

15-410

“...Does this look familiar?...”

File System (Internals)

Mar. 31, 2004

Dave Eckhardt

Bruce Maggs

Synchronization

Project 3 status

- Several groups skipped Checkpoint 3 (the easiest one!)
- Not everybody took advantage of opportunity to *plan*
- Several groups seem on track to finish early
- Several groups dangerously close to the “90% problem”
 - First 90% of the work takes the first 90% of the time
 - Last 10% of the work takes the second 90% of the time

We want everybody to finish!

- Project 3 is the core experience of the class
- Can't bury it and move on!

Synchronization

Project 3 / Project 4 “hurdle” test suite

- Released this week
- Two sections
 - Basic tests, solidity tests

At P3 deadline, you will run the tests

- Goal: pass ~80% of *each section*
- Register to begin Project 4 (some P3 extensions)

Not passing the hurdle?

- Extra week to work on P3
- *Cannot submit P4, grade will be 0%*

Synchronization

Today

- Chapter 12 (not: Log-structured, NFS)

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” – Windows, FreeBSD, Plan 9
- Mac: “volumes” – OS 9, OS X, system vs. user data

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 # \Rightarrow block list
 - Directory: name \Rightarrow file #

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

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”

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

```
INT 54
```

```
vfs_read()
```

```
ufs_read()
```

```
procfs_read()
```

```
namei()
```

```
procfs_domem()
```

```
iget()
```

```
iput()
```

VFS layer – file system operations

```
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 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 \Rightarrow inode mappings)

Directories

External interface

- `vnode2 = 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?

Mapping problem

- Where is block 32 of this file?
- 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 – 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

Allocation – Linked

Approach

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

Motivation

- Avoid fragmentation problems
- Allow 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

- Linked 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

Links stored “out of band” in 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

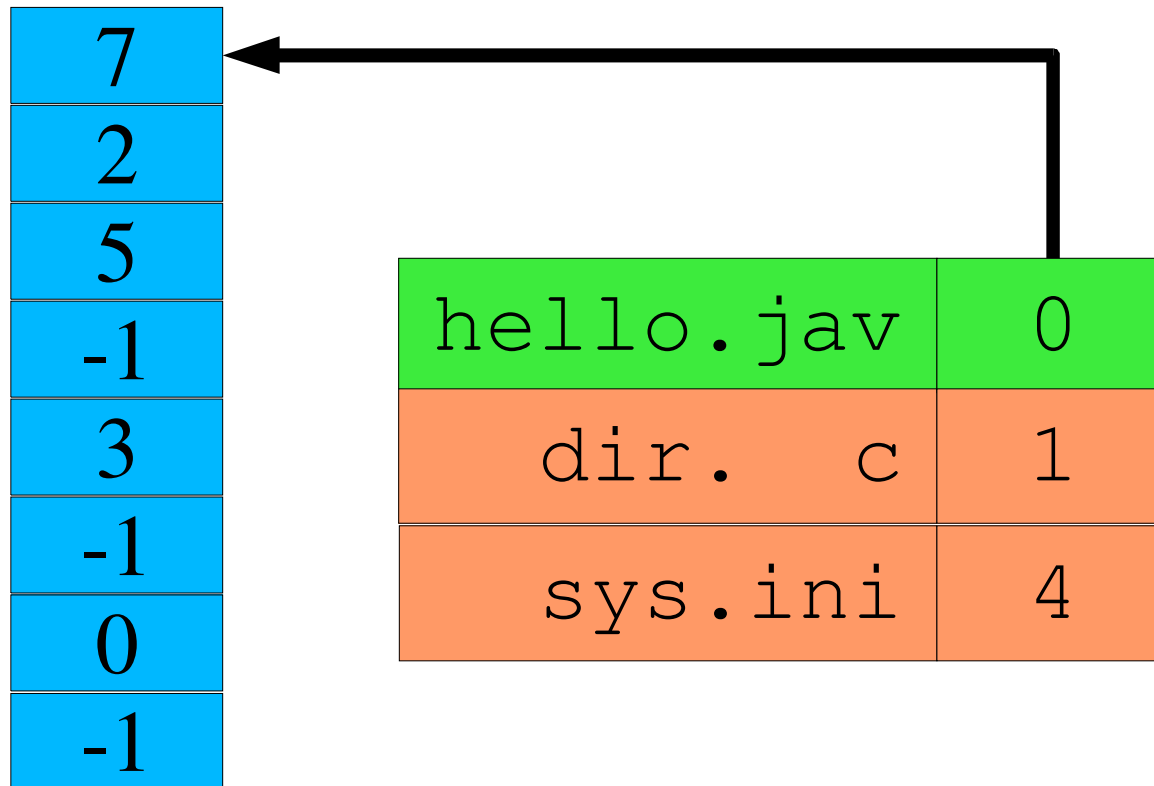
hello.jav	0
dir. c	1
sys.ini	4

Allocation - FAT

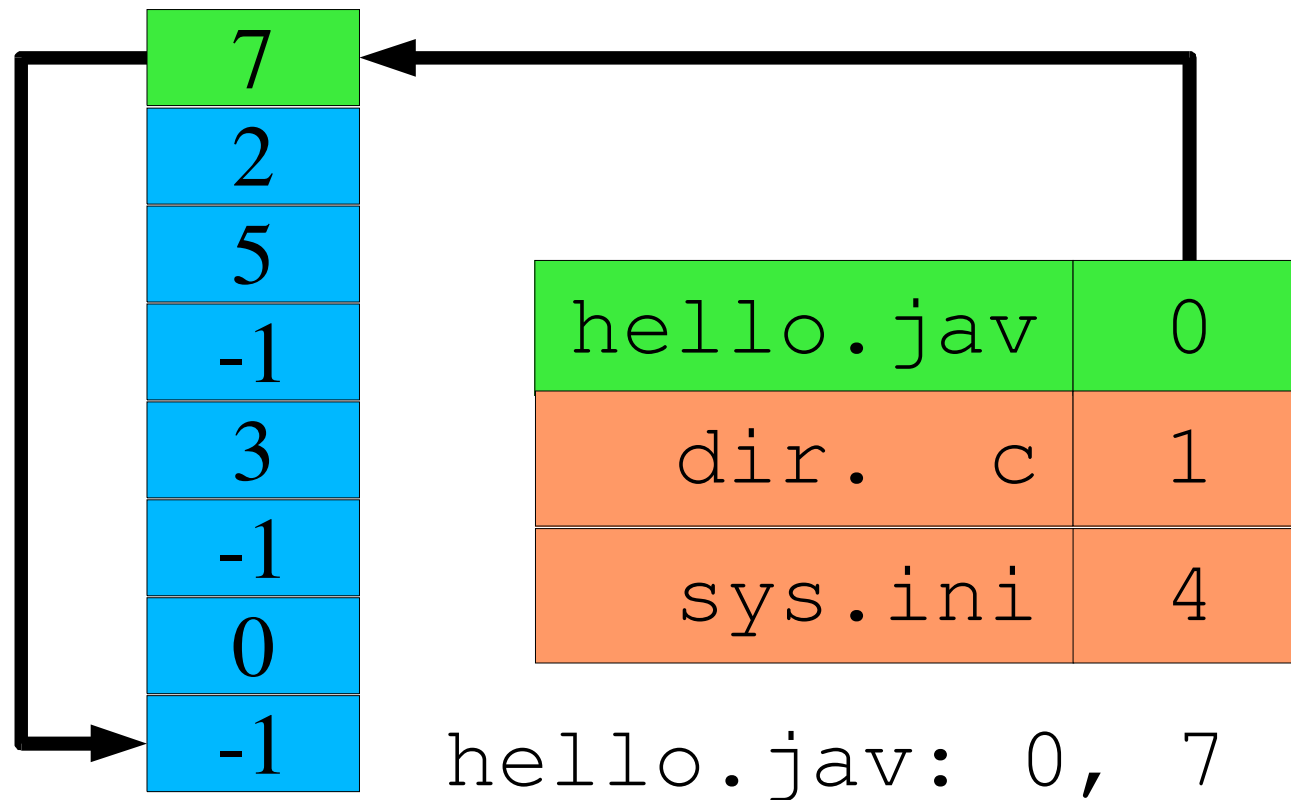
7
2
5
-1
3
-1
0
-1

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

Allocation - FAT



Allocation - FAT



Allocation – FAT

Issues

- Damage to FAT scrambles entire disk
 - Solution: backup FAT
- Generally *two* seeks per block read/write
 - Seek to FAT, read, seek to actual block (repeat)
 - Unless FAT can be cached
- Still *very* hard to access random file blocks
 - Linear time to walk through FAT

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 \Rightarrow -1
 - logically “blank”

“sparse allocation”

- a.k.a. “holes”
- `read()` returns nulls
- `write()` requires alloc
- file “size” file “size”
 - `ls -l`
 - `ls -s`

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

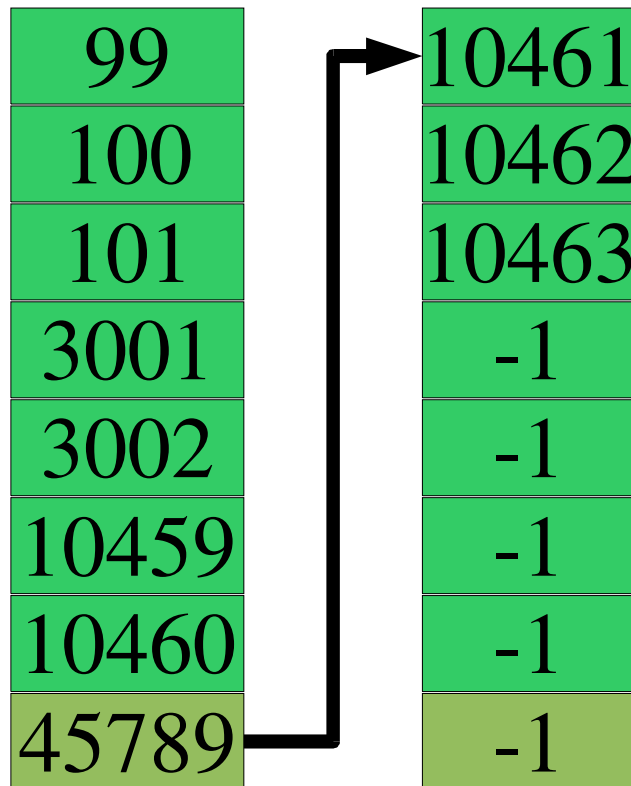
Linked Index Blocks

**Last pointer indicates
next index block**

Simple

Access is not-so-random

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

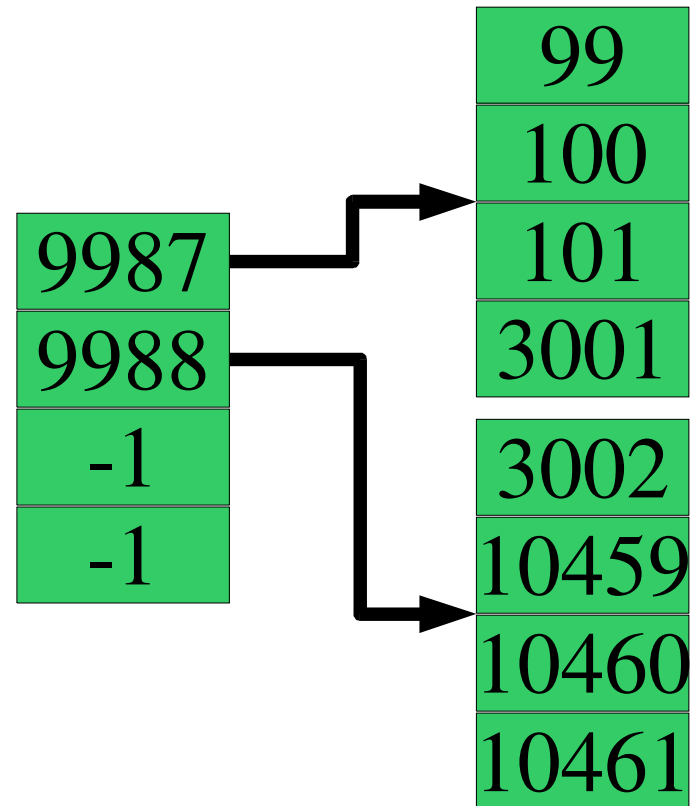


Multi-Level Index Blocks

Index blocks of index
blocks

Does this look familiar?

Allows *big* holes



Unix Index Blocks

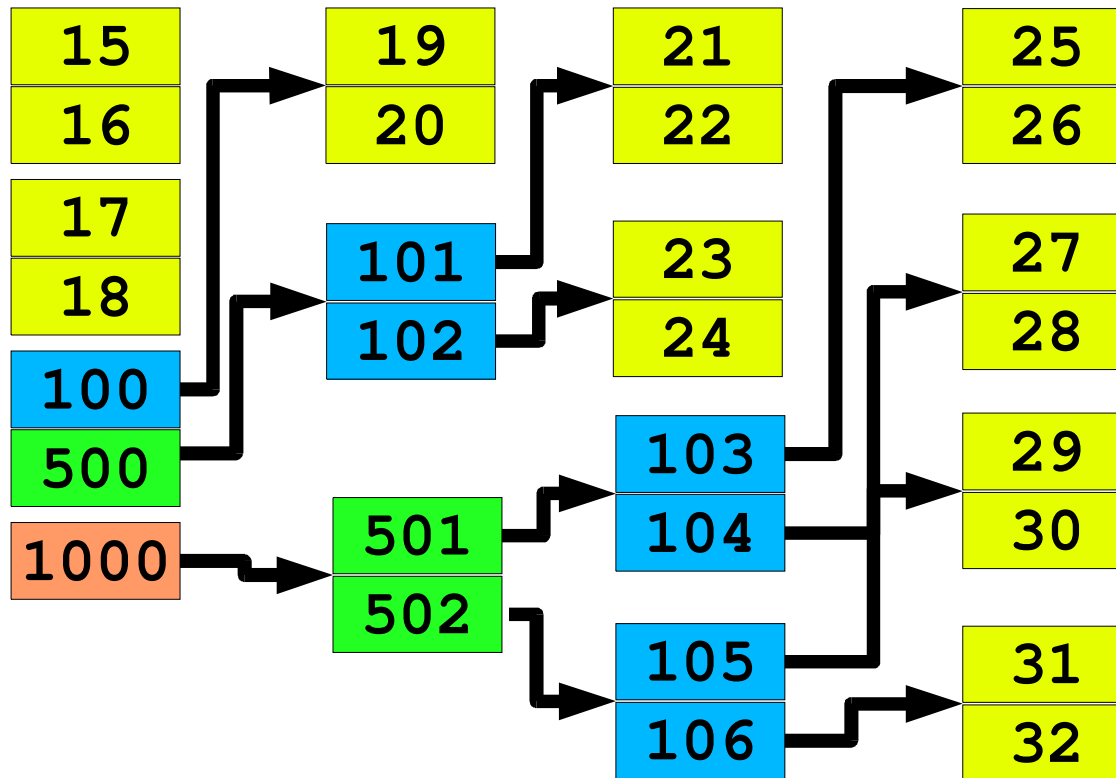
Intuition

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

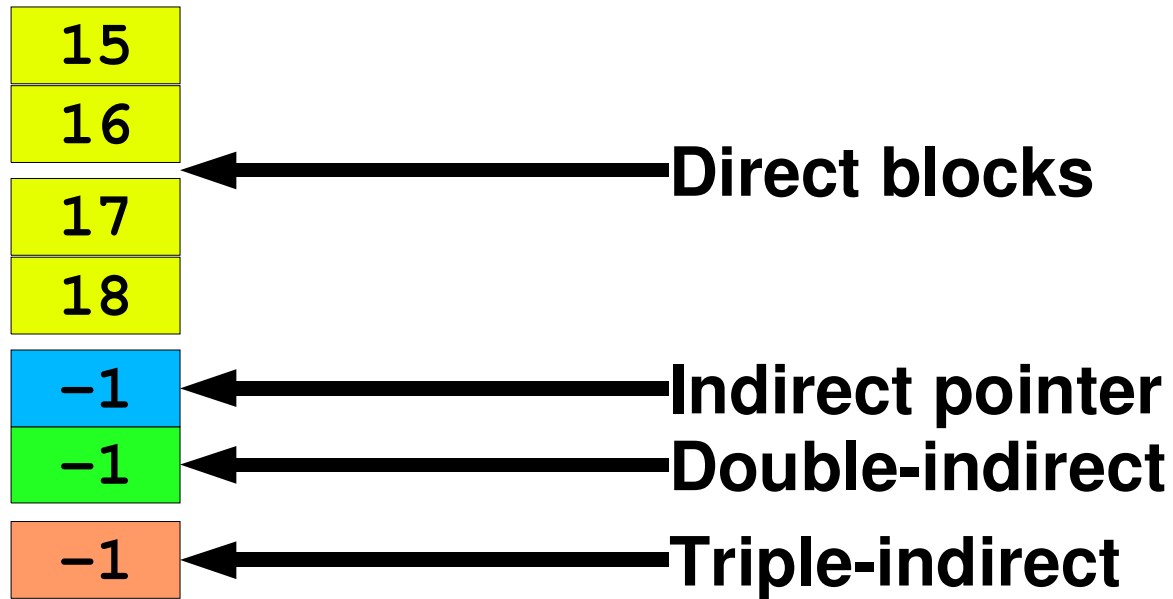
“Clever heuristic” in Unix FFS inode

- 12 (direct) block pointers: $12 * 8 \text{ KB} = 96 \text{ KB}$
 - Availability is “free” - you need inode to open() file anyway
- 3 indirect block pointers
 - single, double, triple

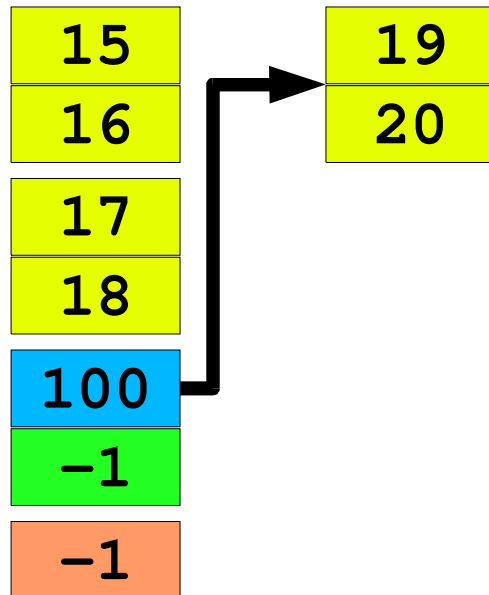
Unix Index Blocks



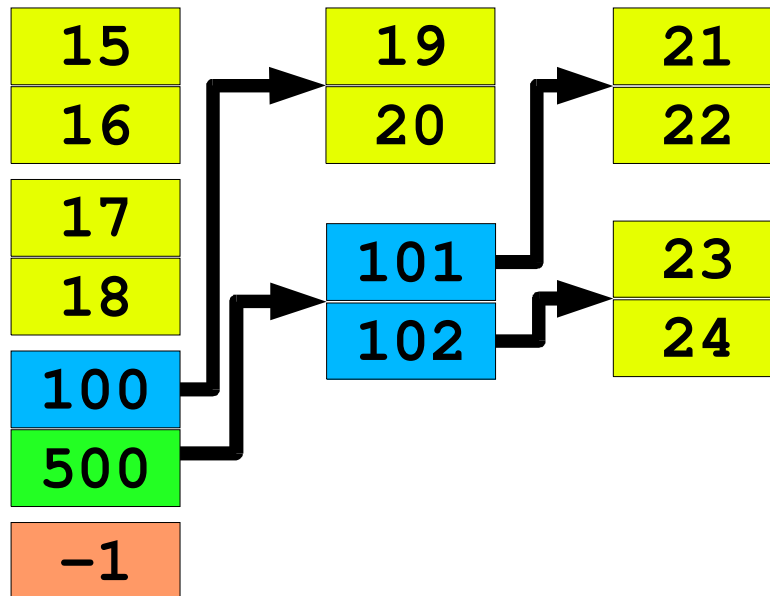
Unix Index Blocks



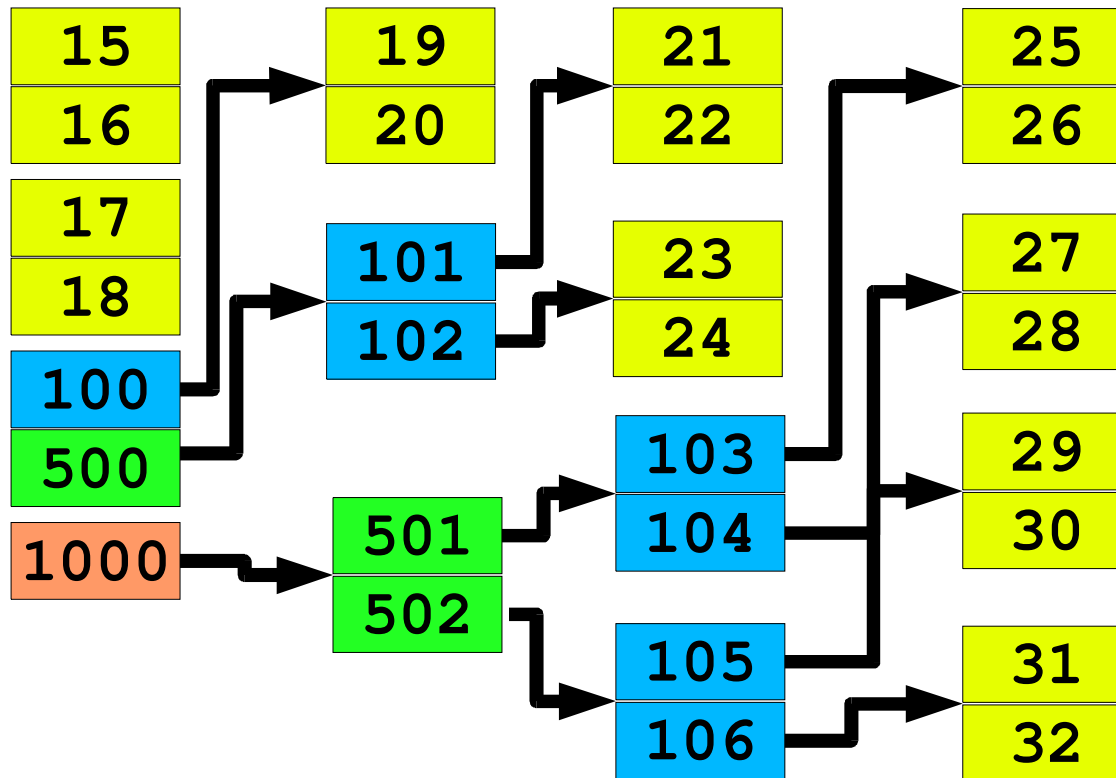
Unix Index Blocks



Unix Index Blocks



Unix Index Blocks



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

Some memory frames back virtual pages

Some memory frames cache file blocks

Would be silly to double-cache vmem pages

- 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

Observation

- How much RAM to devote to each one?
- Why not have just one cache?
 - Mix automatically varies according to load

Cache tricks

Read-ahead

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

- **System observes sequential reads**
 - can pipeline reads to overlap “computation”, read latency

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???

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
- ...