virtual memory 4 / I/O

paging homework note

pagingtest tests **both parts**

some copy-on-write tests run out of memory if no copy on write

some copy-on-write tests can hang if they run out of memory

- fork fails and the error handling code for this isn't great

copy-on-write tests failing or hanging after printing 'fork failed' is okay

kernel panics are **not** okay

last time

page cache data structures
- cache hit: page table + file→cached page lookup
- cache miss: location on disk (filesystem or record in invalid PTE)

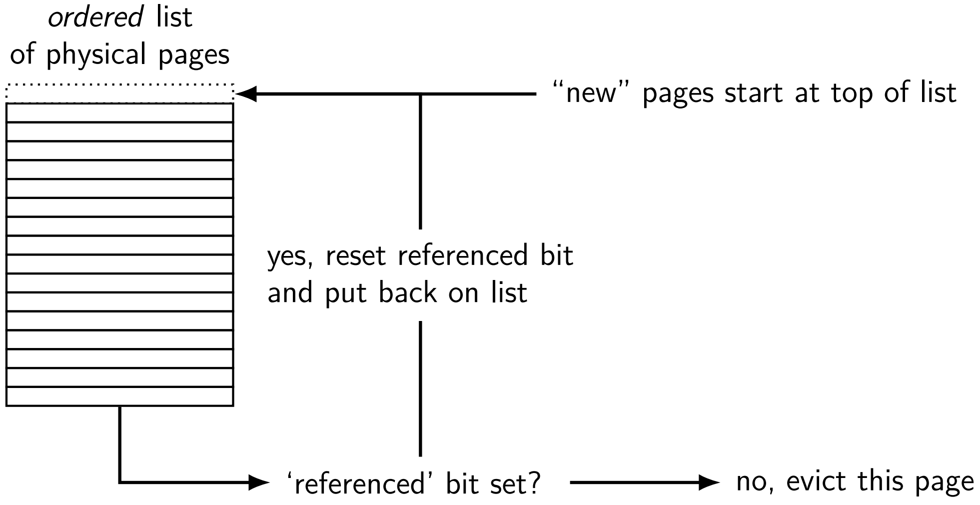
working set model
- set of ‘currently used’ pages
- changes throughout program execution
- easy approximation: most recently used pages

alternate model: Zipf/power law

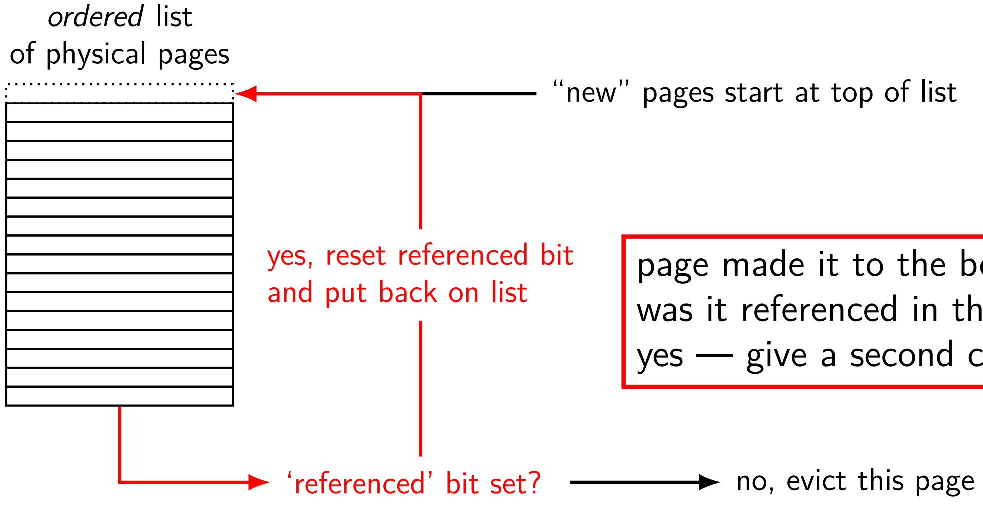
page replacement choices: hit rate v. throughput v. fairness

practical LRU:
- detect accesses via temporarily invalid PTE or accessed/referenced bit
- look for not-recently used stuff

approximating LRU: second chance

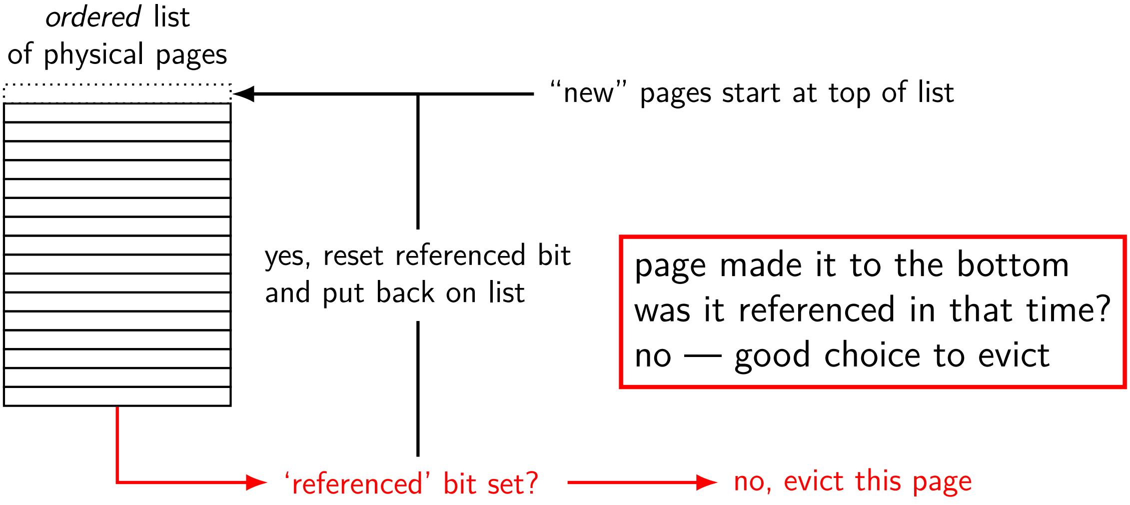


approximating LRU: second chance



page made it to the bottom
was it referenced in that time?
yes — give a second chance

approximating LRU: second chance



second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 2 was at bottom of list
is not referenced
okay to use

	A	B							A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

page 1 was at bottom of list
reference — give second chance
moves to top of list
clear referenced bit

	A	B							A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

eventually page 1 gets to bottom of list again
but now not referenced — use

	A									—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

second chance example

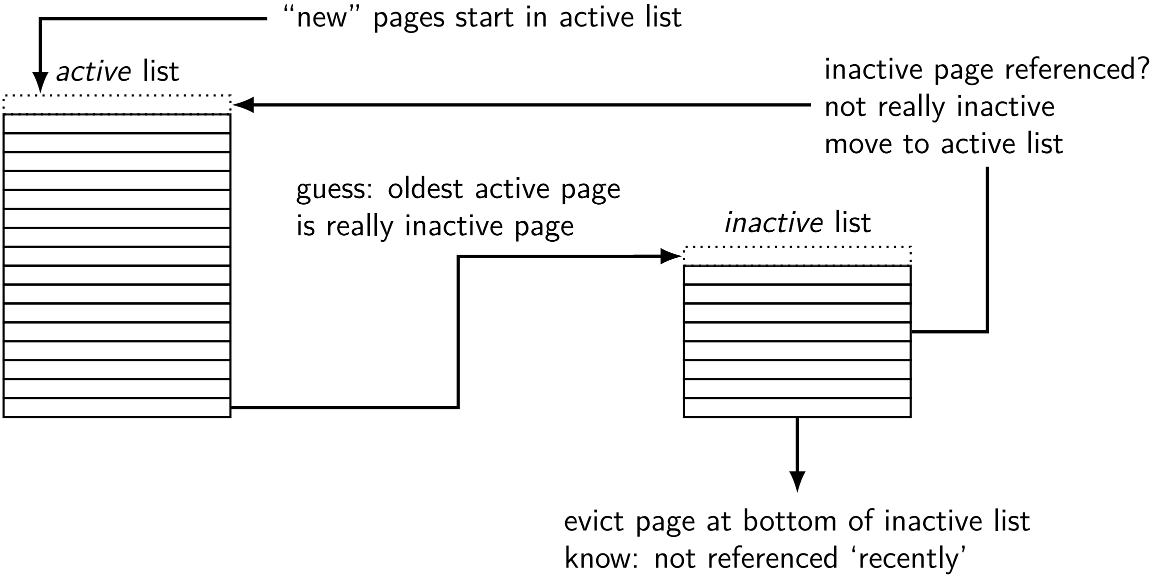
B referenced — flips referenced bit

	A									—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

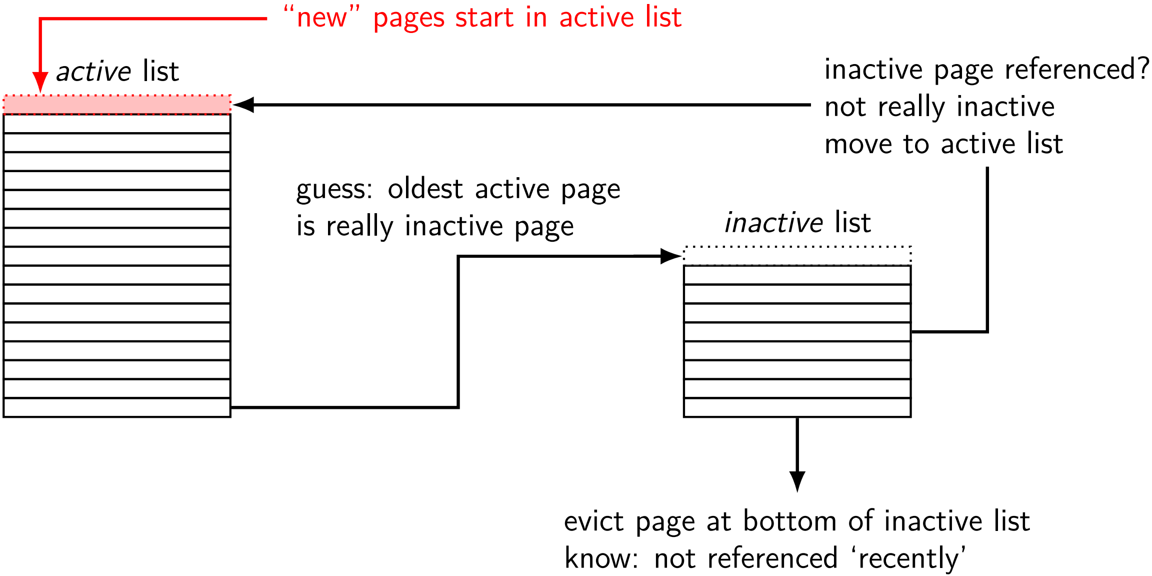
second chance example

	A	B	C	D	—	—	—	B	A	—	C	—
1	A						D					
2		B										C
3			C			C				A		
page list												
last added	*1R	*2R	*3R	1NR	2NR	3NR	*1R	1R	2NR	*3R	1NR	*2R
—	3NR	1R	2R	3R	1NR	2NR	3NR	3NR	1R	2NR	3R	1NR
end of list	2NR	3NR	1R	2R	3R	1NR	2NR	*2R	3NR	1R	2NR	3R

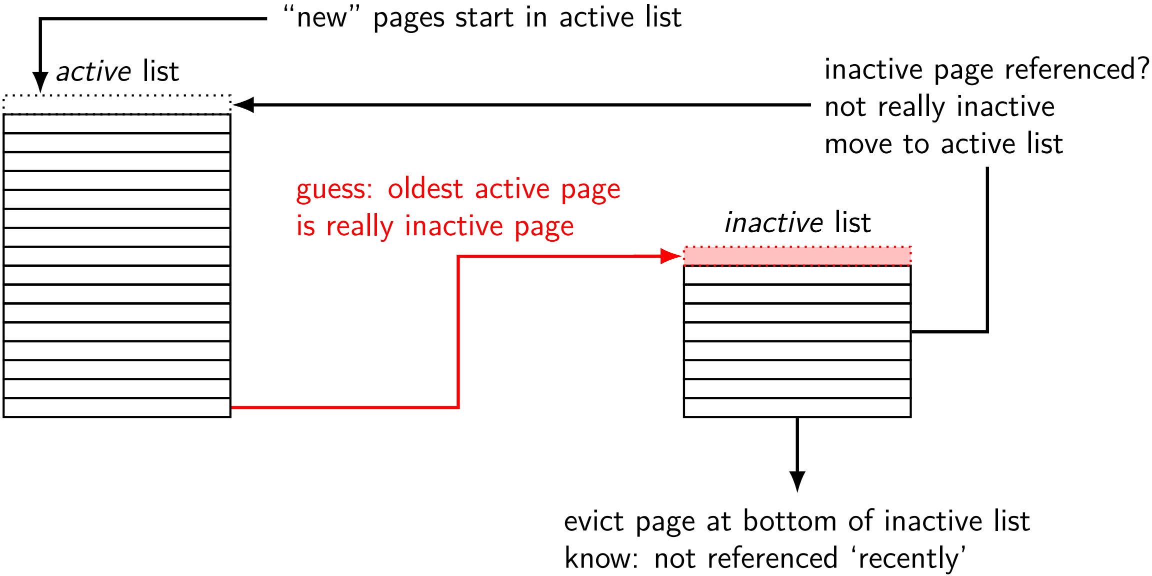
approximating LRU: SEQ



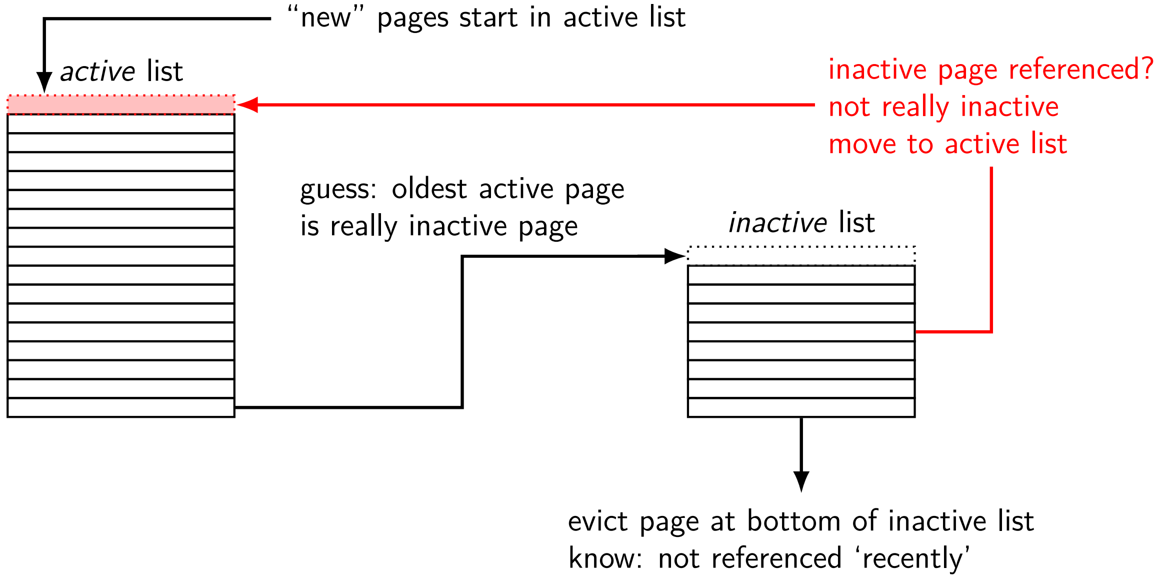
approximating LRU: SEQ



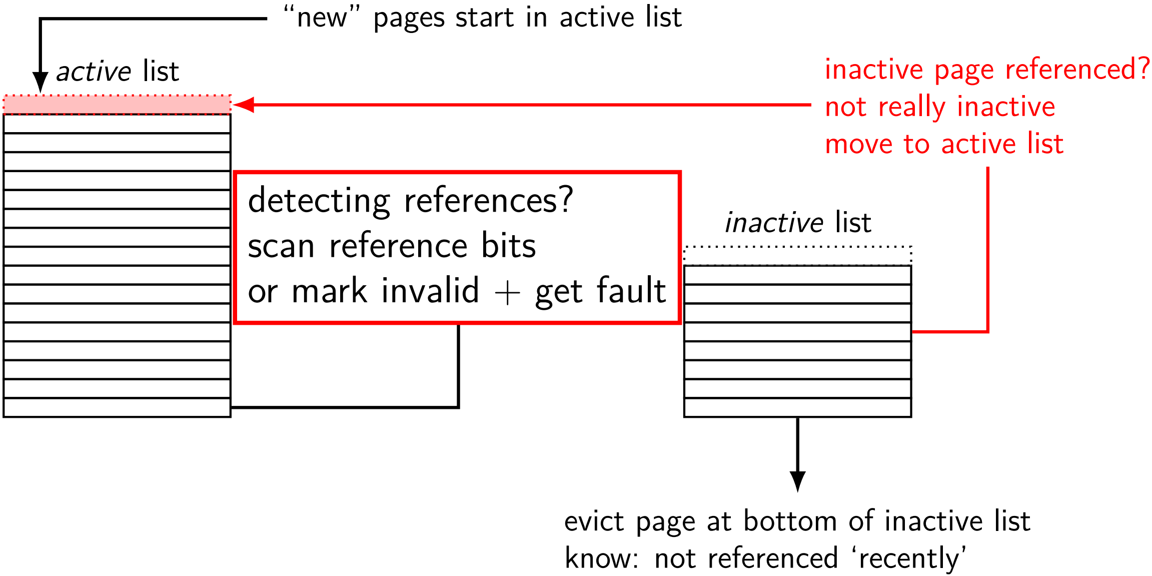
approximating LRU: SEQ



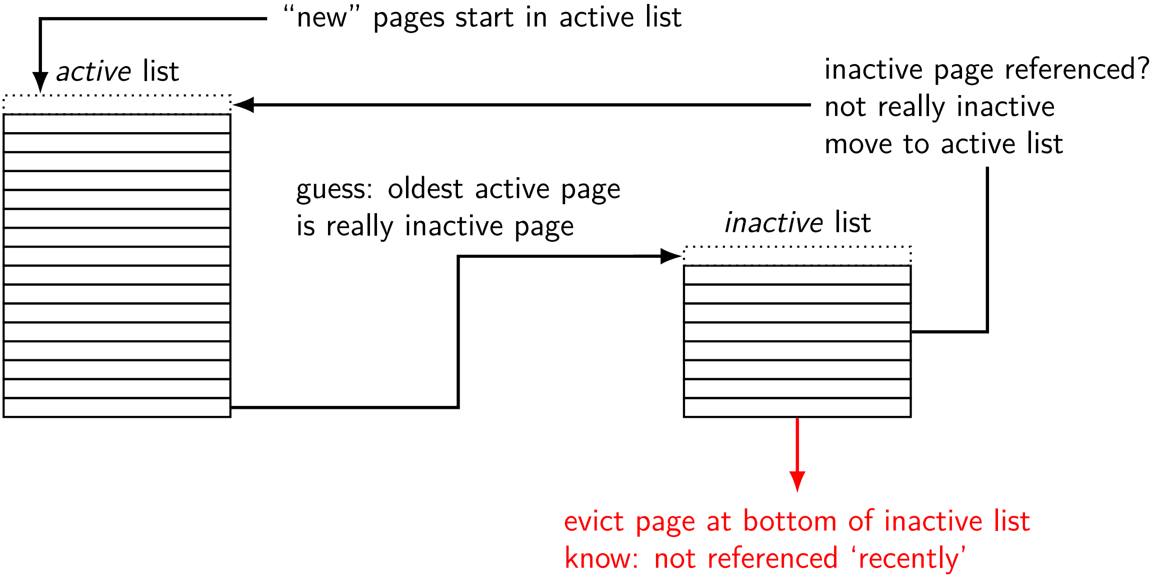
approximating LRU: SEQ



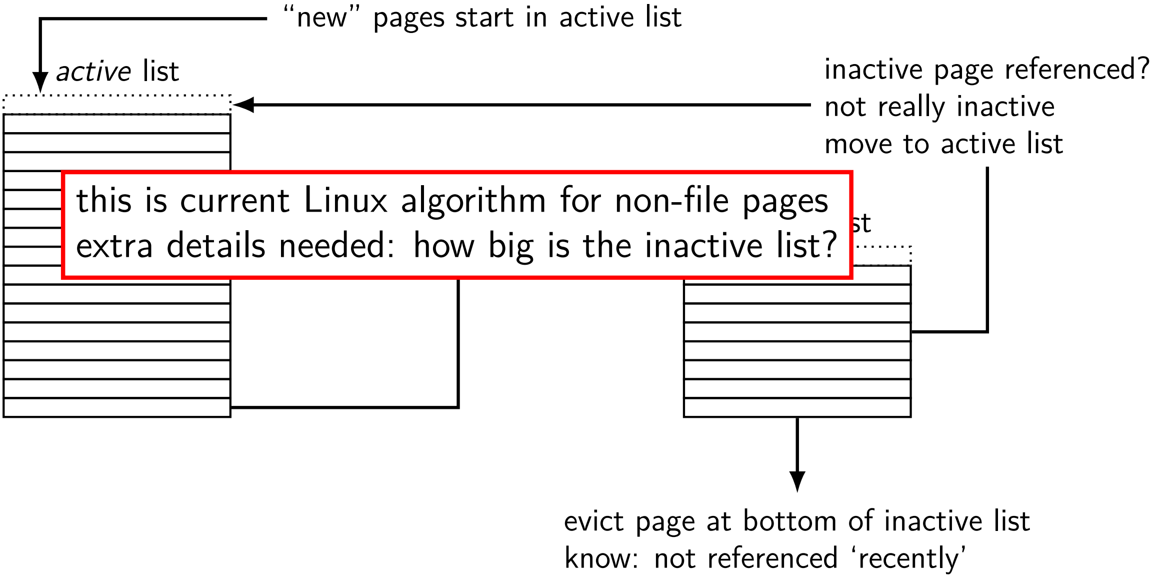
approximating LRU: SEQ



approximating LRU: SEQ

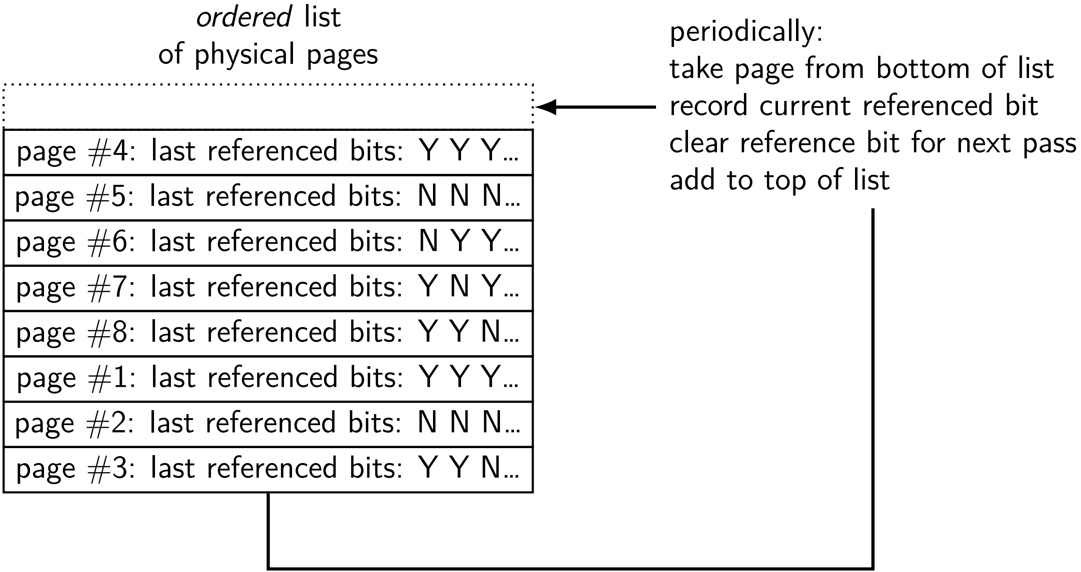


approximating LRU: SEQ

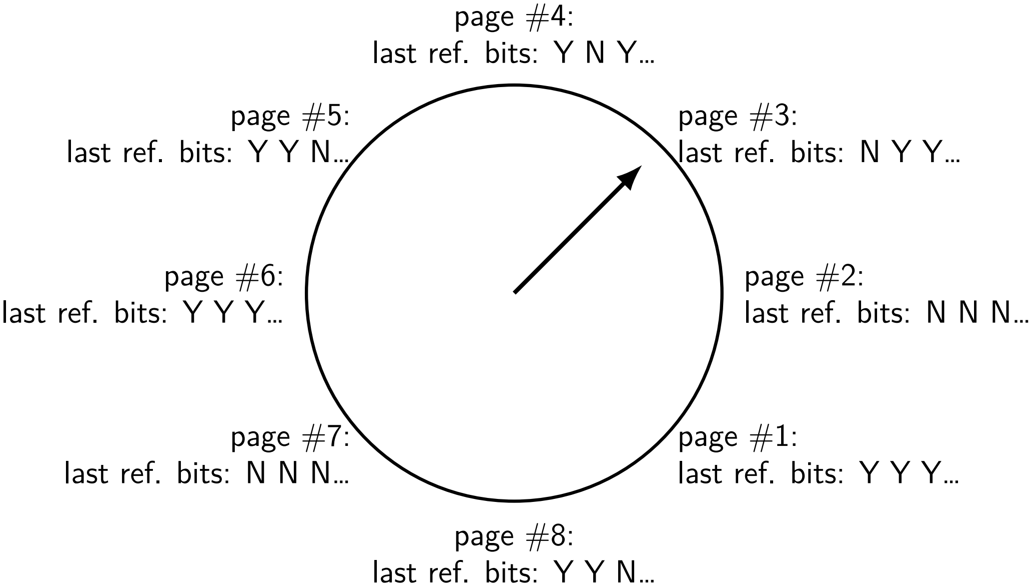


this is current Linux algorithm for non-file pages
extra details needed: how big is the inactive list?

tracking usage: CLOCK (view 1)



tracking usage: CLOCK (view 2)



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 neve run out of memory…all of it?

- no changed data written back

solution: scan for dirty bits periodicially and writeback

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?

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

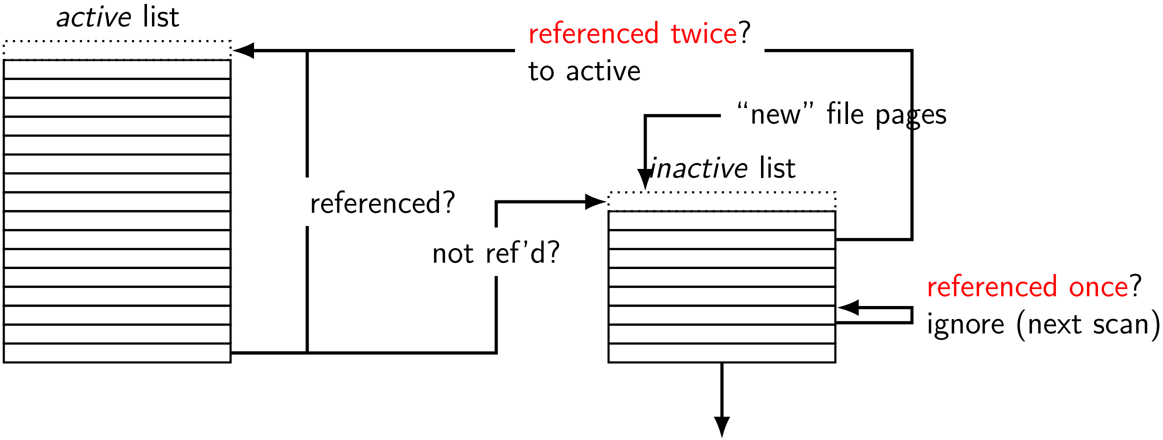
basic idea: don't consider pages active until the second access

single scans of file won't "pollute" cache

without this change: reading large files slows down other programs

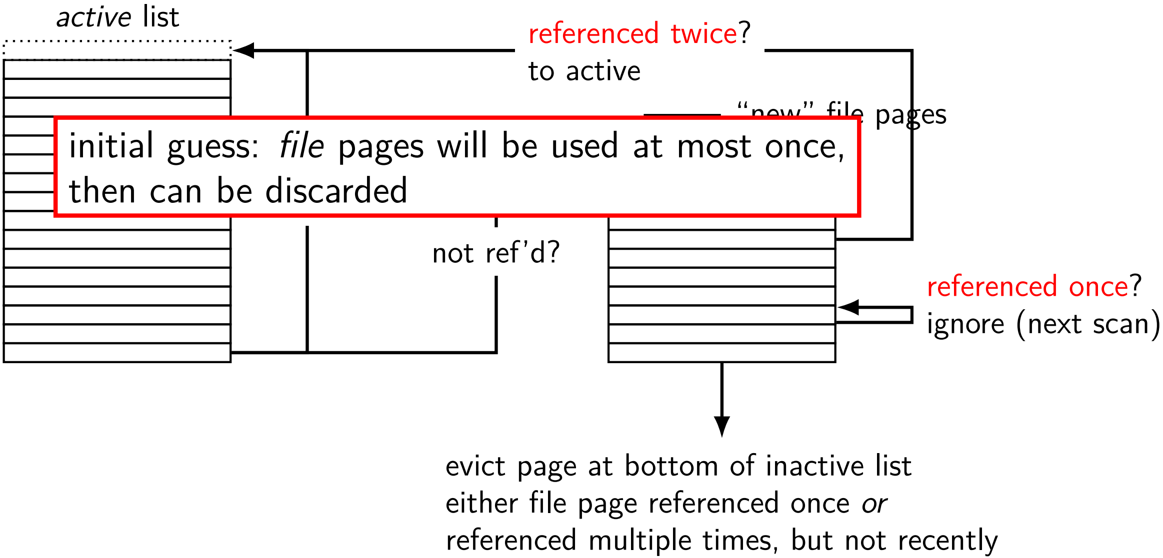
- recently read part of large file steals space from active programs

CLOCK-Pro: special casing for one-use pages



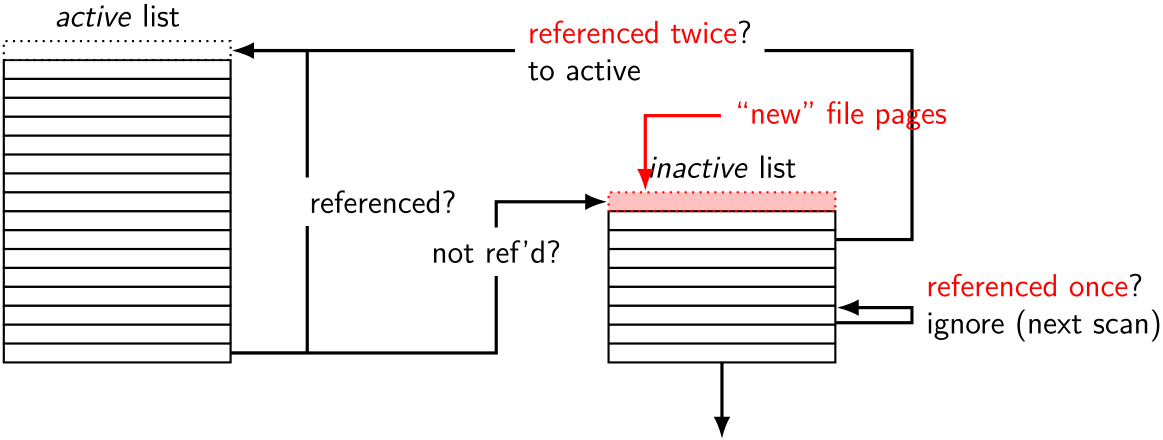
evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



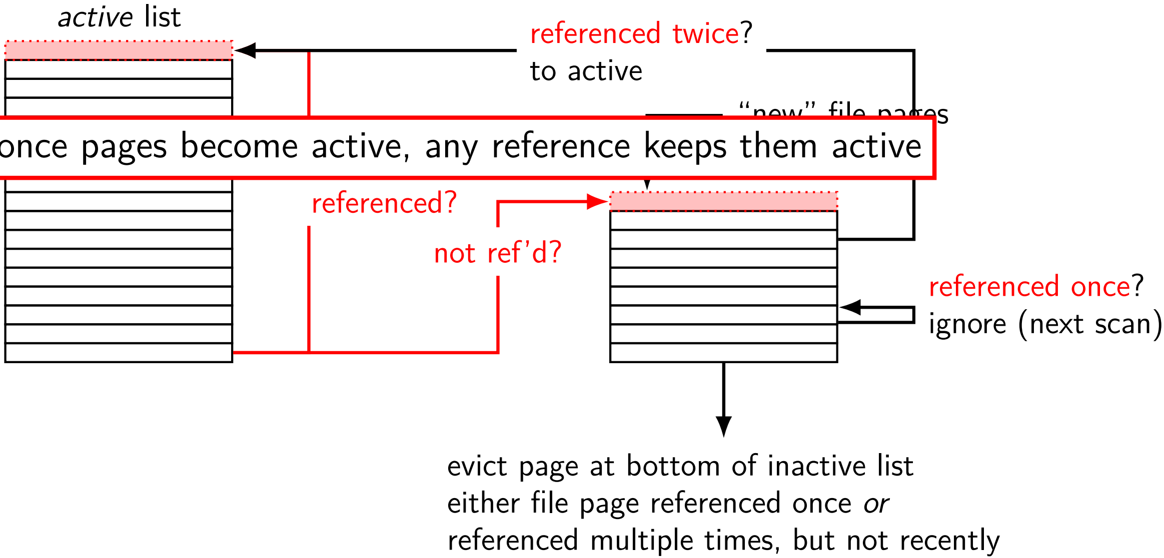
initial guess: *file* pages will be used at most once, then can be discarded

CLOCK-Pro: special casing for one-use pages



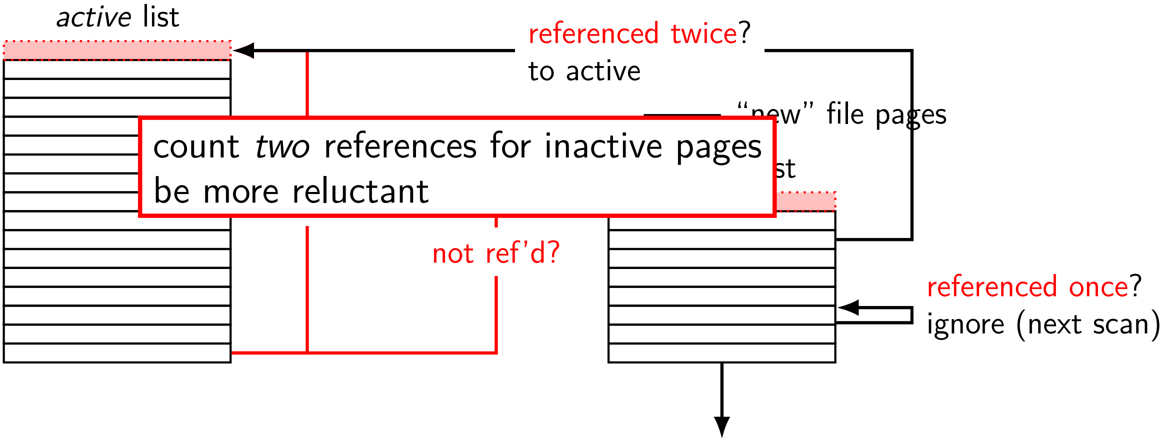
evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



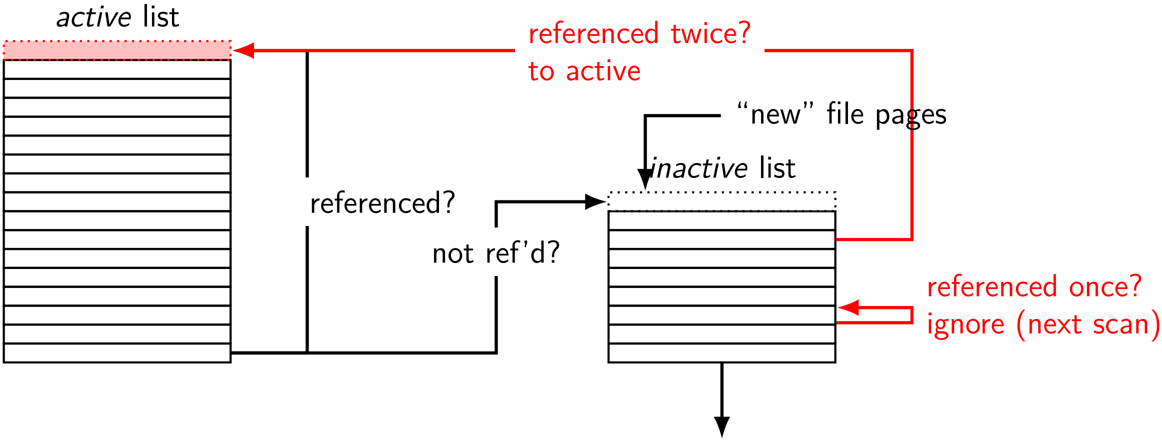
once pages become active, any reference keeps them active

CLOCK-Pro: special casing for one-use pages



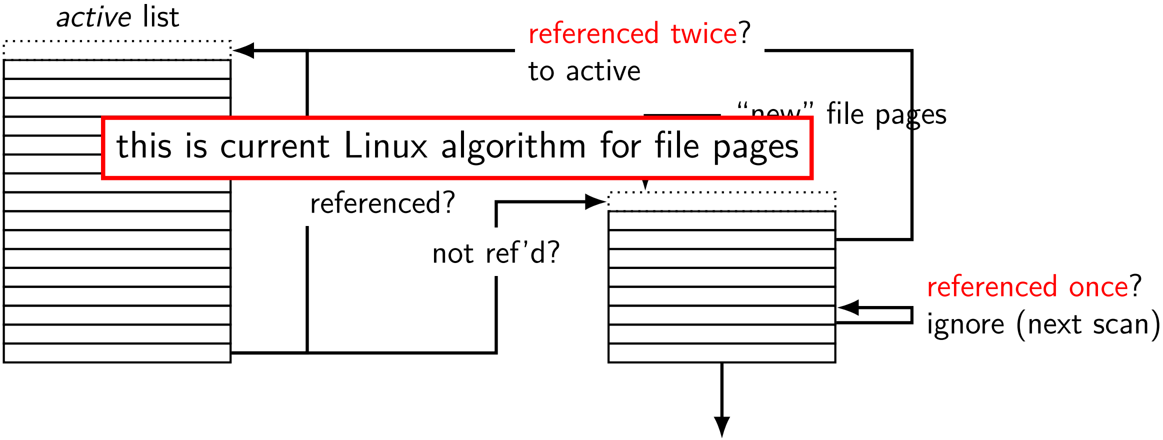
evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

CLOCK-Pro: special casing for one-use pages



evict page at bottom of inactive list
either file page referenced once *or*
referenced multiple times, but not recently

default Linux page replacement summary

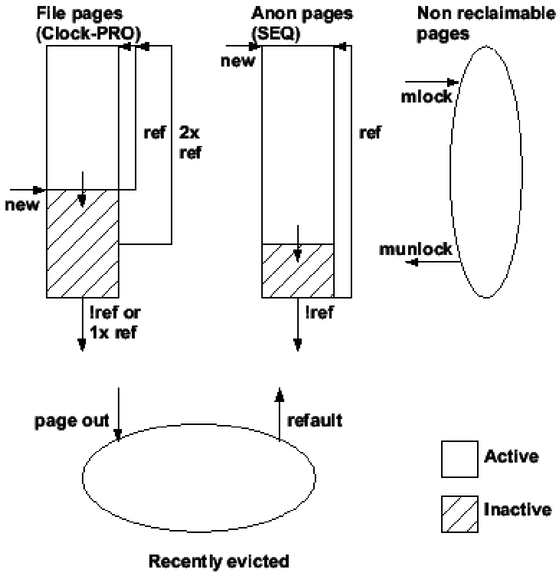


Figure: https://linux-mm.org/PageReplacementDesign

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 → 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

exercise: devise an algorithm to detect to do readahead.

- how to detect the reading pattern?
- when to start reads?
- how much to readahead?
- what state to keep?

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

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 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

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)

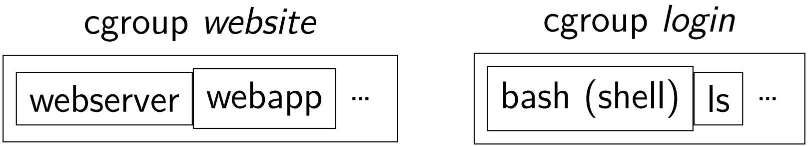
high limit: never let group use more than this

replace pages from this group before anything else

…

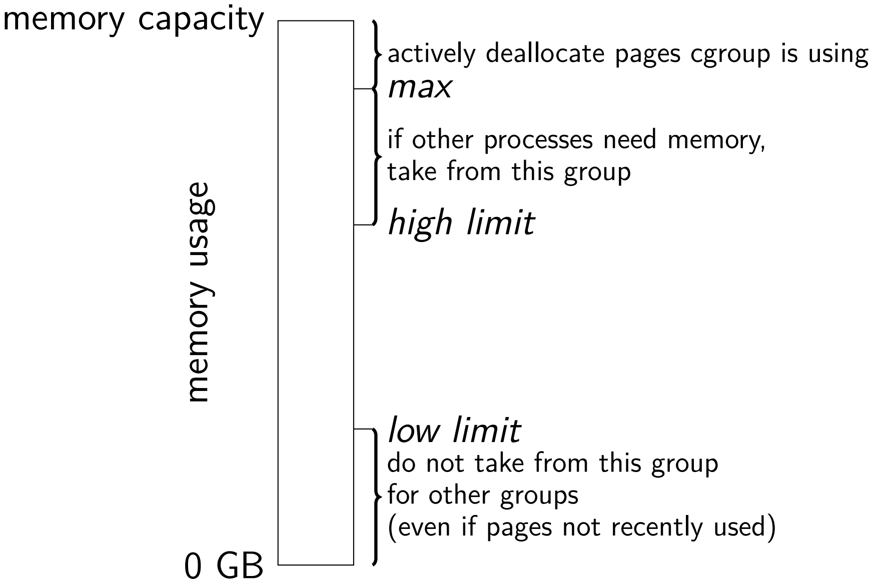
Linux cgroups

Linux mechanism: seperate processes into groups:



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

Linux cgroup memory limits



page cache/replacement summary

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

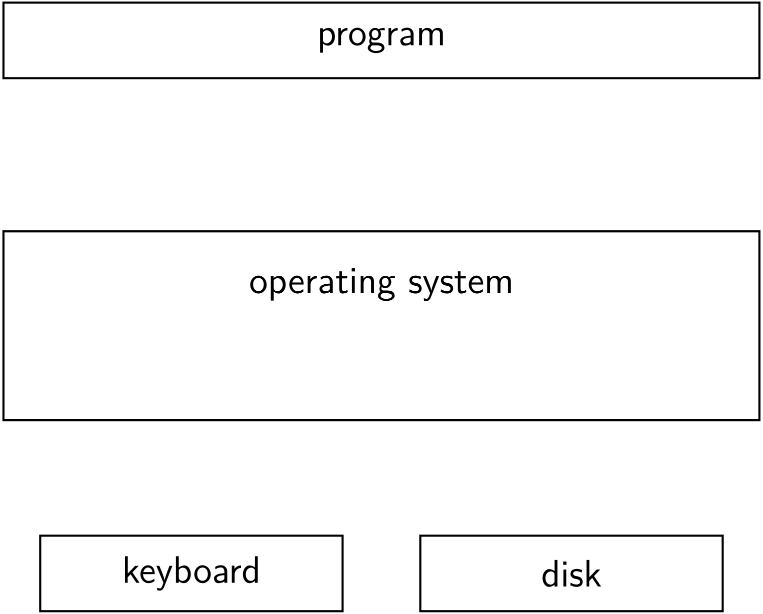
mostly, assume working set model
- keep (hopefully) small active set in memory
- least recently used variants

special cases for non-LRU-friendly patterns (e.g. scans)
- maybe more we haven't discussed?

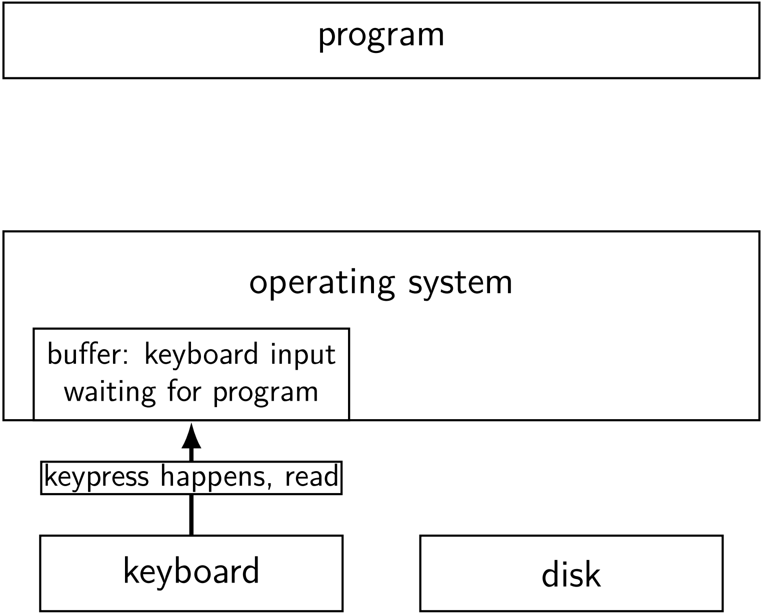
being proactive (writeback when idle, readahead, pool of pre-evicted pages)

handling non-miss-rate goals
- fair replacement: limit active memory per user?
- probably more we haven't discussed here? optimizing throughput? fair throughput between users?

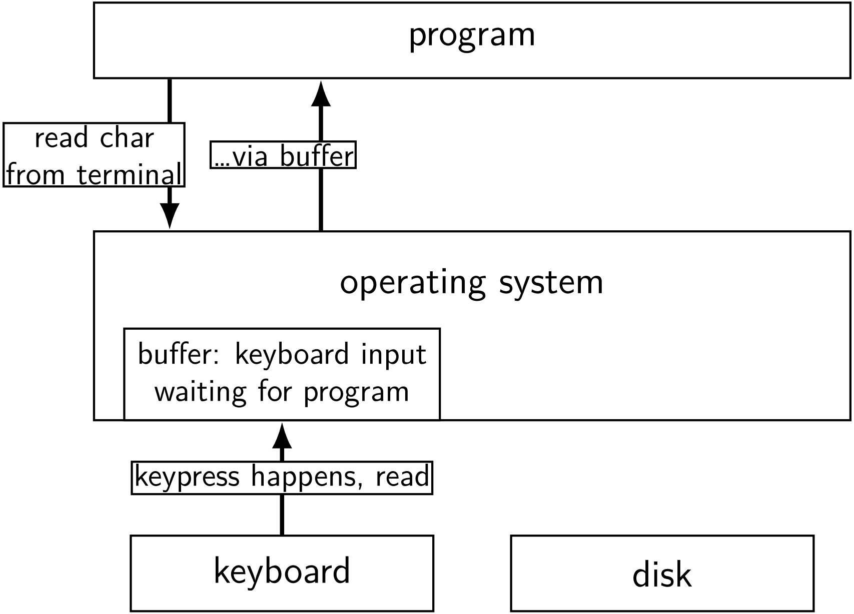
recall: kernel buffering (reads)



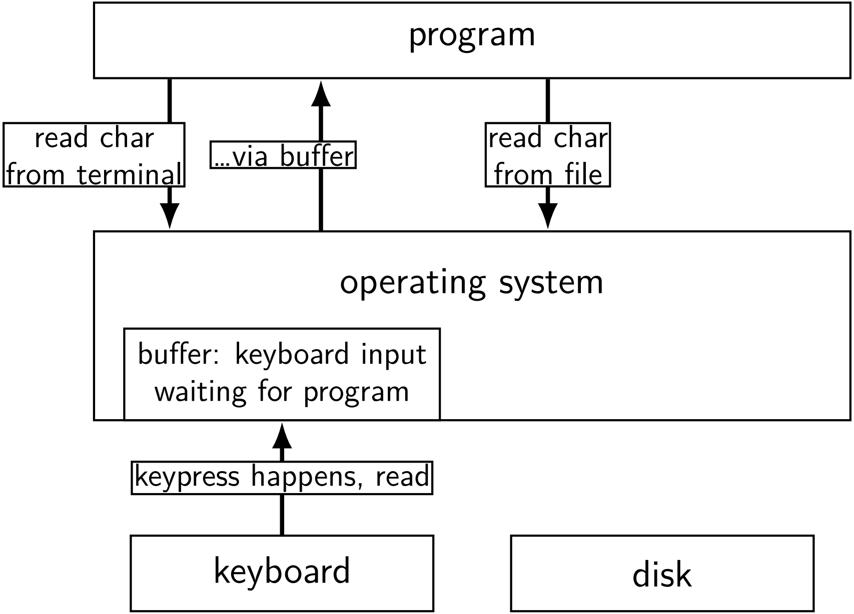
recall: kernel buffering (reads)



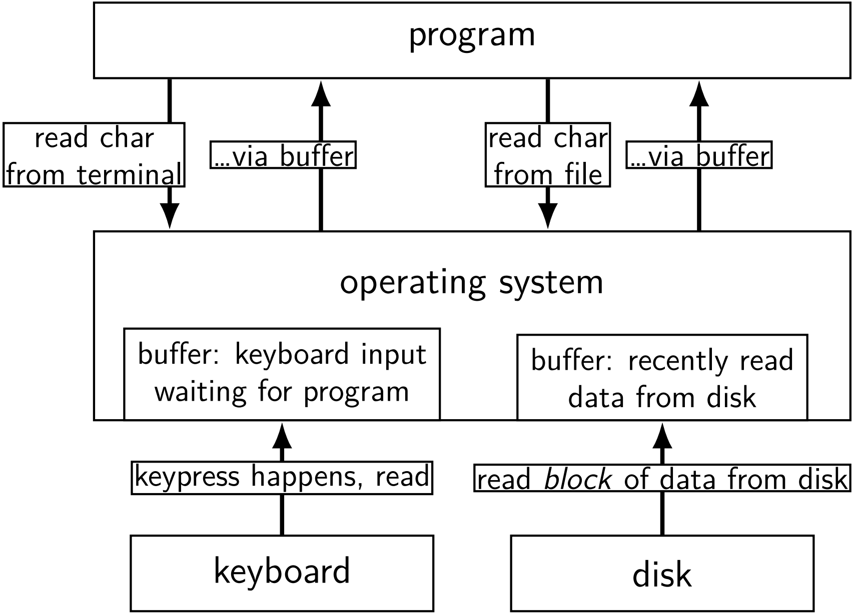
recall: kernel buffering (reads)



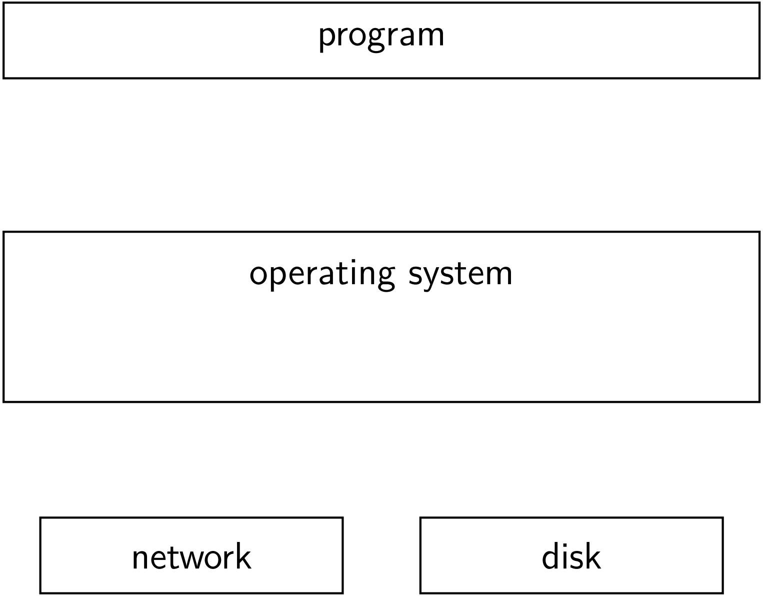
recall: kernel buffering (reads)



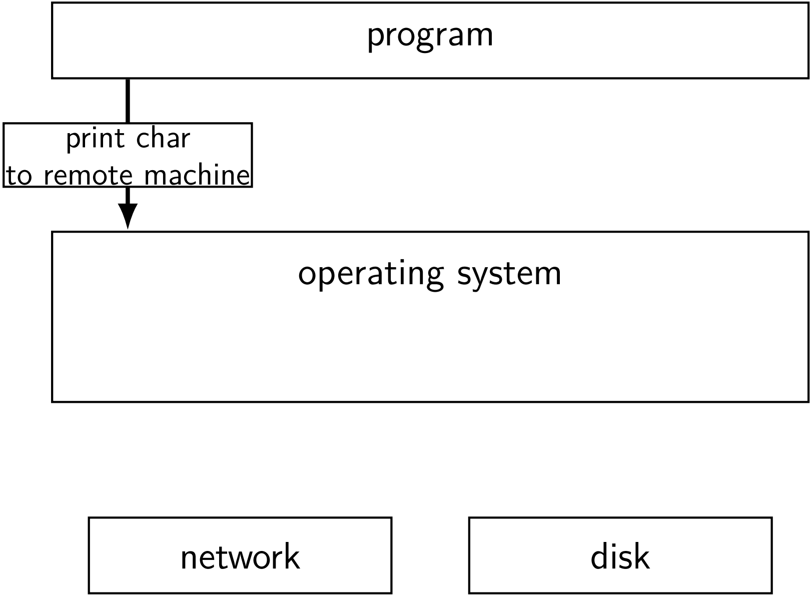
recall: kernel buffering (reads)



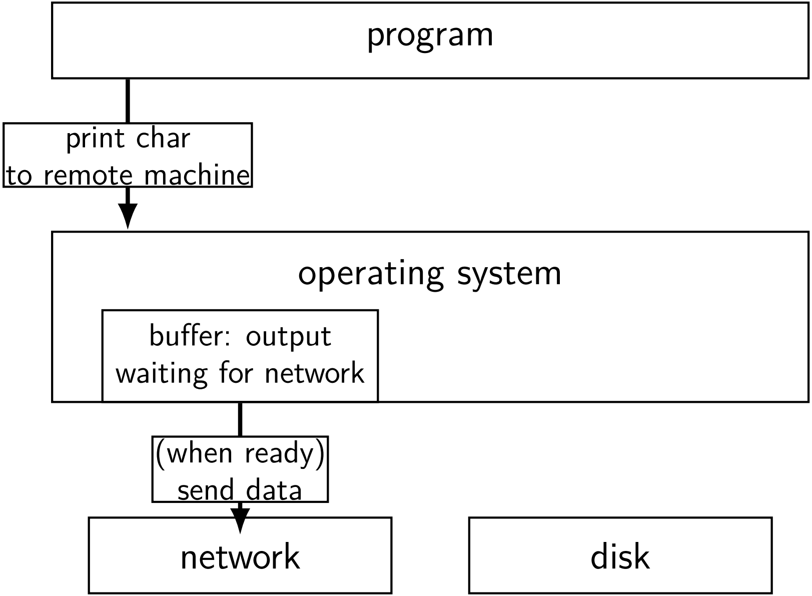
recall: kernel buffering (writes)



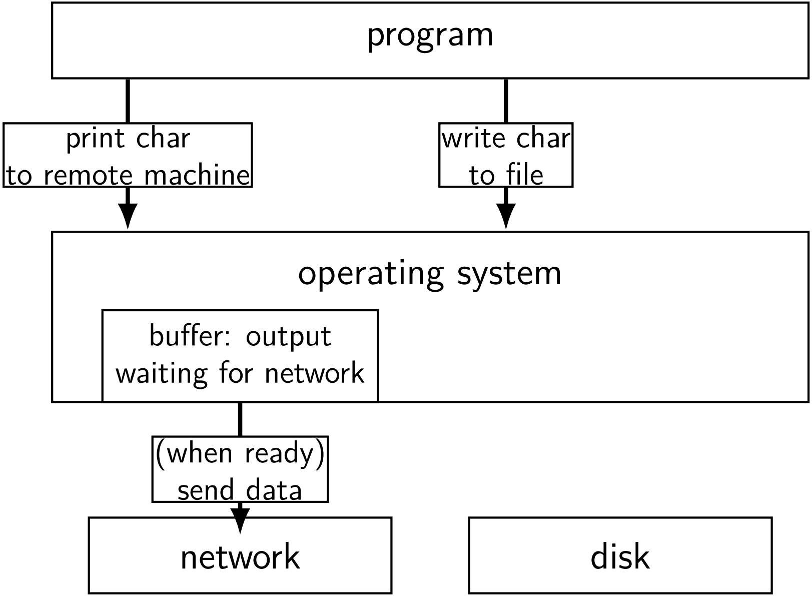
recall: kernel buffering (writes)



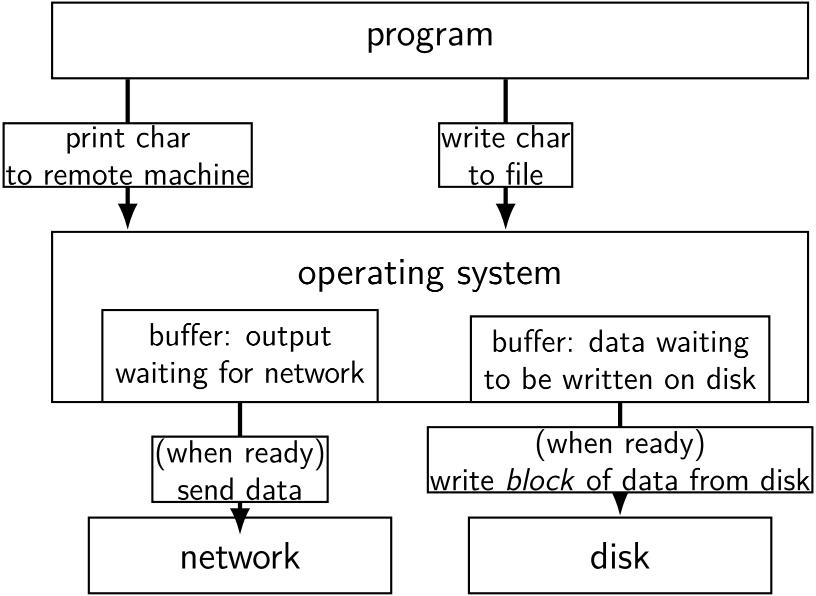
recall: kernel buffering (writes)



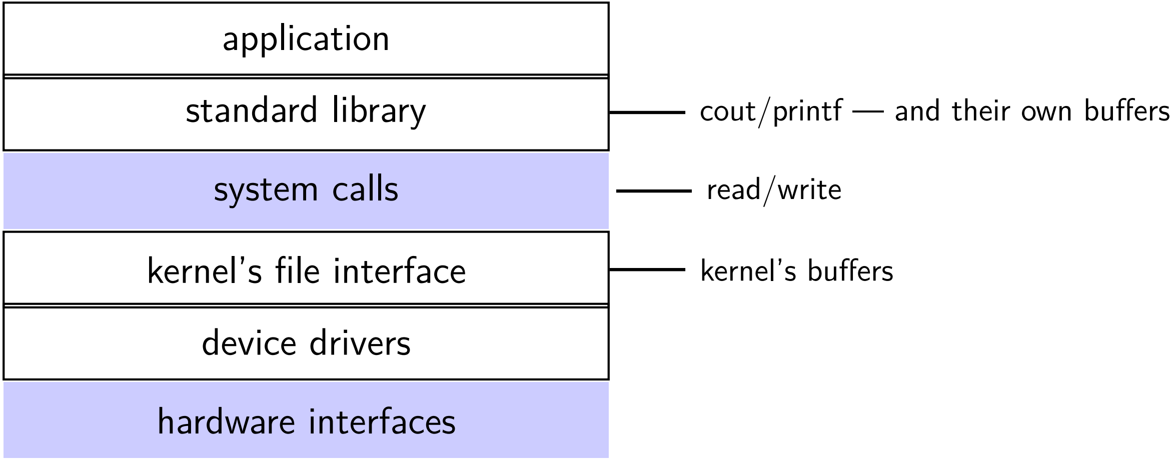
recall: kernel buffering (writes)



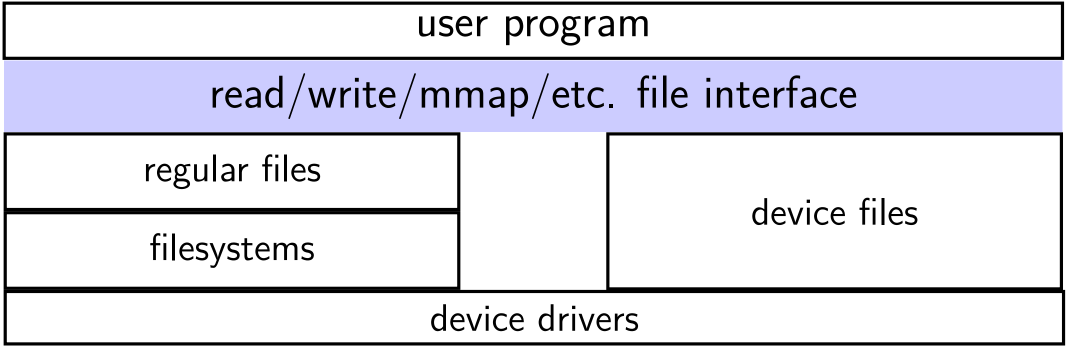
recall: kernel buffering (writes)



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?

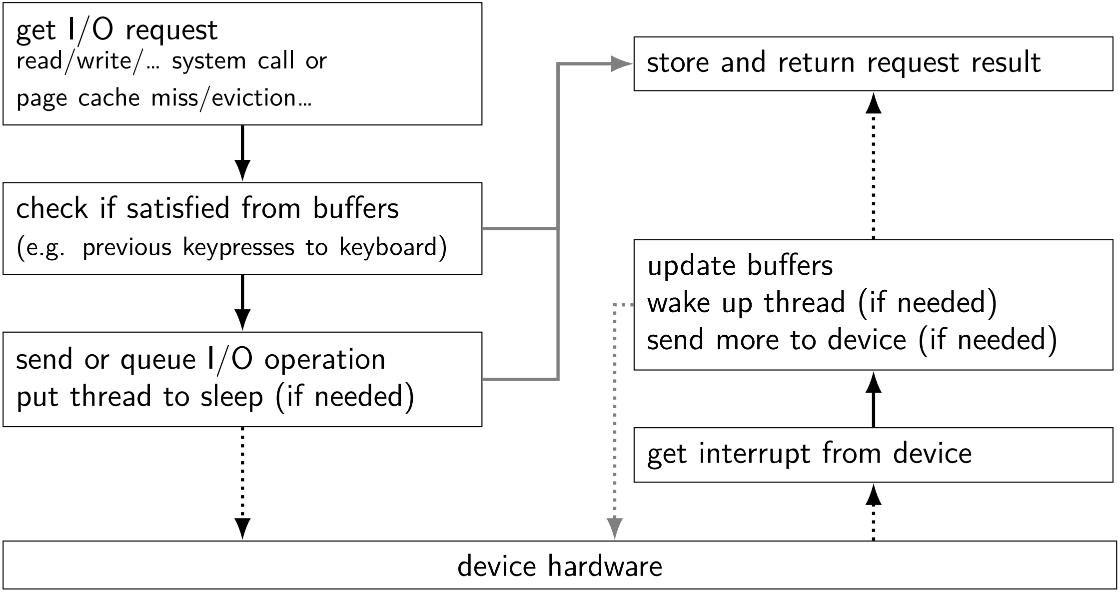
read/write/mmap not enough?

- 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?
- ...

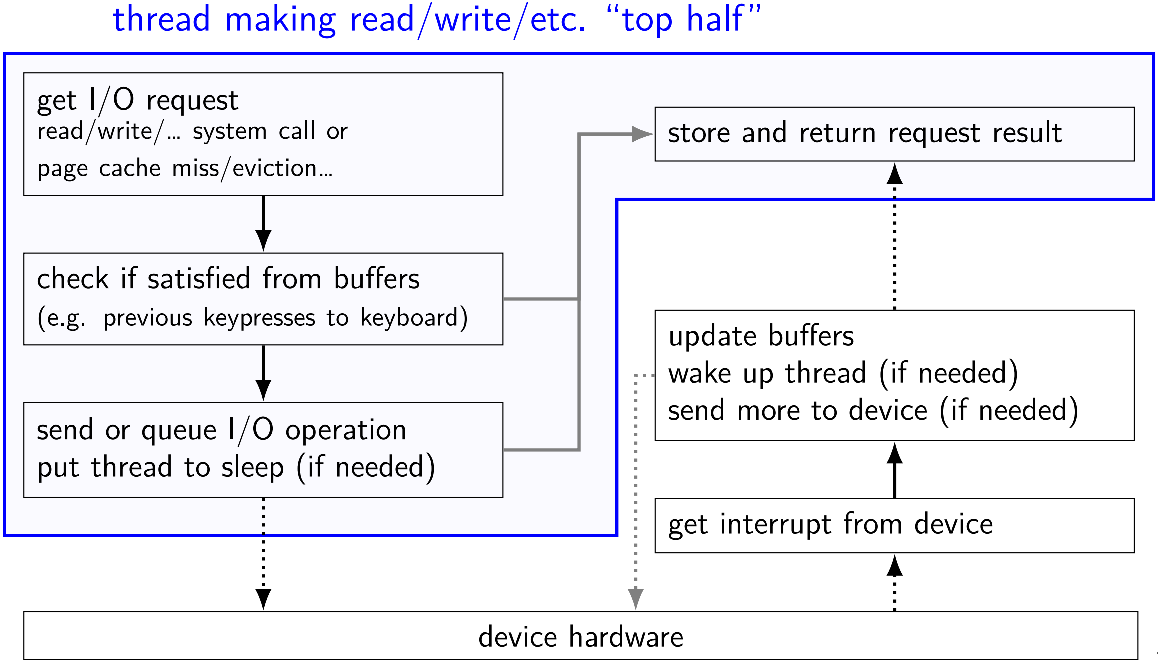
extra POSIX file descriptor operations:

- ioctl (general I/O control)
- tcget/setaddr (for terminal settings)
- fcntl
- ...

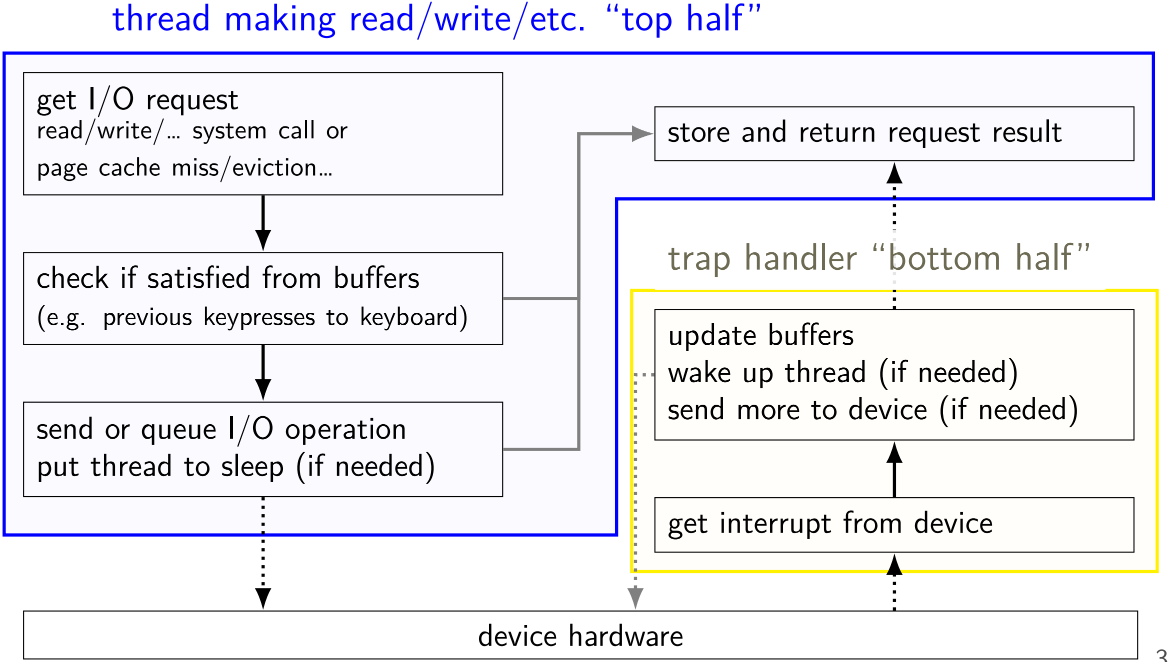
device driver flow



device driver flow



device driver flow



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

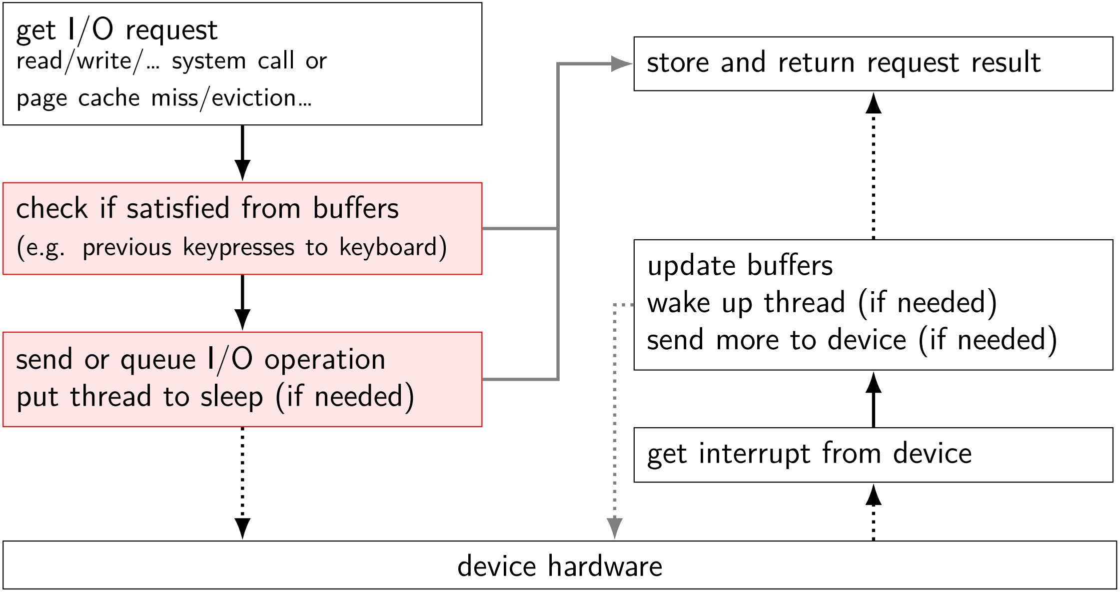
code run at boot:

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

CONSOLE is a constant

consoleread/consolewrite: run when you read/write console

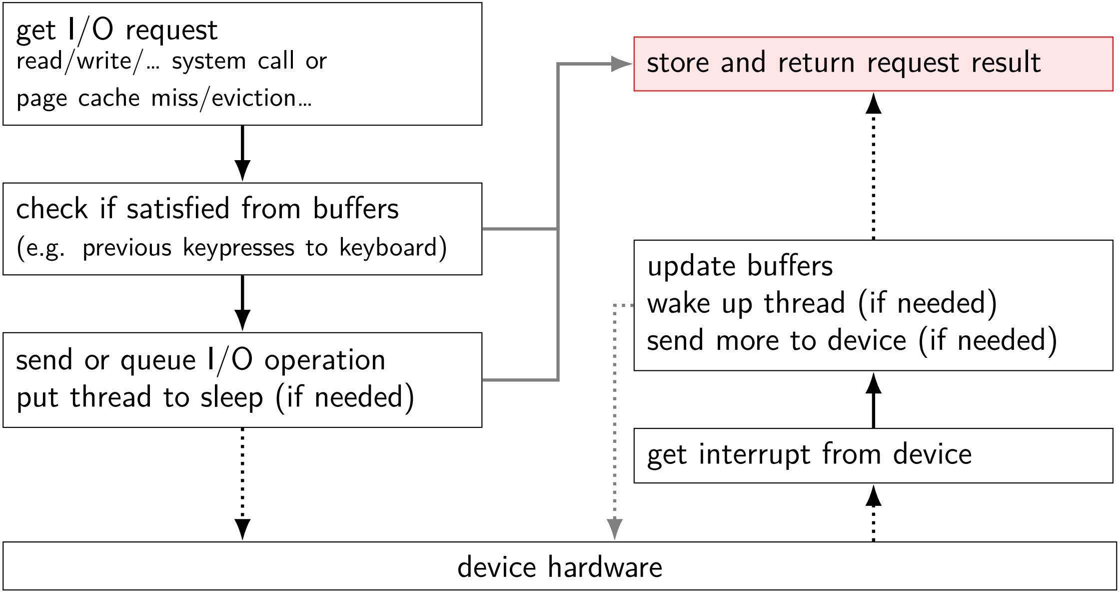
device driver flow



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

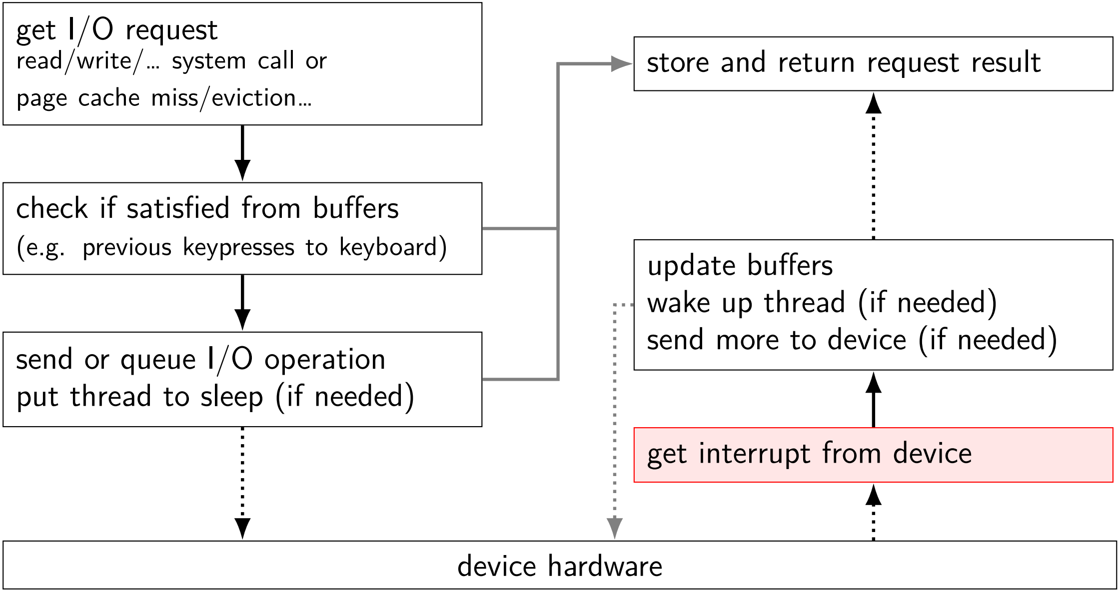
wait for buffer to fill

- 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”

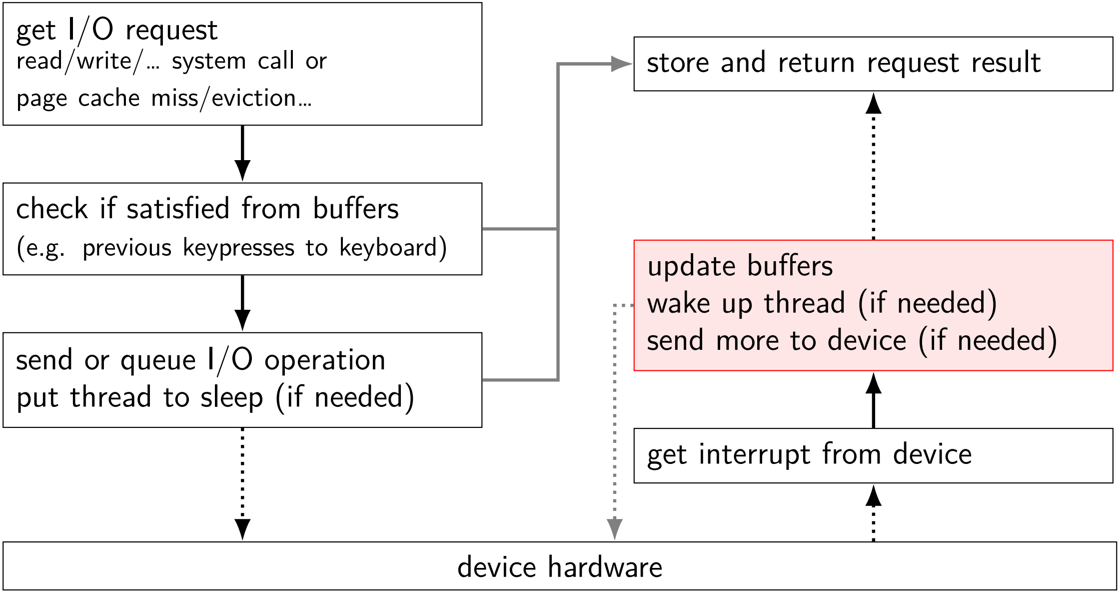
xv6: console interrupt (one case)

```
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



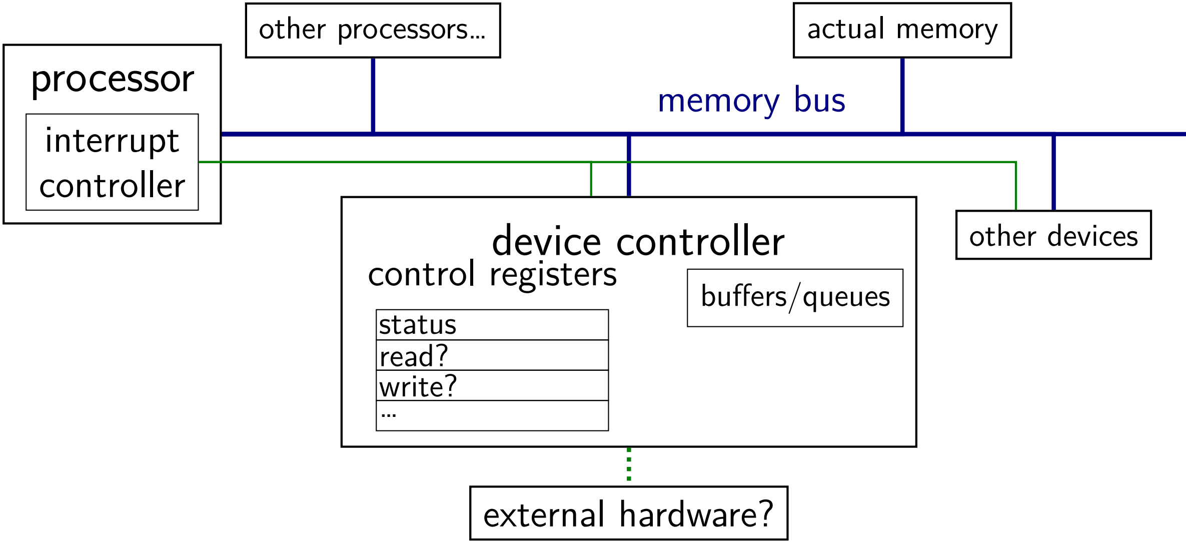
xv6: console interrupt reading

kbdintr fuction actually reads from device

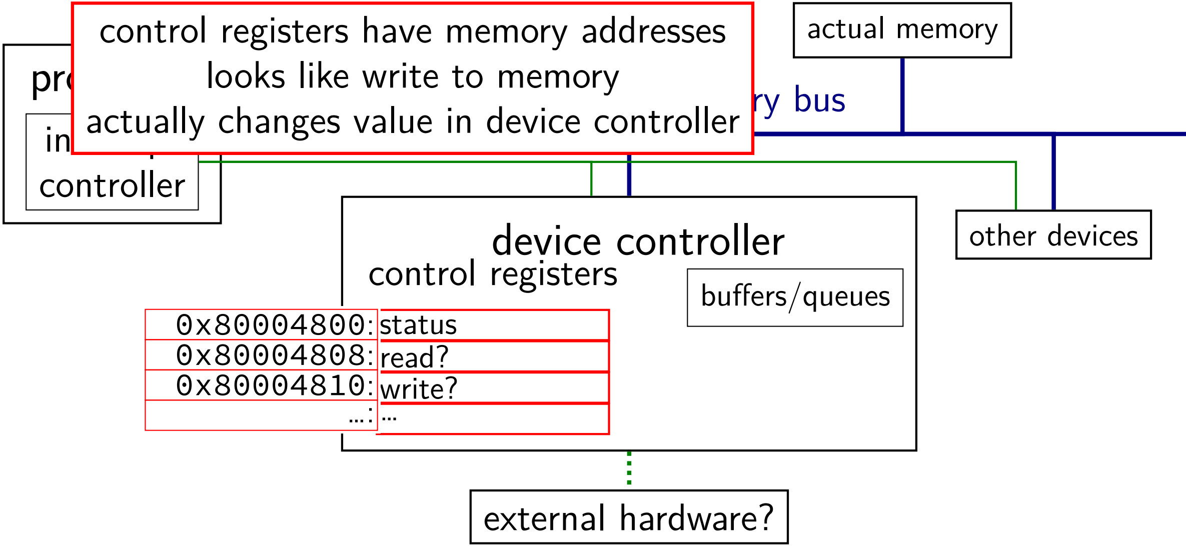
adds data to buffer (if room)

wakes up sleeping thread (if any)

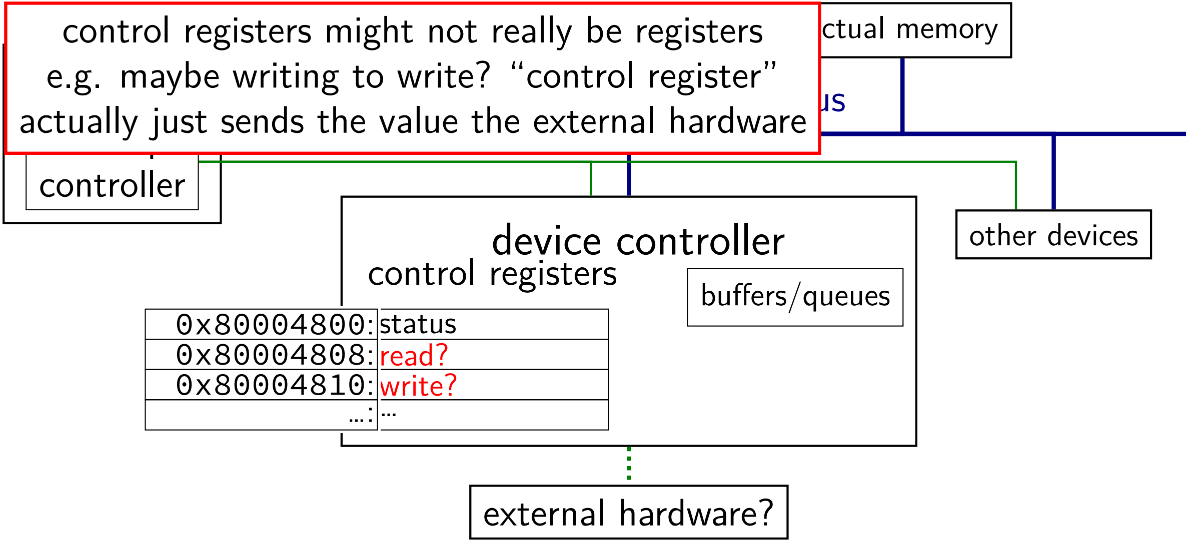
connecting devices



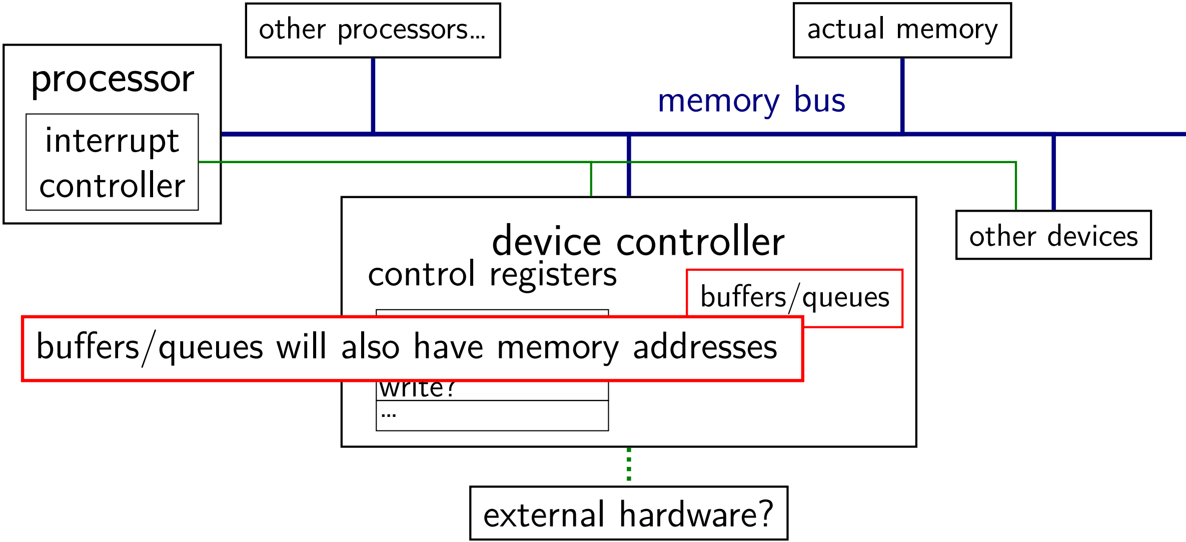
connecting devices



connecting devices

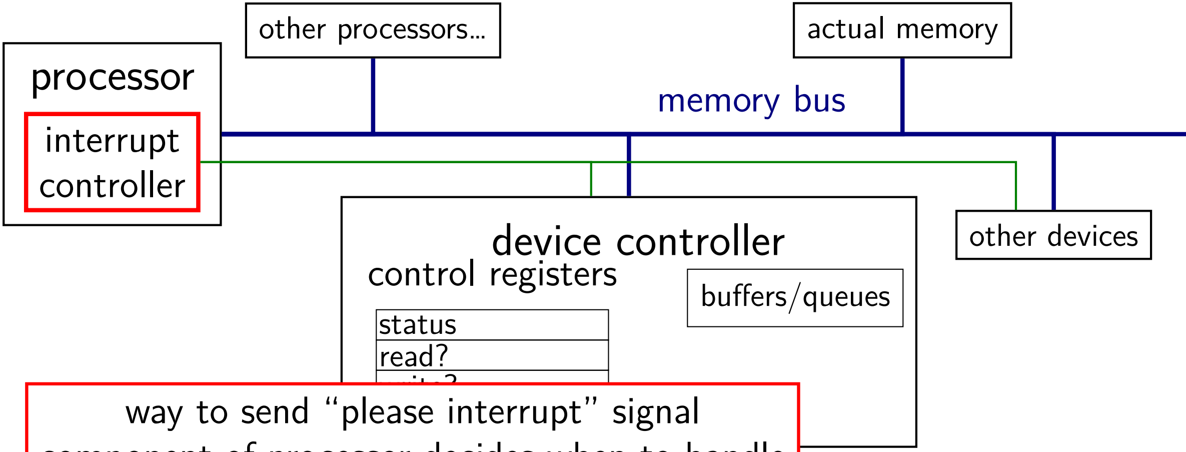


connecting devices



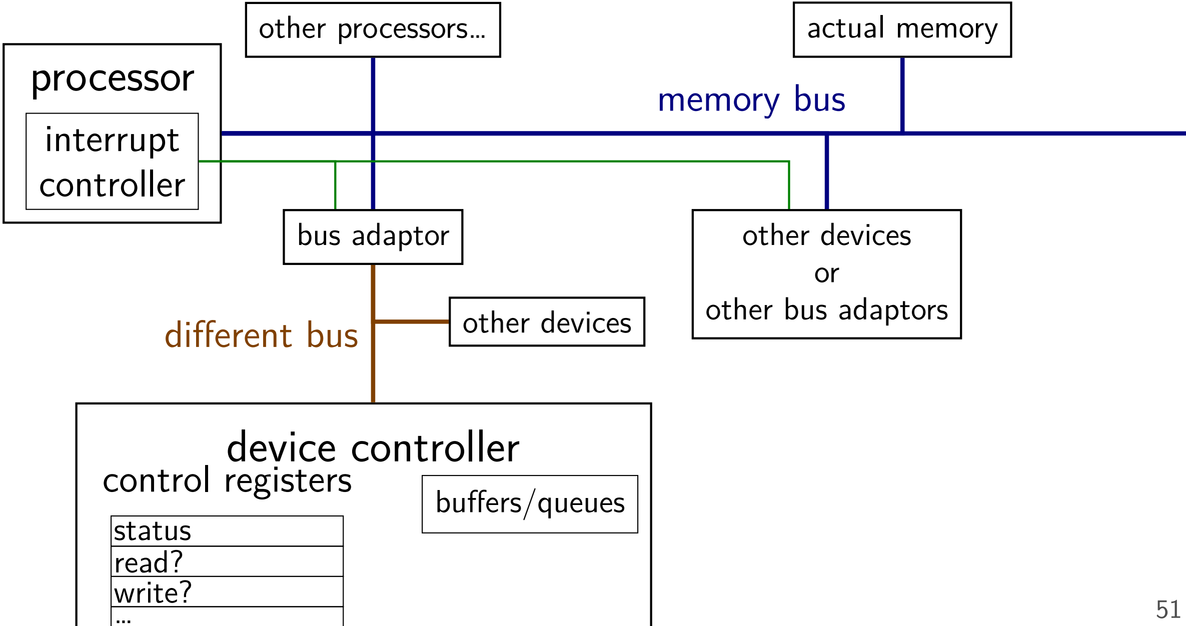
buffers/queues will also have memory addresses

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

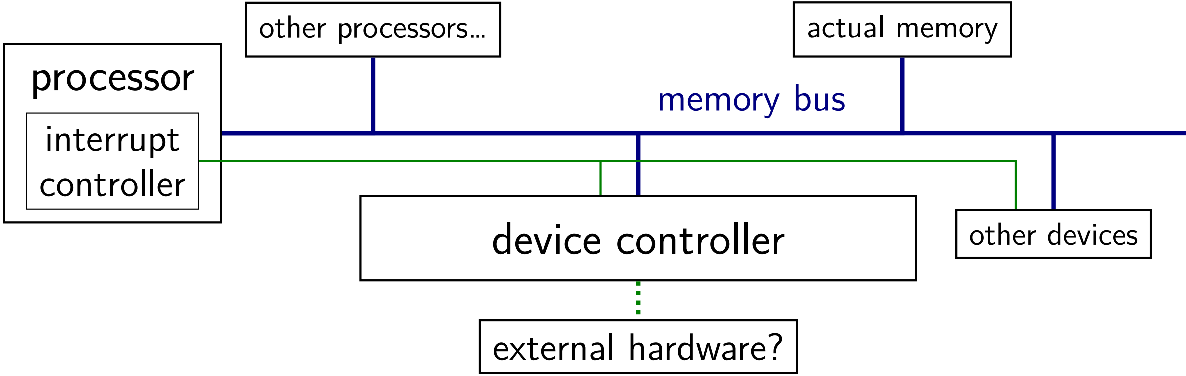
"programmed I/O": write to or read from device controller buffers directly

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?

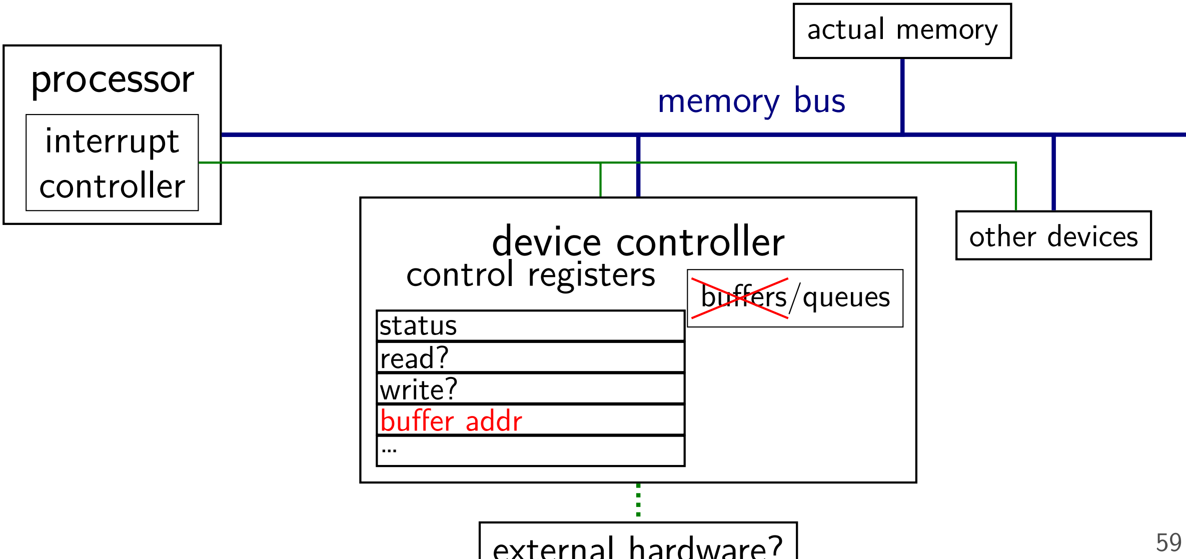
direct memory access (DMA)



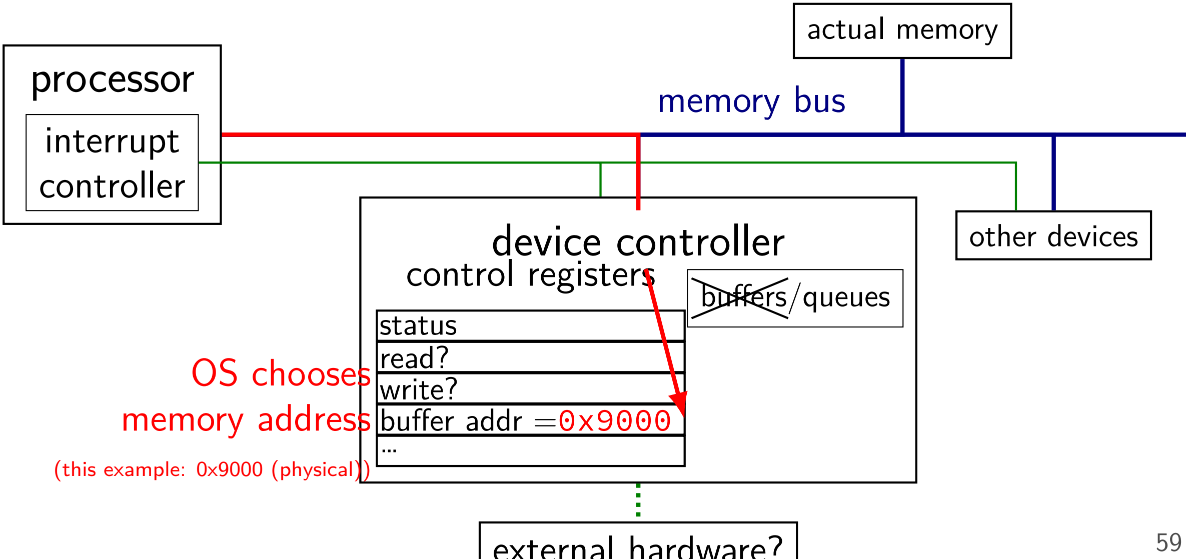
observation: devices can read/write memory

can have device copy data to/from memory

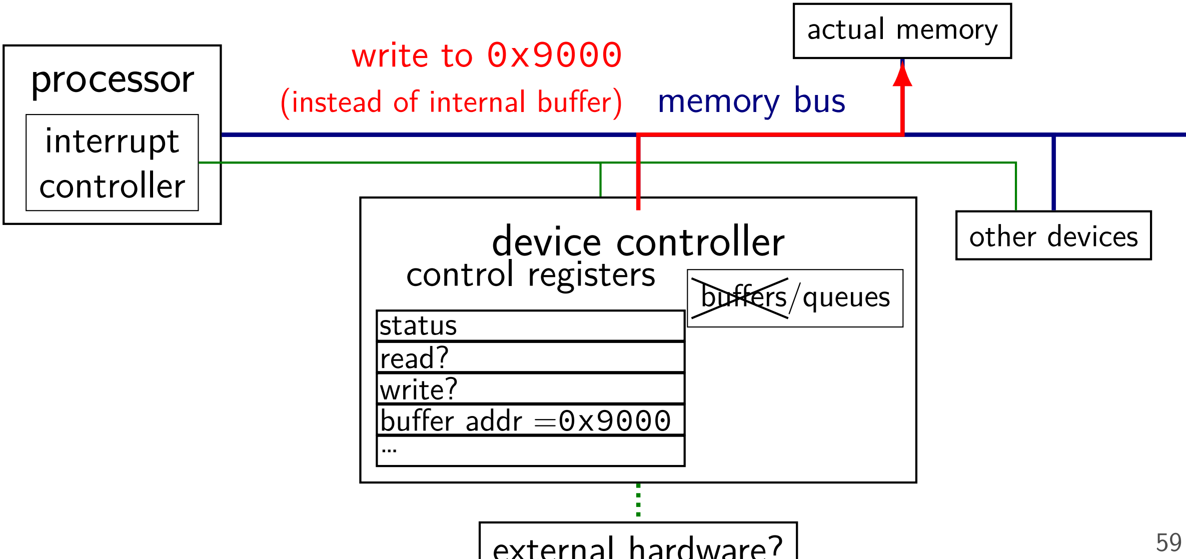
direct memory access (DMA)



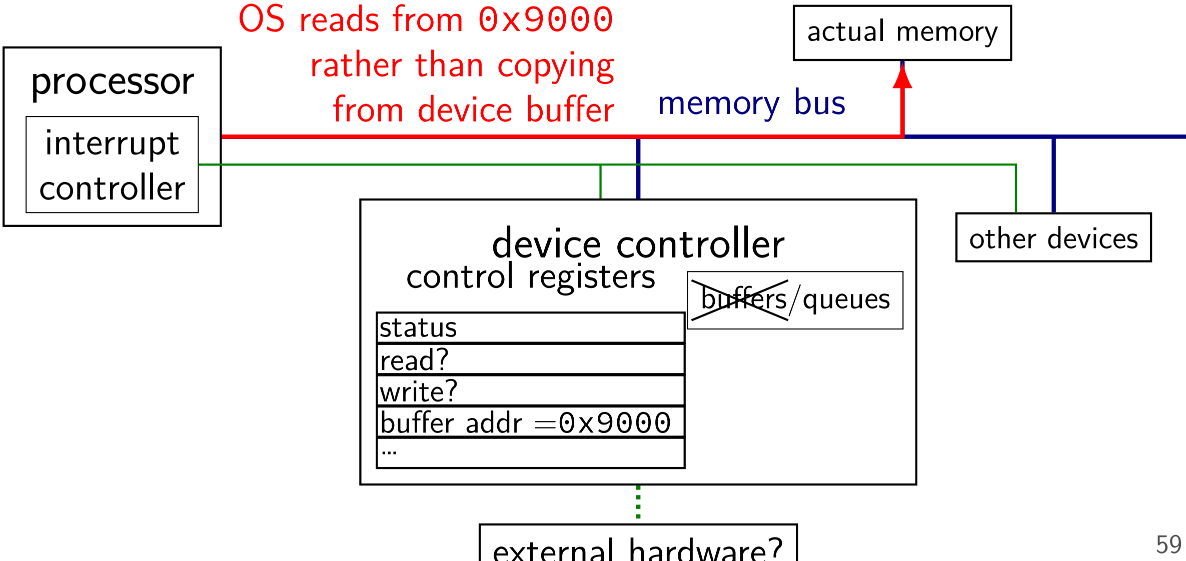
direct memory access (DMA)



direct memory access (DMA)

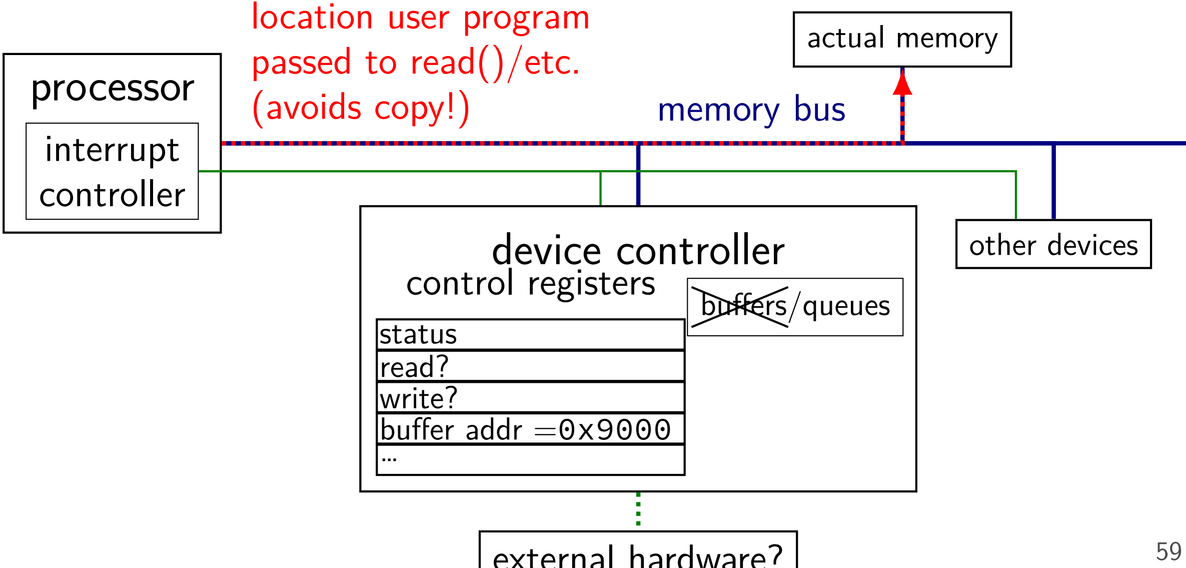


direct memory access (DMA)



direct memory access (DMA)

best case: OS chooses location user program passed to read()/etc. (avoids copy!)



direct memory access (DMA)

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

avoids having processor run a loop

allows device to use memory as very large buffer space

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

allows device to put data where OS wants it directly (maybe)

direct memory access protocol

device stores buffer *address* instead of buffer

OS's job: allocate buffer, keep around while data being transferred

end of transfer indicated via interrupt and/or control registers

IOMMUs

typically, direct memory access requires using physical addresses

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

recent systems have an IO Memory Management Unit

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

IOMMUs

typically, direct memory access requires using physical addresses

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

recent systems have an IO Memory Management Unit

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

hard drive interfaces

hard drives and solid state disks are divided into sectors

historically 512 bytes (larger on recent disks)

disk commands:

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

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

filesystems

filesystems: store hierarchy of directories on disk

disk is a flat list of blocks of data

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

which sectors? parts of sectors?

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

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

making a new file: where to put it?

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

the FAT filesystem

FAT: File Allocation Table

probably simplest widely used filesystem (family)

named for important data structure: *file allocation table*

FAT and sectors

FAT divides disk into *clusters*

- composed of one or more sectors
- sector = minimum amount hardware can read

cluster: typically 512 to 4096 bytes

a file's data is stored in clusters

reading a file: determine the list of clusters

FAT: the file allocation table

big array on disk, one entry per cluster

each entry contains a number — usually "next cluster"

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

FAT: reading a file (1)

get (from elsewhere) first cluster of data

linked list of cluster numbers

next pointers? file allocation table entry for cluster

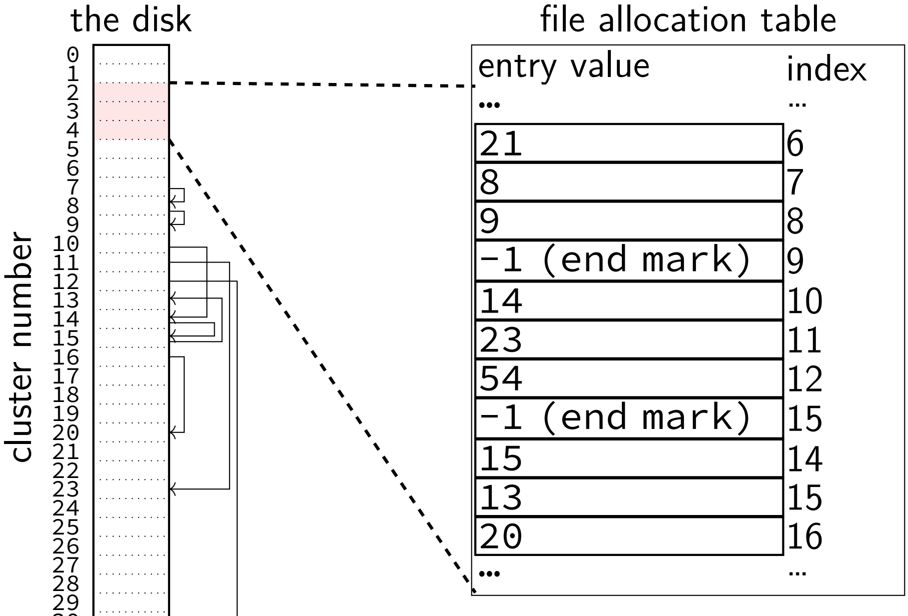
special value for NULL

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

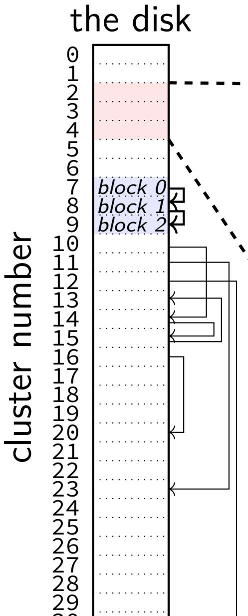
file starting at cluster 10 contains data i

cluster 10, then 14, then 15, then 13

FAT: reading a file (2)



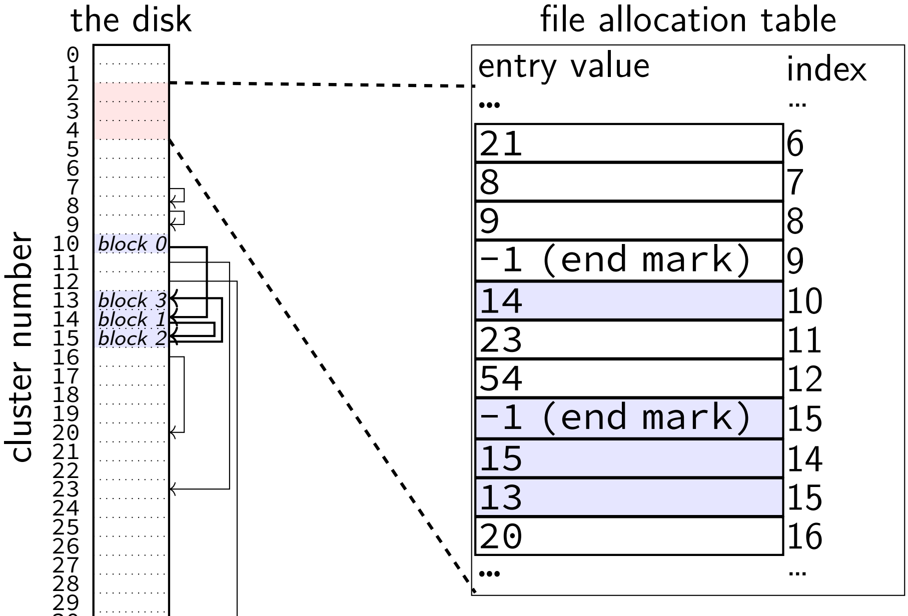
FAT: reading a file (2)



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 files

to read a file given it's start location

read the starting cluster X

get the next cluster Y from FAT entry X

read the next cluster

get the next cluster from FAT entry Y

…

until you see an end marker

start locations?

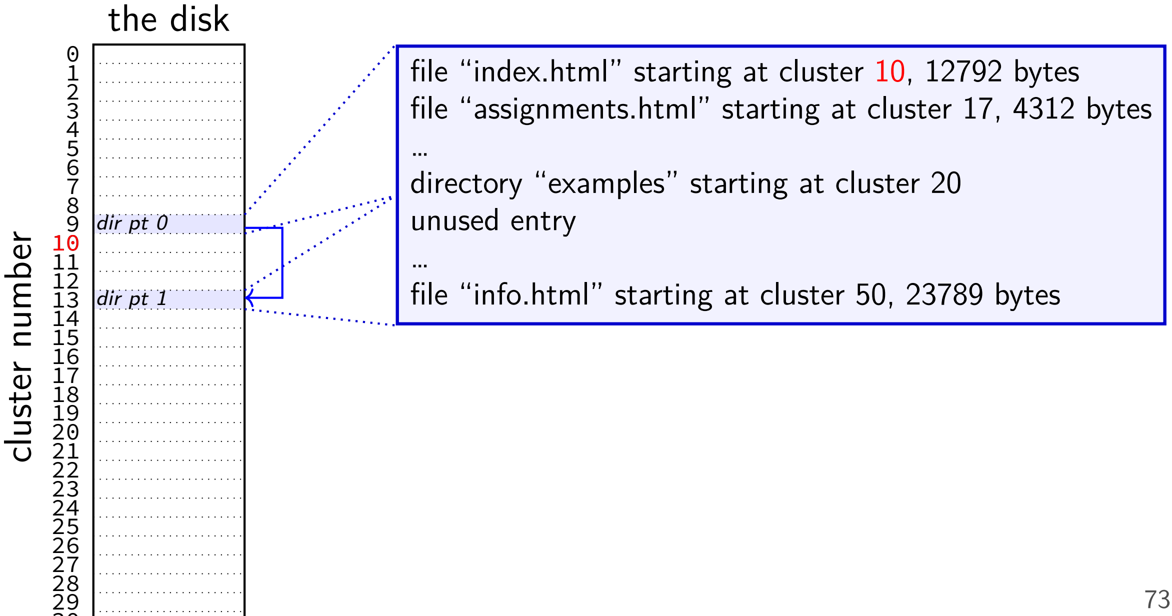
really want filenames

stored in directories!

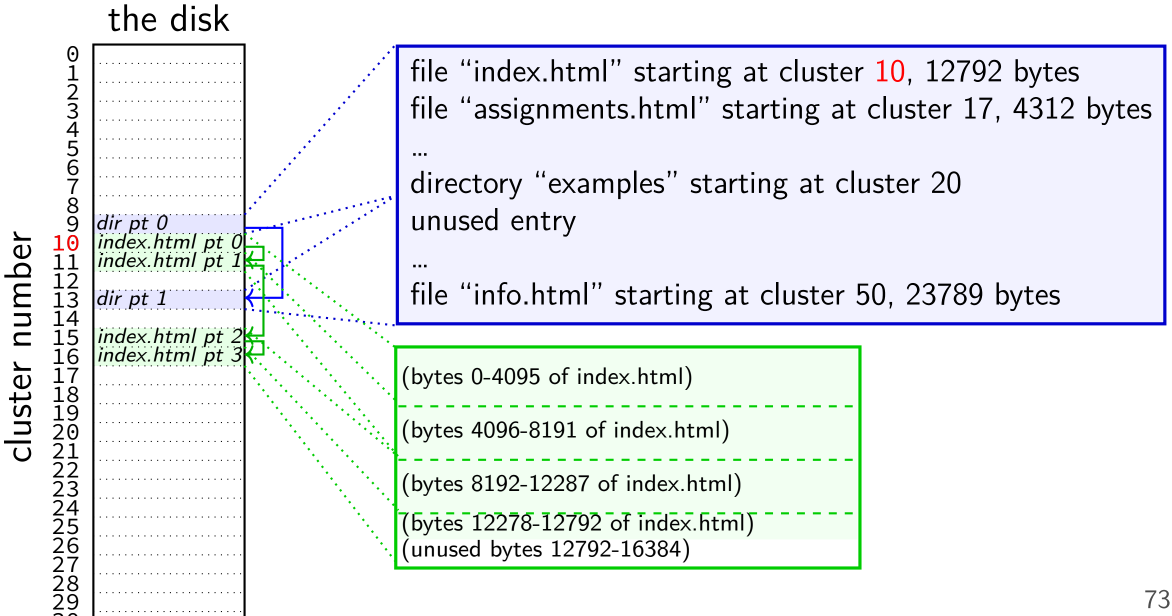
in FAT: directory is a list of:

(name, starting location, other data about file)

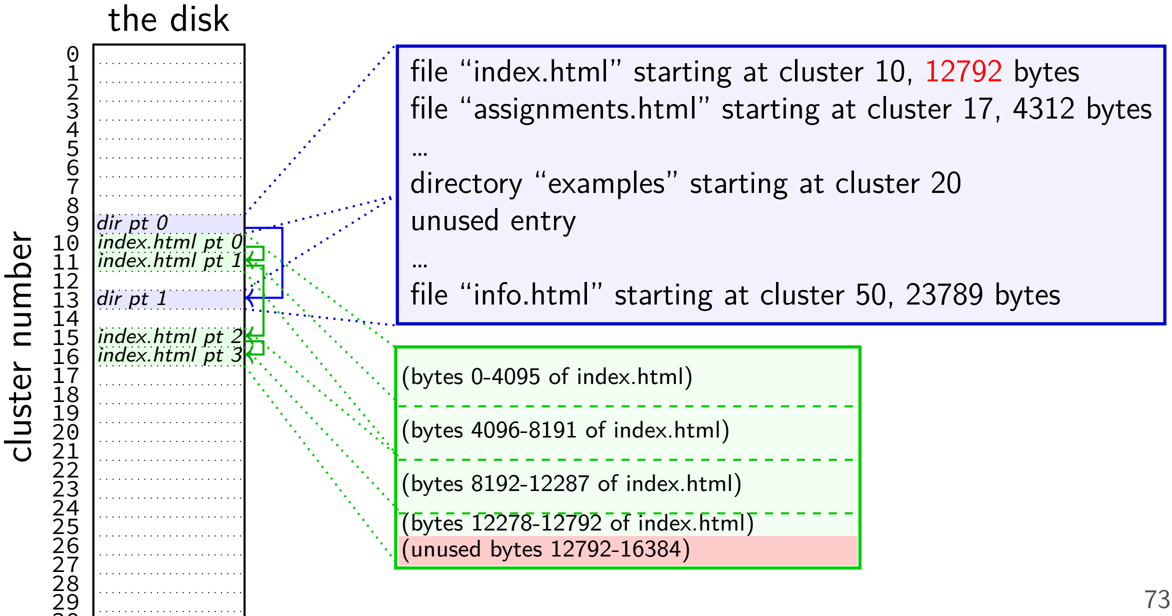
finding files with directory



finding files with directory



finding files with directory



FAT directory entry

box = 1 byte

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

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

FAT directory entries (from C)

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

GCC/Clang extension to disable padding
normally compilers add padding to structs
(to avoid splitting values across cache blocks or pages)

FAT directory entries (from C)

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

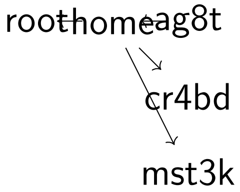
8/16/32-bit unsigned integer
use exact size that's on disk
just copy byte-by-byte from disk to memory
(and everything happens to be little-endian)

FAT directory entries (from C)

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

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

trees of directories



nested directories

foo/bar/baz/file.txt

read root directory entries to find foo

read foo's directory entries to find bar

read bar's directory entries to find baz

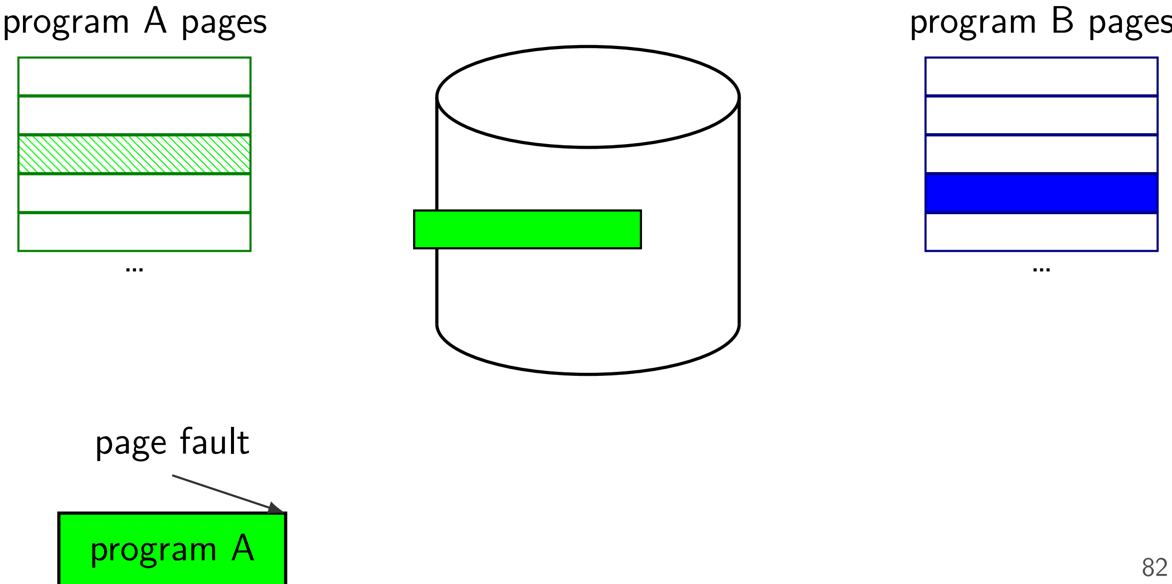
read baz's directory entries to find file.txt

the root directory?

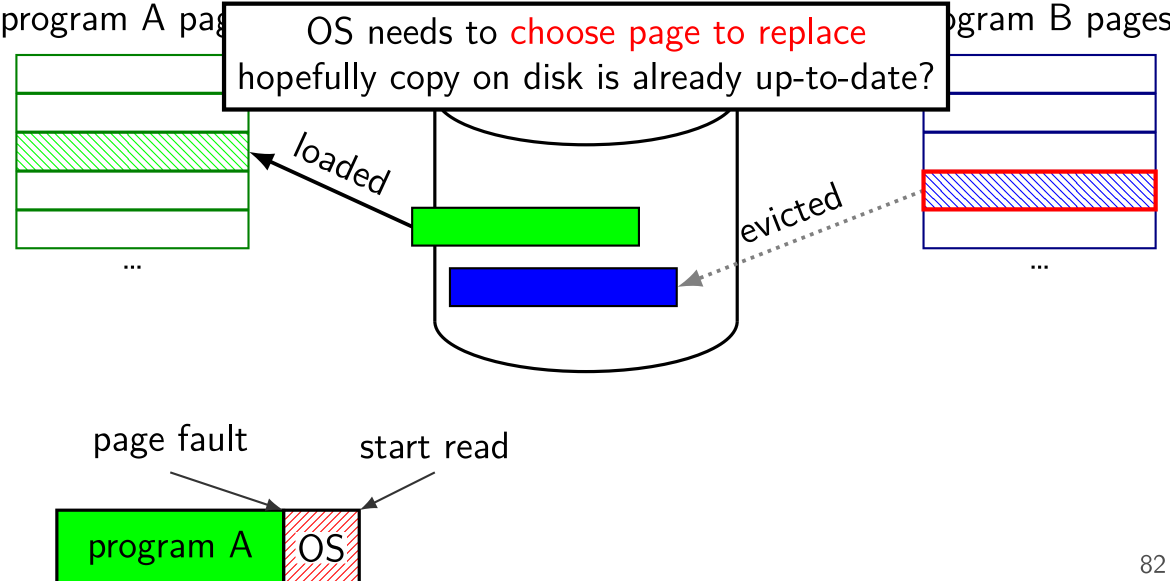
but where is the first directory?

backup slides

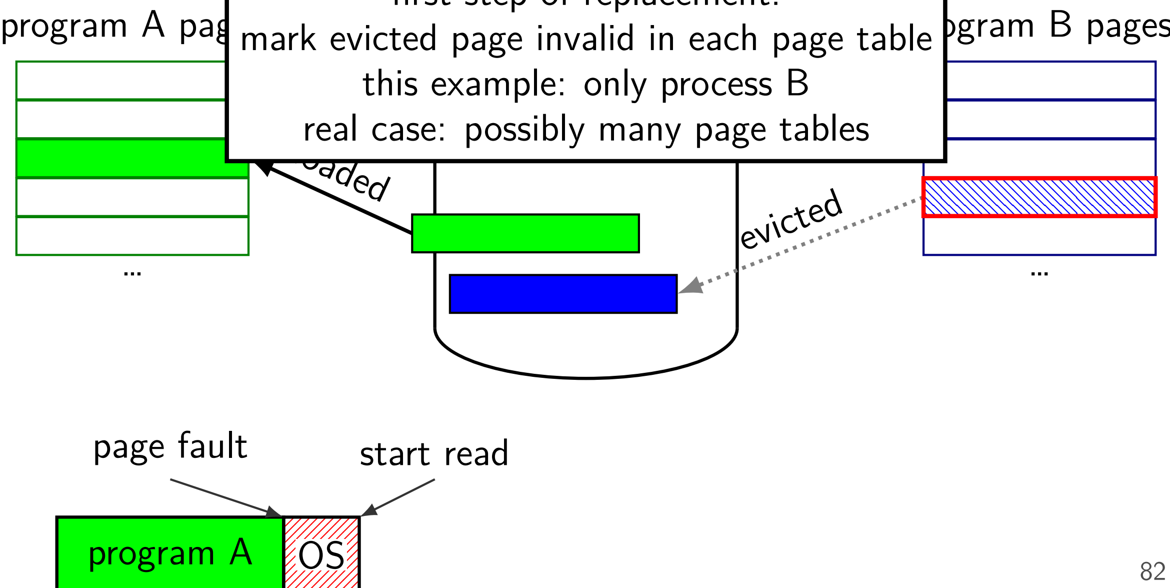
swapping timeline



swapping timeline



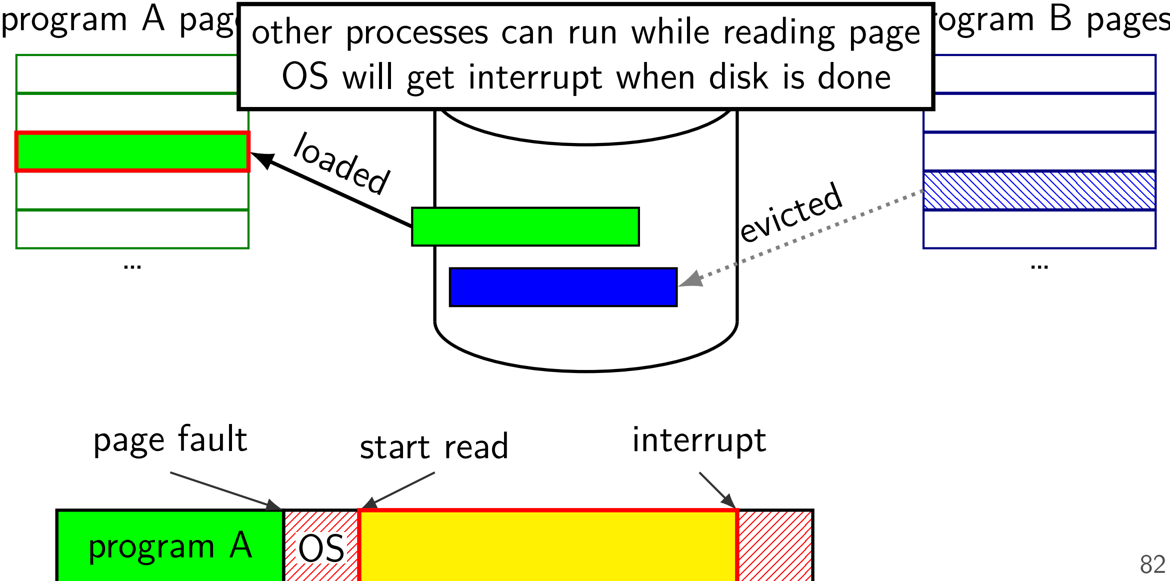
swapping timeline



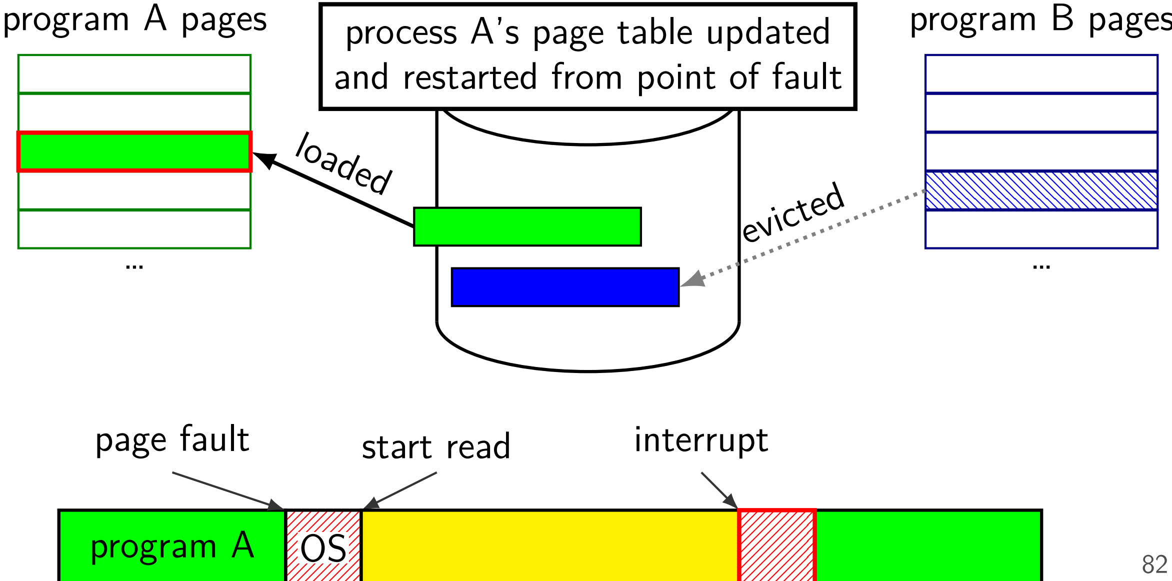
first step of replacement:
mark evicted page invalid in each page table
this example: only process B
real case: possibly many page tables

swapping timeline

other processes can run while reading page
OS will get interrupt when disk is done



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