Virtual Memory 3 / I/O

last time

working set, Zipf usage models

LRU page replacement

approximating LRU by sampling accessed bits or mark invalid

- nit: said Linux marked invalid to test — probably not on x86
  instead periodic scanning of referenced bits set by processor
  (but marking invalid would work/is needed on some platforms)

observation: when LRU fails

on the paging assignment

"(and pointing to the the original physical page) (pointing to the same physical page) Do those two lines mean the same thing?????"

yes — with copy-on-write, the child uses same pages as parent

differences are in how reference count and read-onlyness is maintained when parent page was already copy-on-write from a previous fork

anonymous feedback (1)

"hi can u stop changing the assignment description. just get it right the first time because every time you change, it screws with my understanding of what i'm supposed to do and i'm just super confused."

> I could try to add bullets instead of editing bullets if that's better... (and I did make more serious edits if you started before the assignment wasn't marked tentative, ...)

"also your instructions suck. they don't make sense."

> okay

anonymous feedback (2)

"Super unfair how Grimshaw's class gets 1 more week than we do on FAT homework because they don't have this paging assignment. While we struggle on this assignment, they get more time to figure out the next one"'

our FAT assignment is due 16 November
(checkpoint for ours is due 9 November)
theirs is due 8 November
should get some testing code with our version

problems with LRU

question: when does LRU perform poorly?

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

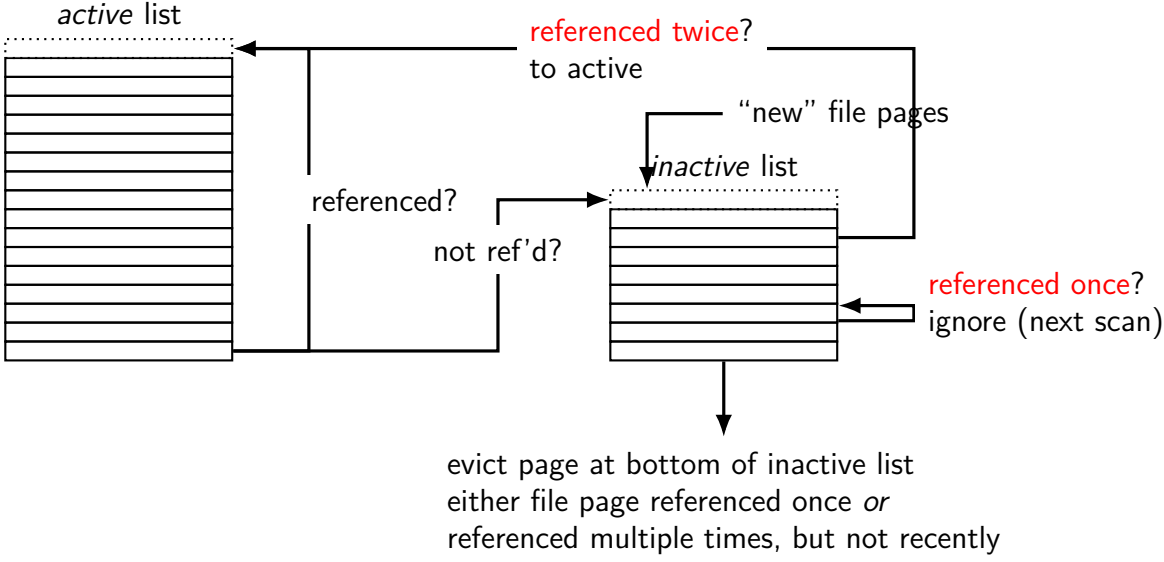
basic idea: don't consider pages active until the second access

single scans of file won't "pollute" cache

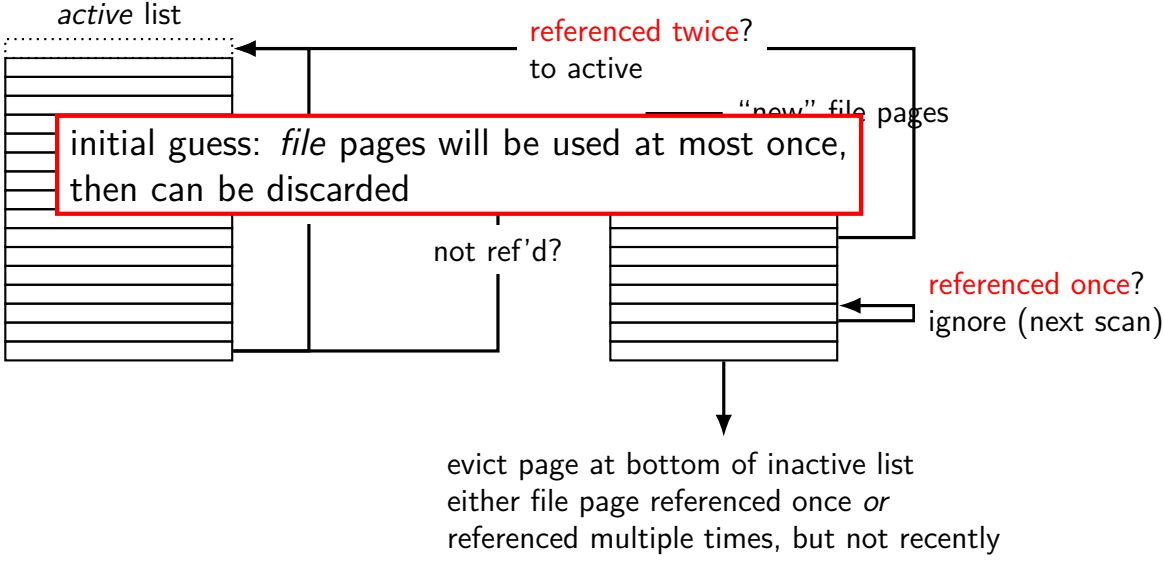
without this change: reading large files slows down other programs

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

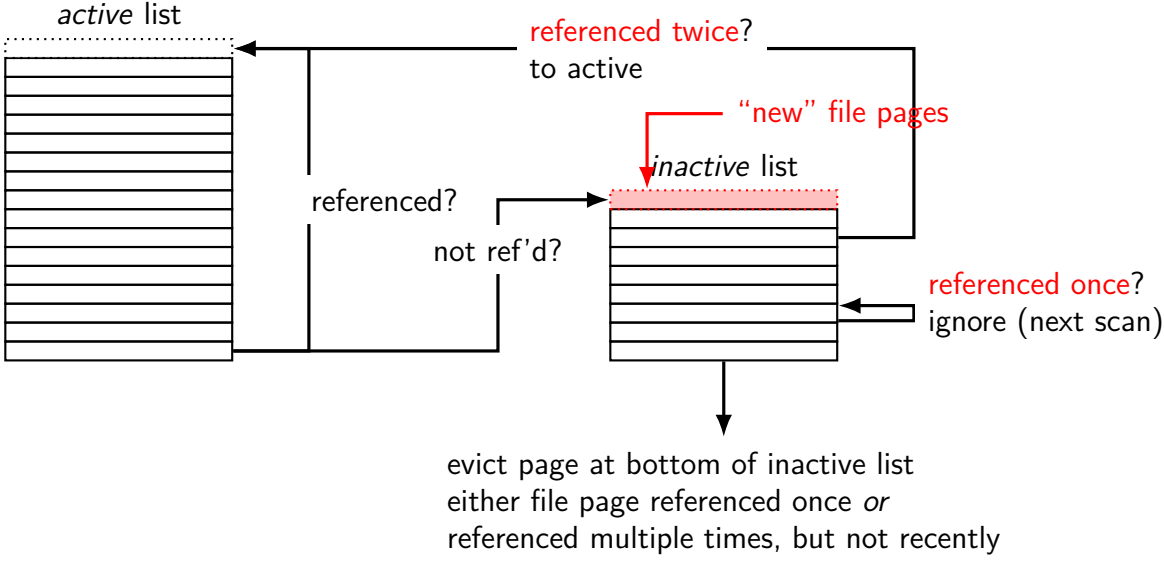
CLOCK-Pro: special casing for one-use pages



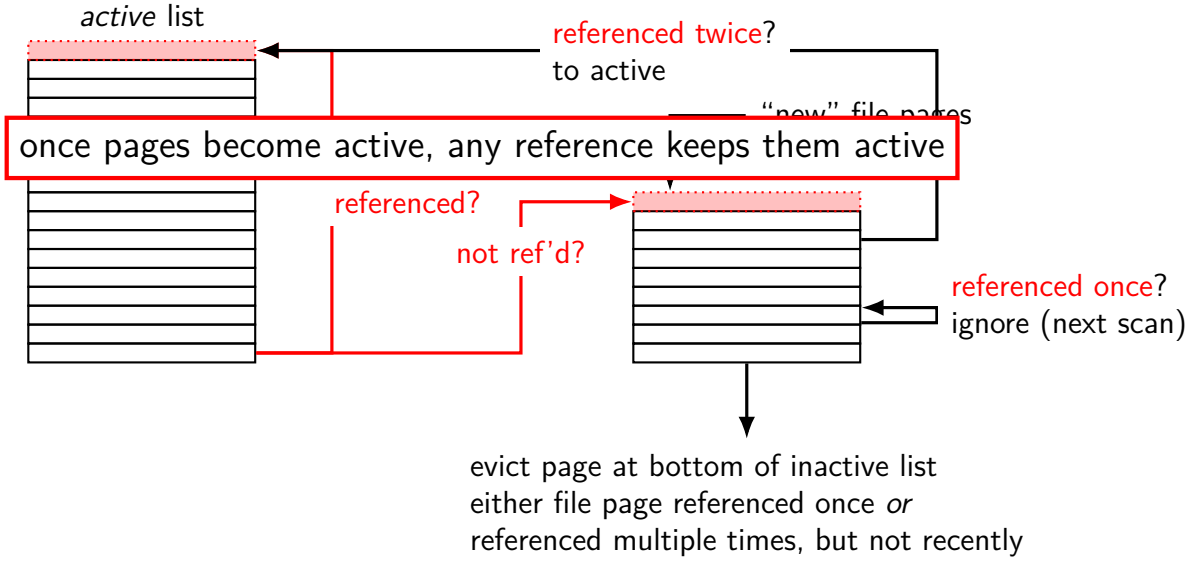
CLOCK-Pro: special casing for one-use pages



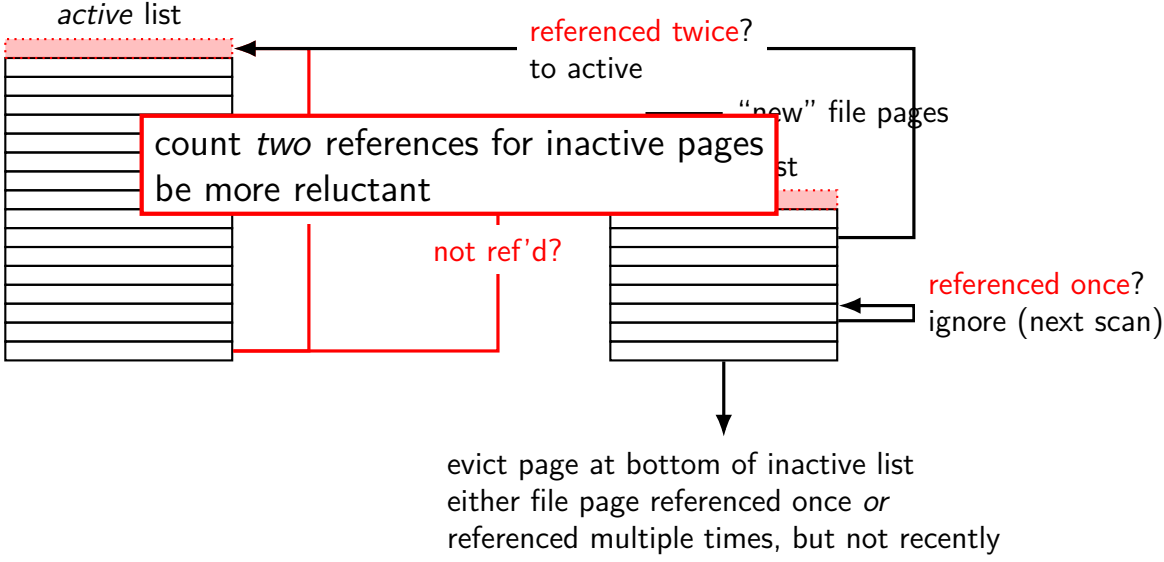
CLOCK-Pro: special casing for one-use pages



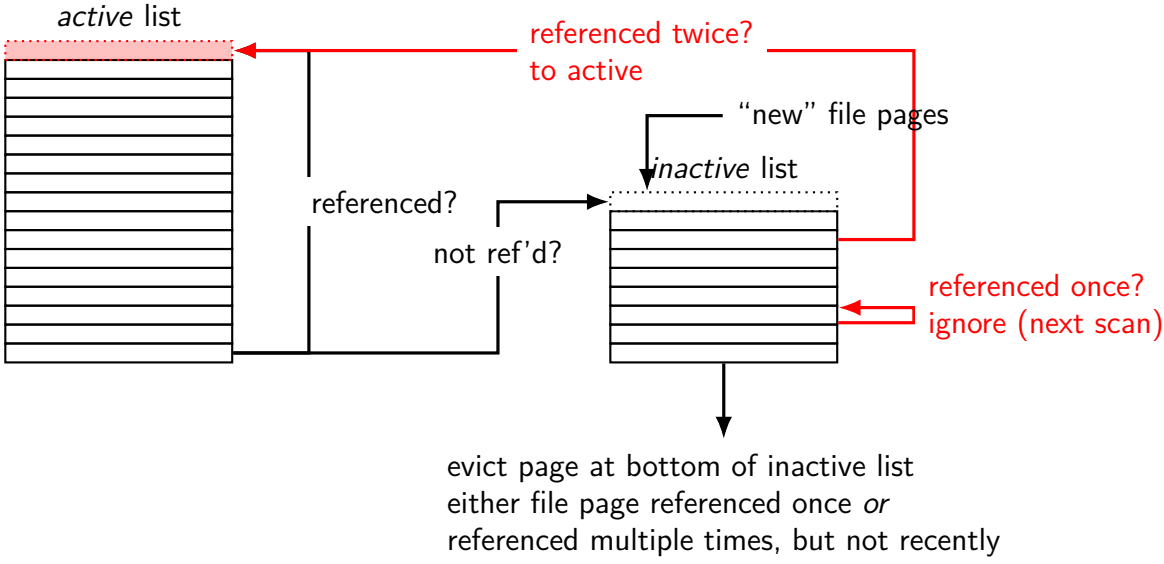
CLOCK-Pro: special casing for one-use pages



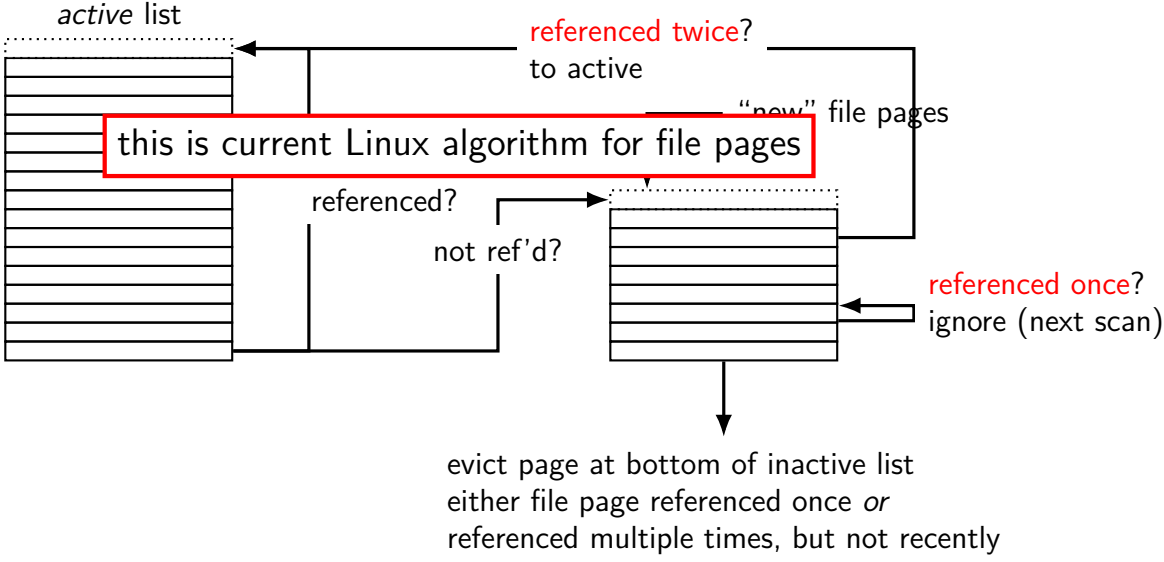
CLOCK-Pro: special casing for one-use pages



CLOCK-Pro: special casing for one-use pages



CLOCK-Pro: special casing for one-use pages



default Linux page replacement summary

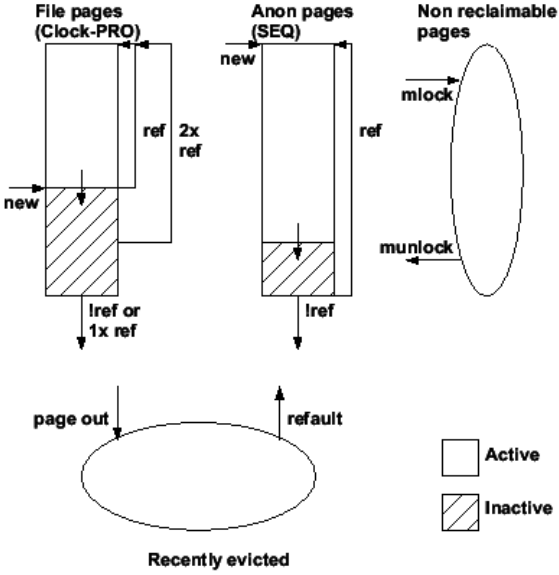


Figure: https://linux-mm.org/PageReplacementDesign

default Linux page replacement summary

identify *inactive* pages — guess: not going to be accessed soon

- file pages which haven't been accessed more than once, or
- any pages which haven't been accessed recently

some minimum threshold of inactive pages

- add to inactive list in background
- detecting references — scan referenced bits
- (I thought Linux marked as invalid — but wrong: not on x86)
- detect enough references — move to active

oldest inactive page still not used $\rightarrow$ evict that one

- otherwise: give it a second chance

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 heuristics (1)

exercise: devise an algorithm to detect to do readahead.

- when to start reads?
- how much to readahead?

want to detect contiguous accesses to mmap'd pages

can mark pages invalid temporarily to detect references to present pages

can add if statement to detect when new pages are brought in

Linux readahead heuristics — how much

how much to readahead?

Linux heuristic: count number of cached pages before

guess we should read about that many more

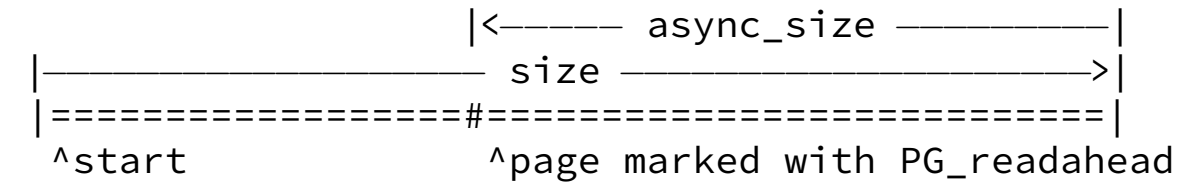
minimum/maximum to avoid extremes

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:



when `async_size` pages left, read next chunk

marked page = detect reads to this page

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

thrashing

what if there's just not enough space?

- for program data, files currently being accessed

always reading things from disk

causes performance collapse — disk is really slow

known as thrashing

‘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 16MB file X

process B

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

process C

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

process D+E

- 4MB of stack+code, 64MB of heap
- but all heap is shared copy-on-write

accounting pages

shared pages make it difficult to count memory usage

Linux *cgroups* accounting: last touch

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

Linux cgroup limits

Linux “control groups” of processes

can set memory limits for group of proceses:

low limit: don’t ‘steal’ pages when group uses less than this

- always take pages someone is using (unless no choice)

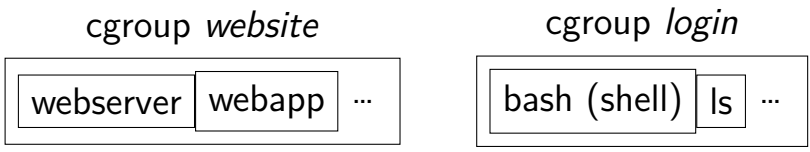
high limit: never let group use more than this

- replace pages from this group before anything else

...

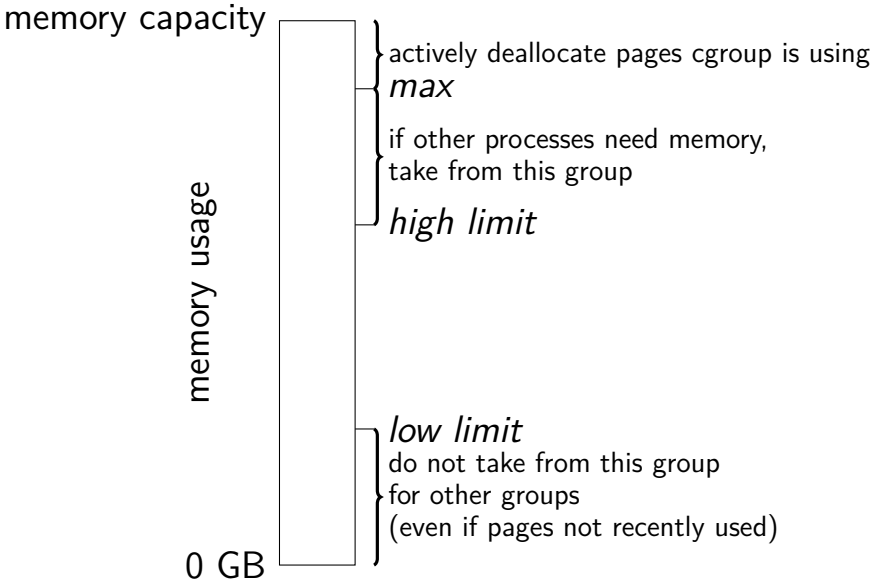
Linux cgroups

Linux mechanism: seperate processes into groups:

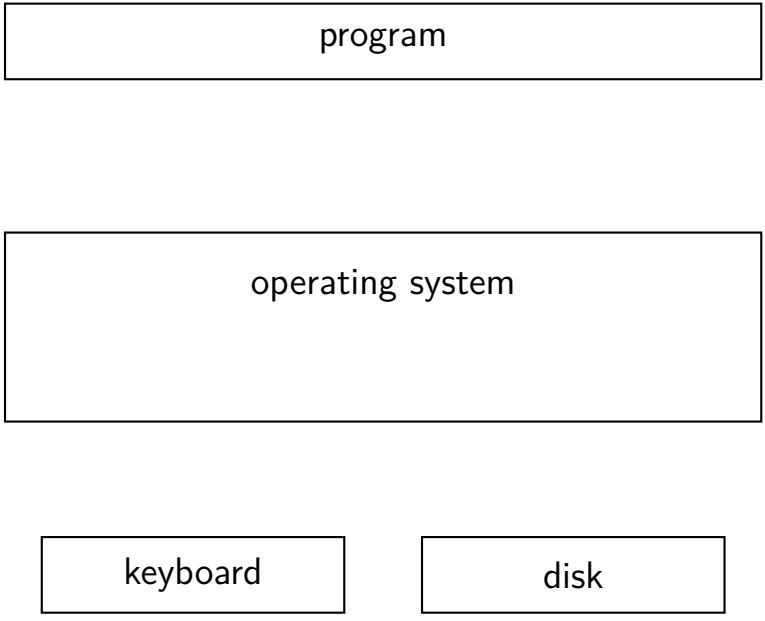


can set memory and CPU and ...shares for each group

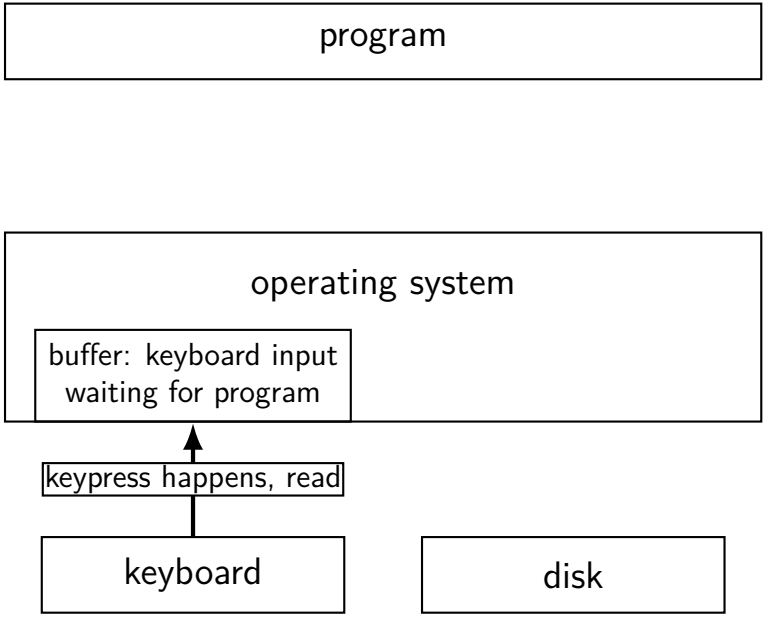
Linux cgroup memory limits



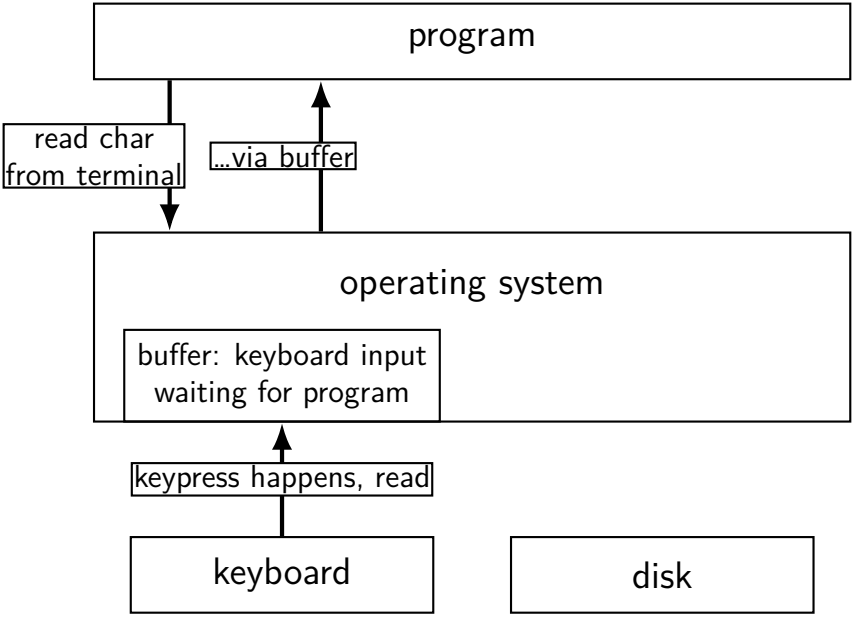
recall: kernel buffering (reads)



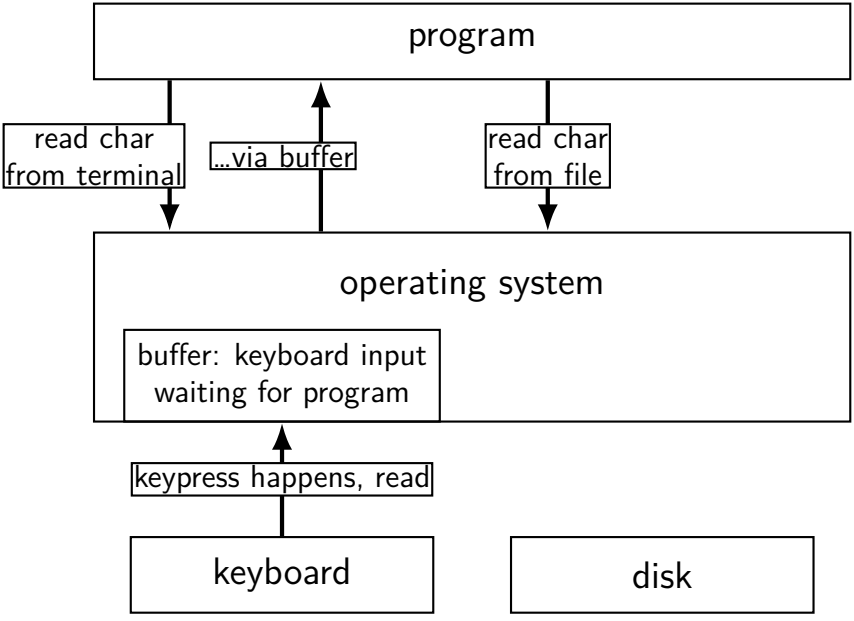
recall: kernel buffering (reads)



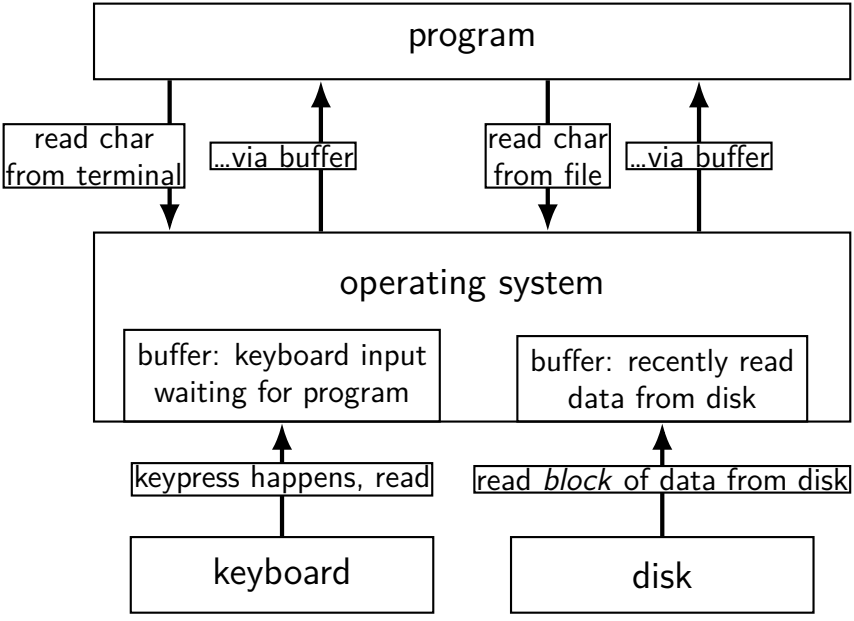
recall: kernel buffering (reads)



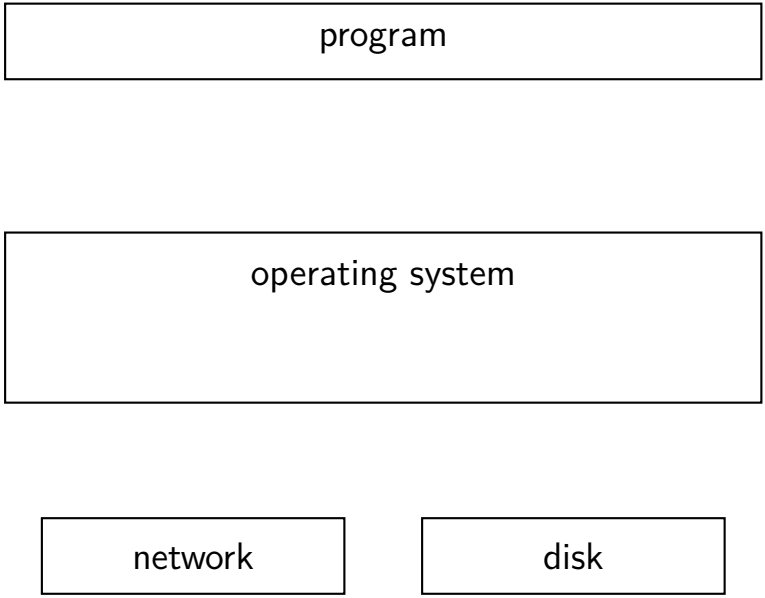
recall: kernel buffering (reads)



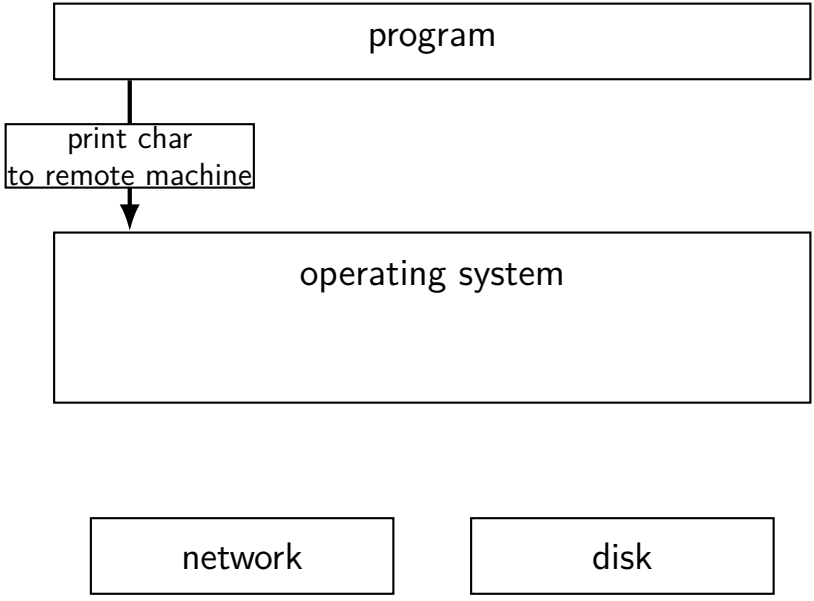
recall: kernel buffering (reads)



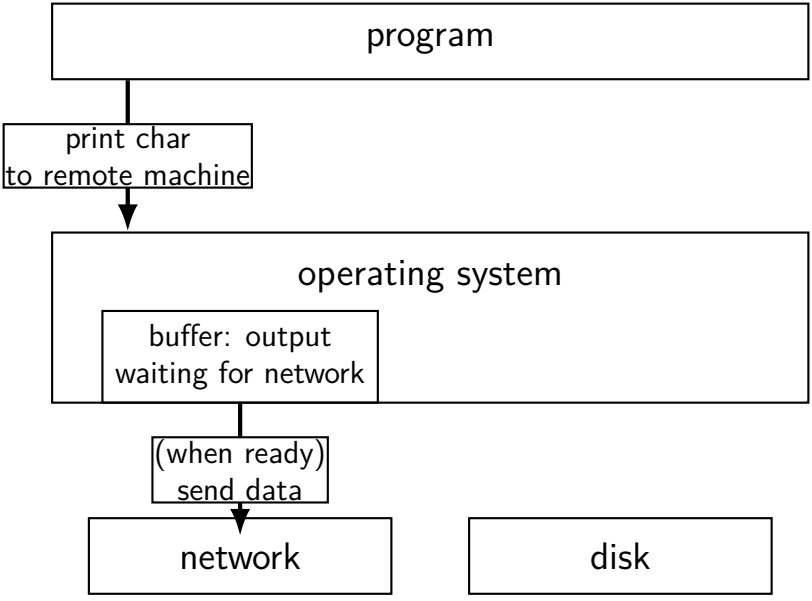
recall: kernel buffering (writes)



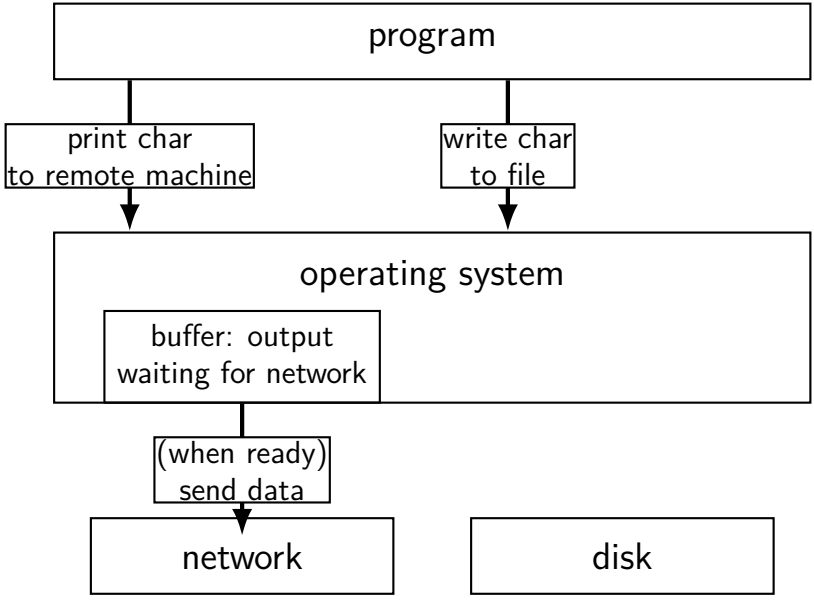
recall: kernel buffering (writes)



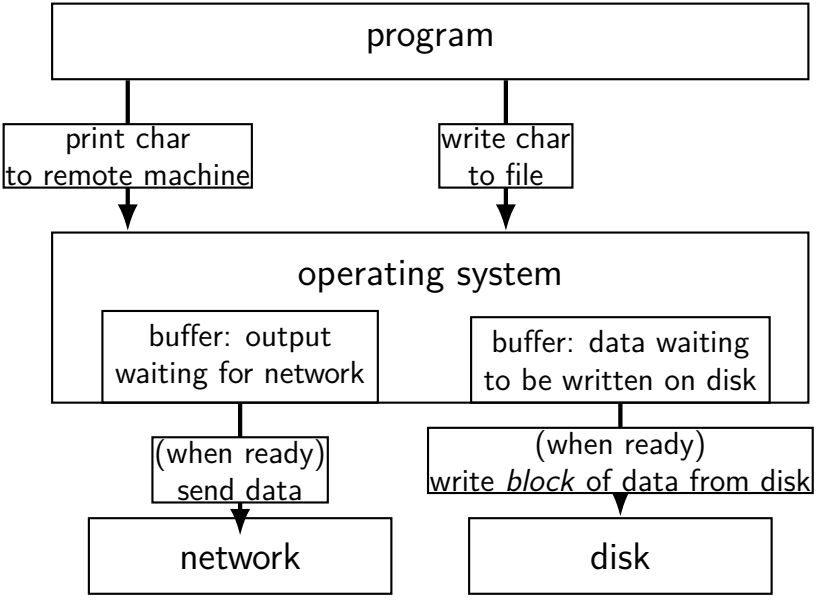
recall: kernel buffering (writes)



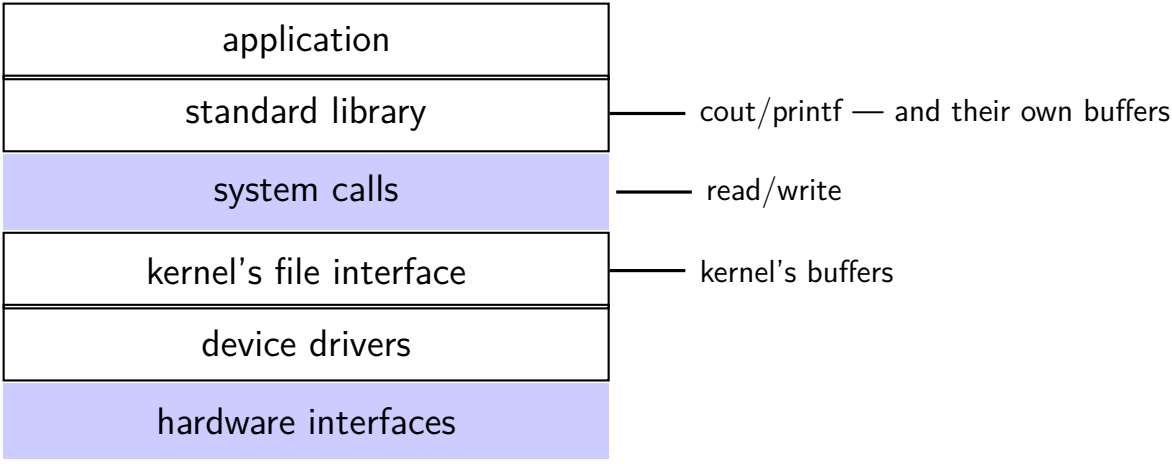
recall: kernel buffering (writes)



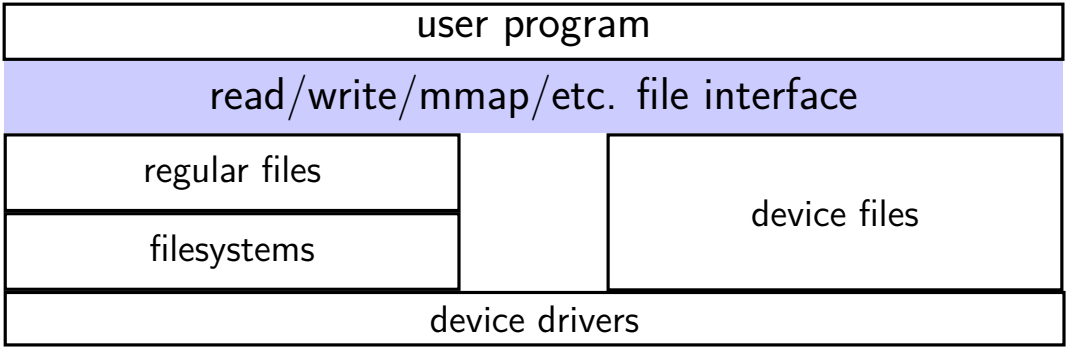
recall: kernel buffering (writes)



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?
- terminal — whether to echo back what user types?
- CD/DVD — open the disk tray? is a disk present?
- ...

POSIX: ioctl (general I/O control), tcget/setaddr (for terminal settings), ...

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'

- instead of bytes

used by *filesystems* — store directories on devices

- filesystems are specialized to know disks aren't byte-based

want to work with page cache — bytes not convenient

- read/write page at a time
- implement read/write to use page cache, not direct

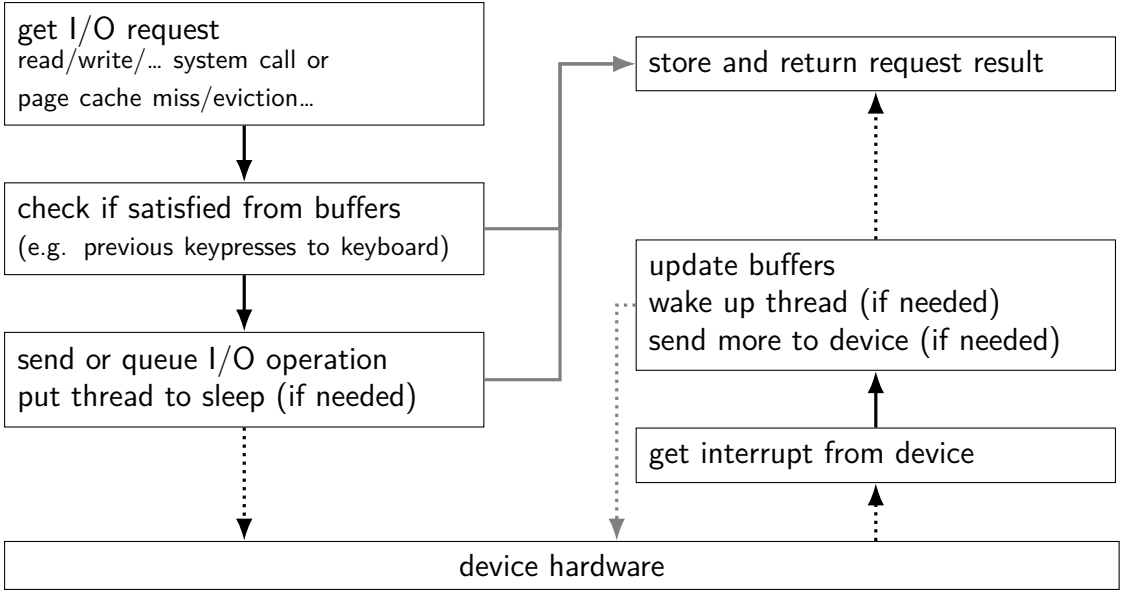
common code to translate from working with bytes to blocks

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
    ...
};
```

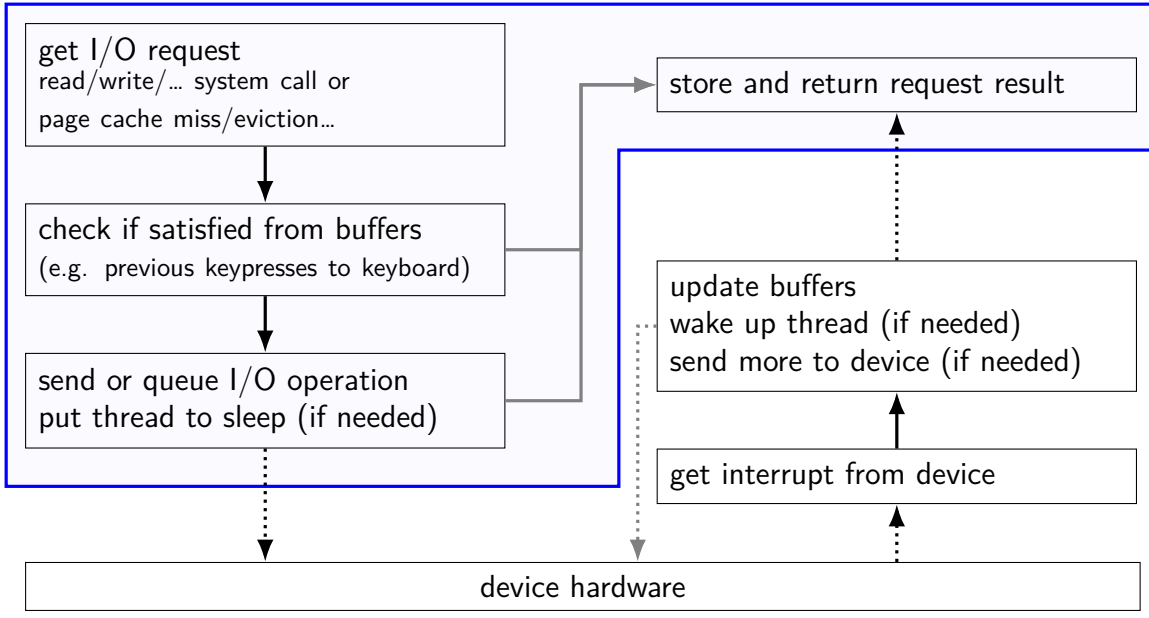
read/write a page for a sector number (= block number)

device driver flow



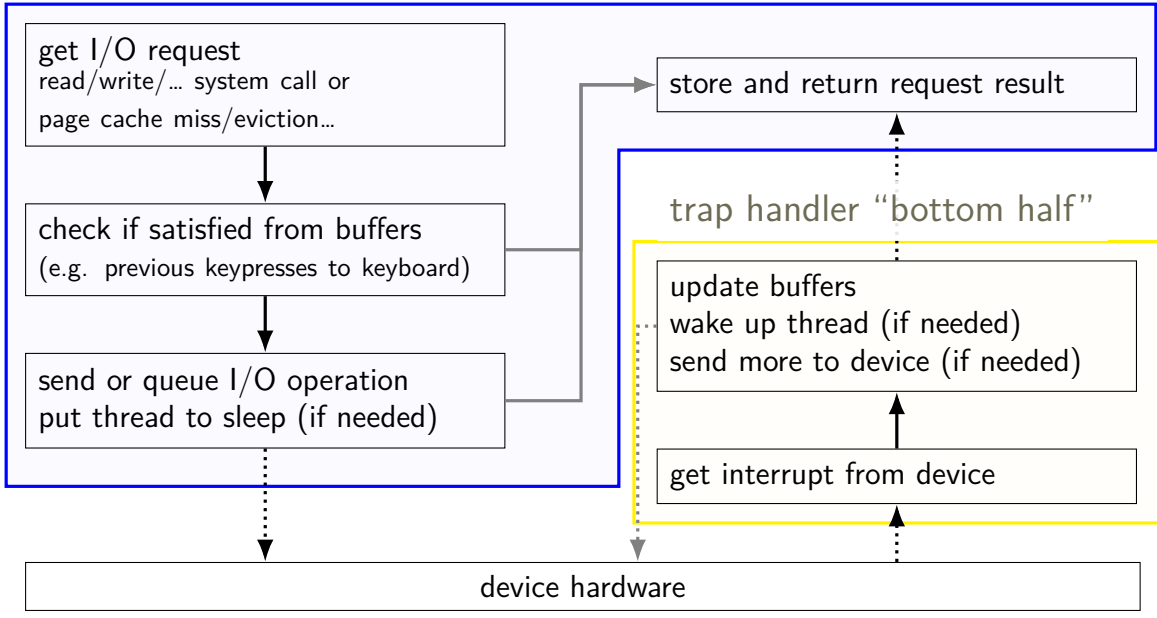
device driver flow

thread making read/write/etc. "top half"



device driver flow

thread making read/write/etc. "top half"



xv6: device files

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

extern struct devsw devsw[];
```

table of devices

device file uses entry in `devsw` array

- filesystem stores name to index lookup

similar scheme used on 'real' Unix/Linux

- files referencing major/minor device number
- table of device numbers in kernel

xv6: console devsw

code run at boot:

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

CONSOLE is a constant

xv6: console devsw

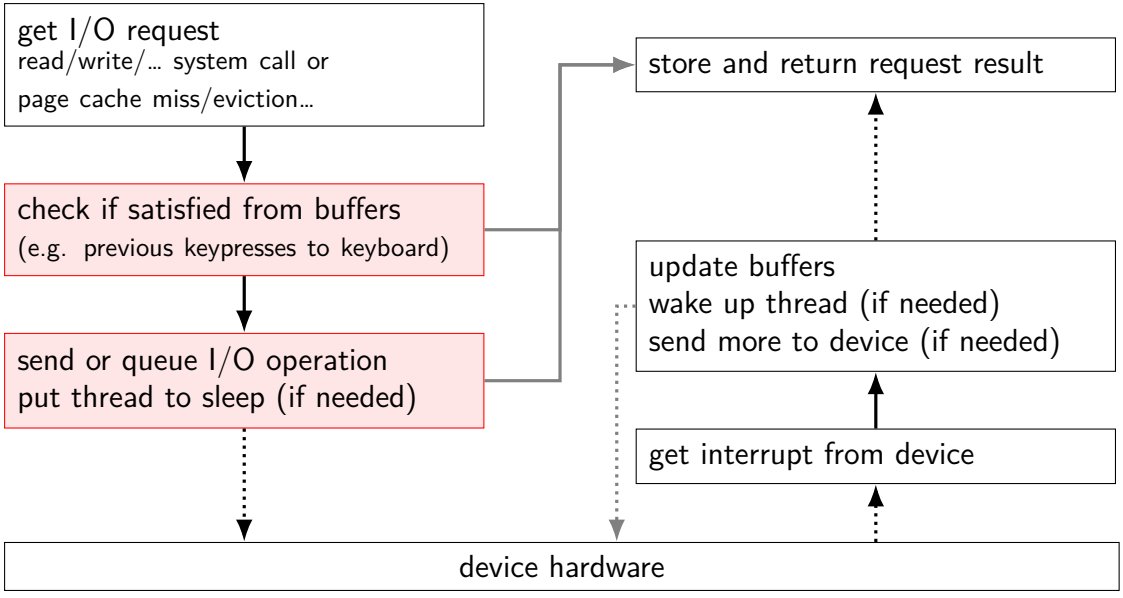
code run at boot:

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

CONSOLE is a constant

consoleread/consolewrite: run when you read/write console

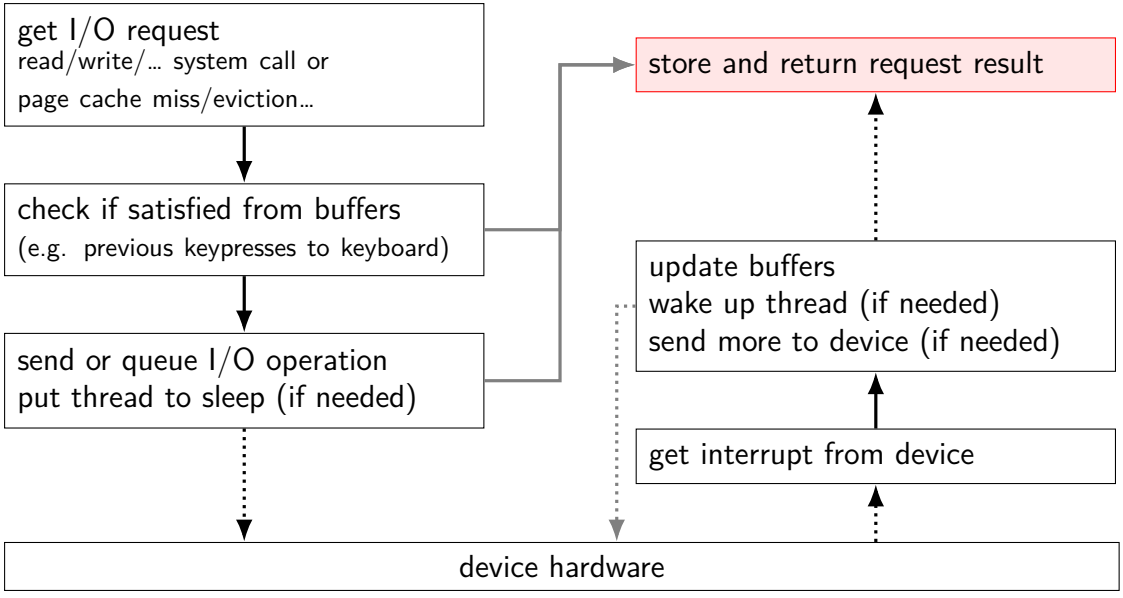
device driver flow



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)
  ...
}
```

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_BUF];
    ...
    *dst++ = c;
    --n;
    if (c == '\n')
      break;
  }
  release(&cons.lock)
  ...
  return target - n;
}
```

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_BUF];
    ...
    *dst++ = c;
    --n;
    if (c == '\n')
      break;
  }
  release(&cons.lock)
  ...
  return target - n;
}
```

xv6: console top half

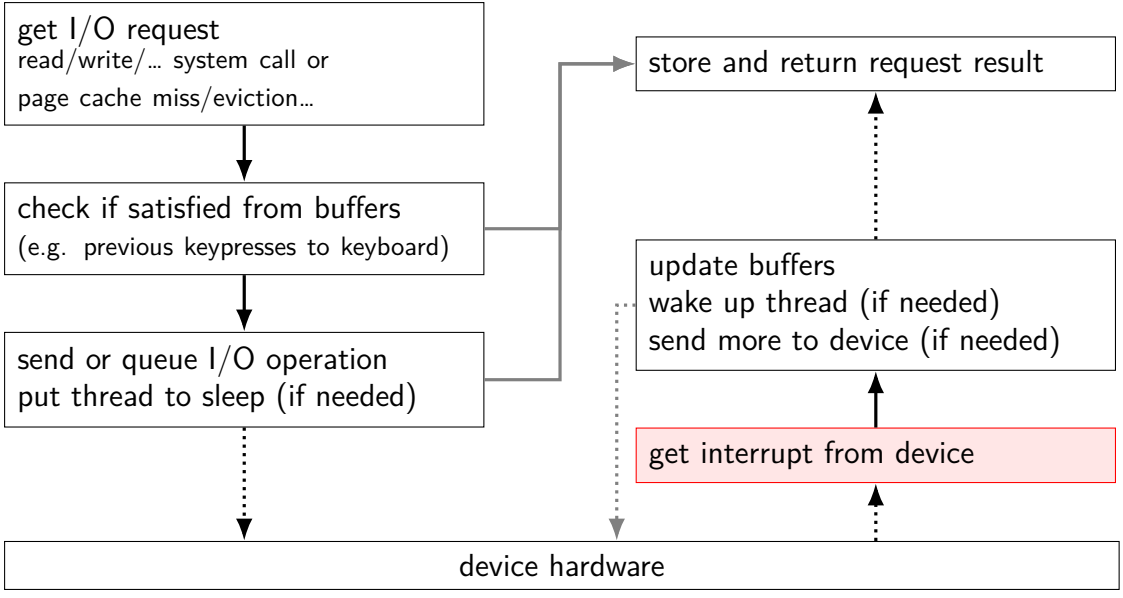
wait for buffer to fill

- 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: atually read from keyboard device

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

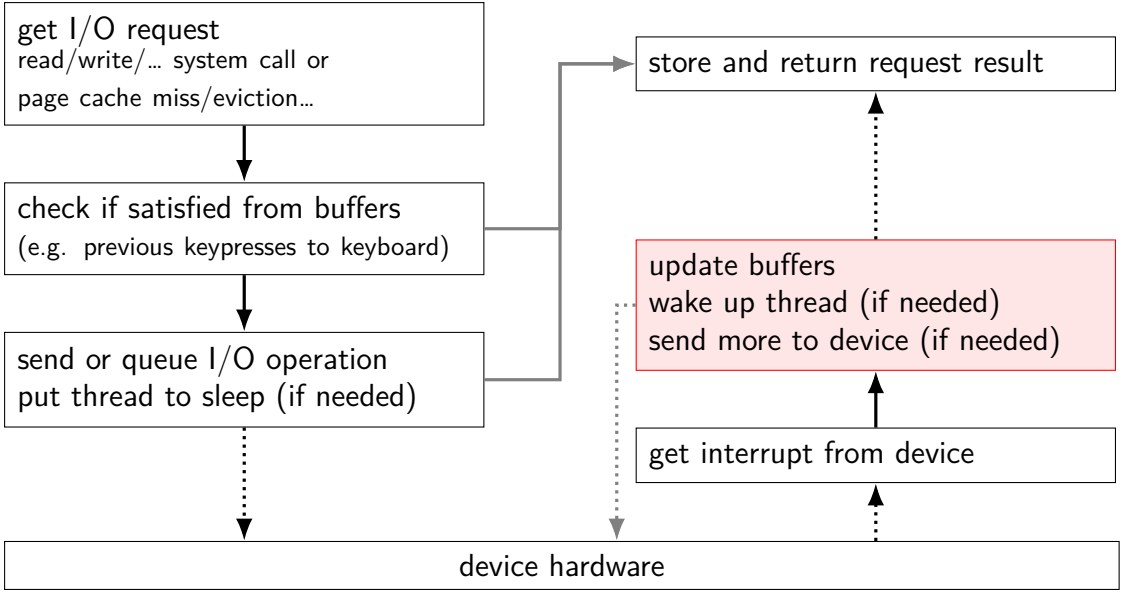
xv6: console interrupt (one case)

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

kbdintr: atually read from keyboard device

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

device driver flow



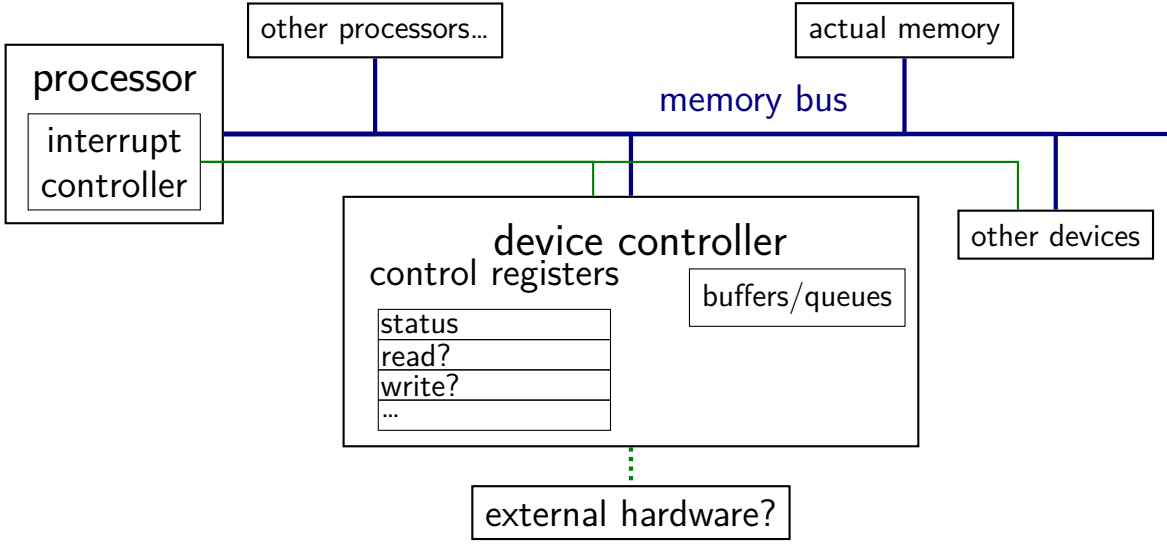
xv6: console interrupt reading

kbdintr fuction actually reads from device

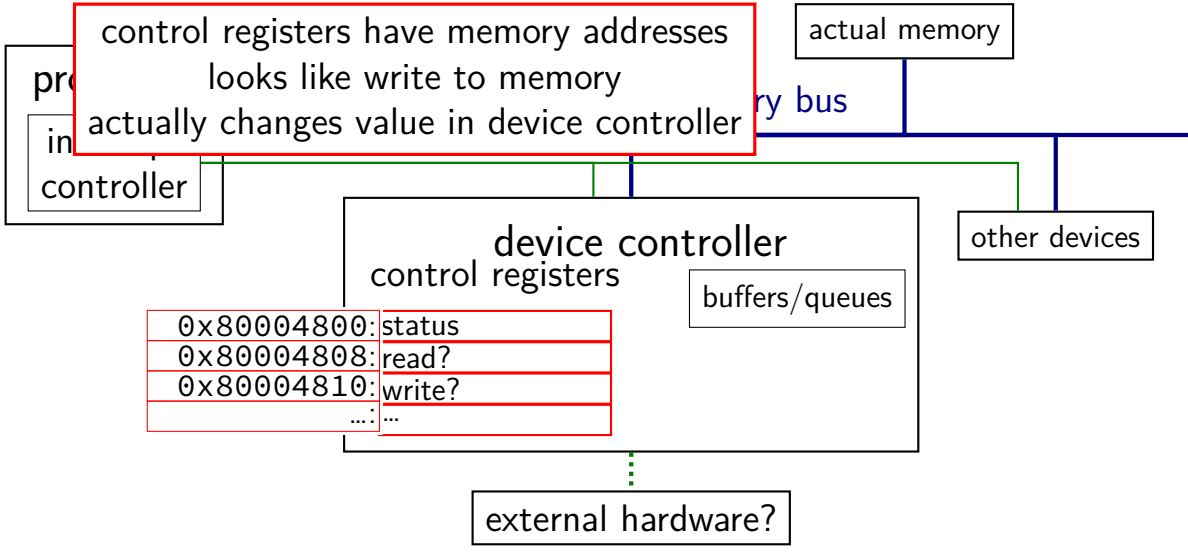
adds data to buffer (if room)

wakes up sleeping thread (if any)

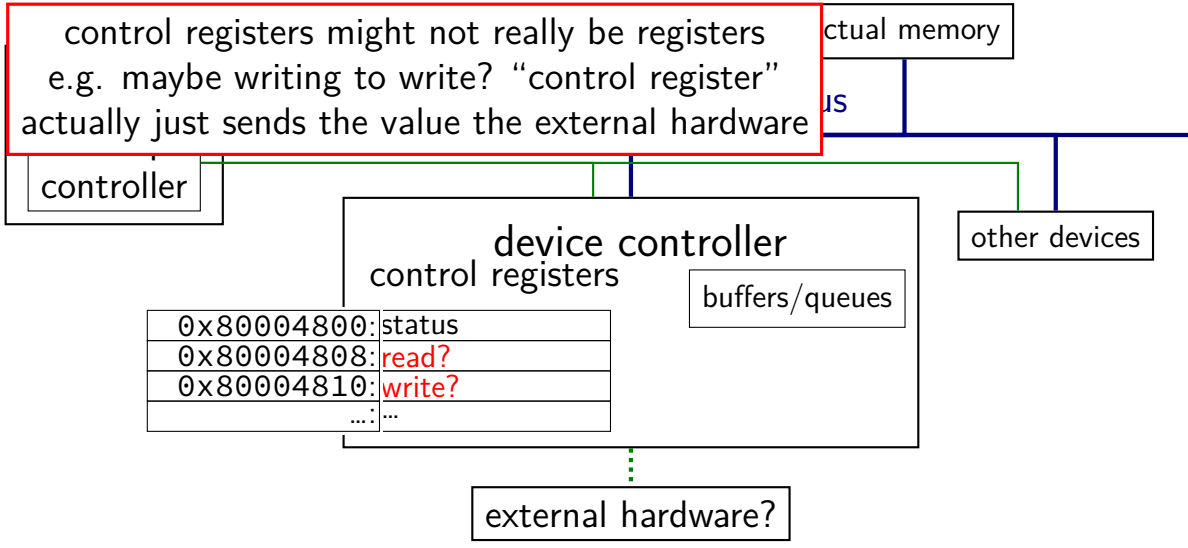
connecting devices



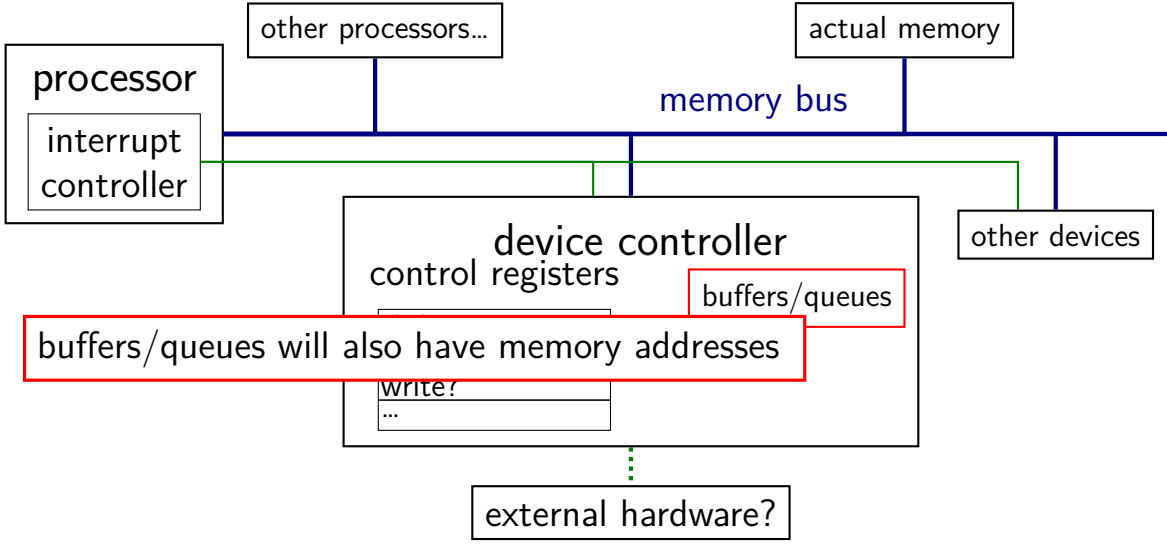
connecting devices



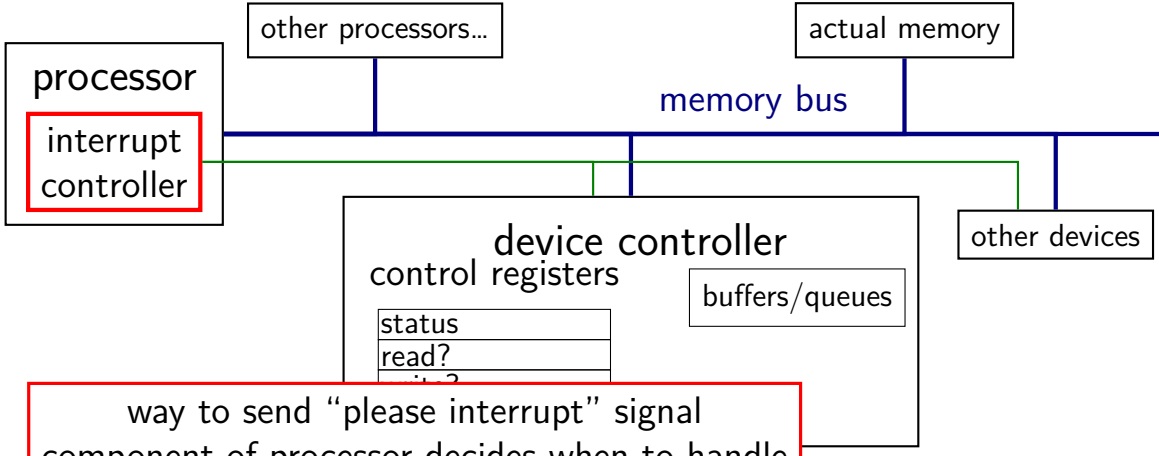
connecting devices



connecting devices

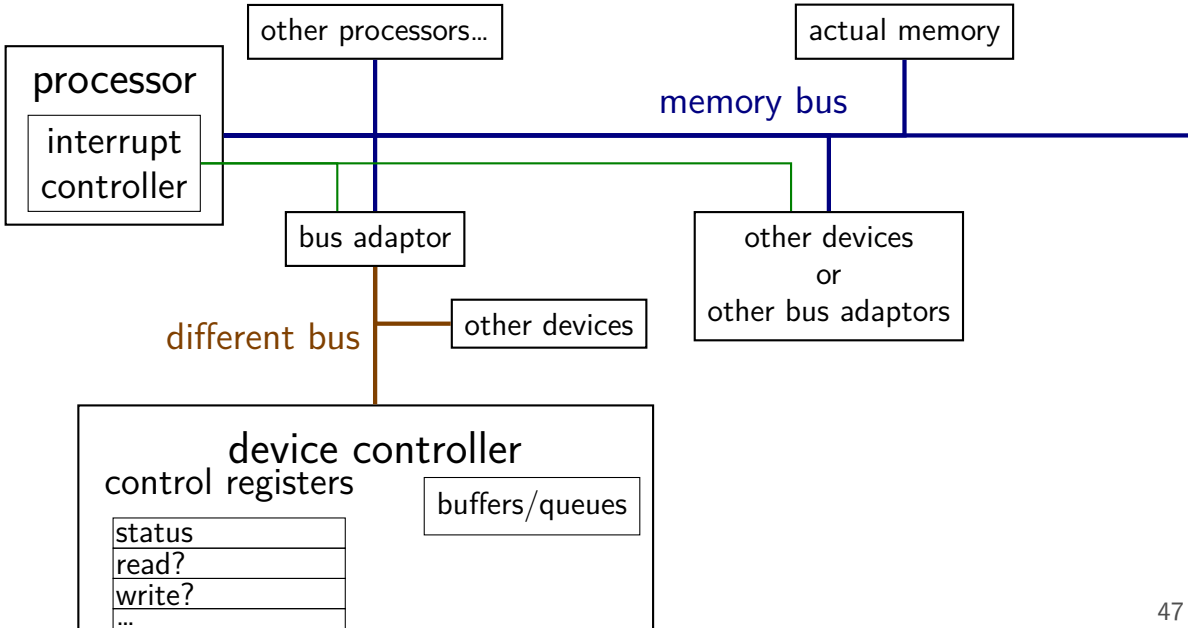


connecting devices



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: separate I/O bus

more recent processors/devices would just use memory addresses

- no need for more instructions, buses
- 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

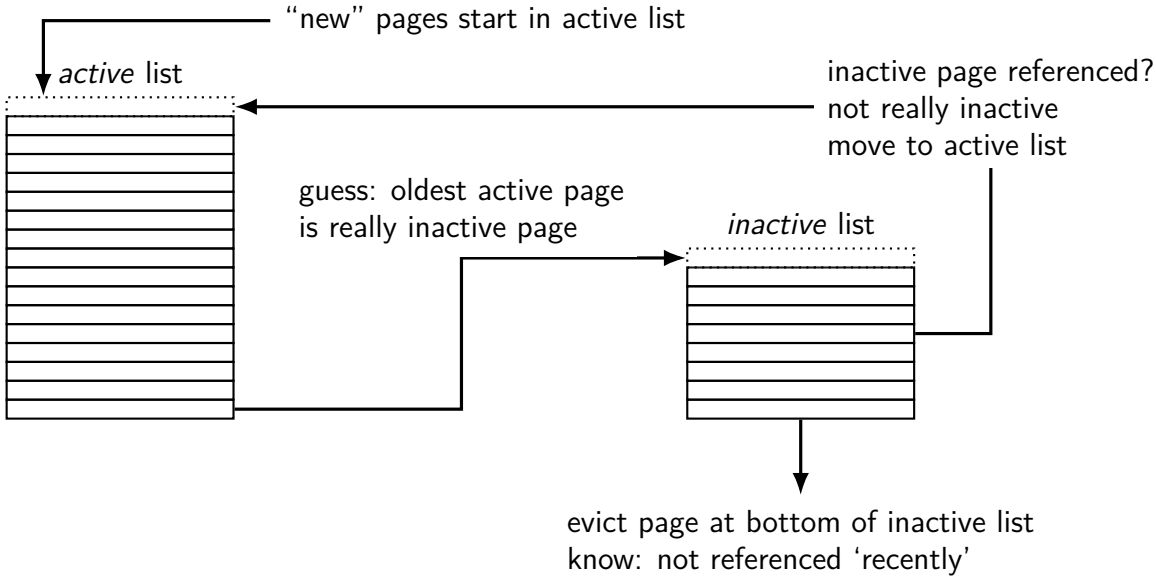
"programmed I/O": write to or read from device buffers directly

OS runs loop to transfer data to or from device

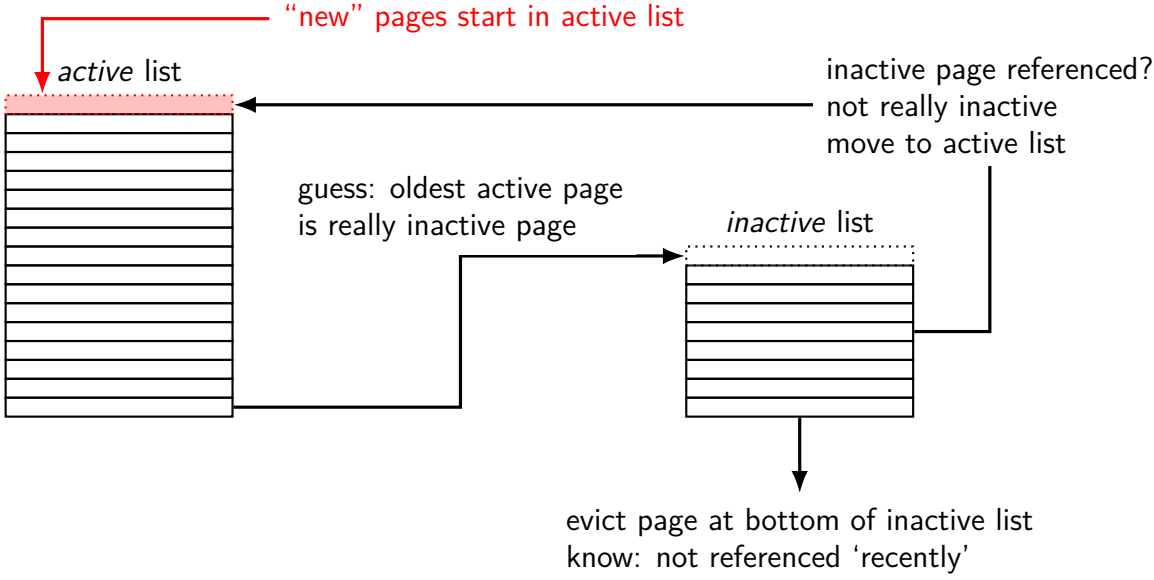
might still be triggered by interrupt

- know/what for "is device ready"

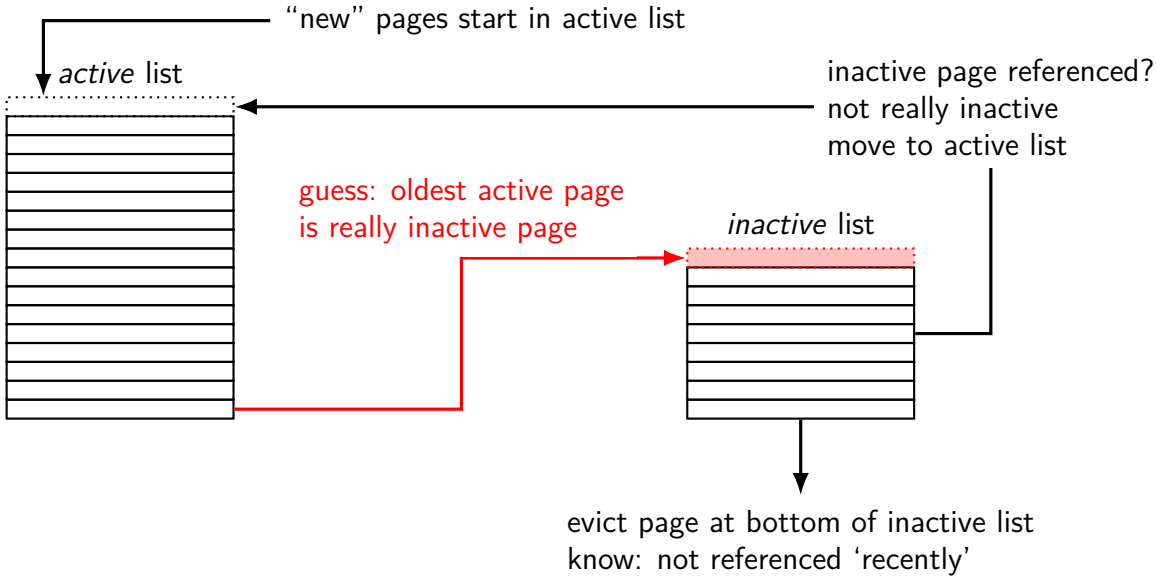
approximating LRU: SEQ



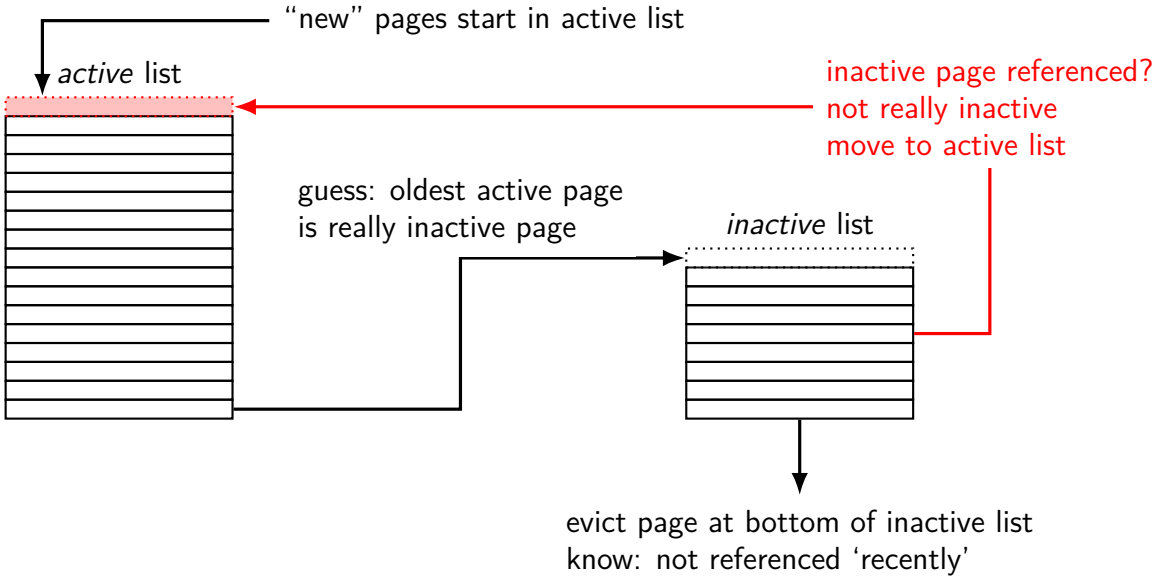
approximating LRU: SEQ



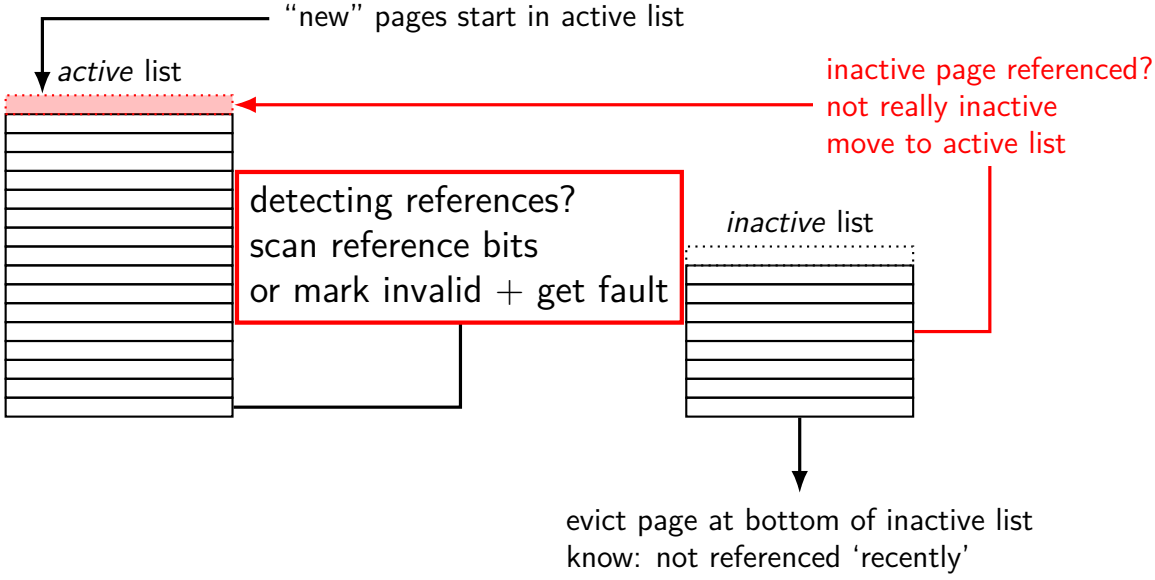
approximating LRU: SEQ



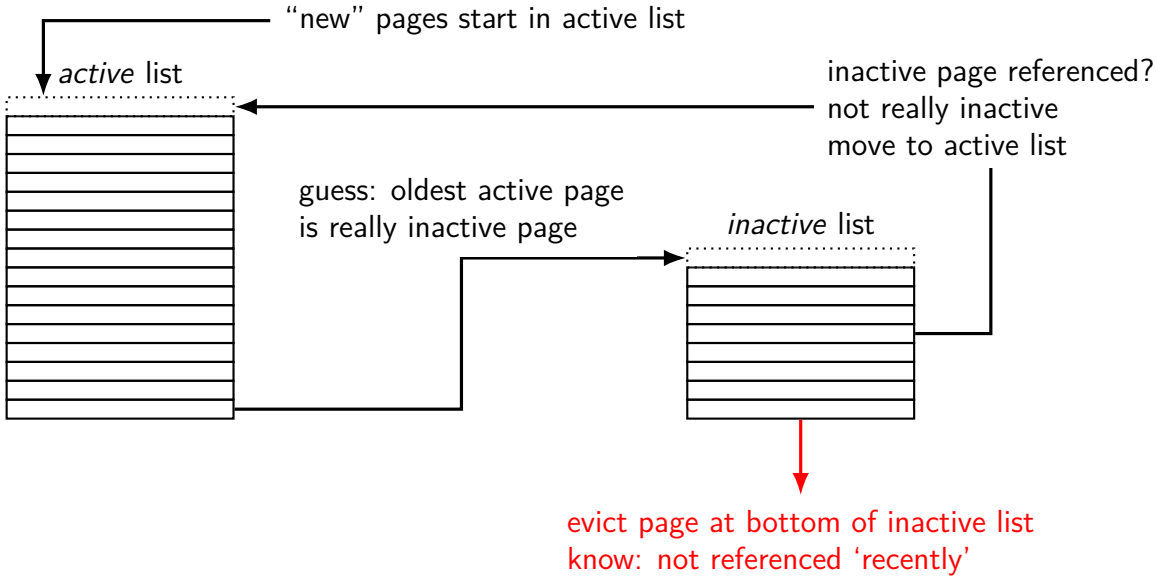
approximating LRU: SEQ



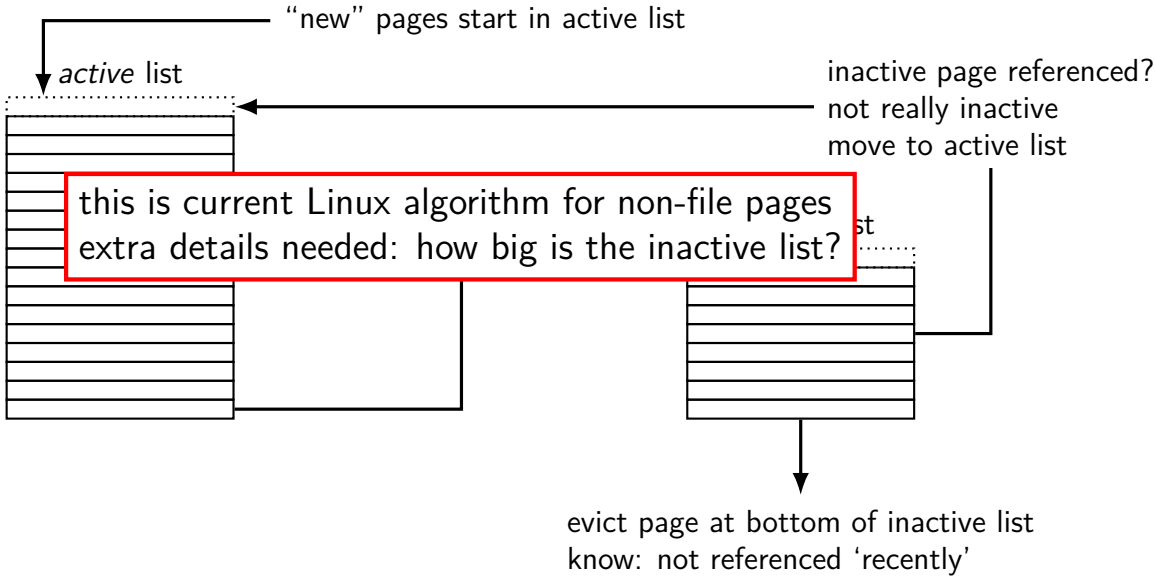
approximating LRU: SEQ



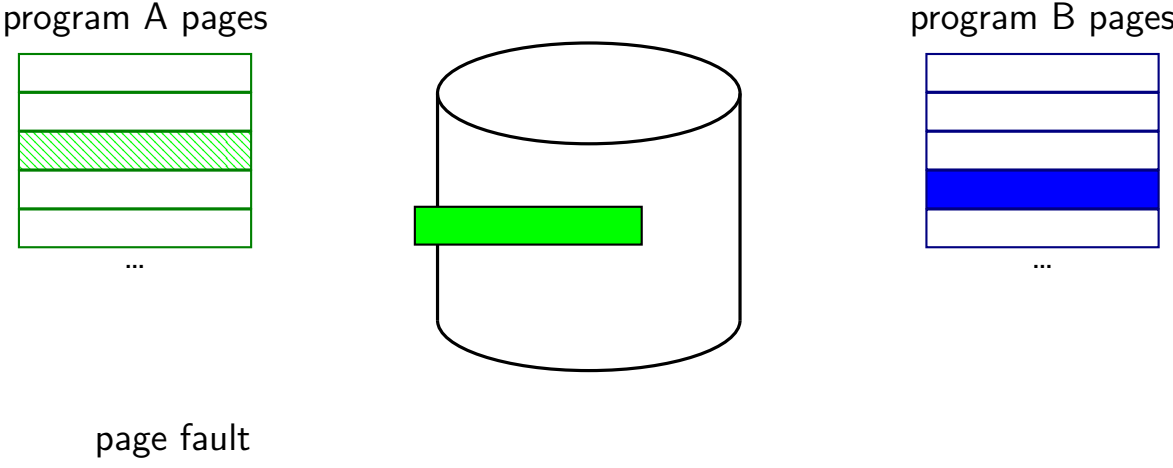
approximating LRU: SEQ



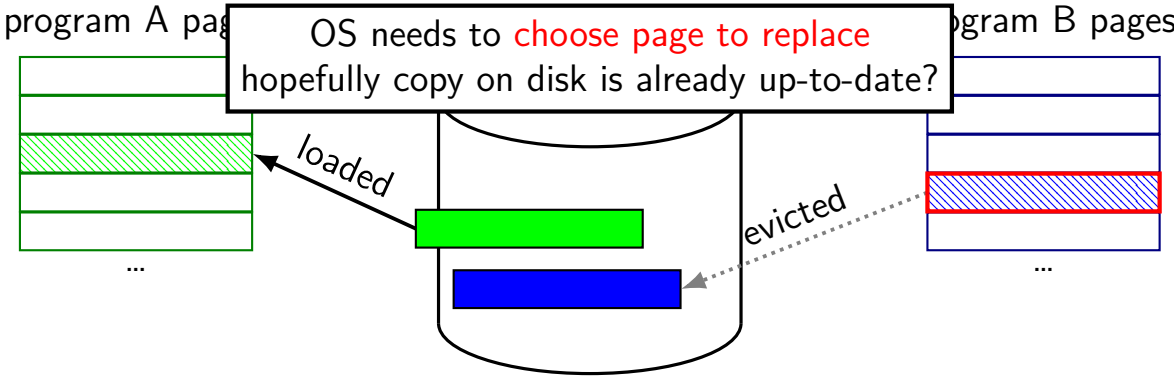
approximating LRU: SEQ

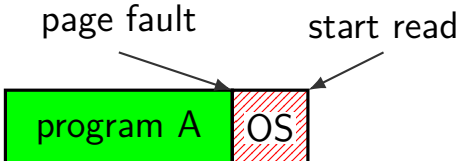


swapping timeline

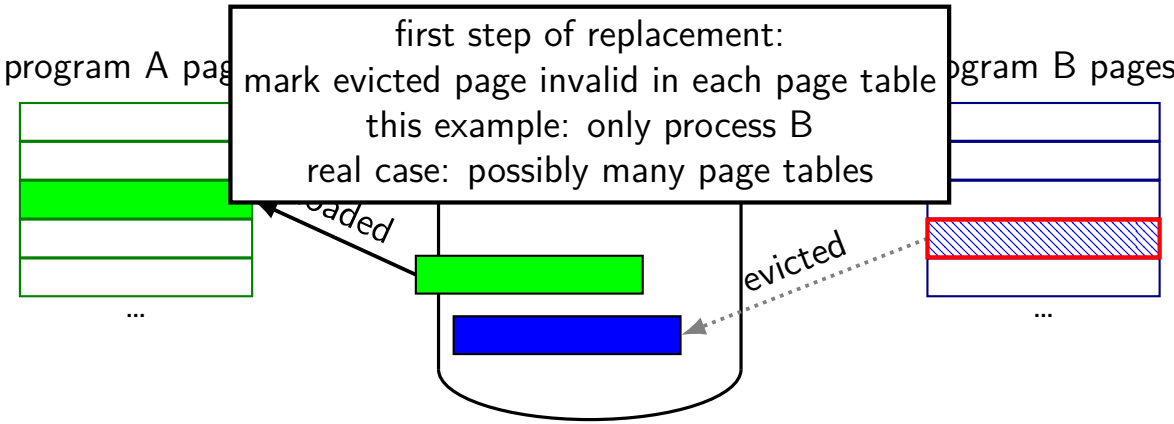


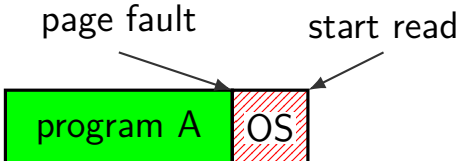
swapping timeline



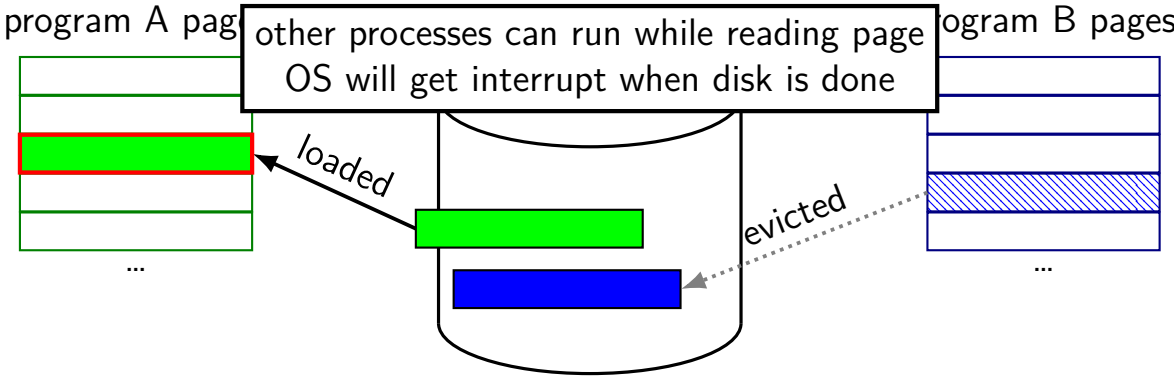


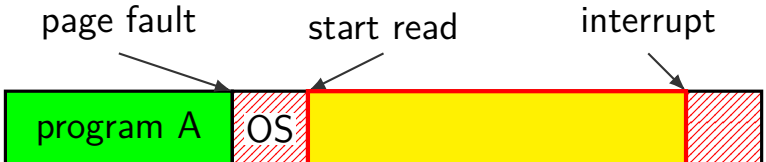
swapping timeline



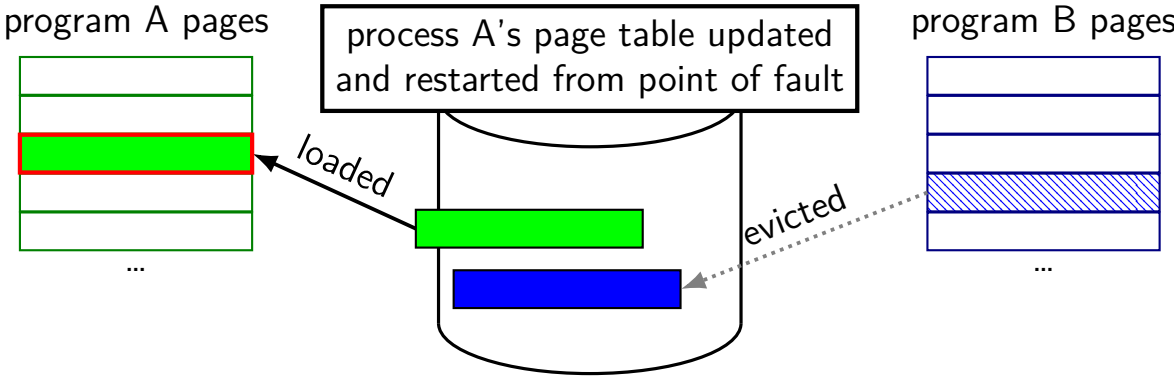


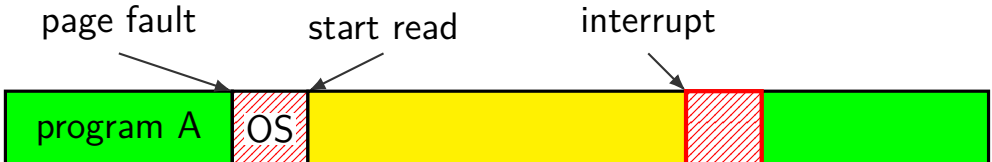
swapping timeline





swapping timeline





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

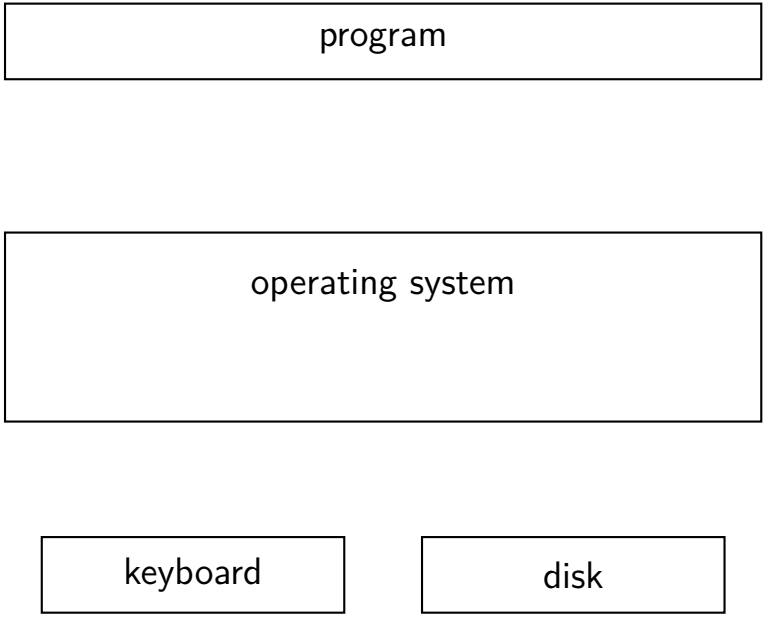
- setup, access control happens here

byte-oriented

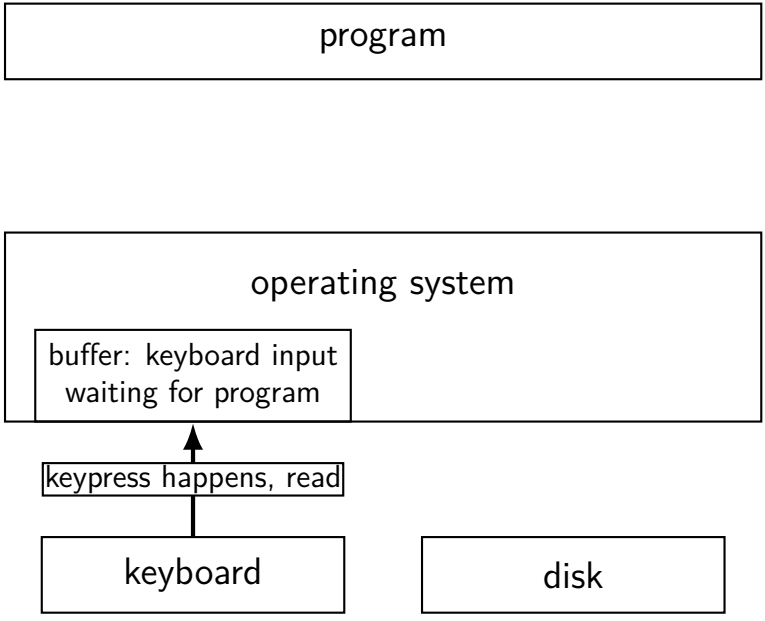
- real device isn't? operating system needs to hide that

explicit close

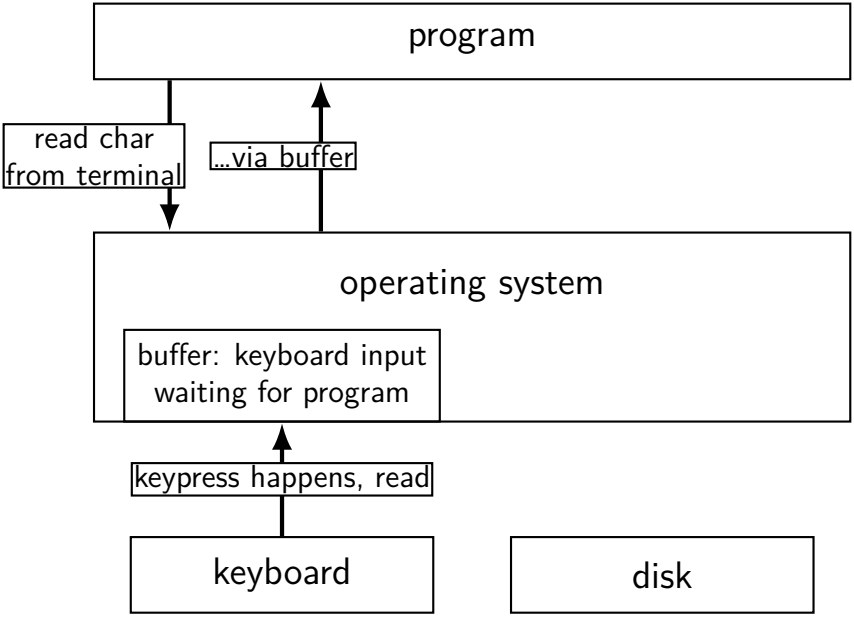
kernel buffering (reads)



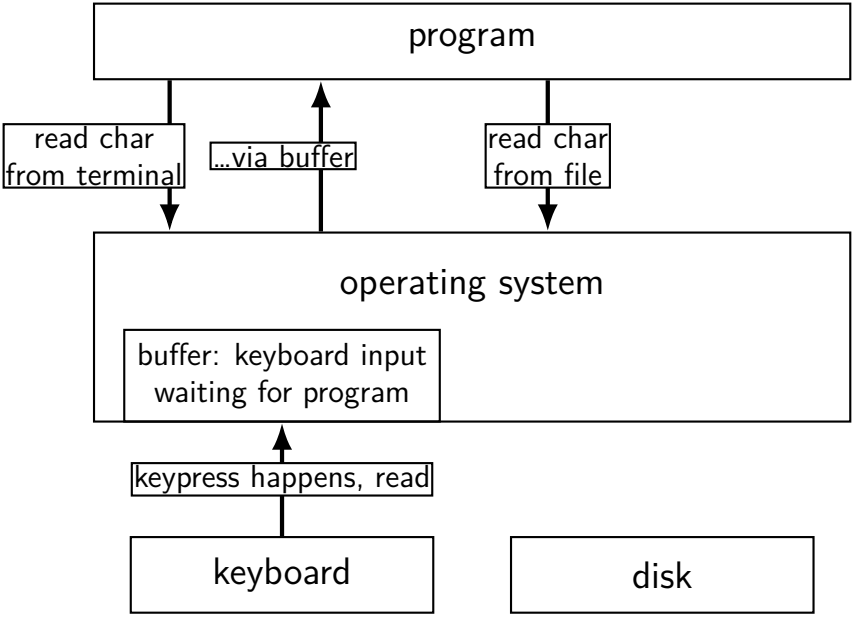
kernel buffering (reads)



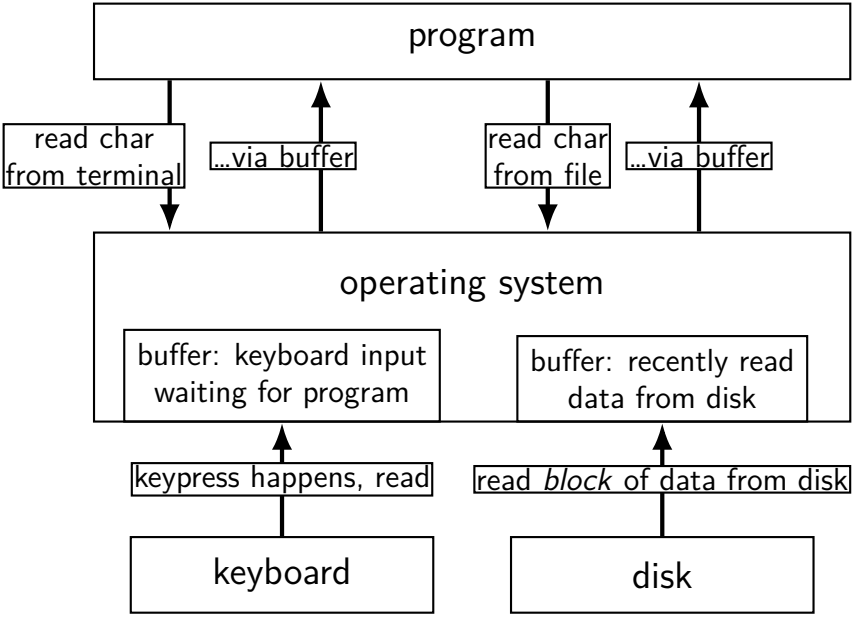
kernel buffering (reads)



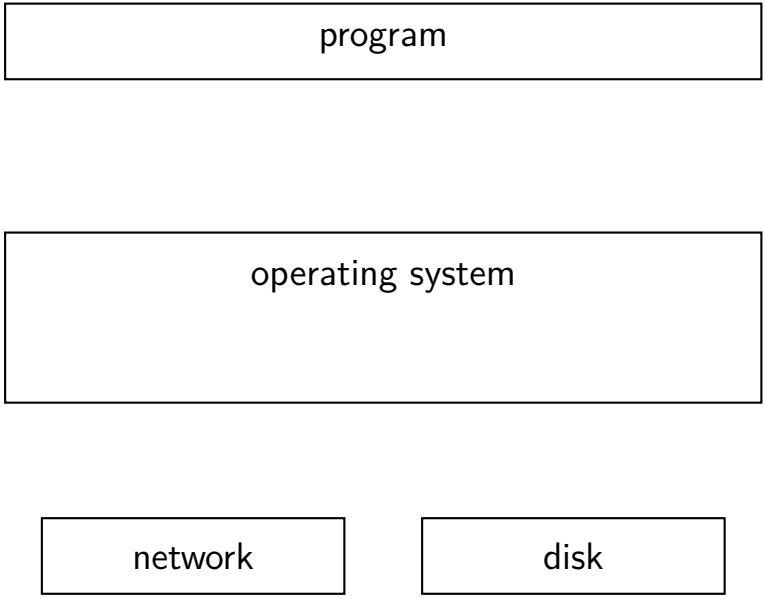
kernel buffering (reads)



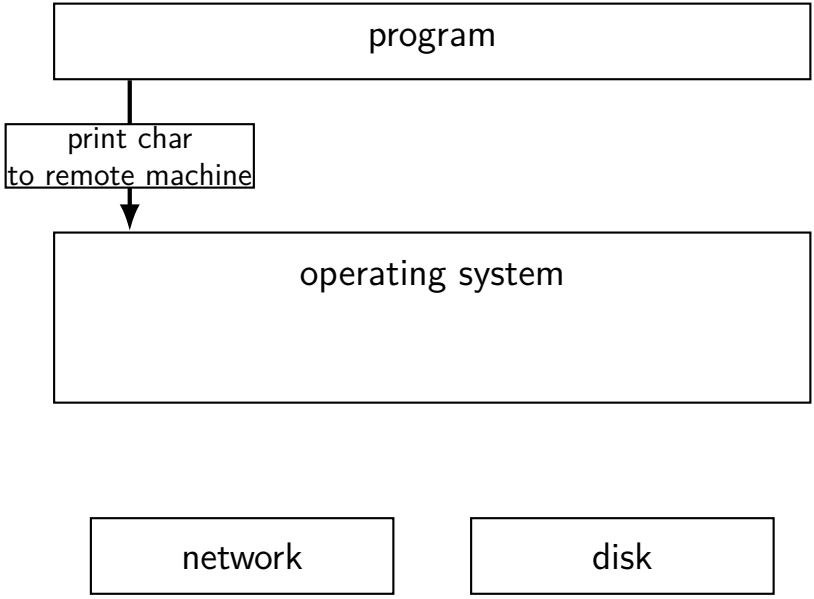
kernel buffering (reads)



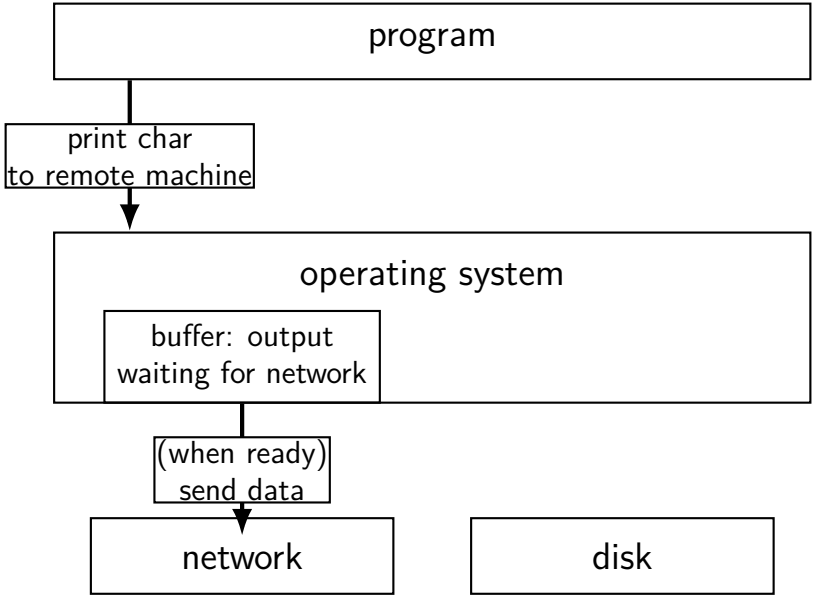
kernel buffering (writes)



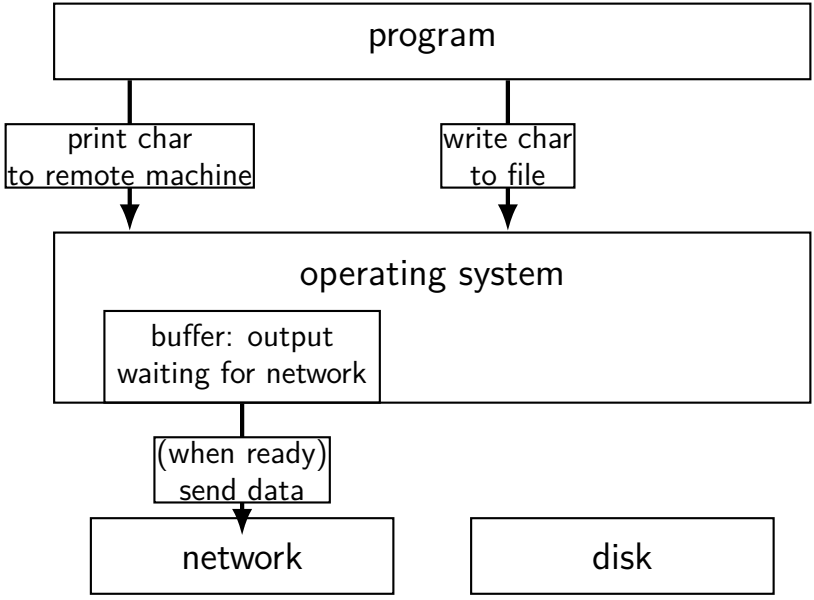
kernel buffering (writes)



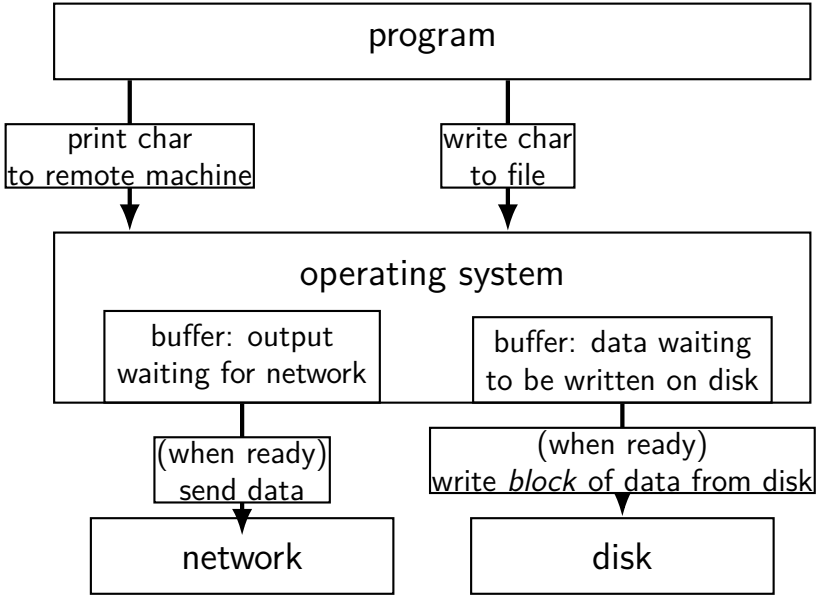
kernel buffering (writes)



kernel buffering (writes)



kernel buffering (writes)



read/write operations

read/write: move data into/out of buffer

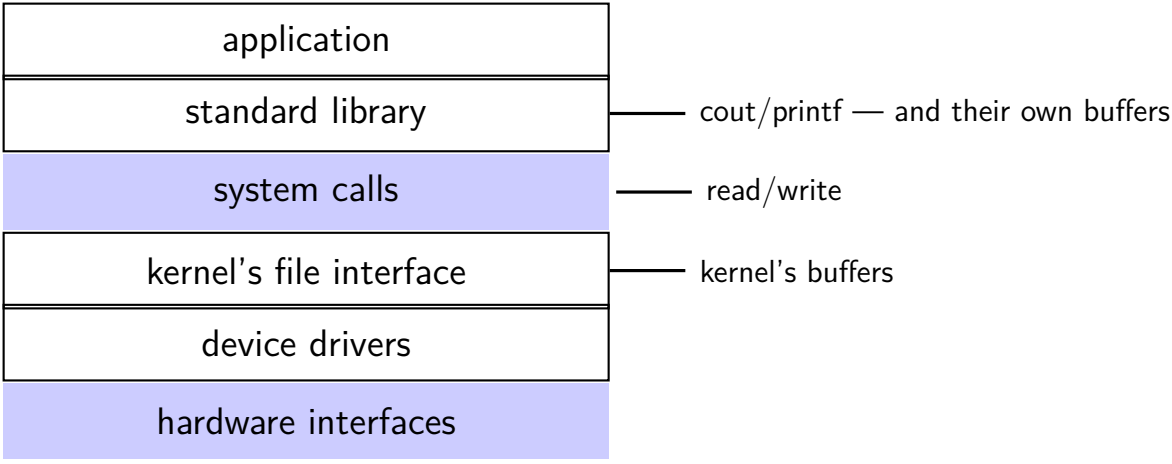
block (make process wait) if buffer is empty (read)/full (write)

(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