Overview of Linux Memory Management Concepts: Slabs

Jim Blakey

Note: In the Linux 2.6 kernels, a new cache manager called the slub allocator is available and may replace the slab allocator described here. Either can be used through kernel configuration parameters. This paper was written a while ago and does not contain the most current kernel information. I will write an update soon - jimb

Introduction to slabs and kernel cache

Kernel modules and drivers often need to allocate temporary storage for non-persistent structures and objects, such as inodes, task structures, and device structures. These objects are uniform in size and are allocated and released many times during the life of the kernel. In earlier Unix and Linux implementations, the usual mechanisms for creating and releasing these objects were the **`kmalloc()`** and **`kfree()`** kernel calls.

However, these used an allocation scheme that was optimized for allocating and releasing pages in multiples of the hardware page size. For the small transient objects often required by the kernel and drivers, these page allocation routines were horribly inefficient, leaving the individual kernel modules and drivers responsible for optimizing their own memory usage.

One solution was to create a global kernel caching allocator which manages individual caches of identical objects on behalf of kernel modules and drivers. Each module or driver can ask the kernel cache allocator to create a private cache of a specific *object type*. The cache allocator handles growing each cache as needed on behalf of the module, and more importantly, the cache allocator can release unused pages back to the free pool in times when memory is in a crunch. The cache allocator works with the rest of the memory system to maintain a balance between the memory needs of each driver or module and the system as a whole.

The Linux 2.4 kernel implements a caching memory allocator to hold caches (called *slabs*) of identical objects. This slab allocator is basically an implementation of the "Slab Allocator" as described in *UNIX Internals: The New Frontiers by Uresh Vahalia, Prentice Hall, ISBN 0-13-101908-2*, with further tweaks based on a *The Slab Allocator: An Object-Caching Kernel Memory Allocator, Jeff Bonwick (Sun Microsystems).* presented at: USENIX Summer 1994 Technical Conference.

The following is a partial list of caches maintained by the Slab Allocator on a typical Linux 2.4 system. This information comes from the /proc/slabinfo file. Note that most kernel modules have requested their own caches. The columns are cache name, active objects, total number of objects, object size, number of full or partial pages, total allocated pages, and pages per slab.

```
slabinfo - version: 1.1
kmem_cache            59     78    100    2    2    1
ip_fib_hash           10    113     32    1    1    1
ip_conntrack           0      0    384    0    0    1
urb_priv               0      0     64    0    0    1
clip_arp_cache         0      0    128    0    0    1
ip_mrt_cache           0      0     96    0    0    1
tcp_tw_bucket          0     30    128    0    1    1
```

```
tcp_bind_bucket        5    113     32    1    1    1
tcp_open_request       0      0     96    0    0    1
inet_peer_cache        0      0     64    0    0    1
ip_dst_cache          23    100    192    5    5    1
arp_cache              2     30    128    1    1    1
blkdev_requests      256    520     96    7   13    1
dnotify cache          0      0     20    0    0    1
file lock cache        2     42     92    1    1    1
fasync cache           1    202     16    1    1    1
uid_cache              4    113     32    1    1    1
skbuff_head_cache     93     96    160    4    4    1
sock                 115    126   1280   40   42    1
sigqueue               0     29    132    0    1    1
cdev_cache           156    177     64    3    3    1
bdev_cache            69    118     64    2    2    1
mnt_cache             13     40     96    1    1    1
inode_cache         5561   5580    416  619  620    1
dentry_cache        7599   7620    128  254  254    1
dquot                  0      0    128    0    0    1
filp                1249   1280     96   32   32    1
names_cache            0      8   4096    0    8    1
buffer_head        15303  16920     96  422  423    1
mm_struct             47     72    160    2    3    1
vm_area_struct      1954   2183     64   34   37    1
fs_cache              46     59     64    1    1    1
files_cache           46     54    416    6    6    1

                <snip>
```

Figure 1: Typical list of caches maintained by kernel

Slab Overview

A *slab* is a set of one or more contiguous pages of memory set aside by the slab allocator for an individual cache. This memory is further divided into equal segments the size of the object type that the cache is managing.

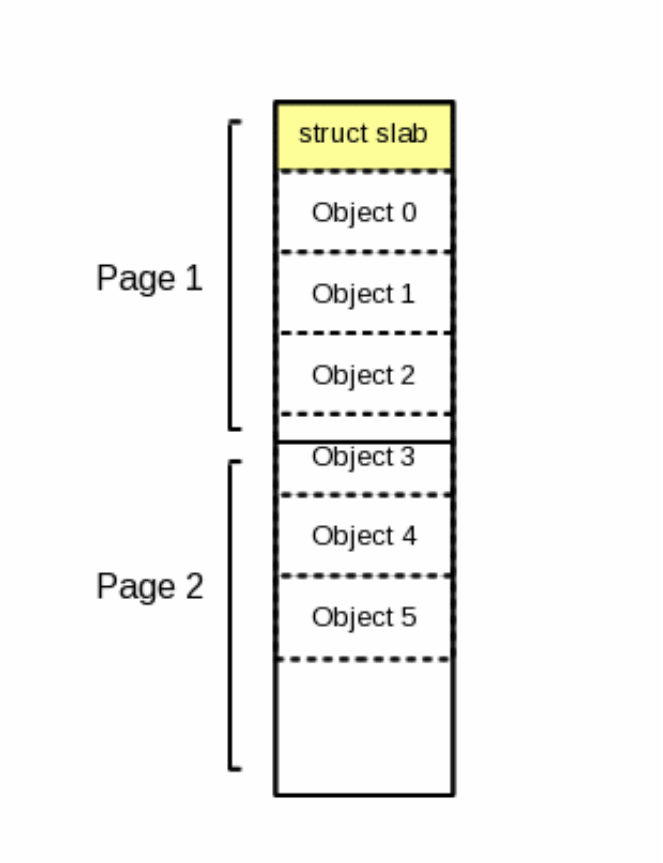

Figure 2: Two page slab with 6 objects

As an example, assume a file-system driver wishes to create a cache of *inodes* that it can pull from. Through the **`kmem_cache_create()`** call, the slab allocator will calculate the optimal number of pages (in powers of 2) required for each slab given the *inode* size and other parameters. A **`kmem_cache_t`** pointer to this new inode cache is returned to the file-system driver.

When the file-system driver needs a new *inode*, it calls **`kmem_cache_alloc()`** with the **`kmem_cache_t`** pointer. The slab allocator will attempt to find a free *inode* object within the slabs currently allocated to that cache. If there are no free objects, or no slabs, then the slab allocator will grow the cache by fetching a new slab from the free page memory and returning an *inode* object from that.

When the file-system driver is finished with the *inode* object, it calls **`kmem_cache_free()`** to release the *inode*. The slab allocator will then mark that object within the slab as free and available.

If all objects within a slab are free, the pages that make up the slab are available to be returned to the free page pool if memory becomes tight. If more *inodes* are required at a later time, the slab allocator will re-grow the cache by fetching more slabs from free page memory. All of this is completely transparent to the file-system driver.

Creation and management of slab pages

Each slab of pages has an associated slab management structure. The slab_t struct is defined in /usr/src/linux/mm/slab.c has the following format:

```
typedef struct slab_s {
        struct list_head        list;
        unsigned long           colouroff;
        void                    *s_mem;     /* including colour offset */
        unsigned int            inuse;      /* num of objs active in slab */
        kmem_bufctl_t           free;
} slab_t;
```

Where:

- `list` is a generic linked list structure found in /usr/include/linux/list.h. This type of list structure is used through out the Linux kernel. It contains a `prev` and `next` which are used to track of where this slab is being used. The various lists the slab_t can be on is described in the next section.
- `colouroff` is an offset within the slab where the slab_t and allocated *objects* begin. This is part of the cache coloring described a little later.
- `s_mem` is a pointer to the first object in the slab. The cache objects are contiguous from this point. Any object in the slab can be referenced by `(object number * object size) + s_mem`
- `inuse` is a counter of the number of objects currently in use.
- `free` is an integer index of the current next free object within the slab.

The slab_t structure is immediately followed by an array of `kmem_bufctl_ts`. There is one of these for for each object in the slab. The purpose of the kmem_bufctl_t array is to keep track of allocated and free objects within the slab. Implementations of slab allocators often have these structs containing forward/backward pointers, reference counts, pointers to the slabs they're in, etc. However, the Linux 2.4 implementation of virtual memory uses the `page struct` to keep track of which caches and slabs cache objects belong to, so the kmem_bufctl_t can be very simple. In fact, it is a single integer index. This array, along with the `slab_t>-free` member in effect implement the cache heap.

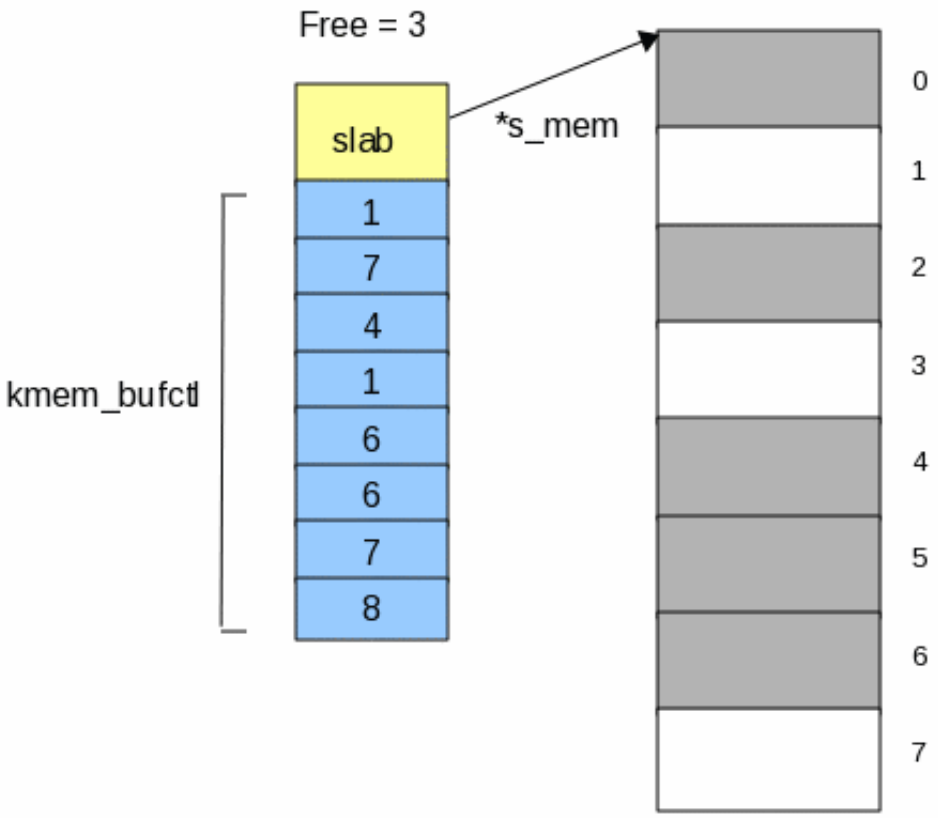

Figure 3. Off slab slab_t structure with 8 kmem_bufctl_t structs

The above figure shows the `bufctl` in use after several slab objects have been allocated and deleted. The next `free` object to be allocated will be in slot 3. Then `free` will assume the value in slot 3's `bufctl` or the value 1. So slot 1 will get the following next object allocated. Etc.

A `slab_t` structure may reside in the slab it describes or it may be allocated as part of the kmem_cache cache, depending on the size of the objects in the slab. Figure 2 shows an on-slab `slab_t` structure, where figure 3 shows on off-slab `slab_t`. The assumption is that if the objects in the cache are large, it is better to place the `slab_t` off-slab for less fragmentation and better packing of the objects. The rule is if the object size is greater than 1/8th the page size, the `slab_t` is allocated off-slab

The number of *objects* per slab (and therefore the number of pages per slab) is calculated when the cache is initially created. The driver or kernel module that asks for the cache only supplies the size of the *objects* to be cached, and should never have to

care about how many objects there are in the cache.

The basic algorithm for calculating the number of objects in a slab is: First see how many objects fit on one page. If the left over is less than 1/8th the total slab size, then the fragmentation (or wastage) is acceptable and we're done.

If there is too much left over empty space on the first try, we try again with a larger slab. Slabs are grown in multiples of powers of 2 of the page size (kept internally as *gfporder*). So the first try, *gfporder* is 0 for 1 page, second try *gfporder* is 1 for 2 pages, next try gets 4 pages, next gets 8 pages, etc.

The actual algorithm used is obviously a little more complex than what I've just described. There are a lot of other factors and limits to take into account, such as whether the `slab_t` is kept on the slab (so its size figures into the equation), cache coloring, L1 cache alignment of the objects to reduce cache hits, etc. But the general idea is to find a balance of the number of pages in the slab and the number of objects in the slab for most efficient use of the space available.

The cache

A cache is a group of one or more slabs of *object type*. The structure that maintains each cache is the `kmem_cache_t`. The following is a partial list of important fields within the `kmem_cache_t` structure. This is not a complete structure, as all SMP and statistics related fields were removed for brevity. The full structure is defined in /usr/src/linux/mm/slab.c.

```
struct kmem_cache_s {
        struct list_head        slabs_full;
        struct list_head        slabs_partial;
        struct list_head        slabs_free;
        unsigned int            objsize;
        unsigned int            flags;  /* constant flags */
        unsigned int            num;    /* # of objs per slab */
        spinlock_t              spinlock;
        ...
        unsigned int            gfporder;
        unsigned int            gfpflags;
        size_t                  colour;         /* cache colouring range */
        unsigned int            colour_off;     /* colour offset */
        unsigned int            colour_next;    /* cache colouring */
        ...
        kmem_cache_t            *slabp_cache;
        unsigned int            growing;
        ...
        void (*ctor)(void *, kmem_cache_t *, unsigned long);
        void (*dtor)(void *, kmem_cache_t *, unsigned long);
        char                    name[CACHE_NAMELEN];
        struct list_head        next;
        ...
};
```

Where: Where:

- `slabs_full, slabs_partial,`and `slabs_free` are lists of slabs associated with this cache.
- `objsize` is the size of the objects contained within the cache.
- `num` is the number of cache objects per slab. This is calculated when the cache is created as a function of the size of each object and the best fit for a given number of memory pages per slab.

- `gfporder` is the number of pages per slab as a power of two. For example, for 1 page/slab, gfporder is 0, for 2 pages/slab, gfporder is 1, for 4 pages, 2, for 8 pages, 3, etc... Pages are always allocated in multiples of powers of two.
- `colour, colour_off` and `colour_next` are used to maintain the cache coloring offset for each slab. This will be discussed later.
- `*slabp_cache` is a pointer to the kernel cache that is used for the `slab_t`, *if* the `slab_t` is maintained off-slab.
- `growing` is a flag that the cache is growing so that the memory pages won't be de-allocated.
- `ctor` and `dtor` are driver callback routines. These are constructor and destructor routines that the driver can supply that will be called when an object is allocated or released. This allows a driver to have a custom initialization or validation routine on object allocation, or to perform any cleanup before an object is released. These are also very useful for driver debugging.
- `name` is a string describing the cache. This will be printed as part of the /proc/slabinfo file. See Slab Figure 1.
- `next` is a list_struct pointer to the next cache in the kernel cache chain.

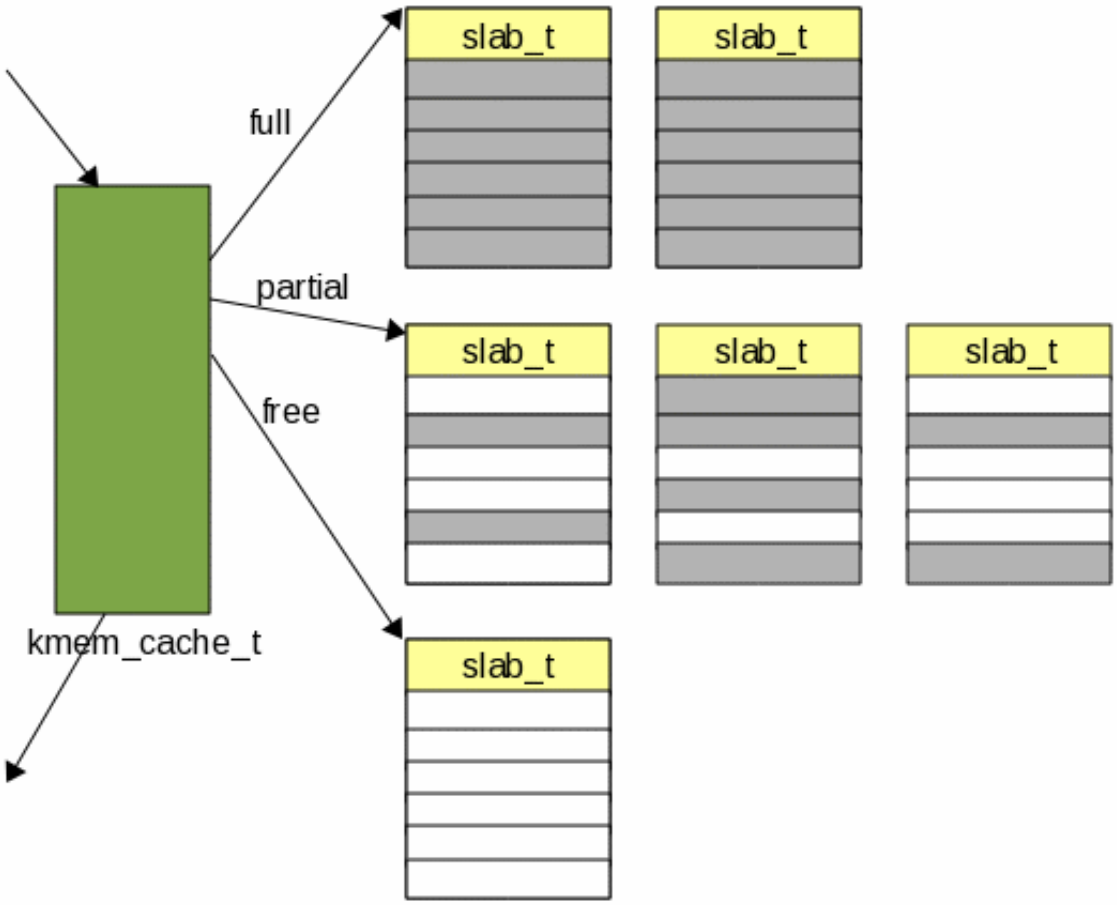

Figure 4. kmem_cache_t and its associated slabs

TODO: Add a couple of paragraphs describing use here

Cache Coloring of Slabs

Cache Coloring is a method to ensure that access to the slabs in kernel memory make the best use of the processor L1 cache. This is a performance tweak to try to ensure that we take as few cache hits as possible. Since slabs begin on page boundaries, it is likely that the objects within several different slab pages map to the same cache line, called 'false sharing'. This leads to less than optimal hardware cache performance. By offsetting each beginning of the first object within each slab by some fragment of the hardware cache line size, processor cache hits are reduced.

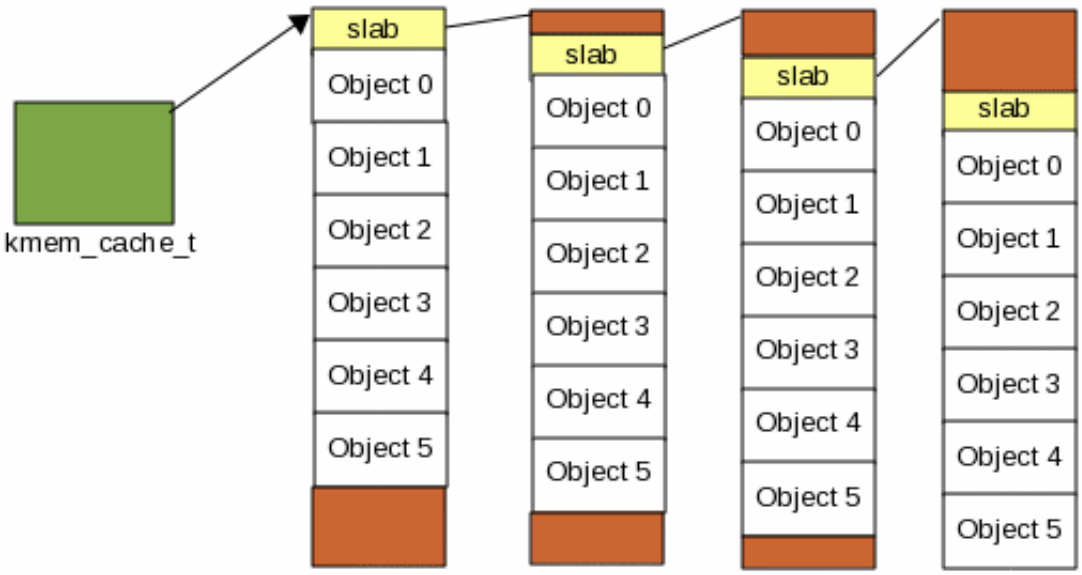

Figure 5. Cache Coloring within slabs

The `kmalloc()` Interface

Of course, there are times when a kernel module or driver needs to allocate memory for an object that doesn't fit one of the uniform types of the other caches, for example string buffers, one-off structures, temporary storage, etc. For those instances drivers and kernel modules use the **`kmalloc()`** and **`kfree()`** routines. The Linux kernel ties these calls into the slab allocator too.

On initialization, the kernel asks the slab allocator to create several caches of varying sizes for this purpose. Caches for generic objects of 32, 64, 128, 256, all the way to 131072 bytes are created for both the GFP_NORMAL and GFP_DMA zones of memory.

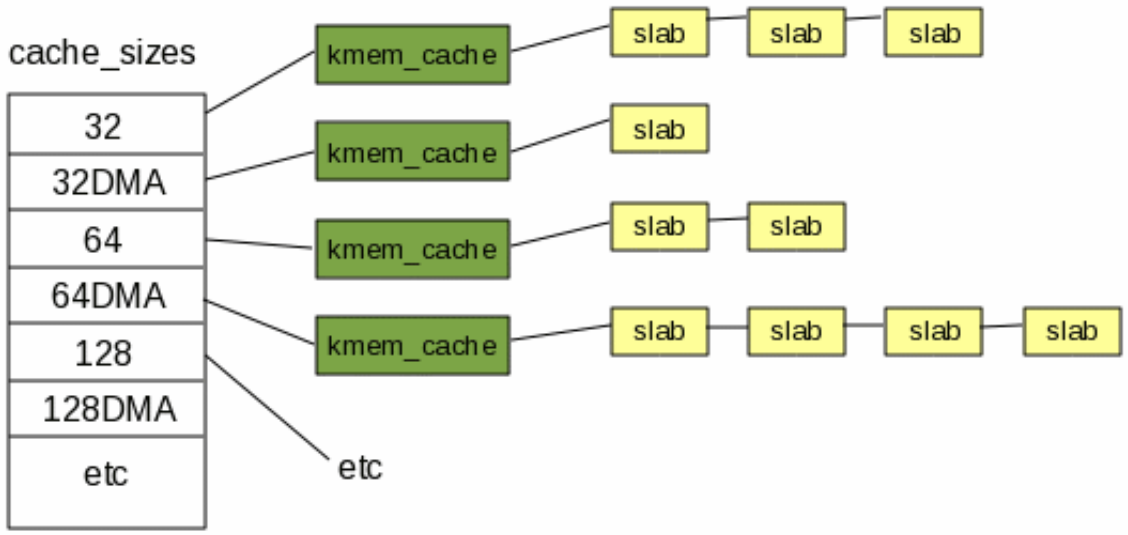

Figure 6: Use of the cache_sizes array

When a kernel module or driver needs memory, the **cache_sizes** array is searched to find the cache with the size appropriate to fit the requested object. For example, if a driver requests 166 bytes of GFP_NORMAL memory through **kmalloc()**, an object from the 256 byte cache would be returned.

When **kfree()** is called to release the object, the page the object resides in is calculated. Then the **page struct** for that page is referenced from **mem_map** (which was set up to point to our **kmem_cache_t** and **slab_t** pointers when the slab was allocated). Since we now have the slab and cache for the object, we an release it with **__kmem_cache_free()**.

TODO: Add a summary paragraph here

---

jdblakey@innovative-as.com