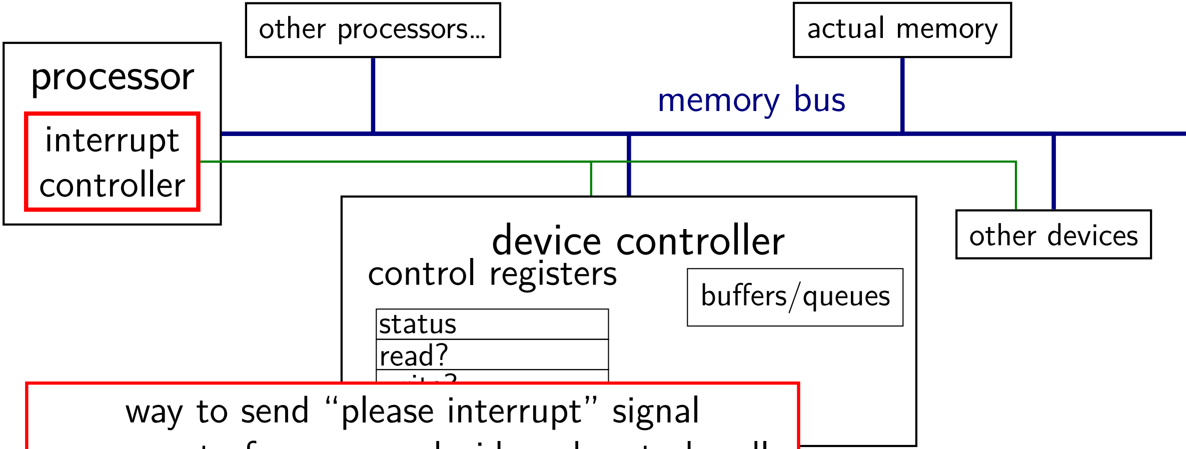
control registers might not really be registers
e.g. maybe writing to write? "control register" actually just sends the value the external hardware



connecting devices

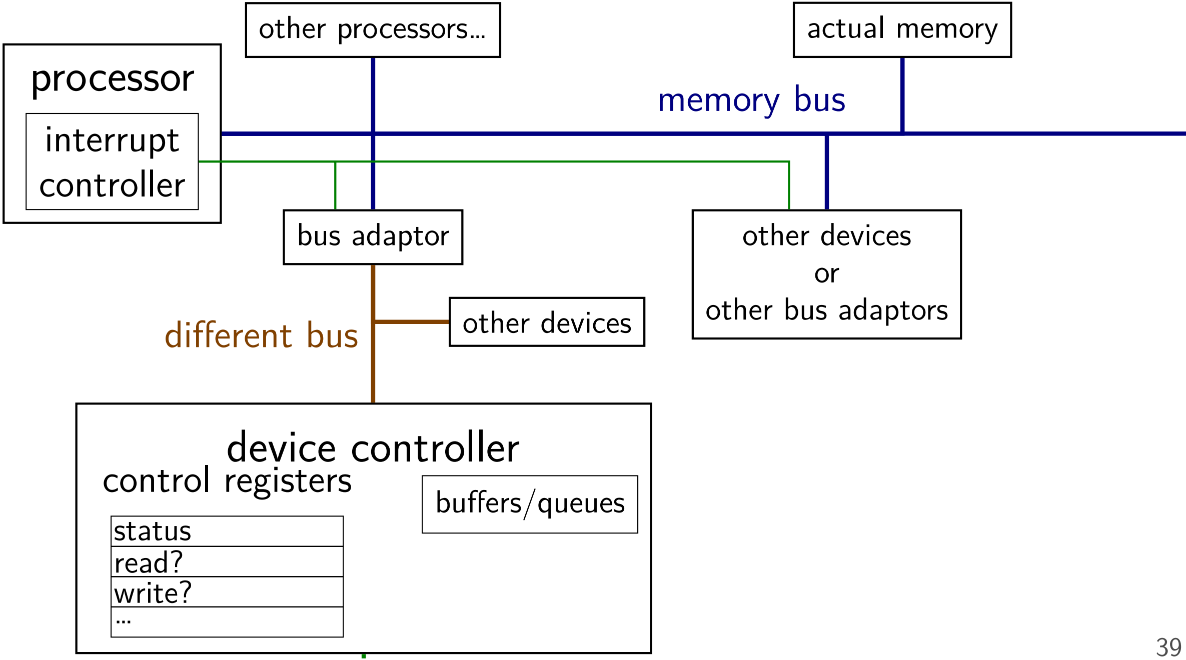


connecting devices



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 */
```

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?

backup slides

‘fair’ page replacement

so far: page replacement about least recently used

what about sharing fairly between users?

sharing fairly?

process A
- 4MB of stack+code, 16MB of heap
- shared cached 24MB file X

process B
- 4MB of stack+code, 16MB of heap
- shared cached 24MB file X

process C
- 4MB of stack+code, 4MB of heap
- cached 32MB file Y

process D+E
- 4MB of stack+code (each), 70MB of heap (each)
- but all heap + most of code is shared copy-on-write

accounting pages

shared pages make it difficult to count memory usage

Linux *cgroups* accounting (mostly): last touch

- count shared file pages for the process that last 'used' them
- ...as detected by page fault for page

then can set per-group (set of process) limits based on this

...and choose victim page based on limits + LRU approximation

Linux readahead heuristics — how much

how much to readahead?

Linux heuristic: count number of cached pages from before

guess we should read about that many more

(plus minimum/maximum to avoid extremes)

goal: readahead more when applications are using file more

goal: don't readahead as much with low memory

Linux readahead heuristics — when

track “readahead windows” — pages read because of guess:

```
                    |<----- async_size ---------|
|------------------- size -------------------->|
|====================#==============================|
 ^start              ^page marked with PG_readahead
```

when `async_size` pages left, read next chunk

marked page = detect reads to this page

- one option: make page temporary invalid

idea: keep up with application, but not too far ahead

ASCII art figure: comments of Linux readahead code