15-410

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

File System (Internals)

Nov. 3, 2008

Dave Eckhardt

Roger Dannenberg

Greg Hartman

Synchronization

Today

- **Chapter 11 (not: Log-structured, NFS, WAFL)**

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

File System Layers

Multi-filesystem namespace

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

Shredding Disks

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

- **PC: 4 "partitions" –e.g., Windows, FreeBSD, Plan 9**
- **Mac: "volumes" –can do: OS 9, OS X, user files**

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

Partition 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)
```

Disk Structures

Boot area (first block/track/cylinder)

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

File system control block

- **Key parameters: #blocks, metadata layout**
- **Unix: "superblock"**

"File control block" (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 in correct partition**

Cached directory information

System-wide open-file table

- **In-memory file control blocks**

Process open-file tables

- **Open mode (read/write/append/...)**
- **“Cursor” (read/write position)**

VFS layer

Goal

- **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**

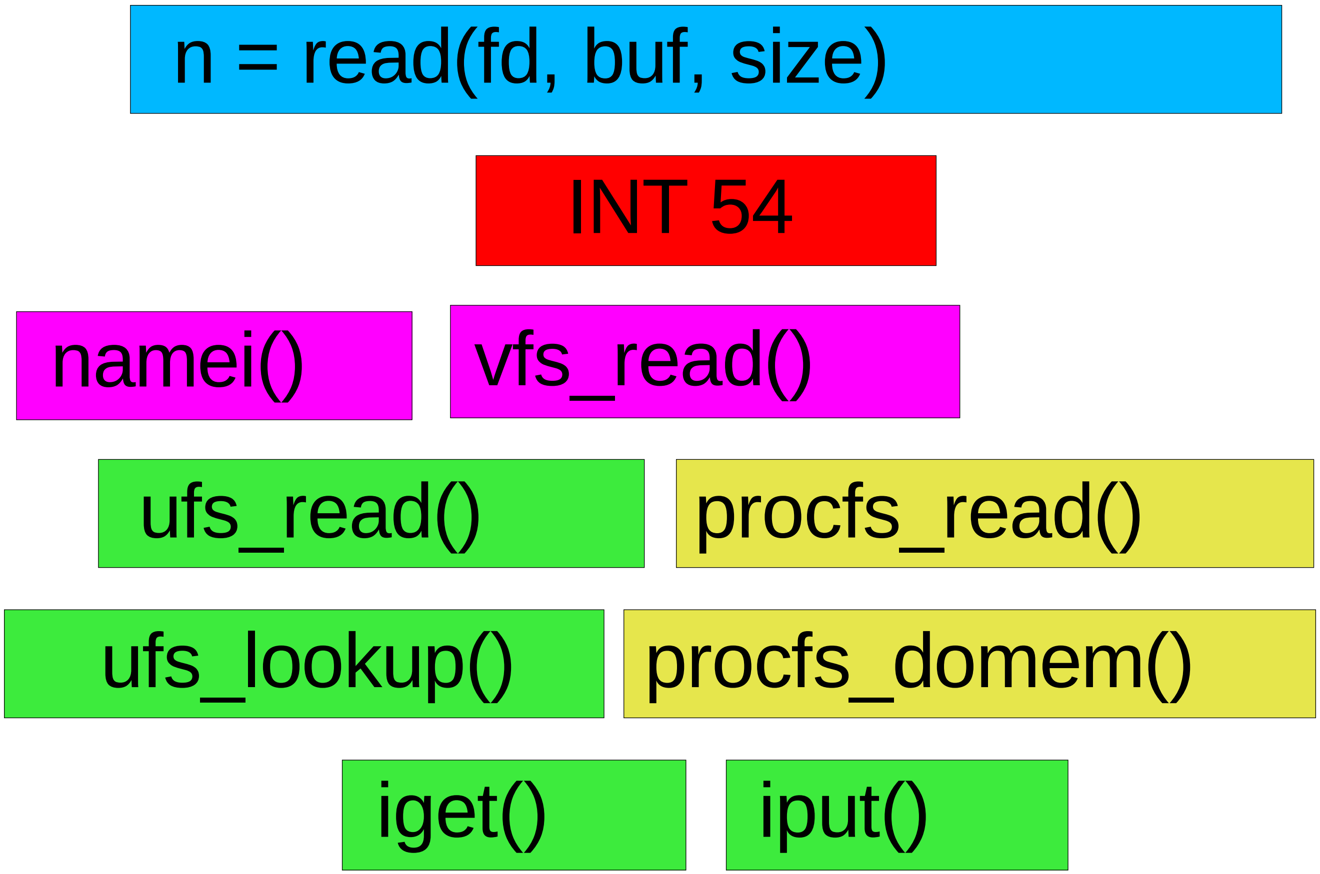
Solution

- **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

- **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

- **Can recover files even if directories destroyed**

Common modification

- **Link multi-block** *clusters***, not blocks**

Allocation –FAT

Used by MS-DOS, OS/2, Windows

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

Semantically same as linked allocation

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”**

Allocation –FAT

7
2
5
-1
3
-1
0
-1

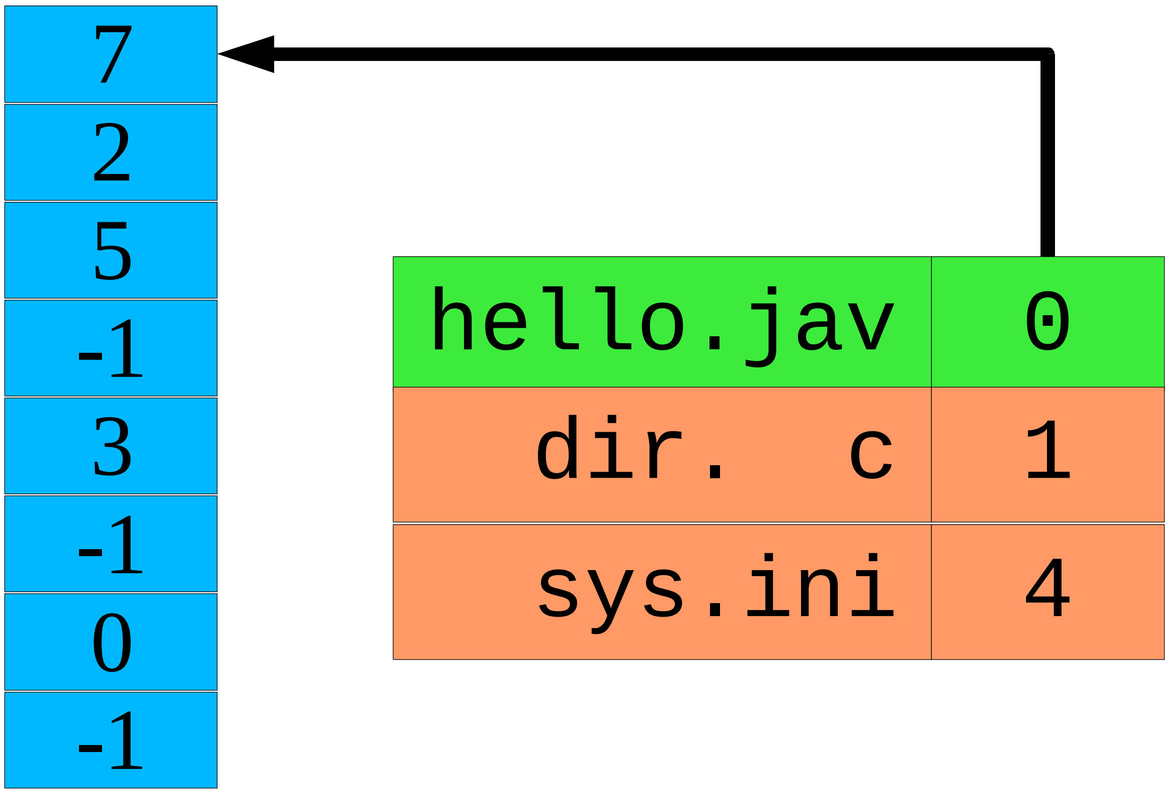
hello.jav	0
dir. c	1
sys.ini	4

Allocation - FAT

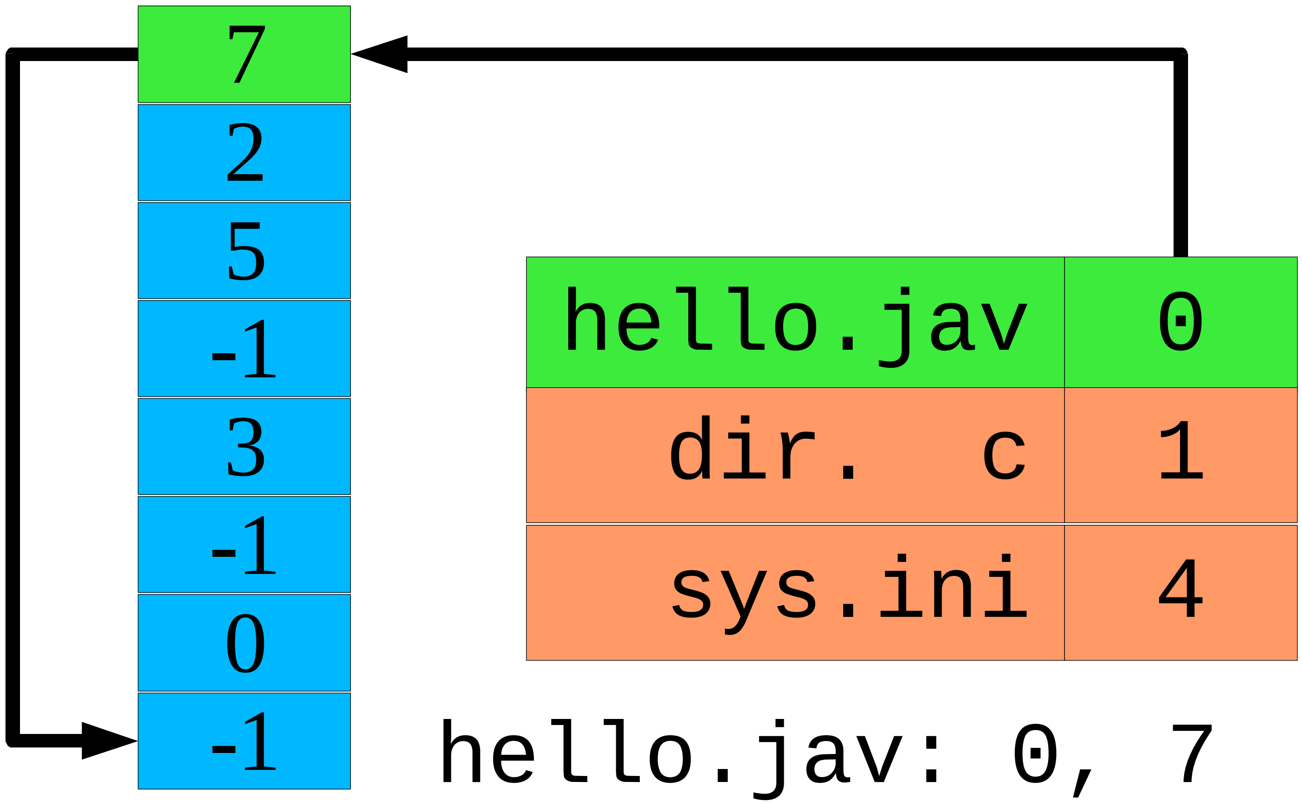
7
2
5
-1
3
-1
0
-1

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

Allocation - FAT



Allocation - FAT



Allocation –FAT

Issues

- **Damage to FAT scrambles entire file system**
  - **Solution: backup 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**

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

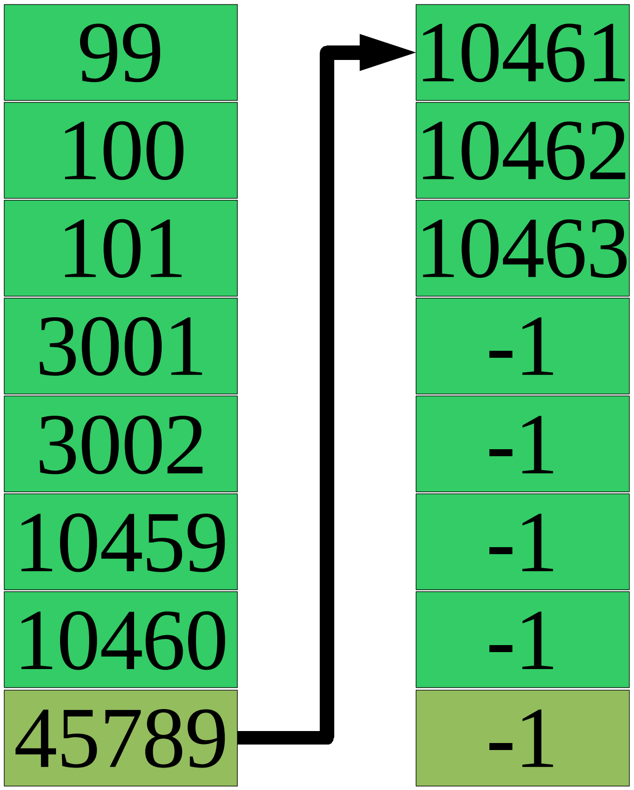
Linked Index Blocks

Last pointer indicates next index block

Simple

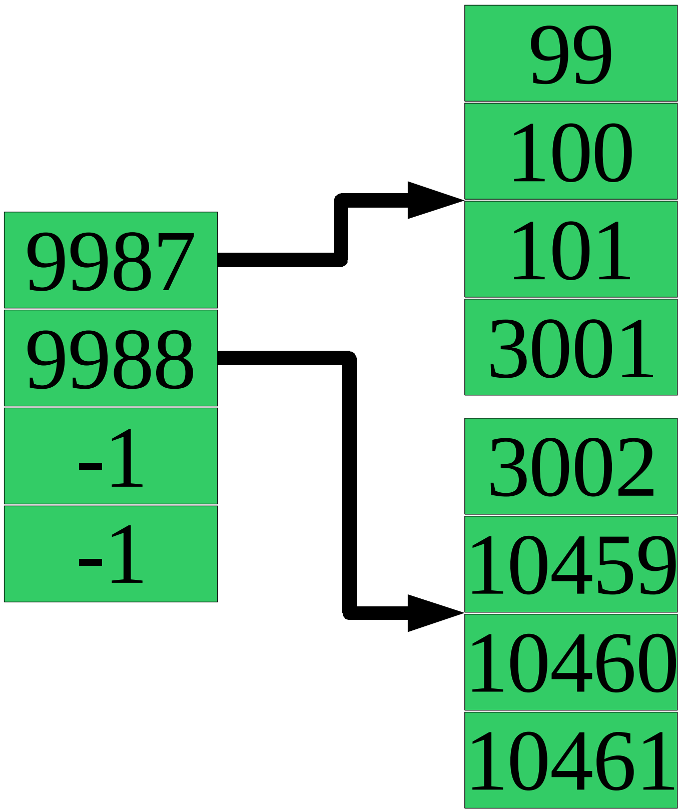
Access is not-so-random

- **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

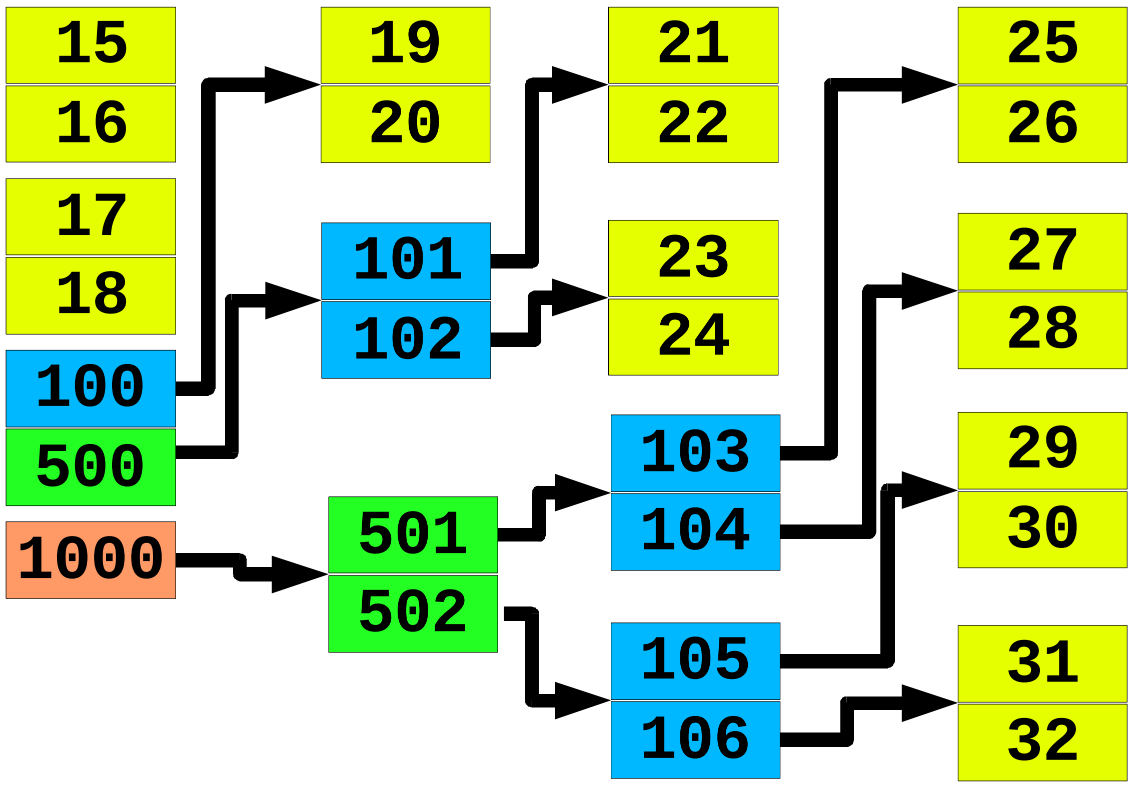
Intuition

- *Many* files are small
  - Length = 0, length = 1, length < 80, ...
- Some files are *huge* (3 gigabytes)

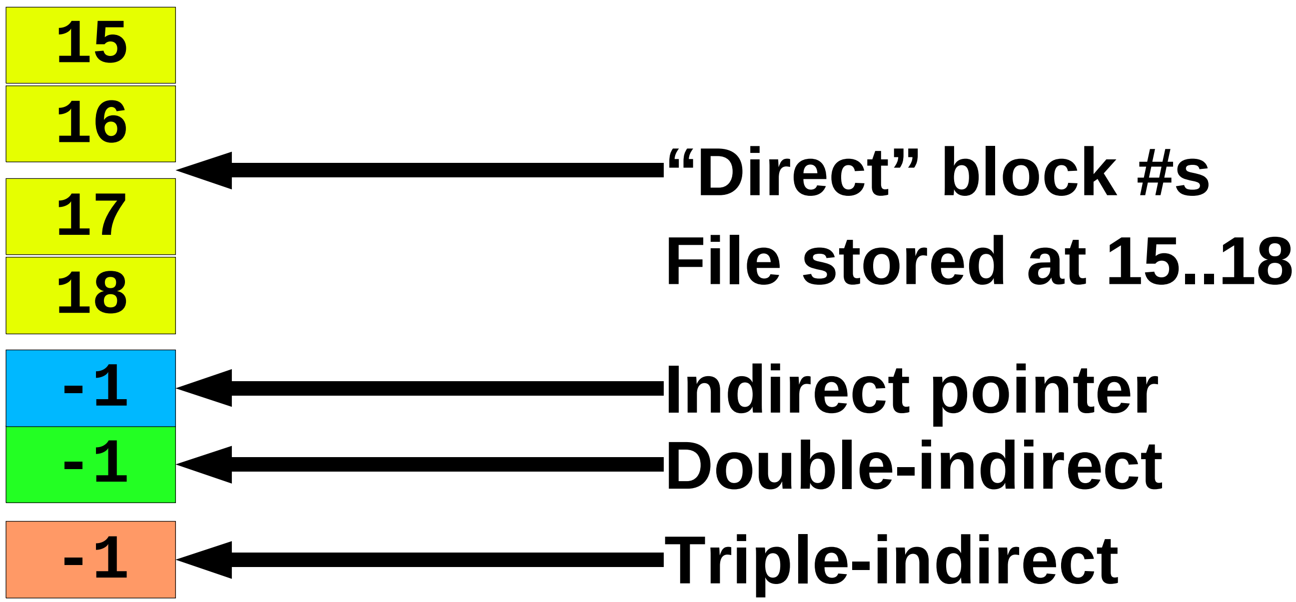
"Clever heuristic" in Unix FFS inode

- 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

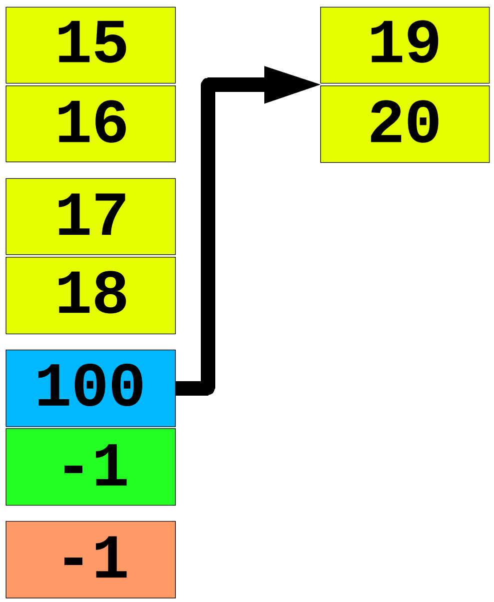
Unix Index Blocks



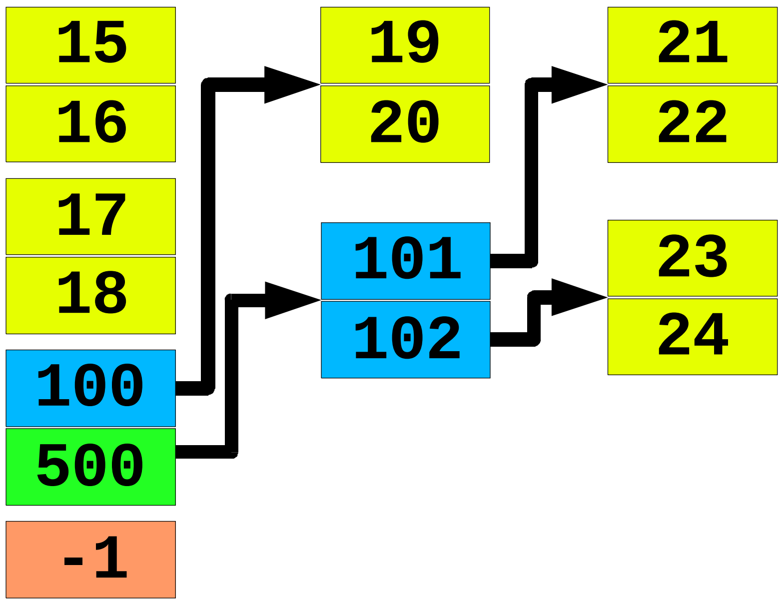
Unix Index Blocks



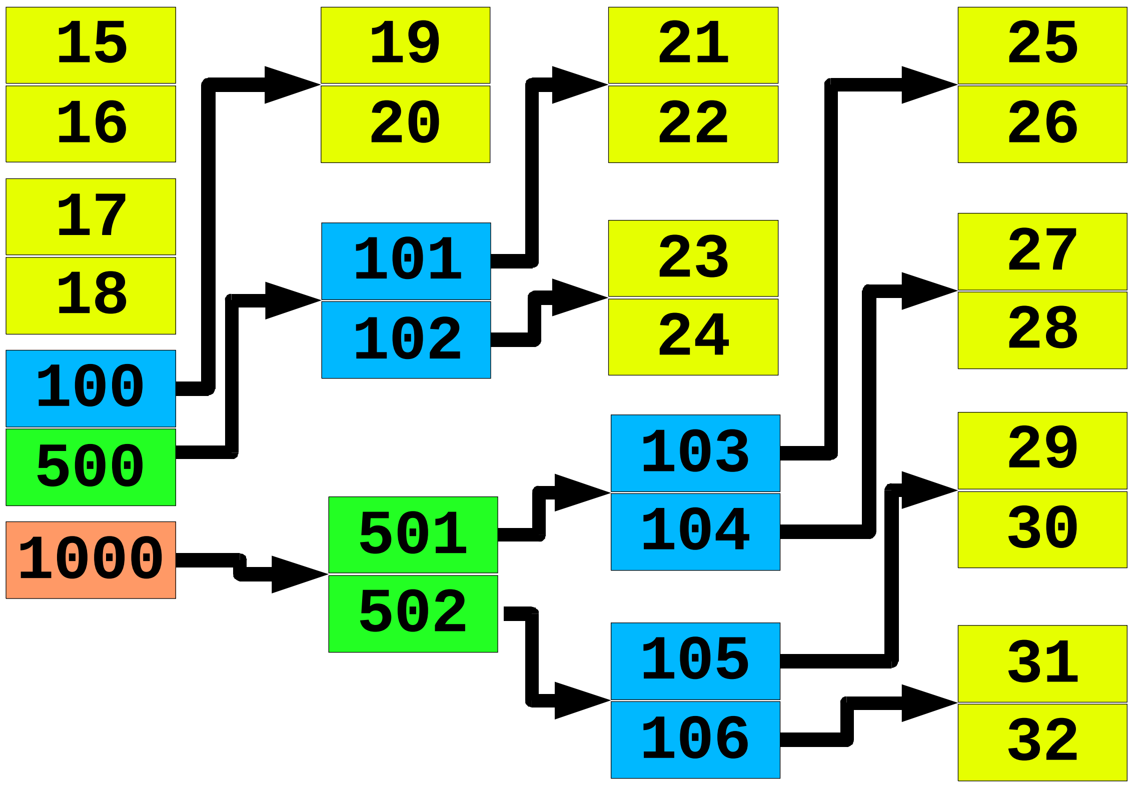
Unix Index Blocks



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**
- **Text example**
  - **1.3 GB disk, 512 B sectors: 332 KB bit vector**

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

- **Free block can point to** *multiple* **free blocks**
  - **512 bytes = 128 (4-byte) block numbers**
- **FAT approach provides free-block list "for 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 cache?
  - Mix automatically varies according to load
    - » “cc” wants more disk cache
    - » Firefox wants more VM cache

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

- Discard buffer from cache when next is requested
- Good for large files
- “Anti-LRU”

Recovery

System crash...now what?

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

Modern approach

- "Journal" changes (see upcoming Transactions lecture)

Backups

Incremental approach

- **Monthly: dump entire file system**
- **Weekly: dump changes since last monthly**
- **Daily: dump changes since last weekly**

Merge approach - www.teradactyl.com

- **Collect changes since yesterday**
  - **Scan file system by modification time**
- **Two tape drives merge yesterday's tape, today's delta**

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