Sistemas Operativos
Capítulo 10
File systems

Patrício Domingues, ESTG/IPLeiria

Adaptado de: CS5600 – Computer Systems – Lecture 9

File systems

- ✓ HDDs/SSDs offer a blank slate of empty blocks
  - How do we store files on these devices, and keep track of them?
  - How do we maintain high performance?
  - How do we maintain consistency in the face of random crashes?
    - Disk organization and file systems

- ✓ Partitions and Mounting
- ✓ Basics (FAT)
- ✓ inodes and Blocks (ext)
- ✓ Block Groups (ext2)
- ✓ Journaling (ext3)
- ✓ Extents and B-Trees (ext4)
- ✓ Log-based File Systems

Building the Root File System

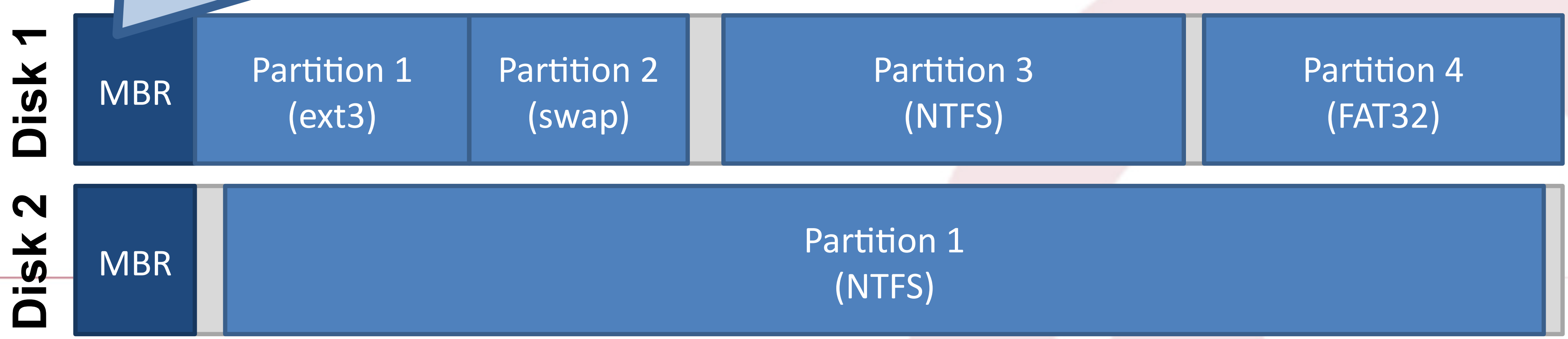
- ✓ One of the first tasks of an OS during bootup is to build the root file system

1. Locate all bootable media
   - Internal and external hard disks
   - SSDs
   - Floppy disks, CDs, DVDs, USB sticks
2. Locate all the partitions on each media
   - Read MBR(s), extended partition tables, etc.
     - MBR: Master Boot Record
3. Mount one or more partitions
   - Makes the file system(s) available for access

The Master Boot Record

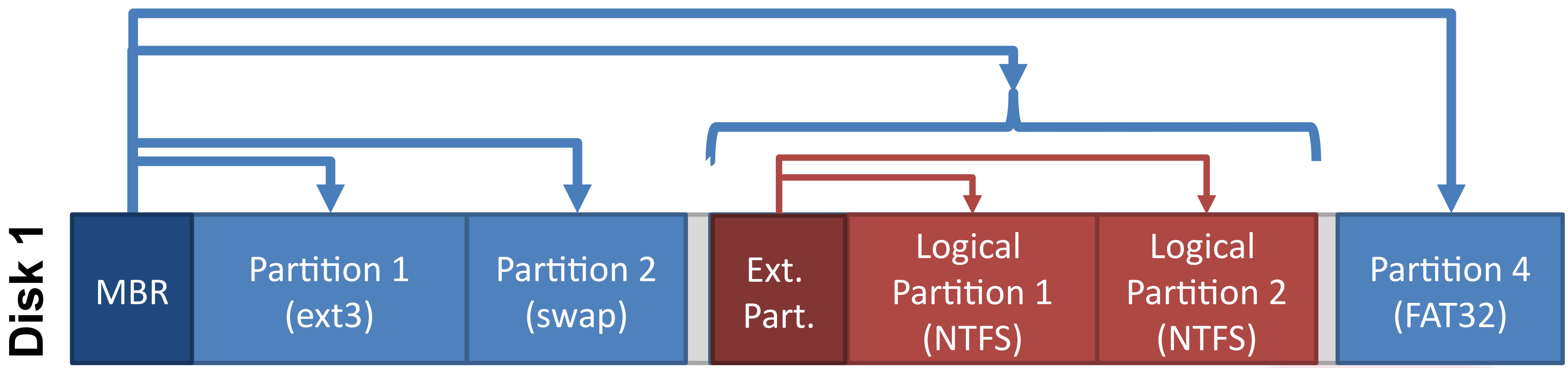
Address		Description	Size (Bytes)
Hex	**Dec.**		
0x000	0	Bootstrap code area	446
0x1BE	446	Partition Entry #1	16
0x1CE	462	Partition Entry #2	16
0x1DE	478	Partition Entry #3	16
0x1EE	494	Partition Entry #4	16
0x1FE	510	Magic Number	2
		Total:	**512**

LBA and length of the partition



Extended Partitions

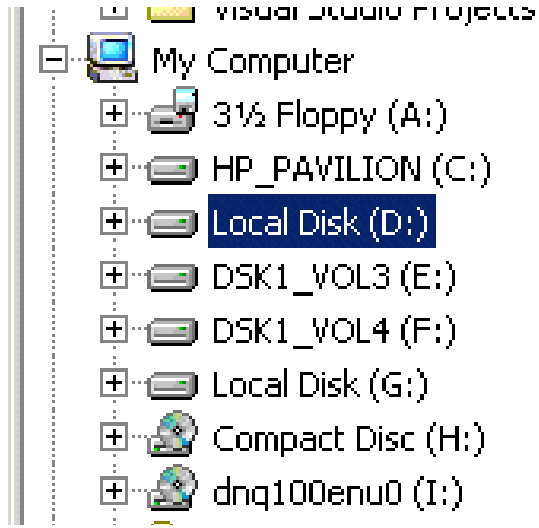
- ✓ In some cases, you may want more than four partitions
- ✓ Modern OSes support extended partitions



- Extended partitions may use OS-specific partition table formats (meta-data)
  - Thus, other OSes may not be able to read the logical partitions

Root File Systems (#1)

- ✓ Windows exposes a multi-rooted system
  - – Each device and partition is assigned a letter
  - – Internally, a single root is maintained



Root File Systems (#2)

- Linux has a single root
- One partition is mounted as /
- All other partitions are mounted somewhere under /
- Typically, the partition containing the kernel is mounted as /

```
[user@linux~] df -h
Filesystem   Size  Used   Avail   Use%   Mounted on
/dev/sda7    39G   14G    23G     38%    /
/dev/sda2    296M  48M    249M    16%    /boot/efi
/dev/sda5    127G  86G    42G     68%    /media/cbw/Data
/dev/sda4    61G   34G    27G     57%    /media/cbw/Windows
/dev/sdb1    1.9G  352K   1.9G    1%     /media/cbw/NDSS-2013
```

1 drive, 4 partitions

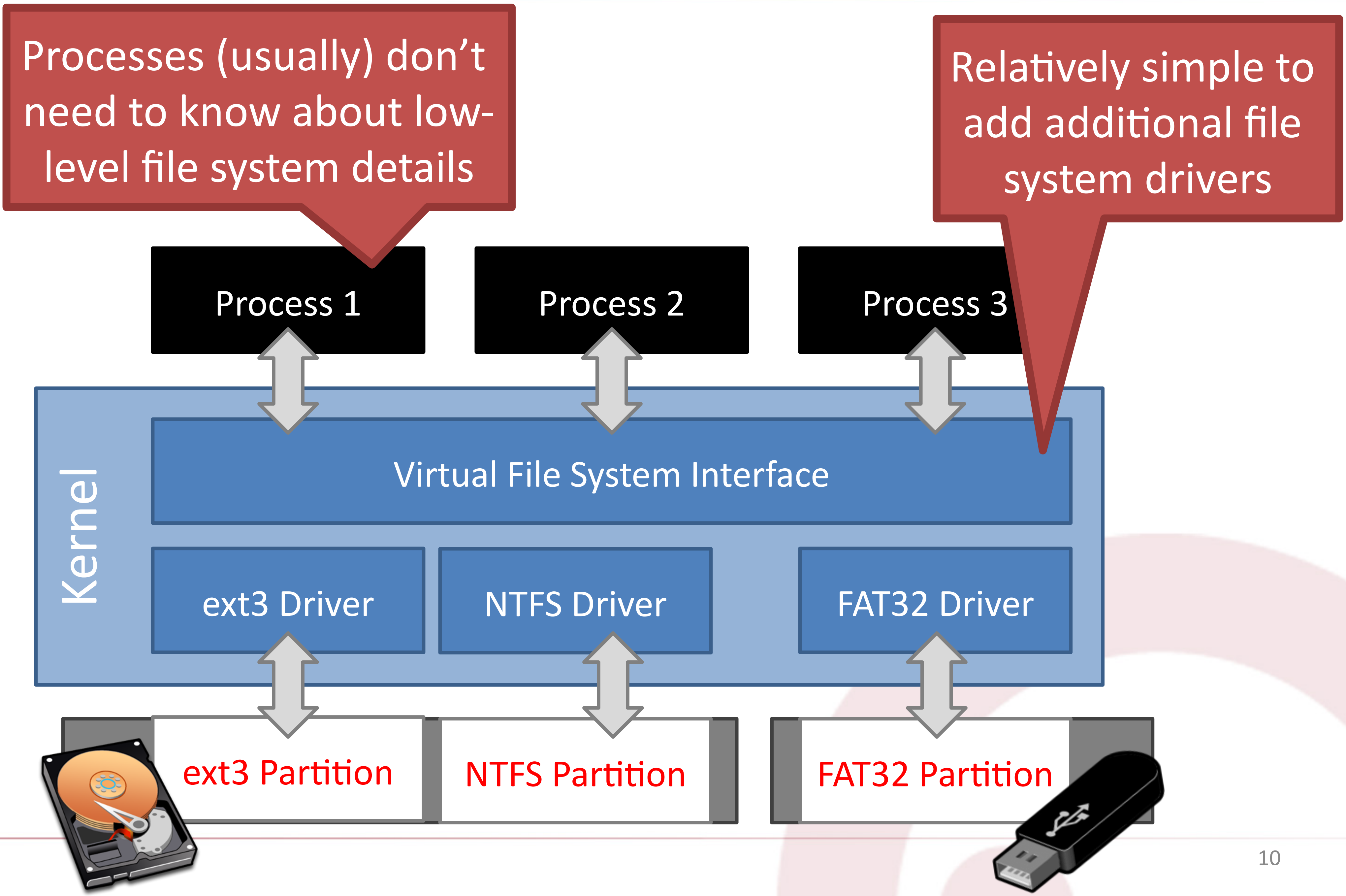
1drive, 1 partition

Virtual File System Interface

- Problem: the OS may mount several partitions containing different underlying file systems
  - It would be bad if processes had to use different APIs for different file systems
- Linux uses a Virtual File System interface (VFS)
  - Exposes POSIX APIs to processes
  - Forwards requests to lower-level file system specific drivers
- Windows uses a similar system

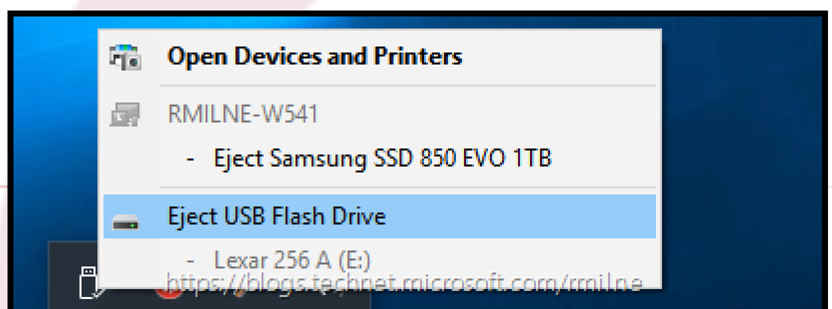
Virtual File System *flowchart* >>

VFS Flowchart



mount isn’t Just for Bootup

- ✓ When you plug storage devices into your running system, mount is executed in the background
- ✓ Example: plugging in a USB stick
- ✓ What does it mean to “safely eject” a device?
  - – Flush cached writes to that device
  - – Cleanly unmount the file system on that device

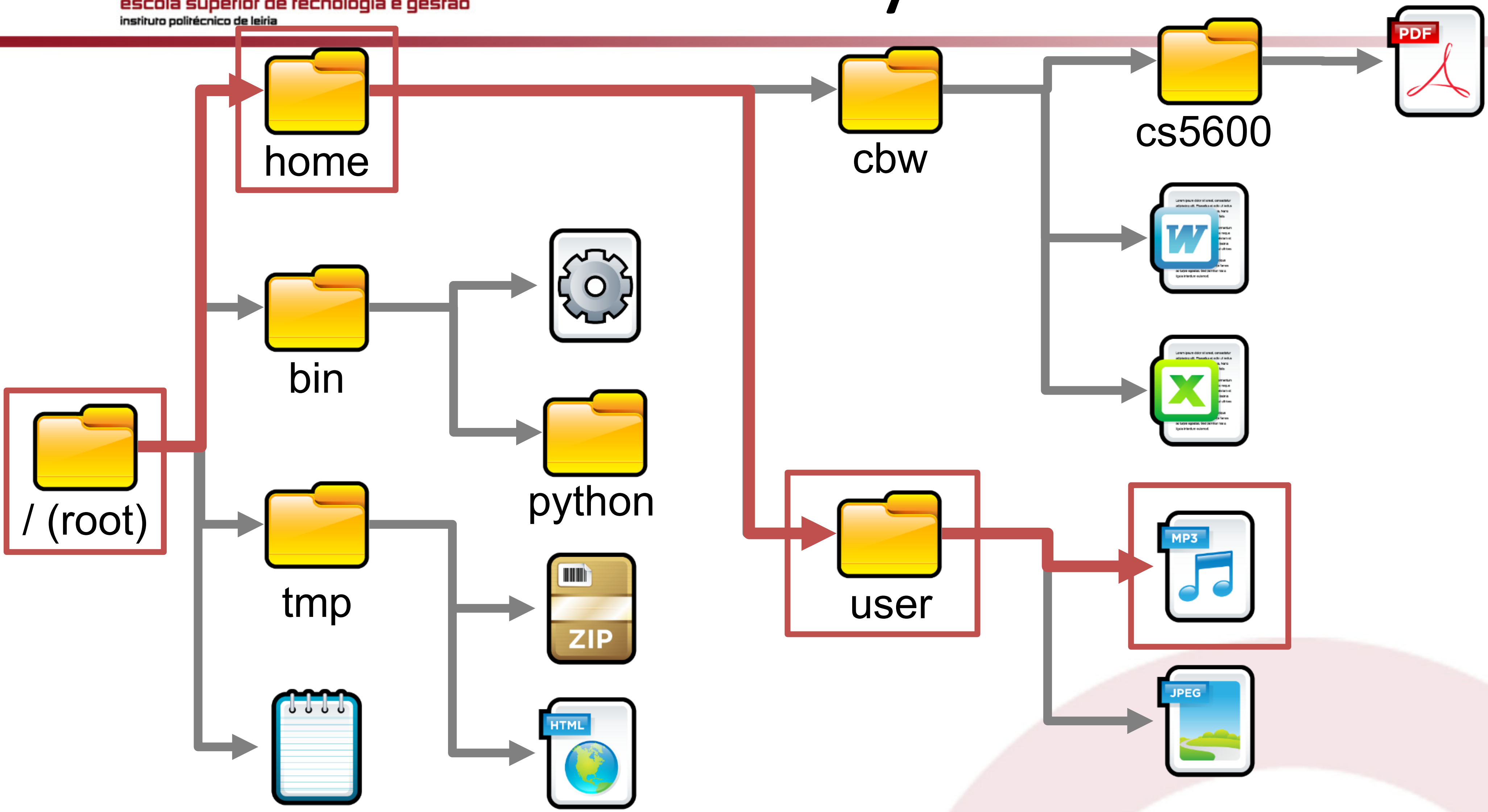


- ✓ Partitions and Mounting
- ✓ Basics (FAT)
- ✓ inodes and Blocks (ext)
- ✓ Block Groups (ext2)
- ✓ Journaling (ext3)
- ✓ Extents and B-Trees (ext4)
- ✓ Log-based File Systems

Status Check

- ✓ At this point, the OS can locate and mount partitions
- ✓ Next step: what is the on-disk layout of the file system?
  - We expect certain features from a file system
    - Named files
    - Nested hierarchy of directories
    - Meta-data like creation time, file permissions, etc.
  - How do we design on-disk structures that support these features?

The Directory Tree



- ✓ Navigated using a path
  - E.g. /home/user/music.mp3

Absolute and Relative Paths

- Two types of file system paths
  - Absolute
    - Full path from the root to the object
    - Example: /home/cbw/cs5600/hw4.pdf
    - Example: C:\Users\cbw\Documents\
  - Relative
    - OS keeps track of the working directory for each process
    - Path relative to the current working directory
    - Examples [working directory = /home/cbw]:
      - syllabus.docx [→ /home/cbw/syllabus.docx]
      - cs5600/hw4.pdf [→ /home/cbw/cs5600/hw4.pdf]
      - ./cs5600/hw4.pdf [→ /home/cbw/cs5600/hw4.pdf]
      - ../amislove/music.mp3 [→ /home/amislove/music.mp3]

Files (reminder)

- ✓ A file is a composed of two components
  - – The file data itself
    - One or more blocks (sectors) of binary data
    - A file can contain anything
  - – Meta-data about the file
    - Name, total size
    - What directory is it in?
    - Created time, modified time, access time
    - Hidden or system file?
    - Owner and owner’s group
    - Permissions: read/write/execute

File Extensions (#1)

- ✓ File name are often written in dotted notation
  - E.g. program.exe, image.jpg, music.mp3
- ✓ A file's extension **does not mean anything**
  - Any file (regardless of its contents) can be given any name or extension

Has the data in the file changed from music to an image?

- Graphical shells (like Windows explorer) use extensions to try and match files to programs
  - This mapping may fail for a variety of reasons

File Extensions (#2)

- ✓ The **`file`** utility available in Unix aims to detect the type of a given file regardless of its extension

Examples

`file screenshot.png`

```
screenshot.png: PNG image data, 816 x 573, 8-bit/color RGB, non-interlaced
```

`file memtest86+.bin`

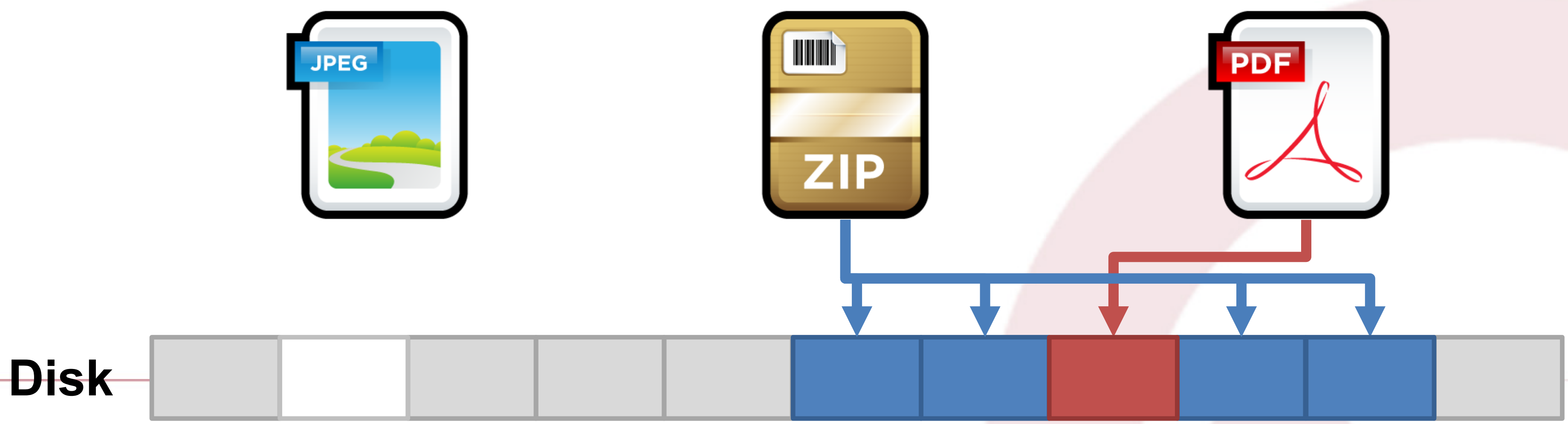
```
memtest86+.bin: DOS/MBR boot sector
```

`file EmptyProject-Templatev3.03.zip`

```
EmptyProject-Templatev3.03.zip: Zip archive data, at least v2.0 to extract
```

More File Meta-Data

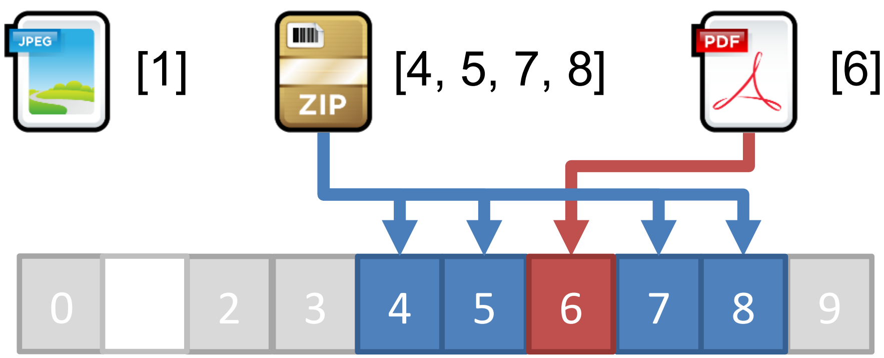
- ✓ Files have additional meta-data that is not typically shown to users
  - – Unique identifier (file names may not be unique)
  - – Structure that maps the file to blocks on the disk
- ✓ Managing the mapping from files to blocks is one of the key jobs of the file system



Mapping Files to Blocks

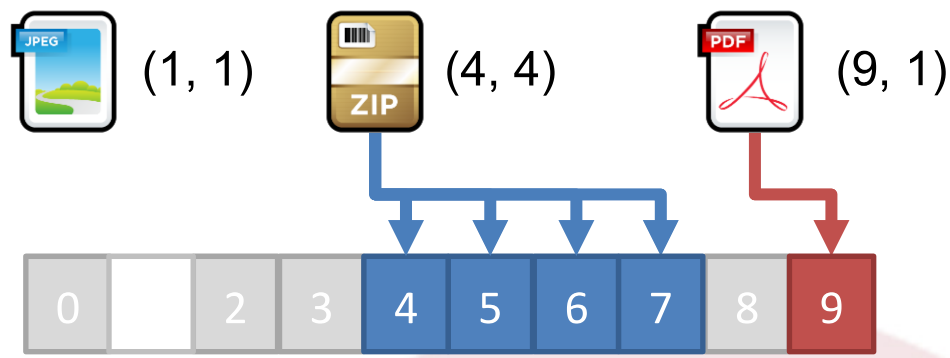
- ✓ Every file is composed of >=1 blocks
- ✓ Key question: how do we map a file to its blocks?

List of blocks



- Problem?
  - Really large files

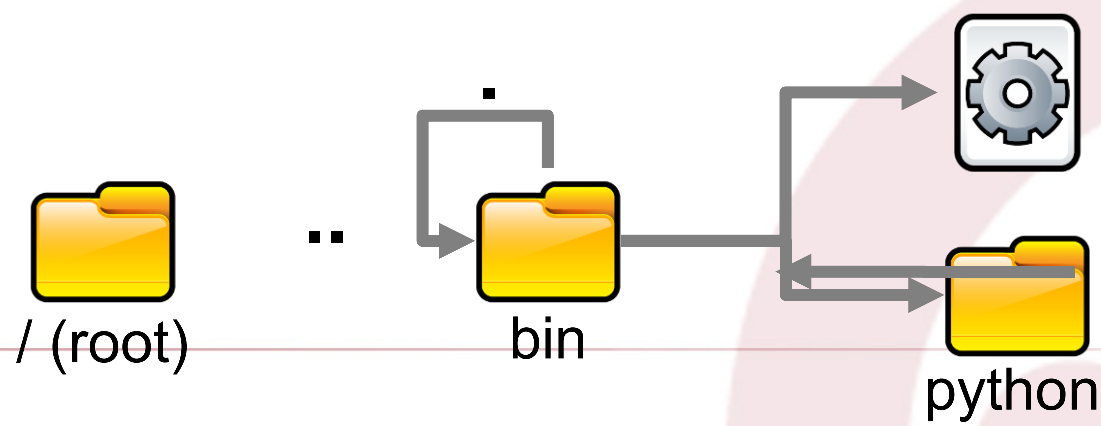
As (start, length) pairs



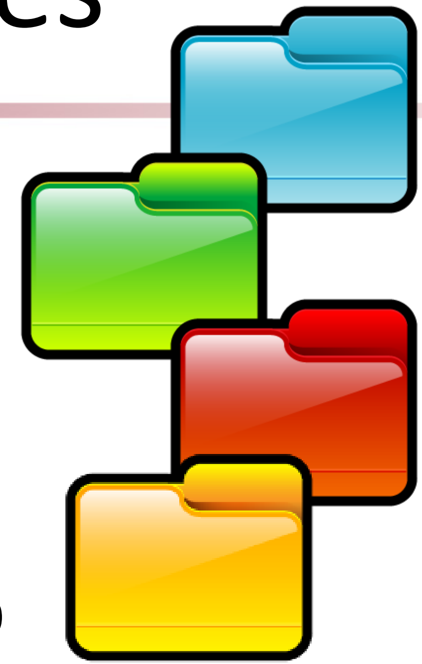
- Problem?
  - Fragmentation
  - E.g. try to add a new file with 3 blocks

Directories

- ✓ Traditionally, file systems have used a hierarchical, tree-structured namespace
  - – Directories are objects that contain other objects
    - i.e. a directory may (or may not) have children
  - – Files are leaves in the tree
- ✓ By default, directories contain at least two entries

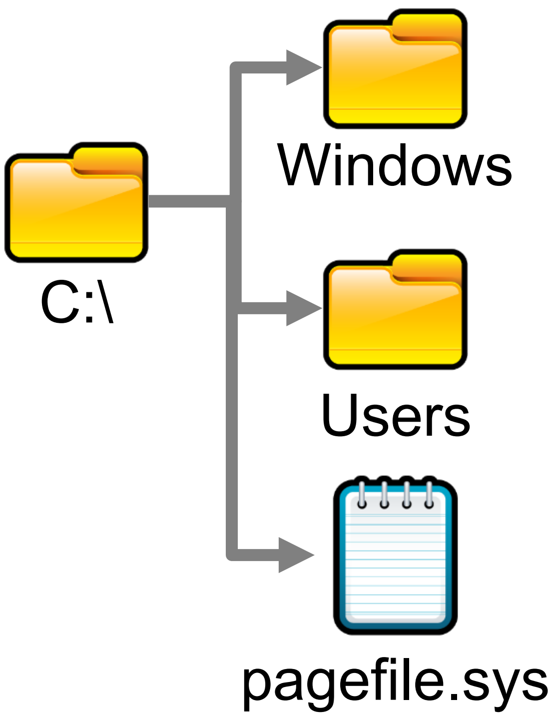


More on Directories



- ✓ Directories have associated meta-data
  - – Name, number of entries
  - – Created time, modified time, access time
  - – Permissions (read/write), owner, and group
- ✓ The file system must encode directories and store them on the disk
  - – Typically, directories are stored as a *special type of file*
  - – File contains a list of entries inside the directory, plus some meta-data for each entry

Example Directory File



Name	Index	Dir?	Perms
.	2	Y	rwx
Windows	3	Y	rwx
Users	4	Y	rwx
pagefile.sys	5	N	r

File Allocation Tables (FAT)

- ✓ Simple file system popularized by MS-DOS
  - First introduced in 1977
  - Most devices today use the FAT32 spec from 1996
  - FAT12, FAT16, VFAT, FAT32, ExFAT, etc.
- ✓ Still quite popular today
  - Default format for USB sticks and memory cards
  - Used for EFI boot partitions
- ✓ Name comes from the index table used to track directories and files
  - FAT: File Allocation Table

- Stores basic info about the file system
- FAT version, location of boot files
- Total number of blocks
- Index of the root directory in the FAT

File allocation table (FAT)
Marks which blocks are free or in-use
Linked-list structure to manage large files

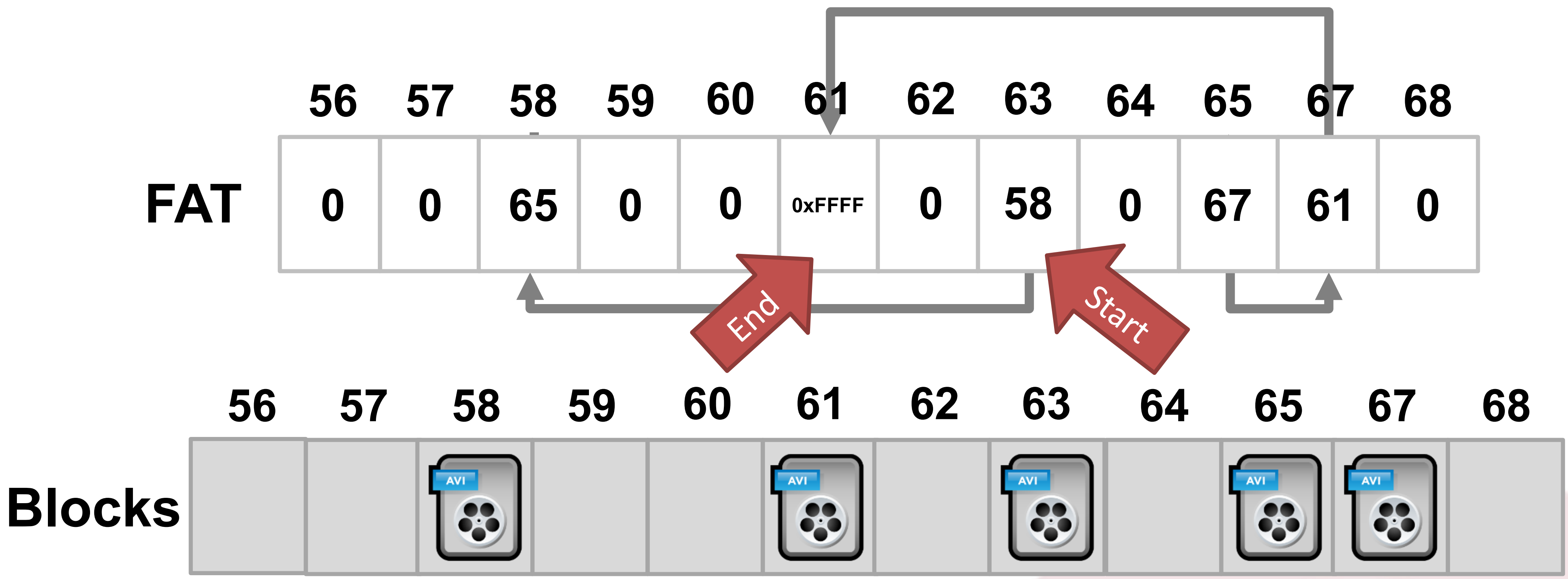
- Store file and directory data
- Each block is a fixed size (4KB – 64KB)
- Files may span multiple blocks

Fat Table Entries

- ✓ len(FAT) == Number of clusters on the disk
  - – A cluster is a set with a given number of blocks
    - – The number of blocks is constant across the filesystem
  - – Max number of files/directories is bounded
    - – Decided when you format the partition
- ✓ The FAT version roughly corresponds to the size in bits of each FAT entry
  - – E.g. FAT16: each FAT entry is 16 bits, meaning that the disk can only have 2^{16} entries (65536)
  - – More bits (e.g., FAT32): supports larger disks

Fragmentation

- ✓ Blocks for a file need not be contiguous



Possible values for FAT entries:

- 0 – entry is empty
- 1 < N < 0xFFFF – next block in a chain
- 0xFFFF – end of a chain

FAT: The Good and the Bad

- ✓ The Good – FAT supports:
  - Hierarchical tree of directories and files
  - Variable length files
  - Basic file and directory meta-data
- ✓ The Bad
  - At most, FAT32 supports 2TB disks
  - Locating free chunks requires scanning the entire FAT
    - Slow…
  - Prone to internal and external fragmentation
    - Large blocks/clusters: internal fragmentation
  - **Reads require a lot of random seeking**
    - Not good, especially for HDD

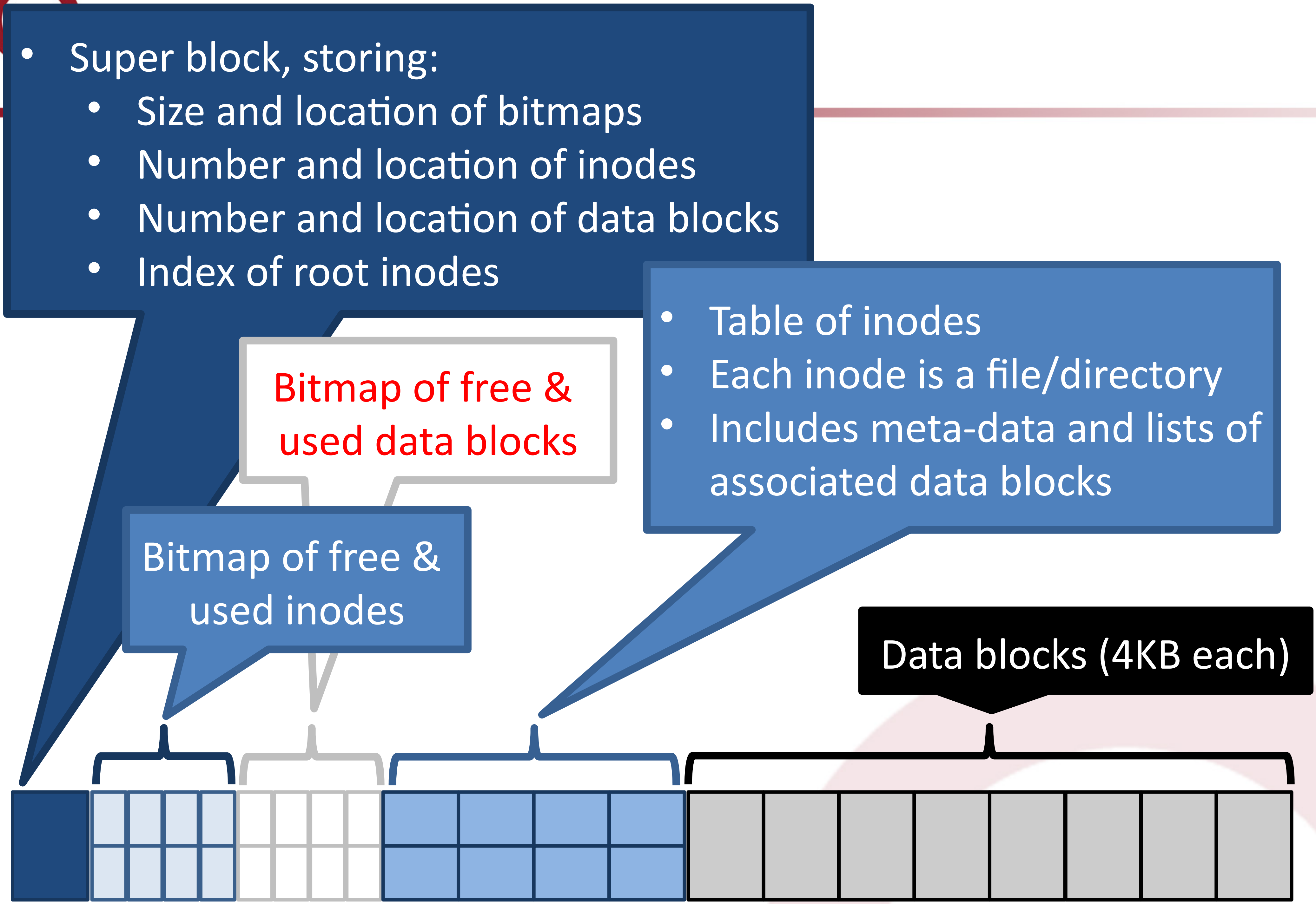
- ✓ Partitions and Mounting
- ✓ Basics (FAT)
- ✓ inodes and Blocks (ext)
- ✓ Block Groups (ext2)
- ✓ Journaling (ext3)
- ✓ Extents and B-Trees (ext4)
- ✓ Log-based File Systems

Status Check

- ✓ At this point, we have on-disk structures for:
  - Building a directory tree
  - Storing variable length files
- ✓ But, the efficiency of FAT is very low
  - Lots of seeking over file chains in FAT
  - Only way to identify free space is to scan over the entire FAT
- ✓ Linux file system uses more efficient structures
  - Extended File System (ext) uses index nodes (inodes) to track files and directories

Size Distribution of Files

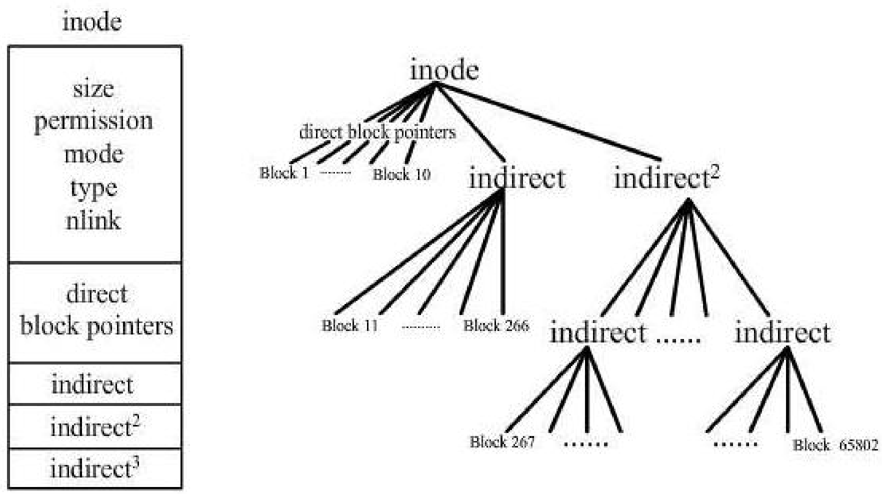
- FAT uses a linked list for all files
  - Simple and uniform mechanism
  - … but, it is not optimized for short or long files
- Question: are short or long files more common?
  - Studies over the last 30 years show that short files are much more common
  - 2KB is the most common file size
  - Average file size is 200KB
    - *biased upward* by a few very large files
- Key idea: optimize the file system for many small files



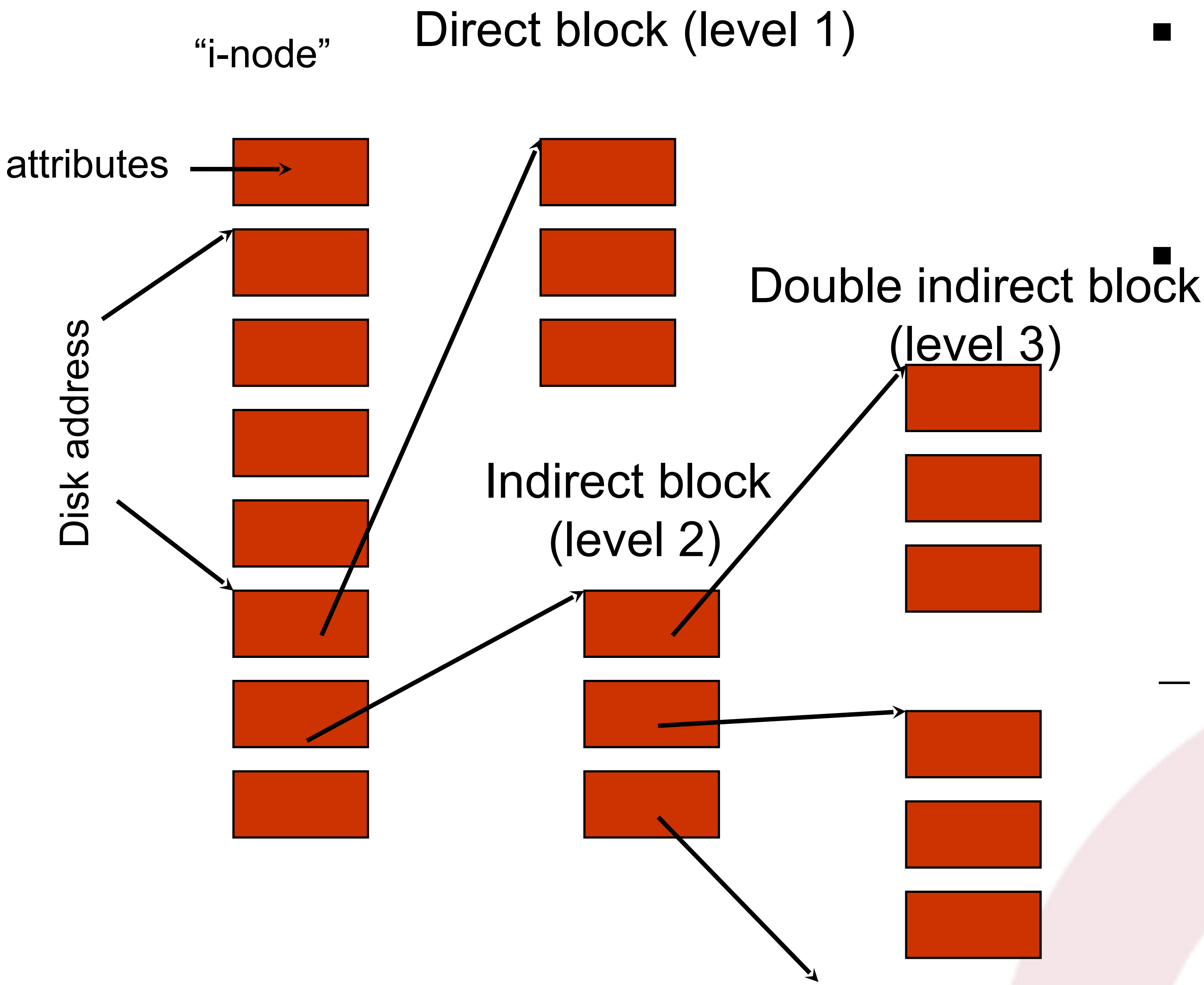
Super block, storing:
Size and location of bitmaps
Number and location of inodes
Number and location of data blocks
Index of root inodes
Bitmap of free & used data blocks
Table of inodes
Each inode is a file/directory
Includes meta-data and lists of associated data blocks
Bitmap of free & used inodes
Data blocks (4KB each)

inodes (1)

- *inode*: data structure used for many Un*x file systems
- Each *inode* holds the location of a file
  - A directory is a special case of a file
- inode is organized in levels
  - A inode has 12 pointers for level 1 blocks. These point directly to the respective 12 data blocks
  - A inode has also a pointer for level 2 block: this level 2 block points to a block that points to other data blocks (*single indirect block*)
  - A inode has also a pointer for level 3 block: this level 3 block points to a block which points to a set of data blocks (*double indirect block*)
  - A inode has also a pointer for level 4 block (*triple indirect block*)



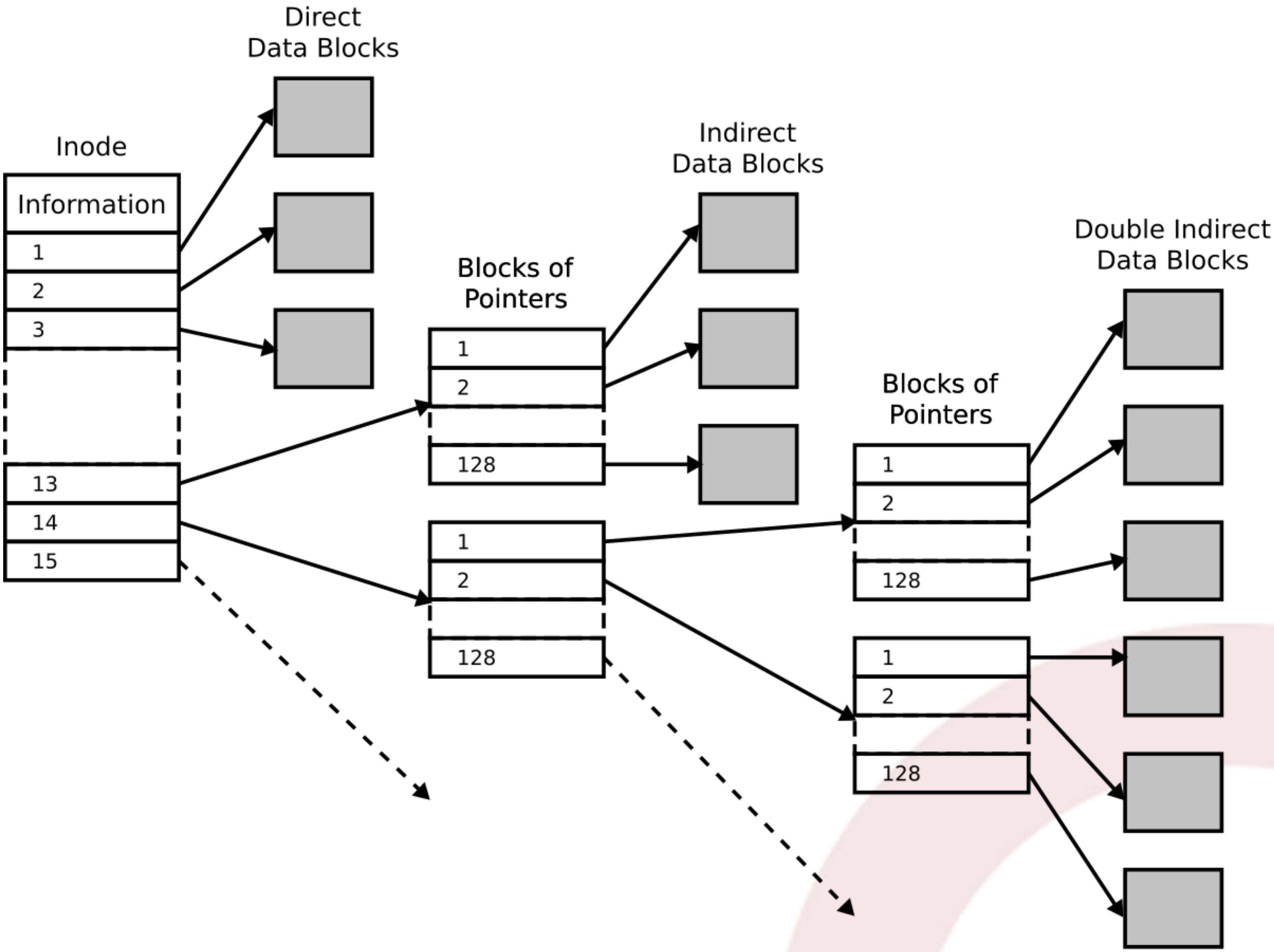
inodes (2)



- Optimized for small files
  - Direct access
- It handles large files
  - Large files are accessed more slowly
    - More levels need to be accessed

– In Unix, many files are small

inodes (3)

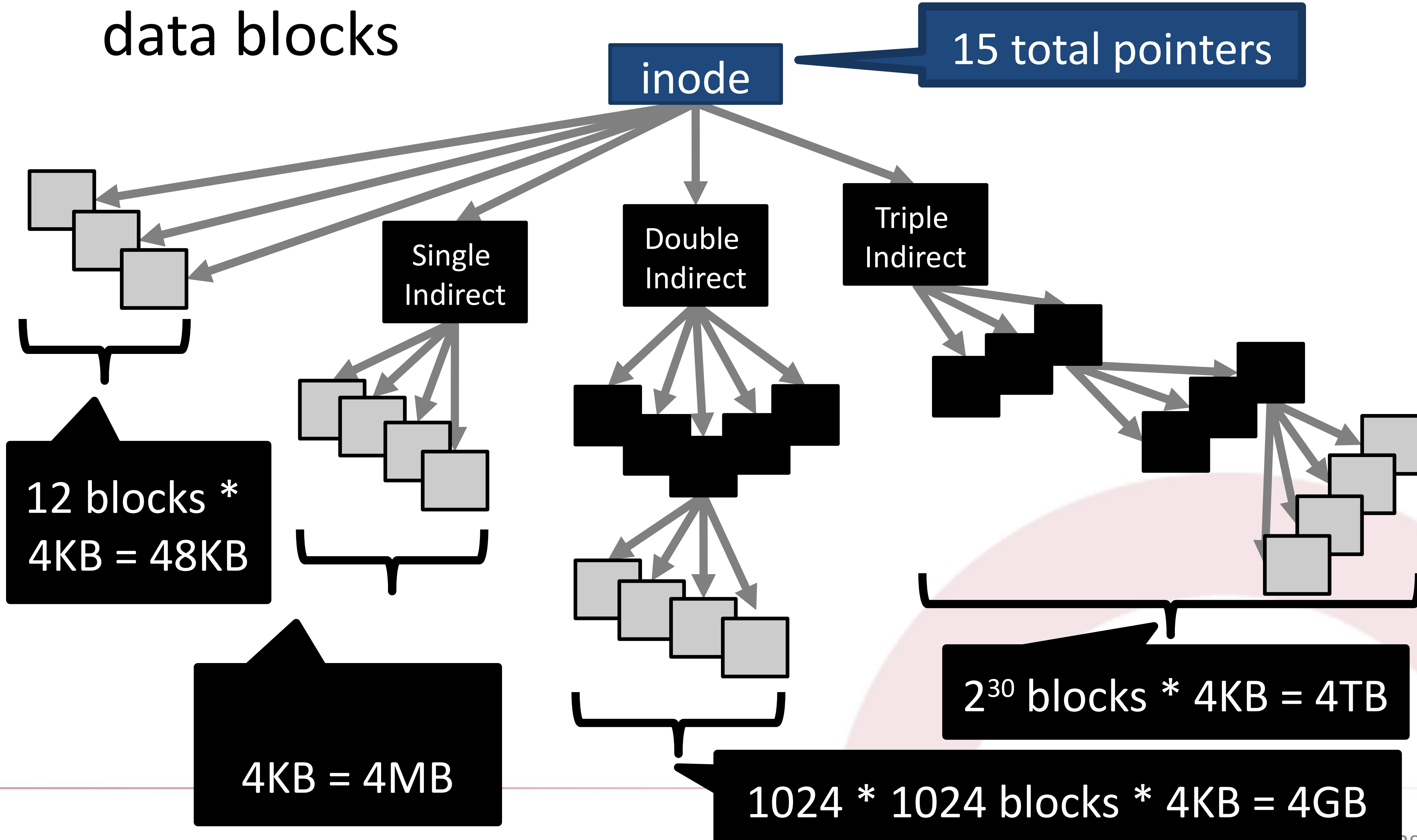

Direct
Data Blocks
Inode
Information
1
2
3
13
14
15
Blocks of
Pointers
1
2
128
1
2
128
Indirect
Data Blocks
Blocks of
Pointers
1
2
128
1
2
128
Double Indirect
Data Blocks

Ext2: content of an inode

Size (bytes)	Name	What is this field for?
2	mode	Read/write/execute?
2	uid	User ID of the file owner
4	size	Size of the file in bytes
4	time	Last access time
4	ctime	Creation time
4	mtime	Last modification time
4	dtime	Deletion time
2	gid	Group ID of the file
2	links_count	How many hard links point to this file?
4	blocks	How many data blocks are allocated to this file?
4	flags	File or directory? Plus, other simple flags
60	block	15 direct and indirect pointers to data blocks

inode Block Pointers

- ✓ Each *inode* is the root of an unbalanced tree of data blocks

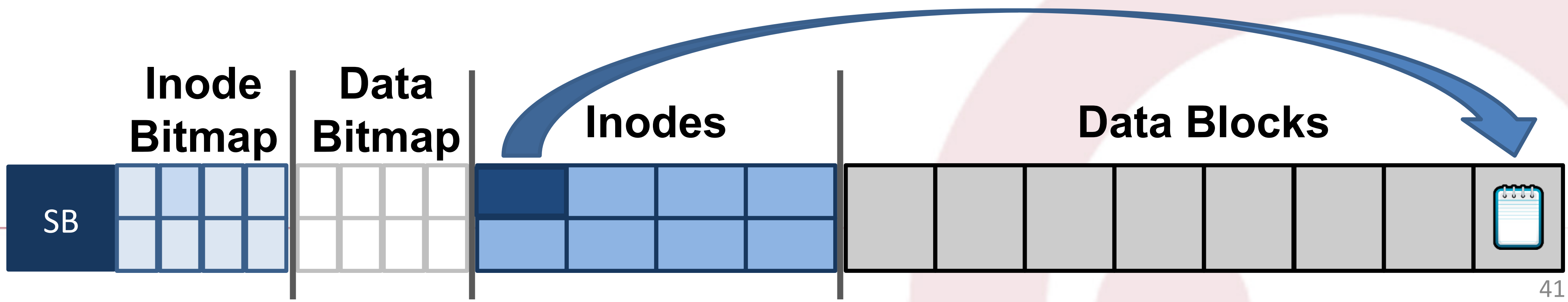


Advantages of *inodes*

- ✓ Optimized for file systems with many small files
  - Each *inode* can directly point to 48KB of data
  - Only one layer of indirection needed for 4MB files
- ✓ Faster file access
  - Greater meta-data locality
    - less random seeking
  - No need to traverse long, chained FAT entries
- ✓ Easier free space management
  - Bitmaps can be cached in memory for fast access
  - *inode* and data space handled independently

ext: The Good and the Bad

- The Good – ext file system (inodes) support:
  - All the typical file/directory features
  - More performance (less seeking) than FAT
- The Bad: poor locality
  - **ext** is optimized for a particular file size distribution
  - However, it is not optimized for spinning disks
  - inodes and associated data are far apart on the disk!



- ✓ Partitions and Mounting
- ✓ Basics (FAT)
- ✓ inodes and Blocks (ext)
- ✓ Block Groups (ext2)
- ✓ Journaling (ext3)
- ✓ Extents and B-Trees (ext4)
- ✓ Log-based File Systems

Status Check

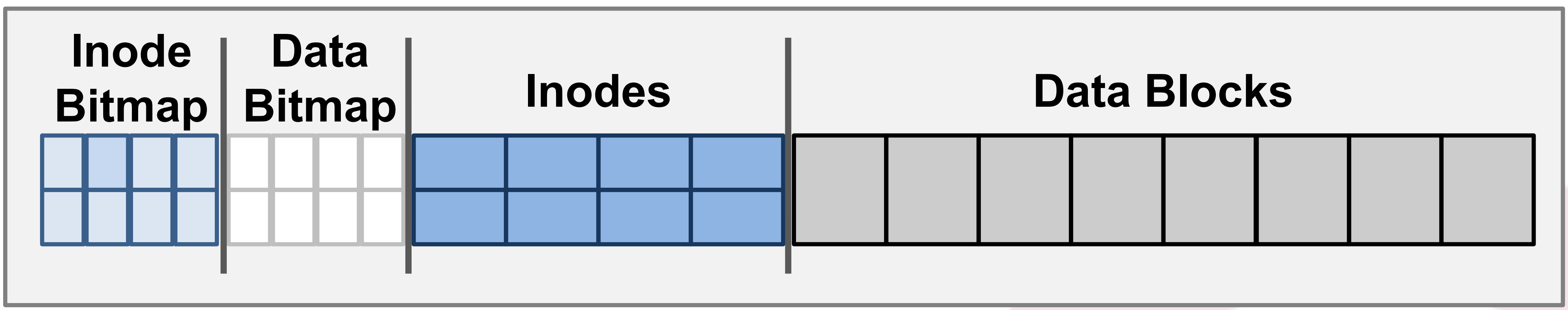
- ✓ At this point, we've moved from FAT to ext
  - – inodes are imbalanced trees of data blocks
  - – Optimized for the common case: small files
- ✓ Problem: ext has poor locality
  - – inodes are far from their corresponding data
  - – This is going to result in long seeks across the disk
- ✓ Problem: ext is prone to fragmentation
  - – ext chooses the first available blocks for new data
  - – No attempt is made to keep the blocks of a file contiguous

Fast File System (FFS)

- ✓ FFS developed at Berkeley in 1984
  - First attempt at a disk aware file system
  - i.e. optimized for performance on spinning disks
- ✓ Observation: *processes tend to access files that are in the same (or close) directories*
  - Spatial locality
- ✓ Key idea: place groups of directories and their files into cylinder groups
  - Introduced into **ext2**, called block groups

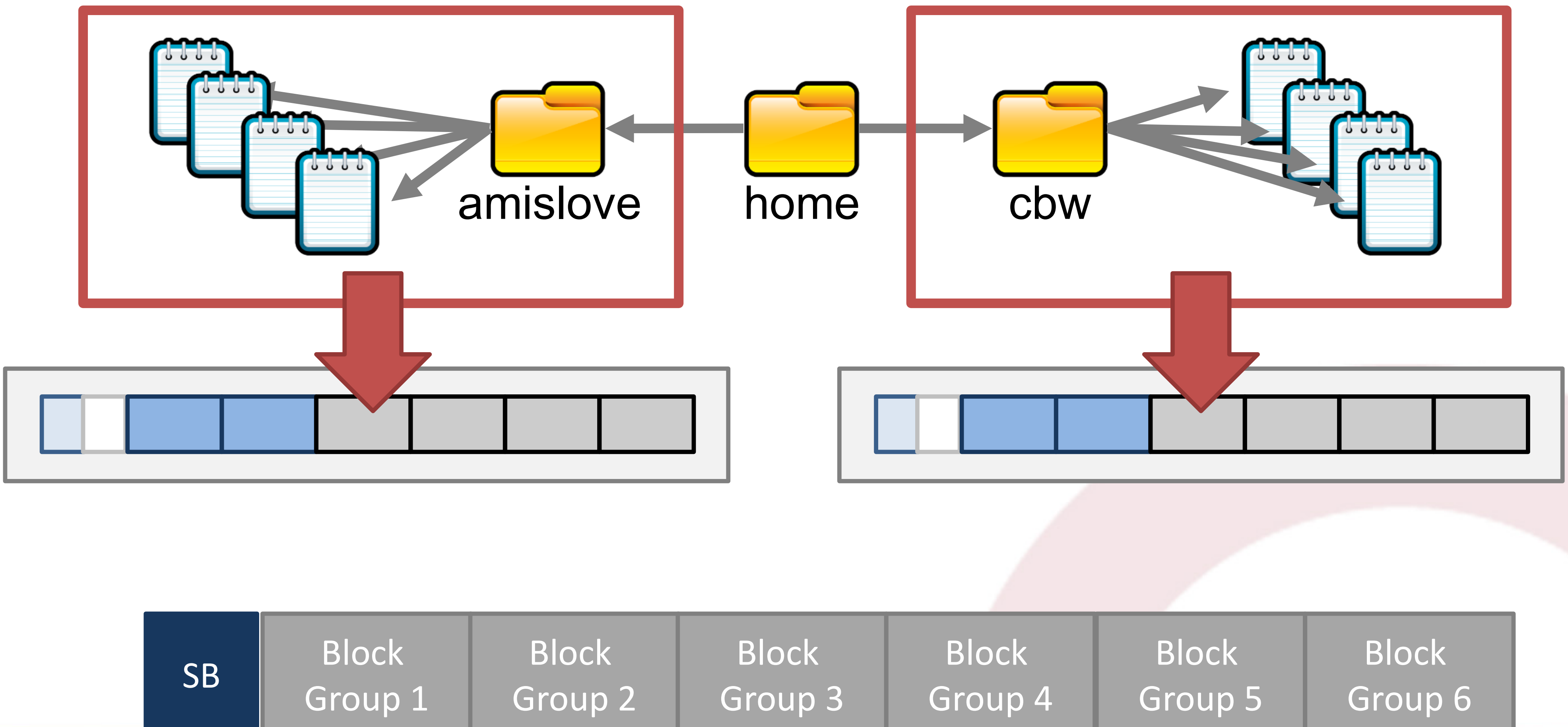
Block Groups

- ✓ In ext, there is a single set of key data structures
  - One data bitmap, one inode bitmap
  - One inode table, one array of data blocks
- ✓ In ext2, each block group contains its own key data structures



Allocation Policy

- ext2 attempts to keep related files and directories within the same block group



ext2: The Good and the Bad

- ✓ The good – ext2 supports:
  - All the features of ext…
  - … with even better performance (because of increased spatial locality)
- ✓ The bad
  - Large files must cross block groups
  - As the file system becomes more complex, the chance of file system corruption grows
    - E.g. invalid inodes, incorrect directory entries, etc.

- ✓ Partitions and Mounting
- ✓ Basics (FAT)
- ✓ inodes and Blocks (ext)
- ✓ Block Groups (ext2)
- ✓ Journaling (ext3)
- ✓ Extents and B-Trees (ext4)
- ✓ Log-based File Systems

Status Check

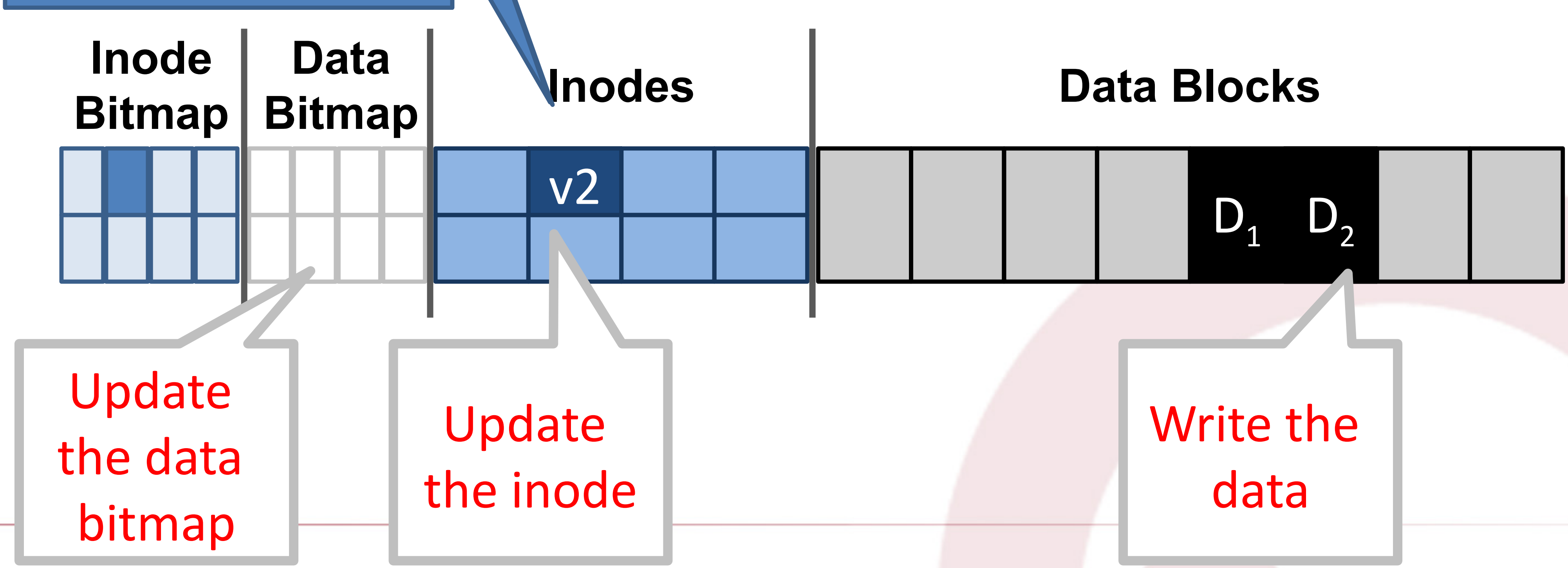
- ✓ At this point, we have a full featured file system
  - Directories
  - Fine-grained data allocation
- ✓ File system is optimized for spinning disks
  - *inodes* are optimized for small files
  - Block groups improve locality
- ✓ What's next?
  - Consistency and reliability

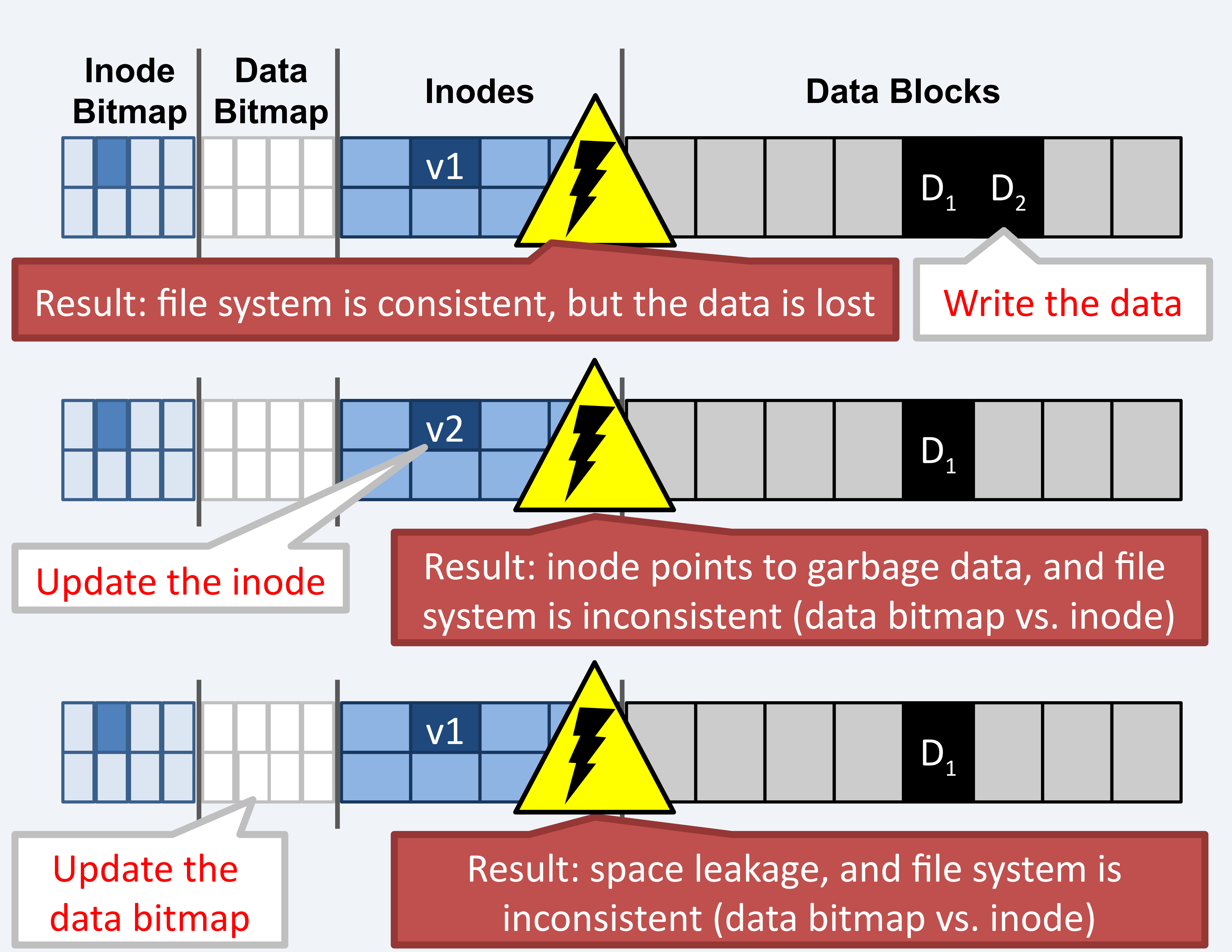
Maintaining Consistency

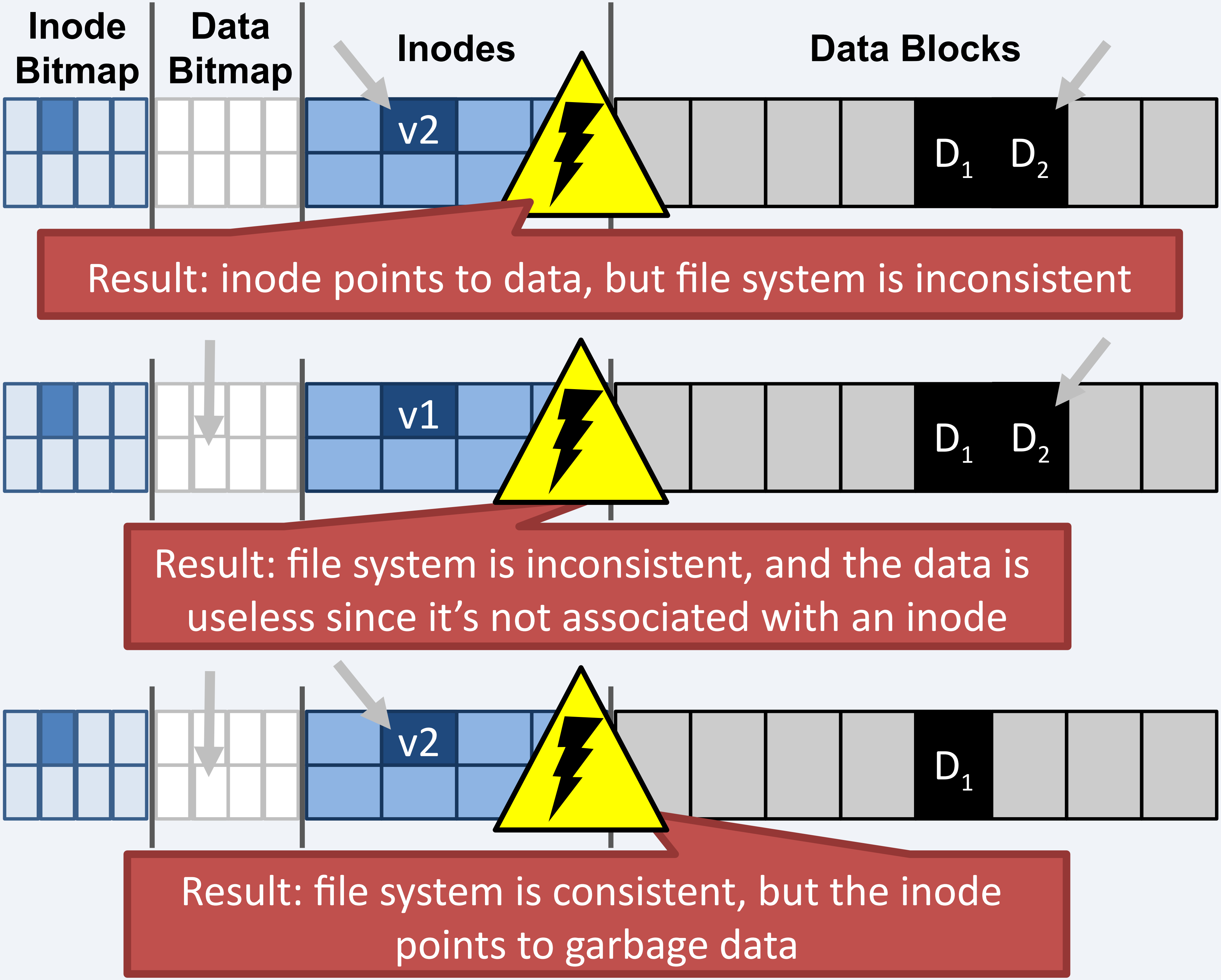
- ✓ Many operations results in multiple, independent writes to the file system
  - – Example: append a block to an existing file
  1. Update the free data bitmap
  2. Update the inode
  3. Write the user data
- ✓ What happens if the computer crashes in the middle of this process?

File Append Example

- ✓ These three operations can potentially be done in any order
- ✓ … but the system can crash at any time




Inode Bitmap
Data Bitmap
Inodes
Data Blocks
v1
D_1 D_2
Result: file system is consistent, but the data is lost
Write the data
v2
D_1
Update the inode
Result: inode points to garbage data, and file system is inconsistent (data bitmap vs. inode)
v1
D_1
Update the data bitmap
Result: space leakage, and file system is inconsistent (data bitmap vs. inode)


Inode Bitmap
Data Bitmap
Inodes
Data Blocks
v2
D1 D2
Result: inode points to data, but file system is inconsistent
v1
D1 D2
Result: file system is inconsistent, and the data is useless since it's not associated with an inode
v2
D1
Result: file system is consistent, but the inode points to garbage data

The Crash Consistency Problem

- ✓ The disk guarantees that sector writes are *atomic*
  - – No way to make multi-sector writes atomic
- ✓ How to ensure consistency after a crash?
  1. Don't bother to ensure consistency
     - Accept that the file system may be inconsistent after a crash
     - Run a program that fixes the file system during bootup
     - File system checker (*fsck*)
  2. Use a transaction log to make multi-writes atomic
     - Log stores a history of all writes to the disk
     - After a crash the log can be "replayed" to finish updates
     - Journaling file system

Approach 1: File System Checker

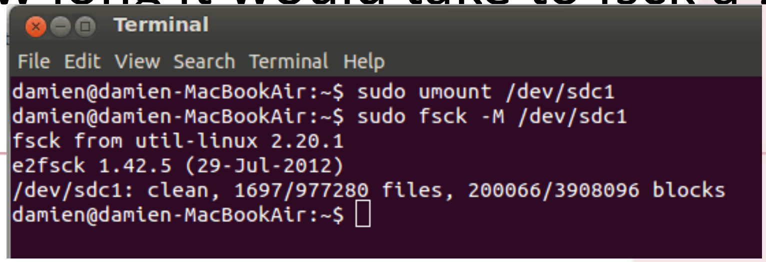
- ✓ Key idea: fix inconsistent file systems during bootup
  - Unix utility called *fsck* (*chkdsk* on Windows)
  - Scans the entire file system multiple times, identifying and correcting inconsistencies
- ✓ Why during bootup?
  - No other file system activity can be going on
  - After fsck runs, bootup/mounting can continue

fsck Tasks

- ✓ **Superblock:** validate the superblock, replace it with a backup if it is corrupted
- ✓ **Free blocks and inodes:** rebuild the bitmaps by scanning all inodes
- ✓ **Reachability:** make sure all inodes are reachable from the root of the file system
- ✓ **inodes:** delete all corrupted inodes, and rebuild their link counts by walking the directory tree
- ✓ **directories:** verify the integrity of all directories
- ✓ … and many other minor consistency checks

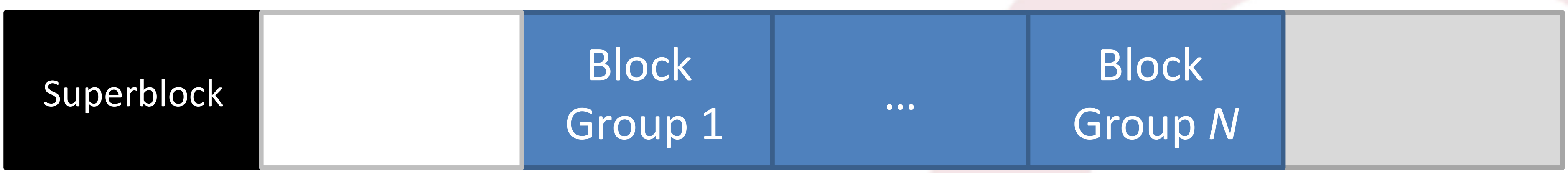
fsck: the Good and the Bad

- ✓ Advantages of *fsck*
  - Doesn't require the file system to do any work to ensure consistency
  - Makes the file system implementation simpler
- ✓ Disadvantages of *fsck*
  - Very complicated to implement the *fsck* program
    - Many possible inconsistencies that must be identified
    - Many difficult corner cases to consider and handle
  - *fsck* is **super slow**
    - Scans the entire file system multiple times
    - Imagine how long it would take to fsck a 10 TB disk…



Approach 2: Journaling

- ✓ Problem: *fsck* is slow because it checks the entire file system after a crash
  - What if we knew where the last writes were before the crash, and just checked those?
- ✓ Key idea: make writes transactional by using a write-ahead log (WAL)
  - Commonly referred to as a journal
- ✓ Ext3 and NTFS use *journaling*

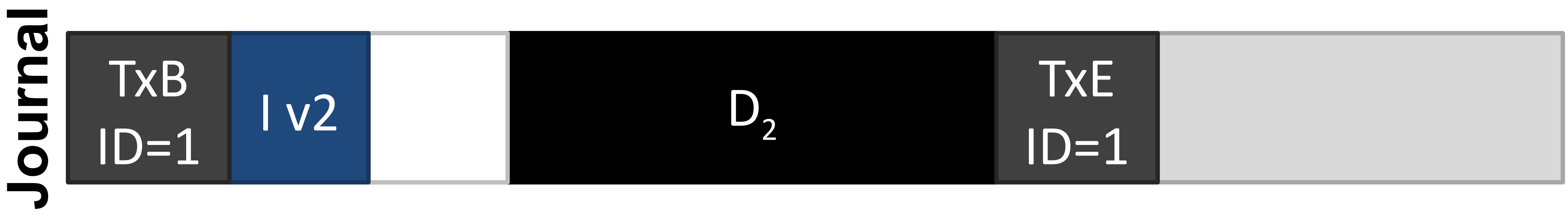


Write-Ahead Log (WAL)

- ✓ Key idea: writes to disk are first written into a log
  - – After the log is written, the writes execute normally
  - – In essence, the log records transactions
- ✓ What happens after a crash…
  - – If the writes to the log are interrupted?
    - The transaction is incomplete
    - The user's data is lost, but the file system is consistent
  - – If the writes to the log succeed, but the normal writes are interrupted?
    - The file system may be inconsistent, but…
    - The log has exactly the right information to fix the problem

Data Journaling Example

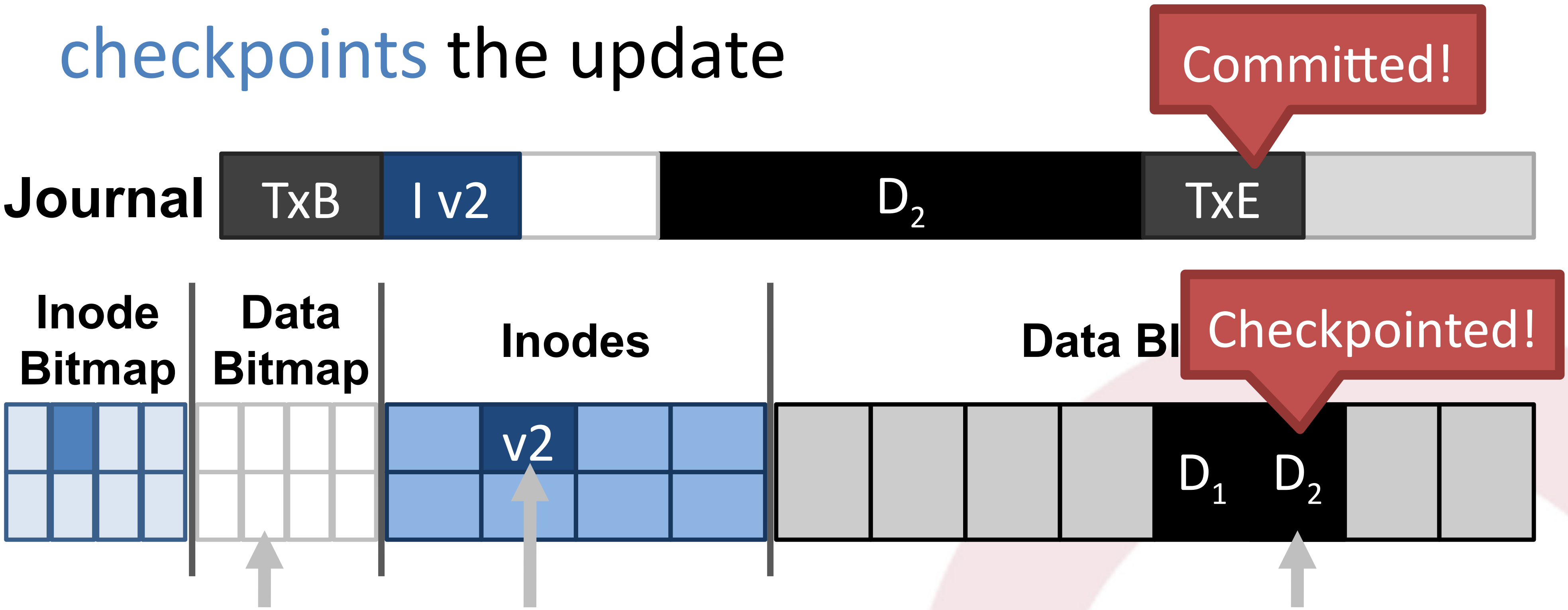
- ✓ Assume we are appending to a file
  - – Three writes: inode v2, data bitmap v2, data D_2
- ✓ Before executing these writes, first log them



1. Begin a new transaction with a unique ID=*k*
2. Write the updated meta-data block(s)
3. Write the file data block(s)
4. Write an end-of-transaction with ID=*k*

Commits and Checkpoints

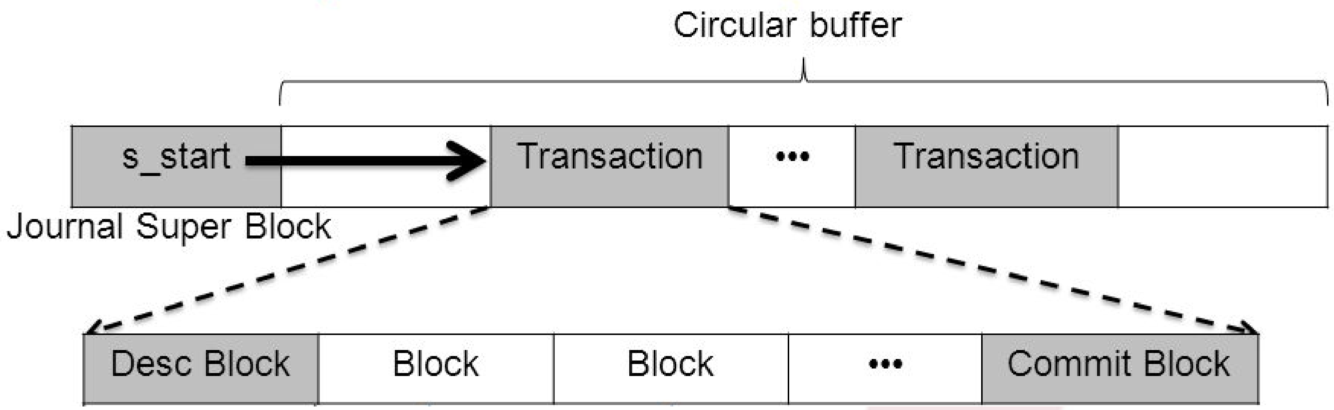
- ✓ We say a transaction is committed after all writes to the log are complete
- ✓ After a transaction is committed, the OS checkpoints the update



- Final step: free the checkpointed transaction

Journal Implementation

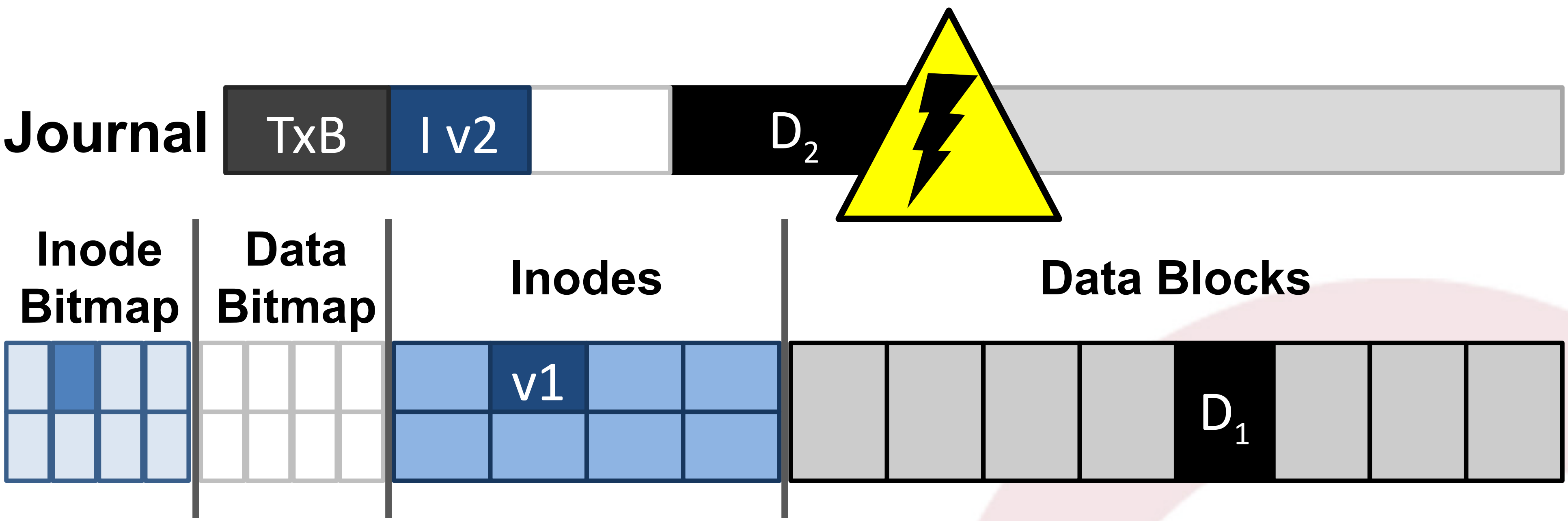
- ✓ Journals are typically implemented as a circular buffer
  - Journal is **append-only**
- ✓ OS maintains pointers to the front and back of the transactions in the buffer
  - As transactions are freed, the back is moved up
- ✓ Thus, the contents of the journal are never deleted, they are just overwritten over time



https://bit.ly/2ZBPGCn

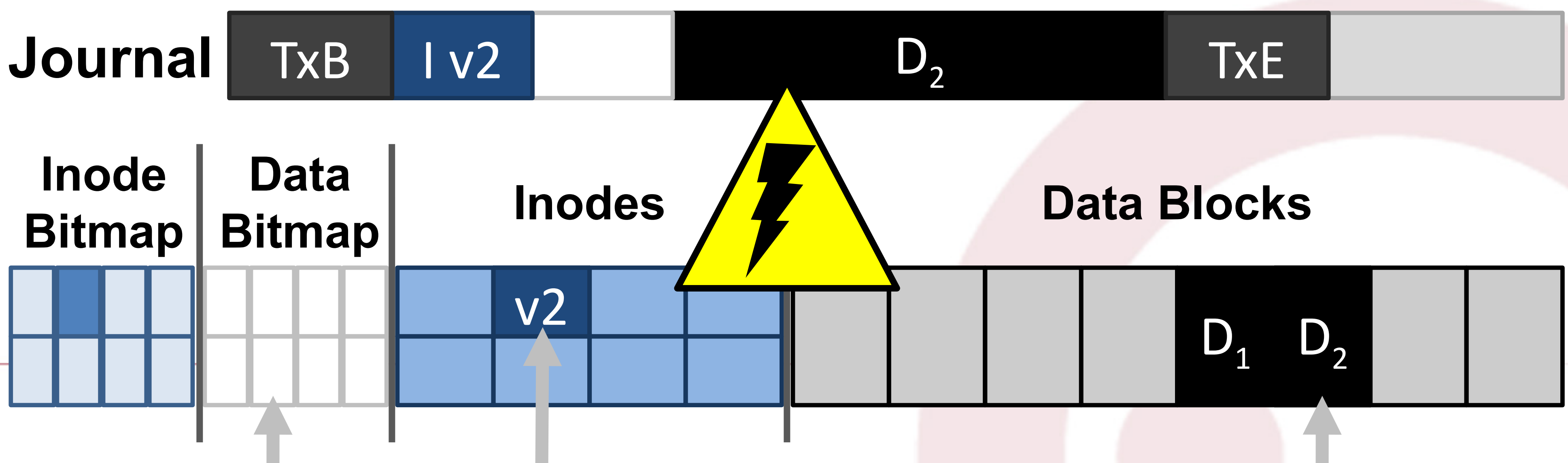
Crash Recovery (1)

- What if the system crashes during logging?
  - If the transaction is not committed, data is lost
  - But, the file system remains consistent



Crash Recovery (2)

- What if the system crashes during the checkpoint?
  - File system may be inconsistent
  - During reboot, transactions that are committed but not free are replayed in order
  - Thus, no data is lost and consistency is restored



Journaling: The Good and the Bad

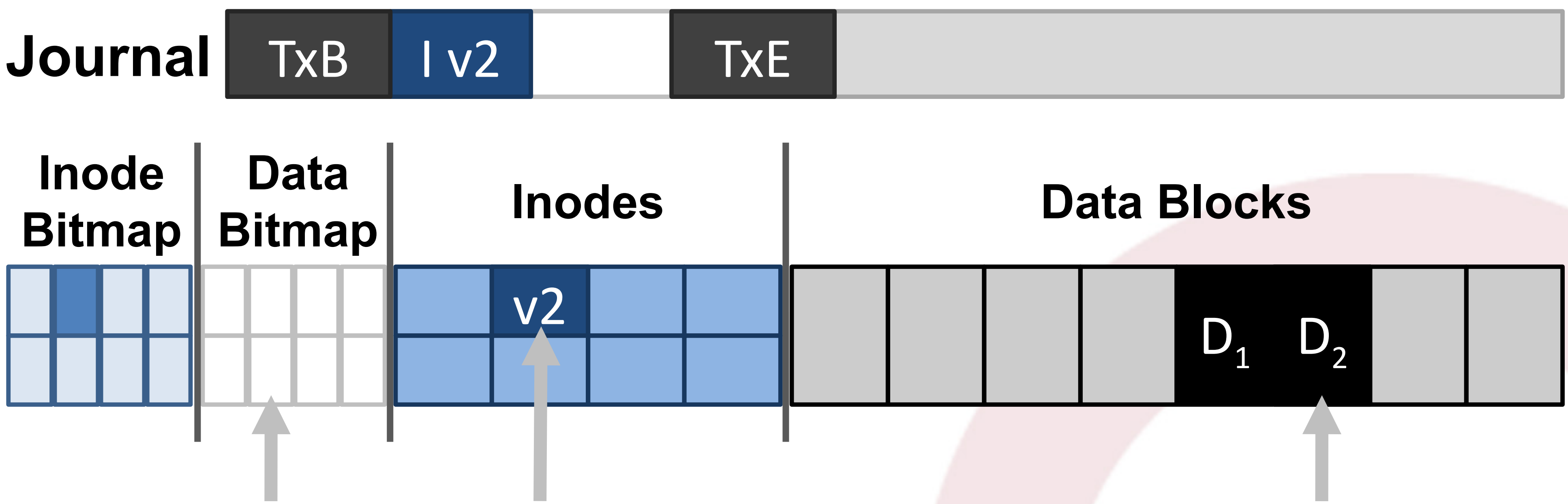
- ✓ Advantages of journaling
  - – Robust, fast file system recovery
    - ▪ No need to scan the entire journal or file system
  - – Relatively straight forward to implement
- ✓ Disadvantages of journaling
  - – Write traffic to the disk is doubled
    - ▪ Especially the file data, which is probably large
  - – Deletes are very hard to correctly log

Making Journaling Faster

- Journaling adds a lot of **write overhead**
- OSes typically batch updates to the journal
  - Buffer sequential writes in memory, then issue one large write to the log
  - **Example:** ext3 batches updates for 5 seconds
- Tradeoff between performance and persistence
  - Long batch interval = fewer, larger writes to the log
    - Improved performance due to large sequential writes
  - But, if there is a crash, everything in the buffer will be lost

Meta-Data Journaling

- ✓ The most expensive part of data journaling is writing the file data twice
  - Meta-data is small (~1 sector), file data is large
- ✓ ext3 implements **meta-data journaling**



Journaling Wrap-Up

- ✓ Today, most OSes use journaling file systems
  - ext3/ext4 on Linux
  - NTFS on Windows
- ✓ Provides excellent crash recovery with relatively low space and performance overhead
- ✓ Next-gen OSes will likely move to file systems with copy-on-write semantics
  - *btrfs* and *zfs* on Linux

- ✓ Partitions and Mounting
- ✓ Basics (FAT)
- ✓ inodes and Blocks (ext)
- ✓ Block Groups (ext2)
- ✓ Journaling (ext3)
- ✓ Extents and B-Trees (ext4)
- ✓ Log-based File Systems

Status Check

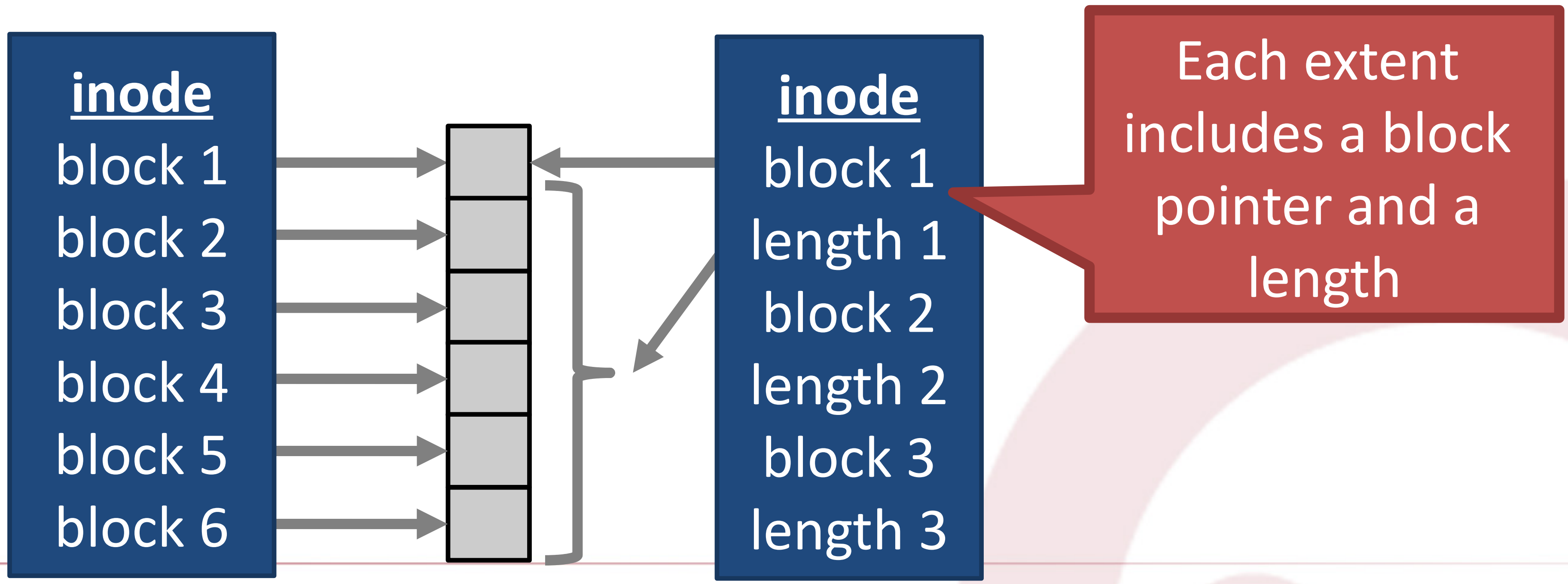
- ✓ At this point:
  - – We not only have a fast file system
  - – But it is also resilient against corruption
- ✓ What's next?
  - – More efficiency improvements!

Revisiting inodes

- ✓ Recall: inodes use **indirection** to acquire additional blocks of pointers
- ✓ Problem: inodes are not efficient for large files
  - – Example: for a 100MB file, you need 25600 block pointers (assuming 4KB blocks)
- ✓ This is unavoidable if the file is 100% fragmented
  - – However, what if large groups of blocks are contiguous?

From Pointers to Extents

- ✓ Modern file systems try hard to minimize fragmentation
  - Since it results in many seeks, thus low performance
- ✓ Extents are better suited for contiguous files



escola superior de tecnologia e gestão
instituto politécnico de leiria

Implementing Extents

- ✓ ext4 and NTFS use extents
- ✓ ext4 inodes include 4 extents instead of block pointers
  - Each extent can address at most 128MB of contiguous space (assuming 4KB blocks)
  - If more extents are needed, a data block is allocated
  - Similar to a block of indirect pointers

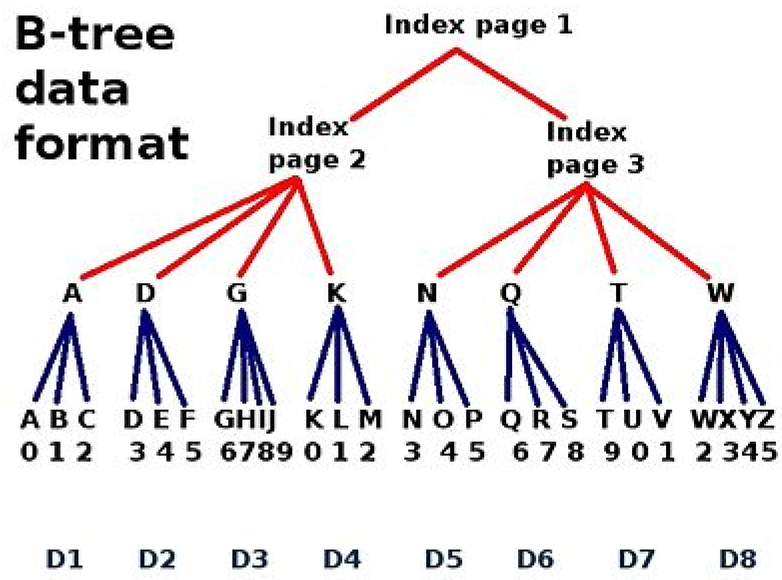
Revisiting Directories

- ✓ In ext, ext2, and ext3, each directory is a file with a list of entries
  - – Entries are not stored in sorted order
  - – Some entries may be blank, if they have been deleted
- ✓ Problem: searching for files in large directories takes O(n) time
  - – Practically, you can't store >10K files in a directory
  - – It takes way too long to locate and open files

Ext4 optimization for directories >>

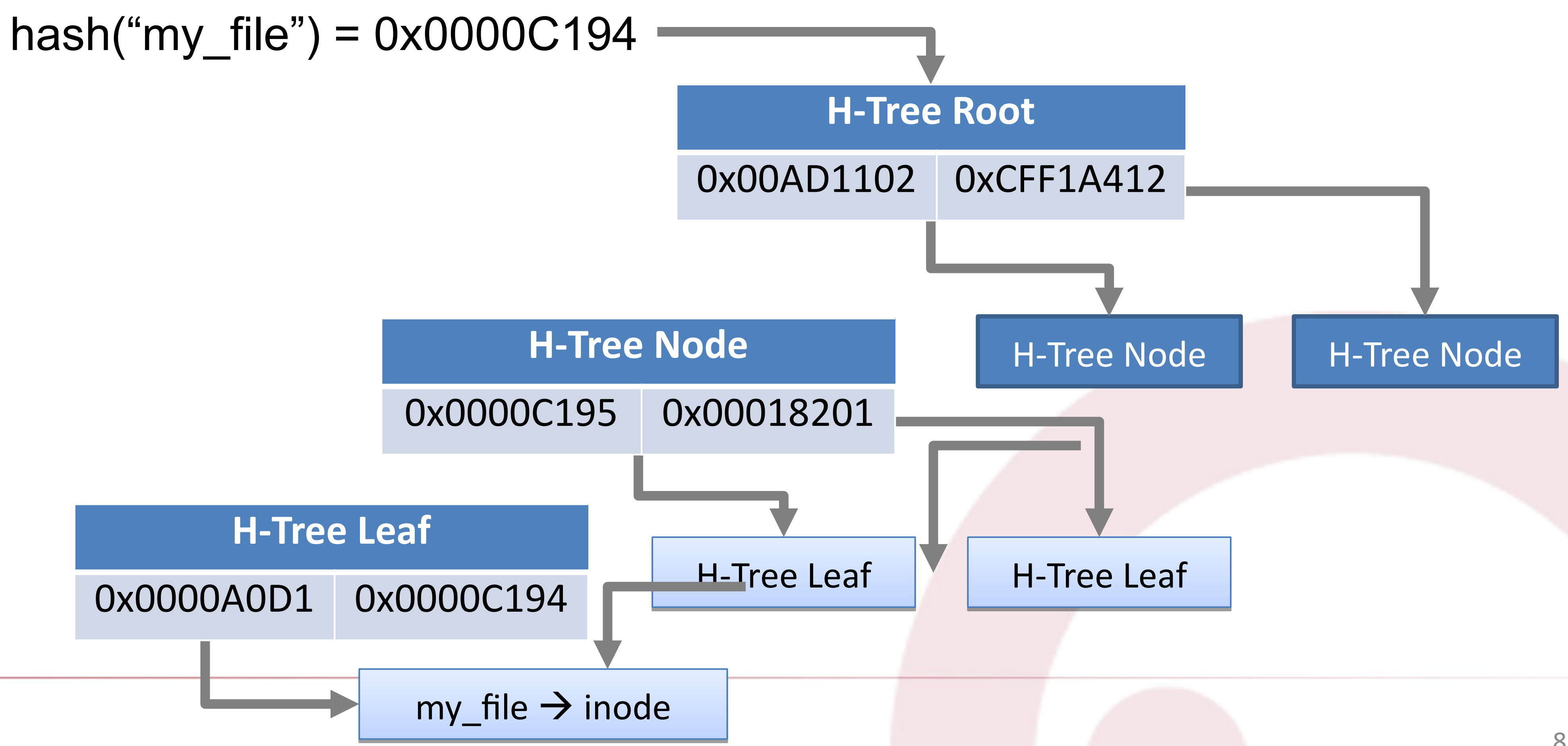
From Lists to B-Trees

- ext4 and NTFS encode directories as B-Trees to improve lookup time to O(log N)
- A B-Tree is a type of balanced tree that is optimized for storage on disk
  - Items are stored in sorted order in blocks
  - Each block stores between *m* and 2*m* items



Example B-Tree

- ext4 uses a B-Tree variant known as a H-Tree
  - The *H* stands for *hash* (sometime called B+Tree)
- Suppose you try to open("my_file", "r")



ext4: The Good and the Bad

- ✓ The good – ext4 (and NTFS) supports:
  - All of the basic file system functionality we require
  - Improved performance from ext3’s block groups
  - Additional performance gains from extents and B-Tree directory files
- ✓ The bad:
  - ext4 is an incremental improvement over ext3
  - Next-gen file systems have even nicer features
    - Copy-on-write semantics (btrfs and ZFS)

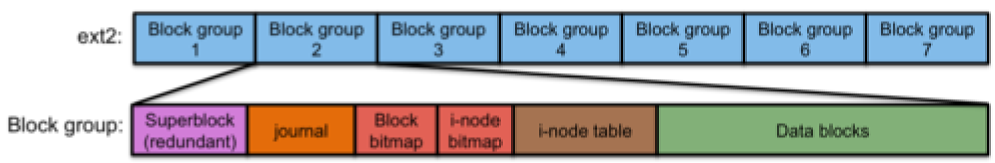
Ext4 – some numbers

- Linux supports several file systems: ext2, ext3 e ext4, XFS, JFS, ReiserFS, btrfs
  - ext4 – 4th extended filesystem
    - Also use in android
  - Limits
    - Partitions up to 1 ExaByte (EiB)
    - Max size for file 16 TiB
    - Maximum 64000 entries per directory
      - ext3 only supports 32000 entries per directory

Ext4 File System

Limit	ext3	ext4	XFS
max file system size	16 TiB	16 TiB	16 EiB
max file size	2 TiB	16 TiB	8 EiB
max extent size	4 kiB	128 MiB	8 GiB
max extended attribute size	4 kiB	4 kiB	64 kiB
max inode number	2^{32}	2^{32}	2^{64}

http://linuxmantra.com/2013/09/xfs-in-rhel6.html



http://www.cs.rutgers.edu/~pxk/416/notes/13-fs-studies.html

MARS SPIRIT ROVER PROBLEMS (2004)

Mars Rover Spirit’s problems (2004) #1

- ✓ Problem within the flash memory
  - – Two autonomous vehicle (“rovers”) of NASA
    - Spirit & Opportunity
  - – Spirit had problems in 2004.01.21
    - Anomalous behavior
    - *debugging* done at million of kms...
  - – Root cause: trouble with the file system’s software of the flash emory
    - Two configuration errors amplified the crash/reboot situation caused by the main error
      - – Typical scenario of “*chain of errors*”

Mars Rover Spirit's problems (2004) #2

✓ Troubles in contacting the Spirit rover
  – Many crash/reboot cycles

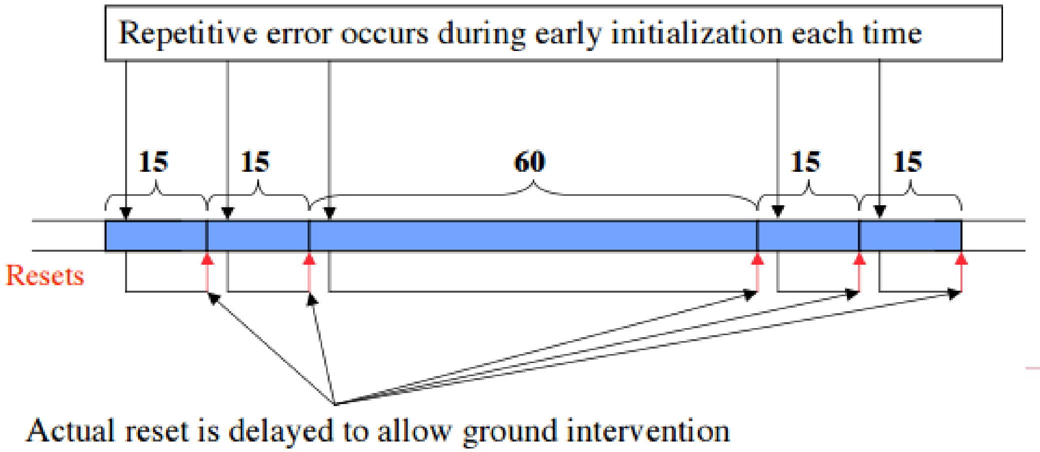
- Lack of data in the flash memory pointed out for the occurrence of one (or more) of the following situations:
  – Flash memory
  – File systems that manages the flash memory
  – Software that read data and prepare them to be sent to Earth

So, what was wrong? >>

Mars Rover Spirit's problems (2004) #3

✓ Main problem

- The library which provides the file system (DOS's like) holds in memory the whole structure of the file system:
  - **i)** directories and files
  - **ii)** the structure also holds the deleted files (these ones are labeled with the E5 byte)
- The structure is created when the OS boot
  - The structure is too large, filling up the memory
  - The lack of space in memory triggers the hardware watchdog which reboots the OS
  - The system reboots endlessly...



Mars Rover Spirit's problems (2004) #4

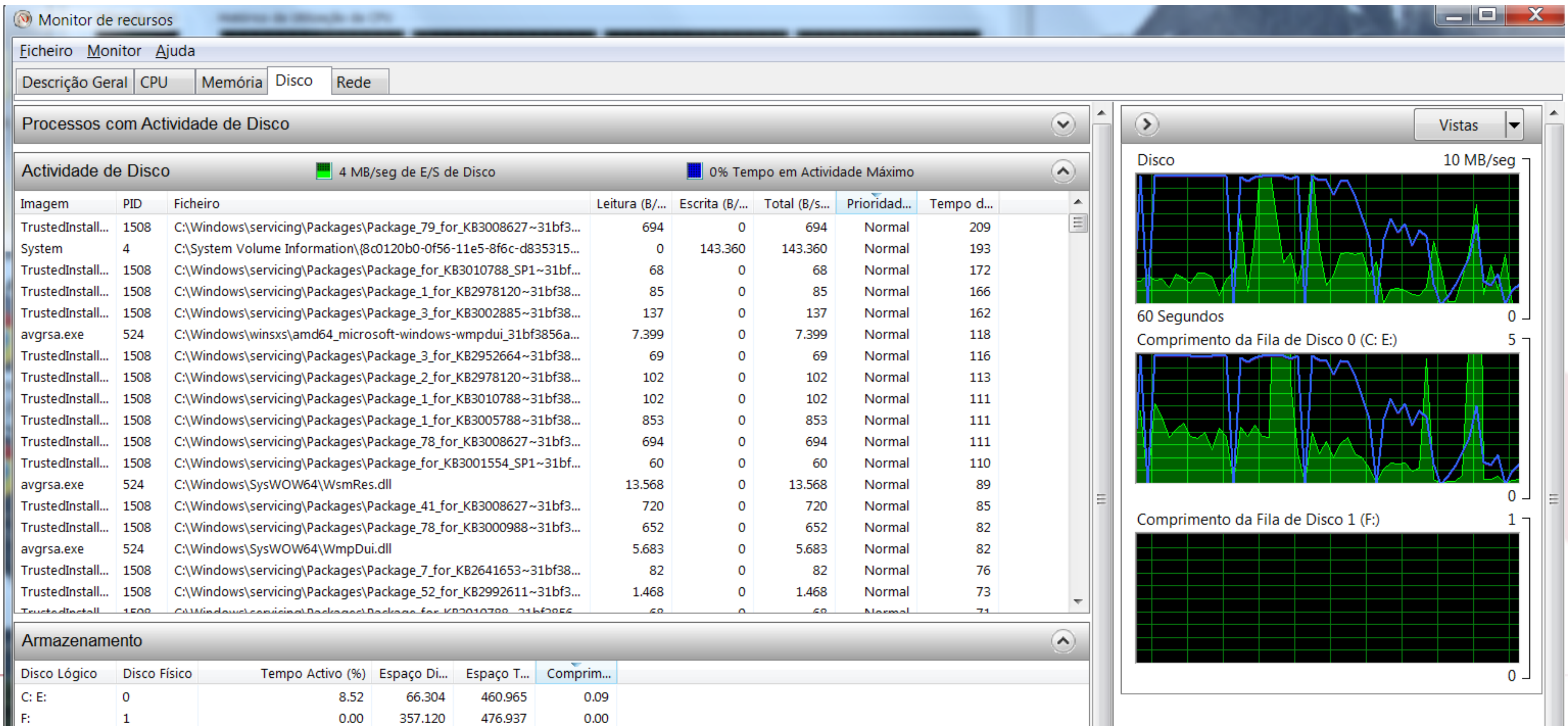
- ✓ After many analysis (and days!), the file system was formatted
  - – Problem solved!

Source: "The Mars Rover Spirit FLASH Anomaly", Glenn Reeves, Tracy Neilson, 2004 (https://www.cs.princeton.edu/courses/archive/fall11/cos109/mars.rover.pdf)

File system activity

✓ Resource monitor (>= windows 7)

Gestor de tarefas do Windows → Monitor de Recursos...



Utility `df`

- Unix – utility `df`
  - Option `-T` shows the file system type

```
user@ubuntu: ~
File  Edit  Tabs  Help
user@ubuntu:~$ df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
/dev/sda2      ext3      20414332 2952088  16418584  16% /
none           tmpfs            4       0         4   0% /sys/fs/cgroup
udev           devtmpfs    503420      12    503408   1% /dev
tmpfs          tmpfs       102604     844    101760   1% /run
none           tmpfs         5120       0      5120   0% /run/lock
none           tmpfs       513004       0    513004   0% /run/shm
none           tmpfs       102400      16    102384   1% /run/user
/dev/sdb1      ext3     103079200  519600  97316824   1% /home
```

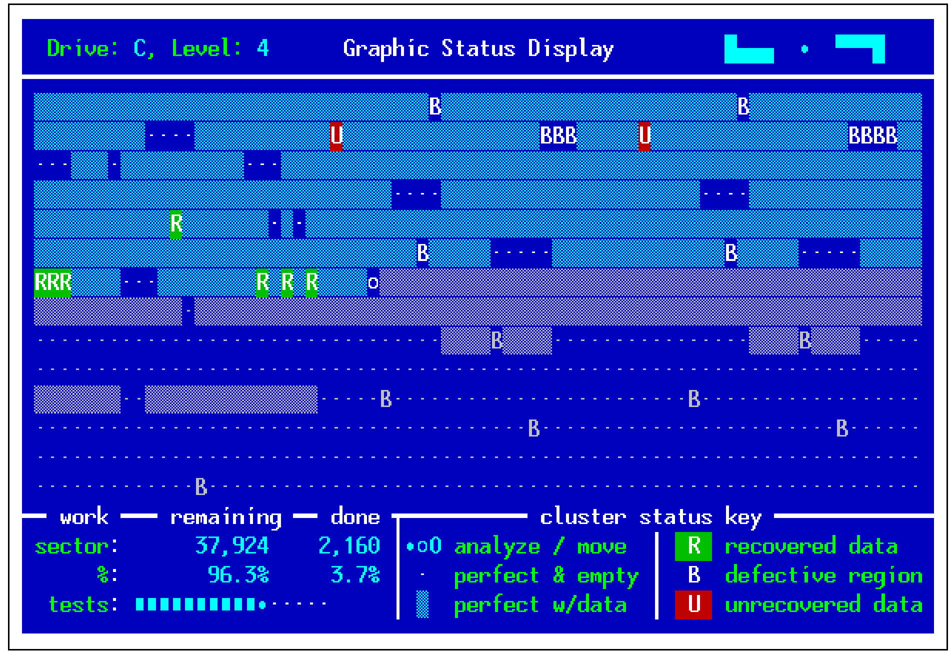
File system maintenance (1)

- Formatting
  - Create the structures of the file system: super block, “inodes”
  - `commands`
    - `format` (Win32), `mke2fs` (Linux)
- Non usable sectors
  - Almost all disks have “broken” sectores
    - Wear of material combined with high density of modern disks
    - `scandisk` (Win32) or `badblocks` (Linux)
    - The blocks are added to the list of “*bad-blocks*”
      - Nowadays, the disks are the one managing their own bad blocks

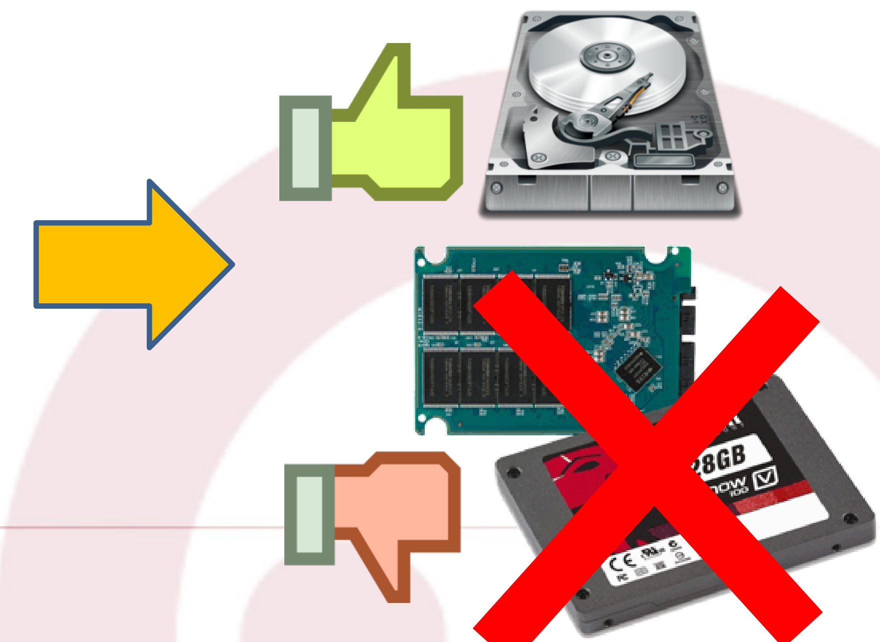
IPL
escola superior de tecnologia e gestão
instituto politécnico de leiria

File system maintenance (2)

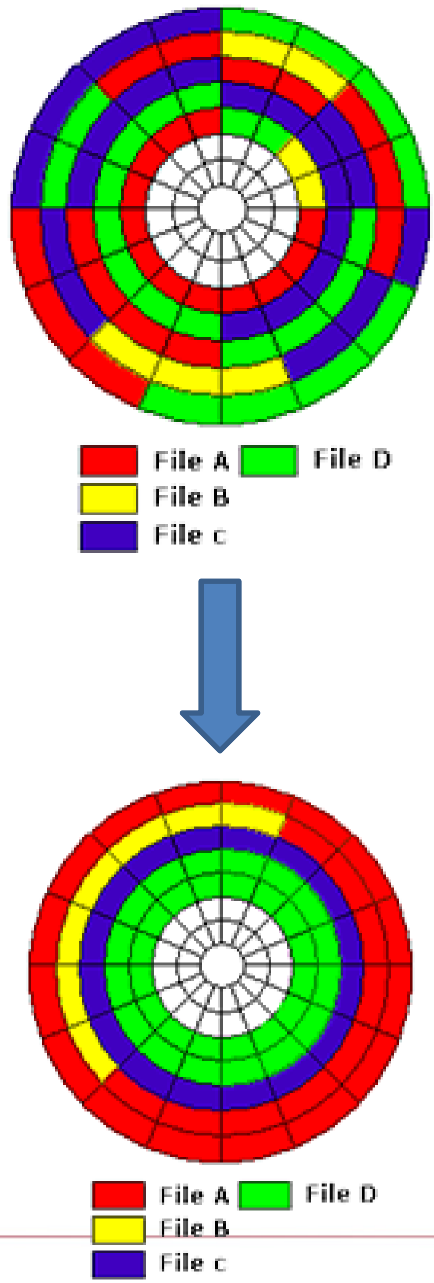
- ✓ Recovery
  - "lost+found" (unix), correcting of file systems, etc.
  - Use of external tools
    - Spinrite

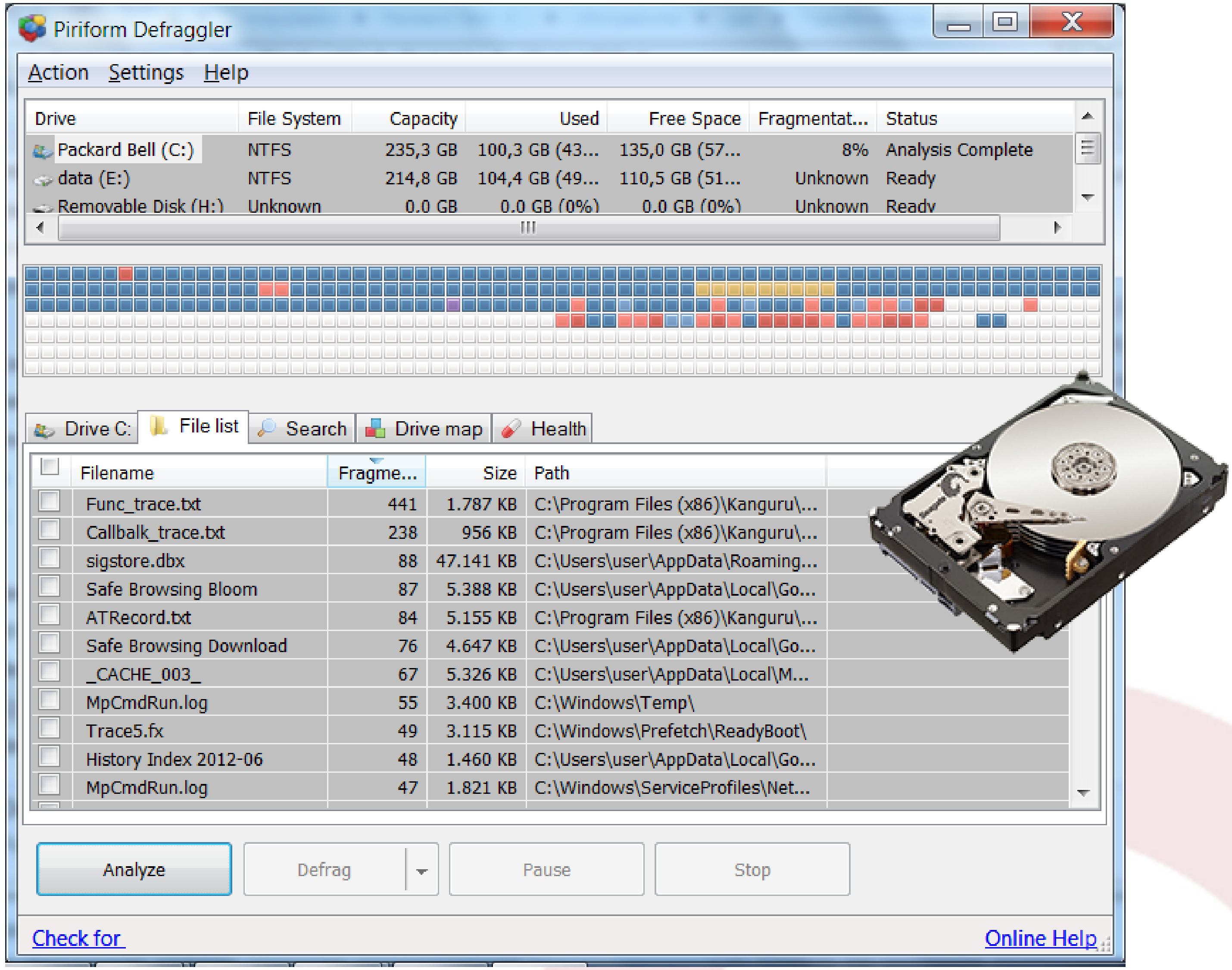


- ✓ Defragment
  - Reorganize the data blocks to maximize the sequentiality of the blocks of the files
  - Important for HDD, but not recommended for SSD
    - No advantages for SSD, on the contrary, "defragmenting" would cause unneeded wear of the disk



Defragmenting (windows)





http://learn.caconnects.org/mod/resource/view.php?id=138

Sistema ficheiro *Linux* – “procfs” (1)

- “procfs” file system
  - Pseudo file system
    - Mounted in **/proc**
  - The content of files/directories does not exist in disk. It exists on memory
- It acts as “interface” to kernel stats and kernel parameters
- As data are available in files and directory, procfs can be accessed with traditional file tools
  - Example
    - cat /proc/interrupts
    - cat /proc/cpuinfo

IPL
escola superior de tecnologia e gestão
instituto politécnico de leiria

Sistema ficheiro Linux – “procfs” (2)

✓ cat /proc/cpuinfo

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 8
model name      : Pentium III (Coppermine)
stepping        : 10
cpu MHz         : 930.335
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 sep mtrr pge
 mca cmov pat pse36 mmx fxsr sse
bogomips        : 1862.26
```

/proc/cpuinfo

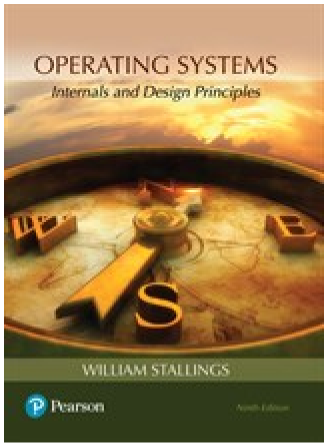
✓ cat /cpu/procinfo

```
processor	: 0
vendor_id	: GenuineIntel
cpu family	: 6
model	: 42
model name	: Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz
stepping	: 7
microcode	: 0x29
cpu MHz		: 2301.000
cache size	: 3072 KB
physical id	: 0
siblings	: 1
core id		: 0
cpu cores	: 1
apicid		: 0
initial apicid	: 0
fdiv_bug	: no
f00f_bug	: no
coma_bug	: no
fpu		: yes
fpu_exception	: yes
cpuid level	: 13
wp		: yes
flags	: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse sse2 ss nx rdtscp lm constant_tsc arch_perfmon pebs bts tsc_reliable nonstop_tsc aperfmperf pni pclmulqdq ssse3 cx16 sse4_1 sse4_2 popcnt aes lahf_lm epb dtherm ida arat pln pts
bugs		:
bogomips	: 4602.00
clflush size	: 64
cache_alignment	: 64
address sizes	: 36 bits physical, 48 bits virtual
power management:
```

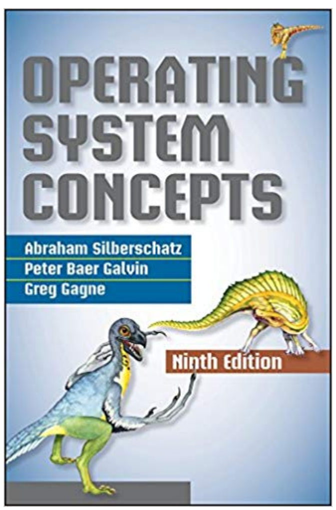
File systems

- ✓ Linked to DOS/Windows
  - FAT12,FAT16,FAT32, FAT64/exFAT
  - NTFS
- ✓ Linked to Unix
  - ext2, ext3, ext4, reiserfs, btrfs, jfs, zfs
- ✓ Linked to macOS
  - MFS (Macintosh File System)
  - HFS (Hierarchical File System) e HFS+
  - HFSX (dispositivos móveis)
  - APFS (Apple File System) (2016)

Bibliography



- Chapters 11 & 12 of "Operating Systems – Internals and Design Principles", William Stallings, 9th edition, 2018



- Chapters 12 of "Operating Systems Concepts", A. Silberschatz, 9th edition, 2016