CSC369 Lecture 10

Larry Zhang, November 23, 2015

Announcements

- ➔ Ex14 will be posted later today. In this week's tutorial you'll get marks for both Ex13 and Ex14
- ➔ If you were in the last tutorial but cannot make it to the next tutorial, let me know so you get credit for last week.
- ➔ Start **really** working on A3 if you haven't
- ➔ Keep looking up **the spec** whenever something is not clear, instead of trial-and-error.
- ➔ Asking questions on Piazza helps everyone.
- ➔ Course evaluation
  - ◆ http://uoft.me/course-evals

A3 tips

Line number statistic from a working solution

```
 35 lines in   ext2_ls.c
 81 lines in   ext2_cp.c
 72 lines in   ext2_mkdir.c
 84 lines in   ext2_ln.c
 88 lines in   ext2_rm.c
 11 lines in   Makefile
326 lines in   ext2_utils.c
```

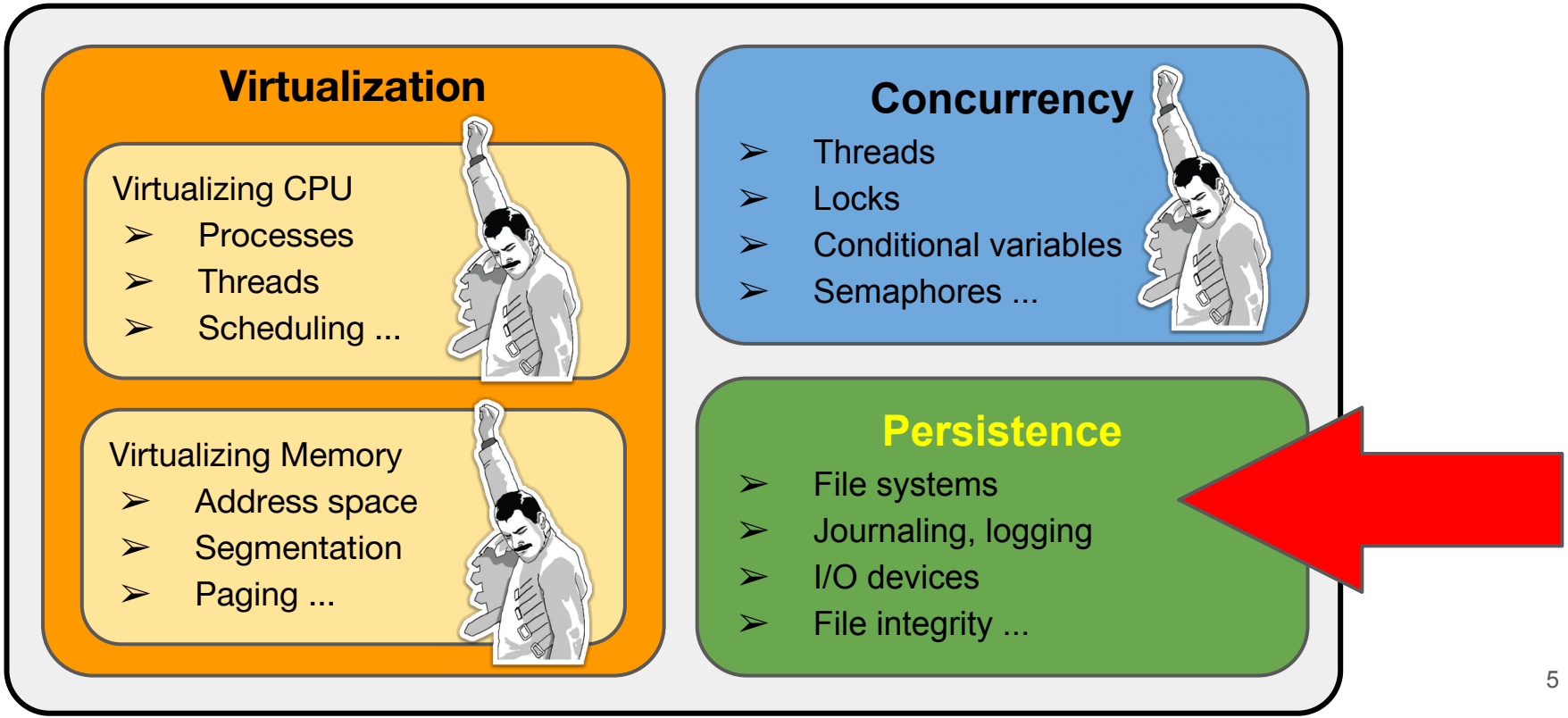
another required file: **readme.txt**, describe what has been implemented and what has not, especially important things are partially finished.

Put a lot of stuff into the shared utility file

A3 tips

- ➔ use `sizeof(struct_name)` to get the size of a struct.
- ➔ An ext2 visualizer: http://wisdi.me/ext2minator/
- ➔ When you read a string (like file name), need to know the **length** to read.
- ➔ Use `mmap` to open an fs, as in `readimage.c`
- ➔ ext2_dir_entry and ext2_dir_entry_2, you can use either one, but ext2_dir_entry_2 is more readable, so recommended.

We are here


Virtualization
Virtualizing CPU
Processes
Threads
Scheduling ...
Virtualizing Memory
Address space
Segmentation
Paging ...
Concurrency
Threads
Locks
Conditional variables
Semaphores ...
Persistence
File systems
Journaling, logging
I/O devices
File integrity ...

We have learned ...

- ➔ File system interface
  - ◆ files, directories, open(), write(), lseek(), rename(), etc...
- ➔ Simple File system implementation
  - ◆ what data structure is used organized data on disk (inode, bitmap, etc.)
  - ◆ what happens when we read / write a file.
- ➔ Make file systems faster
  - ◆ caching, buffering, FFS
- ➔ We haven’t really talked about **persistence** yet.

Persistence

- ➔ Data stay after you power-off the computer.
  - ◆ that’s just because the physics of the disks
- ➔ If the computer crashes during the file system read/write, data on disk are still consistent.
  - ◆ this is a more interesting problem for OS
  - ◆ i.e., the **crash-consistency problem**

Example: a crash scenario

BONUS!

- a 4KB file on the disk, within a single data block
- then we write an additional 4KB to that file, i.e., adding a data block to the file
- if everything went well ...

inode bitmap				data bitmap				inode table				data blocks							
0	1	0	0	0	0	0	0		I1			0	1	2	3	D4	5	6	7
0	0	0	0	1	0	0	0												

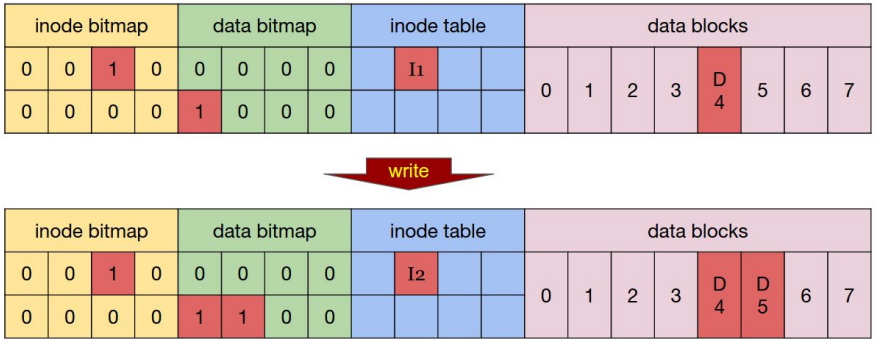
write

three things to be updated (data bitmap, inode, data block)

inode bitmap				data bitmap				inode table				data blocks							
0	1	0	0	0	0	0	0		I2			0	1	2	3	D4	D5	6	7
0	0	0	0	1	1	0	0												

if the computer crashed during the write ...

BONUS!



What are all the possible **inconsistent states** that the FS can be in after the crash?

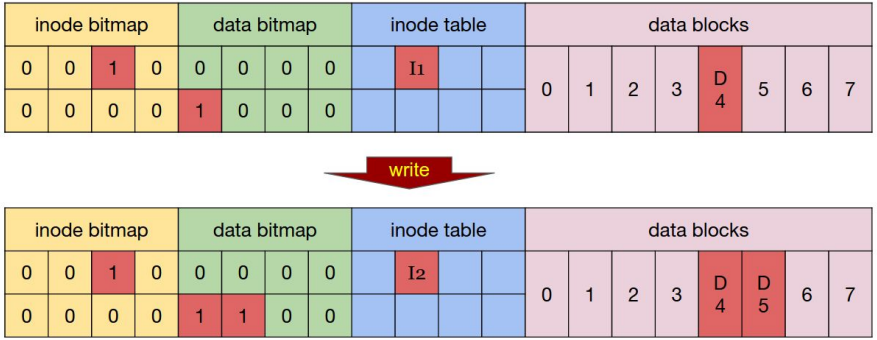
Only one thing updated

- ➔ Case 1: Just the **data block** is updated, but not the data bitmap and inode
- ➔ Case 2: Just the **inode** is updated, but but not the data bitmap and data block
- ➔ Case 3: Just the **data bitmap** is updated, but not the inode and data block

Only two things updated

- ➔ Case 4: **inode** and **data bitmap** updated, but not data block
- ➔ Case 5: **inode** and **data block** updated, but not data bitmap
- ➔ Case 6: **data bitmap** and **data block** updated, but not onde

Let's analyse these cases one by one.



BONUS!

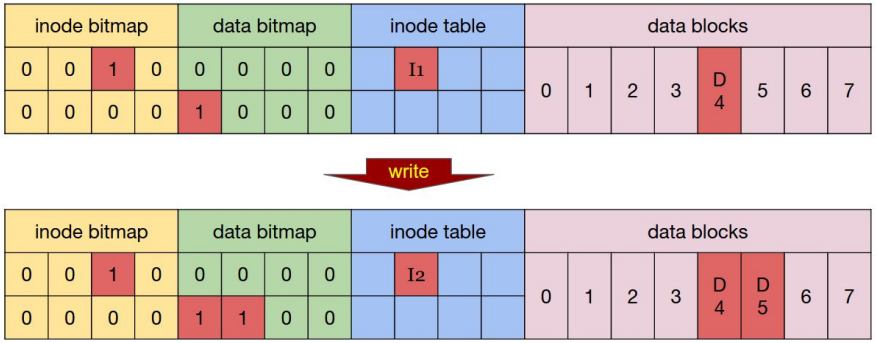
Case 1: only **data block** is updated

➔ the data is on disk

➔ but nobody ever knows, because inode and data bitmap are not updated

➔ the file system itself is **still consistent**, it is just like nothing happened

➔ No need to fix anything



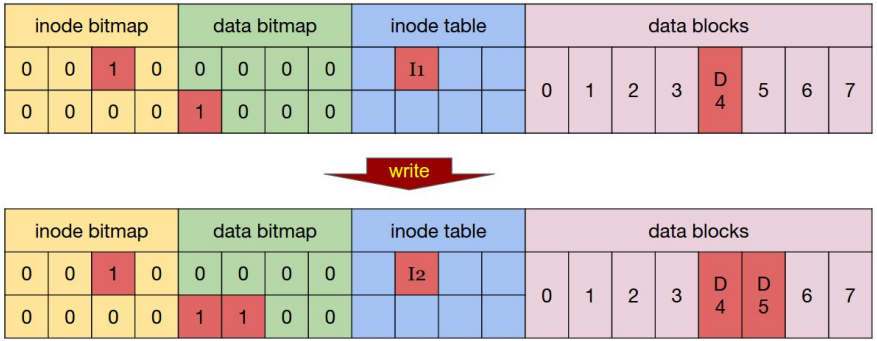
Case 2: only **inode** is updated

➔ **inode** has data block pointer pointing to an **unwritten** data block

➔ if we trust the inode, we will read **garbage** data

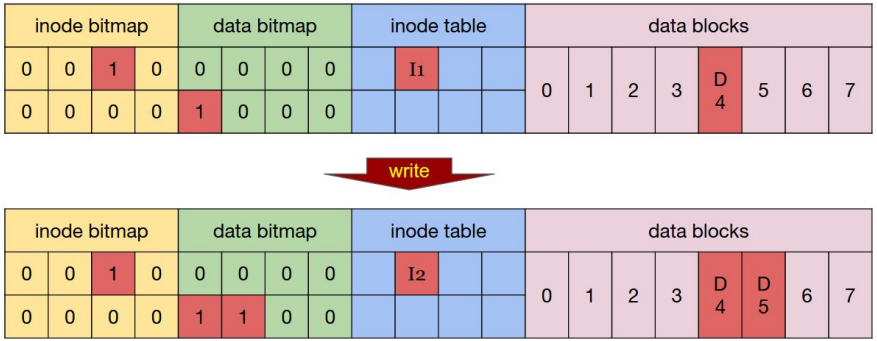
➔ also there is **inconsistency** between **data bitmap** and the **inode**

- inode says that the data block #5 is used, but data bitmap say it is not
- if not fixed, could allocate block #5 again and overwrite its data by mistake



Case 3: only **data bitmap** is updated

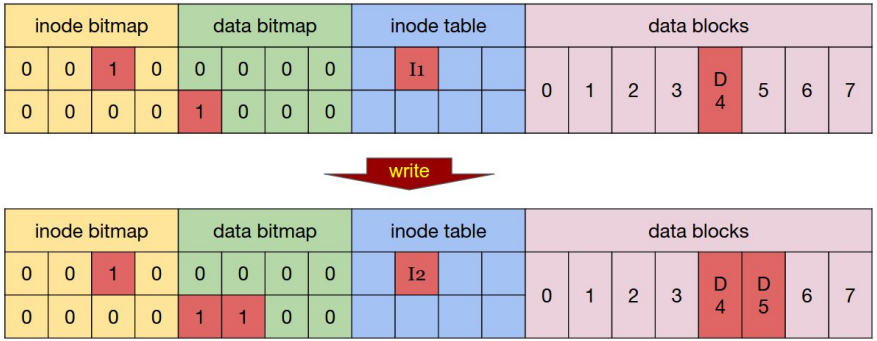
- ➔ **inode** is not pointing data block #5, so no risk of reading garbage data
- ➔ **data bitmap** says data block #5 is used, but in fact it is not
- ➔ data block #5 will **never** be used again
- ➔ this is called a **space leak**



Case 4: only **inode** and **data bitmap** are updated

➔ will read garbage data from block #5 again

➔ but the file system doesn't even realized anything wrong, because the **inode** and the **data bitmap** are **consistent** with each other.



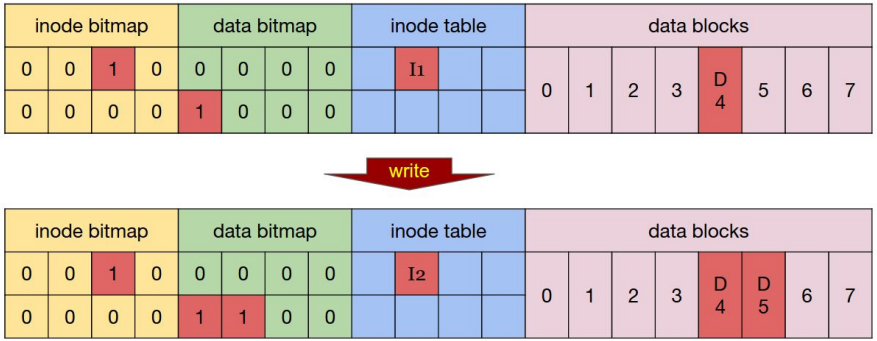
BONUS!

Case 5: only **inode** and **data block** are updated

➔ will NOT read garbage data

➔ but again, data bitmap and inode are **inconsistent** between each other

- inode says that the data block #5 is used, but data bitmap say it is not
- if not fixed, could allocate block #5 again and overwrite its data by mistake

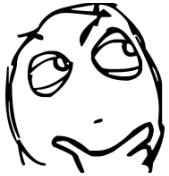


BONUS!

Case 6: only **data bitmap** and **data block** are updated

➔ again, **inconsistency** between **inode** and **data bitmap**

➔ we know data block #5 is used, but will never know which file uses it



We wish file system updates were **atomic**, but they are not.

The there are many different problems that can occur if the computer crashes during a file system update.

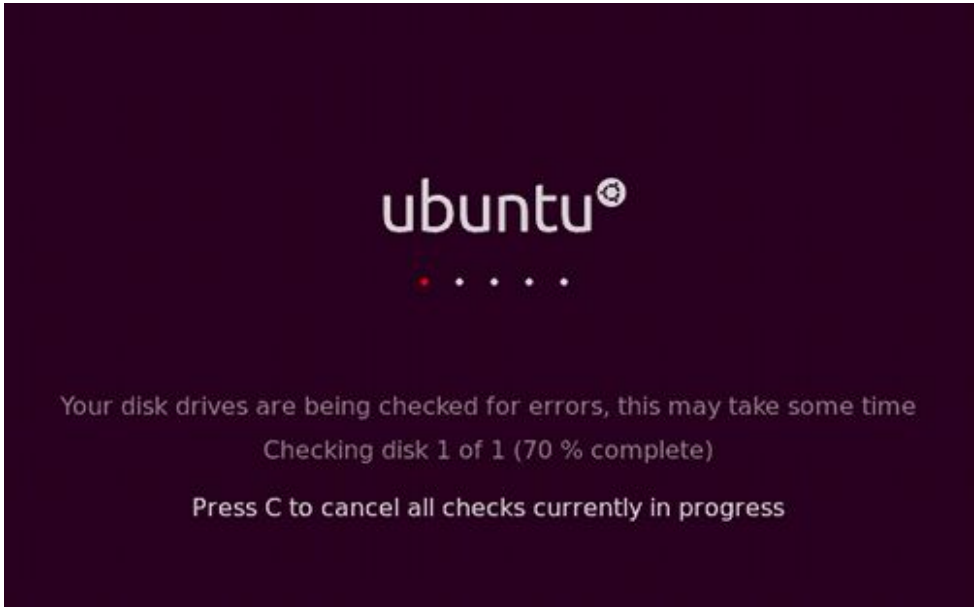
➔ inconsistency in file system data structure
➔ reading garbage data
➔ space leak

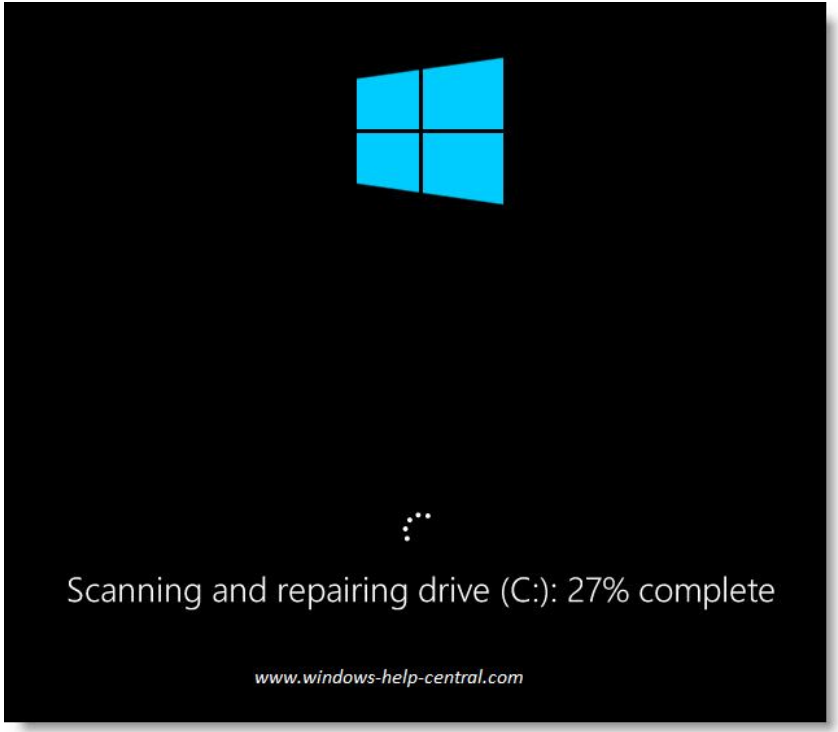
We call them **crash-consistency problems**

We need to fix these problems.

crash consistency problem

Solution #1: FSCK


ubuntu
Your disk drives are being checked for errors, this may take some time
Checking disk 1 of 1 (70 % complete)
Press C to cancel all checks currently in progress


Scanning and repairing drive (C:): 27% complete
www.windows-help-central.com

FSCK (file system check)

A UNIX tool that **scans** the whole disk, **find** inconsistencies and **repair** them.

➔ runs before the file system is mounted, e.g., when booting
➔ when crash happens, let it be, and fix it later (when rebooting)

Typical checks performed

➔ all blocks pointed to by inode or indirect block must be marked “used” in bitmap
➔ all used inodes must be in some directory entry
➔ inode reference count (link count) must match
➔ no duplicate data pointers.
➔ etc.

Limitations of FSCK

- ➔ A working FSCK requires very complicated and detailed knowledge of the file system, and is hard to be implemented correctly.
- ➔ It only cares about the internal consistency of the file system, and does NOT really care about lost data (e.g., Case 1 and Case 4 in previous slides)
- ➔ Bigger problem: it is tooooo slow
  - ◆ with a large hard drive, it can easily take hours to finish FSCK
  - ◆ it’s just a bit irrational: scan the **whole disk** no matter how small an inconsistency needs to be fixed

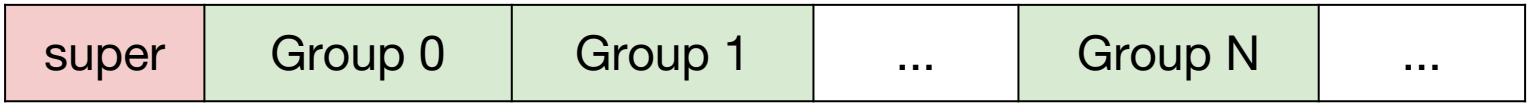
crash consistency problem

Solution #2: Journaling

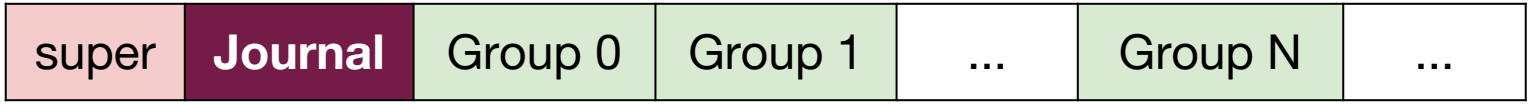
(a.k.a. write-ahead logging)

Additional space in the on-disk data structure

Ext2



Ext3 (which is basically Ext2 plus journaling)



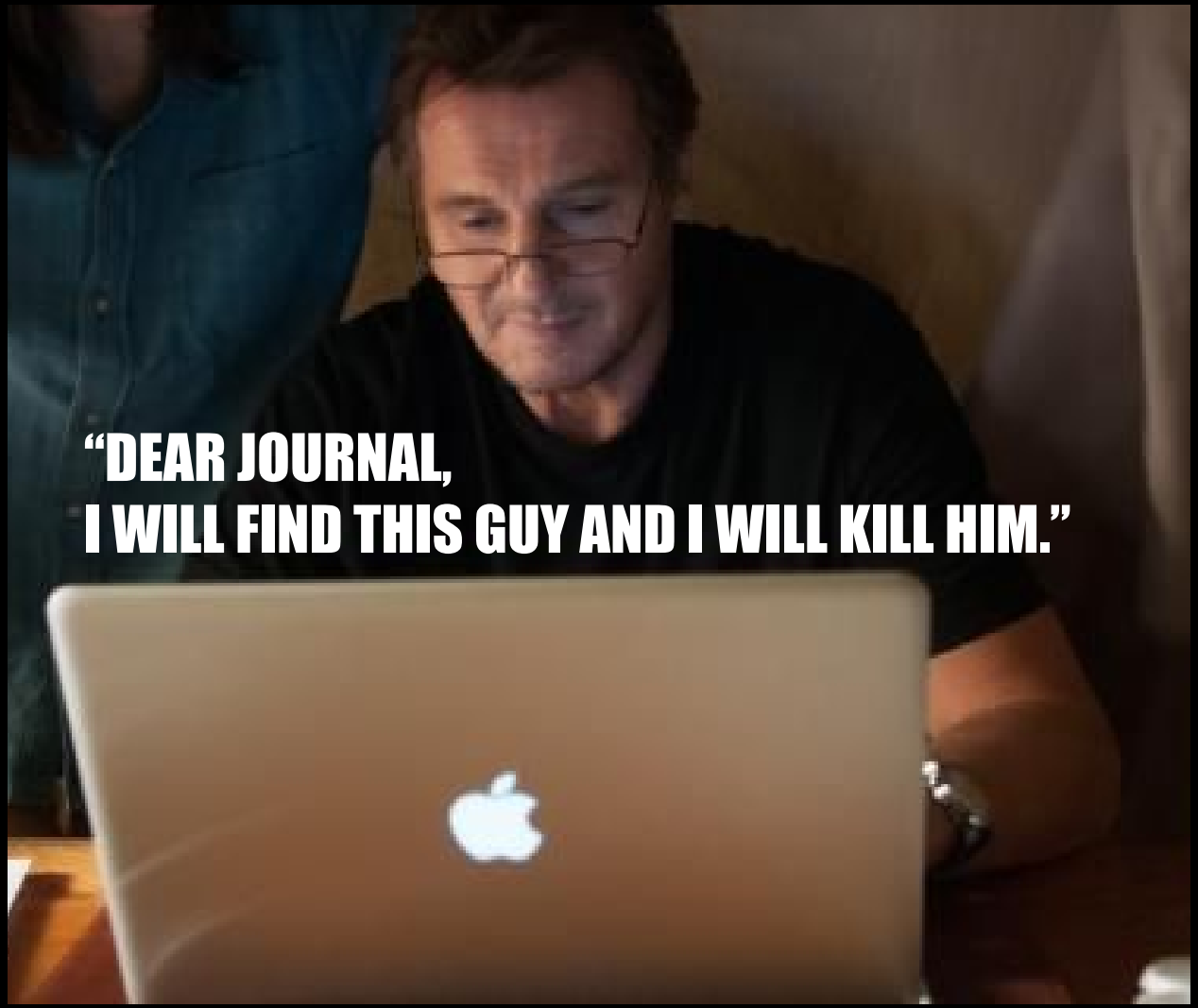
What’s in that journal

When updating the disk, **before** making the actual writes to disk, the FS first writes a little “note” about what it is about to do, to the **journal** (or log, at well known location on disk) .

So failures become recoverable:

➔ if crash happens during actual write (journal write completed), then we can look at the journal and **replay** the actual writes hence **recover** the data.

➔ if crash happens even before journal write finishes, then it doesn’t matter since the actual write has NOT happened at all, nothing is inconsistent.


“DEAR JOURNAL,
I WILL FIND THIS GUY AND I WILL KILL HIM.”

The structure to be written to the journal

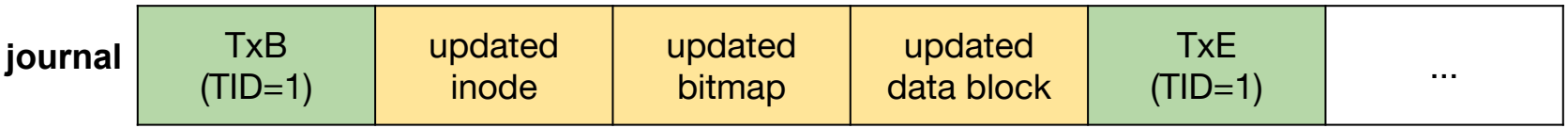
Transaction:

➔ starts with a “**transaction begin**” (TxB) block, containing a **transaction id** (TID)

➔ followed by blocks with the exact content to be written

- physical logging: putting exact physical content
- logical logging: putting more compact logical representation

➔ ends with a “**transaction end**” (TxE) block, containing the TID



The sequence of operations to finish a write

1. **Journal write**: write the transaction, including the **TxB**, **all pending data and metadata updates**, and the **TxE**, to the journal; wait for these writes to complete
2. **Checkpoint**: actually write the pending data and metadata to to their final locations in the file system.

Can this go wrong in some way?

Hint 1: crash can occur during journal writes
Hint 2: when having a batch of writes, the disk may perform some disk scheduling, so the writes in the batch can happen in any order

The actual schedule could be TxE, inode, TxB, bitmap, (**CRASH!**), data; then the journal becomes...

TxB	inode	bitmap	**garbage**	TxE	...

Looks like totally valid journal record to the FS, but has garbage data!

The better sequence: 3 steps

1. **Journal write**: write the transaction, including the TxB, all pending data and metadata updates, **NOT including the TxE**, to the journal; wait for these writes to complete
2. **Journal commit**: write the **TxE** block (a.k.a. transaction commit block); now tractions is said to be committed
3. **Checkpoint**: actually write the pending data and metadata to to their final locations in the file system.

Nuance: hard disk guarantees **atomic** write of a 512-byte sector, the TxE block should fit in a 512-byte sector, so the commit is either fully completed or not completed at all.

So, with journal, how to **recover** from a crash

If crash happens before the transaction is **committed** to the journal / log

➔ simply skip the pending update

if crashed happens during the **checkpoint** step.

➔ when booting, **scan the journal** and lookup for committed transactions (much faster than fsck's scanning the whole disk)

➔ **replay** these transactions

➔ after replay the file system is guaranteed to be consistent, then we can mount it and do other stuff

➔ this is also called **redo logging**

How much space do we need for the journal?

We need to write something to journal for every single disk update, so the journal need to be **huge**, right?

Not really, after checkpoint the transaction in the journal is not useful anymore, so the space can be freed.

So there are indeed 4 steps

1. journal write
2. journal commit
3. checkpoint
4. free

Typical data structure for the journal: **circular log**

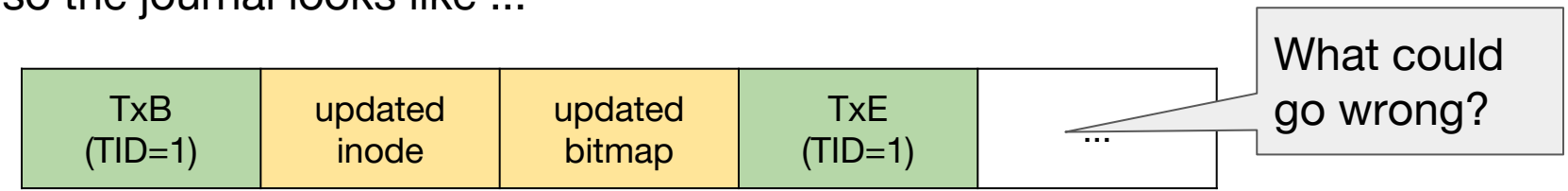
Metadata journaling

Recovery is fast with journaling, but the normal operations are slower, because for **every** update we need to write to the journal first then do the update.

➔ writing time is at least doubled, even worse if journal writing breaks sequential writes and cause jump-back-and-forth between journal and data region.

➔ metadata journaling is just like data journaling (what we just described), except that we only write metadata (NOT data) to the journal.

➔ so the journal looks like ...



Metadata journaling

Say we write data after checkpointing metadata, then write data, if crash occurs before all data is written, the inodes will point to garbage data.

How to solve this problem?

Write data before writing metadata journal!

1. write data, wait until it completes
2. metadata journal write
3. metadata journal commit
4. checkpoint metadata
5. free

If write data fails, then no metadata is written at all, like nothing happened.

if data write succeed, but metadata write fails, still like nothing happened.

if metadata write succeeds, data must be available.

Summary of Journaling

- ➔ Journaling provides file system consistency
- ➔ Time complexity of recovery is **O(size of journal)**, instead of **O(size of disk volume)** in fsck.
- ➔ Widely adopted by most modern file systems, including Linux's Ext3, Ext4, ReiserFS, IBM's JFS, SGI's XFS, and Windows' NTFS
- ➔ Metadata journaling is the most commonly used, since it reduces the amount of traffic to the journal while provide reasonable consistency guarantees.

Log-structured File System (LFS)

a different approach to updating the disk

Log-structured File Systems (LFS)

Invented by John Ousterhout and Mendel Rosenblum in the early 90's.

Goals:

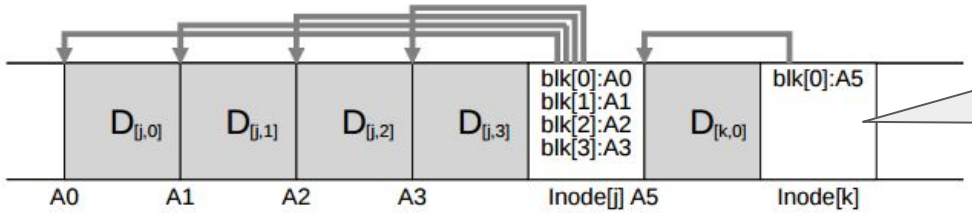
➔ Memory size is growing, so more data can be cached in memory to increase performance

➔ Sequential writes to disk are much faster than random writes, so try to do sequential writes as much as possible.

➔ Existing file system work poorly on many common workloads (reality-inspired optimization)

What does LFS do?

- ➔ Write all file system data and metadata into a continuous log
- ➔ data and metadata are buffered in an in-memory **segment**, and get sequentially written to disk when the segment is full
- ➔ new version of data does not overwrite the old version, it always only use unused space of the disk (so it's like a log).
- ➔ you can undo a write by going to the older version of data
- ➔ the disk layout could look like the following:



Note: inode blocks are all over the place!

Read from an LFS

Write is easy in LFS, but read is more complicated

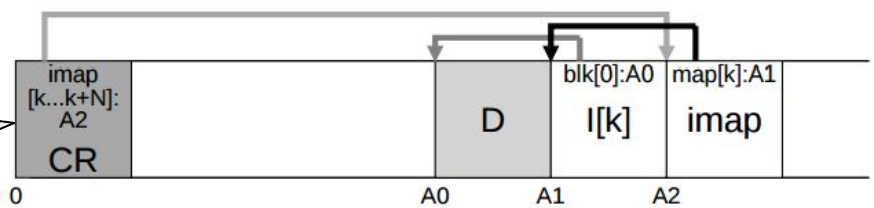
➔ unlike the original UNIX file system, where inode tables are at well-known locations defined in superblock, the inode blocks in LFS are all over the place.

➔ so we need **inode maps (imap)** to keep track of the addresses of the inodes

➔ the imaps are also on-disk, and are also all over the place.

➔ so we have a **checkpoint region**, which keeps track of the address of imaps

It's called log-structured because the structure is like a log (journal), i.e., data, metadata, then checkpoint.



A tricky issue in LFS: **Garbage Collection**

- ➔ LFS repeatedly writes latest version of a file to new locations on disk.
- ➔ Older versions of files (garbage) are scattered throughout the disk.
- ➔ LFS periodically find these old dead version of file data and clean them, thus make the blocks free again for subsequent writes.
- ➔ The cleaning is done on a segment-by-segment basis, since the segment are large chunks cached in memory, it avoids the situation where cleaning yields small holes of free space.
- ➔ GC in LFS is an interesting research question, series papers were published looking at its performance.

Summary of LFS

- ➔ A new approach of updating the disk.
- ➔ Instead of overwriting in place, always write to unused portion of disk, and reclaims old space later from garbage collection.
- ➔ Upside: very efficient writing, approaching full bandwidth of disk
- ➔ Downside: generates garbage, and slightly inefficient reads (more levels of indirection)

Redundancy

another approach to data persistency

Redundancy

- ➔ We have talked about how to use journal to recover from a crash during write, by replaying the logged writes.
- ➔ this relies the assumption that the disk is still usable after rebooting.
- ➔ In reality, disks can just break, and the data on that disk is simply lost.
- ➔ The solution: have more than one copies of the data, i.e., redundancy

Redundant Array of Inexpensive Disks

a.k.a. **RAID**

Goals of RAID

- ➔ **Performance**: use multiple disks in parallel to speed up I/O performance
- ➔ **Capacity**: more disks combined provide more space
- ➔ **Reliability**: with data spread across multiple disks with redundancy, RAID can tolerate the loss of a disk keep operating as if nothing were wrong.
- ➔ **Transparency**: from outside, a RAID just looks like a big disk with good performance, large capacity and great reliability. You can easily unplug a regular disk and replace it with a RAID.

Fault model

In our discussion, we assume the following **fail-stop** fault model

➔ a disk has only two possible state: **working** or **failed**
➔ when a disk fails, we can detect it immediately, i.e., the fail is not silent

Real-world faults can be more complicated than this.

RAID Level 0: Striping

No redundancy yet, simply stripe the blocks across different disks

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Each row is called a stripe

Disk 0	Disk 1	Disk 2	Disk 3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

Another way of striping. Difference?

RAID-0 reliability issue: data is lost whenever a disk fails

RAID Level 1: Mirroring

Have more than one copy of each block

When reading block 5, can choose either Disk 2 or Disk 3

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Issue with RAID-1: bad disk space utilization, what you can store is only half of the total physical capacity

Interlude: Parity Bit

Assume A, B and C are either 0 or 1.

What is the Boolean expression that indicates whether there are an odd number of 1's in A, B and C

P = A xor B xor C

Suppose A = 1, B = 0, C = ??, P = 1. Do we know what C is?

➔ C = 0

What if P = 0?

➔ C = 1

Given parity bit P, we can recover if there is only one bit missing from the input.

RAID Level 4: Save space with parity

Use less space to achieve redundancy, compared to mirroring.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

A dedicated disk for parity checks

- ➔ Each parity bit keeps, among the corresponding bits in the blocks of the same stripe, whether there are an odd or even number of 1's. (Done by XOR).
- ➔ If one disk on the stripe fails, we can recover the lost bit.
- ➔ If the parity of the remaining bits is the same as before, then we lost a 0, otherwise we lost a 1.

RAID Level 4: a problem

Suppose we are writing to Block 4 and Block 13 simultaneously, since they are on different disks we expect good performance due to parallelism.

But no! Both writes cause writes on the parity blocks on Disk 4, which totally spoiled the parallelism.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

The disk with parity blocks is so frequently written that it becomes the bottleneck.

RAID Level 5: Rotating Parity

Distribute the frequently-written parity blocks to different disks.

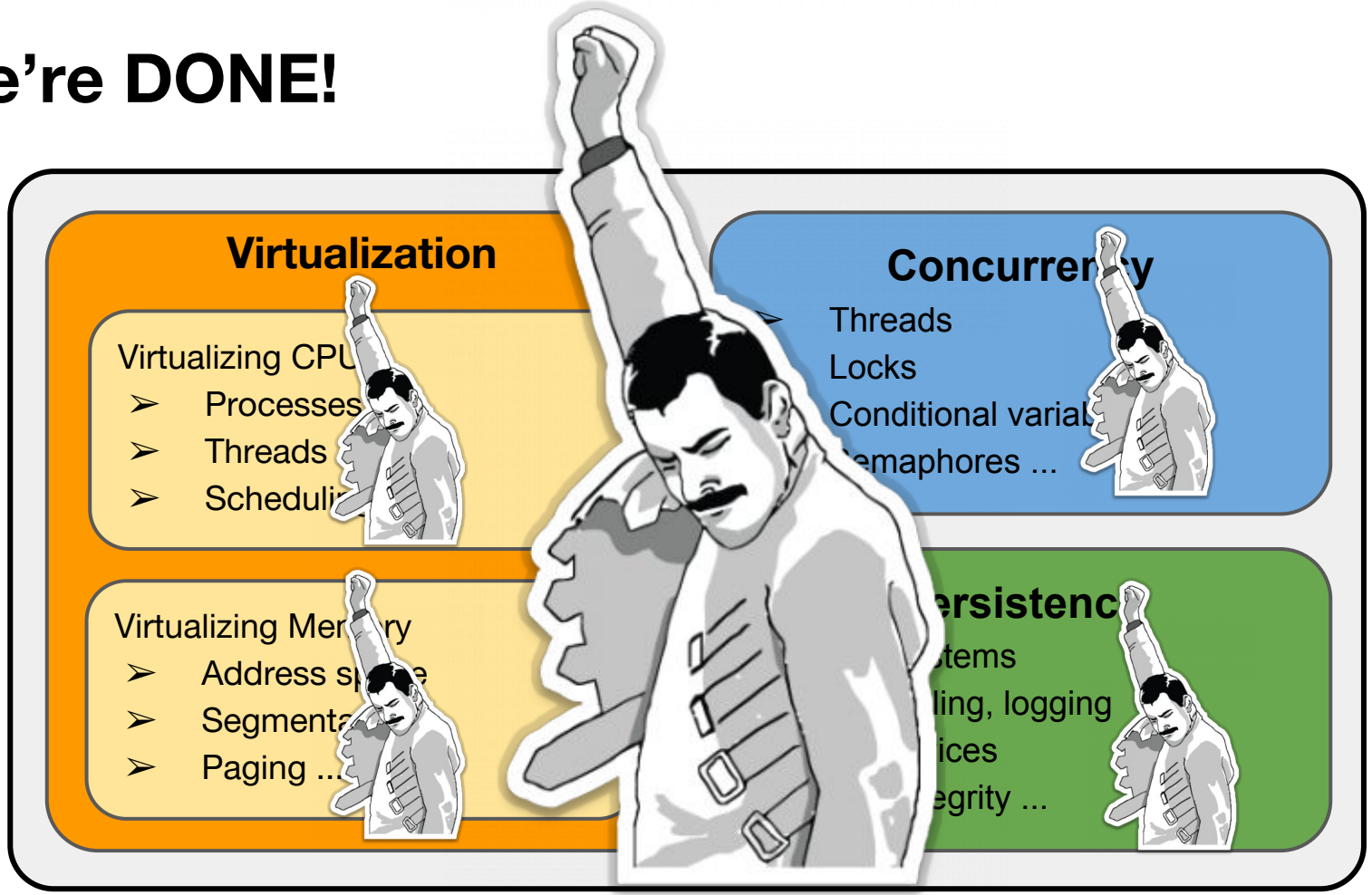
More even workload across disks, better performance.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Summary of RAID

- ➔ Which level of RAID to use depends on what's important for the users, and what the workload is like
- ➔ We are just scratching the surface of RAID, for more details read the textbook.

We're DONE!


Virtualization
Virtualizing CPU
Processes
Threads
Scheduli
Virtualizing Mer ry
Address sp e
Segmenta
Paging ...
Concurrer y
Threads
Locks
Conditional varial
emaphores ...
ersistenc
tems
ling, logging
ices
egrity ...

Next Week

➔ Some stuff that we haven’t talked about

➔ Final exam review

This week’s tutorial:

➔ More exercise for A3