CS-446/646

Linux Filesystems, FSCK & Journaling

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Linux Filesystems, FSCK & Journaling

Filesystems **in Linux**

Linux Second Extended File System (*Ext2*)

- What is the *Ext2* on-Disk Layout?
- What is the *Ext2 Directory* Structure?

- *Linux Third Extended File System* (*Ext3*)
  - What is the *Filesystem Consistency* problem?
  - How to solve the *Consistency* problem using *Journaling*?

- *Virtual File System* (VFS)
  - What is VFS?
  - What are the key data structures of Linux VFS?

Linux *Ext2*

- "Standard" Linux File System
  - Was the most commonly used before *Ext3* came out

- Uses FFS-like Layout
  - Each *Filesystem* is composed of identical *Block Groups*
  - Allocation is designed to improve *Locality*

- *inodes* contain Pointers (32-bit) to *Blocks*
  - *Direct*, *Single Indirect*, *Double Indirect*, *Triple Indirect*
  - Maximum *File Size*: 4.1 TB (4 KB *Block Size*)
  - Maximum *Filesystem* Size: 16 TB (4 KB *Block Size*)

- On-Disk structures: **`/linux/ext2_fs.h`**

Linux *Ext2* Disk Layout

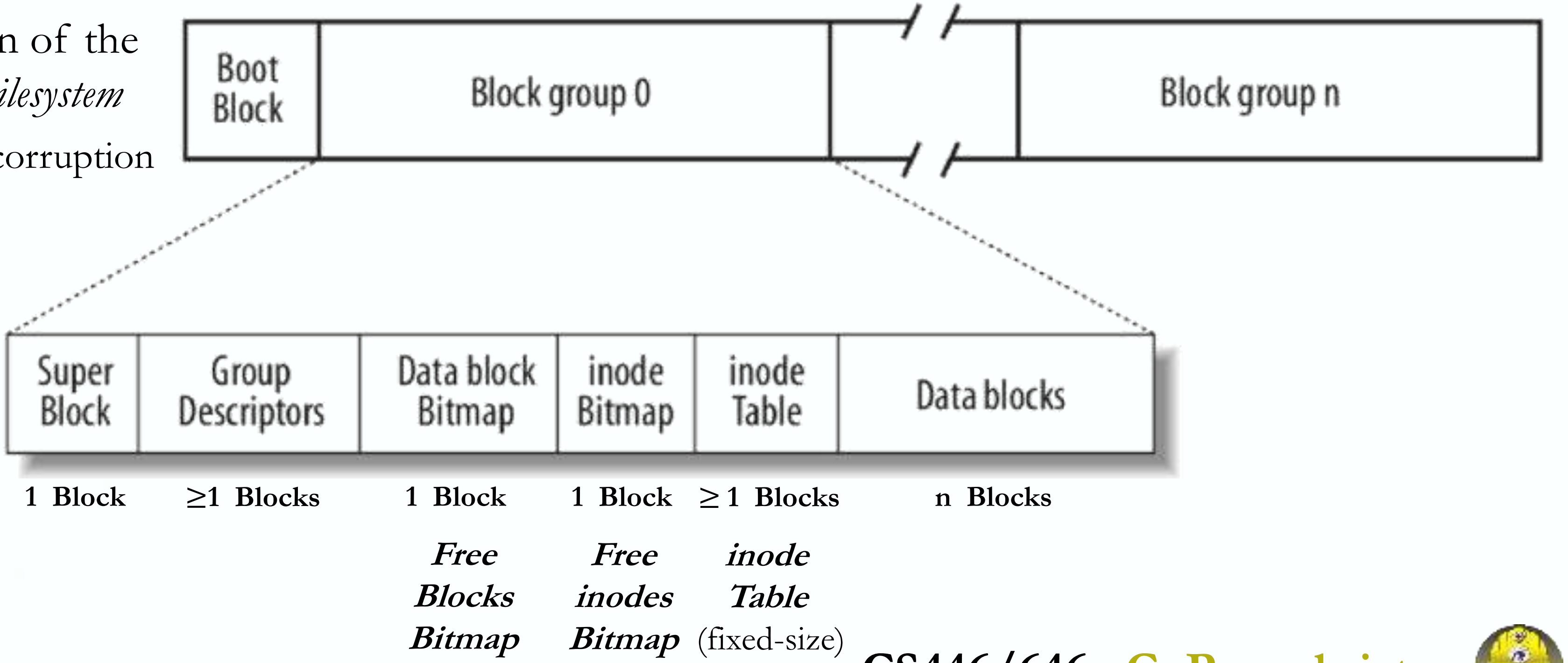
- *Locality:*
  - *Files* in the same *Directory* are stored in the same *Block Group*
  - *Files* in different *Directories* are spread among the *Block Groups*

Superblock: Contains description of the basic Size and Shape of this *Filesystem*

Note: Duplicated in all *Groups* for corruption

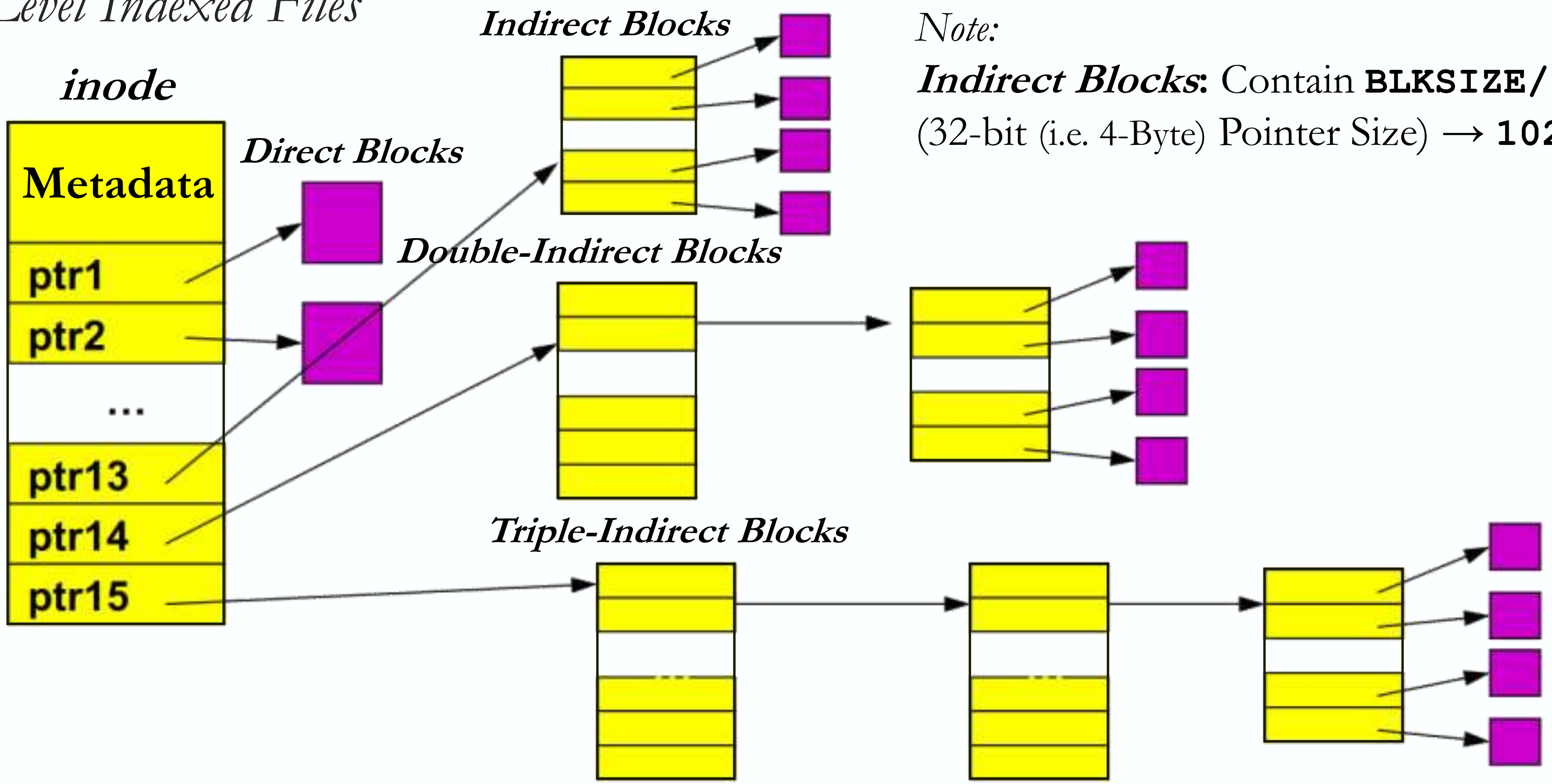
Block Group Descriptor Table: Contains description for each *Block Group* of this *Filesystem* (Address of *Block Bitmap, inode Bitmap, inode Table,* # Free *Blocks*, # Free *inodes*, # *Directories*)

Note: Also duplicated in all *Groups* for corruption



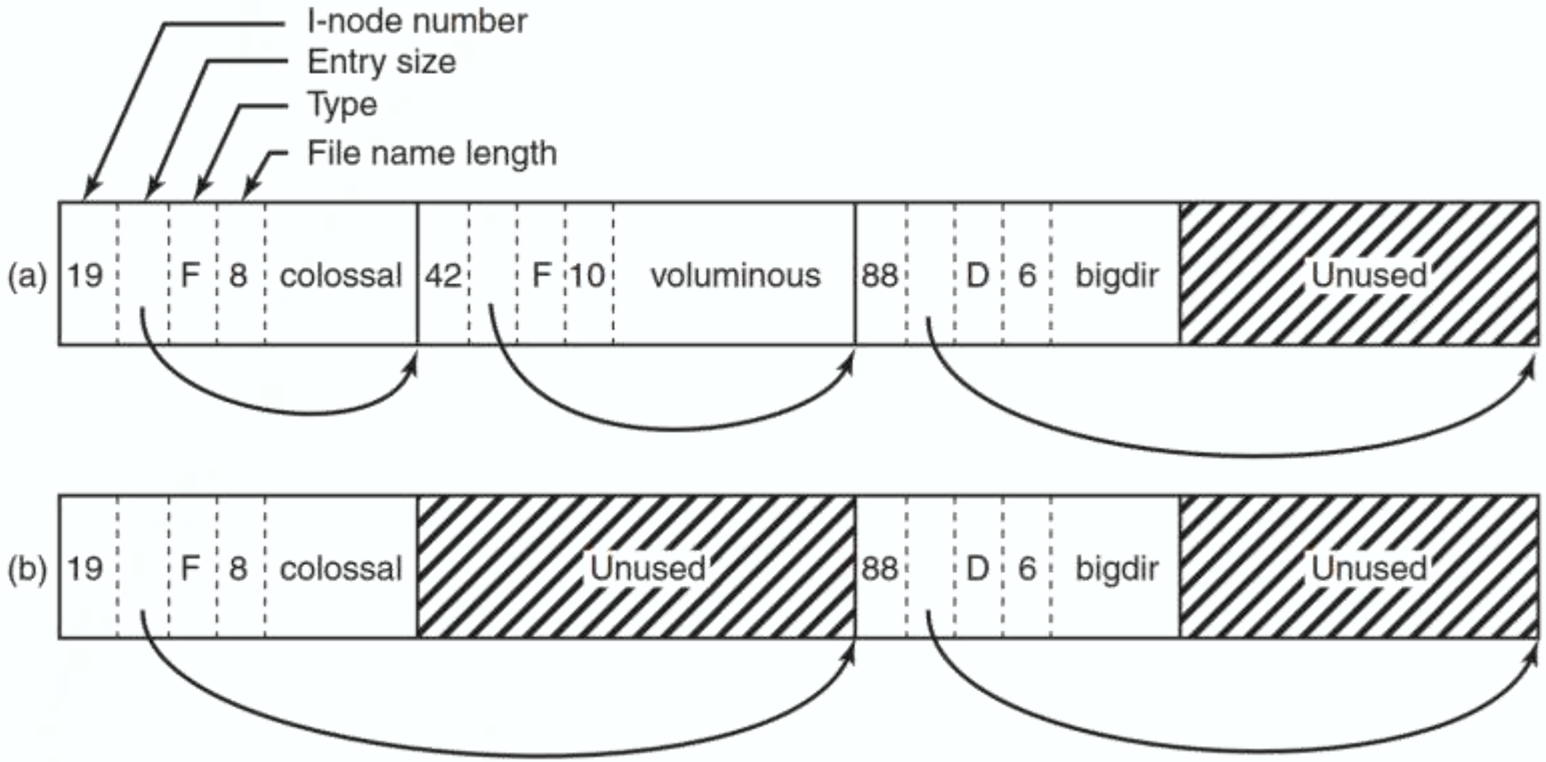
Block Addressing in *Ext2*

- *Multi-Level Indexed Files*



Note:
Indirect Blocks: Contain **`BLKSIZE/4`** *Block Entries* (32-bit (i.e. 4-Byte) Pointer Size) → **`1024`** *Block Entries*

Ext2 Directory Structure



- (a) A Linux *Directory* with 3 *Directory Entries* (e.g. for *Files*: **`colossal`**, **`voluminous`**, **`bigdir`**)
- (b) After the *File* **`voluminous`** has been removed

Linux *Ext3*

The *Consistent Update* Problem – *Filesystem Consistency*

Goal:

- *Atomically* update *Filesystem* from one *Consistent State* to another
  - What is the meaning of "*Consistent State*"?

Challenge:

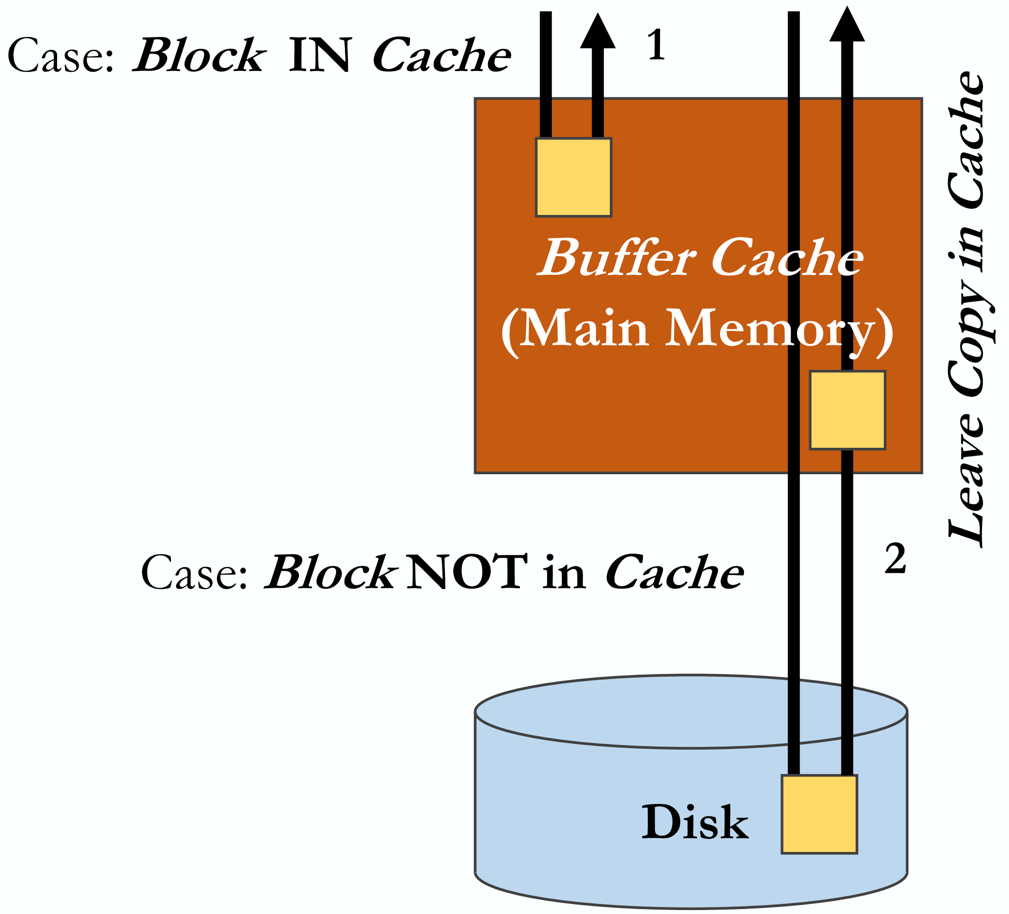
- An update may require modifying **multiple** *Sectors*
- But the Disk Hardware only provides *Atomic* write of one *Sector* at a time

Linux Filesystems, FSCK & Journaling

Linux *Ext3*

Review: *File* I/O Path (Reads)
- *Filesystem* uses *Buffer Cache* to speed-up I/O

`read()` from a *File*
- Check if *Block* inside *Buffer Cache*
- If yes, return *Block*
- If not, read from *Disk*, insert into *Buffer Cache* and return it



Linux *Ext3*

Review: *File* I/O Path (Writes)

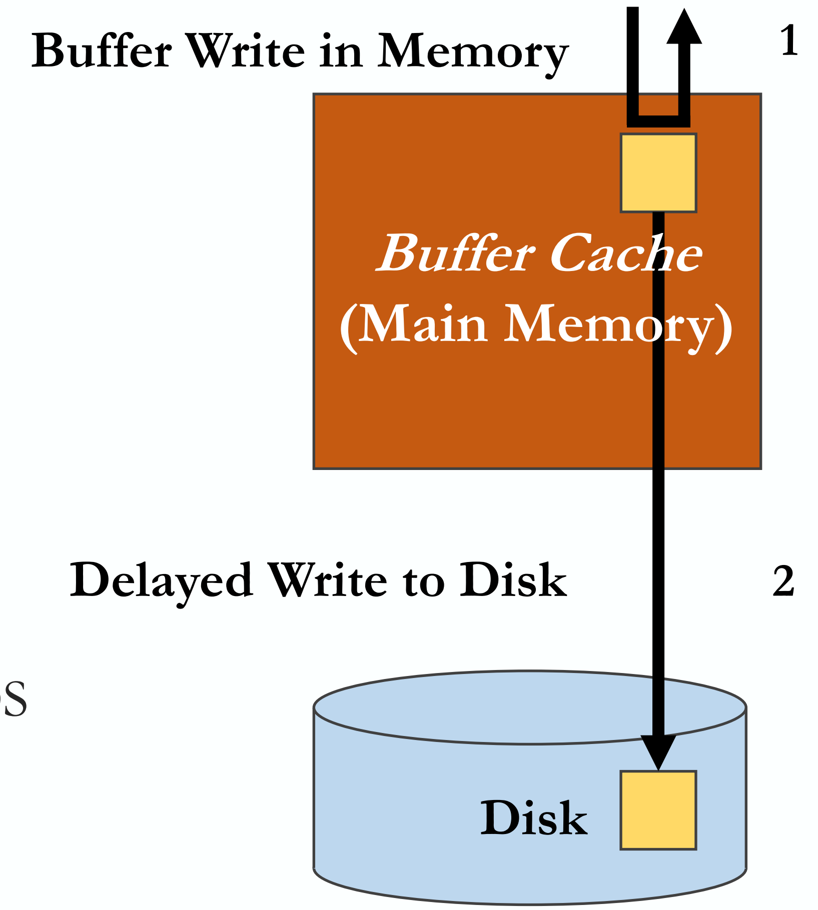
- *Filesystem* uses *Buffer Cache* to speed-up I/O

`write()` to *File*

- Write is Buffered in Memory – "*Write-Back*"
  - (vs "*Write-Through*")
- OS decides when to write to Disk
- Periodic `fsync()` *System Call*

Note:
`fflush()`: Flush internal *Application Buffers* (of a `FILE *`) to OS
`fsync()`: Flush *OS Buffers* to Physical Media (Device)

- Delaying writes important:
  - Has implications for Performance
  - Has implications for Reliability

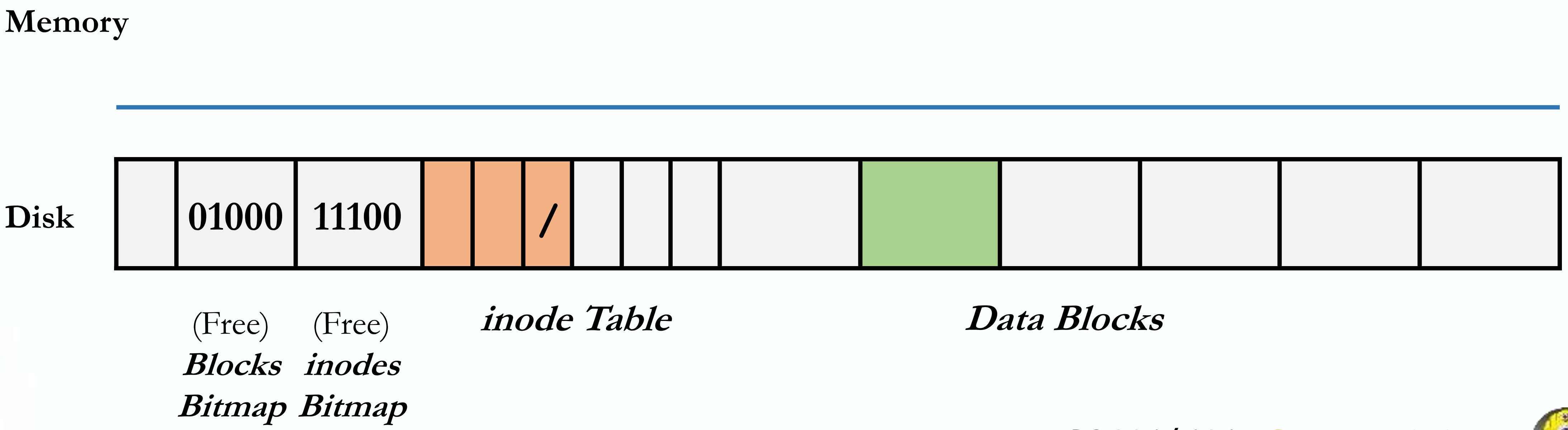


Example: ***File*** **Creation of `/a.txt`**

Initial state:

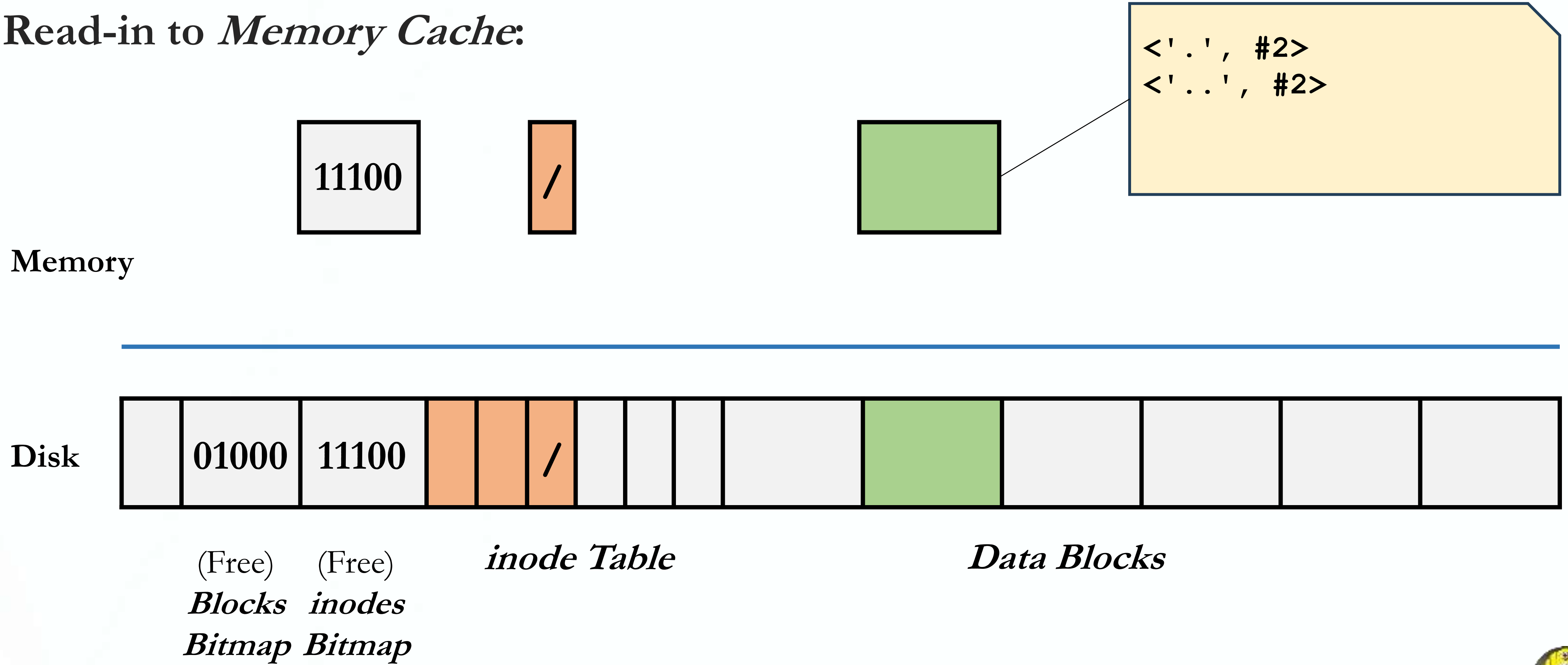
Remember:

- #0 used as a NULL value (indicates an *inode* does not exist), i.e. there is no *inode* #0
- First *inode* is *inode* #1, and is reserved for recording defective *Blocks*)



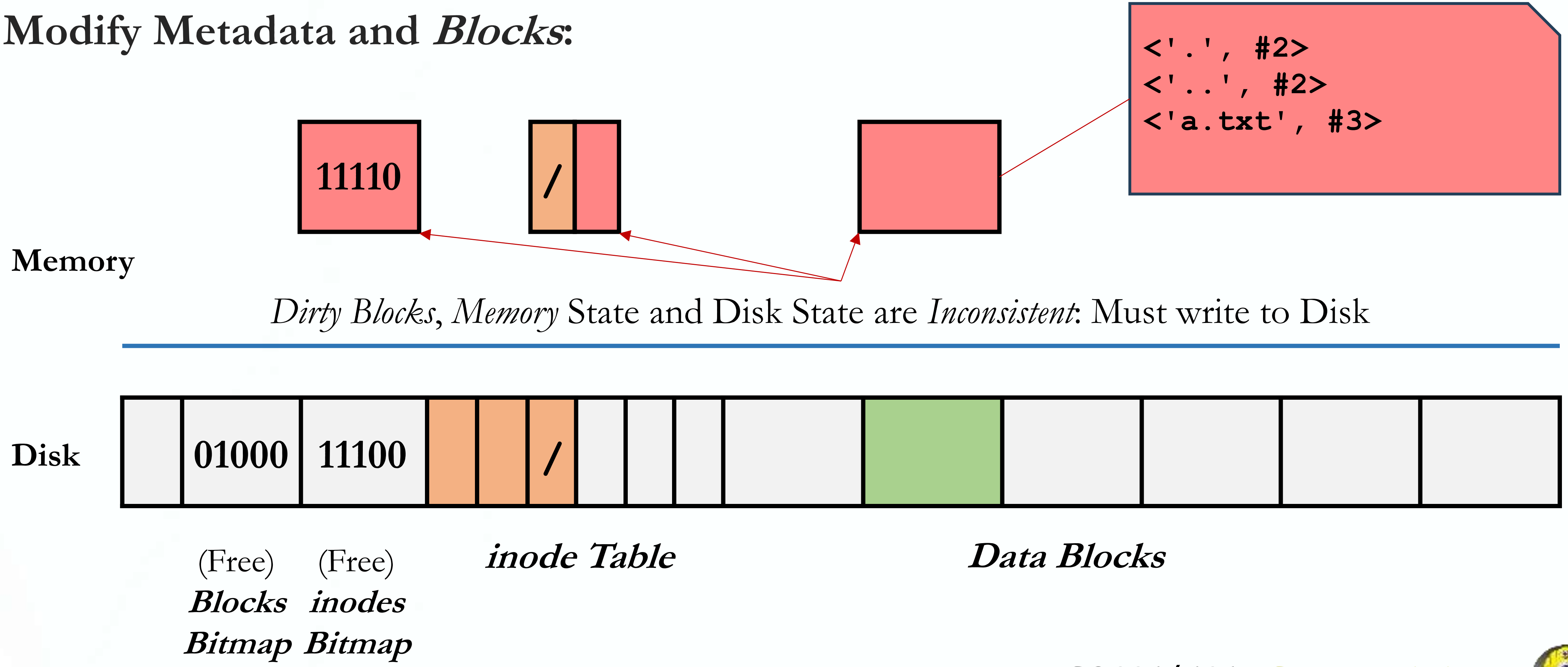
Example: ***File*** **Creation of** `/a.txt`

Read-in to *Memory Cache*:



Example: ***File*** **Creation of `/a.txt`**

Modify Metadata and *Blocks*:



Crash?

Disk Hardware: *Atomically* guarantees writing of single *Sector*

- *Atomic*

  If there is a crash, a *Sector* is either completely written, or none of it is modified

- A *Filesystem* operation may modify multiple *Sectors*

- Crash → *Filesystem* partially updated

Possible Crash Scenarios

File creation dirties 3 *Blocks*

- *inode Bitmap* (**B**)
- *inode* for new *File* (**I**)
- *Data Block* of Parent *Directory* (**D**)

Old and new contents of the *Blocks:*

B = 01000	**B'** = 01100
I = free	**I'** = Allocated, Initialized
D = {`<'.', 2>` `<'..', 2>`}	**D'** = {`<'.', 2>` `<'..', 2>` `<'a.txt', 3>`}

Crash scenarios: **Any subset** can be written

- **B I D**
- **B' I D**
- **B I' D**
- **B I D'**
- **B' I' D**
- **B' I D'**
- **B I' D'**
- **B' I' D'**

Linux Filesystems, FSCK & Journaling

One solution: *File System Consistency checK* (FSCK)

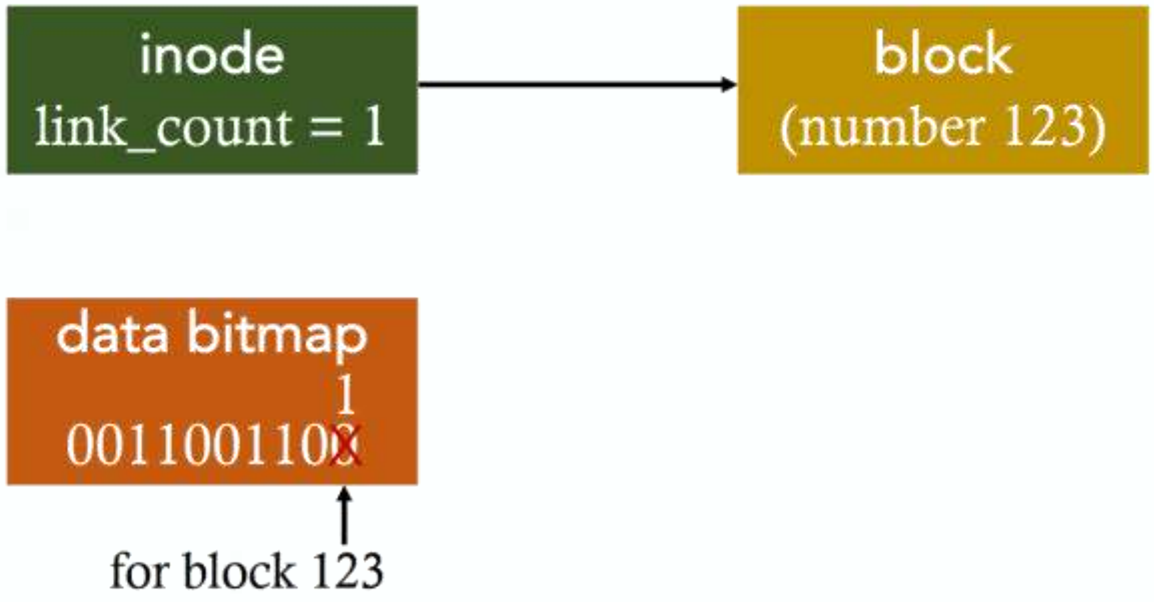
- Upon reboot, scan entire Disk to make *Filesystem Consistent*
  - Scan for *Inconsistencies*, e.g. *inode* Pointers and *Bitmaps*, *Directory Entries* and *inode* reference counts

Advantages
- Simplifies *Filesystem* code
- Can repair more than just crashed *Filesystems* (e.g. *Bad Sector*)

Disadvantages
- Slow (hours-long) to scan large Disks
  - Checking a 600 GB disk takes ~70 minutes
- Cannot fix all Disk Crash Sequences
  - e.g. *File* Creation **B' I D'** – Can this be fixed?
- **Not well-defined *Consistency***

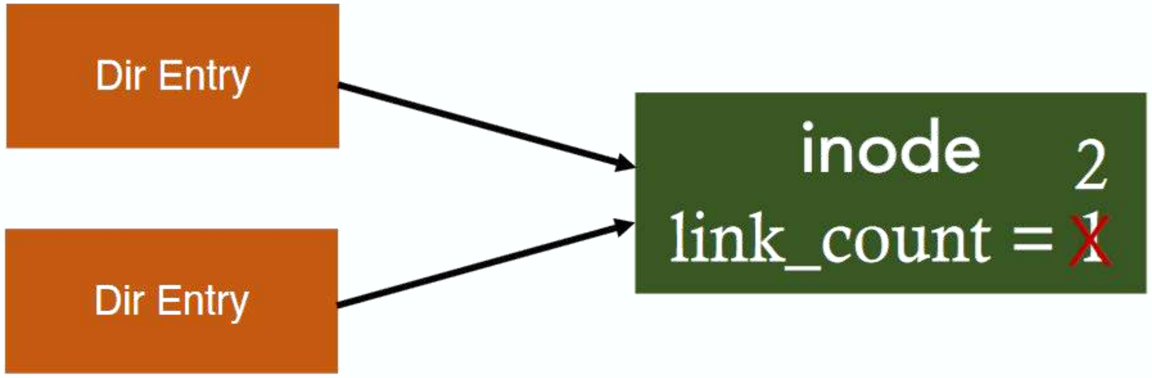
FSCK Example 1:

- Automatically fix:



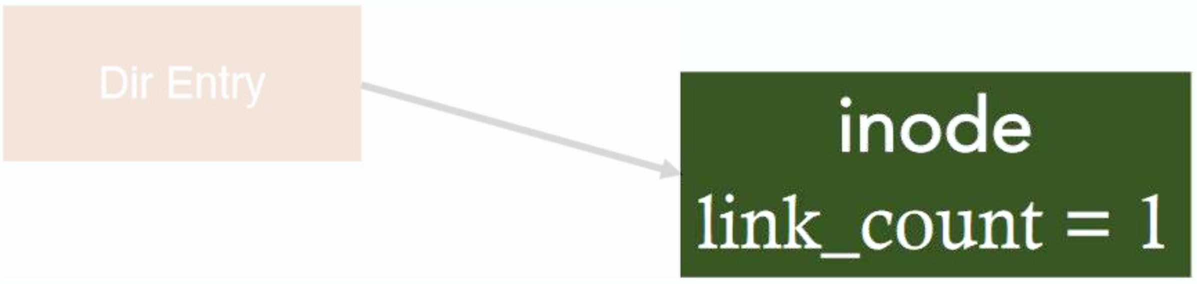
FSCK Example 2:

- Automatically fix:



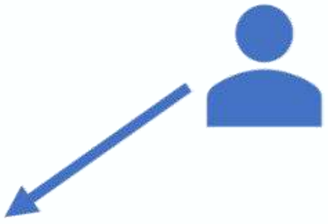
FSCK Example 3:

- Defer to admin to resolve:



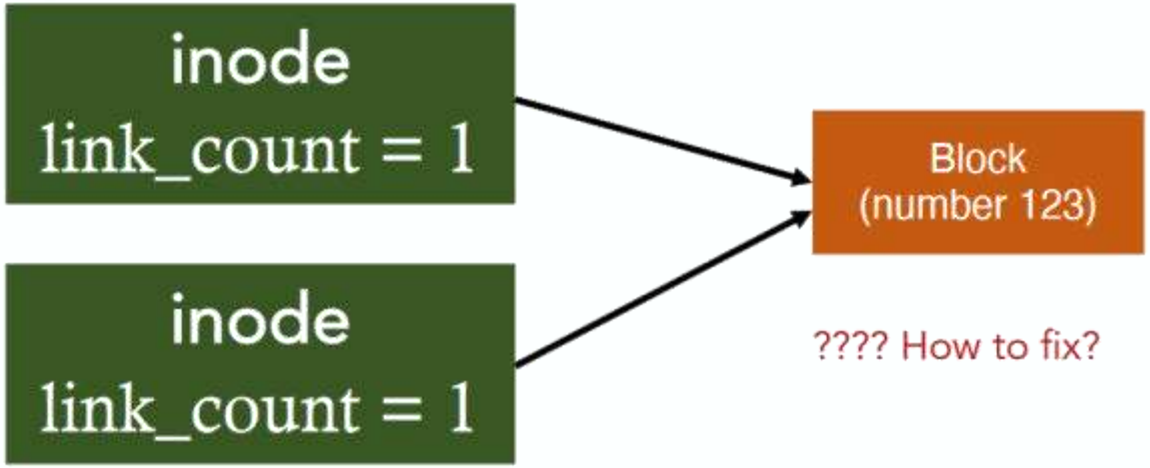
???? How to fix?

```
ls -l /
total 150
drwxr-xr-x   401 18432 Dec 31  1969 afs/
drwxr-xr-x.    2 4096  Nov  3 09:42 bin/
drwxr-xr-x.    5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.   13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.   10 12288 Nov  3 09:41 lib64/
drwx------.    2 16384 Aug  1 10:57 lost+found/
...
```

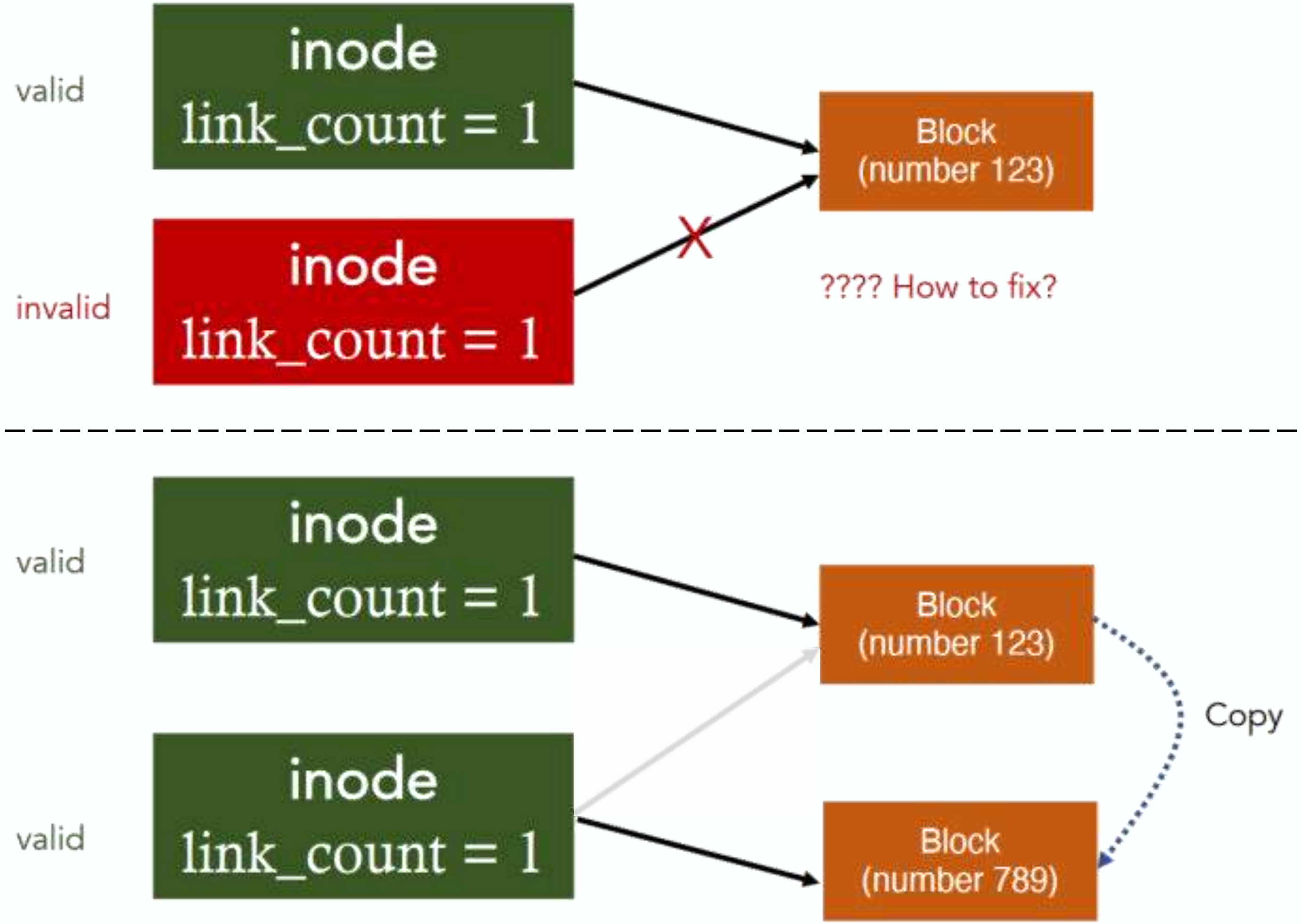


FSCK Example 4:

- Can’t automatically fix:



FSCK Example 4:



Linux Filesystems, FSCK & Journaling

Another solution: *Journaling*

- Leverages "*Write-Ahead Logging*" from Database community

- Persistently write *Intent-to-Log* (or "*Journal*"), then update *Filesystem*
  - Crash before *Intent/Journal* is written == No-op
  - Crash after *Intent/Journal* is written == Redo-op
    - The process is called "*Recovery*"

Advantages:

- No need to scan entire Disk
- Well-defined *Consistency*

Linux *Ext3 Journaling*

- 1 – *Physical Journaling* : Write real *Block* contents of the update to *Log*
  - Four totally ordered steps:

    1) Write *Dirty Blocks* to *Journal* as a single Transaction (**`TxBegin`**, **`I`**, **`B`**, **`D`** *Blocks*)

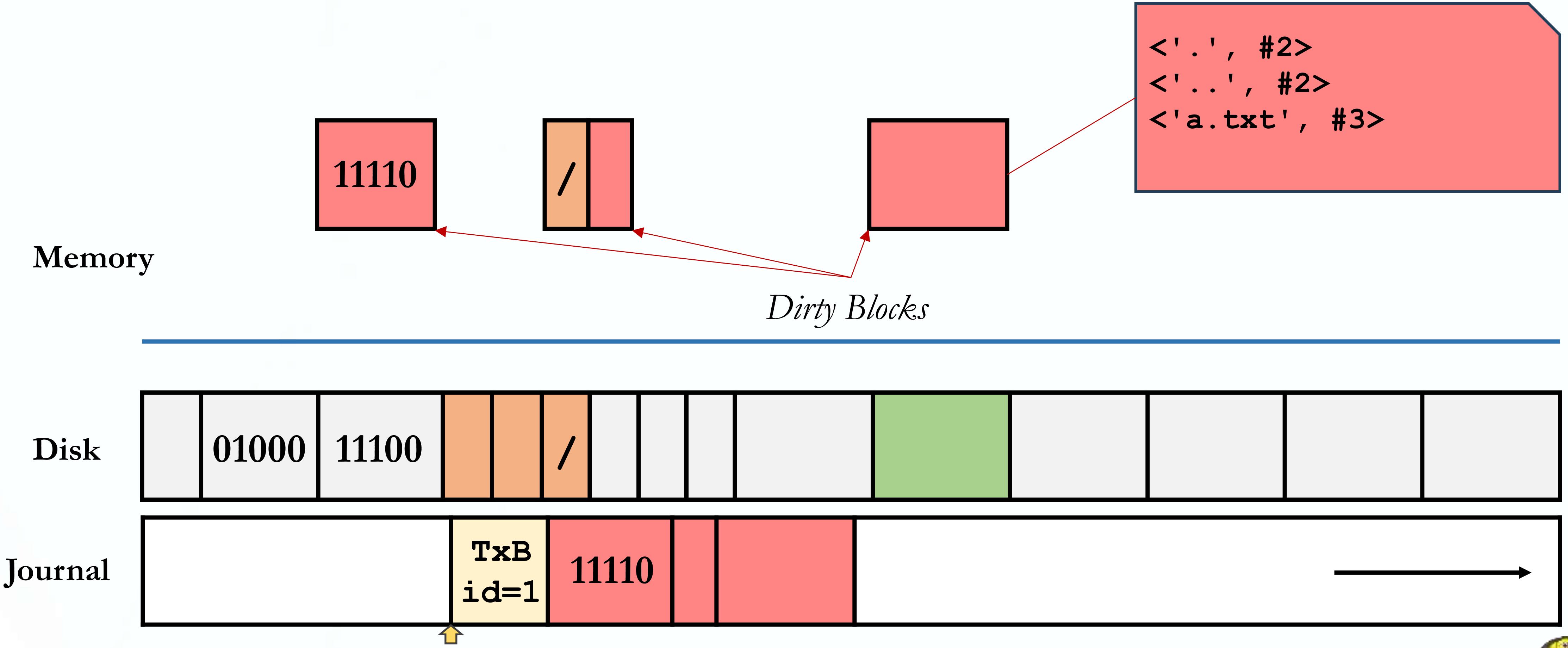
    2) Write **`commit`** *Block* (containing **`TxEnd`**)

    3) Copy *Dirty Blocks* to real *Filesystem* ("*Checkpointing*")

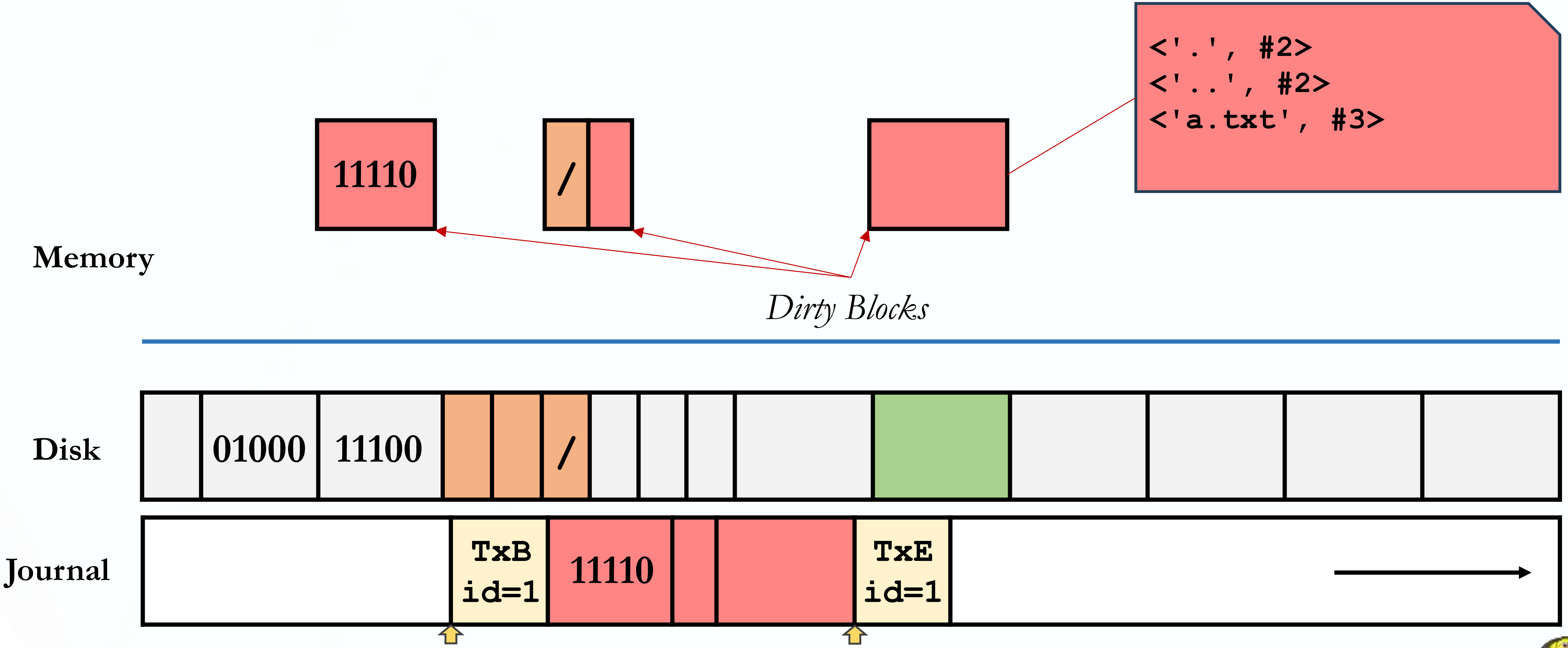
    4) Reclaim the *Journal* Space for the Transaction

- 2 – *Logical Journaling*: Write logical record (logical representation of the intended operation) to *Log*
  - "Add *Directory Entry* F to *Directory Data Block* D"
  - Complex to implement
  - May be faster and save Disk Space

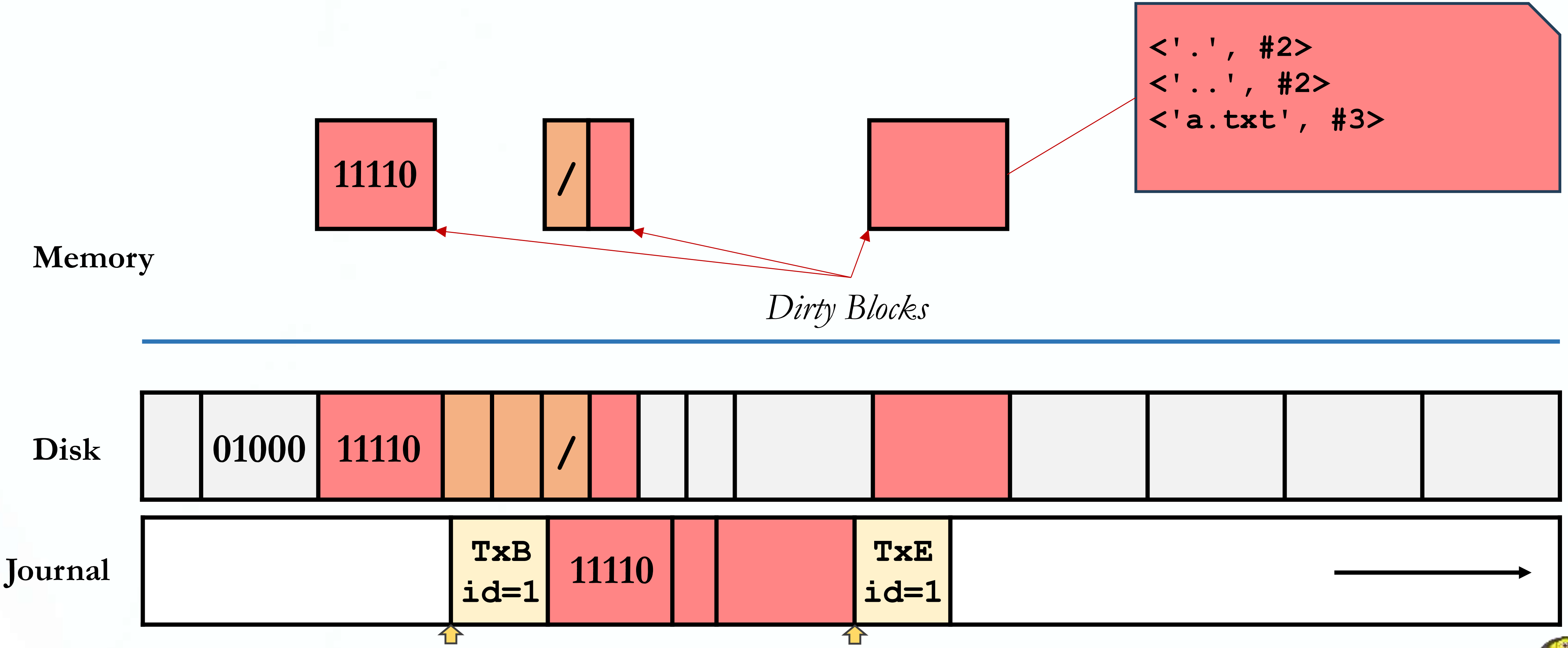
Step 1: Write *Blocks* to *Journal*



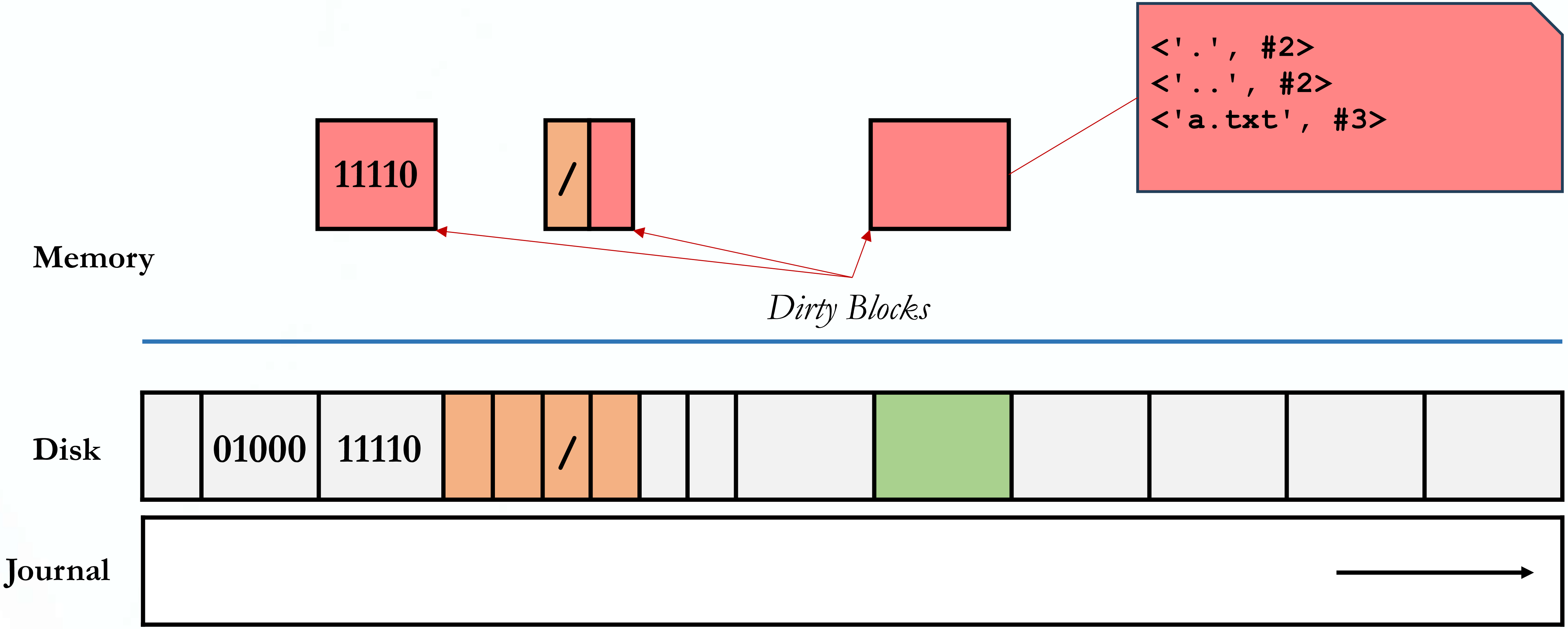
Step 2: Write `commit` Block



Step 3: Copy *Dirty Blocks* to Real *Filesystem*



Step 4: Reclaim *Journal* Space



What if there is a Crash?

Recovery :

- Go through *Log* and "redo" operations that have been successfully committed
- What if:
  - **`TxBegin`** but no **`TxEnd`** in *Log*?
  - **`TxBegin`** through **`TxEnd`** are in *Log*, but no **D** (*Data Block*) in the *Journal*. How?

Journal		TxB id=1	I	B	?	TxE id=1	→

    - Case of deleting a *Directory* (OK)
    - But also *Remember*: Disk Hardware + Sector Mapping + **Scheduling** (Not OK)
      - Why we don't want to merge Step 2 and Step 1
  - **`TxB`**, **`I`**, **`B`**, **`D`**, **`TxEnd`** in *Log*, everything is *Checkpointed* in Disk, but the *Journal* Space has not been reclaimed?

Summary of *Journaling* Write Ordering

- *Journal* writes < *Filesystem* writes
  - Otherwise, Crash → *Filesystem* broken, but no record in *Journal* to patch it up
- *Filesystem* writes < *Journal* clear
  - Otherwise, Crash → *Filesystem* broken, but record in *Journal* is already cleared
- *Journal* writes < `commit` Record write < *Filesystem* writes
  - Otherwise, Crash → record appears fully committed, but contains garbage
    - *Remember*: Why we don't want to merge Step 2 and Step 1 (Not OK)

***Ext3 Journaling* Modes**

- *Journaling* has cost
  - One write = 2 Disk writes, 2 Seeks

Several *Journaling* Modes to balance *Consistency* and Performance:

- *Data Journaling*: *Journal* all writes, including *File* Data
  - Problem: Expensive to *Journal* Data
- *Metadata journaling*: *Journal* only Metadata
  - Used by most Filesystems (IBM JFS, SGI XFS, NTFS)
  - Problem: *File* may contain garbage Data
- *Ordered Mode*: Write *File* Data to real *Filesystem* first, then *Journal* Metadata
  - Default mode for *Ext3*
  - Problem: Old *File* may contain new Data

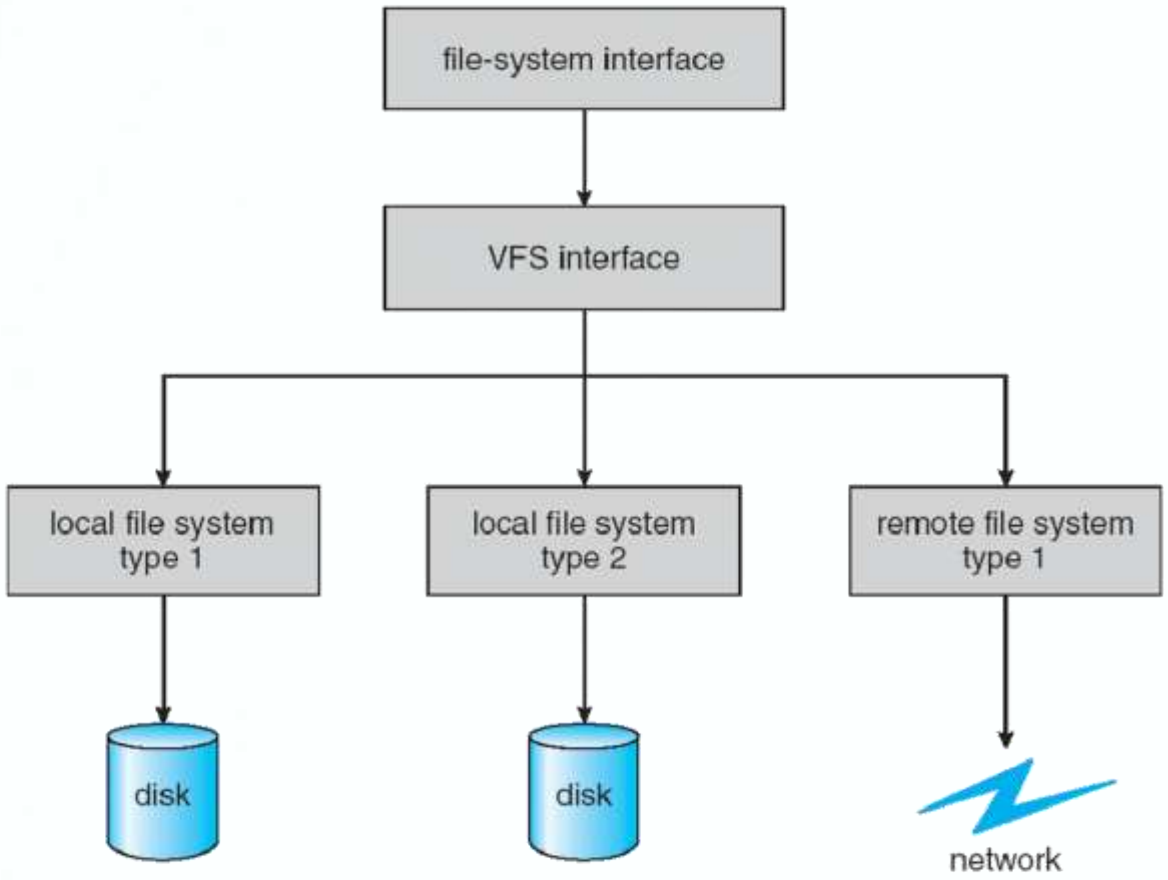
Virtual File System **(VFS)**

- In older times we referred to "the" *Filesystem*
- Nowadays: many *Filesystem* types and instances co-exist

Virtual File System:

- A *Filesystem* abstraction layer that transparently and uniformly supports multiple *Filesystems*
  - A VFS specifies an Interface
  - A specific *Filesystem* implements this Interface
    - Often a struct of function Pointers
  - VFS dispatches *Filesystem* operations through this interface
    - e.g. `dir->inode_op->mkdir();`

VFS Interface Schematic



Key Linux VFS Data Structures

- `struct file`
  - Information about an open *File*
  - Includes current position (*File* offset Pointer)
- `struct dentry`
  - Information about a *Directory Entry*
  - Includes *name* + *inode#*
- `struct inode`
  - Unique descriptor of a *File* or *Directory*
  - Contains Permissions, Timestamps, Block map (Data)
  - *inode#*: integer (unique per mounted *Filesystem*)
  - Pointer to *Filesystem*-specific *inode* structure
    - e.g. `struct ext2_inode_info`
- `struct superblock`
  - Descriptor of a mounted *Filesystem*

CS-446/646

Time for Questions !