CS 1217

File System Data Structures

File Systems

- What are they?

File System Design Goals

- Efficiently translate file name to contents
- Allow files to move, grow, shrink and change as per the user’s requirements
- Optimize access to files
  - Single files
  - Multiple, possibly related files
- Provide reliable access to names and contents and survive failures

File System Implementations

- What makes file systems different?
- **On-disk layout**
    - How does the file system decide where to put data and metadata blocks in order to optimize file access?
- **Data structures**
    - What data structures does the file system use to translate names and locate file data?
- **Crash recovery**
    - How does the file system prepare for and recover from crashes?

Why is this hard: write example

- A process wants to write data to the end of a file. What does the file system have to do?
  - Find empty disk blocks to use and mark them as in use.
  - Associate those blocks with the file that is being written to.
  - Adjust the size of the file that is being written to.
  - *Actually* copy the data to the disk blocks being used.
- From the perspective of a process all of these things need to happen **synchronously**.
- In reality, **many different asynchronous** operations are involved touching many different disk blocks.
- This creates both a consistency and a performance problem!

On Disk Happenings

- Let’s consider the **on-disk structures** used by modern file systems.

- Specifically we are going to investigate how file systems:
    - **translate** paths to file *index nodes* or inodes.
    - **find** data blocks associated with a given inode (file).
    - **allocate and free** inodes and data blocks.

- We’re going to try and keep this at a relatively high level, but draw from ext4 file system

File Metadata

- Information about the file that we’d want to know about
- **When** was the file created, last accessed, or last modified?
- **Who** is allowed to do what to the file—read, write, rename, change other attributes, etc.

Storing Metadata

- An MP3 file contains audio data. But it also has attributes such as:
  - title
  - artist
  - date
- Where should these attributes be stored?
  - In the file itself.
  - In another file.
  - In *attributes* associated with the file and maintained by the file system

Storing Metadata

- **In the file**:
- Example: MP3 ID3 tag, a data container stored within an MP3 file in a prescribed format.
- **Pro**
  - travels along with the file from computer to computer.
- **Con**
  - requires all programs that access the file to understand the format of the embedded metadata

Storing Metadata

- **In another file**
- Example: iTunes database.
- **Pro**
  - can be maintained separately by each application.
- **Con**
  - does not move with the file and the separate file must be kept in sync when the files it stores information about change.

File System Features

- The file systems in our discussions all support the following features:

- **Files**, including some number of file attributes and permissions.
- **Names**, organized into a **hierarchical** name space.

- The interface and feature set remains consistent (mostly)
- The changes happen mostly in implementation / optimization

Implementing Hierarchical File Systems

- Disk blocks can be divided into two types, broadly

- **Data blocks**: contain file data
- **Index nodes** (inodes): **!**(file data)

Sectors, Blocks and Extents

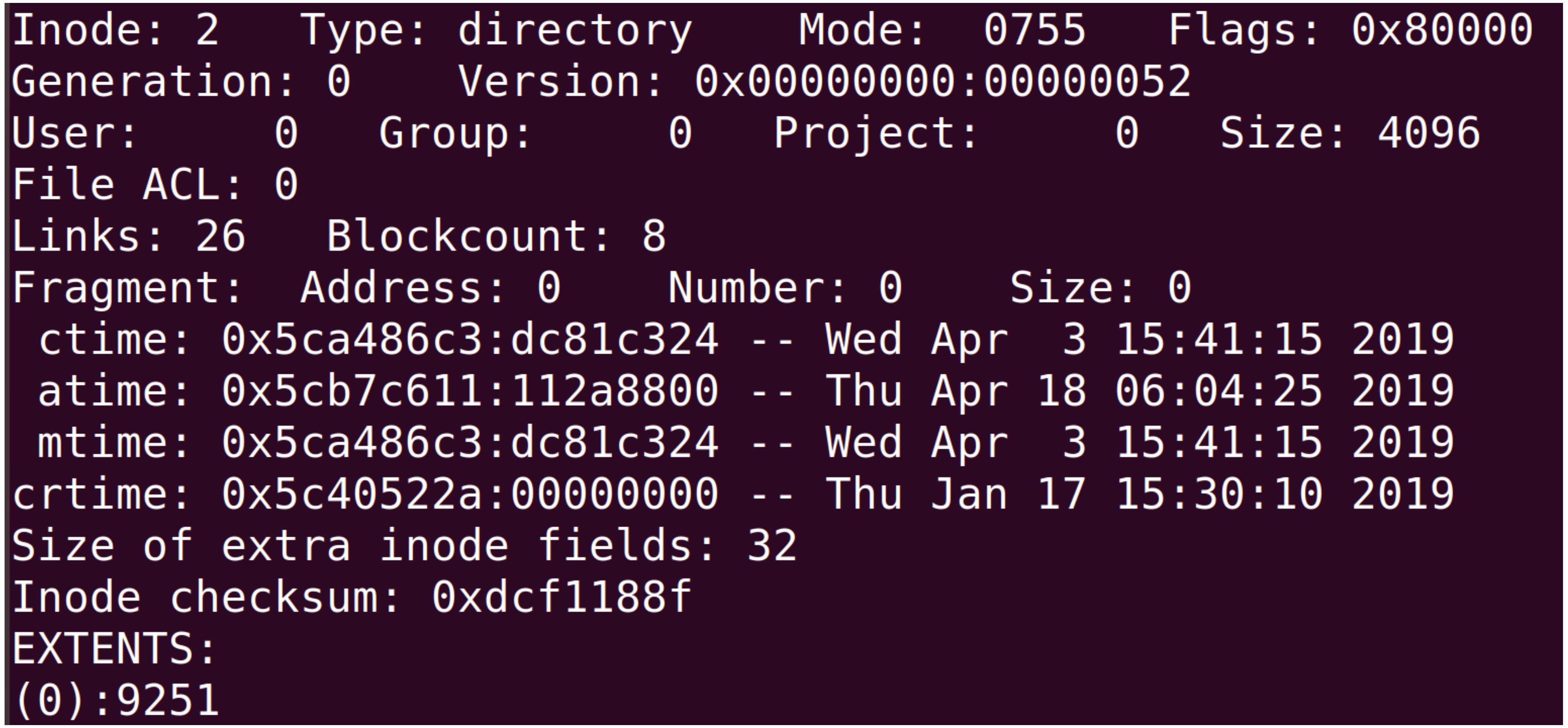
- **Sector**: the smallest unit that the disk allows to be written, usually **256/512** bytes.
- **Block**: the smallest unit that the file system actually writes, usually **4K** bytes.
- **Extent**: a set of contiguous blocks used to hold part of a file. Described by a start and end block.
- Why would file systems not write chunks smaller than 4K?
    - Because contiguous writes are good for disk head scheduling and 4K is the page size which affects in-memory file caching.
- Why would file systems want to write file data in even larger chunks?
    - Because contiguous writes are good for disk head scheduling and many files are larger than 4K!

ext4 inodes

- **1** inode per file.
- **256** bytes, so 1 per sector or 16 per block.
- Contains:
  - **Location** of file data blocks (contents).
  - **Permissions** including user, read/write/execute bits, etc.
  - **Timestamps** including creation (crtime), access (atime), content modification (mtime), attribute modification (ctime) and delete (dtime) times.
  - Named and located by **number**.

ext4 inodes

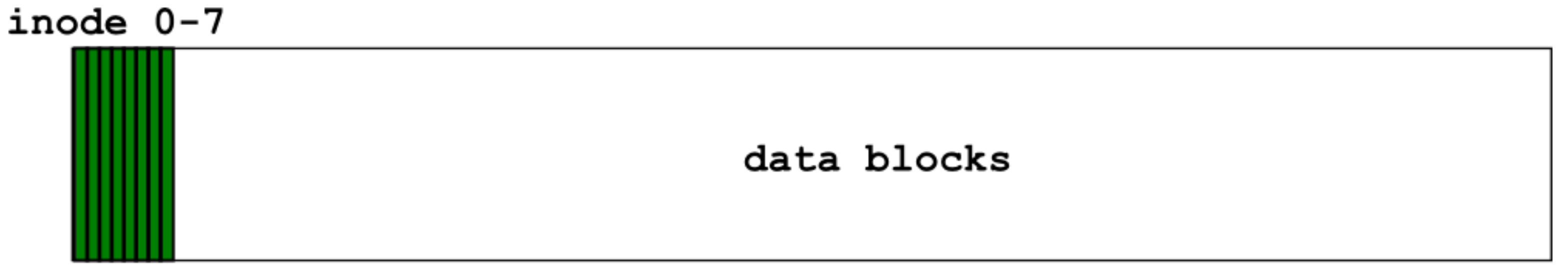
- debugfs –R “stat <2>” /dev/sda1



```
Inode: 2   Type: directory    Mode:  0755   Flags: 0x80000
Generation: 0    Version: 0x00000000:00000052
User:     0   Group:     0   Project:     0   Size: 4096
File ACL: 0
Links: 26   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x5ca486c3:dc81c324 -- Wed Apr  3 15:41:15 2019
 atime: 0x5cb7c611:112a8800 -- Thu Apr 18 06:04:25 2019
 mtime: 0x5ca486c3:dc81c324 -- Wed Apr  3 15:41:15 2019
crtime: 0x5c40522a:00000000 -- Thu Jan 17 15:30:10 2019
Size of extra inode fields: 32
Inode checksum: 0xdcf1188f
EXTENTS:
(0):9251
```

Locating inodes

- How does the system translate an inode number into an inode (data) structure?
- All inodes are created at **format time** at well-known locations.



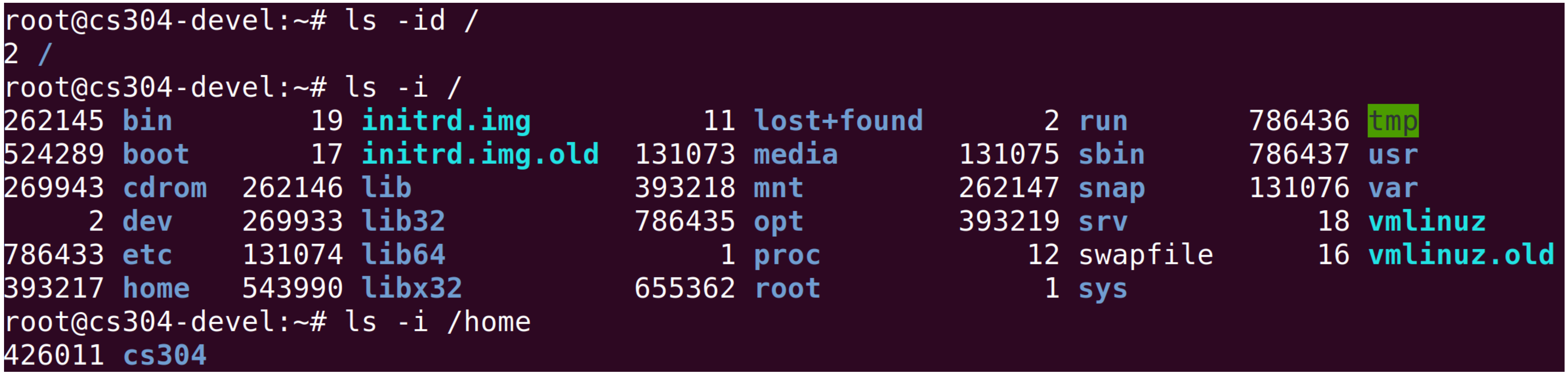
Locating Inodes

- **inodes may not be located near file contents**
    - ext4 creates multiple blocks of inodes within the drive to reduce seek times between inodes and data.

- Fixed number of inodes for the file system.
    - **Can run out of inodes before we run out of data blocks!**
    - ext4 creates approximately one inode per 16 KB of data blocks, but this can be configured at format time.

Directories

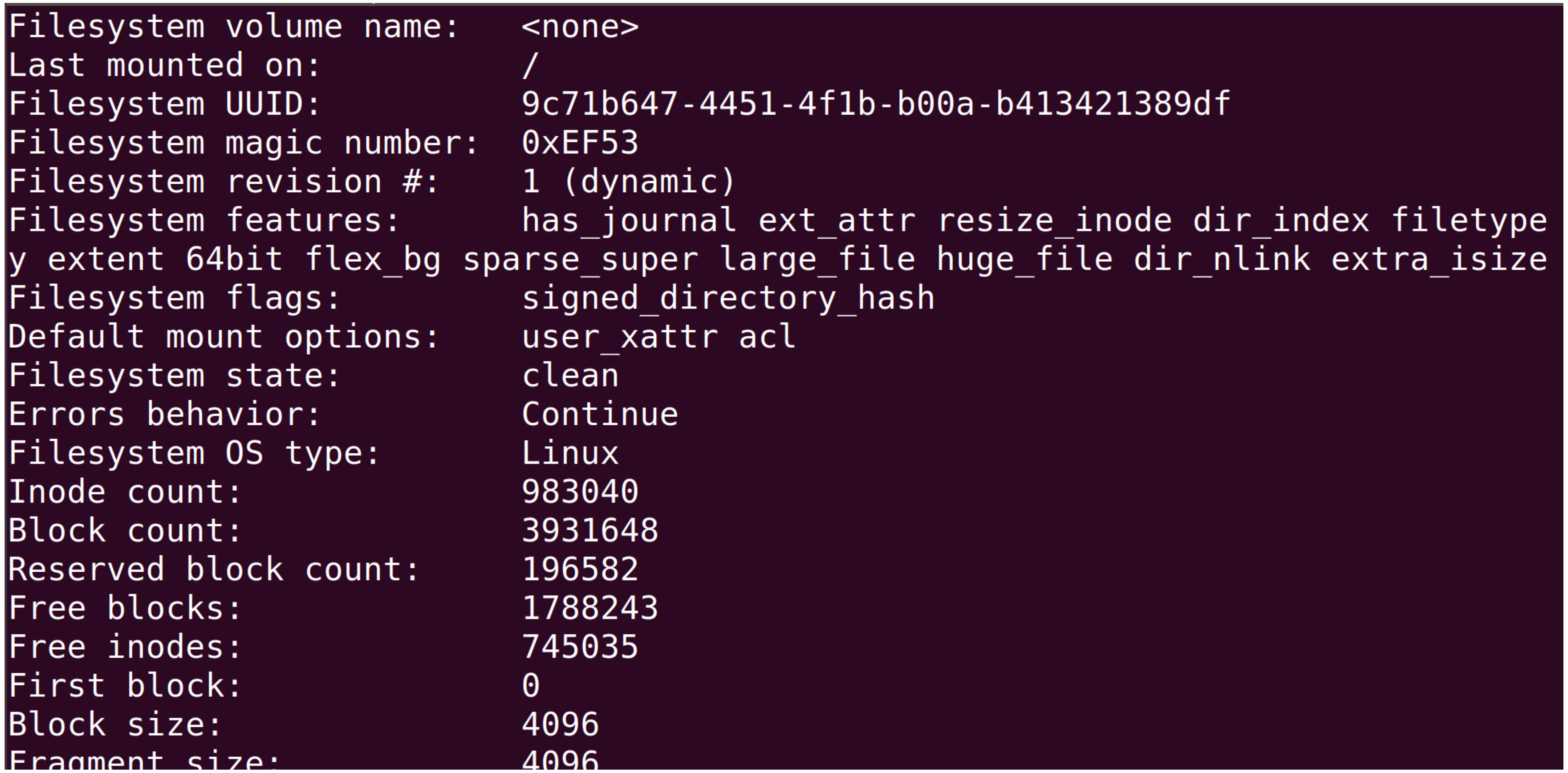
- A special type of file : maps inode numbers to names



```
root@cs304-devel:~# ls -id /
2 /
root@cs304-devel:~# ls -i /
262145 bin         19 initrd.img            11 lost+found        2 run        786436 tmp
524289 boot        17 initrd.img.old    131073 media        131075 sbin       786437 usr
269943 cdrom   262146 lib               393218 mnt          262147 snap       131076 var
     2 dev     269933 lib32             786435 opt          393219 srv            18 vmlinuz
786433 etc     131074 lib64                  1 proc             12 swapfile       16 vmlinuz.old
393217 home    543990 libx32            655362 root              1 sys
root@cs304-devel:~# ls -i /home
426011 cs304
```

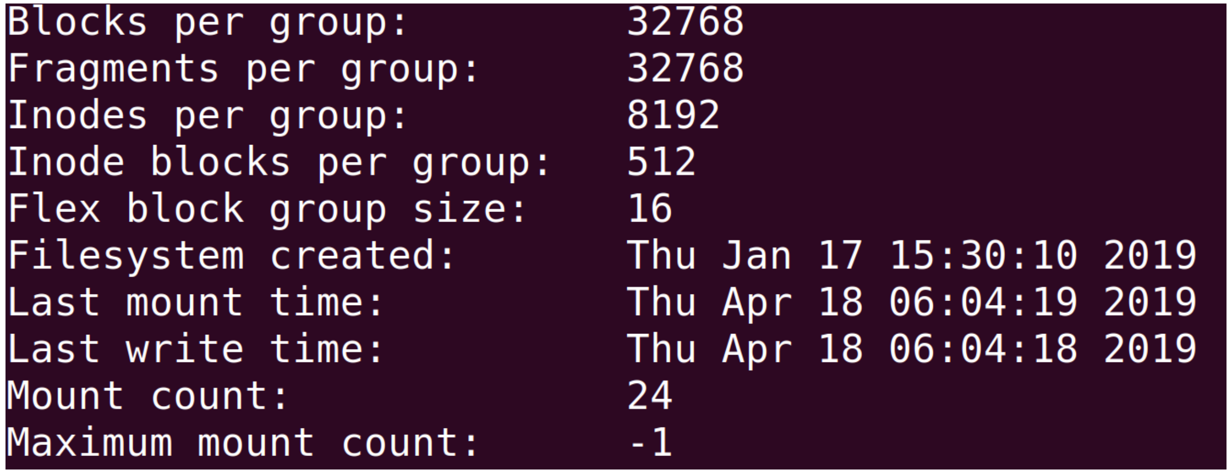
Debugfs : Super Blocks

$ debugfs –R “show_super_stats” /dev/sda1

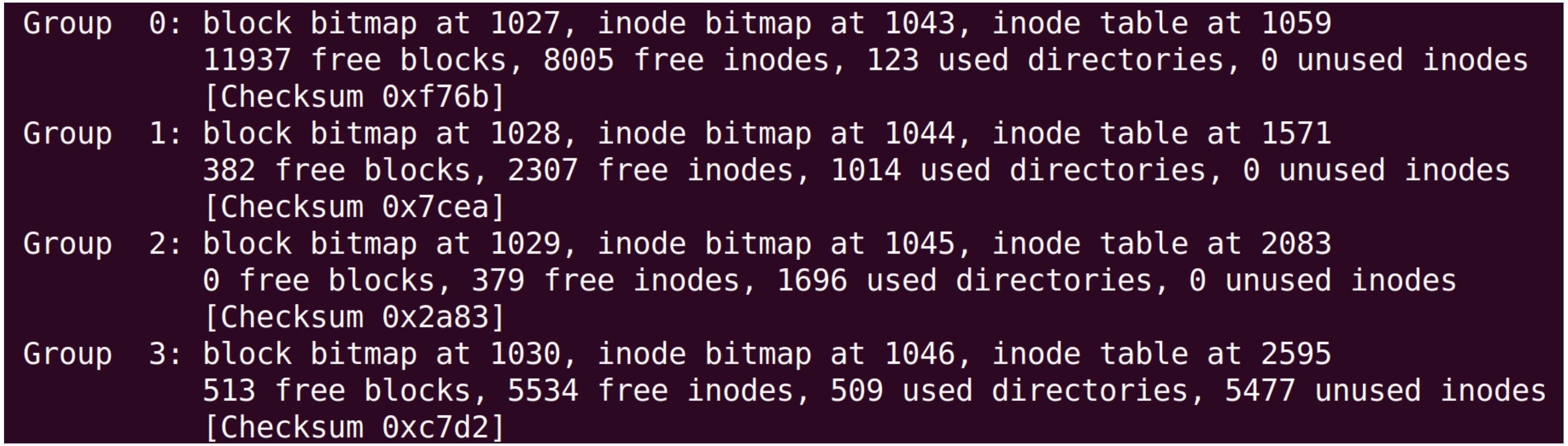


```
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          9c71b647-4451-4f1b-b00a-b413421389df
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype
y extent 64bit flex_bg sparse_super large_file huge_file dir_nlink extra_isize
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              983040
Block count:              3931648
Reserved block count:     196582
Free blocks:              1788243
Free inodes:              745035
First block:              0
Block size:               4096
Fragment size:            4096
```

debugfs



```
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8192
Inode blocks per group:   512
Flex block group size:    16
Filesystem created:       Thu Jan 17 15:30:10 2019
Last mount time:          Thu Apr 18 06:04:19 2019
Last write time:          Thu Apr 18 06:04:18 2019
Mount count:              24
Maximum mount count:      -1
```



```
 Group  0: block bitmap at 1027, inode bitmap at 1043, inode table at 1059
           11937 free blocks, 8005 free inodes, 123 used directories, 0 unused inodes
           [Checksum 0xf76b]
 Group  1: block bitmap at 1028, inode bitmap at 1044, inode table at 1571
           382 free blocks, 2307 free inodes, 1014 used directories, 0 unused inodes
           [Checksum 0x7cea]
 Group  2: block bitmap at 1029, inode bitmap at 1045, inode table at 2083
           0 free blocks, 379 free inodes, 1696 used directories, 0 unused inodes
           [Checksum 0x2a83]
 Group  3: block bitmap at 1030, inode bitmap at 1046, inode table at 2595
           513 free blocks, 5534 free inodes, 509 used directories, 5477 unused inodes
           [Checksum 0xc7d2]
```

Translating Names

- open("/etc/default/keyboard") must translate "/etc/default/keyboard" to an
- Get inode number for **root directory** - usually on a fixed agreed-on inode number, like **2**.
- Open the directory with inode number 2. Look for "etc". Find "etc" with inode number **786433**.
- Open the directory with inode number **786433**. Look for "default". Find "default" with inode number **786471**.
- Open the directory with inode number 786471. Look for "keyboard". Find keyboard with inode number **787706**.
- Open the file with inode number **787706**.

Retrieve and Modify **Data** Blocks

- read/write(filehandle, 345) must translate **345** to a **data block** within the open file to determine what data block to modify.

- How do we do this?

Data Blocks: Linked List

- One solution: organize data blocks into a **linked list**.
- inode contains a pointer to the first data block.
- Each data block contains a pointer to the previous and next data block.
- Pros
    - Simple
- Cons
    - Slow lookups!

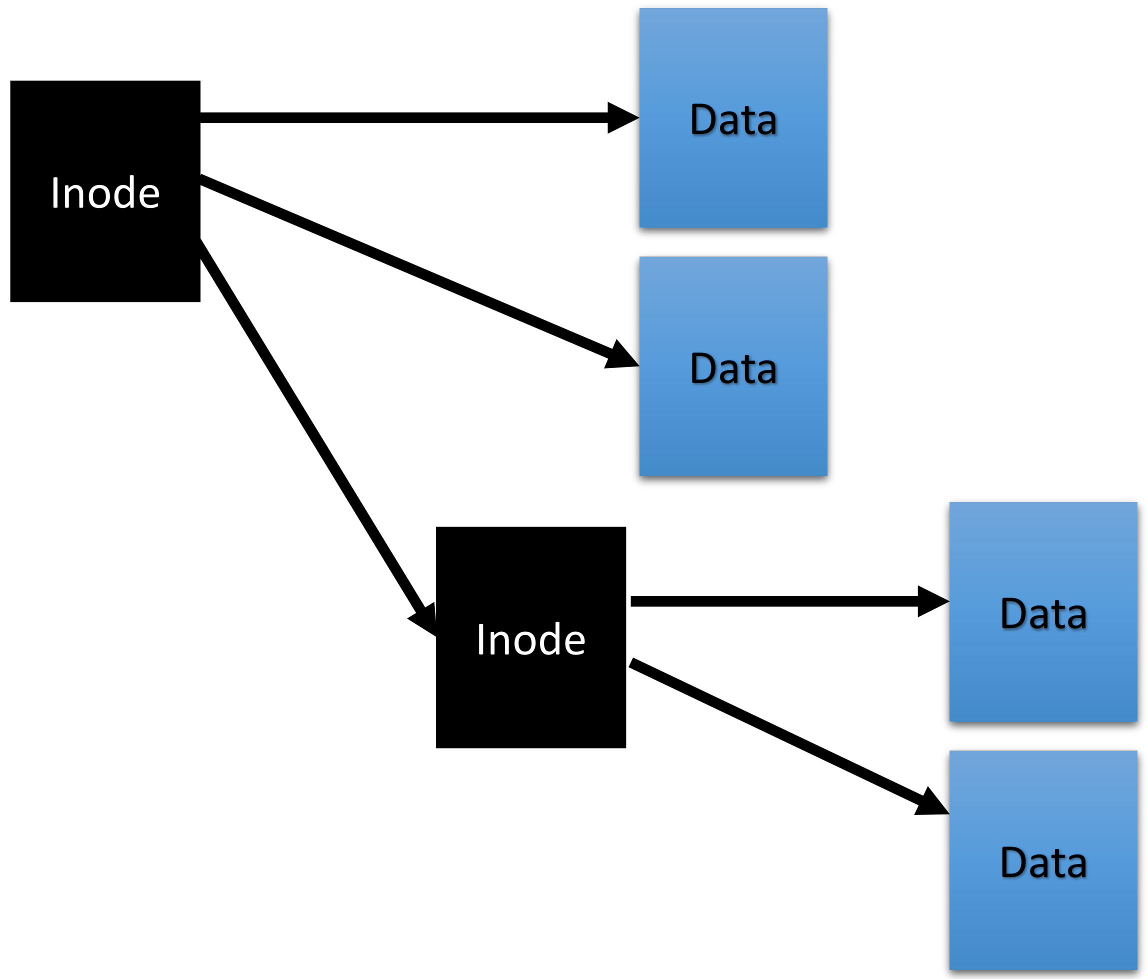
Data Blocks: Flat Array

- store all data blocks in the inode in a single array allocated at file creation time.
- Pros:
  - Simple.
  - Offset look ups are fast, O(1).
- Cons:
  - Small file size fixed at startup time
  - Large portion of array may be unused

Data Blocks: Multilevel Index

- Observation: **most** files are small, but **some** can get very large.
- Have inode store:
  - some pointers to blocks, which we refer to as **direct** blocks.
  - some pointers to blocks containing pointers to blocks, which we refer to as **indirect** blocks.
  - some pointers to blocks containing pointers to blocks containing pointers to blocks, which we refer to as **doubly indirect** blocks.

Data Blocks: Multilevel Index



Data Blocks: Multilevel Index

- Pros:
  - Index scales with the size of the file.
  - Offset look ups are still fairly fast.
  - Small files stay small, but big files can get extremely large.

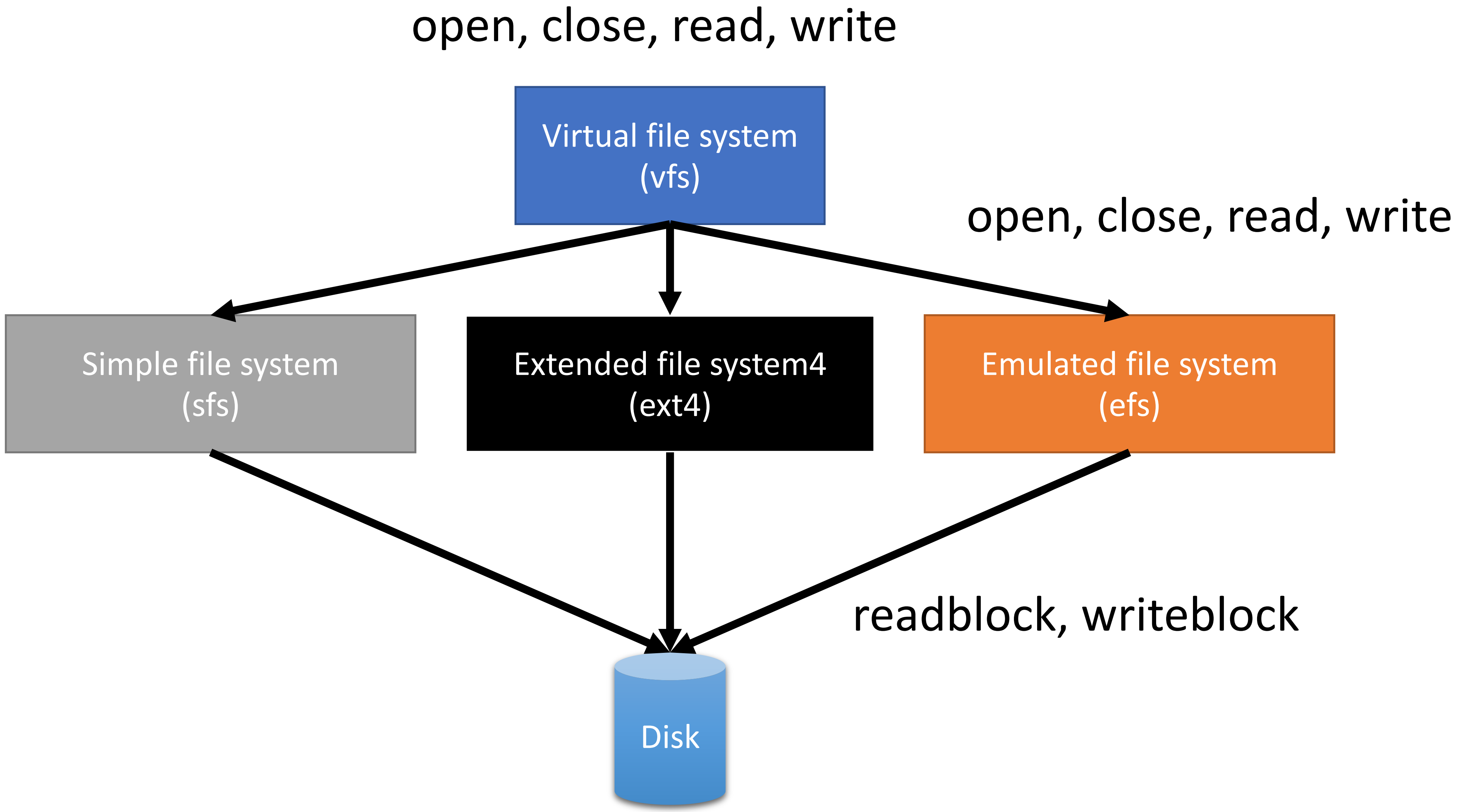
Putting Spare (?!) Memory to Good Use

- Memory / DRAM is used by the OS for
  - As **regular memory** for things like process **address spaces**
  - As **cache** for the **file system** to improve disk performance

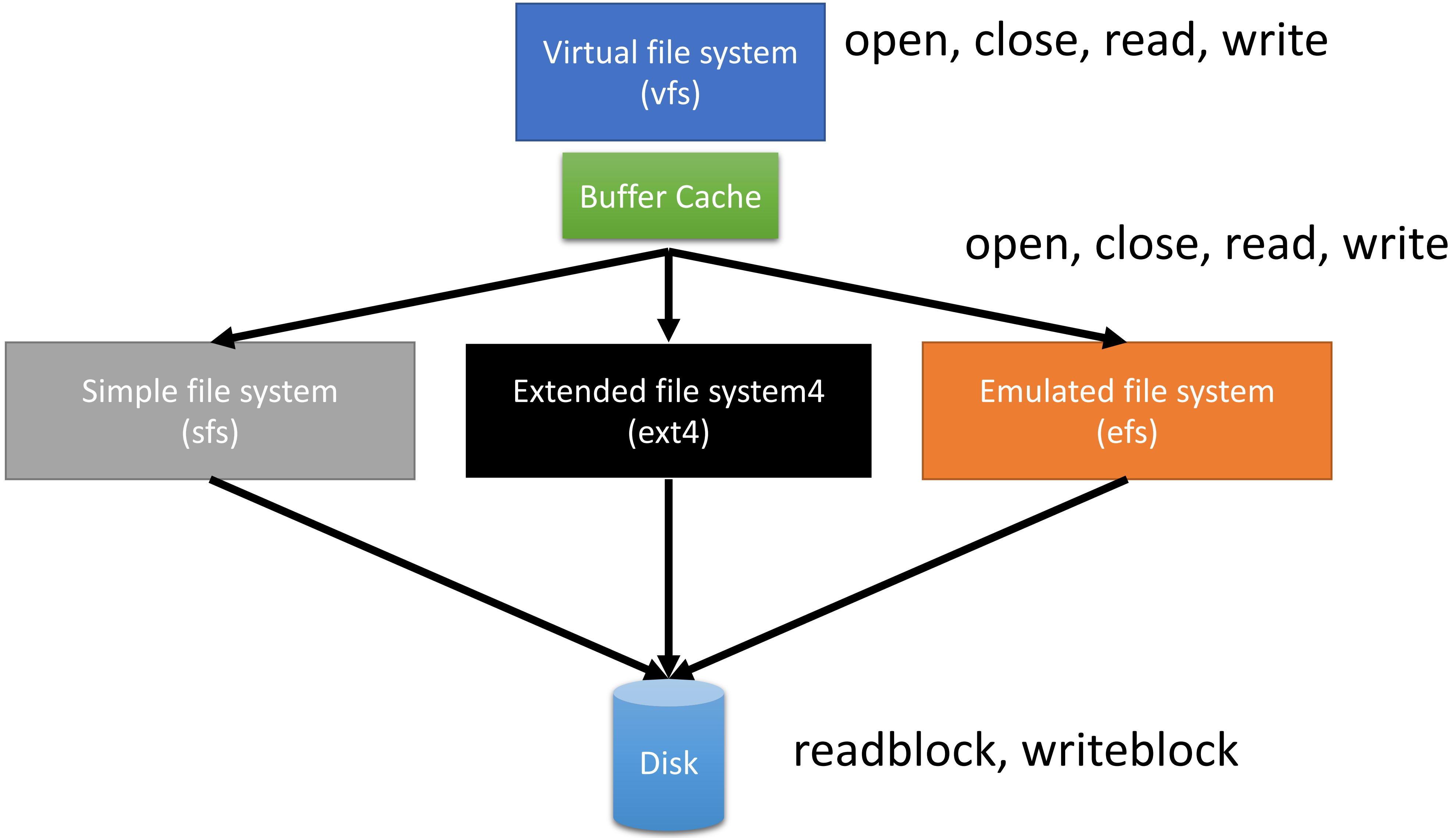
- Both compete for the same memory capacity
- Small Buffer Cache vs Large Memory
  - little swapping occurs but file access is extremely slow
- Large Buffer Cache vs Small Memory
  - file access is fast, but potential thrashing in the memory subsystem…

- `swappiness` : Linux kernel parameter controls how aggressively the operating system prunes unused process memory pages and hence the balance between memory and buffer cache

Where Should the Buffer Cache be Located?



Where Should the Buffer Cache be Located?



Where Should the Buffer Cache be Located?

- What is the buffer cache interface?
  - open, close, read, write. (Same as the file system call interface.)

- What do we cache?
  - **Entire files and directories!**

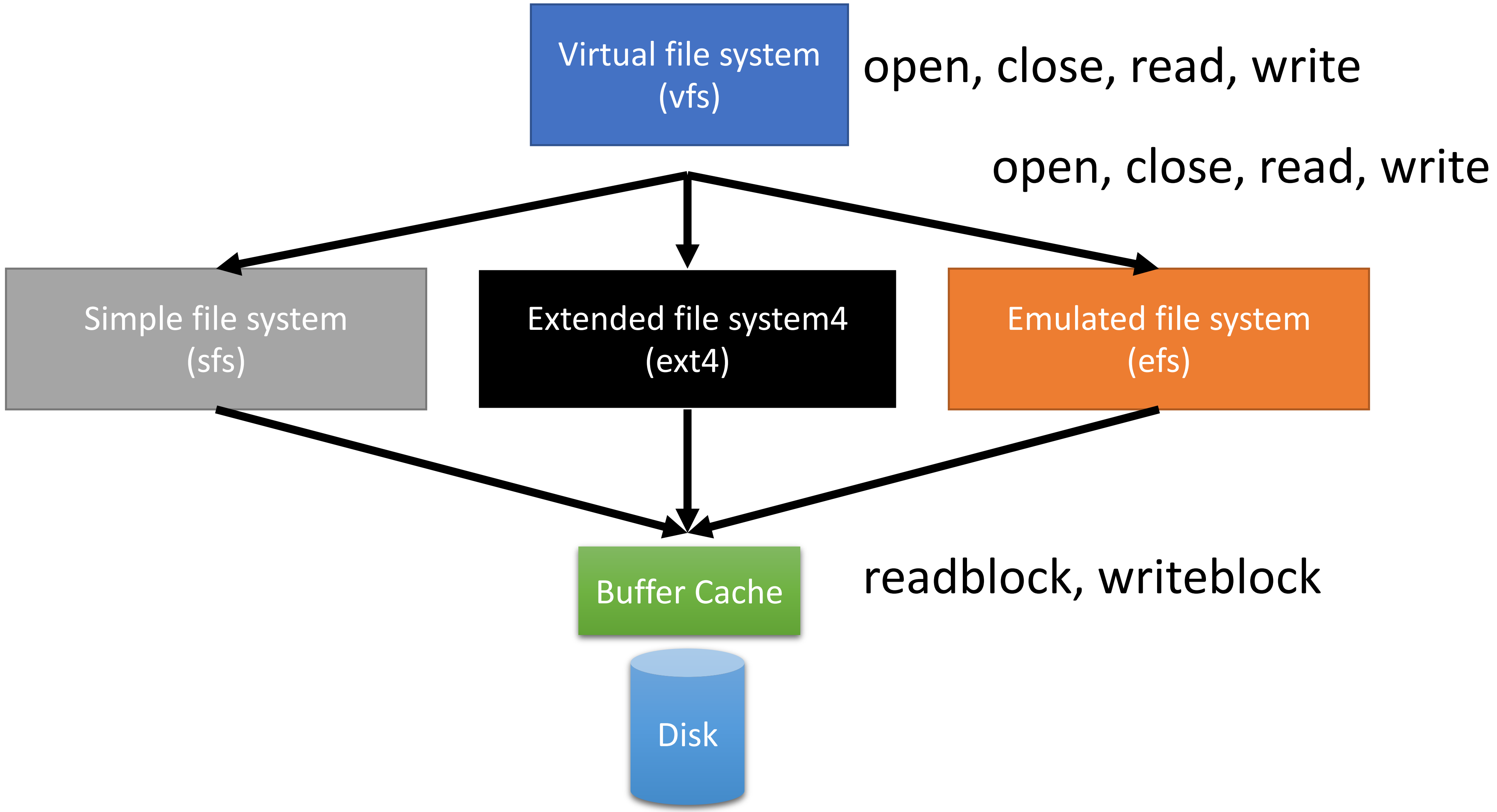
Above the File System: Operations

- open
  - Pass down to underlying file system.
- read
  - If file is not in the buffer cache, pass down to underlying file system and load contents into the buffer cache.
  - If the file is in the cache, return the cached contents.
- write
  - If file is not in the buffer cache, pass load contents into the buffer cache and then modify them.
  - If the file is in the cache, modify the cached contents.
- close
  - Remove from the cache (if necessary) and flush contents through the file system.

Above FS : Pros and Cons

- Pros:
  - Buffer cache sees file operations, may lead to better prediction or performance.
- Cons:
  - Hides many file operations from the file system, preventing it from providing consistency guarantees.
  - Can’t cache file system metadata: inodes, superblocks, etc.

Where Should the Buffer Cache be Located?



Below FS :

- What do we cache?
    - **Disk blocks!**

- What is the buffer cache interface?
    - readblock, writeblock. (Same as the disk interface.)

Below FS: Pros and Cons

- Pros:
  - Can cache all blocks including file system data structures, inodes, superblocks, etc.
  - Allows file system to see all file operations even if they eventually hit the cache.
- Cons:
  - Cannot observe file semantics or relationships.

Buffer Cache v. Process Pages

- Operating systems use memory:
  - **as memory**, but also
  - to cache file data in order to improve performance.

- These two uses of memory **compete** with each other.
  - **Big** buffer cache, **small** main memory: file access is fast, but potential thrashing in the memory subsystem…
  - **Small** buffer cache, **large** main memory: little swapping occurs but file access is extremely slow.

Buffer Cache Recap

- Where is the buffer cache typically located? Above or below the file system?
  - Below.
- What does the buffer cache store?
  - Complete disk blocks, including file system metadata.

Caching and Consistency

- How can the cache cause consistency problems?
  - Items in cache can be lost on failures
- Every file system operation involves modifying **multiple** disk blocks
- Example of creating a new file in an existing directory:
  - Allocate an **inode**, mark the used inode bitmap.
  - Allocate **data blocks**, mark the used data block bitmap.
  - **Associate** data blocks with the file by modifying the inode.
  - Add inode to the given **directory** by modifying the directory file.
  - **Write** data blocks.

Consistency Problems from Caching

- File system operations that modify multiple blocks may leave the file system in an **inconsistent** state if partially completed.
- How does caching exacerbate this situation?

- May increase the time span between when the *first* write of the operation hits the disk and the *last* is completed.

What can go wrong?

- Inconsistencies can happen if the system fails between any of the operations below for a write
- Allocate an **inode**, mark the used inode bitmap.
  - **Inode** incorrectly marked in use.
- Allocate **data blocks**, mark the used data block bitmap.
  - Data blocks incorrectly marked in use.
- **Associate** data blocks with the file by modifying the inode.
- Add inode to the given **directory** by modifying the directory file.
- **Write** data blocks.
  - Data loss!

Maintaining File System Consistency

- What’s the safest approach?
  - **Don’t buffer writes!**
  - *write through* cache : writes do not hit the cache.
- What’s the most dangerous approach?
  - **Buffer all operations until blocks are evicted.**
  - *write back* cache : write to next level on data eviction
- Which approach is better for
  - Performance?
  - Safety?

Maintaining File System Consistency

- Buddha’s mantra : Middle Way
- Write important **file system metadata structures**—superblock, inode maps, bitmaps, etc. -- **immediately**, but **delay data writes**.

- File systems also give use processes some control through **sync** (sync the entire file system) and **fsync** (sync one file).

Another Approach to Consistency

- What's *not* atomic?
  - Writing **multiple** disk blocks

- What *is* atomic?
  - Writing **one** disk block.

Journaling

- Track pending changes to the file system in a special area on disk called the **journal**
- Following a failure, **replay** the journal to bring the file system back to a consistent state.

- List of Tasks
- *Allocate inode 567 for a new file.*
- *Associate data blocks 5, 87, and 98 with inode 567.*
- *Add inode 567 to the directory with inode 33.*
- *That’s it!*

Journaling: Checkpoints

- What happens when we flush cached data to disk?
- Update the journal!
  - called a **checkpoint**.

- List of Tasks
- ~~*Allocate inode 567 for a new file.*~~
- ~~*Associate data blocks 5, 87, and 98 with inode 567.*~~
- ~~*Add inode 567 to the directory with inode 33.*~~
- ~~*That's it!*~~

Journaling: Recovery

- Go through the log and keep updating the disk-structures as and when needed.
  - Might not need to do anything
- Delete the journal entry when done

- That's it! Everything done
- **Checkpoint**
- List of Tasks (for this Checkpoint)
- *Allocate inode 567 for a new file.* ✓
- *Associate data blocks 5, 87, and 98 with inode 567.* ***Not done****.* ✓
- *Add inode 567 to the directory with inode 33.* ***Not done.*** ✓
- *That's it!* ***Not done****.* ✓

Journaling: Recovery

- What about incomplete journal entries?
- These are **ignored** as they may leave the file system in an incomplete state.

- What would happen if we processed the following incomplete journal entry?
- List of Tasks
- *Allocate inode 567 for a new file.* ✓
- *Associate data blocks 5, 87, and 98 with inode 567.* ***Not done****.* ✓
- --

Journaling: Implications for Data

- Observation: **metadata** updates (allocate inode, free data block, add to directory, etc.) can be represented compactly and probably written to the journal **atomically**.

- What about data blocks themselves changed by write()?
  - We could **include them in the journal** meaning that each data block would potentially be written *twice* .
  - We could **exclude them from the journal** meaning that file system structures are maintained but not file data.