Filesystem and Storage Subsystems

Overview

- Introduction to storage and block devices
- Virtual File system
- The Block I/O Layer
- Process Address Space
- Page cache and Page Writeback
- Case study - ToyFS filesystem

What is a File?

Introduction to storage and block devices

What are block devices?

- Storage devices are accessible through sector/block addresses
  - HDDs, SSDs, DVD/Blu-Ray etc
- Using specific communication protocols to access
  - IDE, SCSI, SATA, SAS, etc

The sector as the fundamental unit

- Storage’s smallest addressable unit
- Come by many names
  - Sectors, physical block size, I/O blocks...
- May come in different sizes depending on the media
  - 512 Bytes
  - 4096 Bytes (Advanced Format)
  - 2KiB - 64Kib (Blu-Rays)

Logical Blocks

- Aggregation of one or more consecutive **physical sectors**
- Smallest “logical” addressable unit for logical volumes
  - RAID arrays
  - LVMs volumes (depending on volume type)
  - other volume managers.

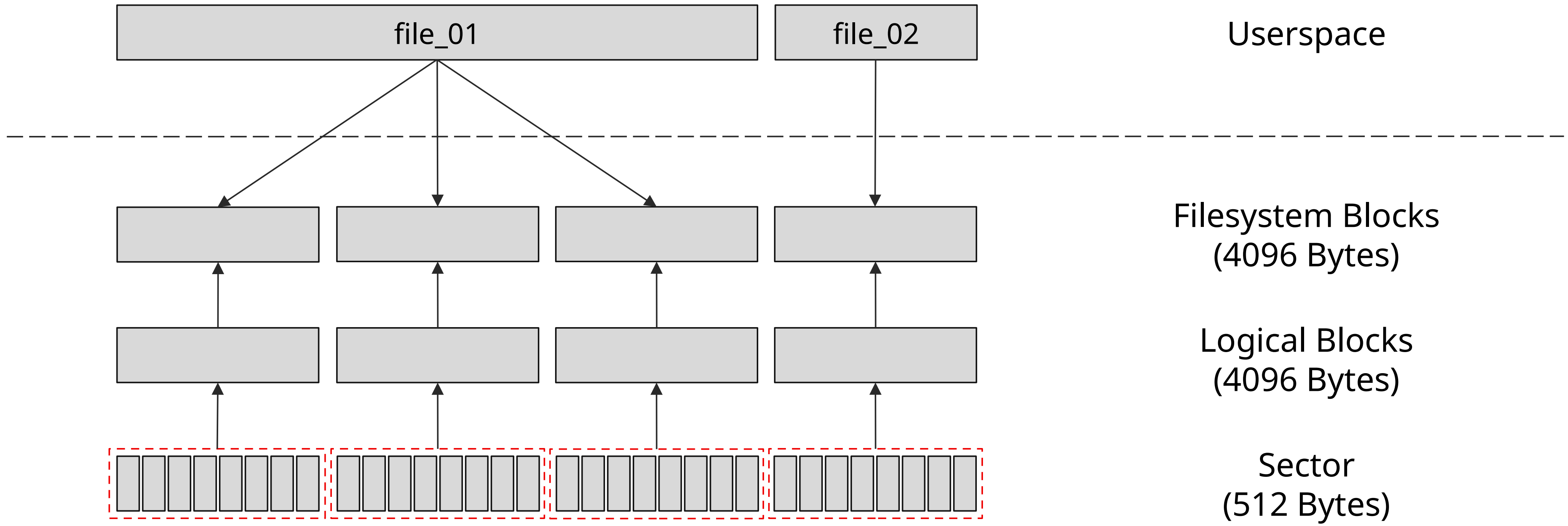
Filesystem Blocks

- Aggregation of one or more consecutive **logical blocks** or **physical sectors**, depending on the underlying media
- Smallest “logical” addressable unit for:
  - Filesystems
  - User applications

Files

- A container of data
  - An “unstructured” array of bytes, nothing more, nothing less
  - Stored on top of **filesystem blocks** (for disk-based filesystems)
- Abstraction used by applications and users to store and retrieve data

“Bringing them all together...”



I/O operations vs File Operations

- I/O operations (**IOPS**)
  - Storage Unit Commands
  - 95% READ and WRITE

- File Operations (**OPS**)
  - File-related operations
    - open(), close()
    - read(), write()
    - stat(), lseek()

Important things to keep in mind...

- Physically, any write other than a sector **IS NOT ATOMIC**
- The Read-Modify-Write curse
- Torn writes
- Storages are usually capable of reading and writing sectors in batches

Virtual File System

The most important subsystem

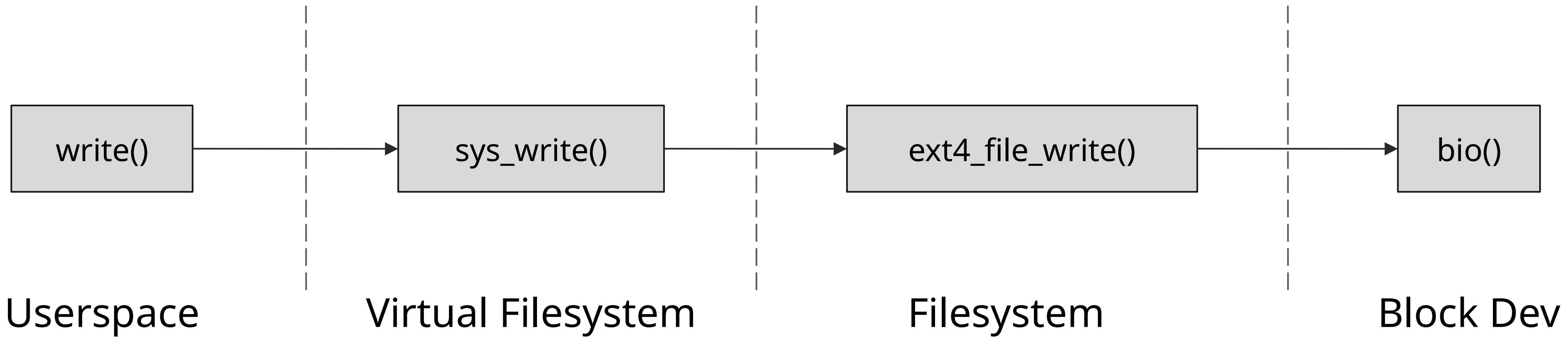
The VFS responsibility

- All file and filesystem-related interfaces available to userspace and other kernel subsystems.
- Virtually everything is interconnected by the VFS
  - You read and write from/to network sockets using VFS
- Abstracts the internal operations of all filesystems
- Most system calls are initially handled by the VFS.
- Together with the block layer, we have all necessary abstractions for user-space to access data in any media using the same generic system calls

The common file model

- VFS presents a “generic” view of files, filesystems, etc.
- And each filesystem must abstract their internal implementation to the VFS using such model.

The journey of a write() syscall



Red Hat

Main VFS Abstractions

Superblock

Represents a **specific mounted** filesystem

Inode

Descriptor containing **metadata details** related to a specific file.

Directory Entries

A **single component** in a path (not a directory).

File

An in-memory representation of an **opened file**

Object Oriented recap

- OOP is not a programming language, it is a programming paradigm
- The VFS (and basically the whole kernel) is objected oriented
- C doesn’t have OOP-specific support, so we need to use some different approaches.

Operations

- Each object provides a “structure” providing a set of operations for that specific object
- Each filesystem will populate this with their own operations
- Not all operations are mandatory and the VFS provide some generic ones if the filesystem doesn’t need any custom behavior
- Yes you can call these operations “methods”

Documentation/filesystem/vfs.rst

VFS data structures definitions

Object	Operations	Location
super_block	super_operations	include/linux/fs.h
inode	inode_operations	include/linux/fs.h
dentry	dentry_operations	include/linux/dcache.h
file	file_operations	include/linux/fs.h

Other important structures

- file_system_type (include/linux/fs.h)
- vfsmount (include/linux/fs.h)
- files_struct (include/linux/fs.h)
- fs_struct (include/linux/fs.h)
- mnt_namespace (include/linux/fs.h)

The Dentry Cache

- dentry object describes components in a path name
- Pathname lookups are very expensive, so we cache it.
- dentries have no on-disk correspondent, even on native Unix filesystems.
- Even invalid lookups are cached.
- Dentry cache also provides a front end for the inode cache

Block I/O Layer

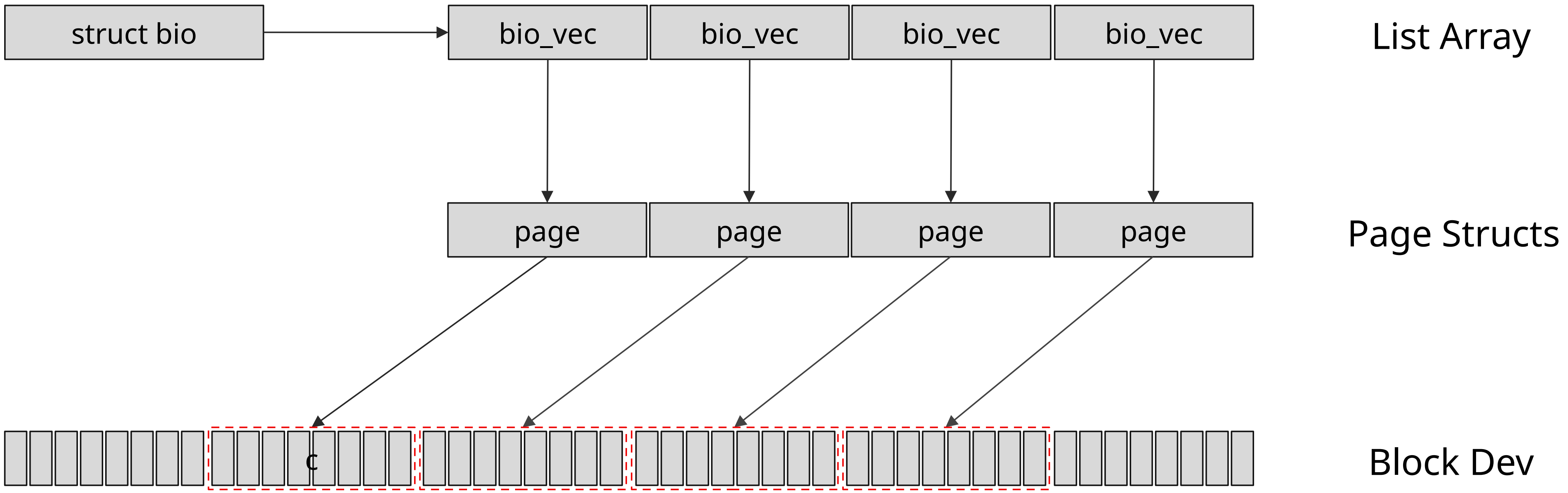
Buffers

- Every block read from disk storage, is cached in memory for some time.
- These blocks are stored in “buffers”
- buffer_heads … (on life support)

struct bio and bvec_iter

- Bio - the basic container for I/O within the kernel
- Represents every “in-flight” IO operation
- A bio describe a SINGLE contiguous storage location.
- Each bio is divided in segments - chunks of contiguous memory.

struct bio and bvec_iter #2



Request queues

- Each block device keeps its own request queue
- Higher level systems add requests to these queues
- The device driver grab such requests and submit them to the hardware

IO Schedulers

- Do not confuse with CPU schedulers
- Decide the order and the time requests are dispatched to the block device
- Most of the time, IO schedulers aim to reduce disk seeks
- Linux provides different scheduling algorithms

“Free Memory is wasted memory.”

The Page cache and Page writeback

Red Hat

Linux page cache

- Introduced initially in SysVr4 meant to cache only FS data
- Linux page cache aims to cache any page-based object
- The goal is to minimize disk I/O
  - milliseconds vs nanoseconds
- Temporal locality
  - Once accessed, data is likely to be accessed again

Linux page cache #2

- Physical pages in RAM related to physical blocks on disk
- Page cache is dynamic
  - Can grow and consume any free memory
  - Can shrink and relieve memory pressure if memory is low

Page cache based WRITES

- Page cache writes can be implemented in different ways
  - No-write - system does not cache write operations
  - Write-through - Write operations update both cache and disk
  - Write-back - Write goes to the cache only (Linux does this)

Page cache based READS

- Kernel first checks if the requested data is in the page cache
  - If we do, we have a **cache hit** and we don't need to go to the disk
- If not, we have a cache miss.
  - The kernel will schedule a block I/O operation to request the data off disk
- Once the data is read, it will now be added to the cache

Page cache based WRITES #2

- write operations write data to the page cache only
- Pages in the cache are marked **dirty** by the write operation
- After a determined amount of time and some rules, the pages are written back to disk.
- After return, a write() call does **not guarantee** the data is on disk
- Applications are responsible for their data integrity, not the kernel.
  - sync(), fsync(), fdatasync()
- System performance is the goal here

Cache eviction

- If memory is running low (or specified limits are being hit), the kernel needs to shrink the page cache.
- Which blocks should be uncached?
- What if there are no ‘clean pages’ in the page cache?
- The **clairvoyant algorithm**

Cache eviction #2

- Linux use a modified LRU, consisting of two lists:
- Active and Inactive list
- Active list contain “hot” pages and can’t be evicted
- Pages in the Inactive list are available for cache eviction
- Only when a page is accessed while in the inactive list, it can be “promoted” to the active list.
- Both lists are balanced. If the active list becomes larger than the inactive one, items are moved from the active to the inactive list

The address_space object

- A page in the page cache, may contain multiple non-contiguous physical disk blocks.
  - As files need not to be contiguous on disk, this works well.
- Linux uses the address_space object to manage entries in the page cache and page I/O operations.
  - By not tying it to specific VFS objects, like the inode, SB, we enable the page cache to be a generic cache, not usable only by filesystems.

The address_space object #2

- A file mapped in memory, will have a single address_space struct representing it.
  - Opposite of VMAs, where we can have several VMAs pointing to the same file.
  - It may have many virtual addresses, but it exists only once in physical memory
- Show me some code

Red Hat

address_space operations

- Yes, address_space also have different behaviors depending on the underlying user.
- The underlying user may be:
  - Filesystems, block devices, the buffer_head cache, swap subsystem.

Red Hat

Flusher Threads

- All storage writes are handled via the page cache
  - We will talk about DIO next
- All writeback is deferred to the “flusher threads”
- If data in the page cache is **dirty**
  - i.e. newer than their respective disk locations.
- The pages will be written back to disk once some conditions are met.
- The writeback is handled by flusher threads, which are kworker threads started on demand as a per-device basis

Flusher Threads #2

- So, when does writeback occurs?
  - Free memory is smaller than a specific threshold
  - Dirty data grows older
  - The user process forces the writeback to disk
    - sync() syscalls family

Disclaimer!!

FILESYSTEMS DON'T CARE ABOUT USER DATA

- It is not uncommon for users and developers to assume once a write() returns, the data is written on disk
- Again, it is user's (or application's) responsibility to ensure data is safe

Direct IO

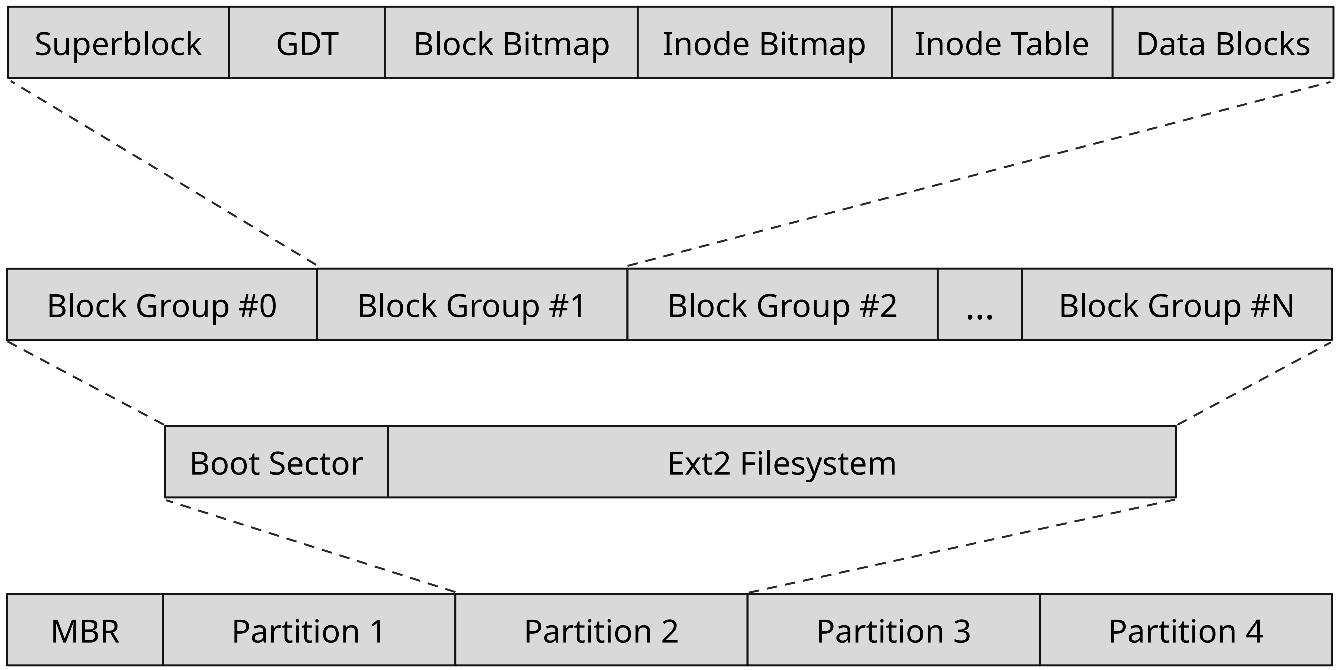
- From userspace, we can bypass the page cache by using Direct IO
  - All reads and writes goes from/to user space memory direct to/from disk using DMA.
- This has a big potential to increase performance
  - But as anything in computer world, there is a trade-off
- With DIO applications have more control over IO
- CPU usage is reduced (and potentially power consumption)
- IO must be **aligned** with the device's sector sizes

Going further

- different filesystem technologies
  - data allocation
  - metadata allocation
  - journaling

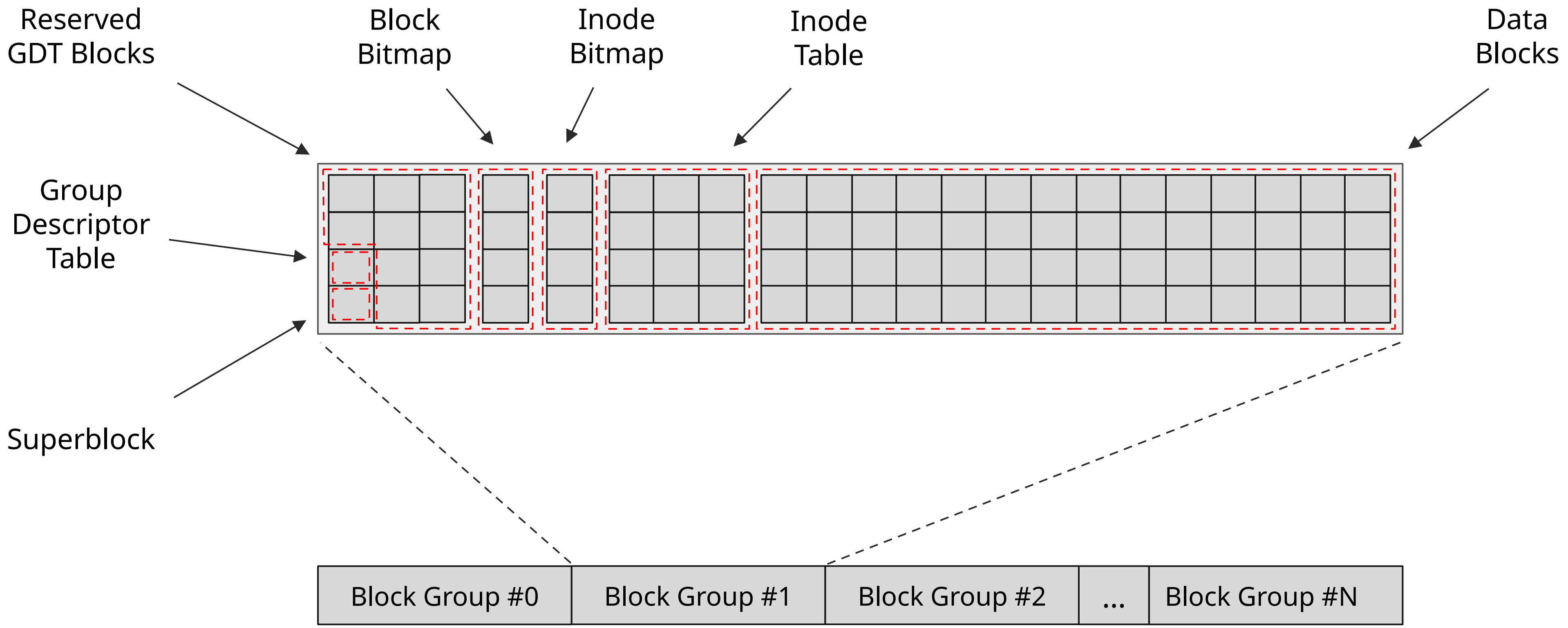
Case study:
The Ext2 Filesystem

Ext2 Disk Layout



https://en.wikipedia.org/wiki/Ext2

Ext2 Disk Layout #2



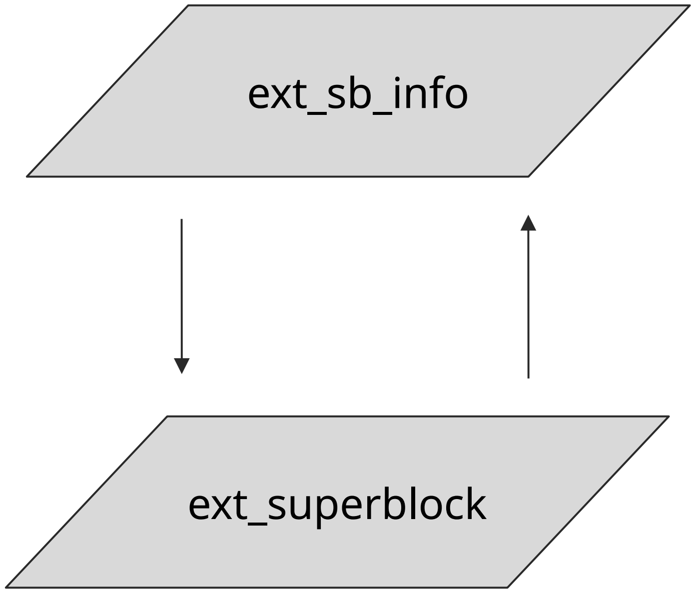
https://en.wikipedia.org/wiki/Ext2

On-disk vs In-memory structures

Object	On-disk	In-memory
Superblock	ext2_super_block	ext2_sb_info
Group Descriptor Table	ext2_group_desc	ext2_group_desc
Block bitmap	Raw format	Raw format
Inode bitmap	Raw format	Raw format
Inode table	Array of inodes	Raw format
Data blocks	file_operations	include/linux/fs.h

https://en.wikipedia.org/wiki/Ext2

On-memory and on-disk structures



Memory

Block Device

https://en.wikipedia.org/wiki/Ext2

Initializing an Ext2 Filesystem

- As virtually any other filesystem - it is initialized in userspace via specific tools (mkfs and friends)
- Goals:
  - parse config options
  - analyze the disk
  - create and initialize all metadata needed so that the kernel can properly mount and operate the filesystem

https://en.wikipedia.org/wiki/Ext2

Ext2 operations (aka methods)

- super_operations -> ext2_sops
- inode_operations ->
  - ext2_file_inode_operations
  - ext2_dir_operations
  - ext2_special_operations
- file_operations -> ext2_file_operations
- vm_operations_struct -> ext2_dax_vm_ops (no ops defined for non-dax)
- address_space_operations -> ext2_aops (ext2_dax_aops)

https://en.wikipedia.org/wiki/Ext2

Metadata management

- File layout on disk may differ from the user perspective
  - File data can be scattered everywhere on disk
  - Even though users just see a contiguous array of bytes
- Files may have holes in them (Sparse files)
- Space management attempts to address 2 main problems
  - Space fragmentation - big deal for spindles
  - Time efficiency

https://en.wikipedia.org/wiki/Ext2

Inode creation

- ext2_new_inode() allocate an ext2 disk inode and returns the address of the corresponding VFS inode object
- file vs directory allocation
- Update block group metadata
- Update superblock
- Initialize inode object
- Initialize quotas, acls, system security
- Pre-fetch the on-disk inode block where the new inode will be written

https://en.wikipedia.org/wiki/Ext2

Inode deletion

- Homework
  - Go and figure out what ext2_free_inode() does

https://en.wikipedia.org/wiki/Ext2

Data Addressing

- Files consists of blocks stored within block groups
- We can refer to them either as:
  - Their relative position inside the file (File block num)
  - Their position within the volume/partition (Logical block num)

https://en.wikipedia.org/wiki/Ext2

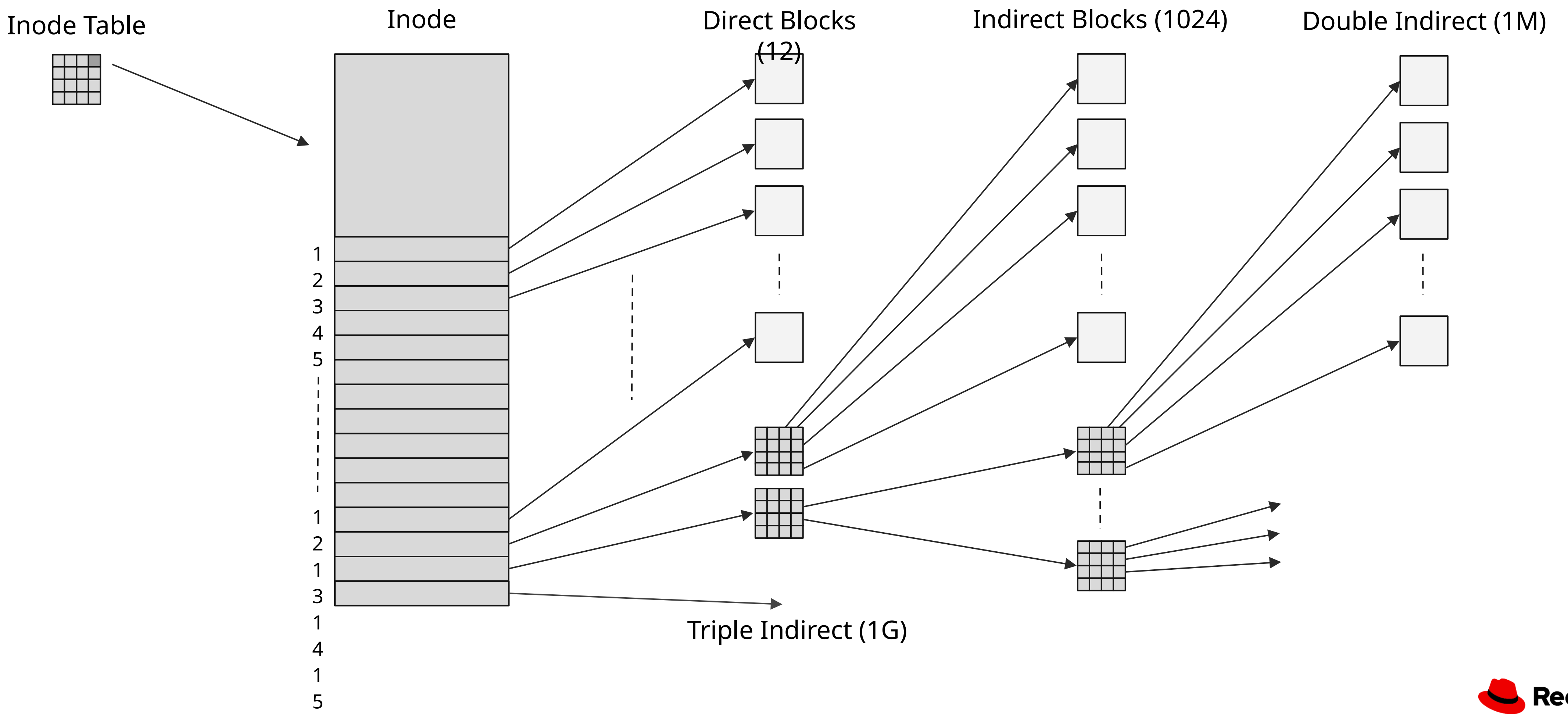
Data Addressing #2

- Retrieval of a file’s logical block number, is a two-step process:
  - 1 - Derive from the file offset, the block index containing such offset
  - 2- Translate the file block number to the corresponding logical block number
- Ext2 uses a simple data blocks management named **Indirect blocks**
- The ext2_inode contains an array of 15 block pointers

https://en.wikipedia.org/wiki/Ext2

Red Hat

Data Addressing #3



https://en.wikipedia.org/wiki/Ext2#Inodes

Block allocation

- ext2_get_block() and ext2_new_blocks()
  - Initially, attempts to find if the block already exists
  - If not, allocate a new one (or several ones)
- The allocator tries to reduce fragmentation, by allocating blocks as close as possible to the last already allocated block.
- The FS also does pre-allocation, by anticipating next writes beyond the first block requested.
- Ext2 allocator is a bit smarter now, and it tries to allocate blocks in batches

https://en.wikipedia.org/wiki/Ext2

Data deletion

- Data blocks must be reclaimed once a file is deleted or truncated
- We can also 'leak' data blocks the same way we leak memory
- Homework
  - Go read what ext2_truncate_blocks() and ext2_free_blocks() do

https://en.wikipedia.org/wiki/Ext2

Modern filesystem technologies

- Journalling
- COW filesystems
- Dynamically allocation of metadata
- Extents

https://en.wikipedia.org/wiki/Ext2

Extra Mile: ToyFS

Overview

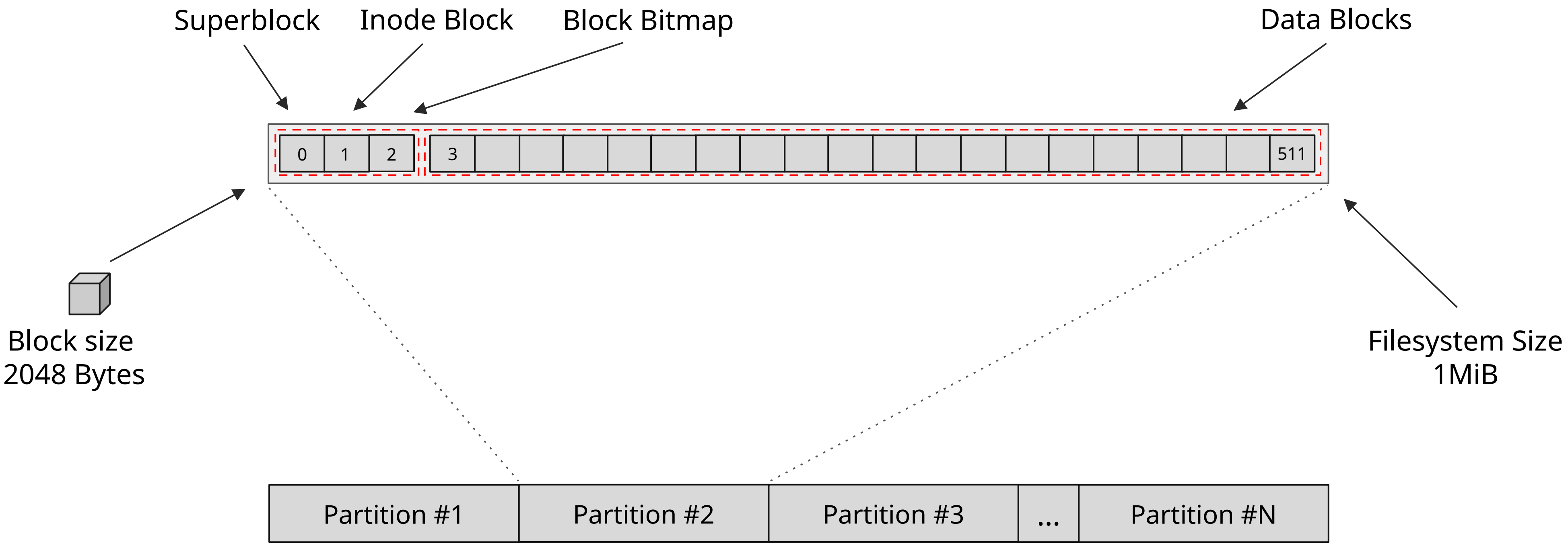
- Simple filesystem inspired on Steve Pate's UXFS
- Fixed 512 blocks of 2048 bytes (total space 1MiB)
- Implements fundamental filesystem operations

https://github.com/linuxtoyfs
https://www.goodreads.com/book/show/16769772-unix-filesystems

On-disk and In-memory structures

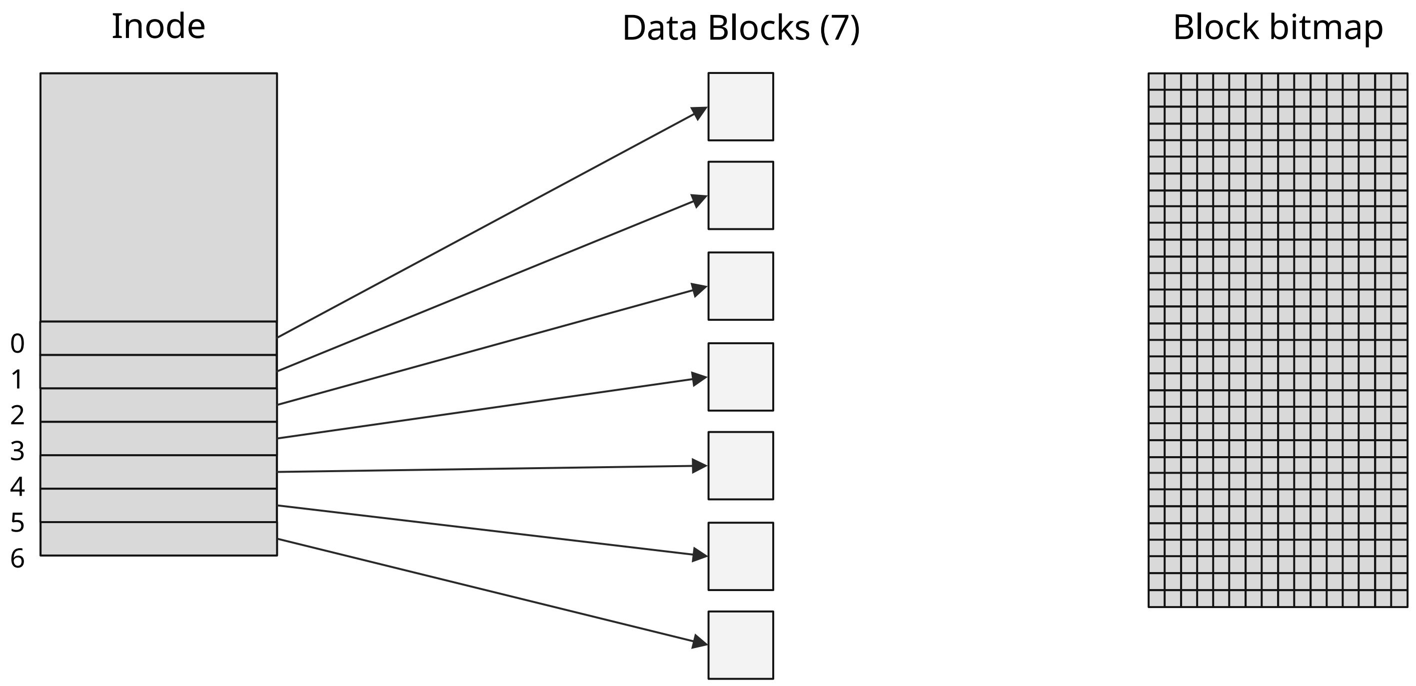
Object	Block	On-disk	In-memory
Superblock	#0	tfs_dsb	tfs_fs_info
Inode list	#1	tfs_dinode	tfs_inode_info
Block bitmap	#2	Raw format	Raw format
Data blocks	#3 to #512	User data	User data
Directory entry	-	tfs_dentry	tfs_dentry

Disk layout



Superblock
Inode Block
Block Bitmap
Data Blocks
0
1
2
3
511
Block size
2048 Bytes
Filesystem Size
1MiB
Partition #1
Partition #2
Partition #3
...
Partition #N

ToyFS data addressing



Red Hat

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make
Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat