Where is the memory going? Memory waste under Linux

Andi Kleen, SUSE Labs

August 15, 2006

Abstract

The original Linux 1.0 kernel ran fine on a PC with 4MB memory. Of that the kernel used a small fraction. Later versions weren't as modest in memory requirements. There have also been some complains that the 2.6 kernel needs more memory than the 2.4 kernel.

Often analysis of memory usage has focused on code size only. This paper focuses instead on runtime allocated memory of the kernel. User space memory consumption will not be covered. The paper has an overview of the various memory allocators in Linux and describes some of their problems.

1 Introduction

When Linux kernel memory usage is discussed, the focus is often on kernel text/data size. Most likely that is because is very easy to see – the kernel vmlinux image can be examined with simple tools like nm and tweaked by changing the kernel configuration. This also shows that Linux is growing for each release[1]

But there is a whole other universe in dynamic memory consumption at runtime, which often needs more memory than the kernel code. This dynamic memory can be also watched and analyzed with simple tools and minor changes as this paper will demonstrate.

When memory waste is discussed on hears about old machines and about embedded systems with limited memory. Additionally, memory consumption can be problematic on large systems too, because the more memory is used the less effective are the caches of the CPUs and the more traffic goes over the system interconnect.

Finally, inefficient memory usage in dynamic structure which might be not a big problem on small systems can sometimes explode on very large systems and lead to excessive growth of data structures. For a wide range of machines reducing memory consumption is therefore a good idea. This requires an analysis of where the dynamic memory is going.

This paper shows some simple techniques and results on how to measure dynamic memory consumption in the Linux 2.6 kernel.

[1]Actually in recent 2.6 text sizes are coming down since there is some trend to avoid excessive in-lining.

2 Memory allocators under Linux

We start with an overview over the main memory allocators that are used in the kernel.

2.1 Early memory setup

After the firmware or BIOS has booted the system it tells the operating system in a memory map where the memory is located. On modern x86 system this comes from the e820 BIOS call [2] which is queried at early boot in real-mode and then its output is passed to the kernel.

After it started the kernel works with the saved e820 map and does some really early allocations like the page tables for the kernel direct mapping. While doing this it has to be careful to not overwrite any already used memory like the kernel text. Then the kernel uses the memory information in e820 to initialize the next allocator, which is called the bootmem allocator.

2.2 Bootmem

Originally Linux did all early memory allocation ad-hoc, but when NUMA support with multiple nodes was introduced it turned out to be too complicated. The answer was a very simple minded allocator called bootmem.

Bootmem organizes its memory into NUMA nodes[3] and works in pages. As a simple optimization it can share the last page allocated to multiple users to avoid wasting some memory. Its main use is to allocate the large data structures used by the page allocator described below.

First the already used memory is reserved in bootmem. This includes data structures used by the firmware (ACPI tables, memory mapped hardware, etc.), the actual kernel text, kernel page tables and some other allocations which have been done early.

Bootmem is also used for a few users who need to allocate longer continuous memory areas than are possible with the page allocator below. These users have to run really early to avoid any memory fragmentation from later code in the kernel and have to use bootmem. When bootmem is finished it frees all left over memory to the page allocator, which will be the main allocator for the rest of the run time.

2.3 Page allocator

The main allocator that manages all memory in Linux. It splits the memory into zones and each zone is maintained by a simple buddy allocator.

Buddy is a simple allocator that manages memory areas that are powers-of-two sized. In Linux these power-of-twos are multiples of pages, called orders (so on a 4K page size system the objects are 4K, 8K, 16K, 32K, ..., 128K) The maximum object size that can be allocated is dependent on the page size and also varies per architecture. Normally it is 128K.

[2]On x86 EFI systems like the Apple MacBooks the memory map doesn't come from e820, but from a similar EFI service.

[3]A non NUMA system has only a single node.

In practice this means that any allocation that doesn't fit $2^n * PAGE\_SIZE$ is rounded up to the next power-of-two boundary. The n above is called the order of the allocation. We will evaluate later how much memory this rounding wastes.

A disadvantage of the buddy allocator is fragmentation. Since the power-of-two objects also have to be aligned to power-of-two areas in memory when the system is running for some time it tends to become difficult to allocate objects

The Linux implementation has some support for caching pages locally per CPU and per node as a performance optimization. It also supports memory policies for NUMA optimization[4].

More details on buddy allocators can be found in [3] or in [2]

2.4 Important users of the page allocator

The Page allocator is normally used to allocate page tables for the MMU [5] All the other allocators like slab (except the early ones) also get their memory from the page allocator. The major user is the page cache which is described next.

2.5 Page cache

The page cache manages cached file data and anonymous memory that is used by programs. This includes all program text and data (except for the kernel), file data that has been read or written and file system meta data.

When a program runs all the memory it can see directly, without accessing it indirectly by calling some kernel service, is stored in the page cache.

The state of the page cache for the whole system can be examined using the /proc/meminfo and /proc/zoneinfo files. Counters for individual processes are in /proc/self/stat and /proc/self/statm. The easiest way to examine it is with various front end tools that read these numbers like free, vmstat, ps, top, etc. An overview of the state of the various zones can be also gotten by enabling the sys-req key [6] and pressing Alt-Sysrq-m[7]. This will dump the state of the various zones and how much memory is still free.

2.6 Slab allocator

Slab is a allocator that caches objects that caches objects that are typically smaller than a page. Slab actually supports objects larger than a page too for consistency, but most objects are smaller and slab isn't very good at dealing with large objects. Most objects handled in the kernel are allocated by the slab allocator.

The kernel objects are managed in slab caches. A slab cache contains objects of the same type. It can cache free together and allocated objects. The free objects can be in a "constructed" state to avoid some overhead in repeatedly

[4]For example, on a NUMA system it tries to place the allocation on memory local to the CPU which is requesting the memory.

[5]There can be exceptions: on a 32Bit PAE system the highest page table level is allocated by slab.

[6]See Documentation/sysrq.txt in the kernel source.

[7]This hot key can vary by architecture

initializing objects. The object caches are supported by having optional per cache constructors and destructor call-backs[8]

Slab gets its memory from the page allocator in pages. Normally that's a single page, but can be more than that for larger objects. A continuous memory area managed in a cache is called a slab. A single slab cache can contain multiple slabs, but a slab is never shared between caches.

Slab is a so called zone allocator. Normally objects of the same type have similar live times. A zone allocator tries to cluster objects of the same type together. This in theory avoids fragmentation of the memory map because otherwise a single long lived object can prevent a whole slab from being "pinned" in memory and not freed. We will check how well this theory works in a later section.

Slab also has a couple of optimizations like cache coloring to use caches better in a CPU, per CPU local caches and NUMA support. For more details see the original slab paper [1] The current Linux code is considerable enhanced over this early implementation.

2.7 The Slob mini allocator

These various optimizations unfortunately made the slab code very complex. Since they are not needed on small single processor system the kernel supports an optional alternative mini-slab allocator called slob. Enabling slob requires to recompile the kernel. Most of the savings in slob come from smaller code size, not more efficient dynamic memory usage. Slob is not designed to scale to large memory sizes or multiple CPUs. Slob is not evaluated here.

2.8 The kmalloc slab frontend

kmalloc is a simple allocator that is built on top of slab. kmalloc's interface resembles the ISO-C malloc function. Basically it keeps a number of caches for objects of fixed size. The callers passes kmalloc a length parameter and kmalloc looks for the next biggest cache that fits the length.

kmalloc is used by users in the kernel who need variable amounts of memory (like network packets which don't have a fixed size) or who allocate only very infrequently so that it is not worth it to set up an own slab cache for them. One disadvantage of kmalloc is that it violates the zone allocator rule that all objects in a slab cache should be of the same type and have similar live-times. This can lead to more fragmentation.

The kmalloc caches are all power-of-two sized, starting with 32bytes upto 128KB[9]

2.9 Important slab caches

Typically the most important slab caches are the inode and dentry caches. In some network intensive workloads the slab caches used by memory packets can also use a lot of memory.

[8]This facility is only rarely used in Linux and might be removed at some point

[9]The original slab paper [1] actually mentioned that power-of-twos are a bad choice for this. But Linux for some reason uses it anyways.

A full list of the slab caches in a running system can be seen in /proc/slabinfo. The current activity can be watched using the slabtop [10] too.

2.10 Freeing on demand

Normally Linux tries to use all memory because free memory is useless to the system[11]. This means after the system has run for some time usually nearly all memory is filled up. Linux instead uses the memory for various caches to speed up operation.

However, when the kernel runs low on memory and needs more memory for some operation – like a user program allocating some memory – it has to free some of its caches to give memory back to other users. The actual VM algorithms for this are complicated and beyond the scope of this paper.

2.11 Mempools

Freeing on demand unfortunately has a corner case. Sometimes freeing a page might require more memory allocation. A classical example is to write out and free a dirty NFS page requires allocating multiple network packet objects (sk_buffs). This can lead to a deadlock situation when the system runs out of memory and cannot free memory because doing that that would require even more memory.

Similar problems apply to file systems and block drivers. In general block drivers cannot allocate any memory when they write something because the write could be under memory pressure.

This can be a big problem with very complex block drivers, like a software RAID-5 which might require to read some blocks first to write out parity. Or when when they are reliant on a cluster stack which can be very complex and typically allocates some memory somewhere. This is also the reason why swapping over NFS is not supported. There was recently some work to address this fundamental problem by preventing the kernelin the first time to accumulate too much dirty memory, but, so far the work is still in the prototype stage.

To avoid this for block IO block drivers usually use a special mempool allocator. mempool is a primitive pool that gets its memory from a backing allocator (usually either kmalloc or the page allocator directly). It guarantees that there are always enough objects in the pool for the specific subsystem to do critical operations without dead-lock. To do that mempool always keeps enough free based on a simple estimate from the caller.

The main disadvantage of mempools is that they always keep some memory allocated even when it is not needed – just to prevent others from using it.

3 Evaluation of memory usage

3.1 Test setup

The test system is a EM64T Intel Core2 Duo system with 1GB of memory running a 2.6.18rc4 64bit Linux kernel. It is fresh after bootup sitting in a

[10]The tool is unfortunately kernel version dependent and often has to be updated.

[11]Except for a small emergency reserve for interrupts and some high priority allocation.

SUSE SLES10 gnome desktop with an opened Firefox window and a GNOME terminal. The machine has chipset-integrated graphics which already takes some memory away.

3.2 Examining dynamic memory consumption

The kernel already has a wide number of statistic counters that can be examined in /proc. These are used when possible. Often there are already standard tools to present these counters in a nice human readable format. Examples of this would be top, vmstat, slabtop, free. In other cases custom instrumentation is needed to find out where the memory is going, but that is relatively rare.

3.3 Early booting

The e820 memory map is printed by the kernel at the beginning of the kernel log, so the memory that is used by the firmware can be calculated by examining dmesg output:

```
> dmesg | grep usable |
awk ' { n += strtonum("0x" $4) - strtonum("0x" $2) }
      END { print n/(1024*1024) }'
1013.95
```

This means the firmware or hardware already loses about 10.05MB of the memory[12] or 0.98% of the total memory.

The kernel used for the test was a non modular x86-64 2.6.18-rc4 kernel compiled with the x86-64 default configuration under gcc 4.0 The resulting vmlinux image has the following size:

```
> size vmlinux
   text    data     bss     dec     hex filename
4791288 1185948  626328 6603564  64c32c vmlinux
```

This gives about 6.3MB for the kernel image with its data. This is a pretty quite generic kernel with many drivers and several file systems. A custom compiled kernel will be usually smaller.

However some of that is initdata and init code that is freed after booting:

```
> dmesg | grep Memory
Memory: 1014792k/1039360k available (3211k kernel code, 23488k reserved,
                                     2303k data, 204k init)
```

Minus the 204k of init data this shows that the kernel code and static data use around 6.1% of the total memory. All other memory is dynamic memory.

3.4 Bootmem

The memory line above also shows how much memory comes out of bootmem. The difference of the two numbers is the memory lost to bootmem users as seen by the kernel. In all measurements here we use the total memory (1GB) as base. With that bootmem takes 32.99MB (3.22% of total) away. This includes the kernel text and mem_map array discussed below.

[12]The integrated graphics on the system uses 8MB. This means the BIOS uses 2.05MB.

3.5 Page allocator

3.5.1 Used memory at various times

The total memory used by the page allocator can be dumped with the sysrq-m key combination. This includes memory used by the kernel and memory used by user space, but not memory that has been lost earlier to bootmem or the BIOS.

Here is the memory consumptions at various points of time:

Time	Memory used	Percent
After early boot	34.55MB	3.37%
After initializing all drivers	44.02MB	4.29%
In first boot script	48.55MB	4.74%
In Gnome desktop	372.90MB	36.42%

3.5.2 Overhead of struct page

Each page managed by the page allocator has "struct page" data structure associated with it. These structures are stored in an array [13] called mem_map.

On a 64bit system without any debugging options enabled [14] struct page uses 56 bytes per page. On 32bit each structure is 32bytes. Since they need to be allocated for all pages in the system this gives a memory overhead on a 4K page 64bit x86_64 system of around 1.37% of all memory. On the 32bit 4K page system i386 it is around 0.78%. On the 1GB 64bit test system this is 14.3MB. On systems with larger pages like IA64 (16K) or Alpha (8K) the overhead is smaller, but still considerable (0.34% and 0.68% of all memory)

The struct page overhead cannot be only blamed only the page allocator: the structure is used by many other subsystems too and if the page allocator was the only user struct page could be considerably smaller. However it is its first user so we account it here.

3.5.3 Overhead in the buddy algorithm

As described earlier the buddy algorithm used in the page allocator rounds all allocations to the next power of two of the page size.

Fortunately most allocations tend to be only a single page (user/file data) or two pages (for kernel stacks) which are both powers-of-two. Allocating two pages can be already problematic because they need to be aligned on a two page boundary, which might not be available anymore due to memory fragmentation. With larger allocations the problem becomes worse. When a subsystem needs to allocate memory that doesn't fit into a power-of-two then buddy has to round the length up to the next power-of-two and align it to that boundary too. This can cause large overhead and can also make the allocator unreliable due to memory fragmentation.

For example this can be a considerable problem for network interfaces with large 9k MTUs. The 9K have to be rounded up to 16K and about 7K are wasted per packet. Even worse, the system at some point doesn't have enough free continuous aligned 16K areas anymore and may fail allocations.

[13]Multinode NUMA systems can be actually more efficient here because they have multiple mem_maps and can avoid allocating struct pages for memory holes between nodes.

[14]Especially spinlock debugging which increases the size of struct page considerably.

3.6 Page tables

Page tables don't need custom instrumentation because the kernel already accounts the page tables in a counter in /proc/meminfo. In the test setup, we get 5.3MB memory (0.52%) in page tables when the system is sitting in gnome (without firefox). This is slightly more than the kernel text.

Most of these page tables are in the various shared libraries used by the desktop programs. One possible way to reduce the page tables overhead would be to share not only the code, but also the page tables for shared libraries. There is currently some work going on in this direction.

3.7 Slab allocator in general

The slab allocator already provides plenty of statistics on its behavior in a standard system in /proc/slabinfo and doesn't need much additional instrumentation. /proc/slabinfo actually provides too much information for easy browsing. The easiest way to get a high level overview of the slab allocations is from slabtop -o, which parses /proc/slabinfo.

The most interesting information at any point is how many pages(slabs) are used by slab and how many objects are active. The difference is the waste in slab.

Time	Slabs total	Slabs active	Waste	Total
After Boot	6.04MB	5.84MB	3.4%	0.59%
In Desktop	20.93MB	20.55MB	1.7%	2.04%

At the presentation there will be more detailed data on the slab callers using some systemtap scripts.

3.8 Dentry cache and inode cache

One of the heaviest users of the slab cache are the dentry cache (dcache) and the inode cache. A dentry is a named reference to a file and a inode is the file itself. Usually there is a one to one relation ship between inodes and dentries.

Since each file can be cached this way and a modern system tends to store many files on their hard disks dcache can be quite large. Search tools like find, beagle or locate/update typically walk the complete file system. When this happens, the dcache/inode slabs can take up considerable memory. Using up so much memory can be a problem because it will compete with other possibly more valuable data that could be stored in memory. Users often notice this problem when they see their desktop being unresponsive when it was running over night – the updatedb cronjob walking the full file system has pushed out a lot of memory used by the desktop and replaced it with dentries and inodes.

When the system comes under memory pressure it tries to free unused dentries and inodes. This is unfortunately not always very effective as seen below. Memory pressure is simulated by allocating and touching 900MB of virtual memory in a simple test program ("memhog"). We can then observe the changes in the dcache.

As can be seen the system is not very effective at freeing these caches. The total percentage of the inode and dentry caches only go down by 0.19%.

Time	d/icache	Total
Gnome desktop	7.58MB	0.74%
After kernel compilation	23.66MB	2.31%
After find /	17.94MB	1.75%
After memhog 900M	15.97MB	1.56%

3.9 Mempools

The mempool subsystem was instrumented by adding a new counter to it for its allocated memory and reporting it in /proc/meminfo. After booting up into the desktop the test system has about 0.47 MB in mempools, which isn't a big problem yet.

The memory tied up in mempools scales with the number of block devices. On systems with a lot of block devices – like a system connected to a SAN with a lot of LUNs – this can be considerable memory overhead.

3.10 Kernel stacks

The kernel internally uses a stack for each thread. This can add up because modern systems can have hundreds or thousands of threads running.

```
# ps ax | wc -l
108
```

The kernel stack memory usage was instrumented and yields 1.13MB (0.11% total) in the gnome test setup.

3.11 sysfs

After the desktop has started slabtop tells us that the sysfs directory cache takes 0.41MB (0.04%). It has some more overhead in the dcache, but that is not easily measurable.

4 Problems

4.1 Memory used by firmware on x86 systems

As seen earlier the firmware already eats some memory. On smaller systems this is often used for graphics memory in main memory.

Another problem on systems with 4GB or more of memory is that often the BIOS is not able to remap memory that is below the PCI MMIO hole which is just below the 4GB boundary. When this happens some memory is lost because hardware mappings make it invisible.

4.2 Fragmentation in slab and kmalloc

Data on this will be shown in the presentation.

An obvious improvement would be to not use power of two kmalloc slabs, but more customized sizes based on some profiling of real allocation sizes under various workloads.

4.3 Slab reaping under memory pressure

When the system needs more memory slab tries to free the cached but not active objects in the slab caches. First slab will free all the various per CPU and per node caches. Then the slab raper goes through all the slabs in the caches and checks if a particular slab only contains free cached objects. When fully free the slab is given back to the page allocator.

For some important caches like the dentry cache, who rarely have free objects, slab cannot free much memory by itself. The system asks the higher level user – the dcache and inode cache – to free some objects. Dcache maintains its dentries with a simple Least-Recently-Used (LRU) lists and starts to free the least used objects. The dcache reaper will continue freeing objects until there is enough memory free again.

But no memory is actually freed until a complete dcache slab doesn't have any active objects anymore – after all slab can only return complete pages to the page allocator. But the order in which the dcache frees its object has no direct relationship to their position in the slabs. There are often cases when the dcache reaper has to go through most of its cached list before it can actually make enough slabs freeable. When this happens a large percentage of the useful dentries and inodes will be lost and the user will later have to wait longer again when accessing the file system again.

A secondary effect is that dentries in the dcache have references to inodes. Usually the inode is only freed after its corresponding dentry[15]. This means that dcache can actually tie up even more memory because it controls the slabs of two slab caches.

An obvious improvement would be a more direct freeing strategy for dcache and inode cache that actually takes the slabs in account and tries to first free all objects in a slab before freeing other objects.

5 Hash tables

Various subsystems in the kernel use hash tables to manage their objects. When the hash tables are performance critical these are scaled by memory size. This can give some interesting effects.

```
> 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: 131072 (order: 9, 2097152 bytes)
TCP bind hash table entries: 65536 (order: 8, 1048576 bytes)
TCP: Hash tables configured (established 131072 bind 65536)
```

Adding up this memory gives 4.78MB or 0.46% of total memory. This means on this 1GB machine the hash tables need nearly as much memory as the kernel text.

[15] Or multiple dentries in case of hardlinks.

The author has actually seen boot logs of multi-TB machines which ended up with with several GB of memory tied up in various hash tables. Clearly that could be improved. Any improvement in the memory usage of the hash tables will save considerable memory with less effort than any tuning to the kernel text size or configuration.

6 Conclusion

We have shown that there are many users in the kernel that each need significantly more memory than the kernel text. Future work in reducing memory consumption on Linux should focus on these users instead of only code size.

References

[1] Jeff Bonwick, An Object-Caching Kernel Memory Allocator, Usenix Summer Conference 1994

[2] Donald Knuth, The Art of Computer Programming Volume 1: Fundamental Algorithms, Addison-Wesley

[3] Wikipedia, http://en.wikipedia.org/wiki/Buddy_memory_allocation