Where is the memory going?
Memory usage in the 2.6 kernel

Sep 2006

Andi Kleen, SUSE Labs
ak@suse.de

Why save memory

Weaker reasons

"I’ve got 1GB of memory. Why should I care about memory?"

- Old machines
  - Not too interesting because they tend to run old software too
- Embedded
  - Also not too interesting because the kernels are heavily tweaked
  - But perhaps they want to do less tweaking
- Leave memory for user space
  - One of the better reasons so far.
  - After all the user wants to run applications, not kernels

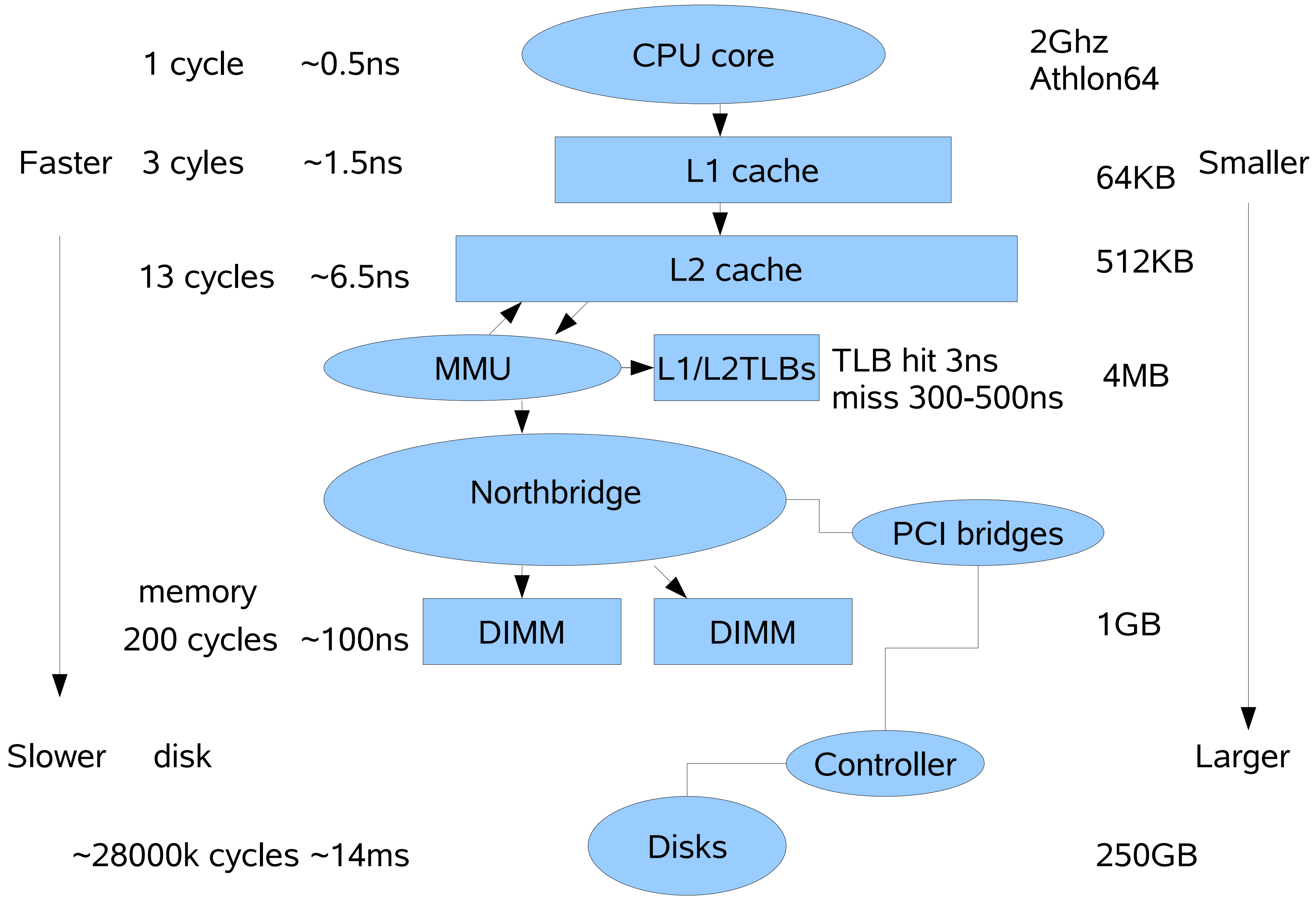
Why save memory

Important reasons

- Scalability
  - Small memory issues often get worse on big systems
    - 1% of 1GB is 10MB, 1% of 100GB is 100MB, 1% of 1TB is 1GB, ...
  - ... a percent here and a percent there ...
  - Causes problems on NUMA systems
    - Some nodes can be nearly filled up by kernel tables
    - Bad performance due to imbalances of traffic
- Virtualization
  - s390 VMs, Xen, vmware, qemu, ...
  - Guests run whole own operating systems
  - Guest systems have limited memory
    - Limits maximum number of VMs per server
    - Shouldn’t or cannot swap guests
    - Main memory limits number of guests
  - 128MB guests are common, 64MB is not unheard of

The most important reason

Smaller is faster!



1 cycle ~0.5ns

CPU core

2Ghz
Athlon64

Faster 3 cyles ~1.5ns

L1 cache

64KB Smaller

13 cycles ~6.5ns

L2 cache

512KB

MMU

L1/L2TLBs

TLB hit 3ns
miss 300-500ns

4MB

Northbridge

PCI bridges

memory
200 cycles ~100ns

DIMM

DIMM

1GB

Slower disk

Controller

Larger

~28000k cycles ~14ms

Disks

250GB

Test setup

- x86-64 Intel Core2 machine with 1GB RAM
- Integrated graphics (8MB frame buffer)
- Running 2.6.18rc4 kernel with some patches for memory measurement
- "Fat" configuration based on defconfig

Measuring kernel memory: dmesg

BIOS 10.05MB (0.98% of total), 980.3MB (95,7%) left after early bot

```
> dmesg
...
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)
 BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
 BIOS-e820: 00000000000e0000 - 0000000000100000 (reserved)
 BIOS-e820: 0000000000100000 - 000000003f5bf000 (usable)
 BIOS-e820: 000000003f5bf000 - 000000003f5cc000 (reserved)
 BIOS-e820: 000000003f5cc000 - 000000003f652000 (usable)
 BIOS-e820: 000000003f652000 - 000000003f6eb000 (ACPI NVS)
 BIOS-e820: 000000003f6eb000 - 000000003f6ef000 (usable)
 BIOS-e820: 000000003f6ef000 - 000000003f6ff000 (ACPI data)
 BIOS-e820: 000000003f6ff000 - 000000003f700000 (usable)
 BIOS-e820: 000000003f700000 - 0000000040000000 (reserved)
 BIOS-e820: 00000000ffe00000 - 0000000100000000 (reserved)
...
On node 0 totalpages: 251483
  DMA zone: 1415 pages, LIFO batch:0
  DMA32 zone: 250068 pages, LIFO batch:31
...
Memory: 1003884k/1039360k available (3384k kernel code, 34360k reserved, 2355k data, 220k init)
```

Measuring kernel code size

6.3MB (6.1%)

Generic x86-64 "defconfig+" kernel 2.6.18-rc4 (+ minor patches)

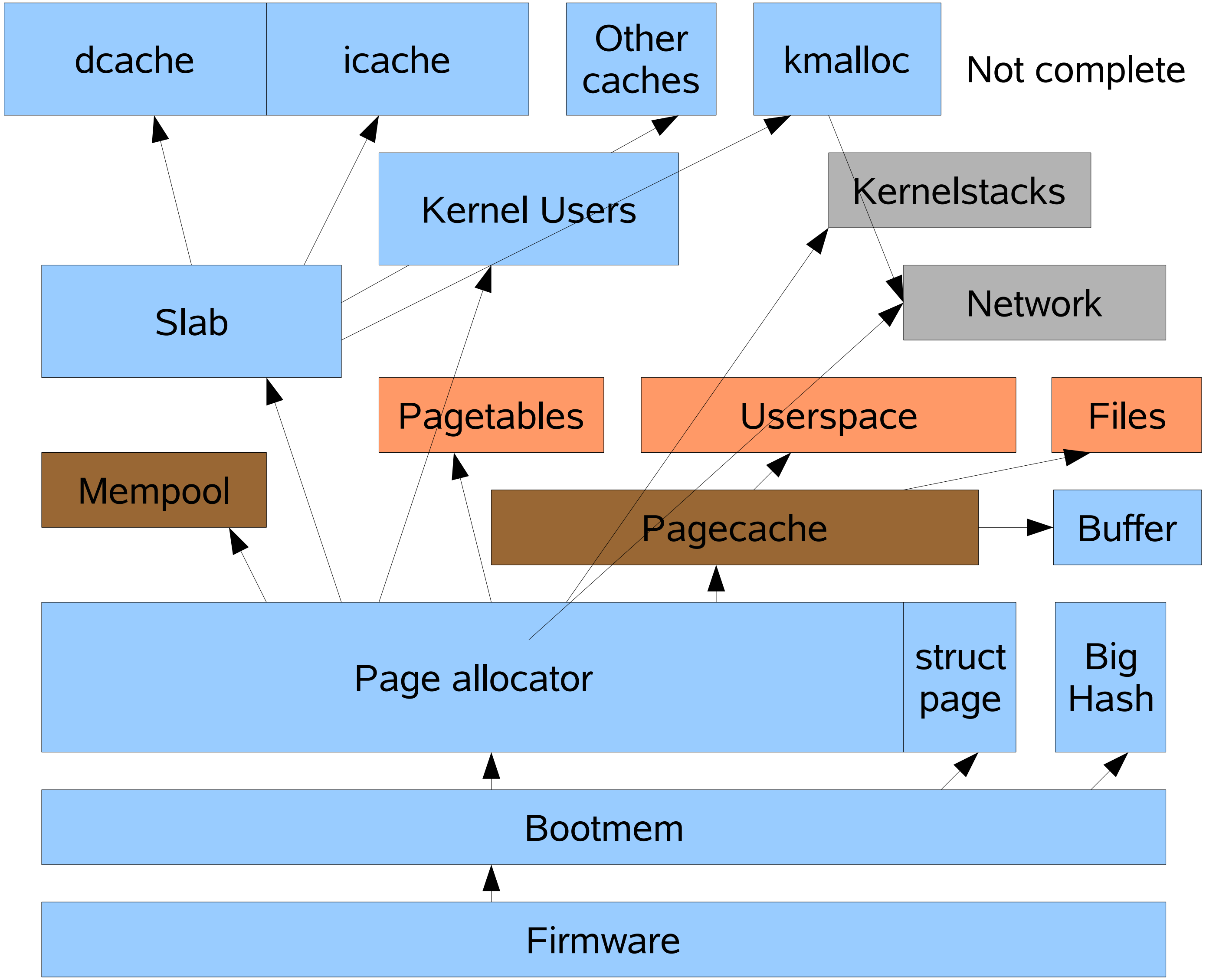
```
> cd /usr/src/linux
> size vmlinux
   text    data     bss     dec     hex filename
4791288 1185948  626328 6603564  64c32c vmlinux
```

Why caring only about code size is bad

Dynamic allocators rule the memory

- Often discussions on kernel bloat focus on code size only
  - Easy to measure with "size vmlinux"
  - Historically trend upwards
    - Actually 2.6 text sizes recently came down
  - Embedded users with flash have some point
    - But for everybody else it is small
    - Percentage larger with small VM guest, but still small
  - 6.1% with "fat" kernel
- Lots of patches to make kernel text smaller
  - Usually by putting in lots of ifdefs
  - Or disabling valuable debugging code that should be enabled by default
- Even with zero byte kernel code you only save 6.1%!
- Dynamic memory is much more important!

Linux memory users



Some allocators

- Bootmem / Early allocator / Firmware
  - Used early in system boot
  - ~43.7MB (~4.26%) lost on test system
  - See paper for details
- Page allocator
  - Main allocator that feeds everybody else
  - Deals in orders of pages (4K on x86)
  - Buddy algorithm
    - All allocations aligned in address/size
  - > Order 0 has fragmentation problems after longer uptime.
- See paper for more allocators

Kernel users

"A megabyte here and a megabyte there and soon we’re talking real memory."

- mem_map / struct page array(s)
  - One entry for each page in the system
  - 1.37% of kernel memory on x86-64 (14.3MB)
  - struct page already quite optimized (32/64bytes)
  - Can be a big problem on large memory 32bit systems
    - But 64bit is fine
    - Sometimes memory holes can be wasteful
    - NUMA/sparsemem can be more efficient with holes
- Page tables
  - Tells the CPU’s MMU about the virtual memory
  - ~8+ bytes per page, ~0.2% of each user mapping
  - SLES10 GNOME+firefox after boot ~5.3MB
  - Shared page tables/large pages might help
    - Automatic large pages would need large VM changes

Kernel users II

- Kernel stacks
  - 8K for each thread in the system (~1MB on test system)
  - Can fail when page allocator is fragmented (order 1)
  - On i386 4K stack option, but dangerous
- Page cache
  - Takes all that kernel leaves over
  - File cache
  - FS metadata
  - User anonymous memory
- Mempools
  - Reserve memory to avoid deadlocks under memory pressure
    - When you need more memory to free memory
  - ~480k (0.04%) on test system
    - Can be much larger on bigger systems
    - Scales with number of block devices etc.
  - Work underway that might allow to eliminate them

The slab allocator

- Main kernel object allocator
  - Memory from page allocator
  - Manages "slab caches" of fixed-size objects
    - Large objects have meta data
  - Can be often majority of kernel memory
  - Performance critical
    - e.g. for networking but many other subsystems too
- Many features
    - NUMA aware
    - per CPU caches
    - cache coloring
- Has object caches that are only freed on demand
  - Intended for "constructed" objects, but nobody uses that

Measuring slab: slabtop

```
> slabtop
 Active / Total Objects (% used)    : 85349 / 88654 (96.3%)
 Active / Total Slabs (% used)      : 12340 / 12340 (100.0%)
 Active / Total Caches (% used)     : 94 / 136 (69.1%)
 Active / Total Size (% used)       : 40022.52K / 40466.88K (98.9%)
 Minimum / Average / Maximum Object : 0.02K / 0.46K / 128.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
 20560  20560 100%    0.24K   1285       16      5140K dentry_cache
 12534  12528  99%    1.35K   6267        2     25068K ext3_inode_cache
  9720   9573  98%    0.09K    243       40       972K buffer_head
  5424   5399  99%    0.08K    113       48       452K sysfs_dir_cache
  5258   5116  97%    0.17K    239       22       956K vm_area_struct
  3815   3802  99%    0.52K    545        7      2180K radix_tree_node
  3548   3540  99%    0.99K    887        4      3548K inode_cache
  3304   3205  97%    0.06K     56       59       224K size-64
  2800   2772  99%    0.03K     25      112       100K size-32
  2410   2295  95%    0.38K    241       10       964K filp
  2065   2011  97%    0.06K     35       59       140K anon_vma
  1740   1737  99%    0.12K     58       30       232K size-128
  1672   1639  98%    0.50K    209        8       836K size-512
  1605   1598  99%    0.25K    107       15       428K size-256
  1395   1395 100%    0.25K     93       15       372K skbuff_head_cache
...
```

More on slab allocator

- kmalloc sits on top and uses power-of-two caches
  - 32bytes ... 128K
- Problems
  - Very complicated code now
  - Unused caches can use a lot of memory
  - Power of two kmalloc slabs often not good fit
  - Freeing not directed at freeing pages
- Rewrite under way now

Interactions

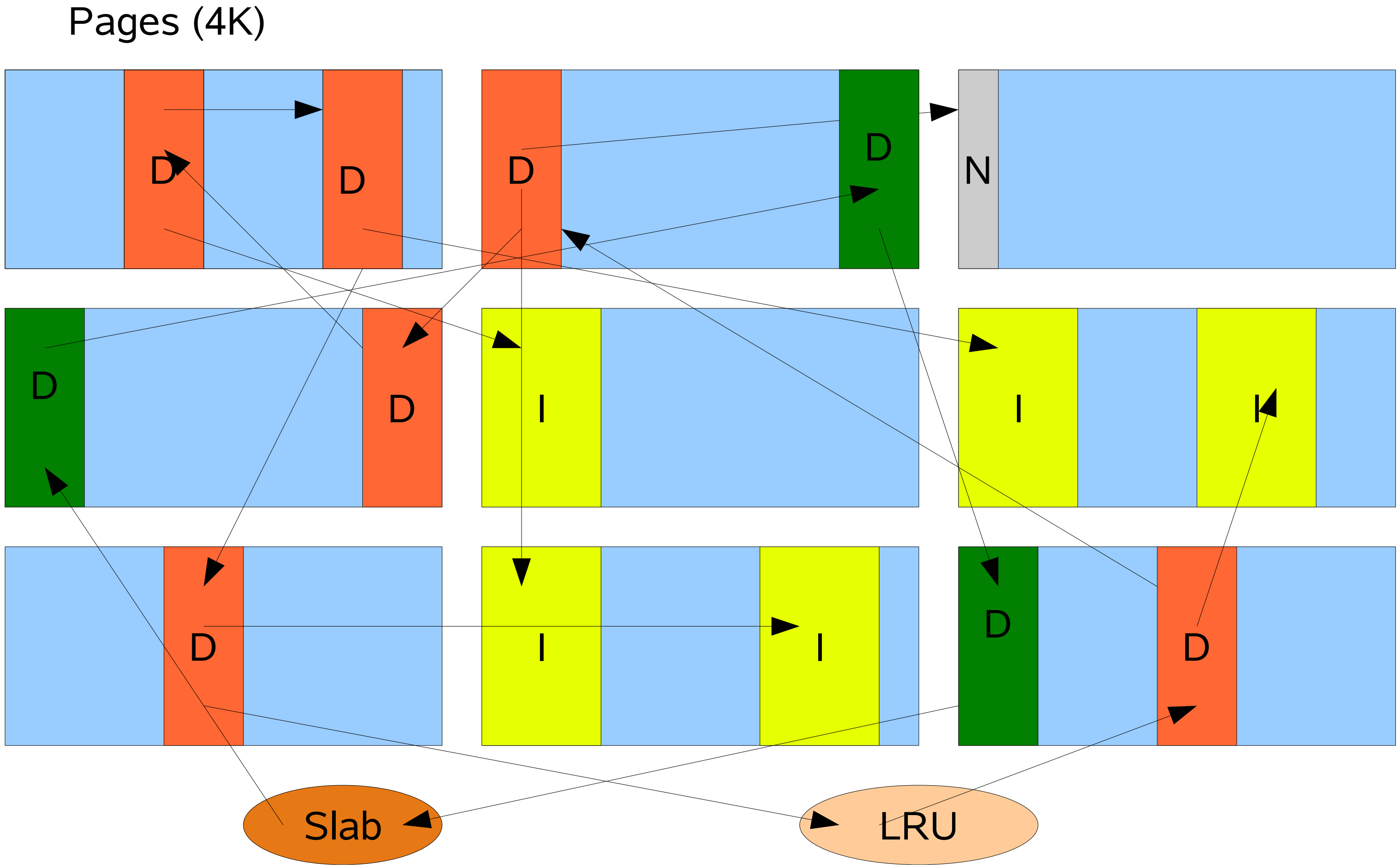
"Free memory is bad memory." - Linus.

- Caches
  - Fill memory
  - Shrink only on demand
  - Free memory isn’t something to look out for
  - Just needs to be freed when needed
- Kernel objects are fixed in memory
  - Cannot be moved, just freed
  - Fragmentation
  - Some objects are "pinned", others cache that could be freed
- Fragmentation
  - Multiple objects in a 4K page
  - Single object can prevent whole page from being freed
  - Even when object is only cache
  - Freers usually have own lists, don’t look at complete pages

The dentry/inode caches

- dentry cache ("dcache") stores directory entries ("names") in memory
  - dentry is primary "handle" for file in kernel
  - fairly large (~200bytes) + file name for names > 36
- Inode cache ("icache") stores inodes in memory
- Linux caches dentries aggressively to give good user experience
  - Only freed on memory pressure
  - Using a LRU list
- Most dentries have a inode object too
  - But separate in memory
  - Much larger (~770bytes)
  - inode cache slave of dcache
  - But separate LRU caches

dcache/icache fragmentation



Hash tables I

```
> dmesg | grep -i hash
PID hash table entries: 4096 (order: 12, 32768 bytes)
Dentry cache hash table entries: 131072 (order: 8, 1048576 bytes)
Inode-cache hash table entries: 65536 (order: 7, 524288 bytes)
Mount-cache hash table entries: 256
IP route cache hash table entries: 32768 (order: 6, 262144 bytes)
TCP established hash table entries: 65536 (order: 9, 3670016 bytes)
TCP bind hash table entries: 32768 (order: 8, 1835008 bytes)
TCP: Hash tables configured (established 65536 bind 32768)
```

Hash tables II

- 4.78MB or 0.46%.
  - Nearly as much as kernel .text!
- Hash tables sized based on memory size
  - Large to make them effectively O(1)
    - But you get the cache misses!
  - Heuristics not very good
  - Hashes sized for worst case workloads
  - Can be tweaked on command line
    - dhash_entries=,ihash_entries=,thash_entries=,rhash_entries=
  - Please benchmark and send feedback!
- Possible solutions:
  - Dynamic hash table growth/shrink
    - Locking tricky
  - Better data structures
    - Various tree variants are looking promising
    - Trees have better cache performance
    - But not O(1) in theory

Summary

- These were just generic examples
- On other workloads kernel users can be quite different
  - But easy to measure
- No easy solution
- But the way to a leaner and faster kernel is to fix inefficient data structures
- Have to work through them one by one
- Needs more work

Wake up! Presentation over.

Paper: http://www.firstfloor.org.org/~andi/memorywaste.pdf

Presentation: http://www.firstfloor.org/~andi/memory.pdf

Or in paper proceedings

Questions?

Thank you!

Backup

Measuring kernel memory: /proc/meminfo

```
MemTotal:     1004104 kB
MemFree:       578576 kB
Buffers:        16436 kB
Cached:        249040 kB
SwapCached:         0 kB
Active:        166312 kB
Inactive:      186184 kB
...
LowTotal:     1004104 kB
LowFree:       578576 kB
SwapTotal:     530104 kB
SwapFree:      530104 kB
Dirty:           2248 kB
Writeback:          0 kB
AnonPages:      86940 kB
Mapped:         37172 kB
Slab:           50008 kB
PageTables:      4932 kB
...
CommitLimit:  1032156 kB
Committed_AS:  194788 kB
```