# Filesystems part 2

*Taco Walstra*

version 2018

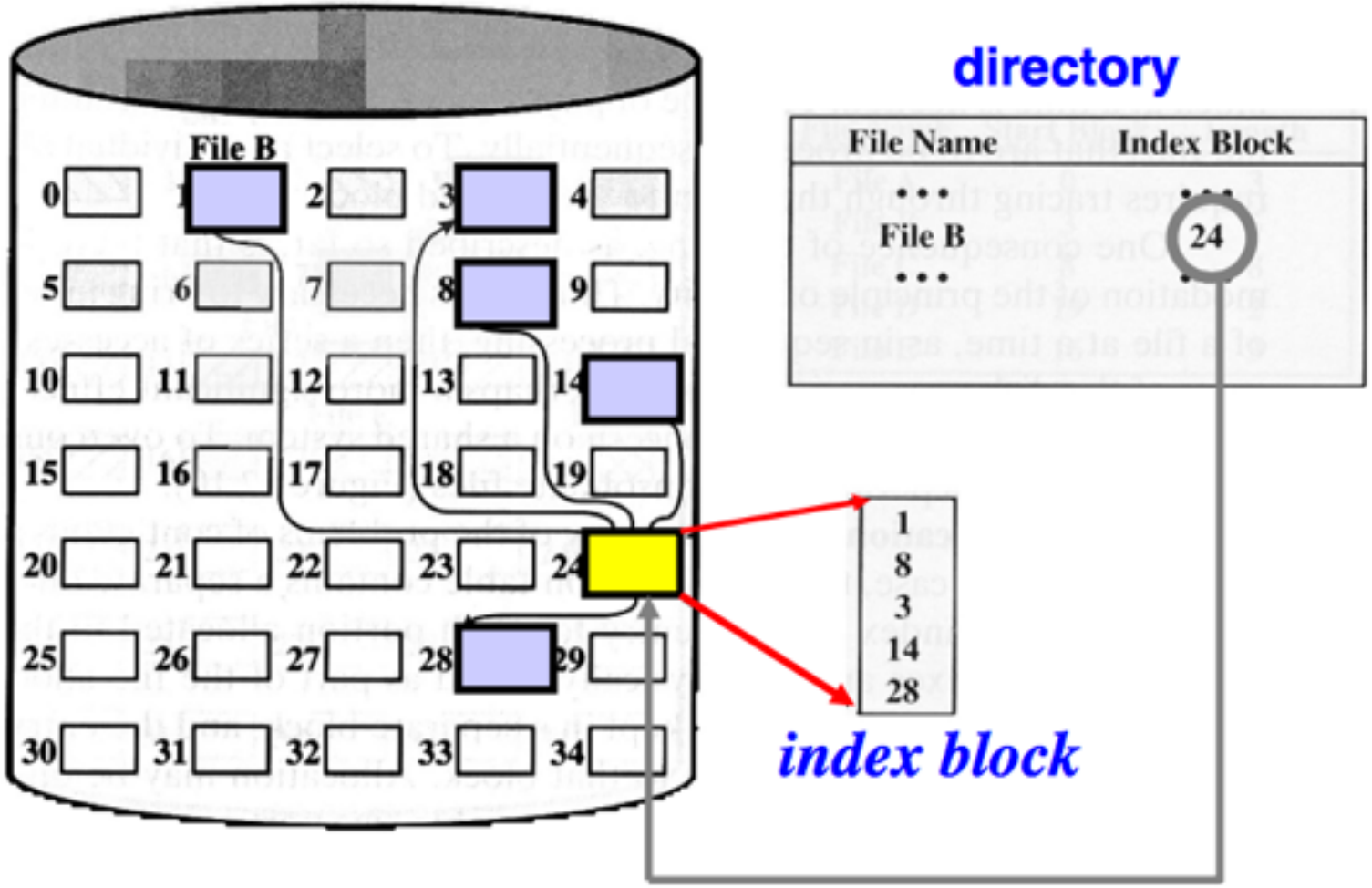
# Learning objectives

- Disk structure terminology: sector, block, track, seek time, settling time, (S)ATA, SCSI, SAS, RAID (only general ideas)
- Memory mapped I/O, instruction based I/O
- Disk scheduling algorithms: FCFS, SFFT, Elevator, C-scan
- Device drivers and devices (block-character)
- I/O scheduling: Deadline, CFQ, NOOP
- Filesystems: FAT,
  EXT, inode concept, journaling filesystems, log structured filesystems, network filesystems, distributed filesystems.

# File allocation

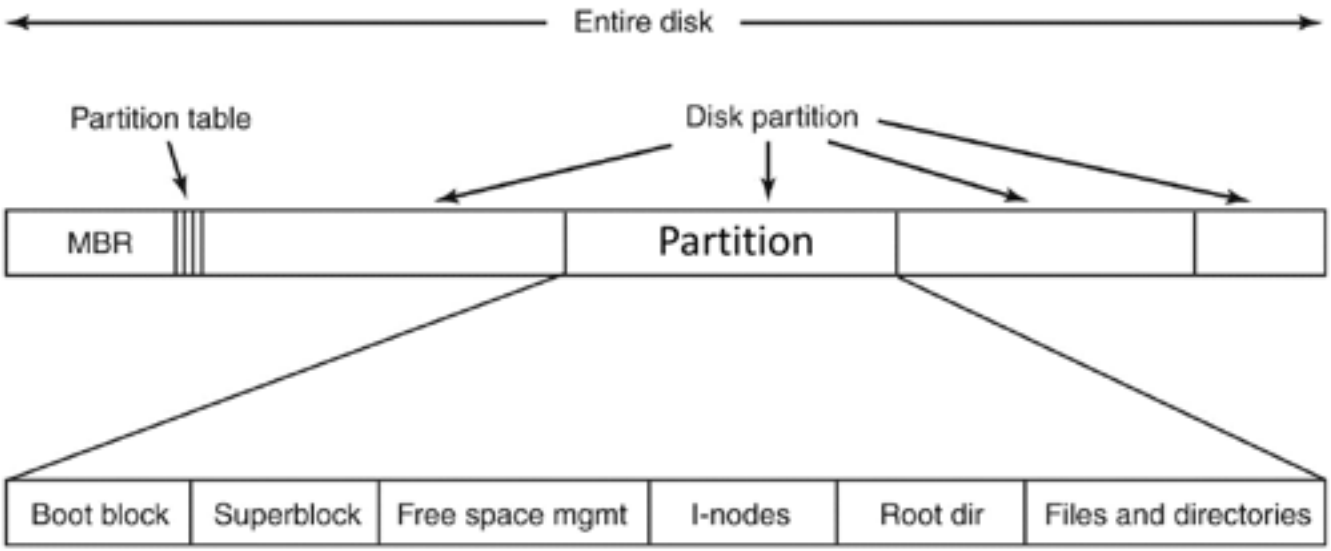
- 3 basic methods:
  - contiguous (see slides previous lecture)
  - linked: FAT table (see slides previous lecture)
  - indexed: UFS, ext2, ext3, ext4

# indexed file allocation



# UFS layout

- Sector 0 of a disk partition is called MBR (Master Boot Record)
  - contains partition table
  - boot loader (grub or windows)
- MBR finds the active partition and boot block
- Boot block loads the OS contained in that partition

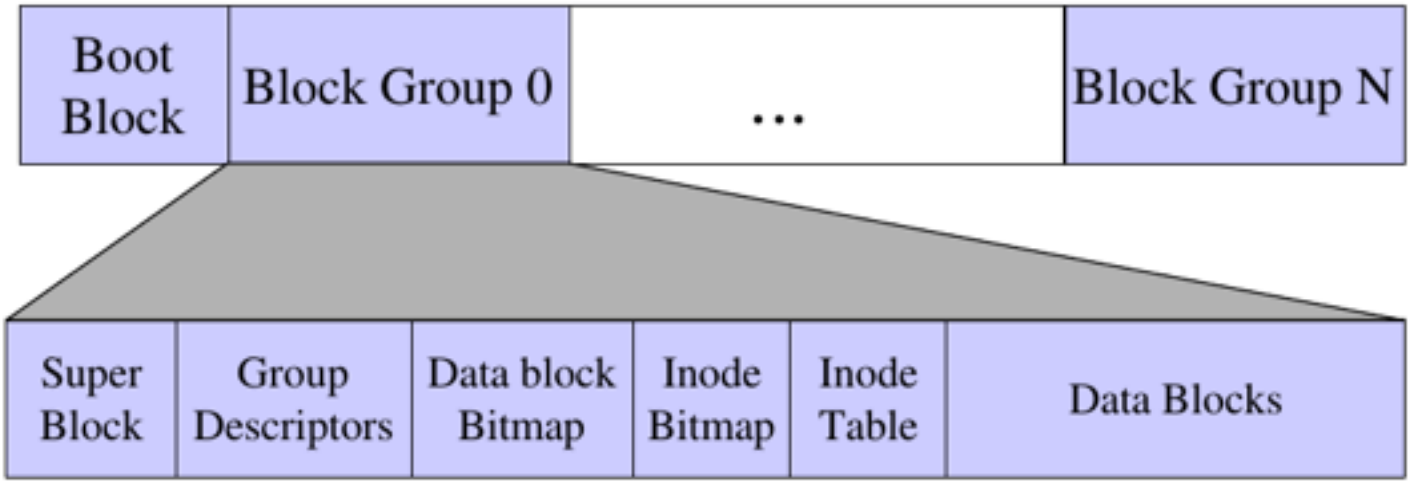


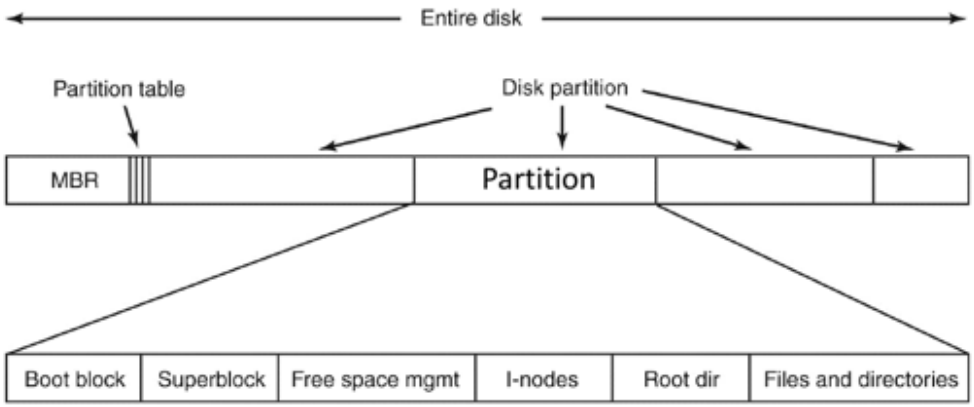
# ufs - ffs - ext2/ext3/ext4

- Origin of the Unix File System is historical (1980's)
  - Modern versions implemented in unix system V, SunOS, Solaris
  - small datablocks of 512 bytes, later 8192 bytes
  - original version no group implementations resulting in random clutter of blocks "all over the place" (file fragmentation)
  - Typical performance 2-4% of the disk raw bandwidth
  - legacy of ufs is the "inode" structure in other unix file systems
- BSD Fast File System (Berkeley Software Distribution, e.g. unix distro)
  - larger blocks of 4k or larger resulting in internal fragmentation increase
  - introduction of "cylinder groups"
  - introduction of "symbolic link" concept
- ext/ext3/ext4 continued along the FFS path
  - fixed size block groups instead of cylinder groups

# File system layout: from UFS to EXT

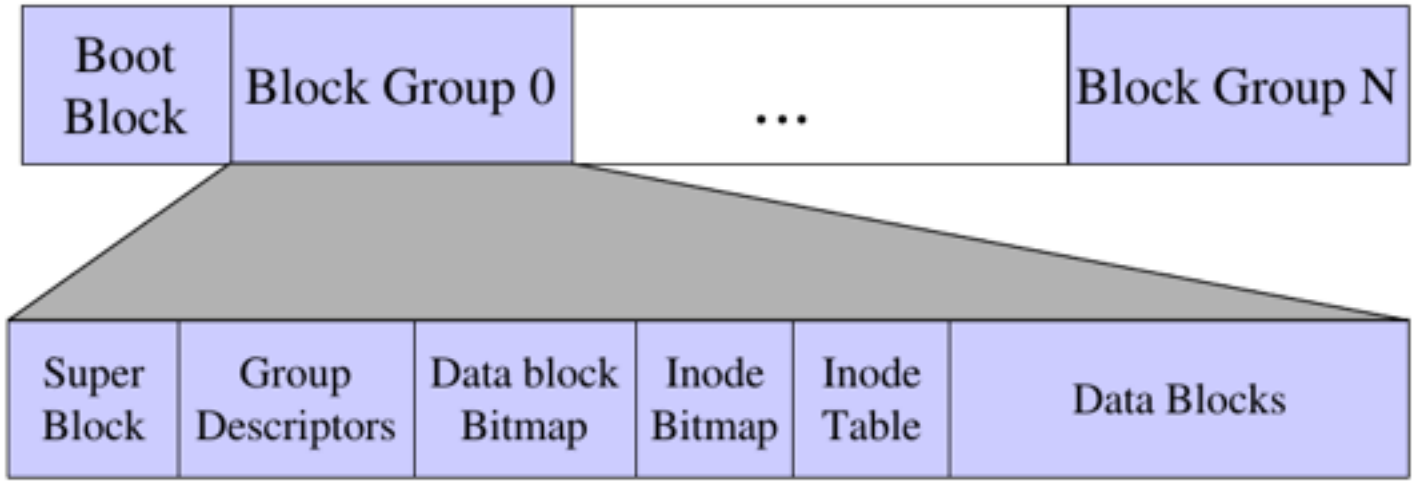
- Partition is split into block groups
- The superblock contains the layout of the file system



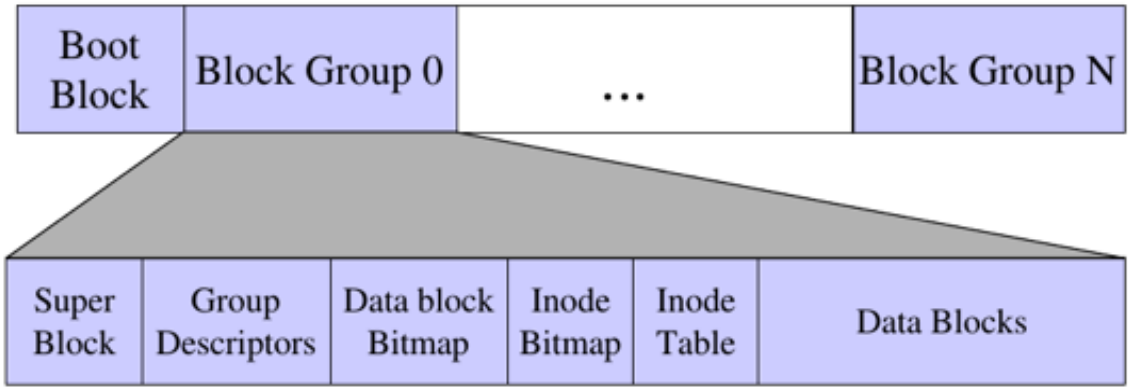
# EXT FS: Block groups

- Block groups are sequentially stored on the disk partition
- All groups have the same size
- Each block group has a copy of the superblock and block group descriptor table
- Advantage: reduce file fragmentation (keep the blocks of a file within one group instead of all over the place)



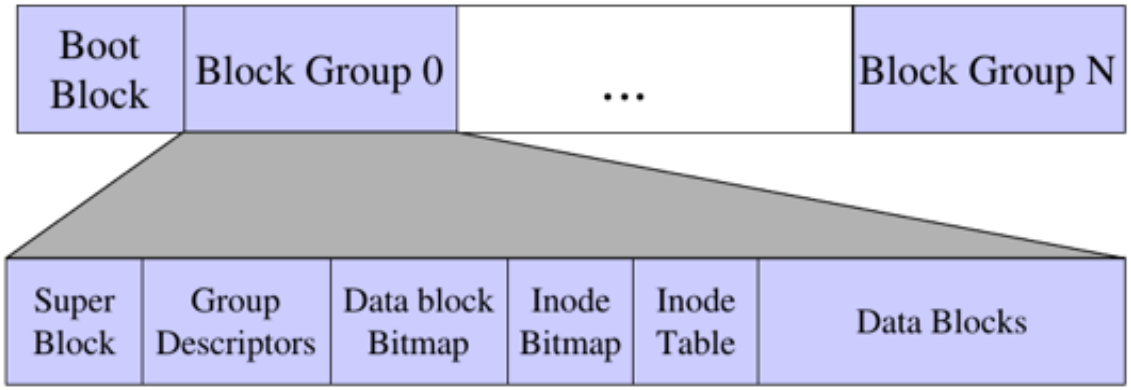
# Superblock

- Superblock contains:
  - size of the filesystem
  - number of free blocks
  - list of free blocks (+ pointer to free block lists)
  - index of the next free block in the free block list
  - size of the inode list
  - number of free inodes in the file system
  - index of the next free inode in the free inode list
- The superblock of block group 0 is always located in block 2 (after the boot block when having a block size of 4 kb, or block 3 when a block size of 1 kb is used)



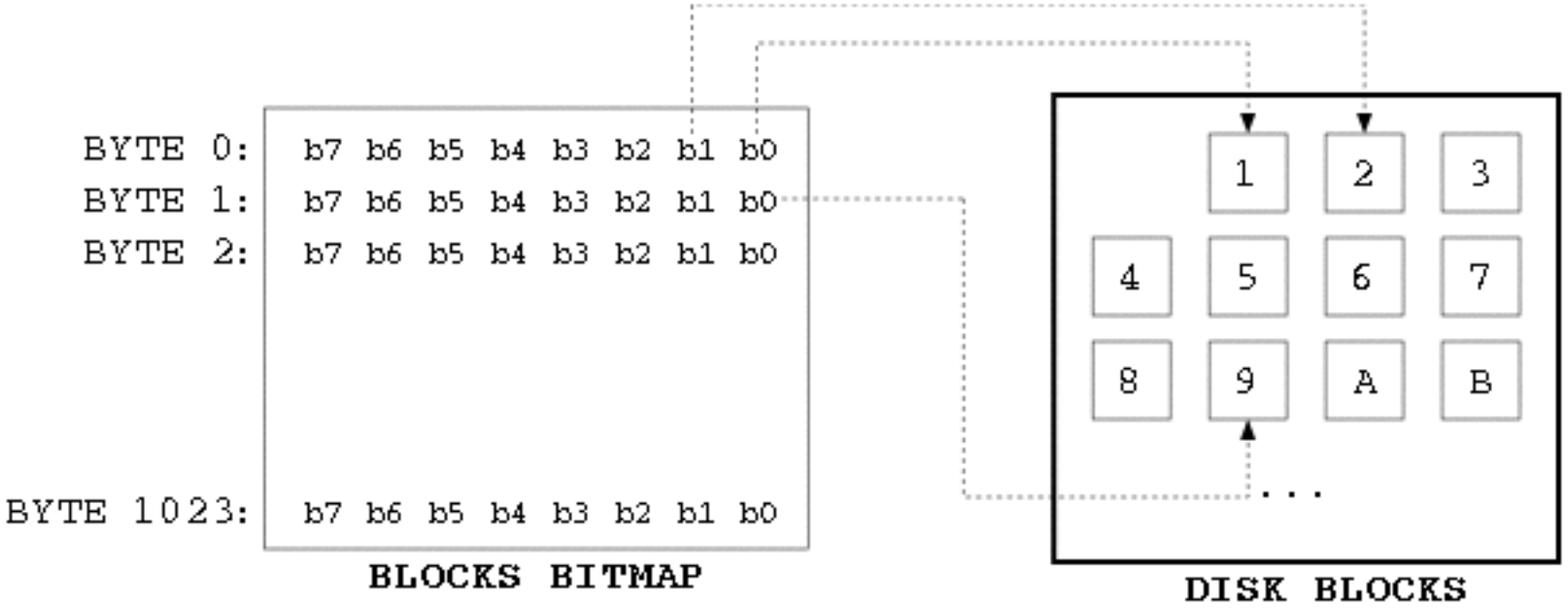
# EXT: block group descriptor table

- A block group descriptor table is an array of block group descriptors
- It's the first block after the superblock
- It's duplicated in every block group.
  - only the first copy in group 0 is actually used by the ext file system
- The block group descriptor contains:
  - location of the inode bitmap
  - location of the data block bitmap
  - a free blocks count
  - a free inodes count

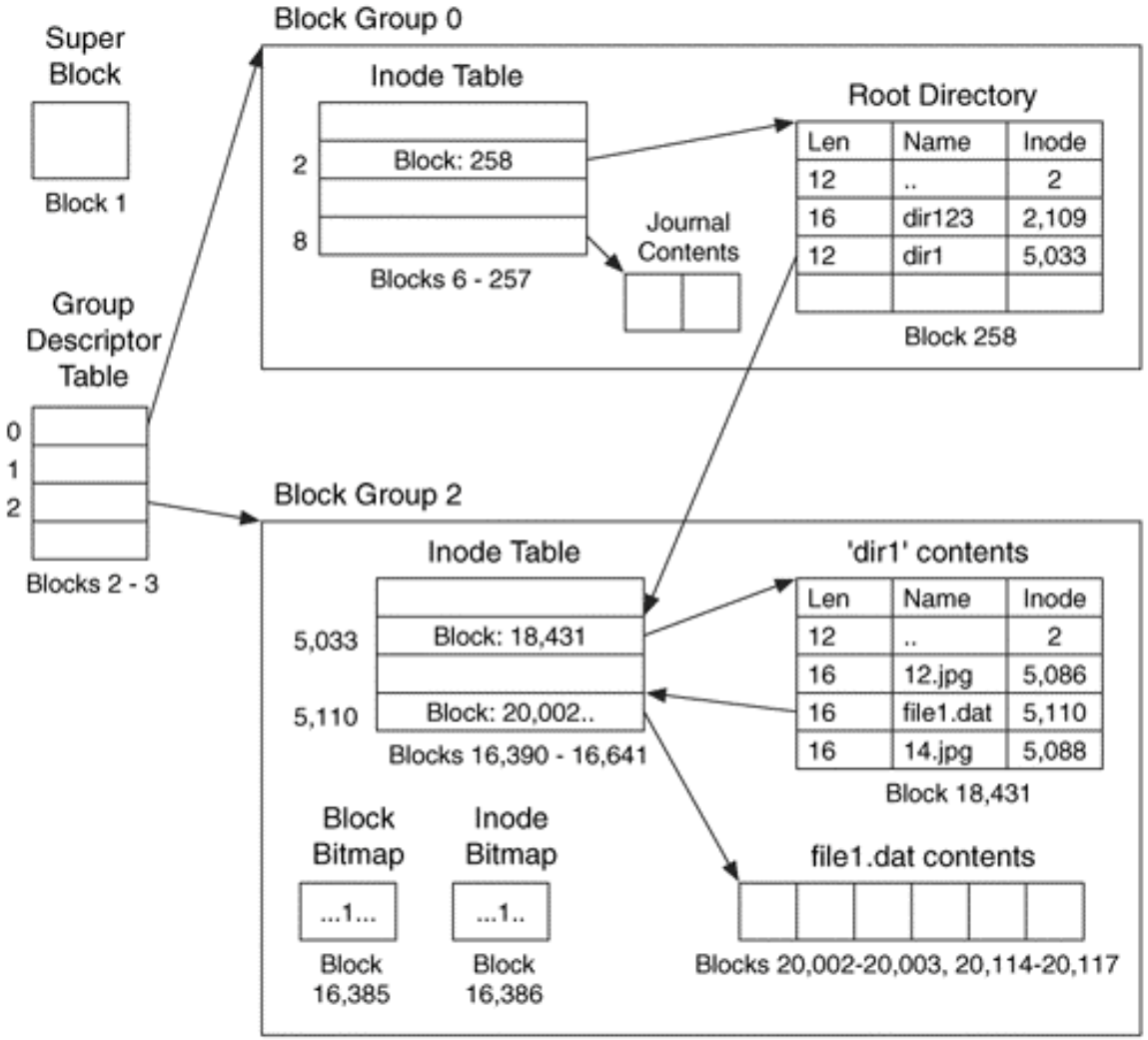


# ext2: block and inode bitmap

- Block bitmap
  - 1 bit for each disk block: 0 – unused, 1 – used
  - size = #blocks / 8
- Inode bitmap
  - 1 bit for each inode: 0 – unused, 1 – used
  - Size = #of inodes / 8

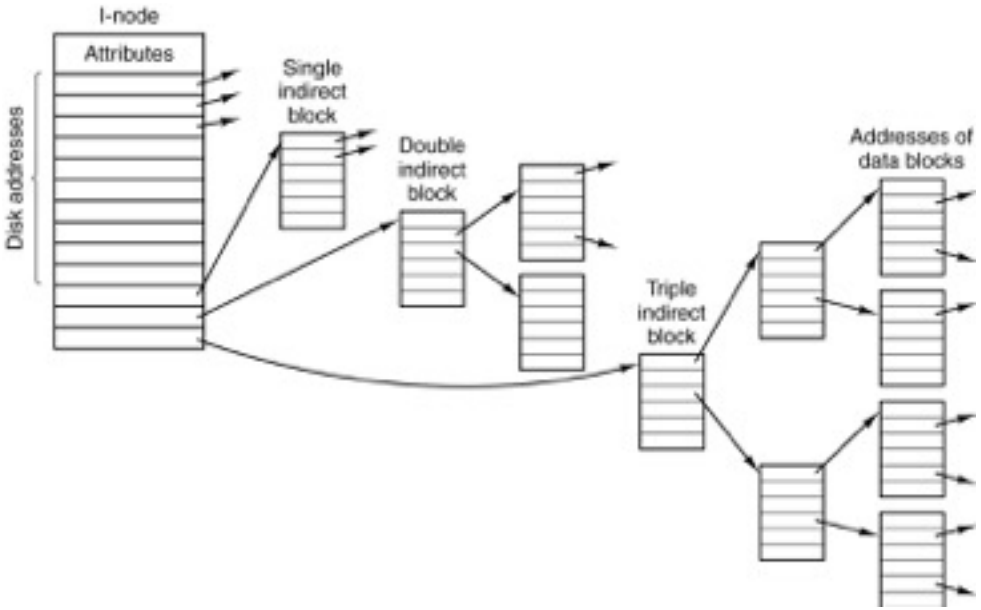


# Ext2 example: look up /dir1/file1.dat



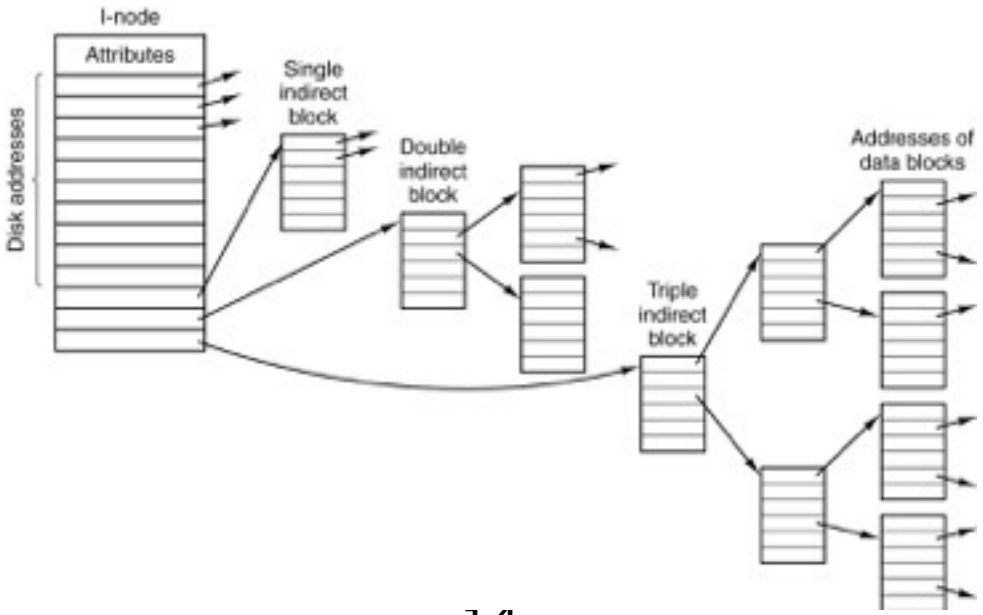
# inode structure

- The inode block may not be fully used, i.e. internal fragmentation
- The number of entries in the inode determines the file size
  - The inode can have indirect index tables chained in tree form
  - or combinations of direct index, single, double or triple indirect index tables.
  - The latter is used in unix file systems.



# Question

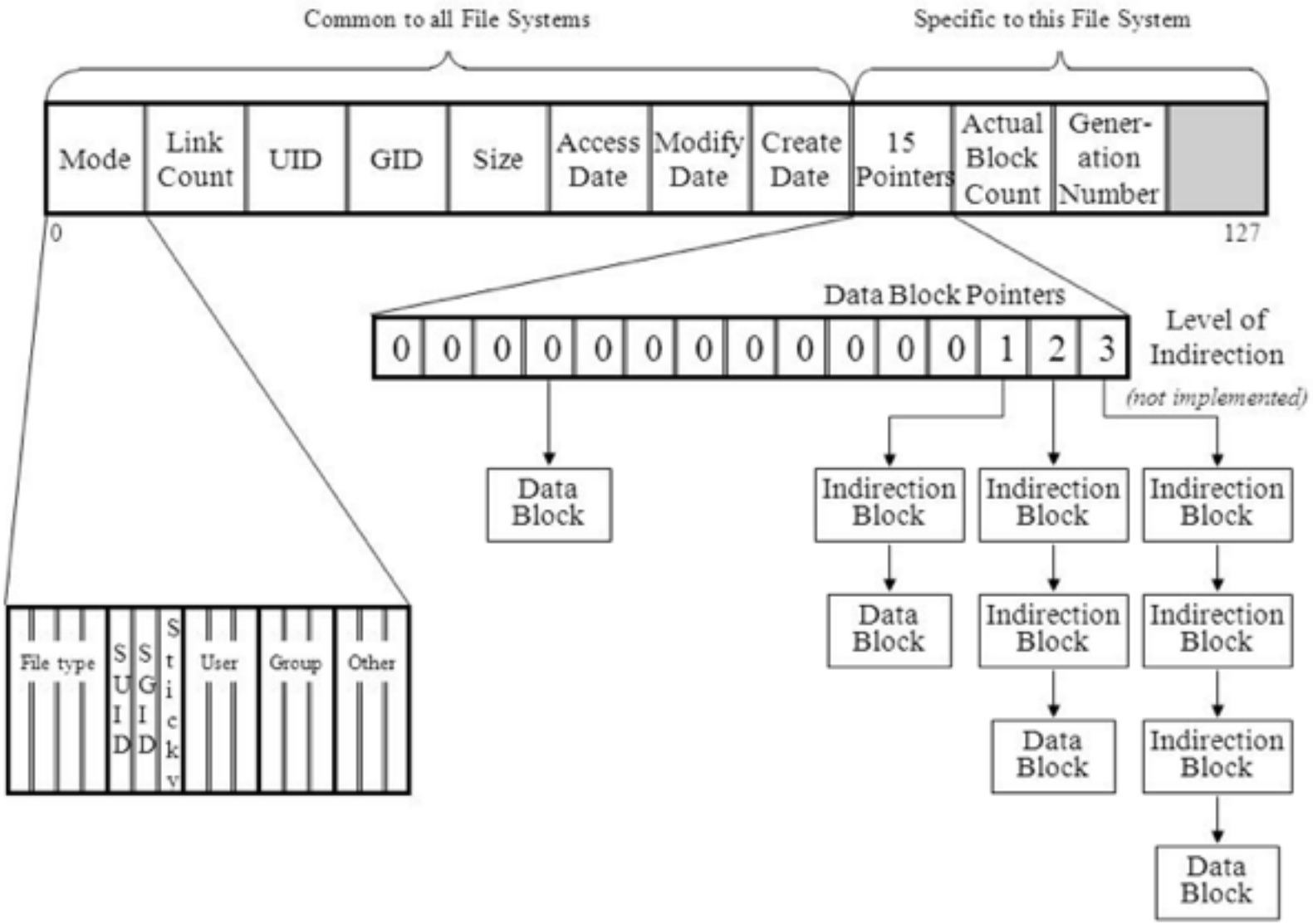
- What is maximum file size when
  - 32 bits filesystem, 4k block size
  - 13 direct pointers, 4 single indirect, 1 double indirect and 1 triple indirect pointer
- Each block pointer needs to address 32 bits, so will need 4 bytes for each pointer, so one indirect pointer can address: 4k/4 = 1k blocks
  Max file size = $4096\ (13 + 4\ .\ 1024 + 1024^2 + 1024^3) = 4\ 2^{10}\ .\ 2^{10x3} = 4\ .\ 2^{40}$ = 4TB



# Question

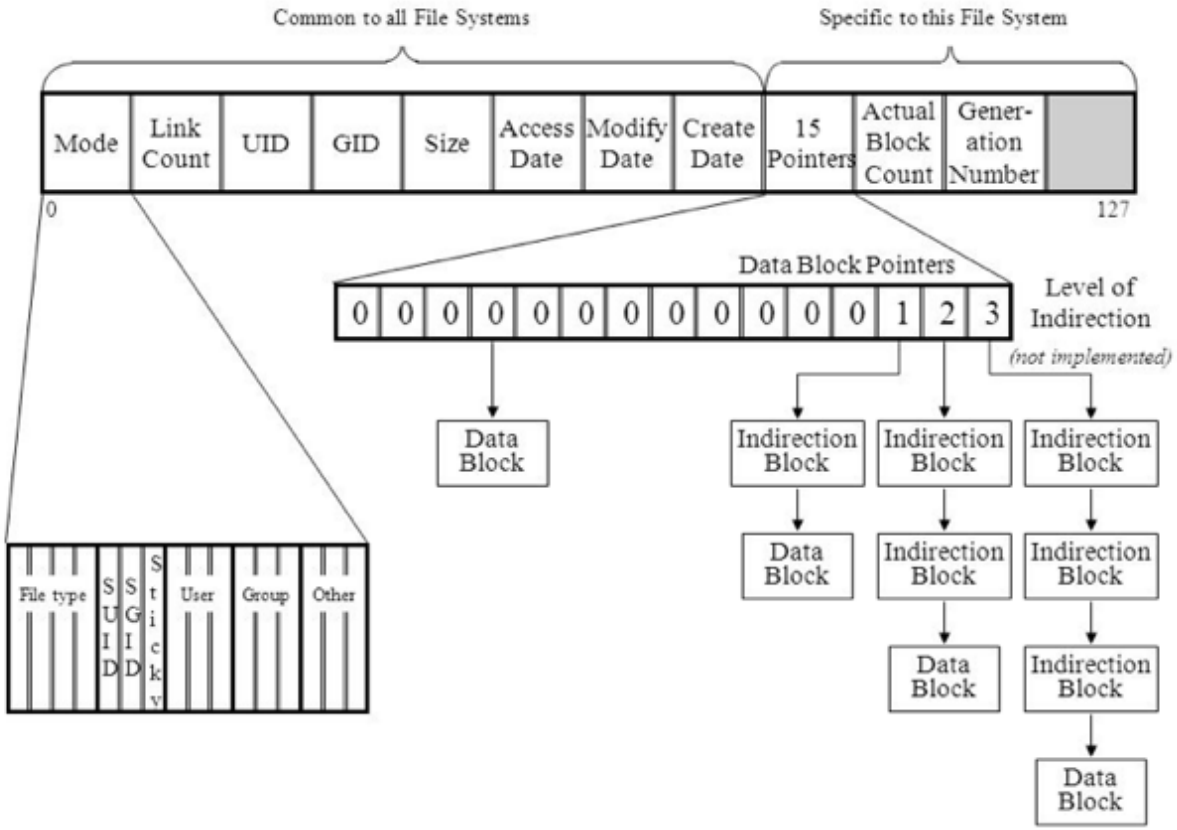
- What happens if I decrease the block size?
- Why not skip the direct pointers and use only indirect pointer?

# i-node structure



# i node modes

- Bitmask with 2 pieces of information:
  - what does the inode describe: regular file, directory, symbolic link, block device, character device…
  - access rights: user read, write,execute, group rwx, other rwx
  - total 16 bits

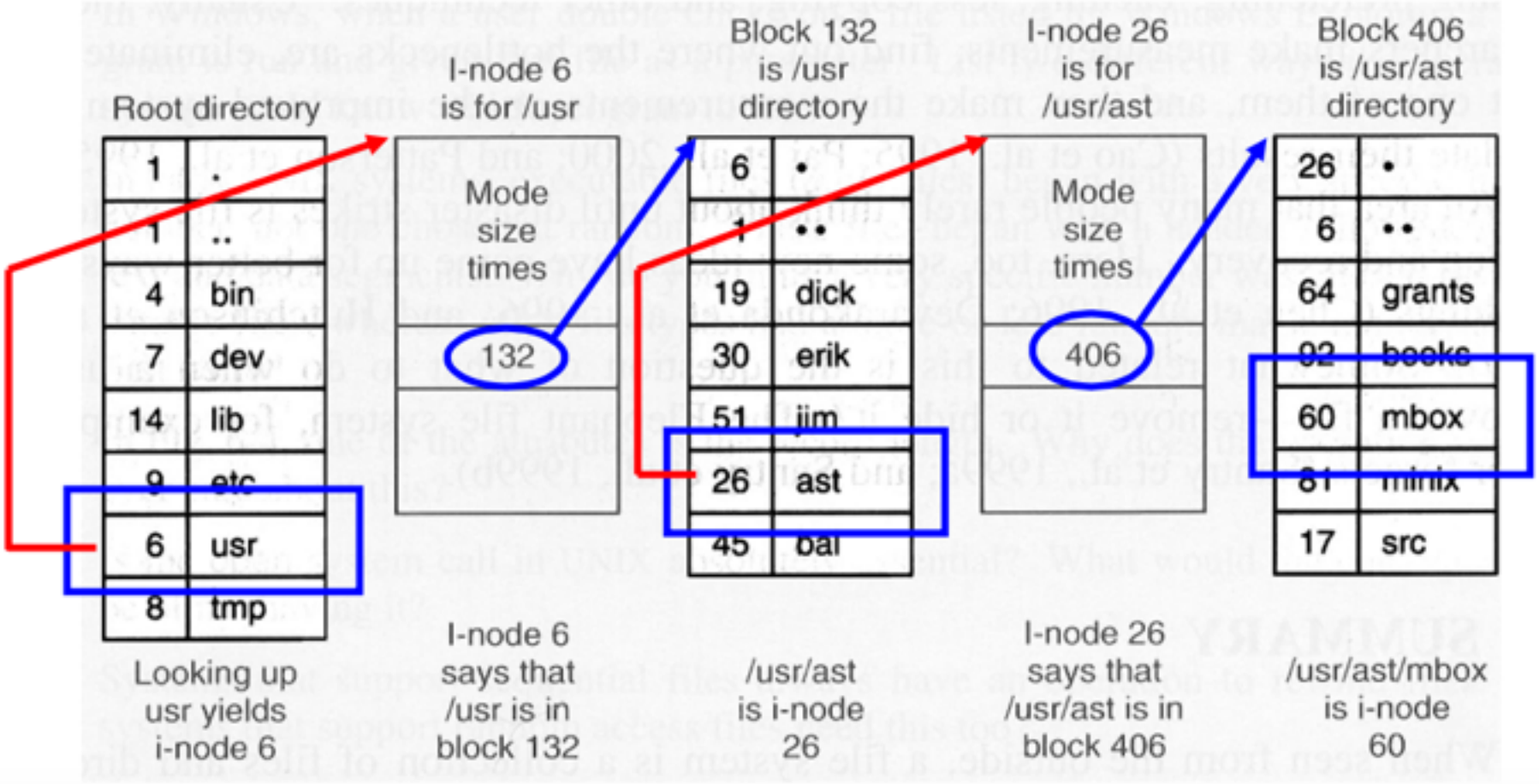


# Question

- We notice that a Linux program reading data from disk behaves very well when the datafiles are small (kilo/mega bytes), but when reading large files > 1GB the throughput is not very good. Why?
  - >1GB means that we will use 3 indirect blocks + datablocks instead of directly reading datablocks. For each read we need 4 disk seeks instead of one.

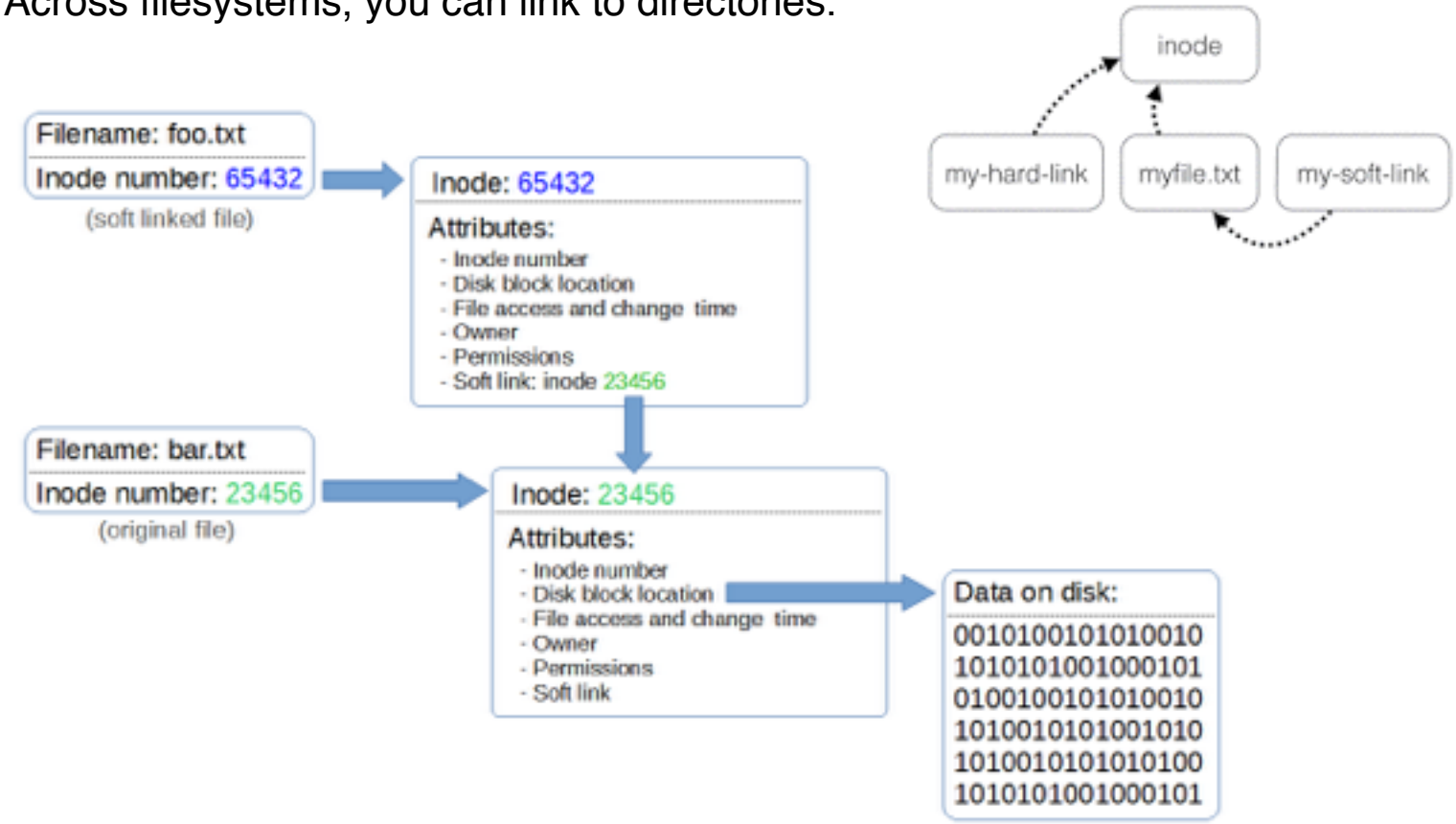
# Example inodes

- Find /usr/ast/mbox on disk



# Hard and Soft Links

- Hard link: do not create new inode. Create new entry in directory to the same inode. Increment “link count” in inode structure.
  - Not across filesystems, not to directories
- Soft link: create new inode. insert soft link to file inode. No change in link count
  - Across filesystems, you can link to directories.



# Question

- When to use hard links?
  - a change to filename will still be linked.
  - you want to have a file to exist in two or more places
  - busybox “bash” commands: 1 program “busybox”
    “ls” is hard linked to “busybox” etc.

# Example

- Tracing the inode path for /home/erna/foo.c (check yourself with cmd: ls -i!!)

- #2: . 2 l .. 2 l home 5 l user 9 l tmp 11 l etc 23 l …..
- #5: . 5 l .. 2 l erna 31 l janpiet 36 l …..
- #31: . 31l .. 5 l foo.c 12 l foo2.c 15 l …..
- #12: file data l file data l file data …..

- exercise:
  cd /tmp
  mkdir foo
  ls -iad foo
  537259076 foo
  cd foo
  ls -iad .
  537259076 foo

# Exercise 1

- I type the following commands on my linux system:
  cd
  ls -ld . ..
- I get the following response:
  drwxr_xr_x 48 walstra users 34789 May 1 13:48 .
  drwxr_xr_x 6 root root 4096 Jan 28 2015 ..
- What do you know by now?

# Exercise 2

- On my mac I do:
  cd /
  ls -i
- And I get as response:

```
    140 Applications                    5 home
     41 Library             179922 installer.failurerequests
324742 Network                   3 net
     37 System                842289 opt
170041 Users                     209 private
 14381 Volumes                  1392 sbin
 11614 bin                 49801 tmp
324948 cores                     40 usr
   300 dev                 50194 var
 19908 etc
```

- This is a list of the inodes. However on my Linux laptop I see the following. Explain why I see 3 times inode 2.

```
      2 drwxr-xr-x  24 root root   4096 Feb 26 13:31 .
      2 drwxr-xr-x  24 root root   4096 Feb 26 13:31 ..
2637825 drwxr-xr-x   2 root root   4096 Jan 14 19:02 bin
 196609 drwxr-xr-x   3 root root   4096 Feb 24 10:41 boot
      3 drwxr-xr-x  16 root root   4460 Mar  5 09:35 dev
 983041 drwxr-xr-x 206 root root  12288 Mar  5 07:45 etc
      2 drwxr-xr-x  14 root root   4096 Dec 29 09:24 home
```

# Do it yourself

- A 32 bits system, 1kb blocksize, 4 byte pointers,
  we have 12 direct pointers, 1 indirect, 1 double indirect, 1 triple indirect
  How to retrieve the block containing byte 1000,000?
- solution:
  byte nr 1000,000 is in block 1000,000/1000 = 1000 (to be more accurate we should divide by 1024)
  There are 12 direct pointers so we are not able to reach this block using these. (they are suitable for 12 * 1k blocksize = 12k files max.)
- Let's check the indirect block: this contains 1000/4 = 250 pointers (or, again, more accurate 1024/4). Single indirect is able to retrieve any file size up to 1024 *250 = 250k files
  Still not enough, so we need double indirect.
- Each pointer in the double indirect block points to a single indirect block, so 250 pointers *250 *1024 = approx. 62M, should be enough.
- retrieval of block 1000 using double indirect: we will need pointer nr 12+1=13 for the double indirect block.
  1000-(12+1+250)div 250 = 2 this is where we find the single indirect pointer. The indirect block is 1000-(12+1+250)mod 250=238 and this will finally point to blocknr 1000

# Journaling File Systems

- what happens if you experience a crash while updating your filesystem?
- Example
  - Create many files in a directory
  - System crashed while updating the directory entry
  - All new files are now “lost”
- Recovery (fsck command used for ext2)
  - may take very long time when using xxx TB disks (~1.5 hours)
  - Restores disk to consistency, but doesn't prevent loss of information; system could end up unusable.

# Journaling File Systems

- Basic idea:
  - update metadata (and possibly all data) as transaction
  - log the action to be performed ahead
    - Log entry: "I'm about to add block 23421 to file descriptor 572 at block index 23”
    - Then the actual block updates can be carried out later.
  - Once on disk, do what you have planned
  - if actions completed successfully, remove the log
  - if not successful rerun from the log next time (after crash)
  - logged operations must be idempotent
    idempotent: doesn’t matter if you execute them more than once

# Journaling File Systems

- Recovery in Journaling Filesystems
  - If a crash occurs, replay the log to make sure all updates are completed on disk.
  - log grows over time, so recovery could be slow.
  - Solution #1: checkpoint
    - Occasionally stop and flush all dirty blocks to disk.
    - Once this is done, the log can be cleared.
  - Solution #2: keep track of which parts of the log correspond to which unwritten blocks; as blocks get written to disk, can gradually delete old portions the log that are no longer needed.
    - Typicallly the log is used only for metadata (free list, file descriptors, indirect blocks), not for actual file data.

# Journaling File Systems

- Logging advantages:
  - Recovery much faster.
  - Eliminate inconsistencies such as blocks confused between files.
  - Log can be localized in one area of disk, so writes are fast (no seeks).
  - Metadata writes can be delayed a long time, for better performance.
- Logging disadvantages:
  - Synchronous disk write before every metadata operation.

# Journaling File Systems, final solution?

- Can still lose recently-written data after crash
  - Solution: apps can use fsync to force data to disk.
- Disks fail
  - One of the greatest causes of problems in large datacenters
  - Solution: replication or backup copies (how???)

# Journaling File Systems, final solution?

- Disk writes are not atomic:
  - If a block is being written at the time of the crash, it may be left in inconsistent state (neither old contents nor new).
  - At the level of sectors, inconsistencies are detectable; after crash, sector will be either
    - ★ 1. Old contents or 2. New contents or 3. Unreadable trash
  - But, blocks are typically multiple sectors. After crash:
    - ★ Sectors 0-5 of block may have new contents.
    - ★ Sectors 6-7 of block may have old contents.
  - Example: appending to log
    - ★ If adding new log entries to an existing log block, crash could cause old info in the block to be lost.
  - Solution:
    - ★ Replicate log writes (if crash corrupts one of the logs, the other will still be safe).
    - ★ Add checksums and/or versions to detect incomplete writes.

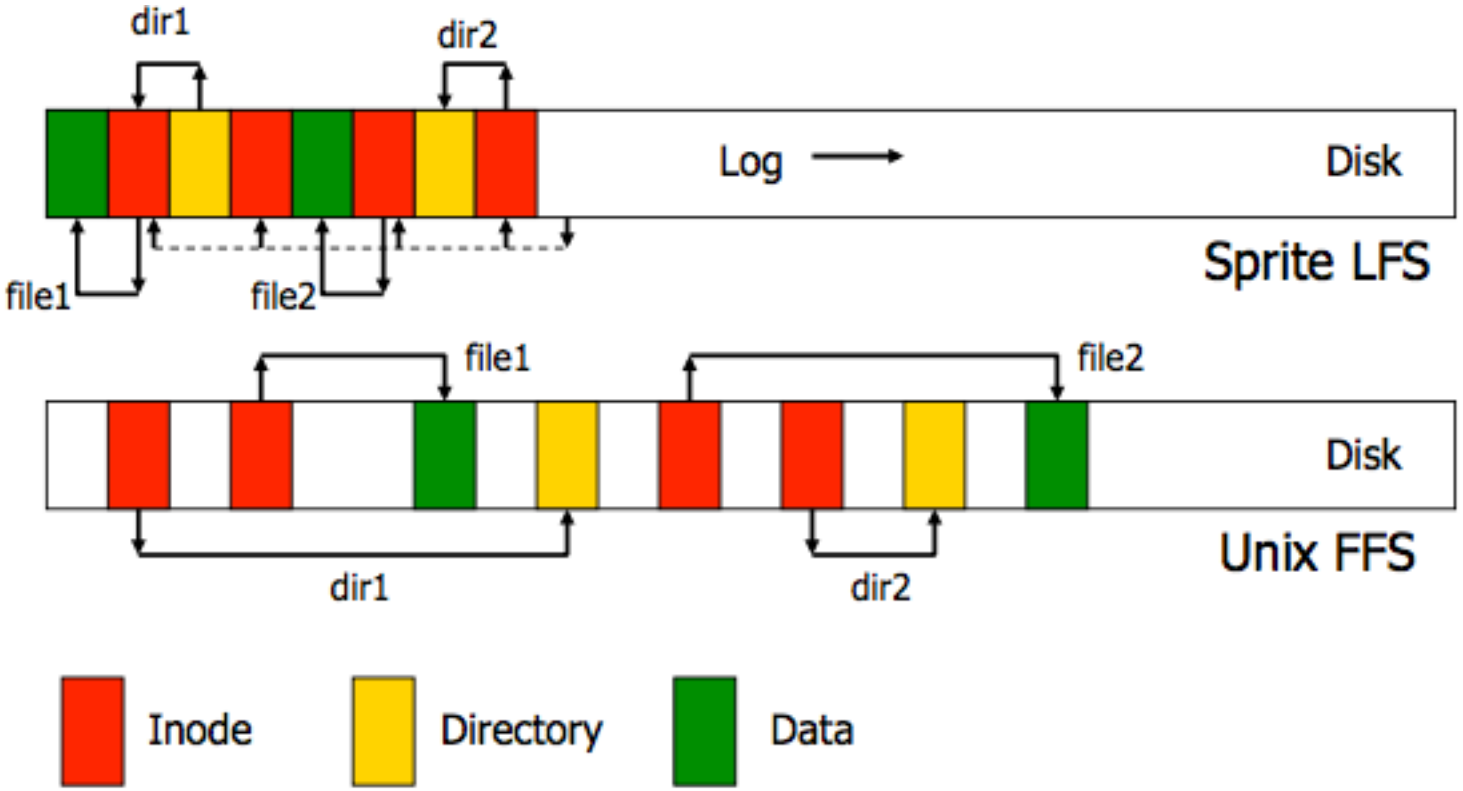
# Journaling File Systems

- Supported journaling modes (well, ext3 and later support all modes, ntfs supports “ordered”):
  - Write-back (metadata journaling):
    - journal metadata,
    - write back data.
    - write data before /after update of journal. If files modified before crash results in corruption (example:journal showing larger file than actual).
  - Ordered (metadata journaling): default for most systems including NTFS, ext3 etc.
    - Write metadata to journal
    - Data blocks are written to disk before metadata is marked as “committed in journal”
  - Data (full journaling)
    - Write metadata to journal
    - Write data blocks to journal
    - Use amount of disk traffic. Not used

# Log structured file systems

- CPU cache memories are growing, so disk traffic increasingly consists of writes. Focus on writes to disk
- There is a gap between random I/O and sequential I/O performance when accessing disks
- Many filesystems perform poor on many common workloads
- Filesystems are not RAID aware: a write to one small block of data causes 4 I/O operations (why 4?)
- Solution of LFS (90's): buffer all writes including metadata in memory. Write the data to disk when the buffer is full in one long sequential transfer Do not overwrite existing data, only use *free* locations.
- Treat the entire disk as a single log for appending. No seeks, but not simple…

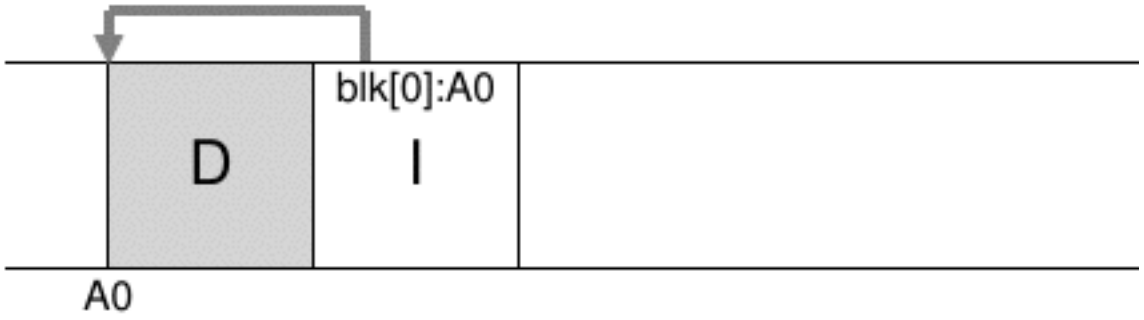
# Log structured file systems



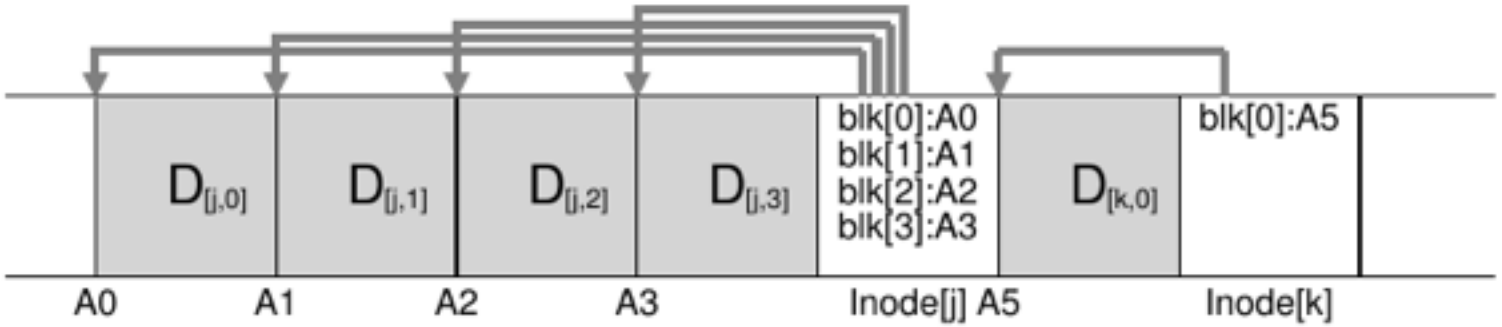

from univ. wisconsin-swift

# Log structured File System

- Write the data block + i-node sequentially to disk

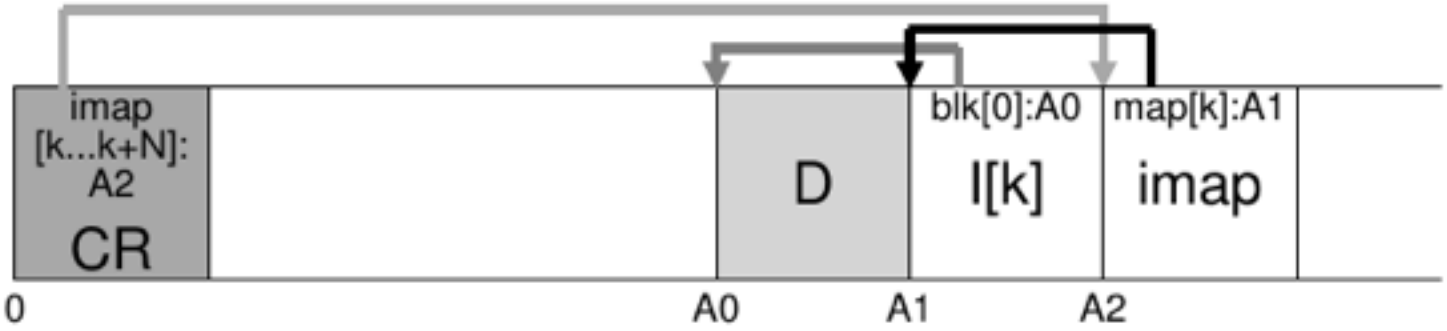



from Arpaci-Dusseau



# Log structured File System

- i-nodes are scattered all over the place on the disk. How to find them?
  - solution in LFS: use an i-map
  - i-map is a structure which maps the i-node number to a disk address




  - But, how to find the i-map? Solution: use a checkpoint region (CR)
    CR located on a fixed place on disk and contains pointers to the i-map
  - CR is updated periodically

# Log structured File System

- Problems of LFS:
  - garbage collection:
    existing file referred by i-node k point to D0.
    update of the file results in (left data_inode garbage):


  - 2 versions of the data are now located on disk. Suppose we append to block D0. This would result in a new i-node and D1 with the old D0 still valid. (we can keep both (versioning fs). LFS keeps only the latest (we need garbage collection)

# Log structured File System

- Garbage collector needs to fix (compact) the holes which tend to get between the datablocks with i-nodes after file modifications
- How should we select which blocks to re-write sequentially by a garbage collector?
- How often should the garbage collector execute?
- What when the system crashes during -for example- CR updates?
  - solution: 2 versions with timestamp. A different timestamp means a problem
- garbage collection became the main focus of controversies of LFS
- LFS lived/lives further in WAFL (NetApp), ZFS (Sun)
- used in YAFFS: yet another flash file system and JFFS (Journaling Flash File System)

# Summary

- What is a track, sector, block. Effect of blocksize.
- Pro and Cons of I/O scheduling models wrt device drivers, file access time
- How does a File Allocation Table work. Calculate the size of the FAT
- Understand how a unix file system works
- What is the maximum file size which an inode can address with direct, single, double, triple indirection. Finding blocks.
- Difference between hard and soft links in a unix file system
- journaling and log structured file systems
- how about distributed filesystems….

# Distributed filesystems

*Taco Walstra*

# topics

- Distributed filesystems
- Remote Procedure Calls
- Network File System (NFS)
- XFS: serverless Network File System

# Distributed File Systems

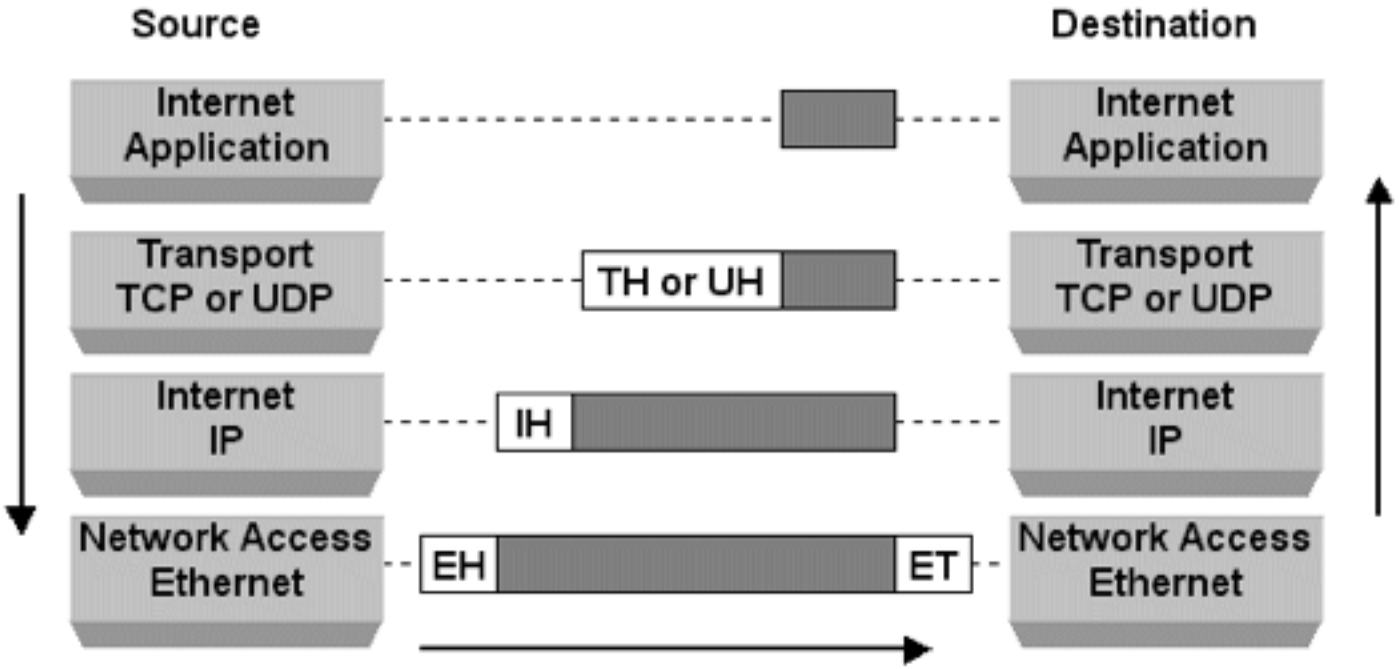
- Design challenges of distributed file systems:
  - Failure: give clients a “24/7” system
  - Performance
  - Security

# Network communication

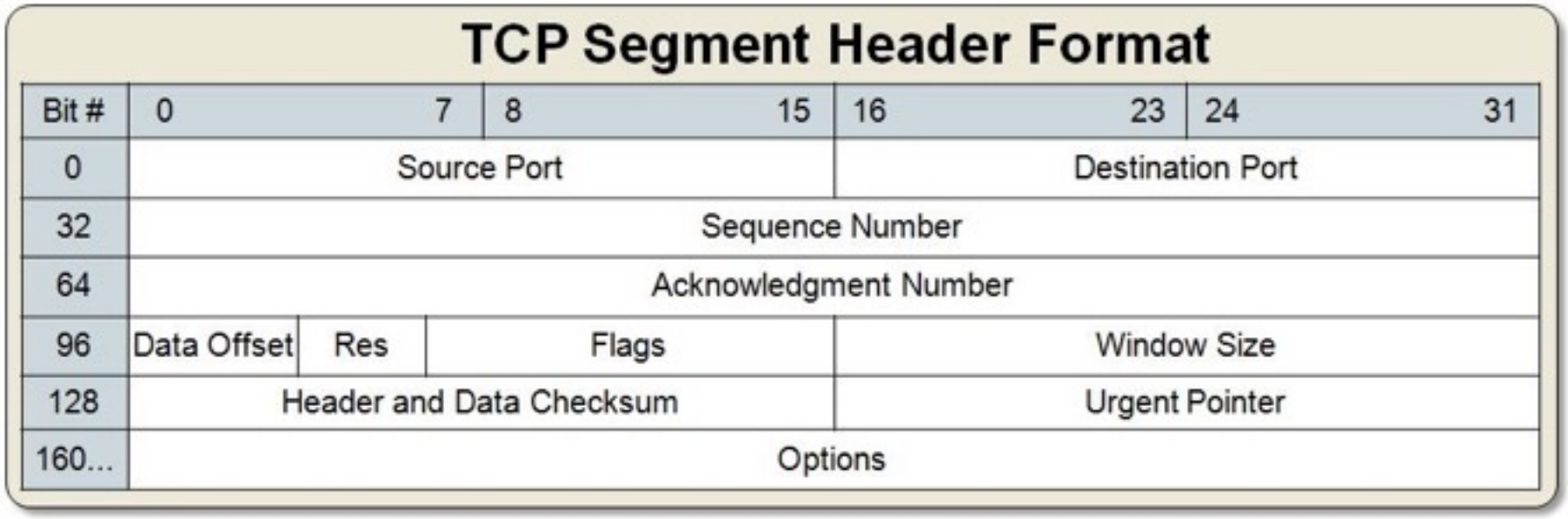
- Reliable and unreliable network communication
  - transmission control protocol (over internet protocol) (tcp/ip)
  - user datagram protocol (udp/ip)
- udp
  - use sockets to create a communication endpoint
  - udp is “connectionless”
  - “multidrop”: several clients can send information to a server port
  - a datagram has a minimum and maximum packet size
  - a datagram has a checksum to detect packet corruption
  - packets can get lost

# Network communication

- tcp
  - connection between client and server over fixed port
  - uses an acknowledge for each packet
  - sender uses a packet timeout with retry if no ack is received
  - packets have a sequence counter to trace dropped packets which can be retransmitted. Consequence: packets can arrive in arbitrary order

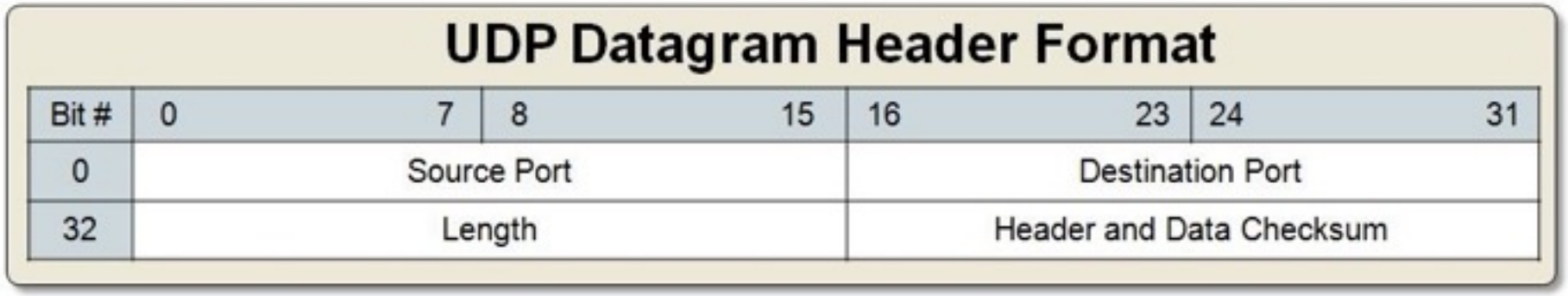


# tcp vs udp



**TCP Segment Header Format**

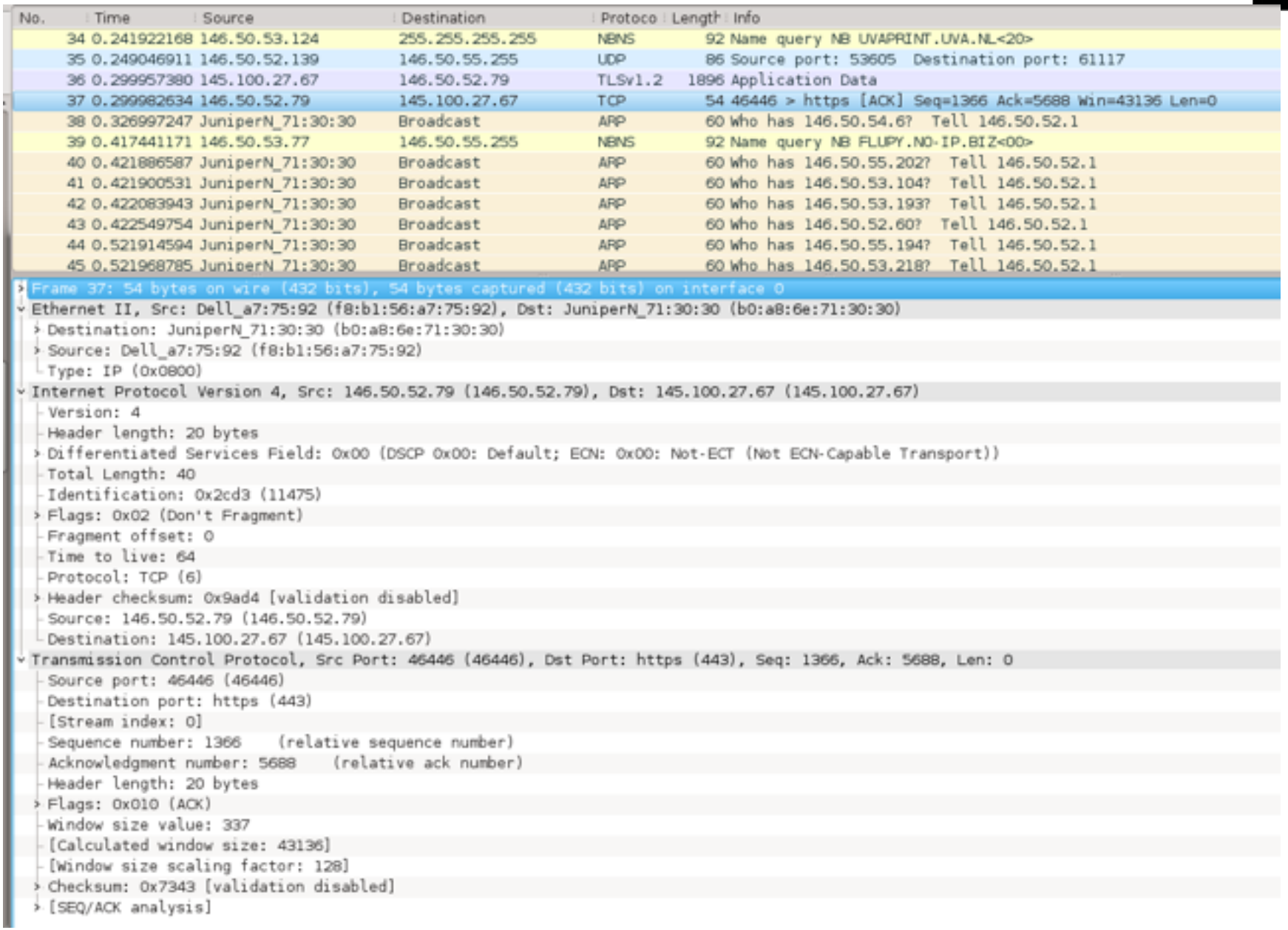
| Bit # | 0 – 7 | 8 – 15 | 16 – 23 | 24 – 31 |
|---|---|---|---|---|
| 0 | Source Port | | Destination Port | |
| 32 | Sequence Number | | | |
| 64 | Acknowledgment Number | | | |
| 96 | Data Offset | Res | Flags | Window Size |
| 128 | Header and Data Checksum | | Urgent Pointer | |
| 160... | Options | | | |



**UDP Datagram Header Format**

| Bit # | 0 – 7 | 8 – 15 | 16 – 23 | 24 – 31 |
|---|---|---|---|---|
| 0 | Source Port | | Destination Port | |
| 32 | Length | | Header and Data Checksum | |

# tcp communication



| No. | Time | Source | Destination | Protoco | Length | Info |
|---|---|---|---|---|---|---|
| 34 | 0.241922168 | 146.50.53.124 | 255.255.255.255 | NBNS | 92 | Name query NB UVAPRINT.UVA.NL<20> |
| 35 | 0.249046911 | 146.50.52.139 | 146.50.55.255 | UDP | 86 | Source port: 53605 Destination port: 61117 |
| 36 | 0.299957380 | 145.100.27.67 | 146.50.52.79 | TLSv1.2 | 1896 | Application Data |
| 37 | 0.299982634 | 146.50.52.79 | 145.100.27.67 | TCP | 54 | 46446 > https [ACK] Seq=1366 Ack=5688 Win=43136 Len=0 |
| 38 | 0.326997247 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.54.6? Tell 146.50.52.1 |
| 39 | 0.417441171 | 146.50.53.77 | 146.50.55.255 | NBNS | 92 | Name query NB FLUPY.NO-IP.BIZ<00> |
| 40 | 0.421886587 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.55.202? Tell 146.50.52.1 |
| 41 | 0.421900531 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.53.104? Tell 146.50.52.1 |
| 42 | 0.422083943 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.53.193? Tell 146.50.52.1 |
| 43 | 0.422549754 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.52.60? Tell 146.50.52.1 |
| 44 | 0.521914594 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.55.194? Tell 146.50.52.1 |
| 45 | 0.521968785 | JuniperN_71:30:30 | Broadcast | ARP | 60 | Who has 146.50.53.218? Tell 146.50.52.1 |

```
> Frame 37: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
v Ethernet II, Src: Dell_a7:75:92 (f8:b1:56:a7:75:92), Dst: JuniperN_71:30:30 (b0:a8:6e:71:30:30)
  > Destination: JuniperN_71:30:30 (b0:a8:6e:71:30:30)
  > Source: Dell_a7:75:92 (f8:b1:56:a7:75:92)
    Type: IP (0x0800)
v Internet Protocol Version 4, Src: 146.50.52.79 (146.50.52.79), Dst: 145.100.27.67 (145.100.27.67)
    Version: 4
    Header length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 40
    Identification: 0x2cd3 (11475)
  > Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
  > Header checksum: 0x9ad4 [validation disabled]
    Source: 146.50.52.79 (146.50.52.79)
    Destination: 145.100.27.67 (145.100.27.67)
v Transmission Control Protocol, Src Port: 46446 (46446), Dst Port: https (443), Seq: 1366, Ack: 5688, Len: 0
    Source port: 46446 (46446)
    Destination port: https (443)
    [Stream index: 0]
    Sequence number: 1366    (relative sequence number)
    Acknowledgment number: 5688    (relative ack number)
    Header length: 20 bytes
  > Flags: 0x010 (ACK)
    Window size value: 337
    [Calculated window size: 43136]
    [Window size scaling factor: 128]
  > Checksum: 0x7343 [validation disabled]
  > [SEQ/ACK analysis]
```

# Remote Procedure Calls

- RPC systems have 2 components
  - protocol compiler: definition of functions to call with their arguments
    - create msg buffer
    - pack information into buffer: function identifier, arguments
    - send msg to RPC server
    - wait for reply
    - unpack return code and arguments
    - return to caller
  - run-time library
    - locate the remote service (using dns, ip address, port number etc.)
    - use a transmission protocol: tcp/ip or udp/ip. Which is better?
    - handle byte ordering: little or big endian
    - handle reassembly of packets
    - handle requests and return synchronously or asynchronously

# Remote Procedure Calls

- RPC can be transmitted over tcp, udp or on http(s)
- Advantage: https security layer as bonus
- Disadvantage: more data/code between actual applications on server and host
- Can be done by xml-rpc (rpc wrapped in xml) over http(s)
  - disadvantage: xml space intensive,
    encode/decode introduces performance penalty,
    navigating DOM/tree complicated

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
        <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>
```

```xml
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

- Can be done by a binary protocol using Google protocol buffers.

# NFS

- History NFS
  - 1974 (!) - 1989 Start of NFSv1 + v2, RFC 1094.
  - 1995 NFS v3, RFC 1813, most popular, tcp support, security, 64 bit
  - 2003-2010 NFSv4, v4.1, parallel support
- Open standard with clear and simple interfaces
- Transparent system with fault tolerance, efficiency, but
- Limited concurrency, some security issues (especially older versions), limited replication
- Early NFS uses udp as transport layer
- Later versions (> v3) use tcp as transport. NO http(s)

# mounting nfs

- mount -t nfs //servername /opt/servername
- /etc/fstab automount

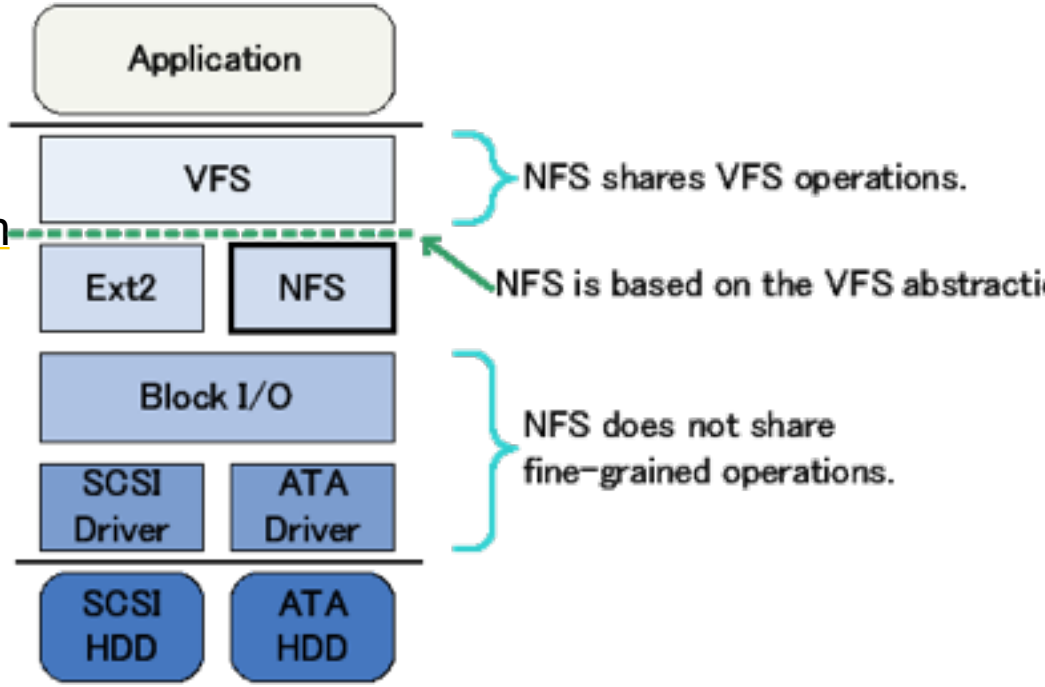
```
[twalstr1@tc etc]$ cat fstab

#
# /etc/fstab
# Created by anaconda on Thu Apr 14 14:18:32 2016
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/centos-root /                       xfs     defaults        0 0
UUID=dff7ff12-4770-403e-9664-5aa13121e251 /boot                   xfs     defaults        0 0
/dev/mapper/centos-swap swap                    swap    defaults        0 0

# sysconf
#146.50.10.33:/export/install/centos7   /opt/install            nfs     _netdev,timeo=600,tcp,intr
# Glados2
146.50.10.144:/fac/tc                   /mnt/glados2            nfs     _netdev,timeo=600,tcp,intr
glados2.science.uva.nl:/fac/tc /fac/tc nfs _netdev,tcp,intr,rw,timeo=600,soft 0 0
[twalstr1@tc etc]$
```

# NFS vs Dropbox/surfdrive

- NFS
  - All data is stored in a remote server
  - Client doesn't have any data on its local storage
  - Network failure => no access to data
- Dropbox/Surfdrive
  - Client store data in its own local storage
  - Differences between the server and the client are exchanges to synchronize
  - Network failure => still can work on local data. Changes are synchronized when the network is recovered
- Which approach do you like more and why?

# Network File System (NFS)

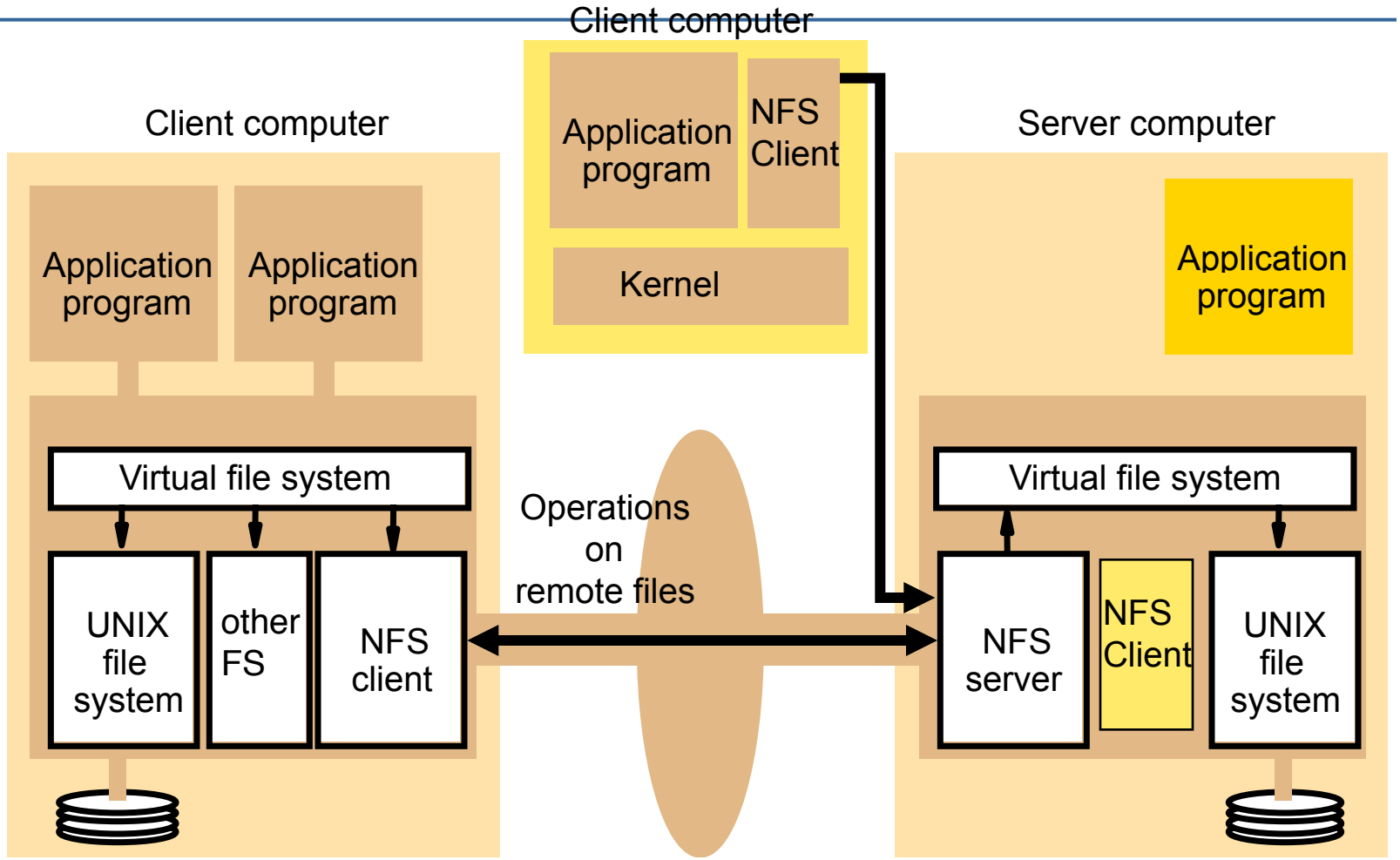
- Three layers
  - Unix file system interface: read, open, write, close, file descriptors
  - VFS layer: distinguish local from remote files. Calls the NFS protocol for remote requests using a virtual abstraction
  - NFS service layer: implements the NFS protocol
- NFS is basically a RPC protocol on top of TCP.
- NFS servers are stateless: each request provides all arguments required for execution
- see extra source material: http://researcher.watson.ibm.com/researcher/files/il-AVISHAY/03-nfs.pdf



# Stateless vs stateful

- Stateless server: loses state in the event of a crash – A stateless server looks to the client like a slow server
- – Simple server design
- – Quick recovery after reboot
- – Client needs to maintain state: if the server crashed, the client does not know if the operation has succeeded, so it must retry
- – File operations must be idempotent
- Stateful server: remembers its state and recovers – Complex server design yet simple client design
- – Longer server recovery time
- Permits non-idempotent operations (i.e. file append)

# NFS


Client computer
Application program
NFS Client
Kernel
Client computer
Application program
Application program
Virtual file system
UNIX file system
other FS
NFS client
Operations on remote files
Server computer
Application program
Virtual file system
NFS server
NFS Client
UNIX file system

# Issues NFSv2/v3/v4

- In NFSv2 the user id /group id is provided in the RPC call and the server checks the permissions as if using the filesystem locally
  Replicate uid/gid on the server? What to do with root?
  - v3 uses secure authorization
- If client1 buffers file data in cache, client2 will read old data from server.
  or: client1 write to file, client2 does the same. What is stored on the server?
  This is arbitrary!
- stale cache problem: Client1 removes file foo, client2 tries to read from foo, but it is non existent. Solution: when unlink is issued, rename it to .foo1234 and unlink it when all clients have closed their filehandle.
- Performance on write: each write should be flushed to disk before returning from call (stateless policy). Result: performance bottleneck (before v4)
- Large number of users is not possible
- v4: parallel data support (pNFS) allows high bandwidth transfers

# Example network bandwidth

- Suppose we want to create a 1MB file on the server and use a 100 Mbps ethernet. How long does this take?
- Assume 4 kb block size. We transmit 1 block at a time over tcp
- Each block takes:
  100us (o_send) + 4KB/100Mbit/s + 1us (latency from last byte off NIC to last byte arrives) + 100us (o_recv) + 10ms (disk) + 100us (o_send) + .5KB/100Mbit/s + 1us + 100us (o_recv)
  = 100us + 300us + 1us + 100us + 10000us + 100us + 35us + 1us + 100us = 10702us
  1MB/4KB = 256 blocks => 256 * 10702us = 2.75s

  Pff that's not fast….

# Example network bandwidth

- Let's take a 1 Gbps network..
  100us (send) + 4KB/1Gbit/s + 1us (latency) + 100us (recv) + 10ms (disk) + 100us (send) + .5KB/1Gbit/s (ack) + 1us (latency) + 100us (recv)
  = 100us + 30us + 1us + 100us + 10000us + 100us + 3.5us + 1us + 100us = 10436us for each block

  = 256 * 10436 = 2.74s !!
- Conclusion a higher bandwidth network doesn't necessarily give you improvement! Disk is the bottleneck still
- Solution: do not ack every block => delay becomes ~3 ms.

# False or True?

- NFS uses what?
  - allow data to be transported on TCP and UDP
  - can operate on a LAN or wide area networks
  - use caching to improve file system performance
  - are friendly for idempotent operations
  - are stateless

# xFS

- xFS = no File System. (NOT "XFS", which is a different filesystem. Case matters). Original design University of Berkeley
- 64 bit and max file size 8 exbibytes (2^63)
- Use network with parallel server clusters
- Software RAID and distributed control. Key ideas:

- 1. Cooperative cache: Do not go to disk on cache miss, but look for it in your mounted xFS system, i.e. someone else cache
  - better hit rate on read of shared data
  - active clients can use the memory of idle clients
- 2. software RAID:distribute data on multiple clients=> better bandwidth
  - but what if one client has crashed?
  - Solution: data redundancy on multiple clients
- 3. Distributed control: instead of one centralized cache, disks, spread the management over multiple systems.

# xFS

- Promising future: crash resilient systems, data can live everywhere
- Ideal system for 24/7 systems, which we need nowadays
- Disk crash or replacement (“hot swaps”) can be easily provided
- Challenge: how do you upgrade OS, new versions of xFS
  Can we build systems which are are able to operate continuously for, say, a decade or more?

# summary

- Remote procedure calls
- udp - tcp network transmission layers
- Network File System
- xFS