The SLUB allocator

[Posted April 11, 2007 by corbet]

The slab allocator has been at the core of the kernel's memory management for many years. This allocator (sitting on top of the low-level page allocator) manages caches of objects of a specific size, allowing for fast and space-efficient allocations. Kernel hackers tend not to wander into the slab code because it's complex and because, for the most part, it works quite well.

Christoph Lameter is one of those people for whom the slab allocator does not work quite so well. Over time, he has come up with a list of complaints that is getting impressively long. The slab allocator maintains a number of queues of objects; these queues can make allocation fast but they also add quite a bit of complexity. Beyond that, the storage overhead tends to grow with the size of the system:

> SLAB Object queues exist per node, per CPU. The alien cache queue even has a queue array that contain a queue for each processor on each node. For very large systems the number of queues and the number of objects that may be caught in those queues grows exponentially. On our systems with 1k nodes / processors we have several gigabytes just tied up for storing references to objects for those queues This does not include the objects that could be on those queues. One fears that the whole memory of the machine could one day be consumed by those queues.

Beyond that, each slab (a group of one or more continuous pages from which objects are allocated) contains a chunk of metadata at the beginning which makes alignment of objects harder. The code for cleaning up caches when memory gets tight adds another level of complexity. And so on.

Christoph's response is the SLUB allocator, a drop-in replacement for the slab code. SLUB promises better performance and scalability by dropping most of the queues and related overhead and simplifying the slab structure in general, while retaining the current slab allocator interface.

In the SLUB allocator, a slab is simply a group of one or more pages neatly packed with objects of a given size. There is no metadata within the slab itself, with the exception that free objects are formed into a simple linked list. When an allocation request is made, the first free object is located, removed from the list, and returned to the caller.

Given the lack of per-slab metadata, one might well wonder just how that first free object is found. The answer is that the SLUB allocator stuffs the relevant information into the system memory map - the `page` structures associated with the pages which make up the slab. Making `struct page` larger is frowned upon in a big way, so the SLUB allocator makes this complicated structure even more so with the addition of another union. The end result is that `struct page` gets three new fields which only have meaning when the associated page is part of a slab:

```
void *freelist;
short unsigned int inuse;
short unsigned int offset;
```

For slab use, `freelist` points to the first free object within a slab, `inuse` is the number of objects which have been allocated from the slab, and `offset` tells the allocator where to find the pointer to the next free object. The SLUB

allocator can use RCU to free objects, but, to do so, it must be able to put the "next object" pointer outside of the object itself; the `offset` pointer is the allocator's way of tracking where that pointer was put.

When a slab is first created by the allocator, it has no objects allocated from it. Once an object has been allocated, it becomes a "partial" slab which is stored on a list in the `kmem_cache` structure. Since this is a patch aimed at scalability, there is, in fact, one "partial" list for each NUMA node on the system. The allocator tries to keep allocations node-local, but it will reach across nodes before filling the system with partial slabs.

There is also a per-CPU array of active slabs, intended to prevent cache line bouncing even within a NUMA node. There is a special thread which runs (via a workqueue) which monitors the usage of per-CPU slabs; if a per-CPU slab is not being used, it gets put back onto the partial list for use by other processors.

If all objects within a slab are allocated, the allocator simply forgets about the slab altogether. Once an object in a full slab is freed, the allocator can relocate the containing slab via the system memory map and put it back onto the appropriate partial list. If all of the objects within a given slab (as tracked by the `inuse` counter) are freed, the entire slab is given back to the page allocator for reuse.

One interesting feature of the SLUB allocator is that it can combine slabs with similar object sizes and parameters. The result is fewer slab caches in the system (a 50% reduction is claimed), better locality of slab allocations, and less fragmentation of slab memory. The patch does note:

> Note that merging can expose heretofore unknown bugs in the kernel because corrupted objects may now be placed differently and corrupt differing neighboring objects. Enable sanity checks to find those.

Causing bugs to stand out is generally considered to be a good thing, but wider use of the SLUB allocator could lead to some quirky behavior until those new bugs are stamped out.

Wider use may be in the cards: the SLUB allocator is in the -mm tree now and could hit the mainline as soon as 2.6.22. The simplified code is attractive, as is the claimed 5-10% performance increase. If merged, SLUB is likely to coexist with the current slab allocator (and the SLOB allocator intended for small systems) for some time. In the longer term, the current slab code may be approaching the end of its life.

Index entries for this article

Kernel Memory management/Slab allocators
Kernel Slab allocator

(Log in to post comments)

The SLUB allocator

Posted Apr 12, 2007 18:29 UTC (Thu) by **rwmj** (subscriber, #5474) [Link]

I'm yet again left thinking what would happen if the kernel devs actually thought about proper garbage collection.

Rich.

Reply to this comment

The SLUB allocator

Posted Apr 12, 2007 21:43 UTC (Thu) by **flewellyn** (subscriber, #5047) [Link]

SSHHHHH! You can't say that word around C programmers! They get all twitchy and start to yell incoherently.

Reply to this comment

The SLUB allocator

Posted Apr 12, 2007 23:57 UTC (Thu) by **rwmj** (subscriber, #5474) [Link]

Yeah .. funny, but that actually happened, recently.

Rich.

Reply to this comment

The SLUB allocator

Posted Apr 12, 2007 23:58 UTC (Thu) by **bronson** (subscriber, #4806) [Link]

What kind? The kernel already has reference counting. If you're advocating mark & sweep... Well, go ahead and give it a shot! It might be a good master's thesis, but I'm not sure it would ever be better than the slightly buggy manual mess that we have now. :)

Reply to this comment

The SLUB allocator

Posted Apr 13, 2007 9:36 UTC (Fri) by **nix** (subscriber, #2304) [Link]

It's already there: net/unix/garbage.c.

(You *need* proper garbage collection handling cycles if you implement AF_UNIX sockets with fd passing, or any bugger can DoS you grotesquely.)

Reply to this comment

The SLUB allocator

Posted Apr 19, 2007 10:20 UTC (Thu) by **jfj** (guest, #37917) [Link]

One problem with GC is that resources are not freed As Soon As Possible. For memory that may be a win, but there are other things: File descriptors, inodes, locks, etc. These things Must be freed as soon as possible, and no you can't do that periodically every time GC is triggered. So since the code is there, it is also better to free the memory as well.

I mean, why do: close the file descriptor, but leave the memory for the GC.

GC collection would then have to traverse the entire object space and kill all the caches for good! Just free the damn thing when you're there.

Reply to this comment

The SLUB allocator

Posted Jul 5, 2010 11:14 UTC (Mon) by **marcH** (subscriber, #57642) [Link]

You just got me thinking... is there any language giving you both? I mean, sometimes GC is best. Some other times in the very same piece of sofware, "Just free the damn thing when you're there" is best.

So can I have both please?

Reply to this comment

The SLUB allocator

Posted Jul 5, 2010 14:49 UTC (Mon) by **nye** (guest, #51576) [Link]

D seems like a good choice, in many ways.

Reply to this comment

The SLUB allocator

Posted Apr 22, 2007 22:50 UTC (Sun) by **RareCactus** (guest, #41198) [Link]

> I'm yet again left thinking what would happen
> if the kernel devs actually thought about proper
> garbage collection.
> Rich.

Things that are good practice for sysadmins writing perl scripts or programmers writing Java or COBOL "business logic" are not good practice for kernel hackers.

Garbage collection is about managing memory lazily. It will always hold on to memory longer than equivalent non-GC code would. It will trash the data cache by rummaging through things unecessarily.

Traditional garbage collection algorithms like mark and sweep and generational GC tend to require the system to come to a "stop" while the garbage collector thread scans through everything. It should be obvious to anyone that this is unacceptable for a kernel, especially on a multi-core system. There are some newer algorithms that mitigate this behavior somewhat, at the cost of even worse performance.

The kernel is about efficiency. Yes, even in 2007. It's about writing code that everyone will use. It's about spending 2x the time to get a 10% speedup. It's about scaling up to 4, 8, 16 CPUs, and down to a 50 MHz antique.

The kernel manages many other resources besides memory: things like network connections, file descriptors, IRQs, hardware stuff. malloc() might make your head spin, but it's not as hard to write as the TCP stack.

Userspace can and should move more and more to high-level languages with garbage collection, first-class functions,

the whole nine yards. I like Ruby, myself. And there are certain resources in the kernel that might benefit from a GC-like approach. But memory is not one of them.

R/C

Reply to this comment

The SLUB allocator

Posted Jan 12, 2009 9:52 UTC (Mon) by **Blaisorblade** (guest, #25465) [Link]

There has been discussion about GC being faster than general purpose memory allocators, and I tend to like that idea and agree with it. Also, I know that most people complaining against Java are not really able to do better than a GC and write leak-free C code, as they assume they can (and by "most people" I already exclude anybody reading LWN - I'm thinking to most lamers or CS students).

But it doesn't apply in kernel space, apart from existing usage of manual refcounting, for several reasons:

- GCs depend on wasting memory space. A copying GC needs twice the space actually used by the application, by definition (one would use it just for the young generation of a generational GC, though). Refcounting uses exactly the needed space, no more, no less.
- a recent paper argued that a GC needs to use 5x the used memory to be faster than manual allocations, so that the cost of a scavenging (which is always big, even with generations) is paid infrequently enough. That's not an option in kernel.
- incremental/real-time GC (which do not stop execution) are much slower if fine tuned, and extremely hard to get right.
- kernel devs are extremely concerned about cache impact, even because they'd like to leave the caches for app usage

as much as possible. And any GC which is not refcounting is notoriously bad about locality.
- when you work in a dynamic language and already have a VM, you might as well implement a GC, but here there is no VM whatsoever.
- All that is with general purpose memory allocators. The kernel makes extensive use of the SLAB/SLUB interface instead of kmalloc(), and that allocates fixed-size objects.

Even kmalloc is implemented on top of that: allocation sizes are rounded up mostly to the next power of 2, with a few exceptions. A 700-byte object will actually use 1024 bytes. See include/linux/kmalloc_sizes.h for the actual existing sizes.

- All the discussion argues about automatic refcounting. That means incrementing refcounts on each function call because you are pushing parameters on the operand stack. No kidding - CPython is like that. Multithreaded CPython was 2x slower because of atomic refcounts and locks around name lookups, and that was CPython 1.5.2 IIRC. Nowadays the rest is more optimized (even if it still hugely sucks), so the same slowdown (for the same number of refcounts) would have a given impact.

Now, the kernel obviously does not use refcounting like that. If they can live without recursive locks, they can also avoid incrementing a refcount in a thread which already owns a reference.

Reply to this comment