I/O / Filesystems 1

last time

when LRU fails

special-case for single-access file data

readahead — handle scans by predicting reads

device driver halfs

- top: from system call, use buffer, request data, wait for data
- bottom: from interrupt, fill buffer, wake up

devices as magic memory

exercise

system is running two applications

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

timeline:

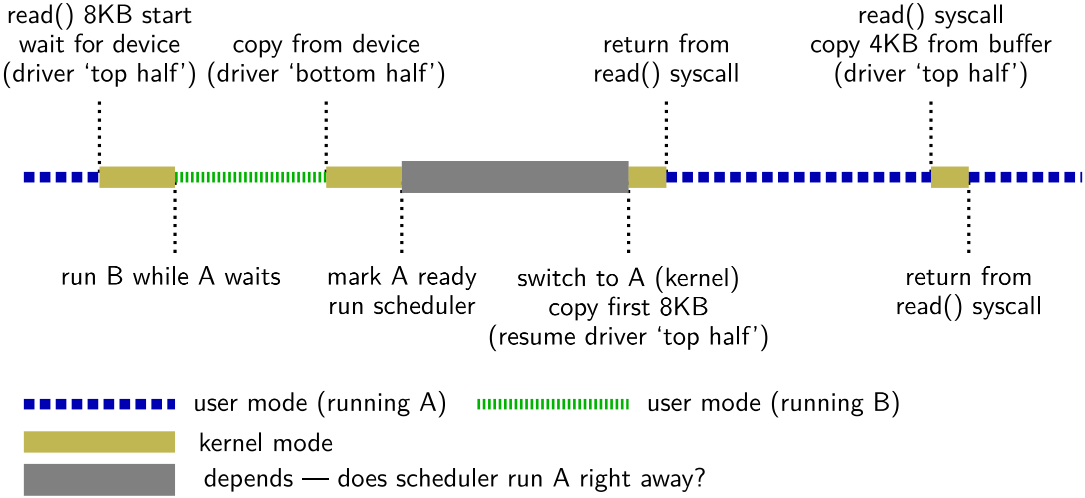
- A calls read() to 8KB of data from network
- 16KB of data comes in 10ms later
- A calls read() again to get 4KB more

exercise 1: how many kernel/user mode switches?

exercise 2: how many context switches?

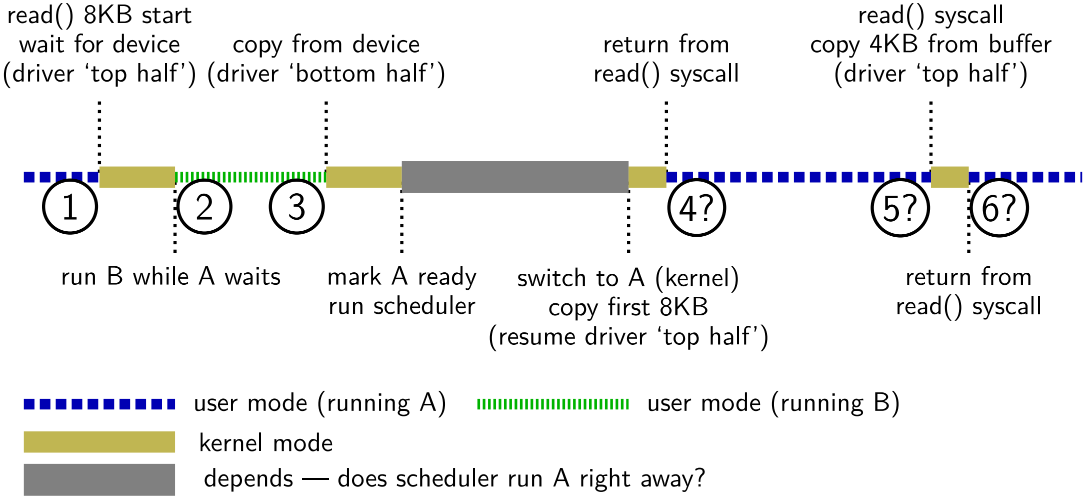
how many mode switches?

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



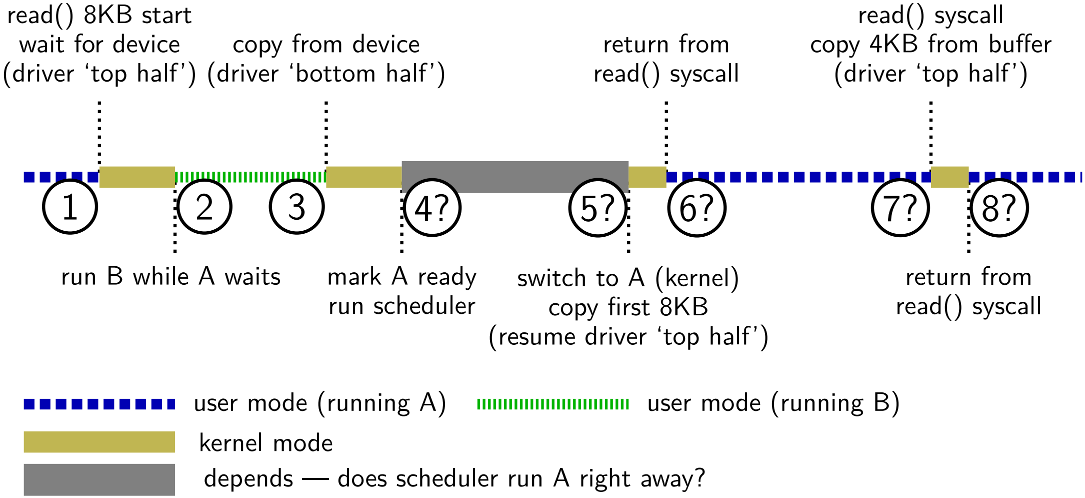
how many mode switches?

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



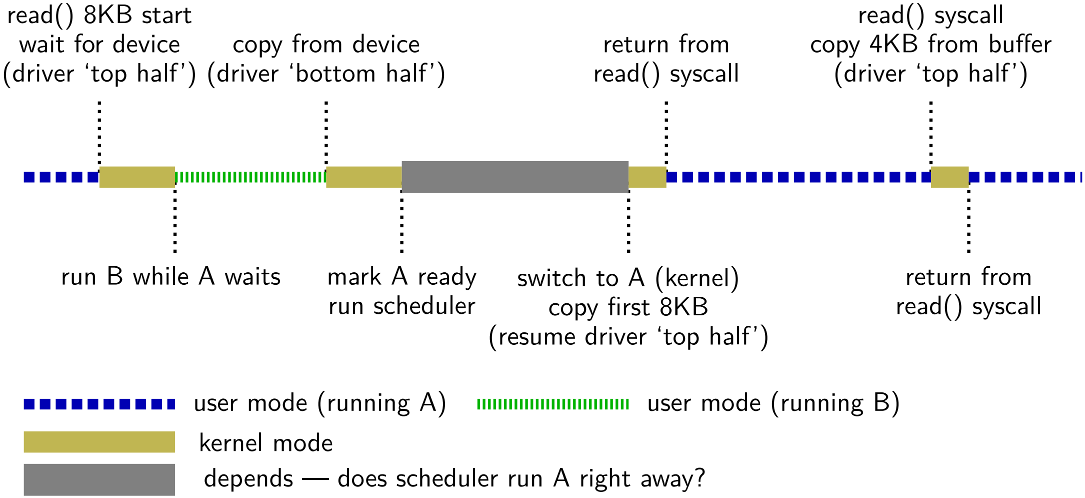
how many mode switches?

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



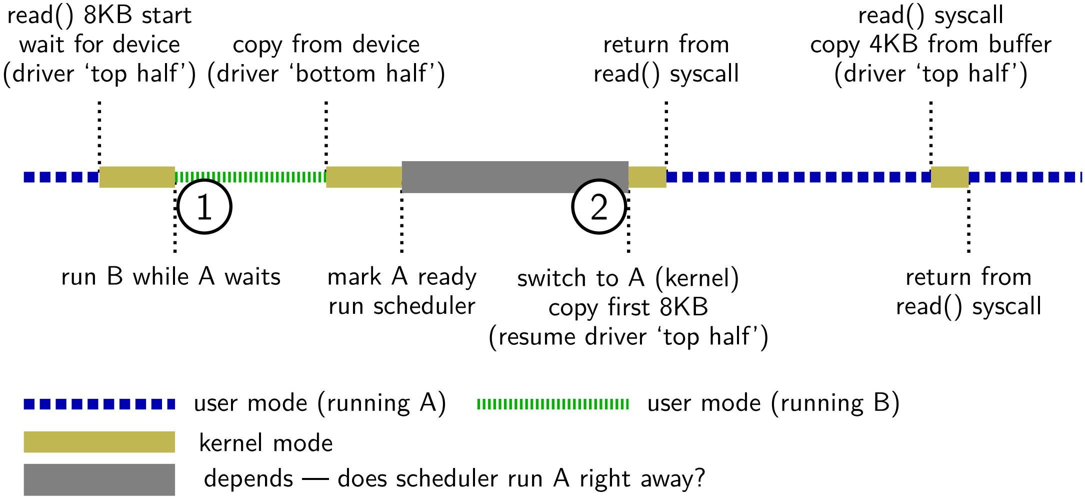
how many context switches?

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

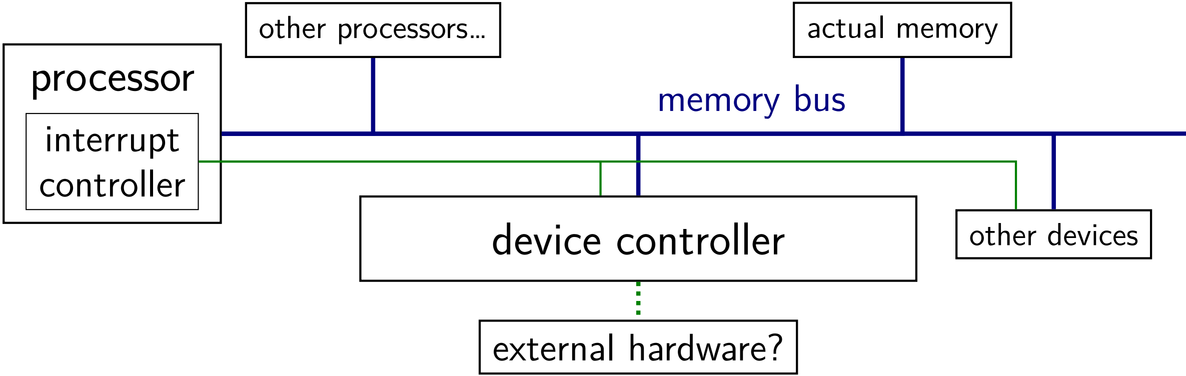


how many context switches?

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



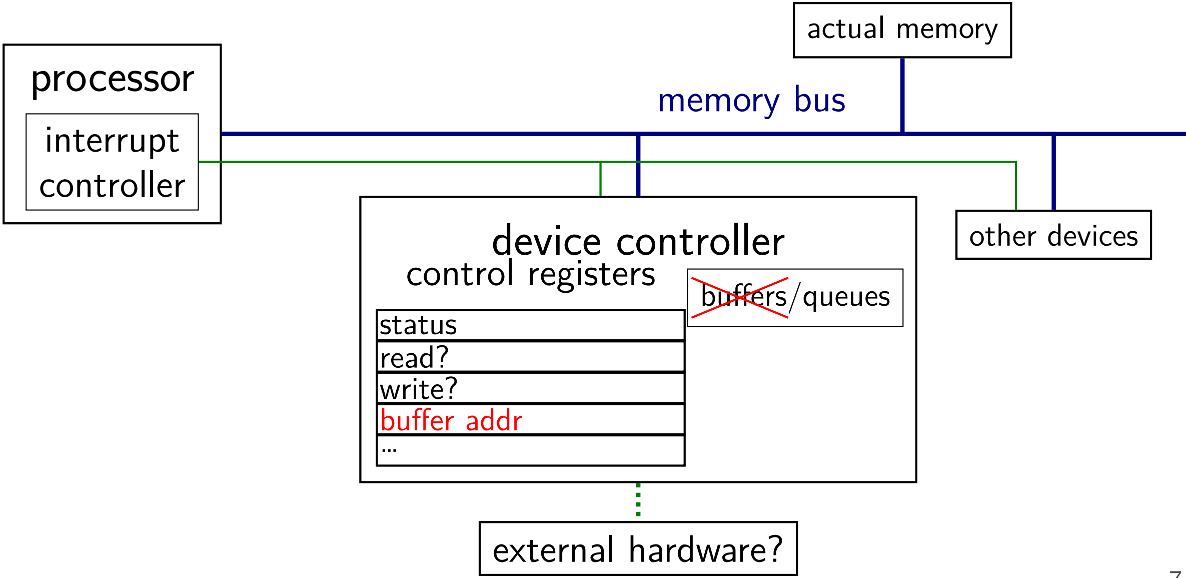
direct memory access (DMA)



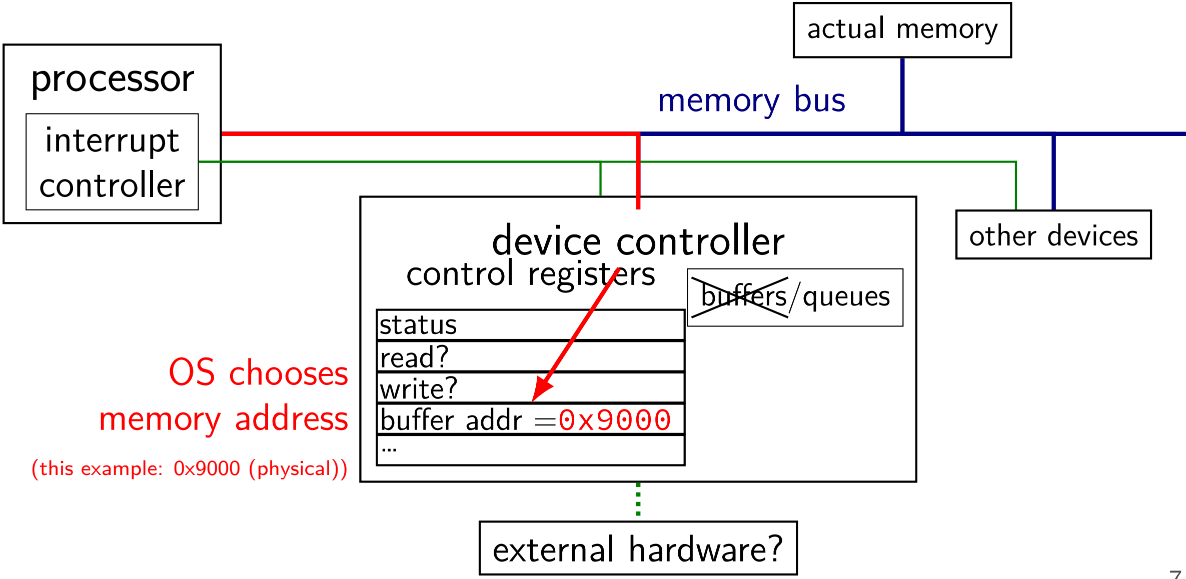
observation: devices can read/write memory

can have device copy data to/from memory

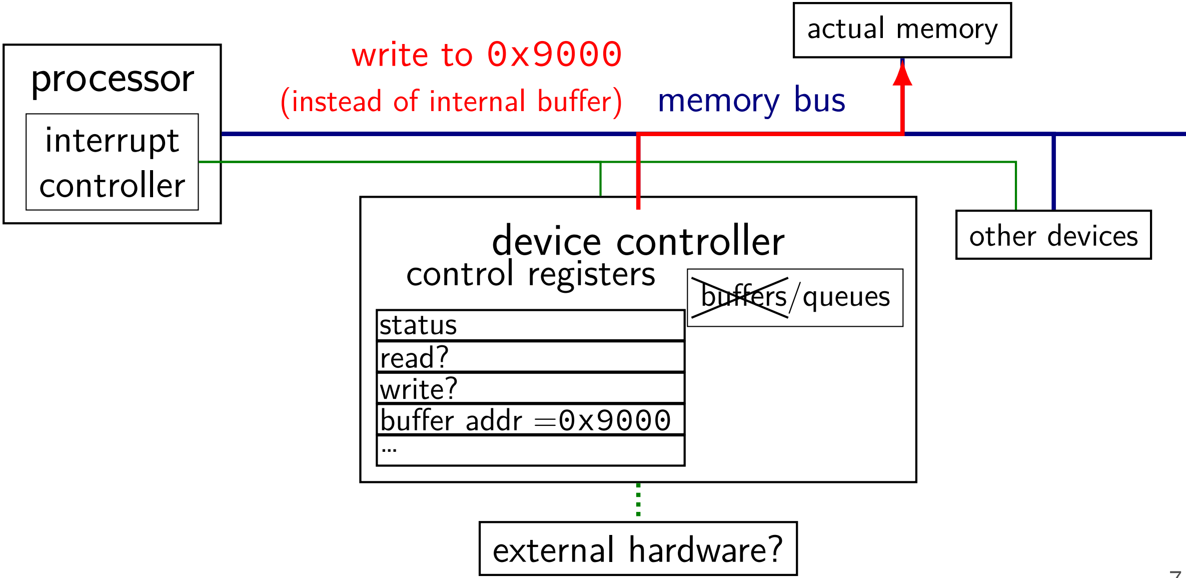
direct memory access (DMA)



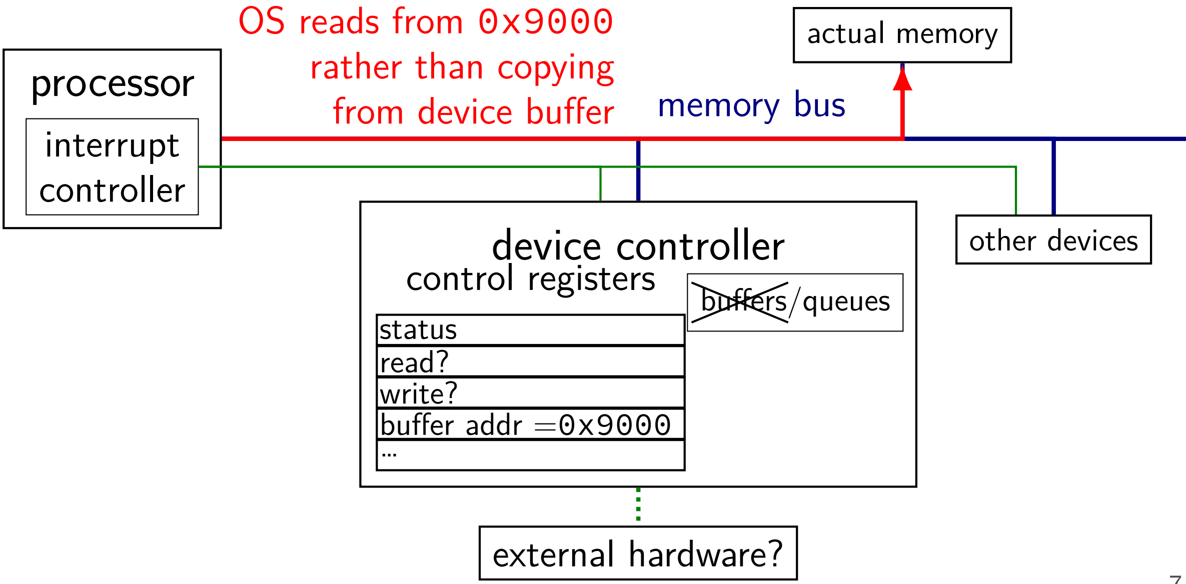
direct memory access (DMA)



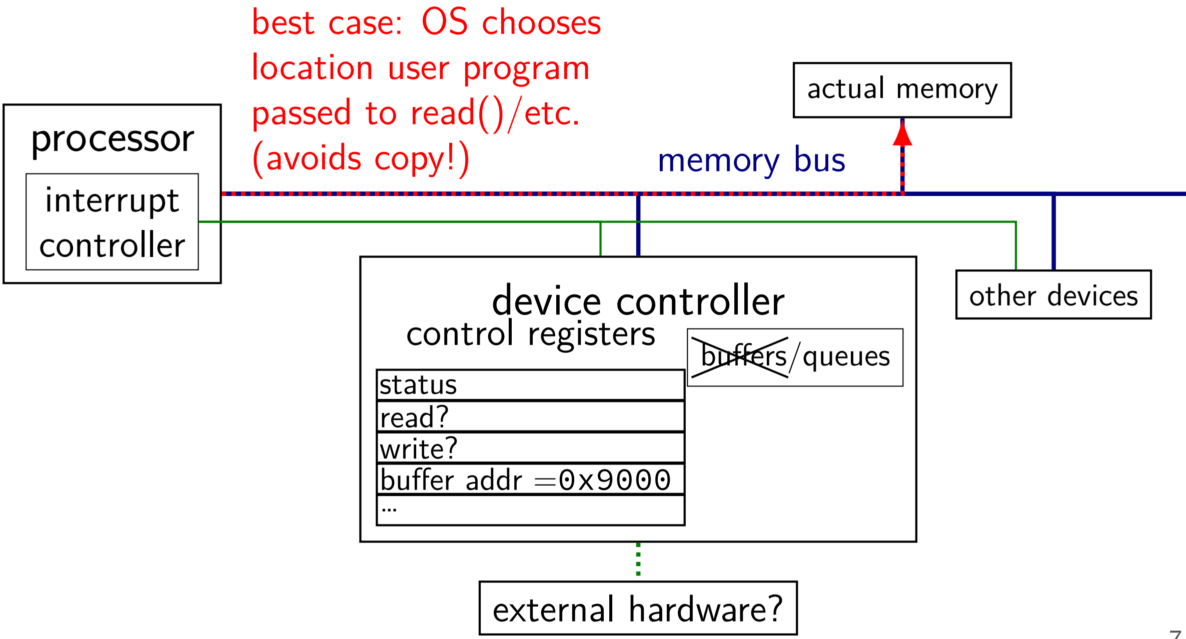
direct memory access (DMA)



direct memory access (DMA)



direct memory access (DMA)



direct memory access (DMA)

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

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

device uses memory as very large buffer space

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

direct memory access (DMA)

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

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

device uses memory as very large buffer space

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

OS puts data where it wants

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

OS puts data where it wants

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

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

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

OS puts data where it wants

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

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

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

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

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

devices summary

device *controllers* connected via memory bus

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

controller looks like "magic memory" to OS

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

two options for data transfer

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

the FAT filesystem

FAT: File Allocation Table

probably simplest widely used filesystem (family)

named for important data structure: *file allocation table*

FAT and sectors

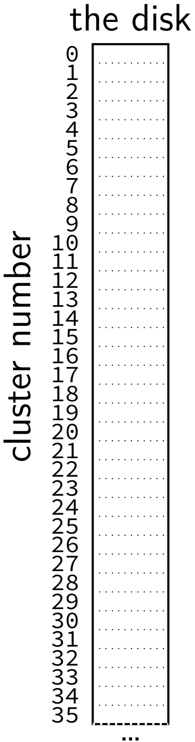
FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read

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

cluster: typically 512 to 4096 bytes



FAT and sectors

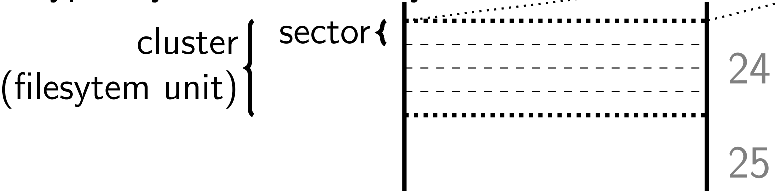
FAT divides disk into *clusters*

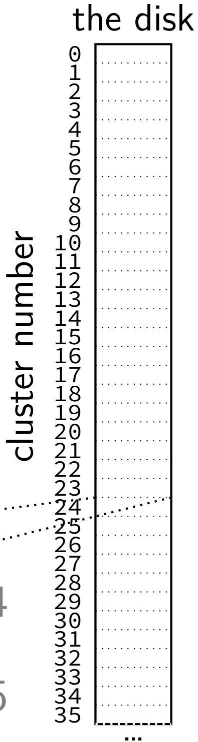
composed of one or more sectors

sector = minimum amount hardware can read

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

cluster: typically 512 to 4096 bytes



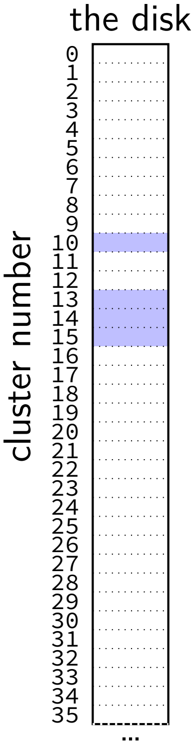


FAT: clusters and files

a file's data stored in a list of clusters

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

reading a file? need to find the list of clusters

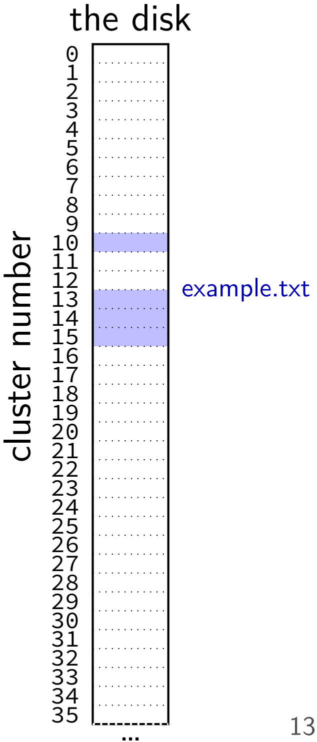


FAT: clusters and files

a file's data stored in a list of clusters

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

reading a file? need to find the list of clusters



FAT: the file allocation table

big array on disk, one entry per cluster

each entry contains a number — usually “next cluster”

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

FAT: reading a file (1)

get (from elsewhere) first cluster of data

linked list of cluster numbers

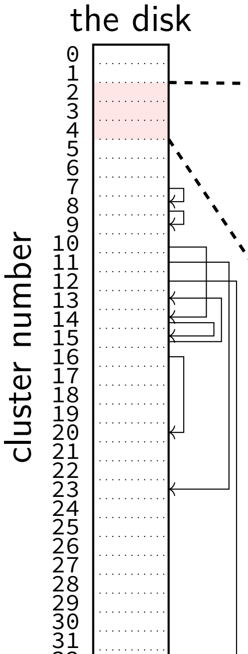
next pointers? file allocation table entry for cluster

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

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

file starting at cluster 10 contains data in:
cluster 10, then 14, then 15, then 13

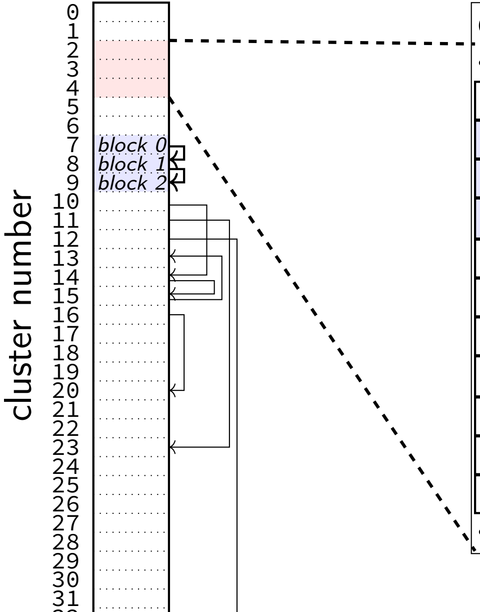
FAT: reading a file (2)



file allocation table

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

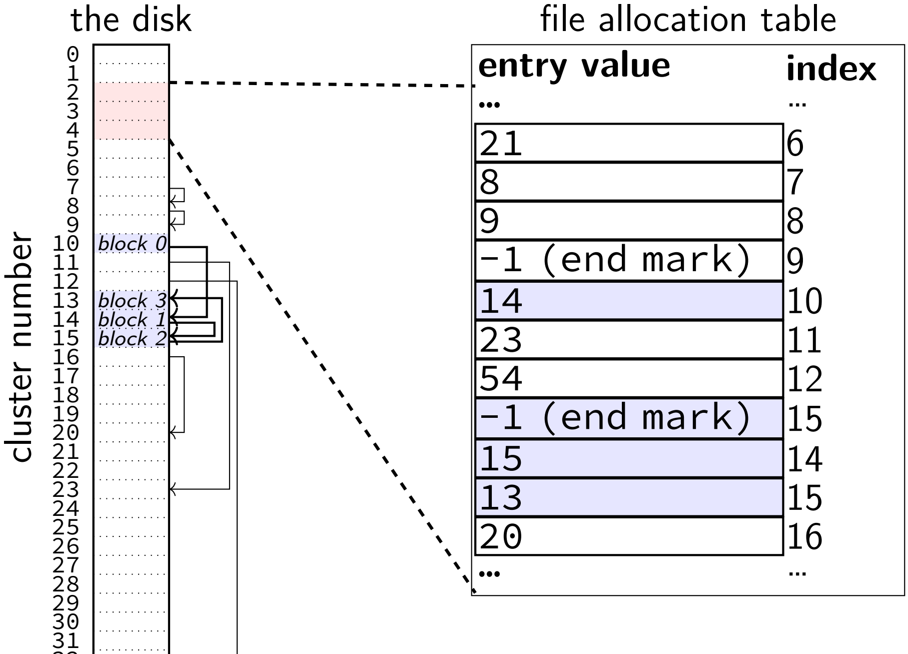
FAT: reading a file (2)



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?

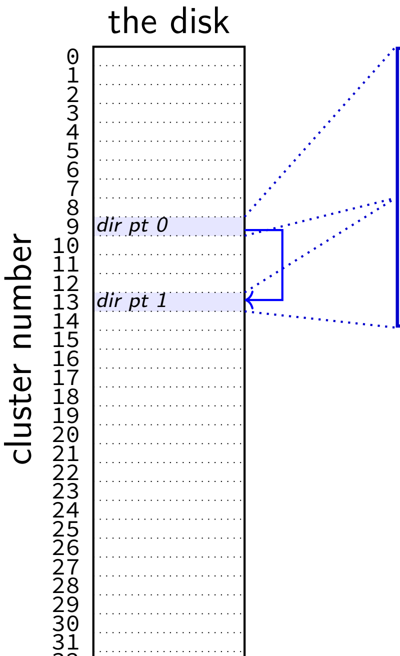
really want filenames

stored in directories!

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

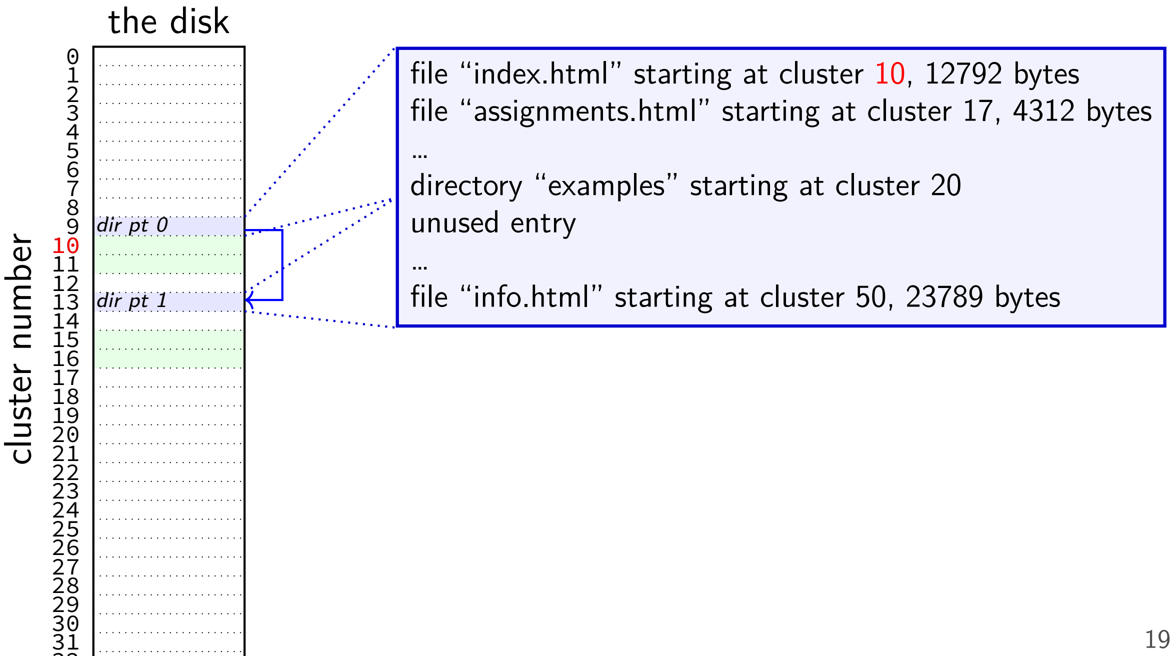
(name, starting location, other data about file)

finding files with directory

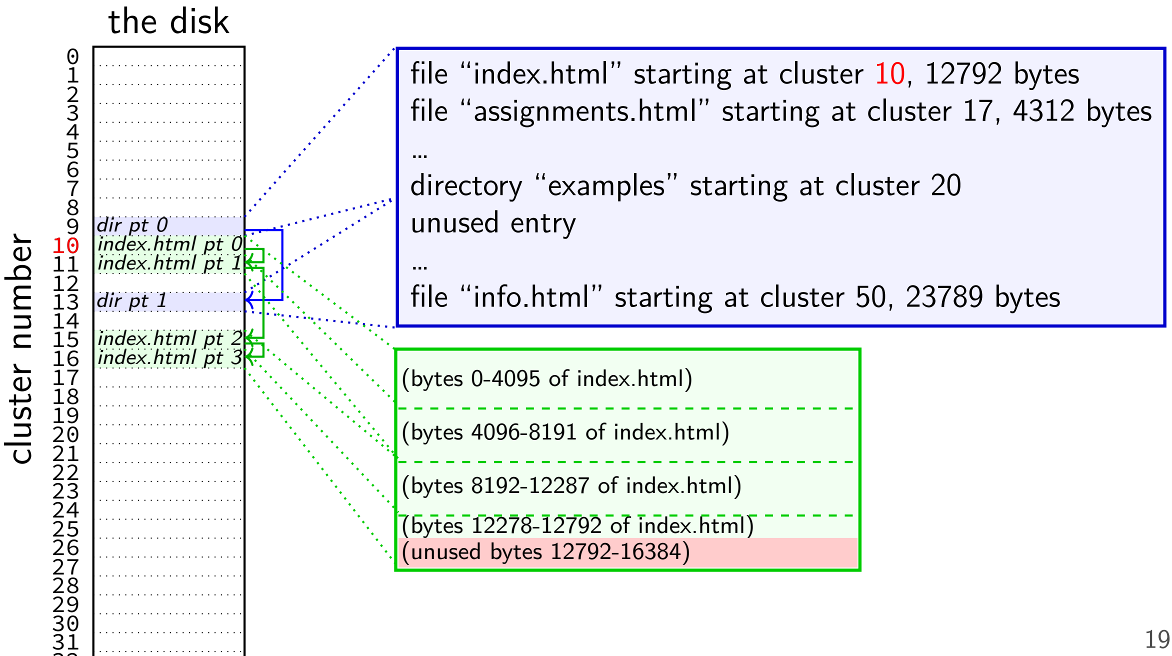


file “index.html” starting at cluster 10, 12792 bytes
file “assignments.html” starting at cluster 17, 4312 bytes
…
directory “examples” starting at cluster 20
unused entry
…
file “info.html” starting at cluster 50, 23789 bytes

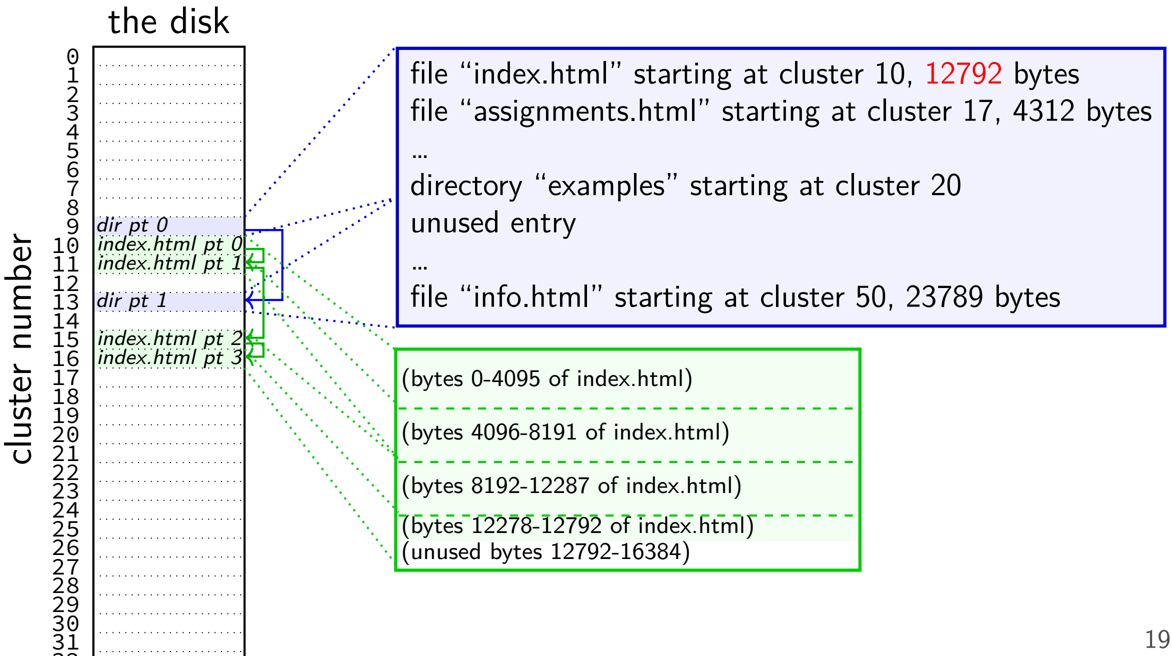
finding files with directory



finding files with directory



finding files with directory



FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?
…

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

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

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?
…

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

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

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

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?
…

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

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

8 character filename + 3 character extension
longer filenames? encoded using extra directory entries
(special `attrs` values to distinguish from normal entries)

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?
...

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

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

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

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?
…

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

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

attributes: is a subdirectory, read-only, …
also marks directory entries used to hold extra filename data

FAT directory entry

box = 1 byte

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

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

directory?
read-only?
hidden?
…

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

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

convention: if first character is 0x0 or 0xE5 — unused
0x00: for filling empty space at end of directory
0xE5: ‘hole’ — e.g. from file deletion

aside: FAT date encoding

seperate date and time fields (16 bits, little-endian integers)

bits 0-4: seconds (divided by 2), 5-10: minute, 11-15: hour

bits 0-4: day, 5-8: month, 9-15: year (minus 1980)

sometimes extra field for 100s(?) of a second

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Name[11];       // short name
  uint8_t DIR_Attr;           // File attribute
  uint8_t DIR_NTRes;          // set value to 0, never change t
  uint8_t DIR_CrtTimeTenth;   // millisecond timestamp for file
  uint16_t DIR_CrtTime;       // time file was created
  uint16_t DIR_CrtDate;       // date file was created
  uint16_t DIR_LstAccDate;    // last access date
  uint16_t DIR_FstClusHI;     // high word of this entry's firs
  uint16_t DIR_WrtTime;       // time of last write
  uint16_t DIR_WrtDate;       // dat eof last write
  uint16_t DIR_FstClusLO;     // low word of this entry's first
  uint32_t DIR_FileSize;      // file size in bytes
};
```

FAT directory entries (from C)

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

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

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Name[11
  uint8_t DIR_Attr;
  uint8_t DIR_NTRes;                                         ge t
  uint8_t DIR_CrtTime                                        file
  uint16_t DIR_CrtTi
  uint16_t DIR_CrtDate;          // date file was created
  uint16_t DIR_LstAccDate;       // last access date
  uint16_t DIR_FstClusHI;        // high word of this entry's firs
  uint16_t DIR_WrtTime;          // time of last write
  uint16_t DIR_WrtDate;          // dat eof last write
  uint16_t DIR_FstClusLO;        // low word of this entry's first
  uint32_t DIR_FileSize;         // file size in bytes
};
```

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

FAT directory entries (from C)

```
struct __attribute__((packed)) DirEntry {
  uint8_t DIR_Nam
  uint8_t DIR_Att
  uint8_t DIR_NTR                                       ge t
  uint8_t DIR_CrtTimeTenth;  // millisecond timestamp for file
  uint16_t DIR_CrtTime;      // time file was created
  uint16_t DIR_CrtDate;      // date file was created
  uint16_t DIR_LstAccDate;   // last access date
  uint16_t DIR_FstClusHI;    // high word of this entry's firs
  uint16_t DIR_WrtTime;      // time of last write
  uint16_t DIR_WrtDate;      // dat eof last write
  uint16_t DIR_FstClusLO;    // low word of this entry's first
  uint32_t DIR_FileSize;     // file size in bytes
};
```

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

nested directories

foo/bar/baz/file.txt

read root directory entries to find foo

read foo's directory entries to find bar

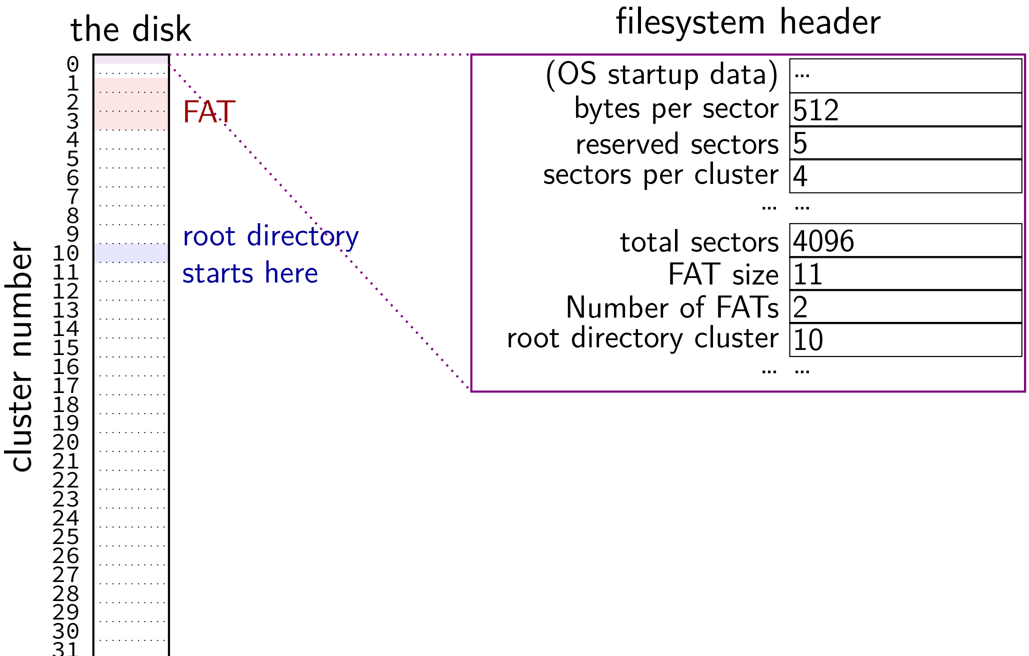
read bar's directory entries to find baz

read baz's directory entries to find file.txt

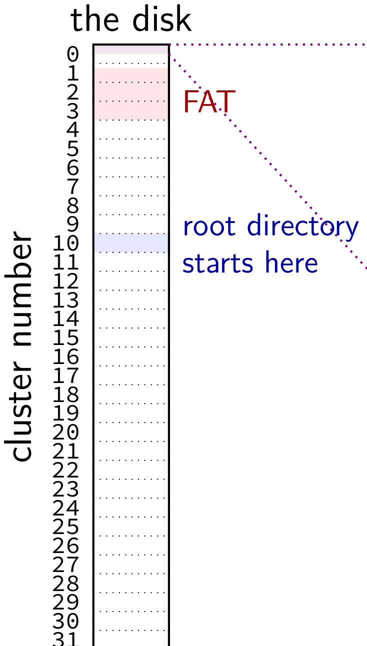
the root directory?

but where is the first directory?

FAT disk header



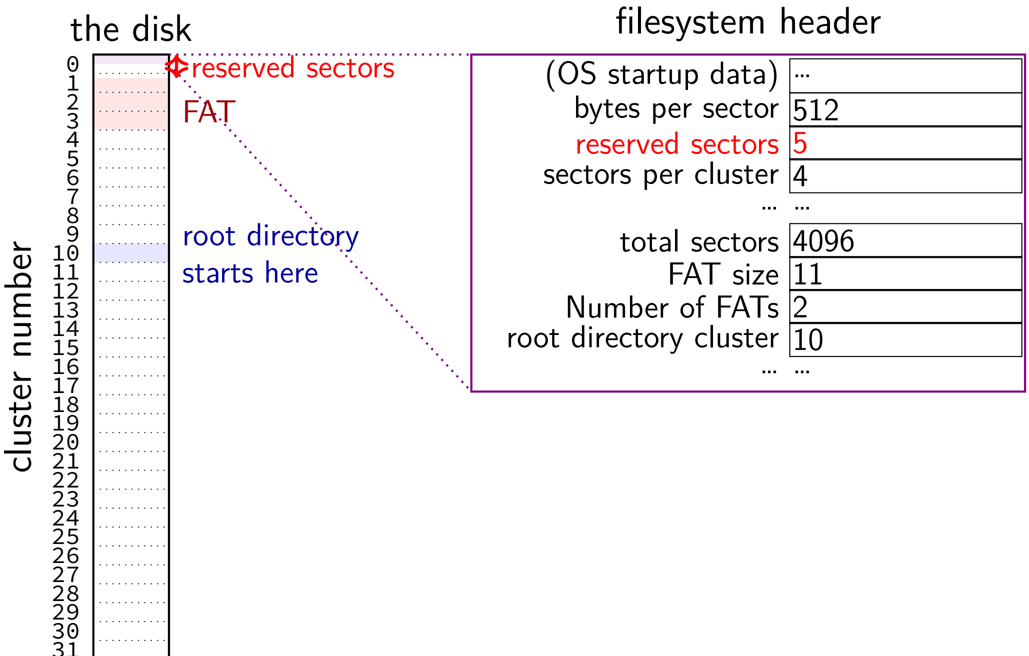
FAT disk header



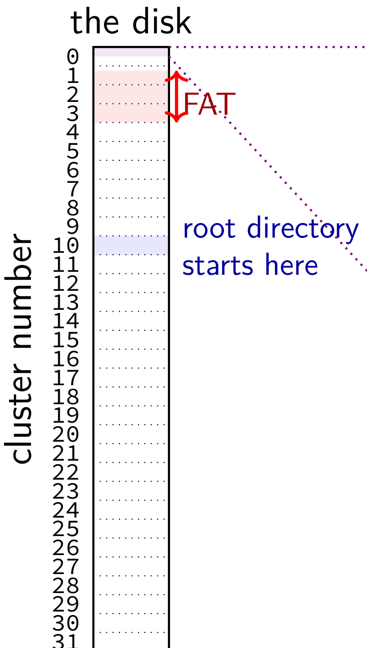
filesystem header

(OS startup data)	...
bytes per sector	512
reserved sectors	5
sectors per cluster	4
...	...
total sectors	4096
FAT size	11
Number of FATs	2
root directory cluster	10
...	...

FAT disk header



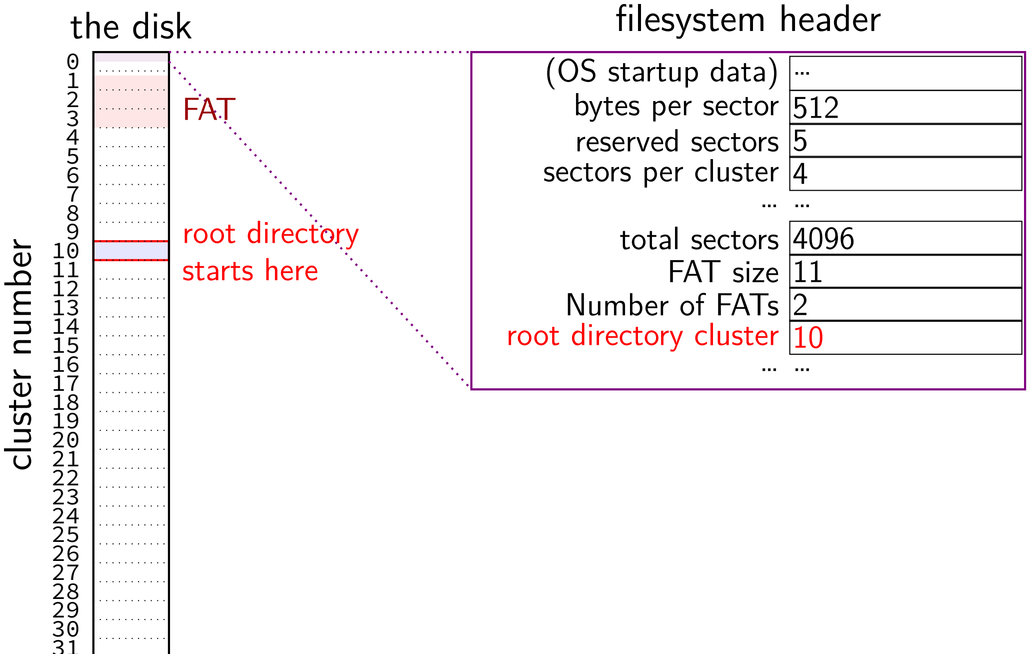
FAT disk header



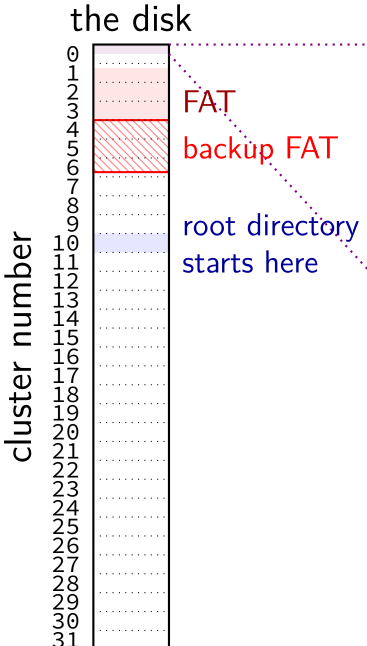
filesystem header

(OS startup data)	...
bytes per sector	512
reserved sectors	5
sectors per cluster	4
...	...
total sectors	4096
FAT size	11
Number of FATs	2
root directory cluster	10
...	...

FAT disk header



FAT disk header



filesystem header

(OS startup data)	...
bytes per sector	512
reserved sectors	5
sectors per cluster	4
...	...
total sectors	4096
FAT size	11
Number of FATs	2
root directory cluster	10
...	...

filesystem header

fixed location near beginning of disk

determines size of clusters, etc.

tells where to find FAT, root directory, etc.

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_jmpBoot[3];     // jmp instr to boot code
  uint8_t BS_oemName[8];     // indicates what system formatted this
  uint16_t BPB_BytsPerSec;   // count of bytes per sector
  uint8_t BPB_SecPerClus;    // no.of sectors per allocation unit
  uint16_t BPB_RsvdSecCnt;   // no.of reserved sectors in the reserve
  uint8_t BPB_NumFATs;       // count of FAT datastructures on the vo
  uint16_t BPB_rootEntCnt;   // count of 32-byte entries in root dir,
  uint16_t BPB_totSec16;     // total sectors on the volume
  uint8_t BPB_media;         // value of fixed media
....
  uint16_t BPB_ExtFlags;     // flags indicating which FATs are activ
```

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_
  uint8_t BS_                                                    this
  uint16_t BPB_BytsPerSec;   // count of bytes per sector
  uint8_t BPB_SecPerClus;    // no.of sectors per allocation unit
  uint16_t BPB_RsvdSecCnt;   // no.of reserved sectors in the reserve
  uint8_t BPB_NumFATs;       // count of FAT datastructures on the vo
  uint16_t BPB_rootEntCnt;   // count of 32-byte entries in root dir,
  uint16_t BPB_totSec16;     // total sectors on the volume
  uint8_t BPB_media;         // value of fixed media
....
  uint16_t BPB_ExtFlags;     // flags indicating which FATs are activ
```

size of sector (in bytes) and size of cluster (in sectors)

FAT header (C)

```
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_jmpBoot[3];      // jmp instr to boot code
  uint8_t BS_oemName[8];      // indicates what system formatted this
  uint16_t BPB_BytsPerSec;    // count of bytes per sector
  uint8_t BPB_SecPerClus;     // no                               t
  uint16_t BPB_RsvdSecCnt;    // no.of reserved sectors in the reserve
  uint8_t BPB_NumFATs;        // count of FAT datastructures on the vo
  uint16_t BPB_rootEntCnt;    // count of 32-byte entries in root dir,
  uint16_t BPB_totSec16;      // total sectors on the volume
  uint8_t BPB_media;          // value of fixed media
....
  uint16_t BPB_ExtFlags;      // flags indicating which FATs are activ
```

space before file allocation table

FAT header (C)

```c
struct __attribute__((packed)) Fat32BPB {
  uint8_t BS_jmpBoot[3];     // jmp instr to boot code
  uint8_t BS_oemName[8];     // indicates what system formatted this
  uint16_t BPB_BytsPerSec;   // count of bytes per sector
  uint8_t BPB_SecPerClus;                                       t
  uint16_t BPB_RsvdSecCnt;                                      serve
  uint8_t BPB_NumFATs;                                          he vo
  uint16_t BPB_rootEntCnt;                                       dir,
  uint16_t BPB_totSec16;     // total sectors on the volume
  uint8_t BPB_media;         // value of fixed media
....
  uint16_t BPB_ExtFlags;     // flags indicating which FATs are acti
```

number of copies of file allocation table
extra copies in case disk is damaged
typically two with writes made to both

FAT: creating a file

add a directory entry

choose clusters to store file data (how???)

update FAT to link clusters together

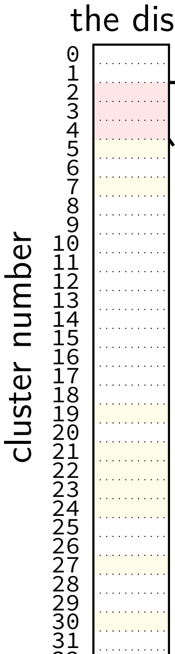
FAT: creating a file

add a directory entry

choose clusters to store file data (how???)

update FAT to link clusters together

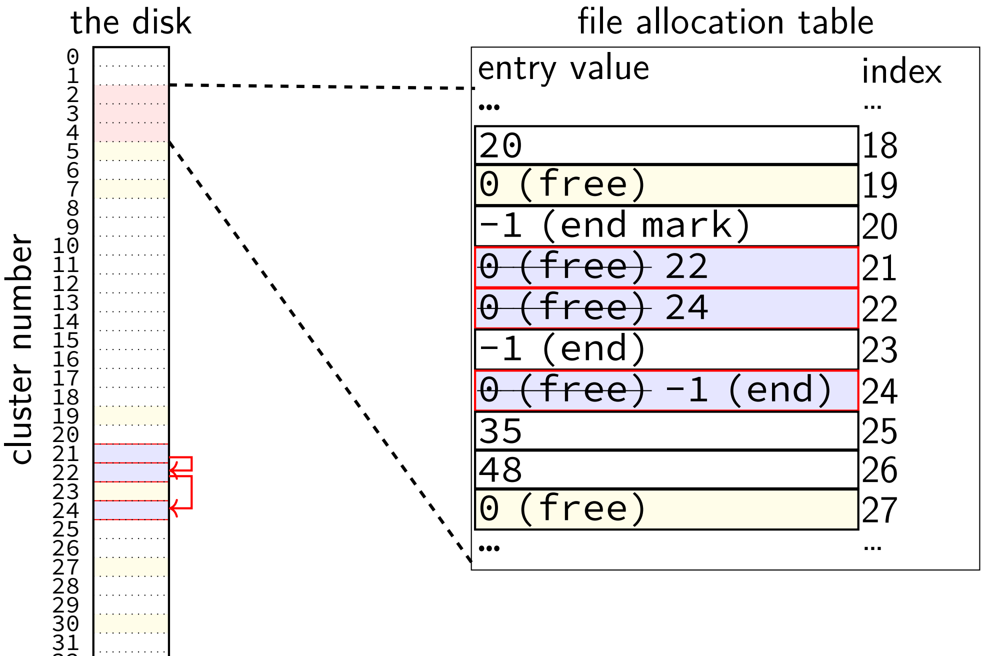
FAT: free clusters



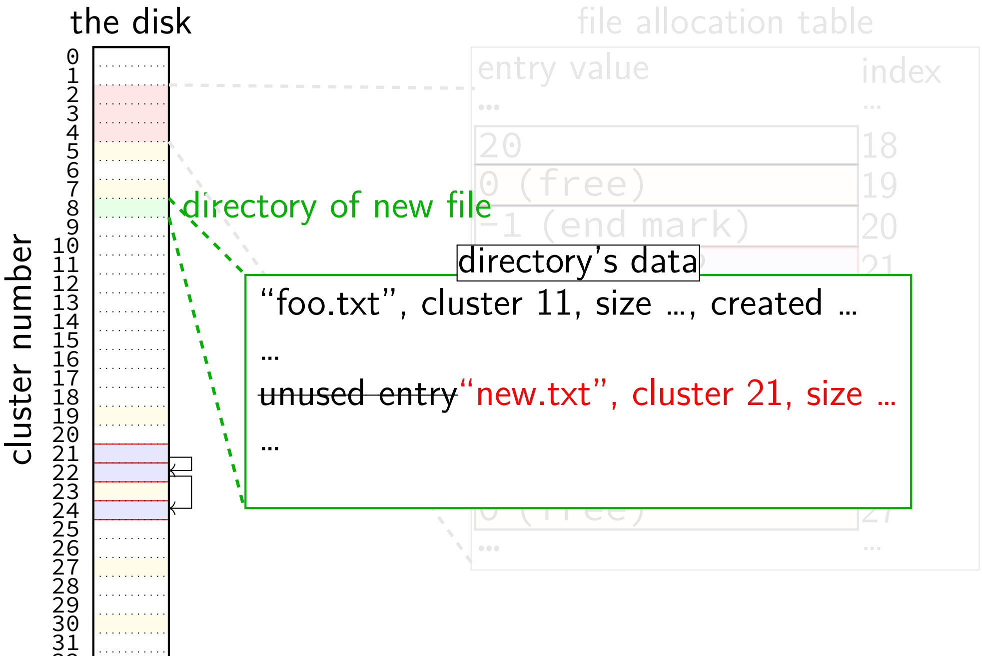
file allocation table

entry value	index
...	...
20	18
0 (free)	19
-1 (end mark)	20
0 (free)	21
0 (free)	22
-1 (end)	23
0 (free)	24
35	25
48	26
0 (free)	27
...	...

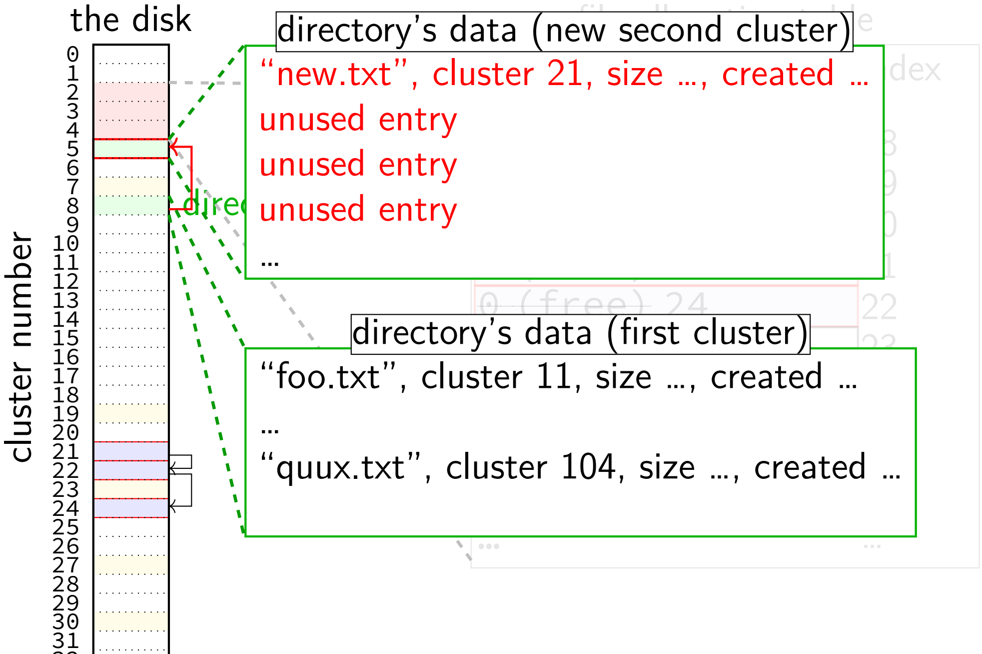
FAT: writing file data



FAT: replacing unused directory entry



FAT: extending directory



FAT: exercise

C.txt is file in directory B which is in directory A

consider the following items on disk:

[a] FAT entries for A
[b] FAT entries for B
[c] FAT entries for C.txt
[d] data clusters for A
[e] data clusters for B
[f] data clusters for C.txt

Ignoring modification timestamp updates,
which of the above **may** be modified to:

1) assuming directores existed previously, create C.txt
2) truncate C.txt, making it have size 0 bytes (assume prev. not empty)
3) move C.txt from directory B into directory A

FAT: deleting files

reset FAT entries for file clusters to free (0)

write "unused" character in filename for directory entry

- maybe rewrite directory if that'll save space?

exercise

say FAT filesystem with:

- 4-byte FAT entries
- 32-byte directory entries
- 2048-byte clusters

how many FAT entries+clusters (outside of the FAT) is used to store a directory of 200 30KB files?

- count clusters for both directory entries and the file data

how many FAT entries+clusters is used to store a directory of 2000 3KB files?

FAT pros and cons?

backup slides

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

disk scheduling

schedule I/O to the disk

schedule = decide what read/write to do next

by OS: what to request from disk next?

by controller: which OS request to do next?

typical goals:

minimize seek time

don't starve requiests

disk scheduling

schedule I/O to the disk

schedule = decide what read/write to do next

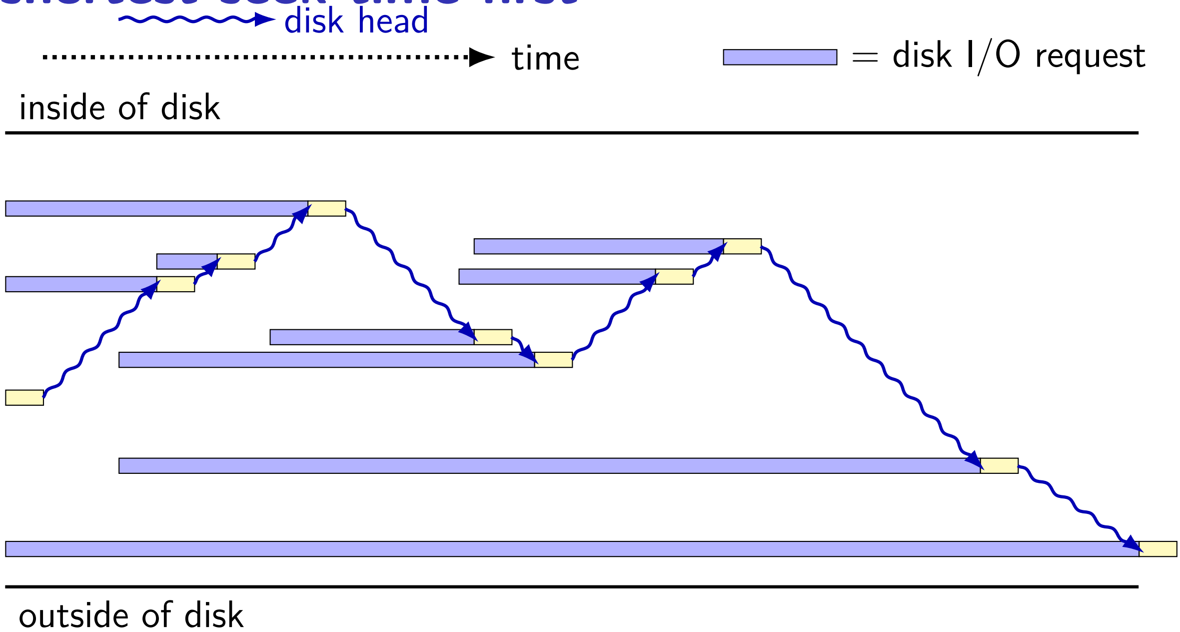
- by OS: what to request from disk next?
- by controller: which OS request to do next?

typical goals:

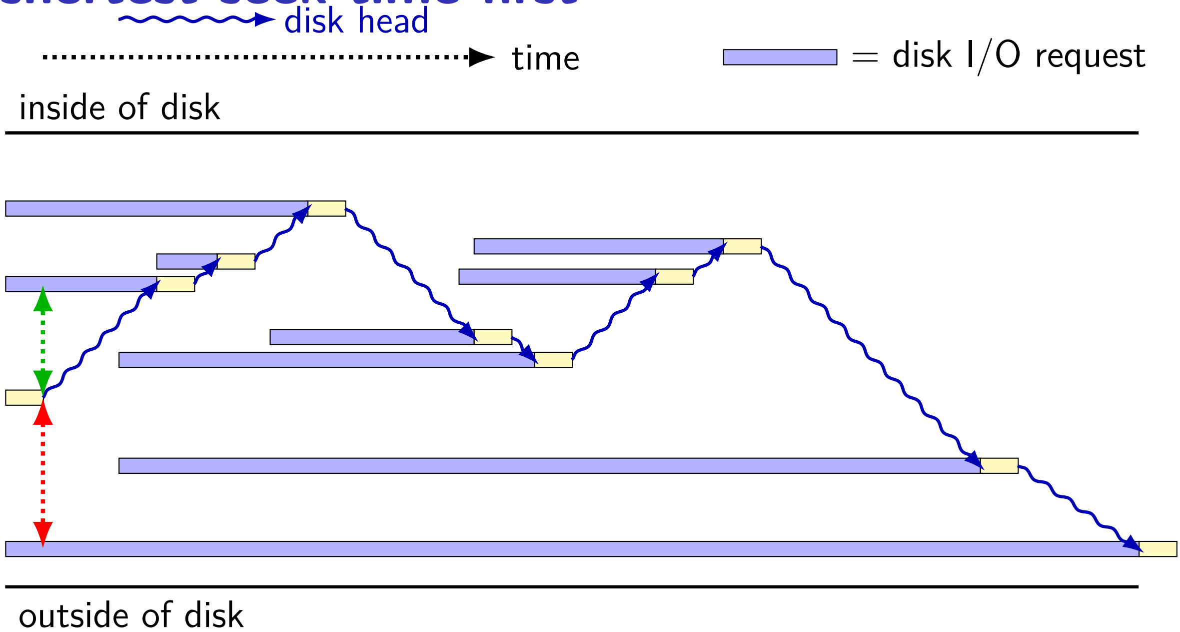
minimize seek time

don't starve requiests

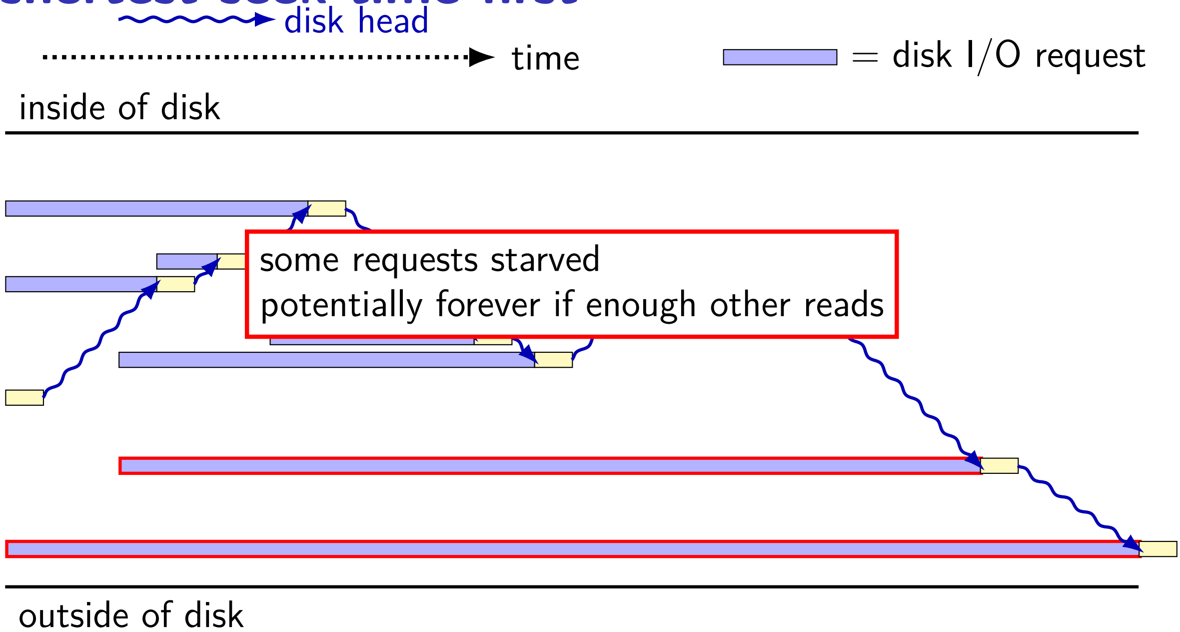
shortest seek time first



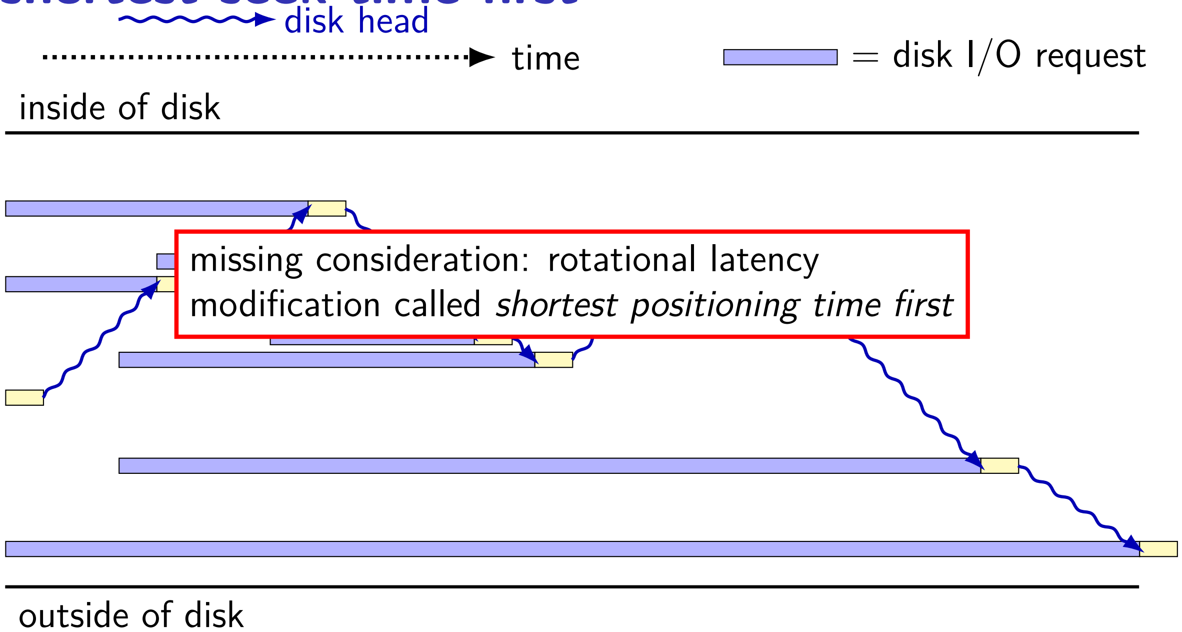
shortest seek time first



shortest seek time first



shortest seek time first



missing consideration: rotational latency
modification called *shortest positioning time first*

disk scheduling

schedule I/O to the disk

schedule = decide what read/write to do next

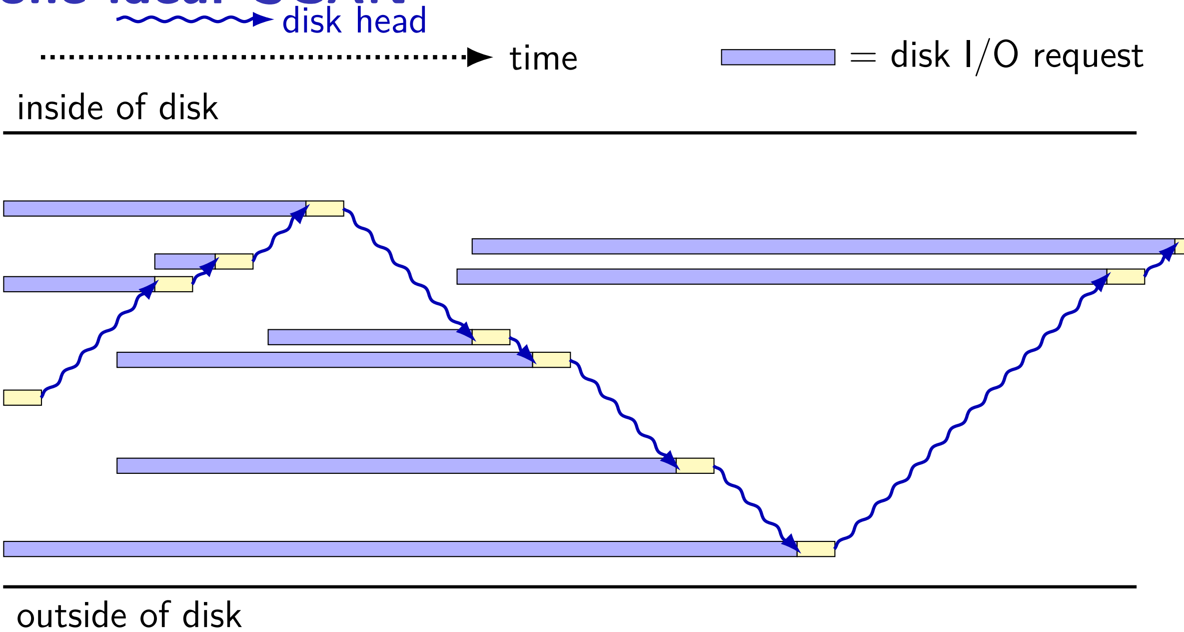
- by OS: what to request from disk next?
- by controller: which OS request to do next?

typical goals:

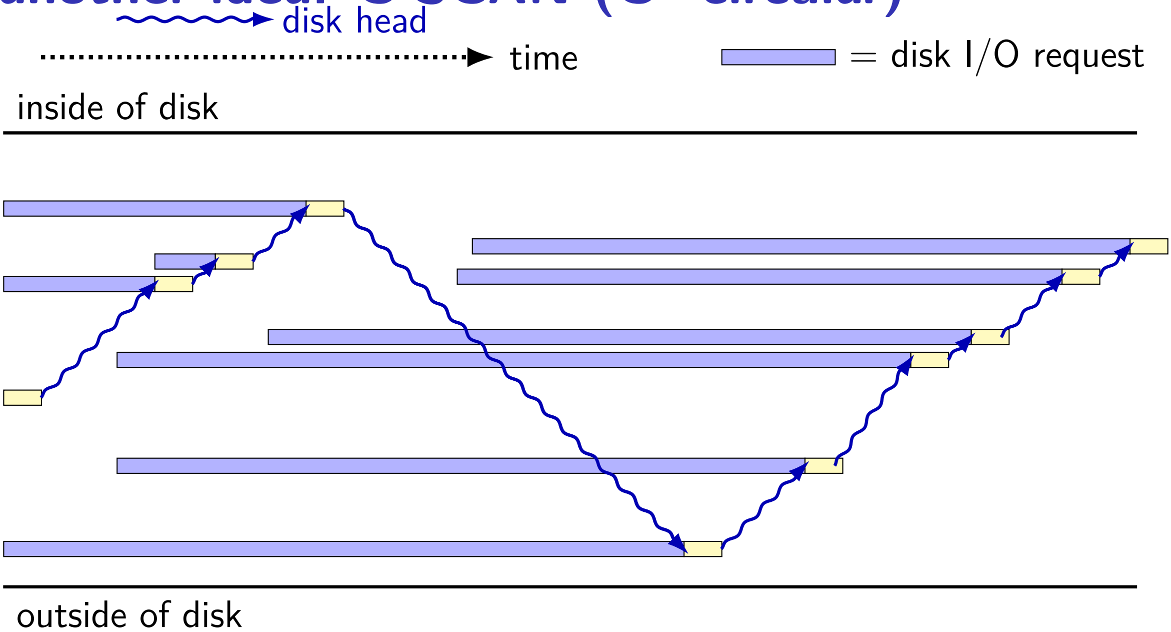
minimize seek time

don't starve requiests

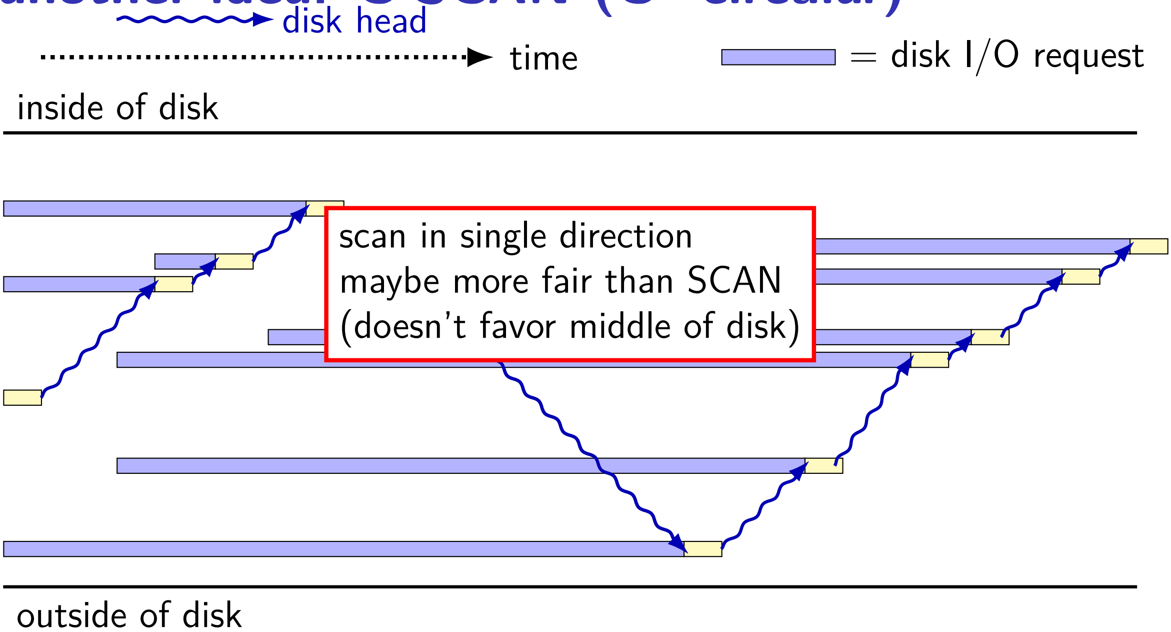
one idea: SCAN



another idea: C-SCAN (C=circular)

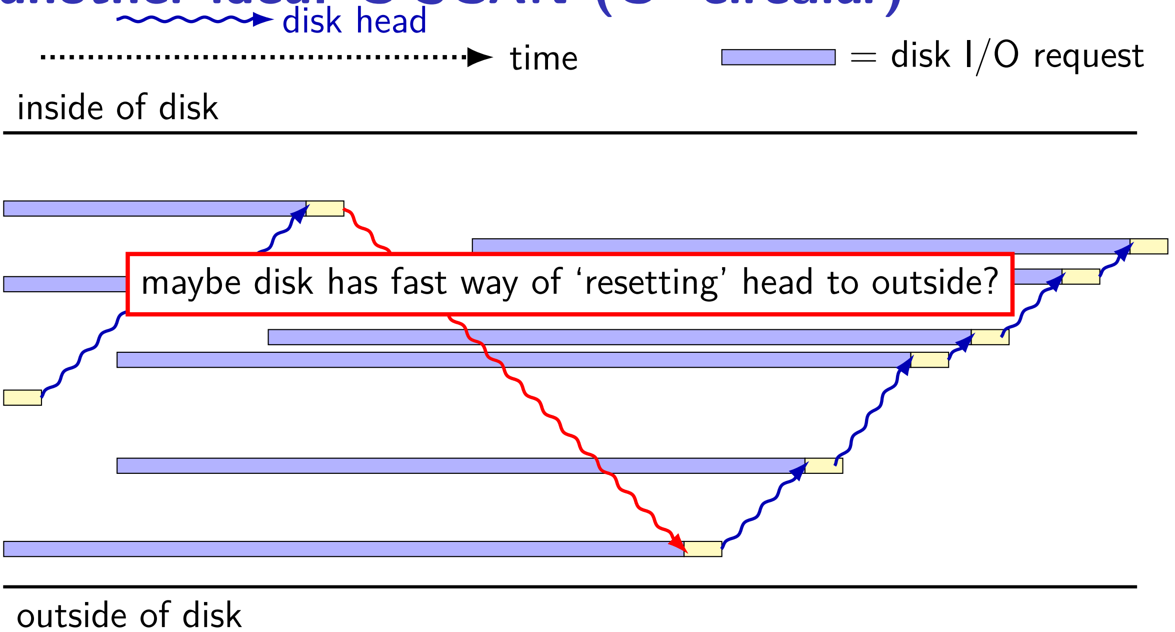


another idea: C-SCAN (C=circular)



scan in single direction
maybe more fair than SCAN
(doesn't favor middle of disk)

another idea: C-SCAN (C=circular)



some disk scheduling algorithms (text)

SSTF: take request with shortest seek time next

- subject to starvation — stuck on one side of disk
- could also take into account rotational latency — yields SPTF
  - shortest positioning time first

SCAN/*elevator*: move disk head towards center, then away

- let requests pile up between passes
- limits starvation; good overall throughput

C-SCAN: take next request closer to center of disk (if any)

- variant of scan that moves head in one direction
- avoids bias towards center of disk