Operating Systems: I/O

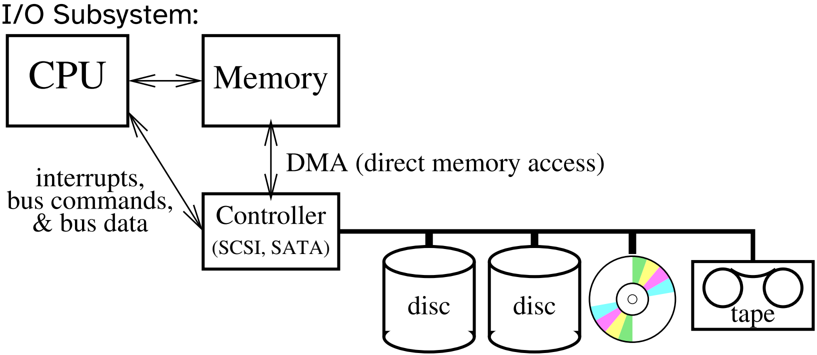
CIS*3110: Operating Systems

Dr. A. Hamilton-Wright

School of Computer Science
University of Guelph

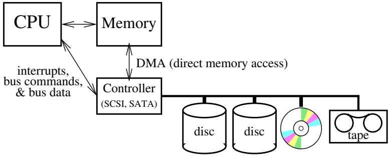
2024-12-01

I/O, I/O it's off to work we go . . .

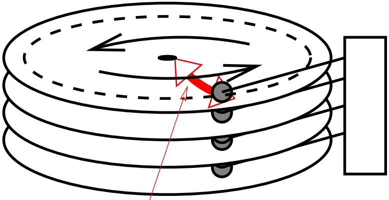


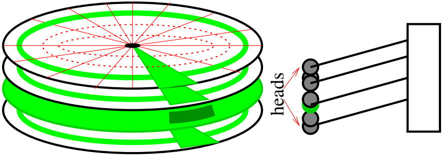
- DMA $\rightarrow$ block device $\rightsquigarrow$ block device driver
- data transfer using data bus $\rightarrow$ character device

I/O Subsystem:



Disk





A disk address is three-dimensional (θ, r, z); data is stored:

in sectors	θ
within cylinders	r
on tracks (surfaces, heads)	z

- **seeking** selects a cylinder
- **rotation** provides access to all sectors

Disk Parameters

Example:
Seagate Cheeta 10K.6 (SCSI bus)

10,000 RPM
→ 6 ms/revolution
→ 3 ms average latency

4 physical platters (discs)
8 heads
49,854 cylinders
772 sectors/track
(@ 512 bytes/sector)
5 ms avg seek time

$8 \times 49,854 \times 772$
$= 307,898,304$sectors @ 1/2k each
$= 153,949,152\text{k} \approx 146.8\text{Gb}$

Data Rate:
772×512 = 395, 264 bytes/track
$\therefore 395,264/6\text{ms}$
= 65,877 bytes per ms
"disk-to-buffer" speed

"buffer-to-computer" speed that of controller/memory bus – currently
~2.5Gbit/s (SCSI-Ultra320)
~3Gbit/s (SATA)

772×8 = 6,6176 sectors/cylinder

Minimize Seek Distance

$\mathcal{Q}$: Which is better?

FIFO (request order)

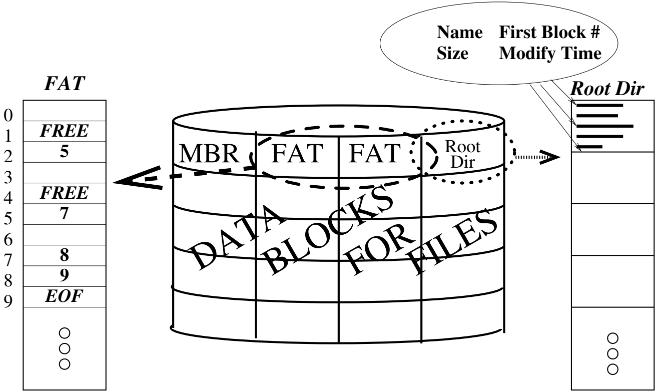
SJF do the request which is in the closest cylinder

SCAN (elevator algorithm) shortest seek distance in the direction you are travelling

$\mathcal{Q}$: What size should the blocks be?

- larger blocks – more waste at EOF, larger payoff when block is found
- smaller blocks – less EOF waste, but likely more indexing data

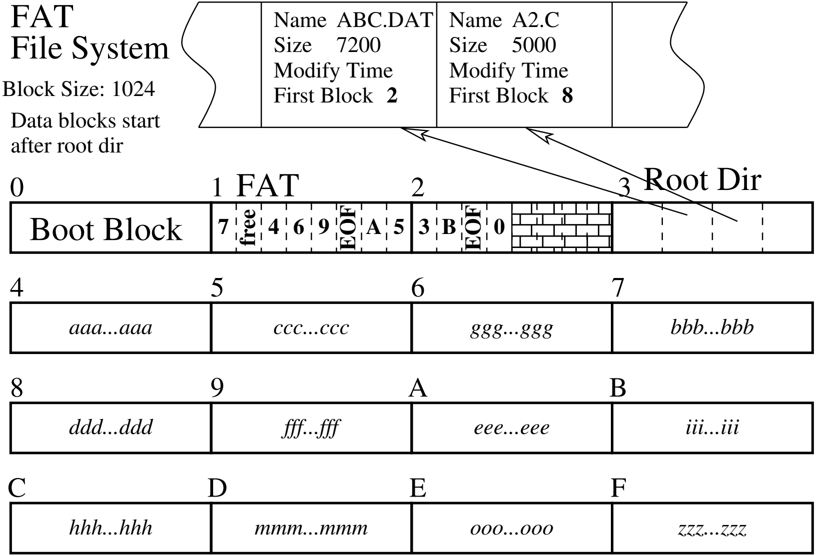
FAT (File Allocation Table) based filesystems

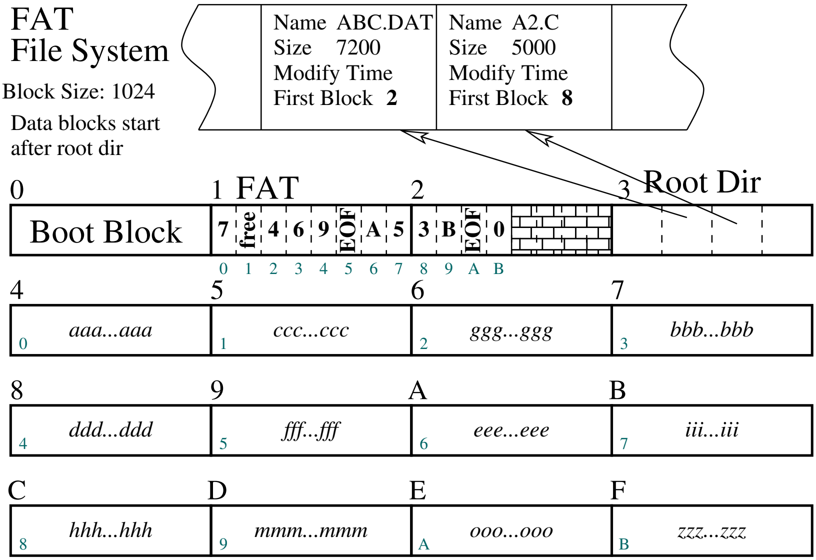


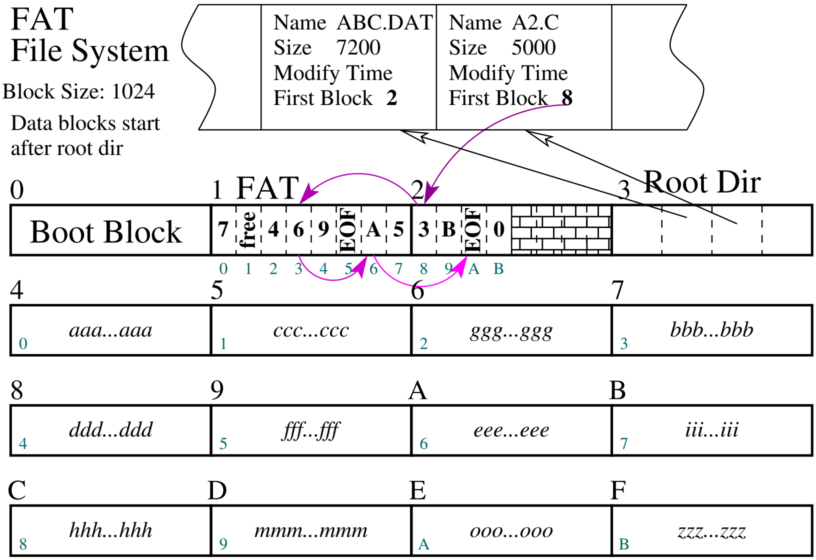
FAT File System

Block Size: 1024

Data blocks start after root dir



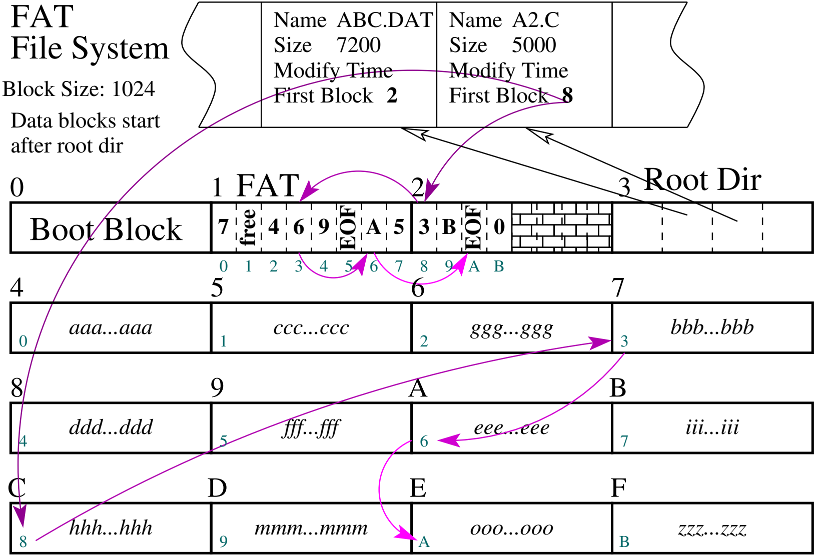

FAT
File System
Block Size: 1024
Data blocks start
after root dir
Name ABC.DAT
Size 7200
Modify Time
First Block 2
Name A2.C
Size 5000
Modify Time
First Block 8
0
Boot Block
1 FAT
7 free 4 6 9 EOF A 5 3 B EOF 0
0 1 2 3 4 5 6 7 8 9 A B
2
3 Root Dir
4
0 aaa...aaa
5
1 ccc...ccc
6
2 ggg...ggg
7
3 bbb...bbb
8
4 ddd...ddd
9
5 fff...fff
A
6 eee...eee
B
7 iii...iii
C
8 hhh...hhh
D
9 mmm...mmm
E
A ooo...ooo
F
B zzz...zzz


FAT
File System
Block Size: 1024
Data blocks start after root dir
Name ABC.DAT
Size 7200
Modify Time
First Block 2
Name A2.C
Size 5000
Modify Time
First Block 8
0
Boot Block
1 FAT
7 free 4 6 9 EOF A 5
2
3 B EOF 0
0 1 2 3 4 5 6 7 8 9 A B
3 Root Dir
4
0 aaa...aaa
5
1 ccc...ccc
6
2 ggg...ggg
7
3 bbb...bbb
8
4 ddd...ddd
9
5 fff...fff
A
6 eee...eee
B
7 iii...iii
C
8 hhh...hhh
D
9 mmm...mmm
E
A ooo...ooo
F
B zzz...zzz

FAT File System

Block Size: 1024

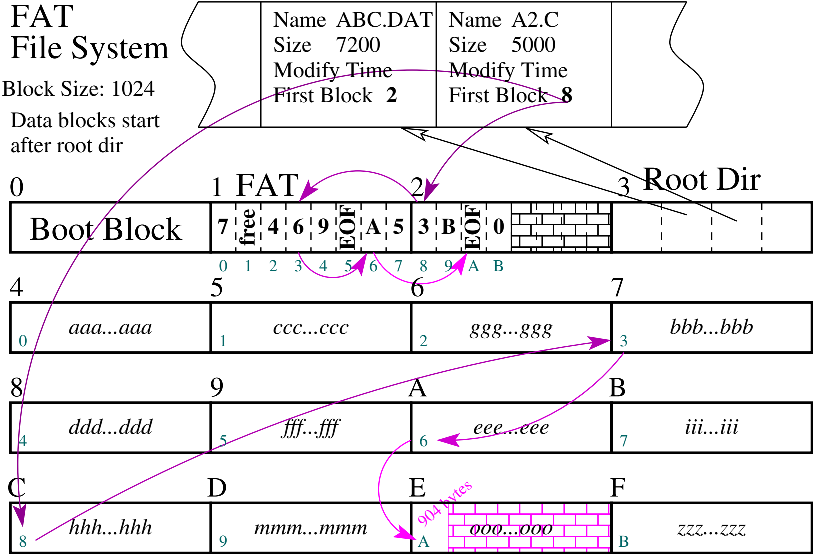
Data blocks start after root dir



FAT File System

Block Size: 1024

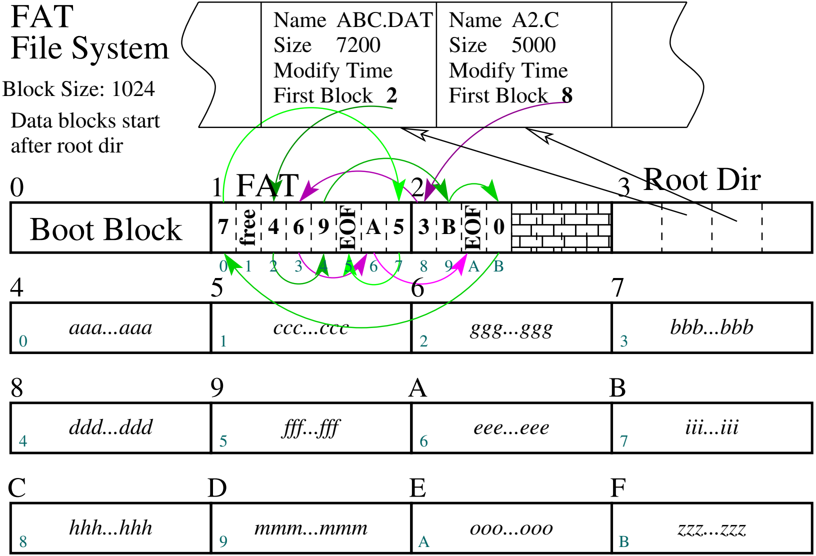
Data blocks start after root dir



FAT
File System

Block Size: 1024

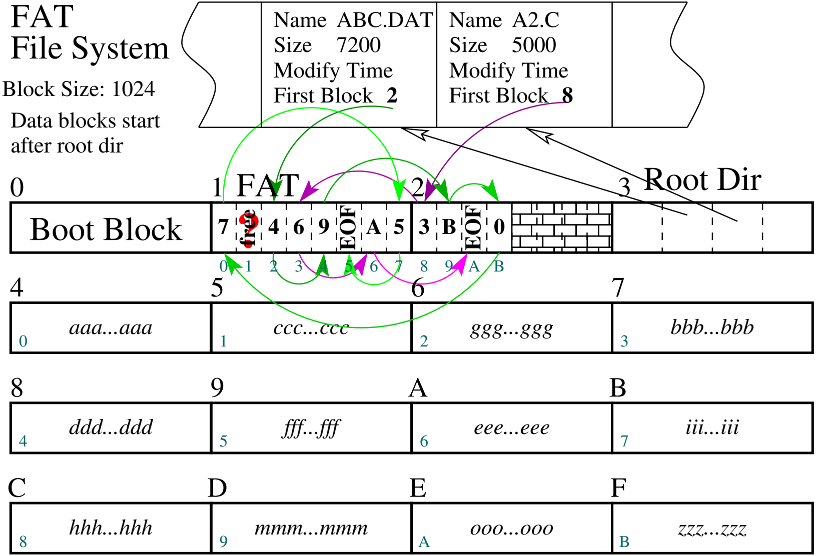
Data blocks start after root dir



FAT File System

Block Size: 1024

Data blocks start after root dir



FAT : Data Structures

MBR:		**Directory Entry**
(boot) jump location	OEM name	file name [8]
sector size	sectors per block	file extension [3]
number of FATs	# of root dir entries	attributes
total num sectors	media type	modification date/time
sectors per track	num heads	file length (bytes)
		first block

FAT : Floppy Disk Cluster/Block Sizes

Drive Size (logical volume)	*FAT Type*	*Sectors Per Cluster*	*Cluster Size*
360K	12-bit	2	1K
720	12-bit	2	1K
1.2Mb	12-bit	1	512b
1.44Mb	12-bit	1	512b
2.88Mb	12-bit	2	1K

From Microsoft KB article 314878,
`http://support.microsoft.com/kb/314878`

FAT : Hard Drive Cluster/Block Sizes

Drive Size (logical volume)	*FAT Type*	*Sectors Per Cluster*	*Cluster Size*
0–15Mb	12-bit	8	4K
16–127Mb	16-bit	4	2K
128–255Mb	16-bit	8	4K
256–511Mb	16-bit	16	8K
512–1023Mb	16-bit	32	16K
1024–2047Mb	16-bit	64	32K
2048–4095Mb	16-bit	128	64K

FAT : “Large” Hard Drive Cluster/Block Sizes

Drive Size (logical volume)	*FAT Type*	*Sectors Per Cluster*	*Cluster Size*
4–8Gb	16-bit	256	128K
8–16Gb	16-bit	512	256K

- not available prior to WinNT4.0, as reflected in the KB article (they are actually available afterwards)
- this is as far as basic FAT will go: $2^{16} \times (512 \times 512) = 17179869184 = 16384 \times (1024 \times 1024)$

FAT32 Cluster Sizes

Drive Size (logical volume)	*Cluster Size*
16–64Mb	512b
64–128Mb	1kb
128–256Mb	2kb
256Mb–8Gb	4kb
8Gb–16Gb	8kb

exFAT Cluster Sizes

Drive Size (logical volume)	*Cluster Size*
7–256Mb	4kb
256Mb–32Gb	32kb
32Gb–256Tb	128kb

NTFS : Cluster/Block Sizes (WinNT4.0 and up)

Drive Size (logical volume)	*Sectors Per Cluster*	*Cluster Size*	*Max Offset*
7Mb–16Tb	8	4k	$2^{32} \times 8 \times 512\text{b} = 2^{44}\text{b} = 16\text{Tb}$
16Tb–32Tb	16	8k	$2^{32} \times 16 \times 512\text{b} = 2^{45}\text{b} = 32\text{Tb}$
32Tb–64Tb	32	16k	
64Tb–128Tb	64	32k	
128Tb–256Tb	128	64k	$2^{32} \times 128 \times 512\text{b} = 2^{48}\text{b} = 32\text{Tb}$

From Microsoft KB article 140365,
`http://support.microsoft.com/kb/140365`

UFS : Unix File System

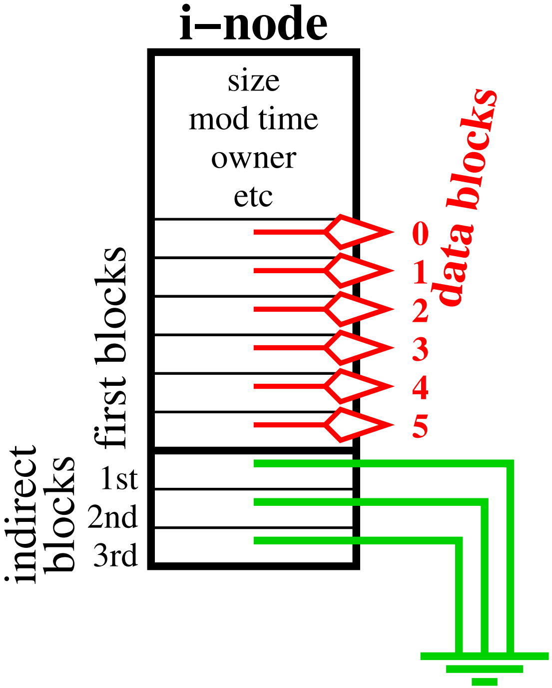
- file blocks are referenced in a **broad tree** whose root is an **index node** (also called an “**i-node**”)
- directories are stored in **files**
- the **root directory** is the **root** of the **filesystem tree**; the i-node of the root directory is in the **superblock**

UFS : Data Structures – **Super Block**

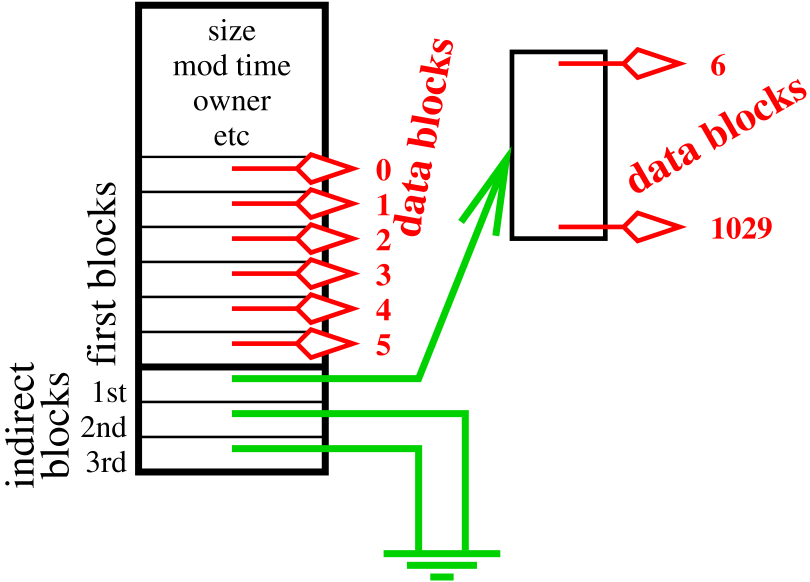
- Allocated to the first block of the filesystem and repeating on a fixed pattern
- Fields:
  - offset of first data block
  - file system size
  - i-node for FS root
  - head of free i-node list
  - head of free block list
  - device info (C/H/S *etc.*)
  - dirty flag
  - blocks/cylinder
  - volume label
  - num free i-nodes
  - num free blocks

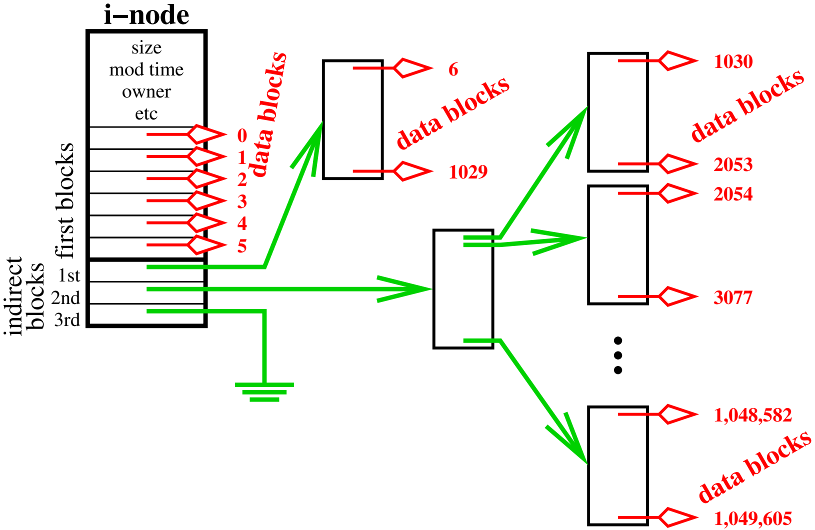
UFS : Data Structures — i-Node

Type	Field name
`long`	file mode
`long`	link count
`long`	owner id
`long`	group id
`di_size`	file size
`time_t`	last accessed
`time_t`	last modified
`time_t`	i-node accessed
	array of file block addresses

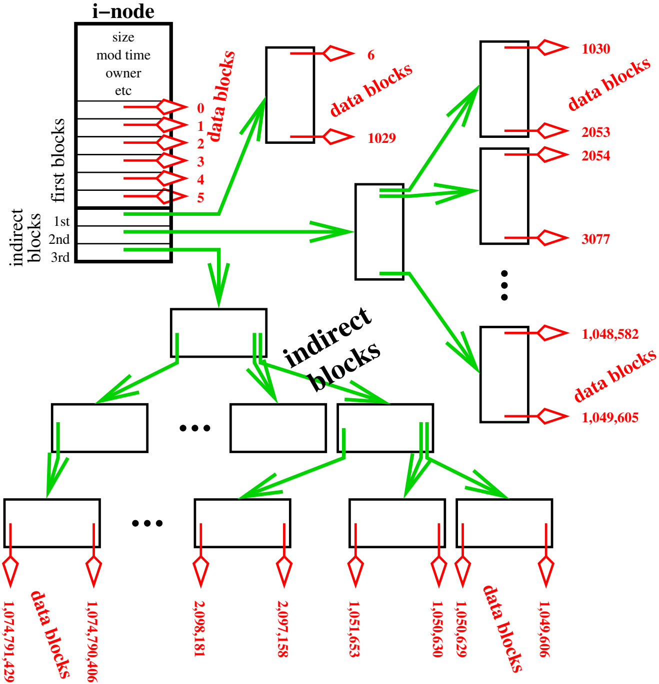

i–node
size
mod time
owner
etc
first blocks
data blocks
0
1
2
3
4
5
indirect blocks
1st
2nd
3rd

i–node

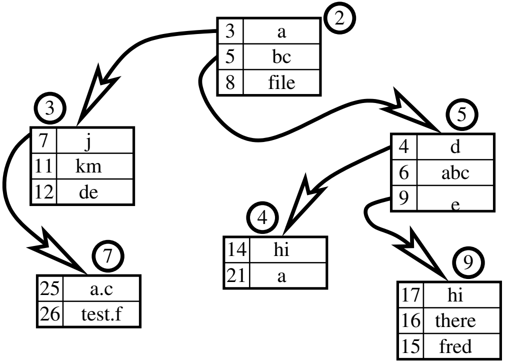

size
mod time
owner
etc
first blocks
0
1
2
3
4
5
data blocks
indirect blocks
1st
2nd
3rd
6
1029
data blocks

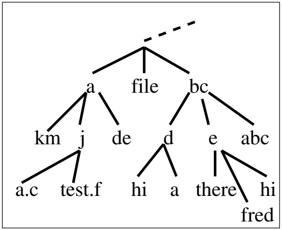


i–node
size
mod time
owner
etc
first blocks
0
1
2
3
4
5
data blocks
indirect blocks
1st
2nd
3rd
6
data blocks
1029
1030
data blocks
2053
2054
3077
1,048,582
data blocks
1,049,605


i–node
size
mod time
owner
etc
first blocks
0
1
2
3
4
5
data blocks
indirect blocks
1st
2nd
3rd
6
1029
data blocks
1030
2053
2054
3077
data blocks
1,048,582
1,049,605
data blocks
indirect blocks
1,074,791,429
data blocks
1,074,790,406
2,098,181
2,097,158
1,051,653
1,050,630
1,050,629
data blocks
1,049,606

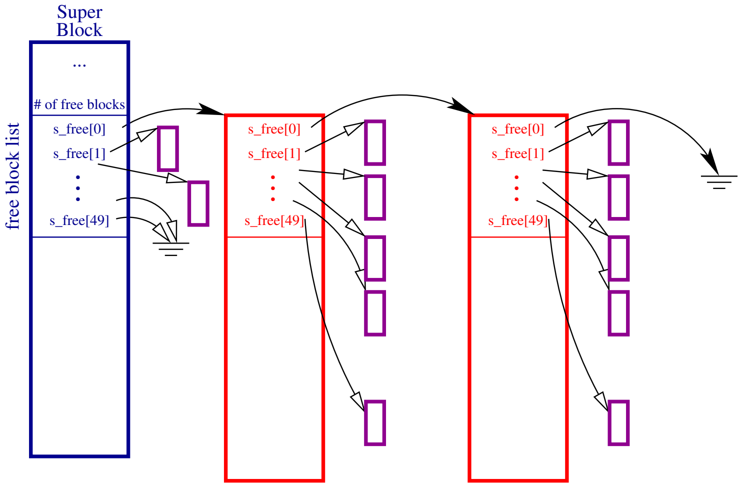
UFS Example





i – number of i–node referencing the data shown with 'dir' mode bit set

The UFS Freelist



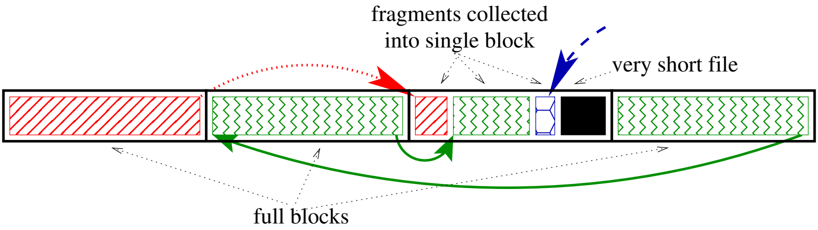
Unix Filesystem Performance

McKusick, Marshall K. *et al*, “A Fast File System for UNIX”, *Computer Systems*, 2(3):181–197, 1984.

Space Used	*Waste*	*Organization*
775.2 Mb	0.0 %	Data only, no separation between files
807.8 Mb	4.2 %	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9 %	Data + inodes, 512 byte block old UFS
866.5 Mb	11.8 %	Data + inodes, 1024 byte block old UFS
948.5 Mb	22.4 %	Data + inodes, 2048 byte block old UFS
1128.3 Mb	**45.6** %	Data + inodes, 4096 byte block old UFS

Speeding up a File System: Increase block size

- but this increases wasted space!
- so collect "fragments" into a single block



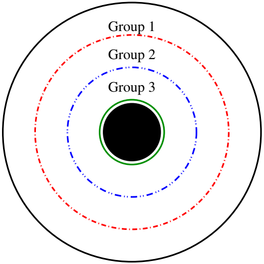
Unix Filesystem Performance, cont'd

Throughput			*kbytes/sec*
read	Old	1024	29
	New	4096	221
	New	8192	233
write	Old	1024	48
	New	4096	142
	New	8192	215

Speeding up a File System: Reduce seek distance

- allocate blocks close together when possible
- one way is to use **extents** — (attempt to) keep file in contiguous blocks
  - DOS/Windows “defragging” operation laboriously reorders files to achieve this
    - operation is expensive $\rightarrow$ user initiated
    - degradation over time
- another way is to use **cylinder groups** and allocate all blocks for a file (both data and metadata) from the same cylinder group

$\hookrightarrow$



Allocation to Cylinder Groups

- growing an i-node
  - allocate within cylinder group for a while (ideally contiguously), *then*
  - spill over into other cylinder groups every few megabytes
- allocate a new i-node (regular file)
  - in same cylinder group as the directory it is in
- allocate a new i-node (directory)
  - in a different cylinder group with lots of free i-nodes and few directories

Reliability -*vs*-Performance

Reliability

- redundant information
- synchronous writing

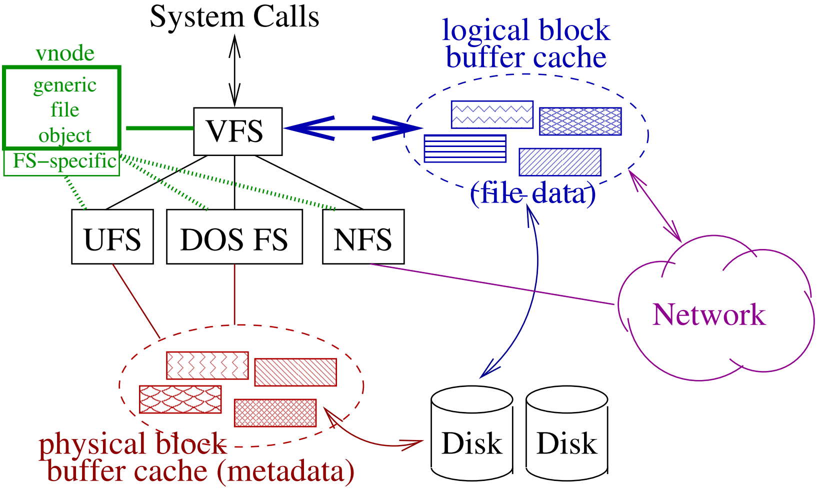
Performance

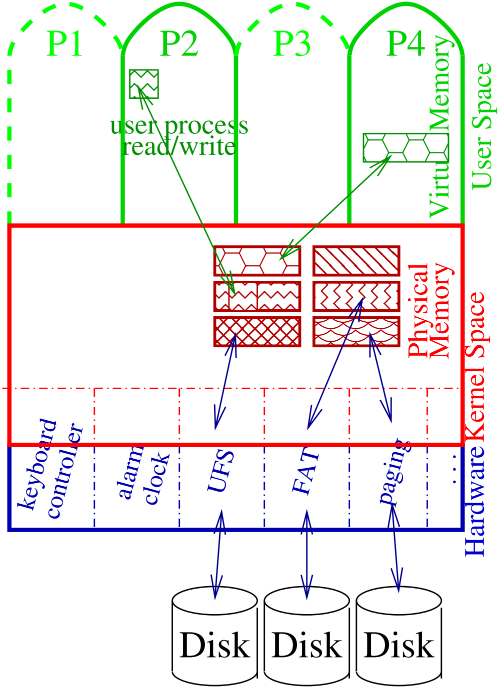
- delayed writing
- increased risk of structural damage when a crash occurs

BSD file system:

- write new i-node before the directory entry that points to it
- remove directory entry before deallocating i-node
- write deallocated i-node to disk before freeing data blocks

- also, speed up writing by
  - collating several writes into one
  - keep the writes close together
  - do the writes when the disk isn't too busy


System Calls
vnode
generic
file
object
FS–specific
VFS
logical block
buffer cache
(file data)
UFS
DOS FS
NFS
Network
physical block
buffer cache (metadata)
Disk
Disk


P1
P2
P3
P4
user process
read/write
Virtual Memory
User Space
Physical
Memory
Kernel Space
keyboard
controller
alarm
clock
UFS
FAT
paging
. . .
Hardware
Disk
Disk
Disk

Reading

- calculate block # (divide byte offset/block size)
- look in logical buffer cache for the block #
- if found
  - copy data to user process data area
- else
  - *next ...*

Reading, cont'd

(*else*)

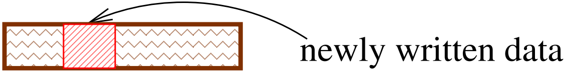
- allocate a block in the logical buffer cache for the block #
- perform read of the block for the file system
    - use meta-data for the file system to map file block # $\Rightarrow$ disk block #
    - if valid data not on the disk $\rightarrow$ fill block with zeros
    - issue disk block read to the disk controller hardware via device driver
    - wait for I/O operation to complete (*i.e.*; wait for interrupt)
    - copy data to user process data area

Writing

- calculate block # (divide byte offset/block size)
- look in logical buffer cache for the block #
- if *not* found:
  - allocate a buffer for the block #
  - if valid data not on the disk $\rightarrow$ fill block with zeros
  - else if the write won't fill the entire buffer
    - perform read of the block from the file system (same as read)
- *next* ...

Writing, cont'd

- copy data from user process data area to buffer



- mark the buffer modified
- schedule a write of the block #
  - use metadata to map file block # to disk block #
  - issue disk block write to the hardware disk device controller via the device driver

Writing Schedules

Scheduling can be:

synchronous $\rightarrow$ waits for write to complete on disk

asynchronous $\rightarrow$ start write, but don't wait for completion

"write back" $\rightarrow$ do write later (*a.k.a.* "delayed write")

I/O Tricks

Read Ahead:

- try and guess the next block that the process might read, and read it into the buffer cache in anticipation

Write Behind:

- start now, but don't wait for completion (asynchronous)
  OR
- do it later (delayed write)
  - could result in several writes being combined into one

  - metadata might be repeatedly modified (file length/modification time updates in particular)

SSD : Solid State Drives

- finite-write technology based on a erase-to-write technology:
  MLC: "multi-level cell" $\propto 10^3$ total writes before failure
  SLC: "single-level cell" $\propto 10^4$
- wear-levelling: "it is better to spread writes across drive"
  - limited cycles per erase block
  - erase block size?
  - typical filesystem (re)writes meta-data many times
  - SSD moves data around $\rightarrow$ fragmentation
- actual solutions:
  - write tricks
  - change FS algorithms?
  - change reported data (access times)?

Journaling

- if a power fails or system crashes without unmount, recovery can be quite costly
- "journaling" takes a database (transaction) approach to filesystem updates
- each write action is a **transaction**, recorded in **journal**
- each transaction is not complete until filesystem integrity is maintained (i-lists, dir entries + blocks all updated)

Journaling Protocol

- record writes to a circular queue, schedule a write
- if queue fills, block
- one physical write at a time, from head of queue
- when write completes (data + metadata) remove write from queue (journal)

Result: actions of write are reproducible

during recovery, knowledge of tasks being performed is available to reconstruct actions on data + metadata.

recovery consists of completing all actions listed in **journal**

Journaling Schemes

Different schemes, differ in **currency**:

regions - ResiserFS

- transactions are recorded in offset+length structures
- may span disk blocks
- may not start/end on block bounds

blocks - ext3

- transactions are recorded in terms of blocks
- more efficient
  - **write-behind**
  - block alignment factors
  - code is simpler as block based structures exist in FS code already

Locks : (File) Data Access Synchronization

- (file) lock $\rightarrow$ **region** of a file (offset + extent)
- raises issue of intersection, inclusion
- two types (four names):

  Read/Shared Lock :
  - multiple locks can exist for same region
  - prevents exclusive lock from being set

  Write/Exclusive Lock :
  - complete access (to region)
- when multiple lock types are in wait and lock comes free, **who gets it?**

Partitions

- divide a large disk into several smaller logical volumes of non-overlapping **extents**
- based on a table in cylinder 0 that indicates the begin/end location of each partition, and usually a filesystem type flag
- some (older) schemes partition in terms of cylinders, others can take any sector numbers

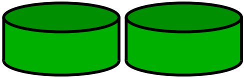
Partitions – DOS style

- DOS style partitions are set up inside the MBR
  - “primary” partition table has four slots
  - partition has an extent and a type: “primary” or “extended”
  - an “extended” partiton is a container for “logical” partitions stored outside the MBR
  - (some tools work badly if there is more than one extended partition, and some boot manager cannot find filesystems in logical partitions)

Partitions – UFS style

- UFS partitions are set up in the first sector of the device, and typically have 8 entries:
  - a: the root filesystem;
  - b: the “swap” partition (paging area);
  - c: span - raw access to entire disk
  - d-h: general use (typically /usr, /var, /home and others)

RAID 0: “striping” or “byte/block splitting”



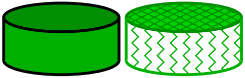
block-level “striping” $\rightarrow$ split the data into N parts, each write is $1/N$ original size

advantage: speed

cost: failure of any disk is fatal

overhead: none

RAID 1: “mirroring”



disk-level mirroring $\rightarrow$ keep a spare copy

advantage: reliability (single-disk failure)

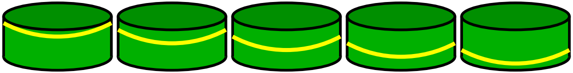
cost: reduces total speed

overhead: double the space requirements

RAID 2,3,4,5: parity plans

- parity calculation allows us to catch 1-bit errors (RAID 2 - never used)
- drives self-detect failure, so we save one bit on RAID 2 plan (RAID 3)
- we can store all the parity bits for a block together (RAID 4)
- if we distribute all this across the drives, no single device is involved in every write (RAID 5)

RAID 5: block-interleaved distributed parity



parity blocks distributed across other disks

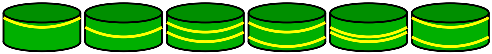
advantage: reliability (single-disk failure)

cost: two disks involved in all I/O

overhead: 1 parity per N data + calculation

parity for block m stored on disk $(m \mod N) + 1$

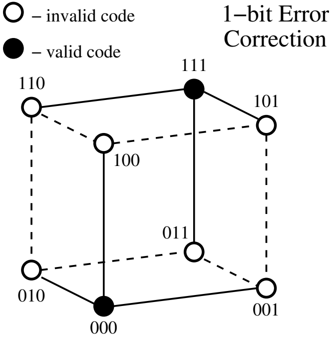
RAID 6: P+Q redundancy



like level 5 but more parity → tolerate more failures

advantage: reliability (multi-disk failure)

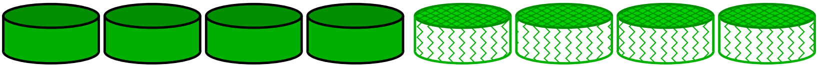
cost: failure level + 1 disks involved in all I/O



overhead: variable $\propto$ redundancy (usually) based on Reed-Solomon codes (data values form coefficients for a polynomial; store values + samples to allow reconstruction)

(The general plan is to use data values as coefficients for a polynomial, and store some points on the resulting line; the line may be interpolated for reconstruction of original coefficients)

RAID 0+1: speed + safety = $$$



do *both* RAID 0 and RAID 1 simultaneously

advantage: speed + reliability (single-disk failure)

cost: best case $\mapsto$ 0 provides performance, 1 provides reliability

overhead: double the disk space needed