15-410

"...Does this look familiar?..."

File System (Internals)
November 1, 2017

Dave Eckhardt

Dave O'Hallaron

Greg Hartman

Synchronization

Today

- **Chapter 12 (except log-structured, NFS, WAFL)**

Outline

File system code layers (abstract)

Disk, memory structures

Directories

Block allocation strategies, free space

Cache tricks

Recovery, backups

Outline

File system code layers (abstract)

Disk, memory structures

Unix "VFS" layering indirection

Directories

Block allocation strategies, free space

Cache tricks

Recovery, backups

File System Layers

Device drivers

- `read/write(disk, start-sector, count)`

Block I/O

- `read/write(partition, block)` [cached]

File I/O

- `read/write(file, block)`

File system

- manage directories, free space

Multi-filesystem namespace

- Partitioning, names for devices
- Mounting
- Unifying multiple file system *types*
  - UFS, ext2fs, ext3fs, zfs, FAT, 9660, ...

Shredding Disks

Split disk into *partitions*/slices/minidisks/...

- **MBR (PC): 4 "partitions" – Windows, FreeBSD, Plan 9, ...**
- **APM (Mac): "volumes" – can split: OS 9, OS X, user files**
- **GPT (new, multi-platform) - many partitions, long names**

Or: glue disks together into *volumes*/logical disks

A partition (of a disk or of a volume) may contain...

- **Paging area**
  - **Indexed by in-memory structures**
  - **"random garbage" when OS shuts down**
- **File system**
  - **Block allocation: file # ⇒ block list**
  - **Directory: name ⇒ file #**

Shredding Disks

```
# fdisk -s
/dev/ad0: 993 cyl 128 hd 63 sec
Part        Start        Size Type Flags
   1:          63     1233729 0x06 0x00
   2:     1233792     6773760 0xa5 0x80
```

(A 4-gigabyte disk)

Shredding Disks

```
8 partitions:
#        size   offset    fstype   [fsize bsize bps/cpg]
  a:   131072        0    4.2BSD   2048 16384   101   # (Cyl.    0 - 16*)
  b:   393216   131072      swap                      # (Cyl.   16*- 65*)
  c:  6773760        0    unused      0     0         # (Cyl.    0 - 839)
  e:    65536   524288    4.2BSD   2048 16384   104   # (Cyl.   65*- 73*)
  f:  6183936   589824    4.2BSD   2048 16384    89   # (Cyl.   73*- 839*)
```

```
Filesystem  1K-blocks    Used  Avail Capacity  Mounted on
/dev/ad0s2a     64462   55928   3378    94%    /
/dev/ad0s2f   3043806 2608458 191844    93%    /usr
/dev/ad0s2e     32206    7496  22134    25%    /var
procfs              4       4      0   100%    /proc
```

(FreeBSD 4.7 on ThinkPad 560X)

Shredding Disks

A disk can be split into *partitions*, each of which can contain:

- **Paging (swap) area**
  - **Indexed by in-memory structures**
  - **"random garbage" when OS shuts down**
- **File system**
  - **Block allocation: file # ⇒ block list**
  - **Directory: name ⇒ file #**

Anything containing a file system is known as a *volume.*

Volumes can span multiple disks.

Shredding Disks

```
bash:whaleshark> sudo fdisk -l

Disk /dev/sda: 1000.2 GB, 1000204886016 bytes
255 heads, 63 sectors/track, 121601 cylinders, total 1953525168
sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disk identifier: 0x00003502

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *        2048      999423      498688   83  Linux
/dev/sda2         1001470  1953523711   976261121    5  Extended
/dev/sda5         1001472    34351103    16674816   82  Linux swap
/dev/sda6        34353152    44351487     4999168   83  Linux
/dev/sda7        44353536    64352255     9999360   83  Linux
/dev/sda8        64354304   144353279    39999488   83  Linux
/dev/sda9       144355328  1953523711   904584192   83  Linux
```

Shredding Disks

```
bash:whaleshark> df -lhT
Filesystem     Type      Size  Used Avail Use% Mounted on
/dev/sda8      ext4       38G  8.6G   27G  25% /
...
/dev/sda1      ext4      464M   64M  373M  15% /boot
/dev/sda7      ext4      9.3G  1.3G  7.5G  15% /var
/dev/sda9      ext4      850G   51G  756G   7% /usr0
/dev/sda6      ext2      4.7G  3.6G  955M  80% /var/cache/openafs
AFS            afs       8.6G     0  8.6G   0% /afs
```

Disk Structures

Boot area (after BIOS)

- **Interpreted by hardware bootstrap (“BIOS”)**
- **May include partition table**

File system (volume) control block (Unix: “superblock)

- **Key parameters: #blocks, metadata layout**

Array of file control blocks (Unix: “inode”)

- **ownership/permissions**
- **data location**

Possibly a free-space map as well

Memory Structures

In-memory partition tables

- Sanity check file system I/O fits in correct partition

Cached directory information

System-wide in-core file state

- In-memory file control blocks

Process open-file tables

- Open mode (read/write/append/...)
- "Cursor" (read/write position)

VFS layer

Goals

- **Allow one machine to use multiple file system *types***
  - **Unix FFS**
  - **MS-DOS FAT**
  - **CD-ROM ISO9660**
  - **Remote/distributed: NFS/AFS**
- **Standard system calls should work transparently**

Solution?

VFS layer

Goals

- **Allow one machine to use multiple file system *types***
  - **Unix FFS**
  - **MS-DOS FAT**
  - **CD-ROM ISO9660**
  - **Remote/distributed: NFS/AFS**
- **Standard system calls should work transparently**

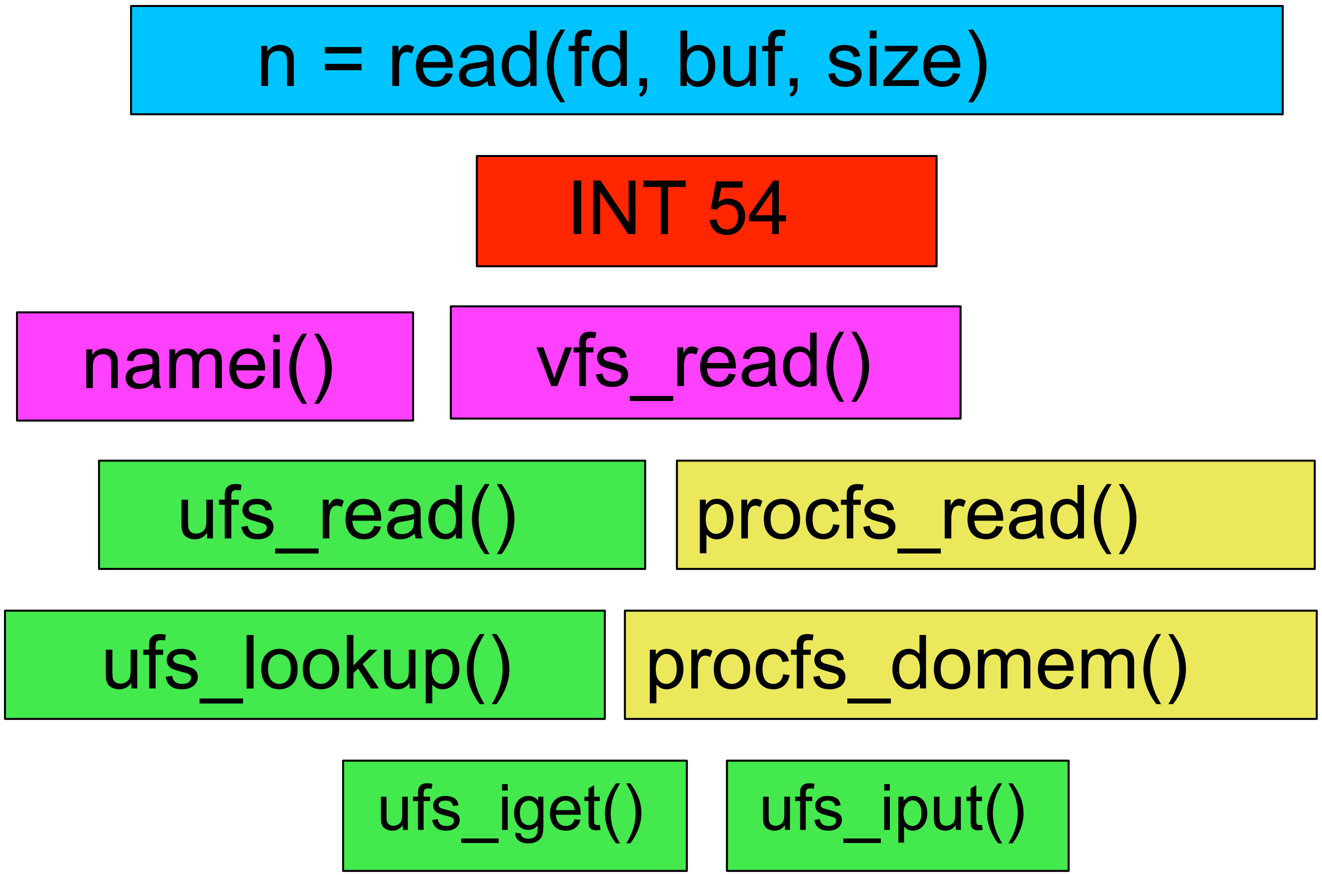
Solution

- **Insert a level of indirection!**

Single File System

```
n = read(fd, buf, size)

INT 54

sys_read(fd, buf, len)

namei()  iget()  iput()

sleep()  rdblk(dev, N)  wakeup()

startIDE()  IDEintr()
```

VFS "Virtualization"



VFS layer – file system operations

These operate on file *systems*, not individual files

```
struct vfsops {
  char *name;
  int (*vfs_mount)();
  int (*vfs_statfs)();
  int (*vfs_vget)();
  int (*vfs_unmount)();
  ...
}
```

VFS layer – file operations

Each VFS provides an array of per-file methods

- VOP_LOOKUP(vnode, new_vnode, name)
- VOP_CREATE(vnode, new_vnode, name, attributes)
- VOP_OPEN(vnode, mode, credentials, process)
- VOP_READ(vnode, uio, readwrite, credentials)

Operating system provides fs-independent code

- Validating system call parameters
- Moving data from/to user memory
- Thread sleep/wakeup
- Caches (data blocks, name ⇒ vnode mappings)

Directories

Old: one namei() ⇒ VFS: fs-provided vnode method

- **vnode2 = VOP_LOOKUP(vnode1, name)**

Traditional Unix FFS directories

- **List of (name,inode #) - not sorted!**
- **Names are variable-length**
- **Lookup is linear**
  - **How long does it take to delete N files?**

Common alternative: hash-table directories

Allocation / Mapping

Allocation problem

- **Where do I put the next block of this file?**
  - **"Near the previous block" is not a bad idea**
  - **Beyond that, it gets complicated**

Mapping problem

- **Where was block 32 of this file previously put?**
- **Similar to virtual memory**
  - **Multiple large "address spaces"** *specific to each file*
  - **Only one underlying "address space" of blocks**
  - **Source address space may be sparse!**

Allocation / Mapping

- **Contiguous**
- **Linked**
- **FAT**
- **Indexed**
  - **Base**
  - **Linked**
  - **Multi-level**
  - **Unix (index tree)**

Allocation – Contiguous

Approach

- File location defined as (start, length)

Motivation

- Sequential disk accesses are cheap
- Bookkeeping is easy

Issues

- Dynamic storage allocation (fragmentation, compaction)
- Must pre-declare file size at creation
- This should sound familiar

Allocation – Linked

Approach

- File location defined as (start)
- Each disk block contains pointer to next block

Motivation

- Avoids fragmentation problems
- Allows file growth

Issues?

Allocation – Linked

Issues

- 508-byte blocks don't match memory pages
- In general, one seek per block read/written - *slow!*
- *Very* hard to access file blocks at random
  - lseek(fd, 37 * 1024, SEEK_SET);

Benefit

- Files can grow dynamically

Common modification

- Link multi-block *clusters*, not blocks

Allocation – FAT

Used by MS-DOS, OS/2, Windows

- Digital cameras, GPS receivers, printers, PalmOS, ...

Semantically the same as linked allocation

But next-block links stored "out of band" in a table

- Result: nice 512-byte sectors for data

Table at start of disk

- Next-block pointer array
- Indexed by block number
- Next = 0 means "free"
- Next = -1 means "end of file"

Allocation – FAT

7
2
5
-1
3
-1
0
-1

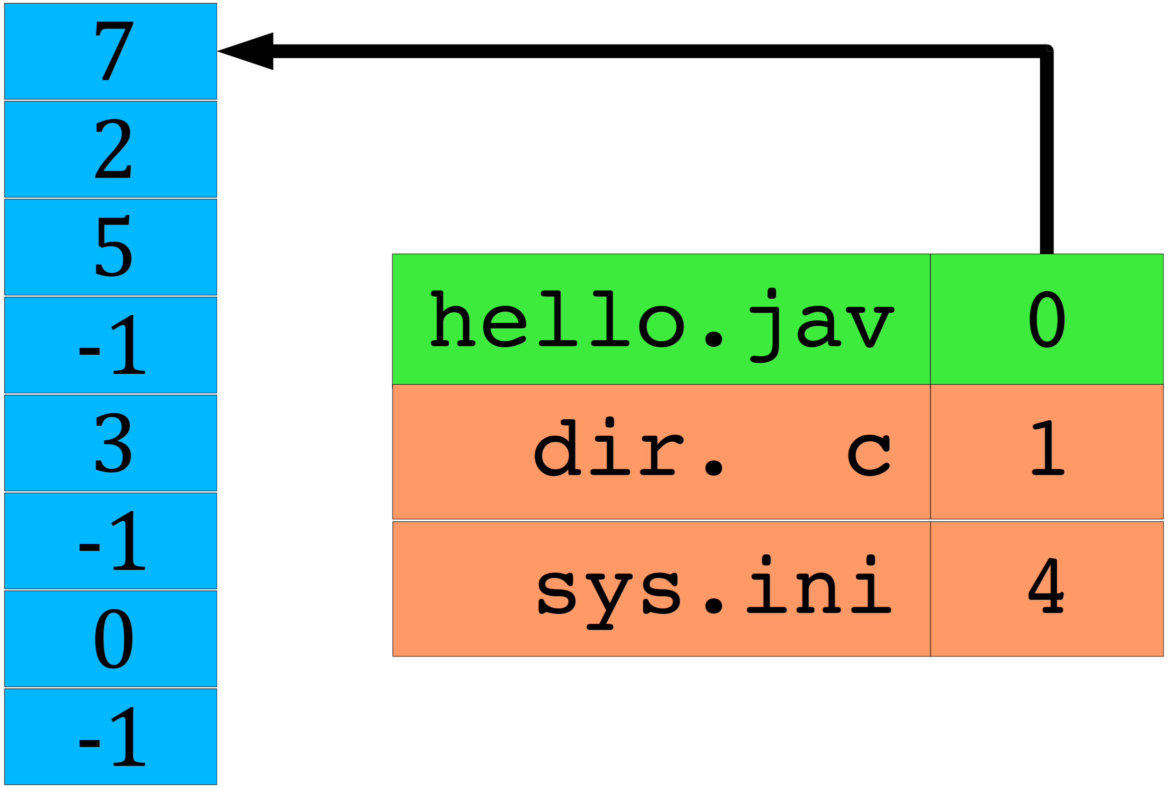
hello.jav	0
dir. c	1
sys.ini	4

Allocation - FAT

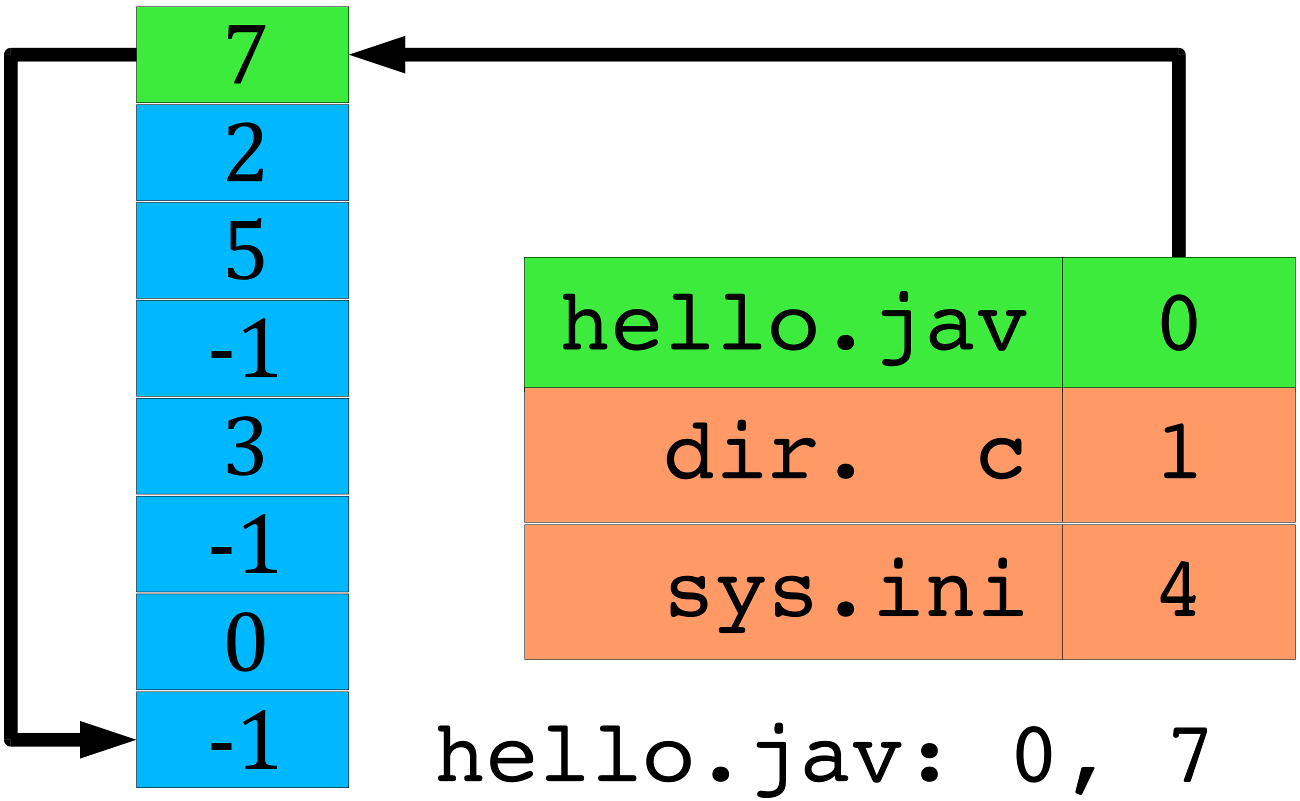
7
2
5
-1
3
-1
0
-1

hello.jav	0
dir. c	1
sys.ini	4

Allocation - FAT



Allocation - FAT



hello.jav: 0, 7

Allocation – FAT

Issues

- Damage to FAT scrambles entire file system
  - Solution: mirror the FAT
- Generally *two* seeks per block read/write
  - Seek to FAT, read, seek to actual block (repeat)
  - Unless FAT can be cached well in RAM
- Still *somewhat* hard to access random file blocks
  - Linear time to walk through FAT
- FAT may be a “hot spot” (everybody needs to access it)
- Lots of FAT updates (near beginning of disk)
  - Even if files being modified are far away

Allocation – Indexed

Motivation

- **Avoid fragmentation problems**
- **Allow file growth**
- *Improve random access*

Approach

- *Per-file* **block array**
- **File block number indexes into table, yields disk block number**
- **No O(n) sequential steps**

99	3004
100	-1
101	-1
3001	-1
3002	6002
-1	-1
-1	-1
-1	-1

Allocation – Indexed

Allows "holes"

- foo.c is sequential
- foo.db, blocks 1..3 ⇒ -1
  - logically "blank"

"sparse allocation"

- a.k.a. "holes"
- read() returns nulls
- write() requires alloc
- file "size" ≠ file "size"
  - `ls -l` index of last byte
  - `ls -s` number of blocks

foo.c	foo.db
99	3004
100	-1
101	-1
3001	-1
3002	6002
-1	-1
-1	-1
-1	-1

Allocation – Indexed

How big should index block be?

- Too small: limits file size
- Too big: lots of wasted pointers

Combining index blocks

- Linked
- Multi-level
- What Unix actually does

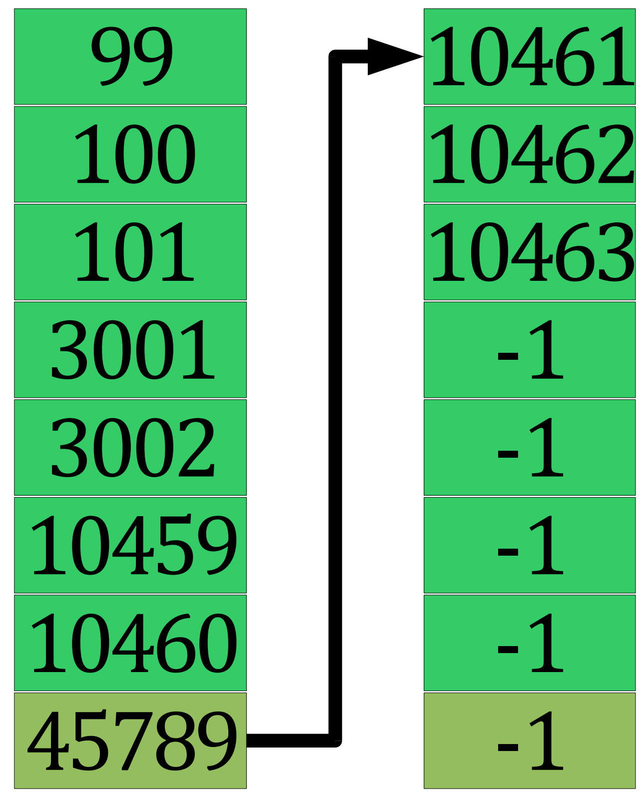
Linked Index Blocks

Last pointer indicates next index block

Simple

Access is not-so-random

- O(n/c) is still O(n)
- O(n) ***disk transfers***

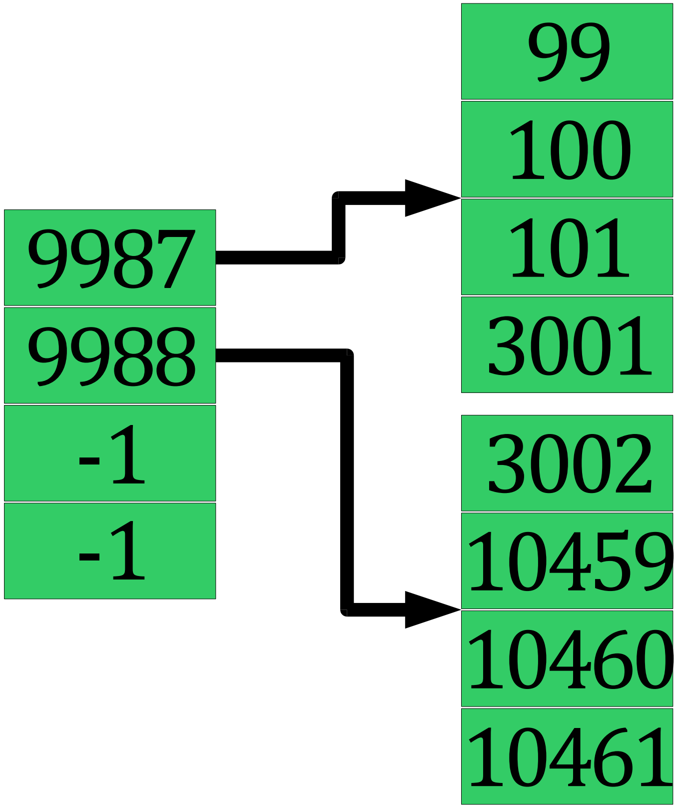


Multi-Level Index Blocks

Index blocks of index blocks

Does this look familiar?

Allows *big* holes



Unix Index Blocks

Intuition

- *Many* files are small
  - Length = 0, length = 1, length < 80, ...
- Some files are *huge* (gigabytes... maybe terabytes!)

How do we solve this problem?

- We are computer scientists!

Unix Index Blocks

Intuition

- *Many* files are small
  - Length = 0, length = 1, length < 80, ...
- Some files are *huge* (gigabytes... maybe terabytes!)

How do we solve this problem?

- We are computer scientists!
  - So we realize when 57 levels of indirection would be slow!!!

Unix Index Blocks

Intuition

- *Many* files are small
  - Length = 0, length = 1, length < 80, ...
- Some files are *huge* (gigabytes... maybe terabytes!)

"Clever heuristic" in Unix FFS inode

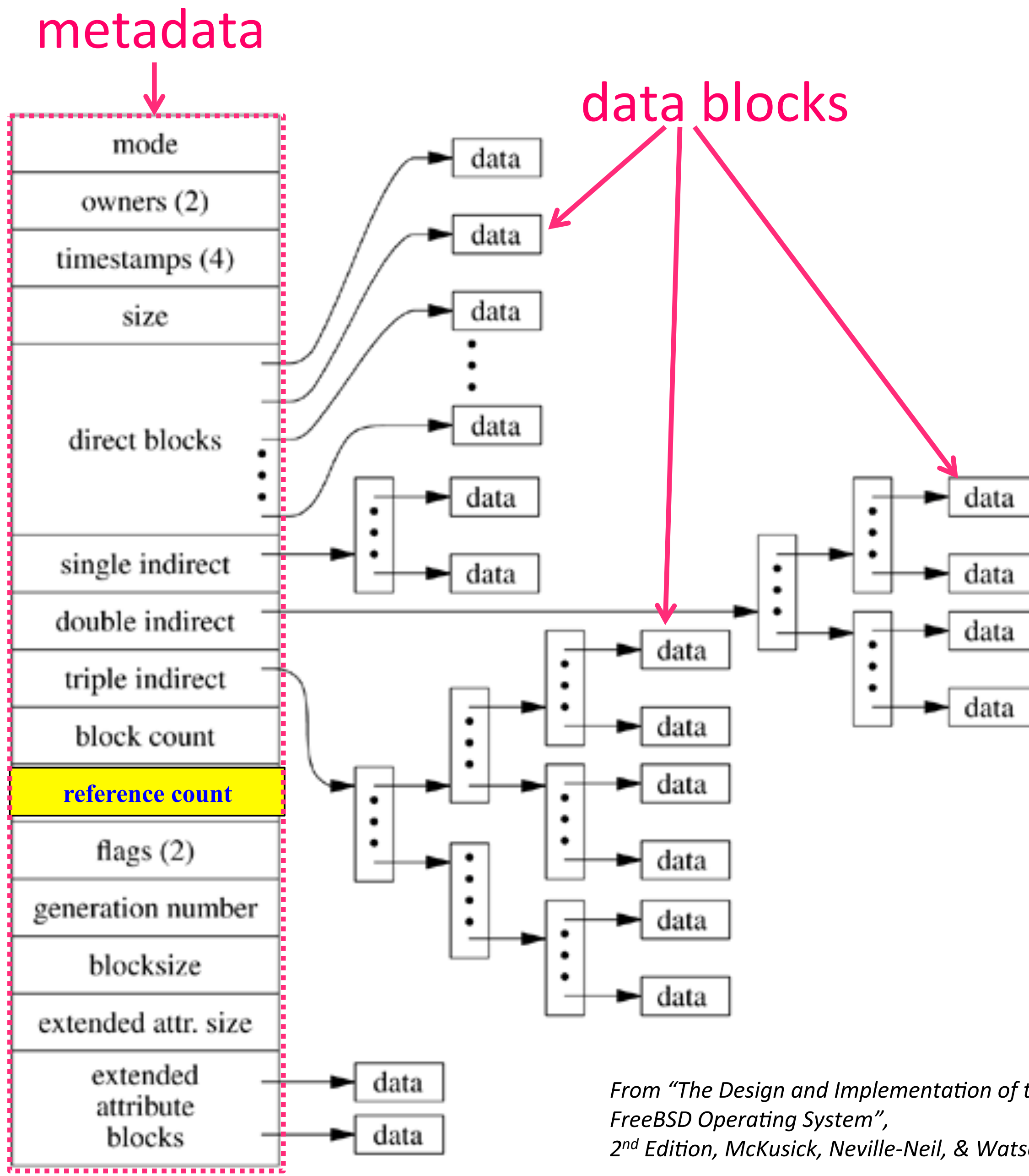
- inode struct contains 12 "direct" block pointers
  - 12 block numbers * 8 KB/block = 96 KB
  - Availability is "free" - must read inode to open() file anyway
- inode struct also contains 3 indirect block pointers
  - single-indirect, double-indirect, triple-indirect

inode (index node) (FreeBSD FFS)

inode structure:

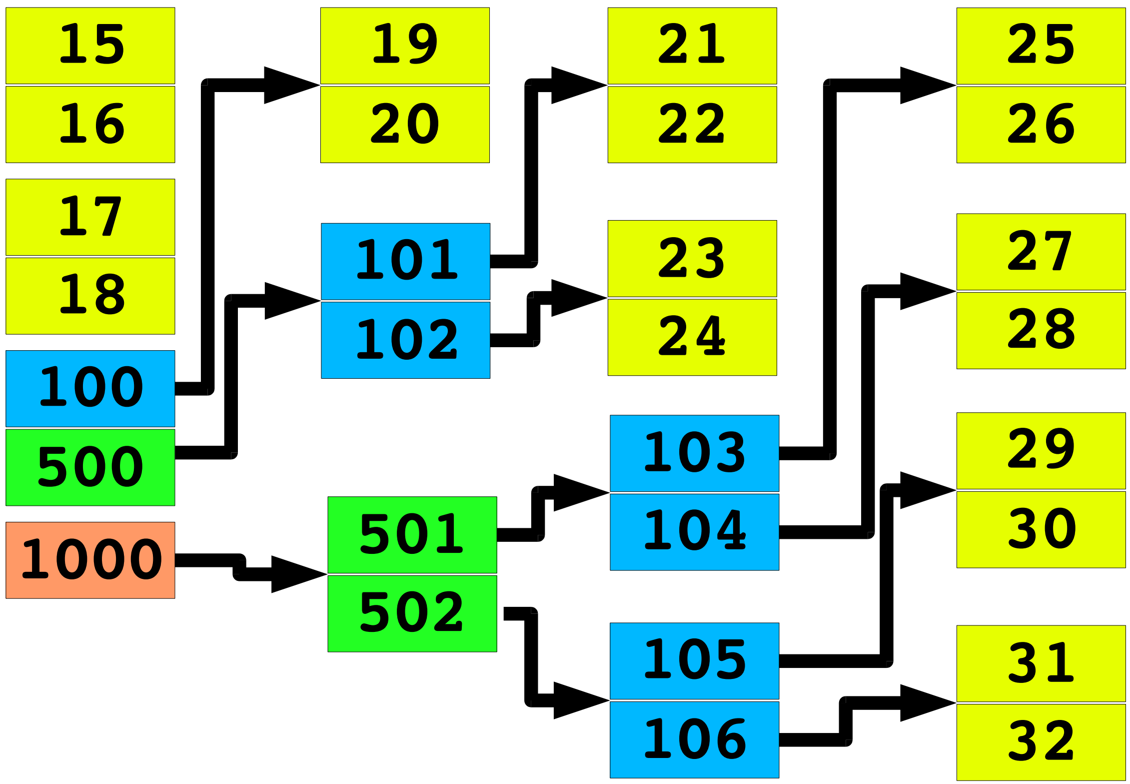
Note: *filename* is not stored here!

How many hard links point to this file? →

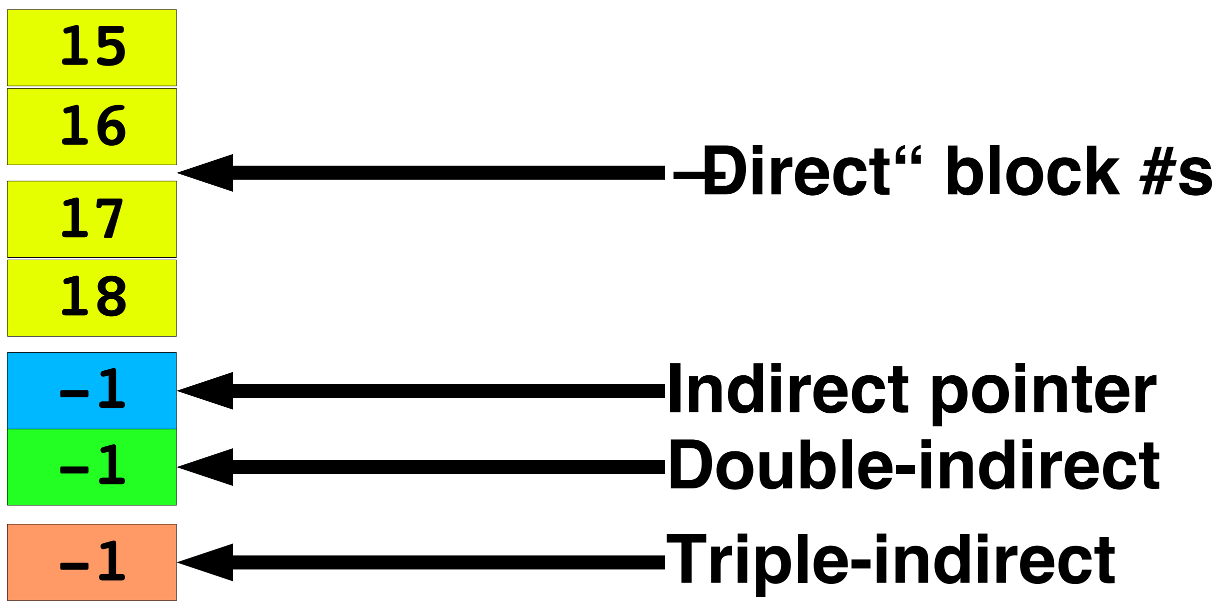


From "The Design and Implementation of the FreeBSD Operating System", 2nd Edition, McKusick, Neville-Neil, & Watson.

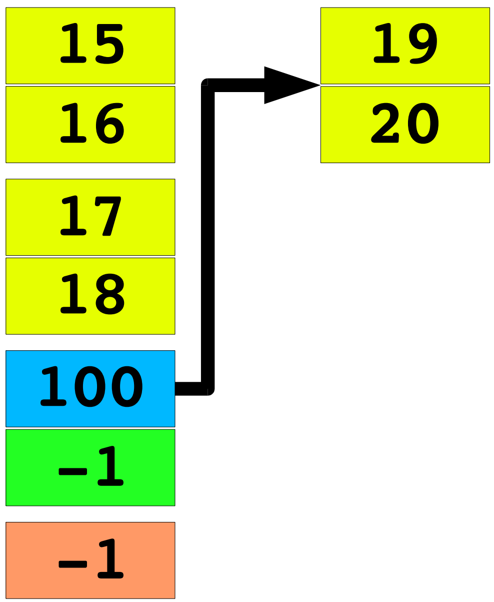
Unix Index Blocks



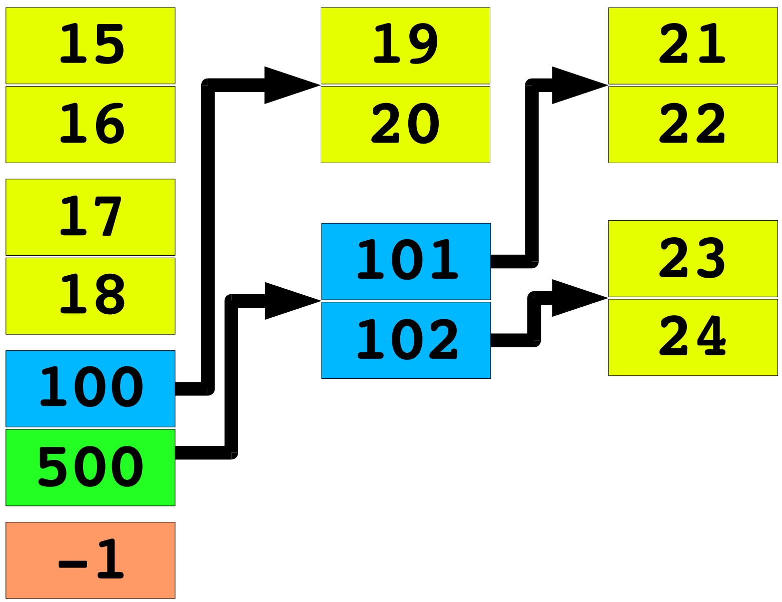
Unix Index Blocks



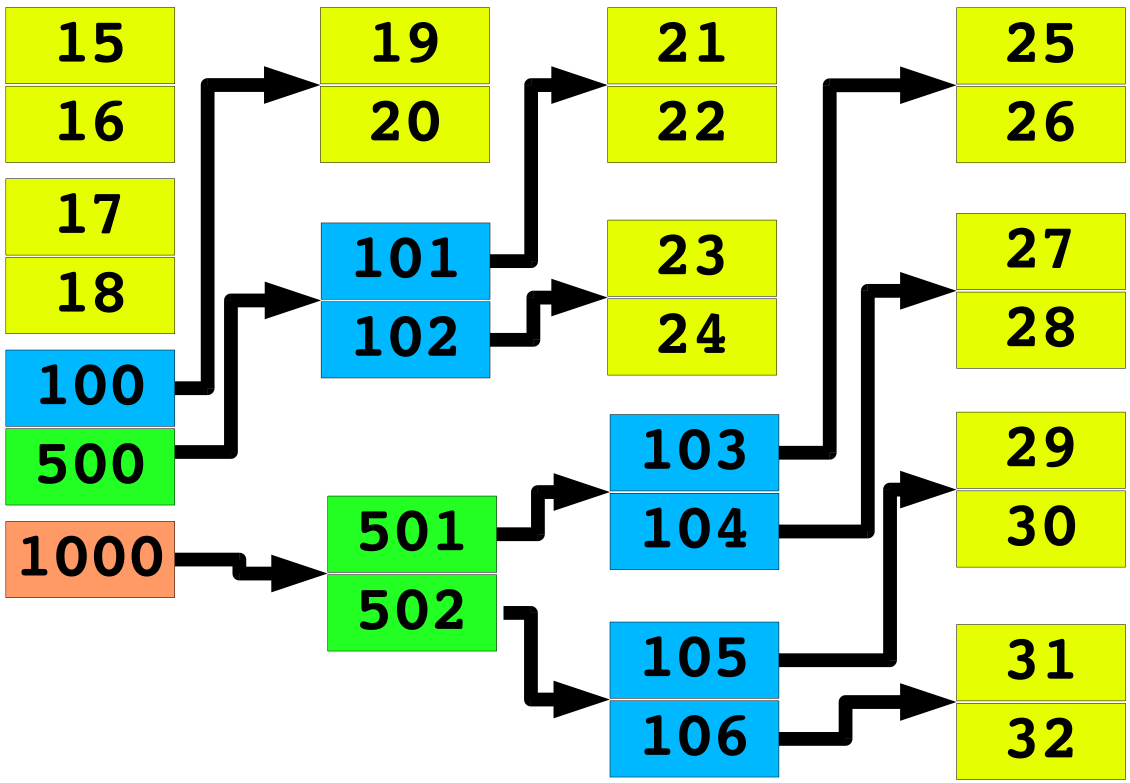
Unix Index Blocks



Unix Index Blocks



Unix Index Blocks



Triple indirect can address >> 2^{32} bytes

Tracking Free Space

Bit-vector

- 1 bit per block: boolean "free"
- Check each word vs. 0
- Use "first bit set" instruction
  - x86: BSF (Bit Scan Forward)
- Text examples
  - 1.3 GB disk, 512 B blocks: 332 KB bit vector
  - 1 TB disk, 4KB blocks: 256 MB

Need to keep (much of) it in RAM

Tracking Free Space

Linked list?

- Superblock points to first free block
- Each free block points to next

Cost to allocate N blocks is linear

FAT approach provides free-block list “for free”

- Table value of 0 -> free

Keep free-*extent* lists

- (block,sequential-block-count)

Unified Buffer Cache

Traditional two-cache approach

- Page cache, file-system cache often totally independent
  - Page cache chunks according to hardware page size
  - File cache chunks according to "file system block" size
  - Different code, different RAM pools
- How much RAM to devote to each one?

Observation

- Why not have just one unified cache?
  - Mix automatically varies according to load
    - » "cc" wants more disk cache
    - » Firefox wants more VM cache
- Examples
  - MacOS X 10.1 (2001)
  - Linux 2.4 (2001)

Unified Buffer Cache - Warning!

“Virtual memory architecture in SunOS”

Gingell, Moran, & Shannon

USENIX 1987 Summer Conference

“The work has consumed approximately four man-years of effort over a year and a half of real time. A surprisingly large amount of effort has been drained by efforts to interpose the VM system as the logical cache manager for the file systems…”

Cache tricks

Read-ahead

```
for (i = 0; i < filesize; ++i)
  putc(getc(infile), outfile);
```

- System observes sequential reads
  - File block 0, 1, 2, ...
  - Can pipeline reads to overlap “computation”, read latency
    - » Request for block 2 triggers disk read of block 3

Free-behind / replace-behind

- Discard buffer from cache when next is requested
- Good for large files
- “Anti-LRU” (evict “MRU” instead of “LRU”)

Recovery

System crash...now what?

- Some RAM contents were lost
- Free-space list on disk may be wrong
- Scan file system (fsck)
  - Check invariants
    - » Unreferenced files
    - » Double-allocated blocks
    - » Unallocated blocks
  - Fix problems
    - » Expert user???

Modern approach

- "Journal" changes (see upcoming Transactions lecture)

Backups

Incremental "Towers of Hanoi" approach - traditional

- Monthly: dump entire file system
- Weekly: dump changes since last monthly
- Daily: dump changes since last weekly
- Restore a file?
  - Most-recent "monthly" tape definitely has a copy
    - » May be stale, so...
  - Any one of the "weekly" tapes might have a copy (scan all)
  - Any one of the "daily" tapes might have a copy (scan all)

Backups

Merge approach ("TiBS") - www.teradactyl.com

- Something special about tape drives
- They run *much* faster when they're "streaming" (continuous full speed, no start/stop)
- Collect changes since yesterday
  - Scan file system by modification time
- "Output" tape drive has a blank tape
- "Input" tape drive streams yesterday's dump in
  - Some files are un-changed: stream to output tape
  - Some files are stale: replace them in output stream
- Keep as many tapes as you want to, recycle the rest
- Restoring is fast (stream *one* tape onto disks)
- Files stored (very) redundantly – good for reliability

Backups

Snapshot approach

- At midnight, stop writing into file system
- New writes go into a new file system
  - Mostly pointers to yesterday's data
  - Changes stored in the live file system
    - » Maybe entire files (copy-on-write)
    - » Maybe just new data blocks
- Great for users
  - Old snapshots can be mounted (read-only)
  - Accidentally delete a file? Get it from yesterday!
  - AFS supports a simple version (see "OldFiles")

Summary

Block-mapping problem

- Similar to virtual-to-physical mapping for memory
- Large, often-sparse "address" spaces
  - "Holes" not the common case, but not impossible
- Map any "logical address" to any "physical address"
- Key difference: file maps often don't fit in memory

"Insert a level of indirection"

- Multiple file system types on one machine
- Grow your block-allocation map
- ...

Summary

Block-mapping problem

- Similar to virtual-to-physical mapping for memory
- Large, often-sparse "address" spaces
  - "Holes" not the common case, but not impossible
- Map any "logical address" to any "physical address"
- Key difference: file maps often don't fit in memory

"Insert a level of indirection"

- Multiple file system types on one machine
- Grow your block-allocation map
- ...

Further Reading

Journaling

- **Prabhakaran et al., Analysis and Evolution of Journaling File Systems (USENIX 2005)**

Something cool which isn't journaling

- **McKusick & Ganger: "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem" (USENIX 1999)**

Both papers appear in the "filesystem reliability" book report paper track