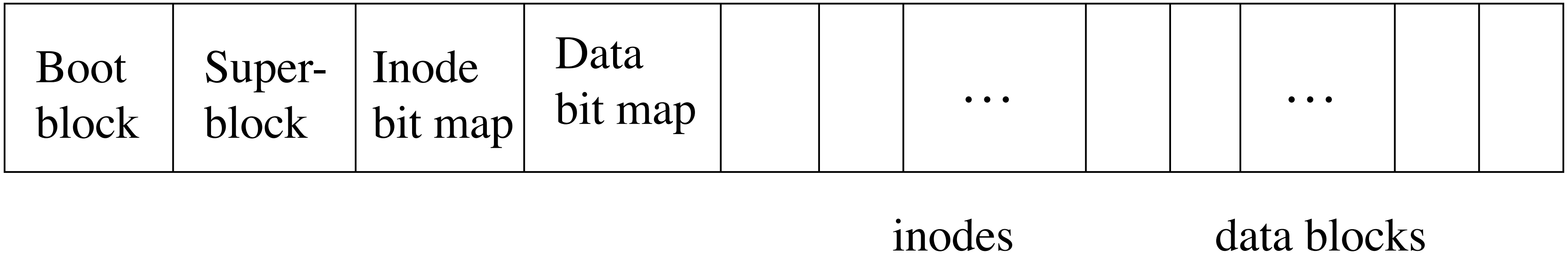
Overview

- This Lecture
  - File systems, performance and durability
  - Source: ULK ch 18 & ch 12, Rethink the sync

File systems

- Implementation of UNIX file systems
  - Basic units: blocks, default size 4096, can be adjusted when a file system is created.
  - Two structures are created for a file system
    - Superblock (with backups) and inodes



Boot block

Super-block

Inode bit map

Data bit map

…

…

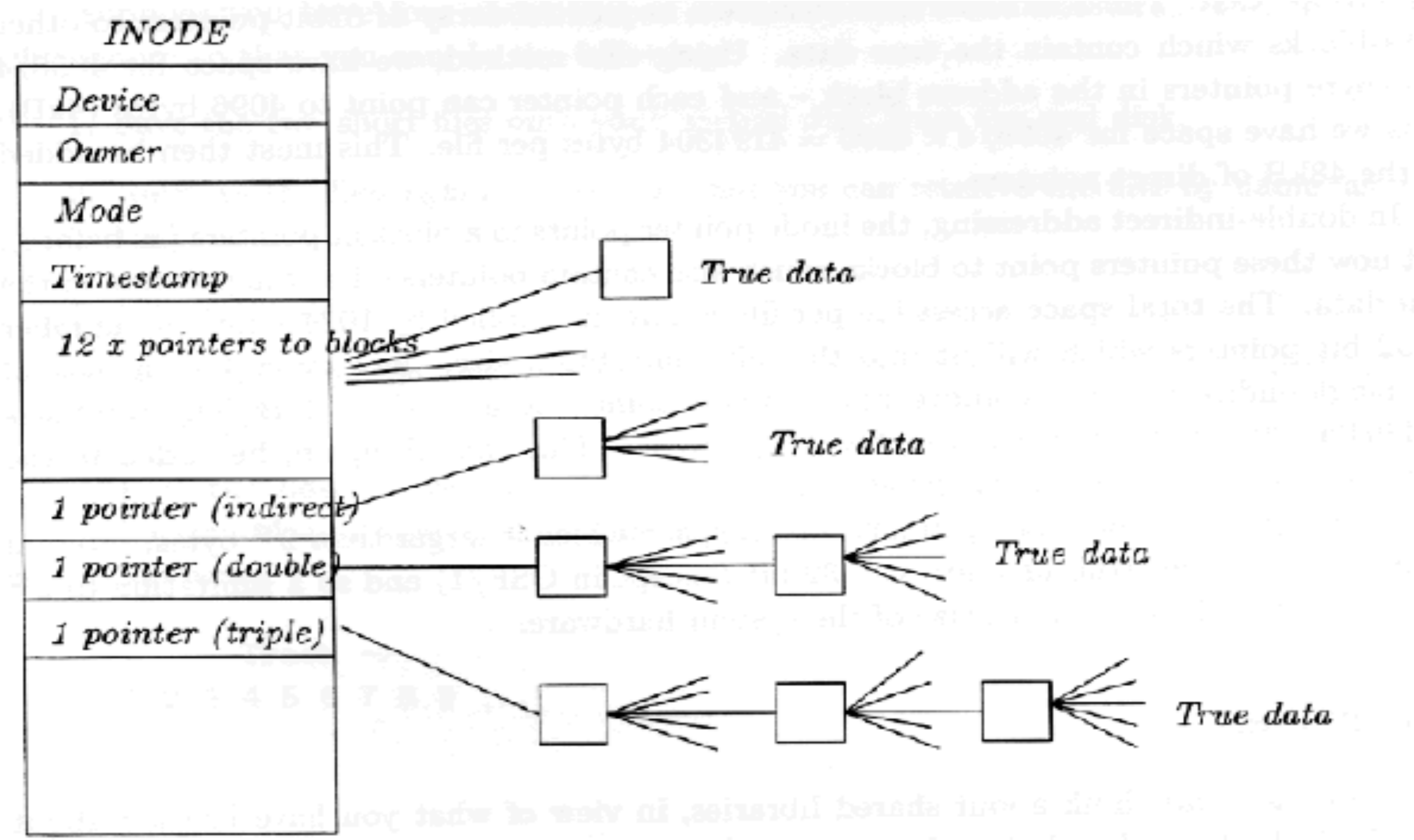
inodes

data blocks

Superblock

- *Superblock*
  - contains the info on the boundaries of the partition, info about where the inode table (and number of inodes) and where data blocks start (and their size). If the superblock is lost or damaged, the whole file system would be unreadable. It is important to make superblock backups when a file system is created, e.g ext2. The Linux file system check program **e2fsck** can do this.
  - Linux ext3 uses block groups. Each group keeps a copy of superblock.

Inode

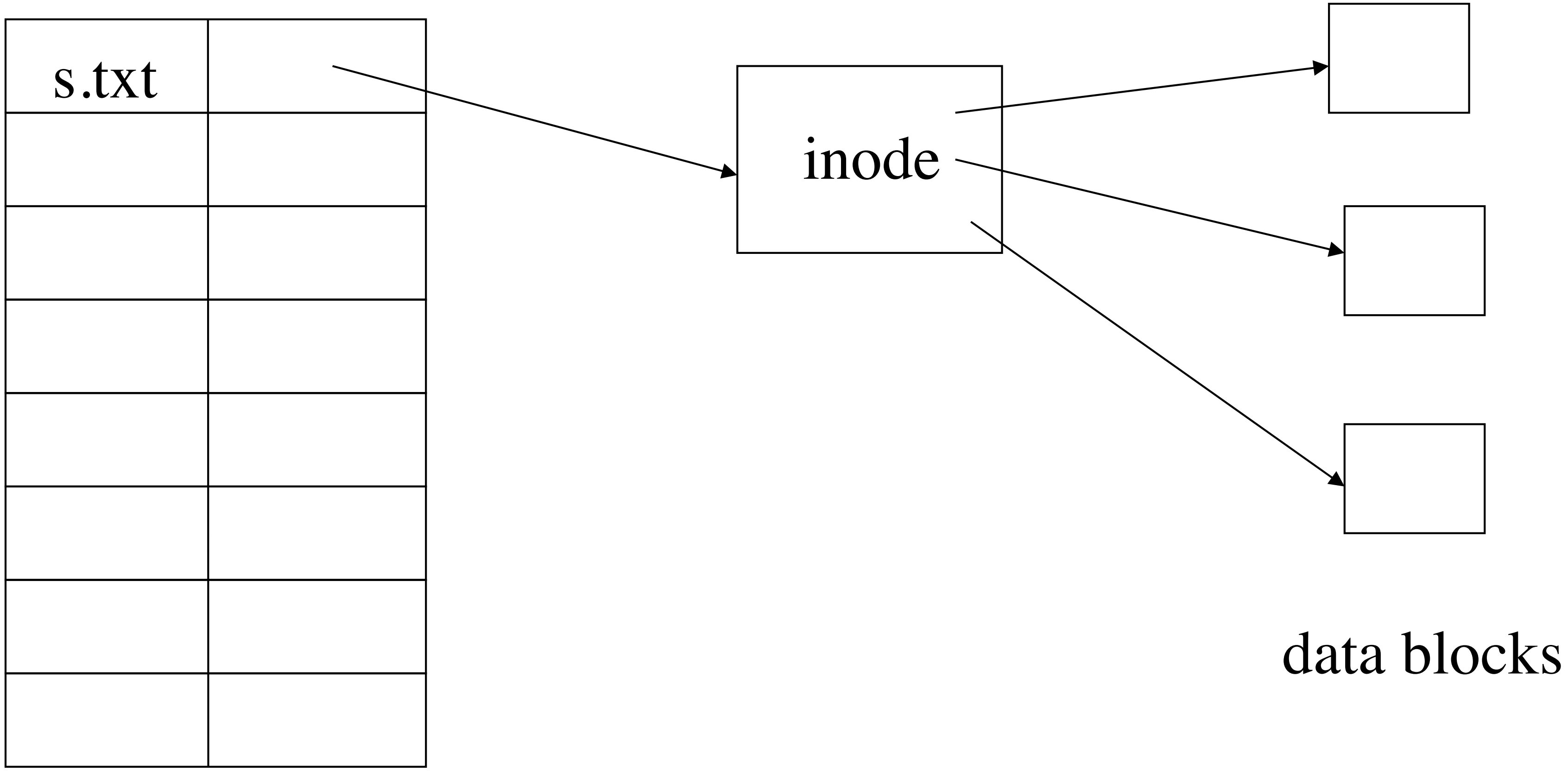

INODE
Device
Owner
Mode
Timestamp
12 x pointers to blocks
1 pointer (indirect)
1 pointer (double)
1 pointer (triple)
True data
True data
True data
True data

Inode (cont.)

- *Inode* (index node) is the data structure which holds the specific data about a particular file. Regardless of how large a file is, there is exactly one inode per file.
- Three ways of addressing data are used in inodes
  - Direct addressing: for files < 48KB
  - Indirect addressing: for files <4 MB
  - Double-indirect: for files < 4GB
  - Triple-indirect: for files < 4TB
- When a file system is created it creates a fixed number (which can be decided by SA) of inodes.
- How to decide the number of inodes?

Structure of a directory

A directory is a file!



Journaling

- Why journaling?
  - A system call like write() consists of many operations on the FS
    - Example: append a block of data to a file
      - Locate a free block, update data block bitmap, make inode point to the block, write the content into the block, update fields of inode
    - A system failure in the middle of the operation will cause inconsistency of the FS
    - FS consistency check is very time consuming
    - The result will be even worse if an operation involves two files.

Journaling (cont.)

- Guarantee the atomicity of syscalls
- A log disk is used to record modified data blocks
  - Data blocks are not written to FS until the syscalls are committed to the log.
  - When all involved data blocks of a transaction are written into FS, the transaction and its data blocks are removed from the log
  - When a system failure occurs, all committed data blocks in the log will be written to FS.
- Metadata journaling
  - Only log the metadata such as inodes, bitmap, superblocks, index blocks

Performance

- Journaling is slow!
- Ext3 uses a few methods to improve performance
  - Batch many system calls per transaction
    - Only one “open transaction” at a time
  - Delay copying cache block to log until a transaction commits to log
    - Hoping many system calls modify the same block, which only needs to write the block once.

System call for ext3 journaling

- System call implementation
  - h = start();
  - get(h, block #)
    - Warn journaling system the cache block will be modified
    - The block is added to list of blocks to be logged
    - Prevent writing block to disk until the transaction commits
  - Modify the cache block
  - stop(h)
  - guarantee: all or nothing

Ext3 transaction

- Only one transaction is open at a time
- While a transaction is open,
  - Add new system call handles
  - Remember their block numbers
- Commit time
  - Commit the current transaction every few seconds
  - Or at fsync()

Commit to log

- When an open transaction is to be committed to log
  - Mark the transaction as "done" so that new system calls must start a new transaction
  - Wait for in-progress systems calls to call stop()
  - For all blocks listed to be logged
    - Append descriptor (block #) to the log (on disk)
    - Append block content from cache to log
  - Wait for all log writes to finish
  - Append the commit record. Now blocks in committed log can be written to FS (disk)

Other issues

- How to free log space?
  - After logged blocks are committed to FS, the related log space is re-used
- A new transaction may start while a previous transaction is committing
  - How about if a syscall in new transaction wants to change a block in old committing transaction?
    - Can't allow change of the cache block
    - Use a new copy of the block for the new transaction
    - The new copy is used to write to FS
    - The old transaction log should be kept until the new transaction committed to log

How to recover?

- System failure may interrupt writing to the log
  - Only recover fully committed transaction in the log
- Recovery steps
  - Find the start and end of the log
  - Replay all blocks from fully committed transactions in log order
- How to guarantee consistency of the log?

Journaling modes

- Ext3 has three journaling modes
  - Journal
    - All data and metadata blocks are logged
    - Slow since every block is written twice to disk
  - Ordered
    - Can avoid inconsistency of FS
    - Only the metadata blocks are logged
    - However, the data blocks are written to FS disk before the metadata are logged
      - e.g. write() makes data blocks go to disk before committing adding block number to inode
      - Stale data won't be seen if there is a crash
  - Writeback: only log metadata (the fastest)

Challenges for ordered

- Case 1
  - rmdir, re-use block for file, ordered write of file, crash before rmdir committed
  - Will get scribbled directory block
  - Solution: defer free of block until the freeing operation forced to log on disk
- Case 2
  - rmdir, commit, re-use block in file, ordered file write, log force, crash, replay rmdir
  - File is left with directory content e.g. . and ..
  - Solution: revoke records, prevent log replay of a given block

Rethink the sync

- Ext3 maintains good internal consistency
  - E.g. no direct point to unallocated inode
- Applications and external users also need consistency
  - The delayed commit in ext3 may cause problem here
  - E.g. write(), printf("OK") may find out the file is not stored in FS.

Sync FS

- Sync FS
  - A sync FS forces updates to disk before returning from a system call, so it is slow.
  - If a system call returns, its effects will be visible after a crash
  - It is easy to reason about the correctness of FS
  - However, in real life, sync FS is not assumed
    - Write(); fsync(); printf(“OK”);

Async FS

- Async FS
  - A system call may return before data is written on disk
  - A write-back cache is used for modification
  - FS internal consistency is left to the system
  - The paper is talking about async logging of FS
- Durability problem with async FS
  - Send email to server and server sends back OK
  - cp * backupdir (is it ok to modify the files now)

Sync vs Async

- Sync FS
  - Slow
  - Durable
- Async FS
  - Non-durable
  - Fast
- Programmers have to choose between durability and performance
  - Or remember to call fsync() often

Key idea of the paper

- Try to achieve both durability and performance
  - Perform as async FS
  - Durability similar to sync FS
- Redefine durability
  - Updates only need to be durable by the time of external output, e.g., display output, network packet
  - If programmers see no output, no reason to expect FS I/O has finished