The Linux Slab Allocator

Brad Fitzgibbons
bradf@cc.gatech.edu

October 30, 2000

1 Introduction

This paper investigates the slab allocator residing in the newest versions of the Linux kernel, specifically version 2.4.0-test9 [2]. Few changes are expected in the implementation between the time of this writing and the eventual stable release of 2.4.0. The implementation is based upon the work by Jeff Bonwick [1].

Most memory allocations within the kernel are typified by small size and consistent structure (e.g., inodes, `vm_area_structs`, various buffers, etc.). The slab allocator (SA) provides an efficient mechanism for allocating sub-page pieces of memory and re-usable objects with static structure. It benefits the system by helping to speed up these types of allocation requests and, in addition, decrease internal fragmentation. A series of caches are kept, each responsible for managing "slabs" of like objects defined by various properties including an optional constructor/deconstructor pair. A slab is a series of contiguous physical pages in which objects are kept.

The SA provides functions to clients (sub-systems, modules) residing in the kernel to create/destroy caches (`kmem_cache_create()`, `kmem_cache_destroy()`), allocate/free objects (`kmem_cache_alloc()`, `kmem_cache_free()`), and allocate/free generic small pieces of memory (`kmalloc()`, `kfree()`). The remainder of this paper documents the internal workings of the SA as it is implemented in the Linux kernel [2].

2 Slab Data Structures

The SA consists of two primary data structures used for management: the slab (`slab_t`) and the slab cache (`kmem_cache_t`). A brief discussion of each follows.

2.1 Caches

Each cache manages a list of slabs sorted into three groups. In order, the groups contain full slabs with 0 free objects, partial slabs, and empty slabs with 0 allocated objects. A cache also maintains a spinlock for synchronization during modification.

A top-level cache (`cache_cache`) initialized at boot performs dual functions. First, it maintains a list of active caches, and second, it acts as a cache for cache objects. It is the only compound data structure within the slab allocator implementation that is not located on a slab or slab object. A global semaphore (`cache_chain_sem`) ensures exclusive access to modifying the list of active caches. See Figure 1.

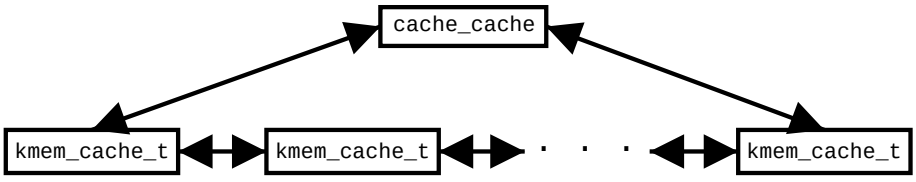
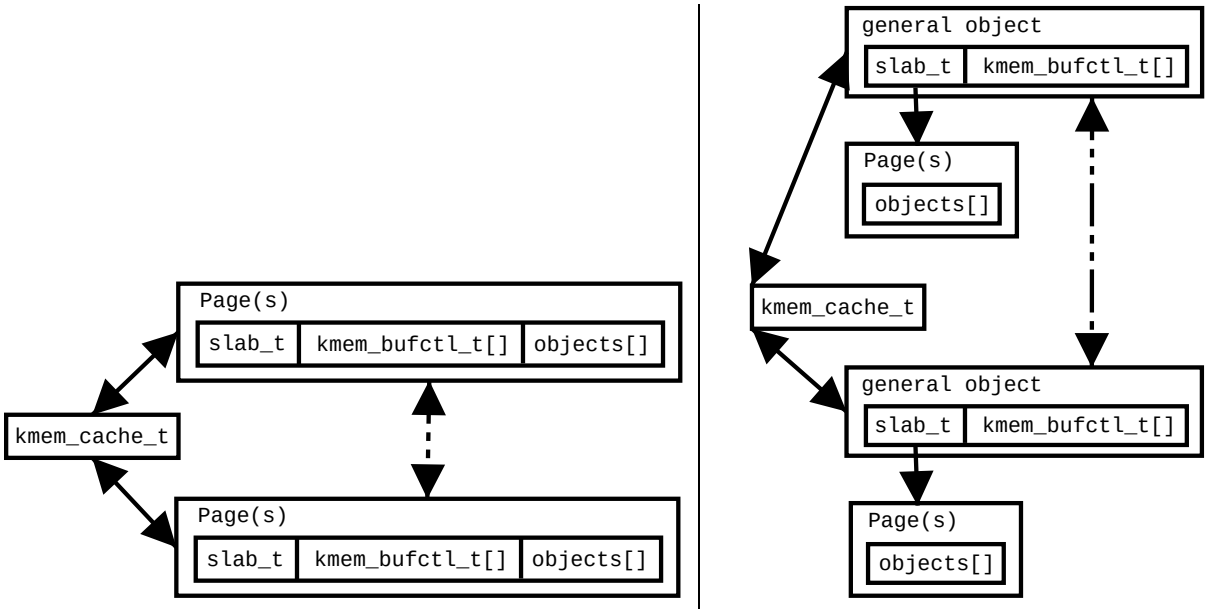


Figure 1: Slab caches in a list held by the `cache_cache`



(a) on-slab control (b) off-slab control

Figure 2: Slab control structures in a list if (a) on-slab or (b) off-slab

2.2 Slabs

The slab is the fundamental structure of the SA. It is an order of 2 contiguous pages of physical memory in which objects are sequentially placed. For small objects, slab management structures are placed on the slab (on-slab). Otherwise, slab management structures are placed off the slab (off-slab) on memory allocated from a general cache (Section 3.2). The primary slab management structure (`slab_t`) maintains a pointer to the beginning of the object array, the number of objects in use, and an index into a free list of objects. The free list is implemented using an array of indices (`kmem_bufctl_t[]`) analogous to the array of objects. Each index in the free list contains the index of the next free object. See Figure 2.

3 Implementation Details

The SA is based on the concept that most allocations from a client are of like objects. In addition, these objects typically require construction/destruction which must be performed for every allocation. The SA alleviates much of this unnecessary computation by allocating a slab of multiple objects and constructing all of them. Allocation then only requires the return of an object to a client, and when freed, the object is placed back in the slab ready for a new allocation. If a slab of objects is destroyed, then each object is destructed and the slab is freed. Therefore, each object is constructed/deconstructed once in its life-cycle and is readily re-used, leading to a more efficient allocation scheme. A reference for the SA client functions is located in Appendix A. The remainder of this section discusses the various pieces of the SA that facilitate object caching in the Linux kernel.

3.1 Cache Creation

Caches are created at the request of a client along with the properties of the objects the cache will manage. The client supplies a unique cache name, object size, optional coloring offset, optional constructor/deconstructor pair, and flags representing the memory type, alignment, and/or resistence to reclamation. Caches are created empty; the allocation of slabs is deferred to object allocation. **The creation request**

must not be made from an interrupt since creation may entail the need to sleep if a new cache object slab needs to be allocated or if it must wait on the global semaphore when linking into the top-level cache. Clients may also request the destruction of a cache as long as there are no allocated objects; modules that do not stay resident when unloaded should destroy any caches they created to ensure they can recreate them if reloaded. The relevant functions are `kmem_cache_create()` and `kmem_cache_destroy()`.

3.1.1 Calculating Slab Size

Upon obtaining a cache object, the size of the object is word aligned and if requested, L1 cache aligned. If the size of the object exceeds $\frac{1}{8}$ of the page size, the slab management structures are initially slated to be off-slab. The number of objects that will fit onto a slab are calculated for increasing page orders, beginning at 0. For on-slab calculations, the process stops when the page order reaches 2 (or 1 for systems with $\leq$ 32M memory) or internal fragmentation is $\leq \frac{1}{8}$, whichever is encountered first. For off-slab calculations, the same restrictions apply except the page order can increase to 5 as long as no objects will fit. Therefore, the object size limit for 4KB pages is $4096 * 2^5 = 131072$ bytes. For the off-slab calculation, the slab management structures will be moved back to the slab if there is ample room left over.

3.1.2 Slab Coloring

Slabs are colored based on an offset requested by the client which is forced to be word aligned if not already. If no offset is given, it is set to be the size of an L1 cache entry. When a slab is allocated, it will be offset by a multiple of this amount. The color is defined as the space left over on a slab divided by the offset. Therefore if the offset is 8 and there are 32 bytes of free space on a slab, as slabs are allocated, slab data will be offset in the order: 0, 8, 16, 24, 0, ...

3.1.3 Linking to the Top-Level Cache

The global semaphore is requested before adding the cache into the list of caches held by the top-leve cache. The new cache is prepended to the list, and the semaphore is then released.

3.2 General Caches

During boot, the top-level cache is initially populated with a series of general purpose caches for allocating contiguous areas of normal and DMA memory. These general caches range in size from $2^5 = 32$ bytes (for 4KB pages) or $2^6 = 64$ bytes (for larger page sizes) to $2^{17} = 131072$ bytes, one for each intermediate power of 2 and type of memory, totaling 26 or 24 general caches. An interface to the general caches is provided by two functions: `kmalloc()` returns a least-fit object and `kfree()` returns the object to the cache. General caches are used by the SA to allocate memory for off-slab management, and they also provide the primary method of allocating general memory throughout the kernel. The general caches are never destroyed.

3.3 Object Allocation

Once a client has created a cache, it may begin allocating objects. Recall that slabs are sorted as full, partial, and empty. Objects are first allocated from partial slabs and then from empty slabs. This allows empty slabs to stay empty in case the system needs to reclaim memory. If a cache has no partial or empty slabs (as after creation), a new slab must be allocated and objects constructed if necessary (i.e., growing a cache). The client should free the object when it is no longer needed. Interrupts are disabled during allocation and freeing to make them thread safe. The relevant functions are `kmem_cache_alloc()` and `kmem_cache_free()`.

3.3.1 Allocation Options

Several options are available to control the object allocation process as it relates to cache growing. The user can request that the cache not grow. So if there are no free objects, the cache will not allocate a new slab and immediately return. In addition, the user can specify an atomic allocation, resisting the urge to sleep if

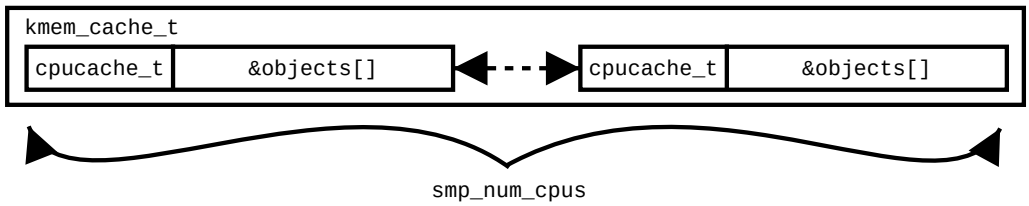


Figure 3: SMP per-CPU cache structure

required. This is necessary for cache growing to occur when an allocation request is made from within an interrupt.

3.3.2 Growing a Cache

When there are no objects free for allocation, a cache must be grown. The cache is locked to update coloring information and mark the cache as growing. This prevents the cache from being shrunk during this process. With the cache again unlocked, pages of the predetermined order are allocated and if slab management is off-slab, memory is allocated from the general caches. The slab management structures are initialized, and if a constructor exists the objects are constructed. Also, if the atomic option was specified, it is passed to the constructor. The cache is again locked to add it into the list of sorted slabs. Now allocation can continue and return a free object from the empty slab.

3.4 Memory Reclamation

The SA provides a function (`kmem_cache_reap()`) for the memory sub-system to reclaim memory. It is called in blocking or non-blocking mode, and if non-blocking, will return if unable to grab the global cache semaphore without sleeping. Once the semaphore is obtained, a modified clock algorithm is used to scan up to 10 caches for free slabs. If a cache is set to resist reclamation, it is ignored. To investigate the cache, its spinlock is grabbed. If the cache is being grown, the lock is dropped, and the next cache in the list is selected. Otherwise, while locked the number of free slabs and resulting free pages are counted. To avoid caches with high growing time, the number of pages is reduced to 80% for either having a constructor or using any order of pages > 0, a possible reduction to 64% for both. The cache spinlock is then dropped, and the algorithm moves to the next cache in the list. The cache with the highest number of free pages is saved from among the 10 unless a cache is found with ≥ 10 free slabs in which case it is saved. If no free slabs are found, the clock pointer is updated and the algorithm terminates. Otherwise, the clock pointer is moved one past the saved cache, and the algorithm proceeds to reclaim the memory (with the cache spinlock held). Upon completion, the global semaphore is released.

3.5 SMP Enhancements

Cacheing behaviors are modified for systems with more than one processor. For objects $\leq$ page size, a cache maintains a per-CPU array of object pointers to cache a stack of free objects. The per-CPU cache also maintains the size of the stack and an index to the stack head (`cpucache_t`). See Figure 3. The size of the stack is set to 252 for objects $\leq$ 256 bytes, 124 for objects $\leq$ 1024 bytes, and 60 for objects $\leq$ page size, respecting the implied intervals.

3.5.1 SMP Allocation

Caches not containing CPU caches require cache locking when allocating an object to ensure that only one processor allocates at a time. Otherwise, if an object is available on the CPU cache, the SA can safely allocate from here without locking since the CPU cache is exclusive to each processor. If the CPU cache is empty,

a batch (half the size of the stack) of objects is allocated and placed in the CPU cache. Batch allocation requires that the cache be locked for the same reason as stated above. Allocation of objects remains thread safe since interrupts are disabled during the process.

When an object is freed, it can be placed on the CPU cache without needing to lock the cache. However, if the CPU cache is full, the cache is locked, and a batch of objects is released back into the cache. The object being freed is then placed at the head of the CPU cache stack.

4 Conclusion

The Linux slab allocator provides an efficient means of allocating objects within the kernel. Small objects are efficiently managed, reducing internal fragmentation when many small allocations are required. In addition, objects are constructed/deconstructed once during their life-cycle, elimating wasted computation for re-usable objects. Per-CPU caches are maintained on SMP systems to further increase allocation speed. Ultimately, the SA provides many benefits over traditional allocators for the allocation of common structures and small pieces of memory.

References

[1] Bonwick, J. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference* (1994).

[2] Torvalds, L. linux-2.4.0-test9.tar.gz, Oct 3 2000. The linux kernel is freely available and located at ftp://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.0-test9.tar.gz.

A Client Functions

Clients residing in the kernel may interface with the slab allocator using the following functions.

A.1 Cache Creation/Destruction

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size, size_t offset,
          unsigned long c_flags,
          void (*ctor) (void *objp, kmem_cache_t *cachep, unsigned long flags),
          void (*dtor) (void *objp, kmem_cache_t *cachep, unsigned long flags))
```

`name`	cache name (< 19 characters)
`size`	object size
`offset`	requested coloring offset
`c_flags`	cache settings (See Table 1)
`ctor`	constructor function (`NULL` for none)
`dtor`	deconstructor function (`NULL` for none)
`objp`	pointer to object
`cachep`	pointer to cache
`flags`	`SLAB_CTOR_CONSTRUCTOR` optionally OR'ed with `SLAB_CTOR_ATOMIC`
return	pointer to cache (`NULL` on error)

```
int kmem_cache_destroy(kmem_cache_t *cachep)
```

`cachep`	pointer to cache to be destroyed
return	0 on success (1 otherwise)

Type	Description
`SLAB_HWCACHE_ALIGN`	Allign slab data with L1 cache
`SLAB_NO_REAP`	Do not allow system to reclaim memory
`SLAB_CACHE_DMA`	Use DMA memory for slabs

Table 1: Cache Creation Flags

A.2 Object Allocations

```
void *kmalloc(size_t size, int flags)
```

`size`	size of requested memory
`flags`	type of memory requested optionally OR'ed with `SLAB_DMA` (see Table 2)
return	pointer to memory (`NULL` on error)

```
void kfree(const void *objp)
```

`objp`	pointer to object to be freed

`void *kmem_cache_alloc(kmem_cache_t *cachep, int flags)`

`cachep`	pointer to cache from which object is requested
`flags`	same as `kmalloc()` but only used if no free objects (see Table 2)
return	pointer to object (`NULL` on error)

`void kmem_cache_free(kmem_cache_t *cachep, void *objp)`

`cachep`	pointer to cache for which objp belongs
`objp`	pointer to object to be freed

Type	Description
`SLAB_ATOMIC`	For use within interrupts (will not sleep)
`SLAB_USER`	For use in user space (may sleep)
`SLAB_KERNEL`	For use in kernel space (may sleep)

Table 2: Memory types for allocation