6

KERNEL MEMORY

In the last chapter, we looked mostly at process address space and process memory, but the kernel also needs memory to run the operating system. Kernel memory is required for the kernel text, kernel data, and kernel data structures. In this chapter, we look at what kernel memory is used for, what the kernel virtual address space looks like, and how kernel memory is allocated and managed.

6.1 Kernel Virtual Memory Layout

The kernel, just like a process, uses virtual memory and uses the memory management unit to translate its virtual memory addresses into physical pages. The kernel has its own address space and corresponding virtual memory layout. The kernel's address space is constructed of address space segments, using the standard Solaris memory architecture framework.

Most of the kernel's memory is nonpageable, or "wired down." The reason is that the kernel requires its memory to complete operating system tasks that could affect other memory-related data structures and, if the kernel had to take a page fault while performing a memory management task (or any other task that affected pages of memory), a deadlock could occur. Solaris does, however, allow some deadlock-safe parts of the Solaris kernel to be allocated from pageable memory, which is used mostly for the lightweight process thread stacks.

Kernel memory consists of a variety of mappings from physical memory (physical memory pages) to the kernel's virtual address space, and memory is allocated by a layered series of kernel memory allocators. Two segment drivers handle the creation and management of the majority of kernel mappings. Nonpageable kernel memory is mapped with the `segkmem` kernel segment driver and pageable kernel memory with the `segkp` segment driver. On platforms that support it, the critical and frequently used portions of the kernel are mapped from large (4-Mbyte) pages to maximize the efficiency of the hardware TLB.

6.1.1 Kernel Address Space

The kernel virtual memory layout differs from platform to platform, mostly based on the platform's MMU architecture. On all platforms except the sun4u, the kernel uses the top 256 Mbytes or 512 Mbytes of a common virtual address space, shared by the process and kernel (see "Virtual Address Spaces" on page 173). Sharing the kernel address space with the process address space limits the amount of usable kernel virtual address space to 256 Mbytes and 512 Mbytes, respectively, which is a substantial limitation on some of the older platforms (e.g., the SPARCcenter 2000). On sun4u platforms, the kernel has its own virtual address space context and consequently can be much larger. The sun4u kernel address space is 4 Gbytes on 32-bit kernels and spans the full 64 bit address range on 64-bit kernels.

The kernel virtual address space contains the following major mappings:

- The kernel text and data (mappings of the kernel binary)
- The kernel map space (data structures, caches, etc.)
- A 32-bit kernel map, for module text and data (64-bit kernels only)
- The trap table
- Critical virtual memory data structures (TSB, etc.)
- A place for mapping the file system cache (`segmap`)

The layout of the kernel's virtual memory address space is mostly platform specific, and as a result, the placement of each mapping is different on each platform. For reference, we show the sun4u 64-bit kernel address space map in Figure 6.1.

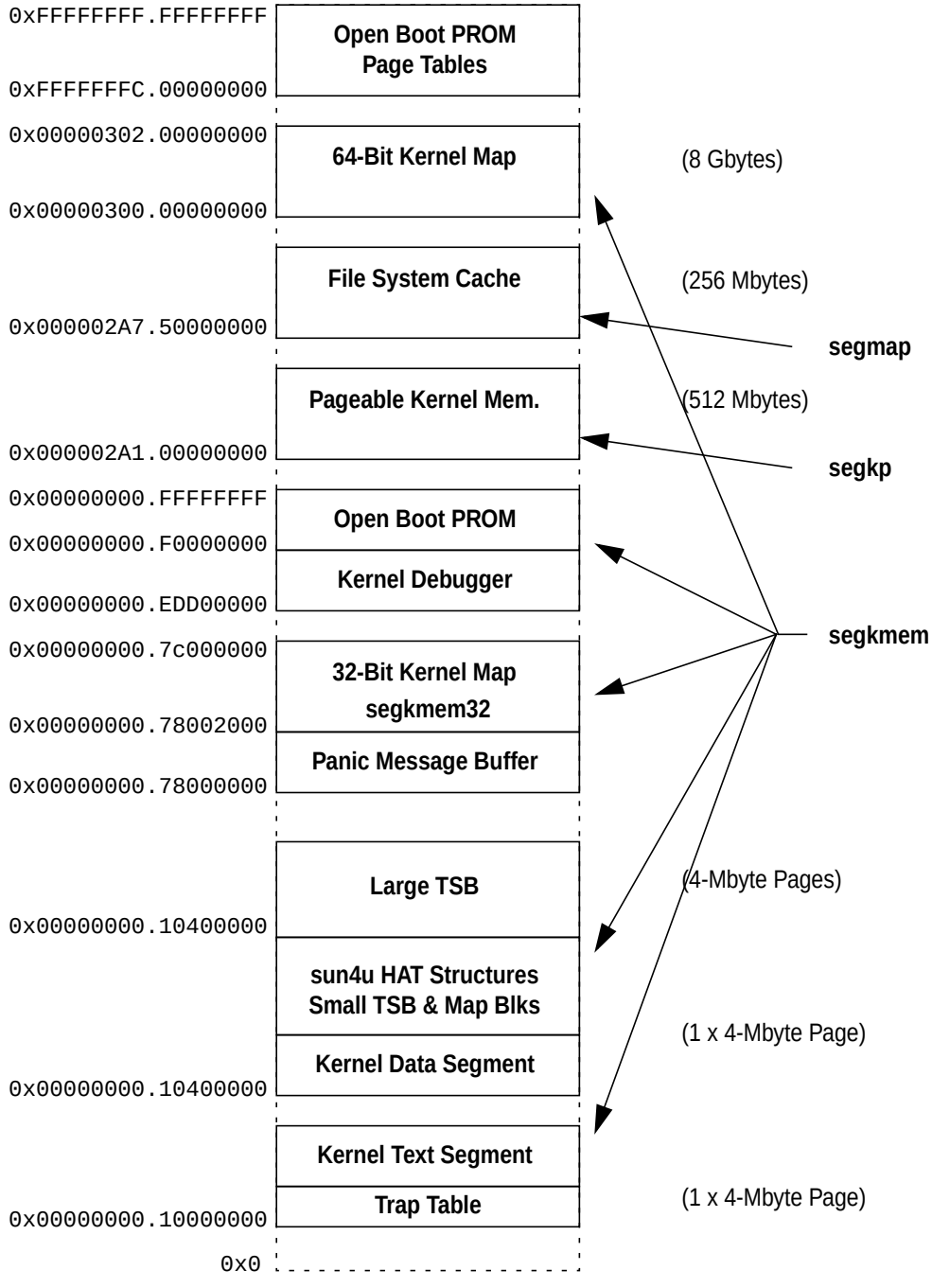

Figure 6.1 Solaris 7 64-Bit Kernel Virtual Address Space

6.1.2 The Kernel Text and Data Segments

The kernel text and data segments are created when the kernel core is loaded and executed. The text segments contain the instructions, and the data segment contains the initialized variables from the kernel/`unix` image file, which is loaded at boot time by the kernel bootstrap loader.

The kernel text and data are mapped into the kernel address space by the Open Boot PROM, prior to general startup of the kernel, to allow the base kernel code to be loaded and executed. Shortly after the kernel loads, the kernel then creates the kernel address space and the segkmem kernel memory driver creates segments for kernel text and kernel data.

On systems that support large pages, the kernel creates a large translation mapping for the first four megabytes of the kernel text and data segments and then locks that mapping into the MMU's TLB. Mapping the kernel into large pages greatly reduces the number of TLB entries required for the kernel's working set and has a dramatic impact on general system performance. Performance was increased by as much as 10 percent, for two reasons:

1. The time spent in TLB miss handlers for kernel code was reduced to almost zero.
2. The number of TLB entries used by the kernel was dramatically reduced, leaving more TLB entries for user code and reducing the amount of time spent in TLB miss handlers for user code.

On SPARC platforms, we also put the trap table at the start of the kernel text (which resides on one large page).

6.1.3 Virtual Memory Data Structures

The kernel keeps most of the virtual memory data structures required for the platform's HAT implementation in a portion of the kernel data segment and a separate memory segment. The data structures and allocation location are typically those summarized in Table 6-1.

Table 6-1 Virtual Memory Data Structures

Platform	Data Structures	Location
sun4u	The Translation Storage Buffer (TSB). The HAT mapping blocks (HME), one for every page-sized virtual address mapping. (See "The UltraSPARC-I and -II HAT" on page 235.)	Allocated initially from the kernel data-segment large page, and overflows into another large-page, mapped segment, just above the kernel data segment.
sun4m	Page Tables, Page Structures	Allocated in the kernel data-segment large page.

Table 6-1 Virtual Memory Data Structures (Continued)

Platform	Data Structures	Location
sun4d	Page Tables, Page Structures	Allocated in the kernel data-segment large page.
x86	Page Tables, Page Structures	Allocated from a separate VM data structure's segment.

6.1.4 The SPARC V8 and V9 Kernel Nucleus

Required on sun4u kernel implementations is a core area of memory that can be accessed without missing in the TLB. This memory area is necessary because the sun4u SPARC implementation uses a software TLB replacement mechanism to fill the TLB, and hence we require all of the TLB miss handler data structures to be available during a TLB miss. As we discuss in "The UltraSPARC-I and -II HAT" on page 235, the TLB is filled from a software buffer, known as the translation storage buffer (TSB), of the TLB entries, and all of the data structures needed to handle a TLB miss and to fill the TLB from the TSB must be available with wired-down TLB mappings. To accommodate this requirement, SPARC V8 and SPARC V9 implement a special core of memory, known as the nucleus. On sun4u systems, the nucleus is the kernel text, kernel data, and the additional "large TSB" area, all of which are allocated from large pages.

6.1.5 Loadable Kernel Module Text and Data

The kernel loadable modules require memory for their executable text and data. On sun4u, up to 256 Kbytes of module text and data are allocated from the same segment as the kernel text and data, and after the module text and data are loaded from the general kernel allocation area, the kernel map segment. The location of kernel module text and data is shown in Table 6-2.

Table 6-2 Kernel Loadable Module Allocation

Platform	Module Kernel and Text Allocation
sun4u 64 bit	Up to 256 Kbytes of kernel module are loaded from the same large pages as the kernel text and data. The remainder are loaded from the 32-bit kernel map segment, a segment that is specifically for module text and data.
sun4u 32 bit	Up to 256 Kbytes of kernel module are loaded from the same large pages as the kernel text and data. The remainder are loaded from the general kernel memory allocation segment, the kernel map segment.
sun4m	Loadable module text and data are loaded from the general kernel memory allocation segment, the `kernelmap` segment.

Table 6-2 Kernel Loadable Module Allocation (Continued)

Platform	Module Kernel and Text Allocation
sun4d	Loadable module text and data are loaded from the general kernel memory allocation segment, the `kernelmap` segment.
x86	Up to 256 Kbytes of kernel module are loaded from the same large pages as the kernel text and data. The remainder are loaded from an additional segment, shared by HAT data structures and module text/data.

We can see which modules fitted into the kernel text and data by looking at the module load addresses with the `modinfo` command.

```
rmc@devhome> modinfo
 Id Loadaddr   Size Info Rev Module Name
  5 1010c000   4b63   1   1  specfs (filesystem for specfs)
  7 10111654   3724   1   1  TS (time sharing sched class)
  8 1011416c    5c0   -   1  TS_DPTBL (Time sharing dispatch table)
  9 101141c0  29680   2   1  ufs (filesystem for ufs)
         .
         .
         .
         .
 97 10309b38   28e0  52   1  shmsys (System V shared memory)
 97 10309b38   28e0  52   1  shmsys (32-bit System V shared memory)
 98 1030bc90    43c   -   1  ipc (common ipc code)
 99 78096000   3723  18   1  ffb (ffb.c 6.42 Aug 11 1998 11:20:45)
100 7809c000   f5ee   -   1  xfb (xfb driver 1.2 Aug 11 1998 11:2)
102 780c2000   1eca   -   1  bootdev (bootdev misc module)
```

Using the `modinfo` command, we can see on a sun4u system that the initial modules are loaded from the kernel-text large page. (Address 0x1030bc90 lies within the kernel-ext large page, which starts at 0x10000000.)

On 64-bit sun4u platforms, we have an additional segment for the spillover kernel text and data. The reason for having the segment is that the address at which the module text is loaded must be within a 32-bit offset from the kernel text. That's because the 64-bit kernel is compiled with the ABS32 flag so that the kernel can fit all instruction addresses within a 32-bit register. The ABS32 instruction mode provides a significant performance increase and allows the 64-bit kernel to provide similar performance to the 32-bit kernel. Because of that, a separate kernel map segment (`segkmem32`) within a 32-bit offset of the kernel text is used for spillover module text and data.

Solaris does allow some portions of the kernel to be allocated from pageable memory. That way, data structures directly related to process context can be swapped out with the process during a process swap-out operation. Pageable memory is restricted to those structures that are not required by the kernel when the process is swapped out:

- Lightweight process stacks

- The TNF Trace buffers
- Special pages, such as the page of memory that is shared between user and kernel for scheduler preemption control

Pageable memory is allocated and swapped by the seg_kp segment and is only swapped out to its backing store when the memory scheduler (swapper) is activated. (See “The Memory Scheduler” on page 231.)

6.1.6 The Kernel Address Space and Segments

The kernel address space is represented by the address space pointed to by the system object, kas. The segment drivers manage the manipulation of the segments within the kernel address space (see Figure 6.2).

The full list of segment drivers the kernel uses to create and manage kernel mappings is shown in Table 6-3. The majority of the kernel segments are manually calculated and placed for each platform, with the base address and offset hard-coded into a platform-specific header file. See Appendix B, “Kernel Virtual Address Maps” for a complete reference of platform-specific kernel allocation and address maps.

Table 6-3 Solaris 7 Kernel Memory Segment Drivers

Segment	Function
seg_kmem	Allocates and maps nonpageable kernel memory pages.
seg_kp	Allocates, maps, and handles page faults for pageable kernel memory.
seg_nf	Nonfaulting kernel memory driver.
seg_map	Maps the file system cache into the kernel address space.

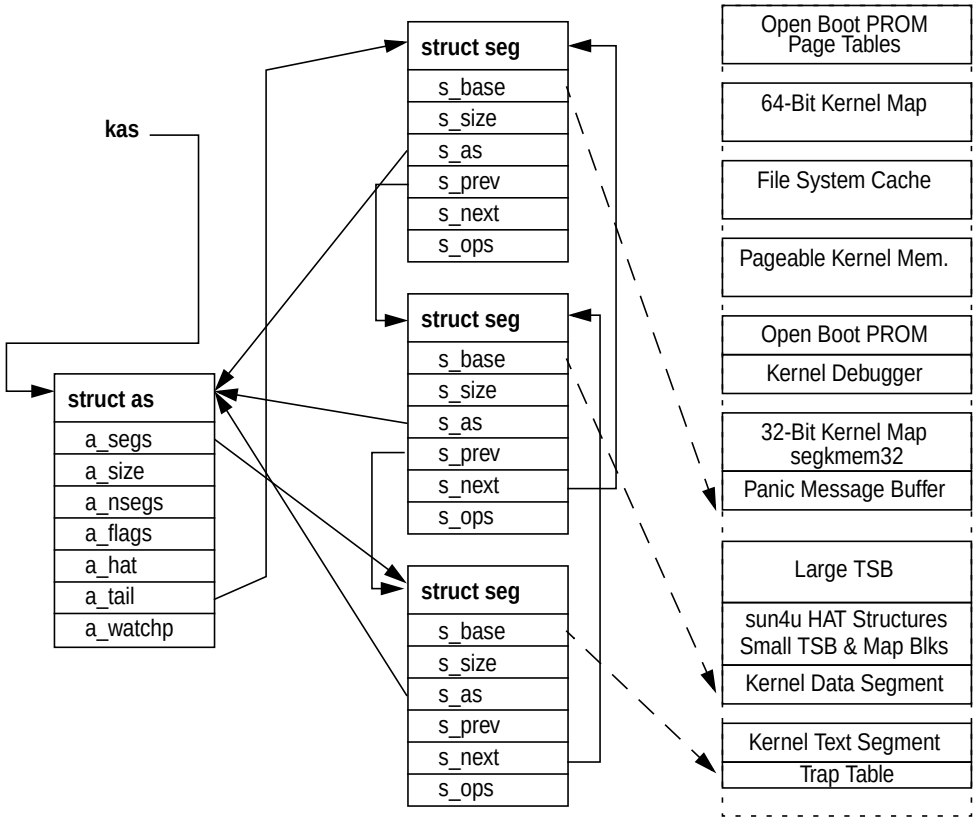

Figure 6.2 Kernel Address Space

6.2 Kernel Memory Allocation

Kernel memory is allocated at different levels, depending on the desired allocation characteristics. At the lowest level is the page allocator, which allocates unmapped pages from the free lists so the pages can then be mapped into the kernel's address space for use by the kernel.

Allocating memory in pages works well for memory allocations that require page-sized chunks, but there are many places where we need memory allocations smaller than one page; for example, an in-kernel inode requires only a few hundred bytes per inode, and allocating one whole page (8 Kbytes) would be wasteful. For this reason, Solaris has an object-level kernel memory allocator in addition to the page-level allocator to allocate arbitrarily sized requests, stacked on top of the

page-level allocator. The kernel also needs to manage where pages are mapped, a function that is provided by the resource map allocator. The high-level interaction between the allocators is shown in Figure 6.3.

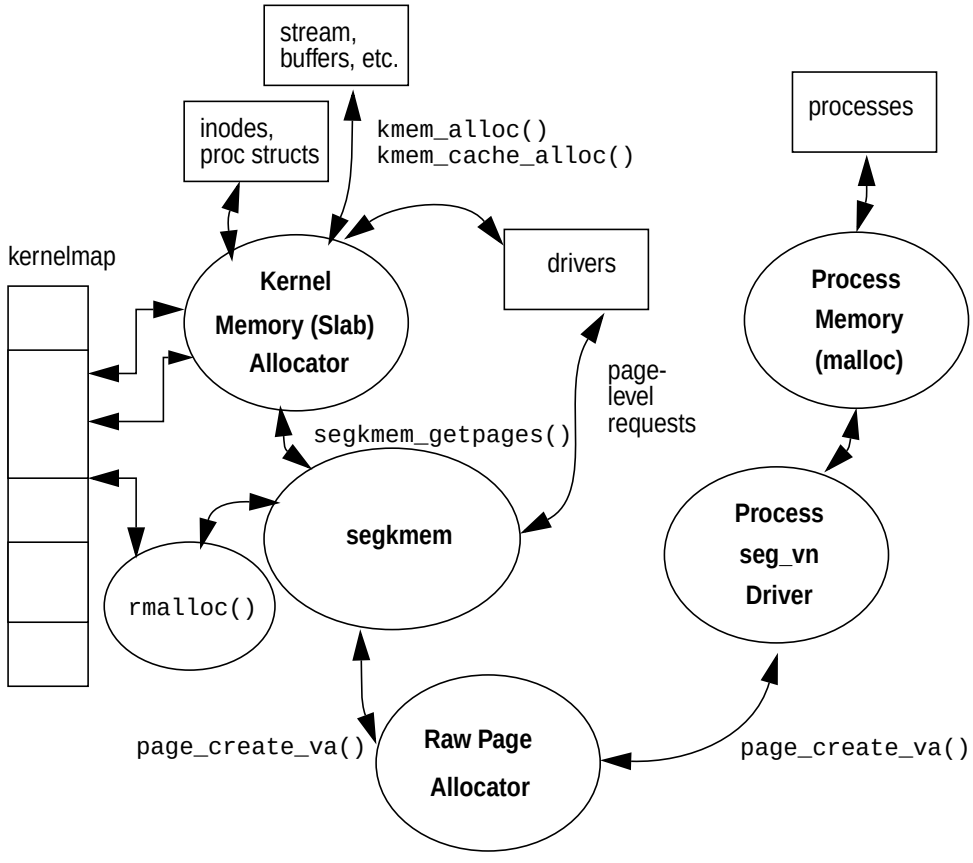

Figure 6.3 Different Levels of Memory Allocation

6.2.1 The Kernel Map

We access memory in the kernel by acquiring a section of the kernel's virtual address space and then mapping physical pages to that address. We can acquire the physical pages one at a time from the page allocator by calling `page_create_va()`, but to use these pages, we first need to map them. A section of the kernel's address space, known as the *kernel map*, is set aside for general-purpose mappings*p*. (See Figure 6.1 for the location of the sun4u kernelmap; see also Appendix B, "Kernel Virtual Address Maps" for kernel maps on other platforms.)

The kernel map is a separate kernel memory segment containing a large area of virtual address space that is available to kernel consumers who require virtual address space for their mappings. Each time a consumer uses a piece of the kernel map, we must record some information about which parts of the kernel map are free and which parts are allocated, so that we know where to satisfy new requests. To record the information, we use a general-purpose allocator to keep track of the start and length of the mappings that are allocated from the kernel map area. The allocator we use is the resource map allocator, which is used almost exclusively for managing the kernel map virtual address space.

The kernel map area is large, up to 8 Gbytes on 64-bit sun4u systems, and can quickly become fragmented if it accommodates many consumers with different-sized requests. It is up to the resource map allocator to try to keep the kernel map area as unfragmented as possible.

6.2.2 The Resource Map Allocator

Solaris uses the resource map allocator to manage the kernel map. To keep track of the areas of free memory within the map, the resource map allocator uses a simple algorithm to keep a list of start/length pairs that point to each area of free memory within the map. The map entries are sorted in ascending order to make it quicker to find entries, allowing faster allocation. The map entries are shown in the following map structure, which can be found in the `<sys/map.h>` header file.

```
struct map {
        size_t  m_size;         /* size of this segment of the map */
        ulong_t m_addr;         /* resource-space addr of start of segment */
};
```

The area managed by the resource map allocator is initially described by just one map entry representing the whole area as one contiguous free chunk. As more allocations are made from the area, more map entries are used to describe the area, and as a result, the map becomes ever more fragmented over time.

The resource map allocator uses a first-fit algorithm to find space in the map to satisfy new requests, which means that it attempts to find the first available slot in the map that fits the request. The first-fit algorithm provides a fast find allocation at the expense of map fragmentation after time. For this reason, it is important to ensure that kernel subsystems do not perform an excessive amount of map allocation and freeing. The kernel slab allocator (discussed next) should be used for these types of requests.

Map resource requests are made with the `rmalloc()` call, and resources are returned to the map by `rmfree()`. Resource maps are created with the `rmallocmap()` function and destroyed with the `rmfreemap()` function. The functions that implement the resource map allocator are shown in Table 6-4.

Table 6-4 Solaris 7 Resource Map Allocator Functions from <sys/map.h>

Function	Description
`rmallocmap()`	Dynamically allocates a map. Does not sleep. Driver-defined basic locks, read/write locks, and sleep locks can be held across calls to this function. DDI-/DKI-conforming drivers may only use map structures that have been allocated and initialized with `rmallocmap()`.
`rmallocmap_wait()`	Dynamically allocates a map. It *does* sleep. DDI-/DKI-conforming drivers can only use map structures that have been allocated and initialized with `rmallocmap()` and `rmallocmap_wait()`.
`rmfreemap()`	Frees a dynamically allocated map. Does not sleep. Driver-defined basic locks, read/write locks, and sleep locks can be held across calls to this function. Before freeing the map, the caller must ensure that nothing is using space managed by the map and that nothing is waiting for space in the map.
`rmalloc()`	Allocates *`size`* units from the given map. Returns the base of the allocated space. In a map, the addresses are increasing and the list is terminated by a 0 size. Algorithm is first-fit.
`rmalloc_wait()`	Like `rmalloc`, but waits if necessary until space is available.
`rmalloc_locked()`	Like `rmalloc`, but called with lock on map held.
`rmfree()`	Frees the previously allocated space at *`addr`* of *`size`* units into the specified map. Sorts *`addr`* into map and combines on one or both ends if possible.
`rmget()`	Allocates *`size`* units from the given map, starting at address *`addr`*. Returns *`addr`* if successful, 0 if not. This may cause the creation or destruction of a resource map segment. This routine returns failure status if there is not enough room for a required additional map segment.

6.2.3 The Kernel Memory Segment Driver

The segkmem segment driver performs two major functions: it manages the creation of general-purpose memory segments in the kernel address space, and it also provides functions that implement a page-level memory allocator by using one of those segments—the kernel map segment.

The segkmem segment driver implements the segment driver methods described in Section 5.4, "Memory Segments," on page 185, to create general-purpose, nonpageable memory segments in the kernel address space. The segment driver does little more than implement the segkmem_create method to simply link segments into the kernel's address space. It also implements protection manipulation methods, which load the correct protection modes via the HAT layer for segkmem segments. The set of methods implemented by the segkmem driver is shown in Table 6-5.

Table 6-5 Solaris 7 segkmem Segment Driver Methods

Function	Description
segkmem_create()	Creates a new kernel memory segment.
segkmem_setprot()	Sets the protection mode for the supplied segment.
segkmem_checkprot()	Checks the protection mode for the supplied segment.
segkmem_getprot()	Gets the protection mode for the current segment.

The second function of the segkmem driver is to implement a page-level memory allocator by combined use of the resource map allocator and page allocator. The page-level memory allocator within the segkmem driver is implemented with the function kmem_getpages(). The kmem_getpages() function is the kernel's central allocator for wired-down, page-sized memory requests. Its main client is the second-level memory allocator, the *slab* allocator, which uses large memory areas allocated from the page-level allocator to allocate arbitrarily sized memory objects. We'll cover more on the slab allocator further in this chapter.

The kmem_getpages() function allocates page-sized chunks of virtual address space from the kernelmap segment. The kernelmap segment is only one of many segments created by the segkmem driver, but it is the only one from which the segkmem driver allocates memory.

The resource map allocator allocates portions of virtual address space within the kernelmap segment but on its own does not allocate any physical memory resources. It is used together with the page allocator, page_create_va(), and the hat_memload() functions to allocate physical mapped memory. The resource map allocator allocates some virtual address space, the page allocator allocates pages, and the hat_memload() function maps those pages into the virtual address space provided by the resource map. A client of the segkmem memory allocator can acquire pages with kmem_getpages and then return them to the map with kmem_freepages, as shown in Table 6-6.

Table 6-6 Solaris 7 Kernel Page Level Memory Allocator

Function	Description
`kmem_getpages()`	Allocates *`npages`* pages worth of system virtual address space, and allocates wired-down page frames to back them. If *`flag`* is `KM_NOSLEEP`, blocks until address space and page frames are available.
`kmem_freepages()`	Frees *`npages`* (MMU) pages allocated with `kmem_getpages()`.

Pages allocated through `kmem_getpages` are not pageable and are one of the few exceptions in the Solaris environment where a mapped page has no logically associated vnode. To accommodate that case, a special vnode, `kvp`, is used. All pages created through the `segkmem` segment have `kvp` as the vnode in their identity—this allows the kernel to identify wired-down kernel pages.

6.2.4 The Kernel Memory Slab Allocator

In this section, we introduce the general-purpose memory allocator, known as the slab allocator. We begin with a quick walk-through of the slab allocator features, then look at how the allocator implements object caching, and follow up with a more detailed discussion on the internal implementation.

6.2.4.1 Slab Allocator Overview

Solaris provides a general-purpose memory allocator that provides arbitrarily sized memory allocations. We refer to this allocator as the slab allocator because it consumes large slabs of memory and then allocates smaller requests with portions of each slab. We use the slab allocator for memory requests that are:

- Smaller than a page size
- Not an even multiple of a page size
- Frequently going to be allocated and *freed*, so would otherwise fragment the kernel map

The slab allocator was introduced in Solaris 2.4, replacing the *buddy* allocator that was part of the original SVR4 Unix. The reasons for introducing the slab allocator were as follows:

- The SVR4 allocator was slow to satisfy allocation requests.
- Significant fragmentation problems arose with use of the SVR4 allocator.
- The allocator footprint was large, wasting a lot of memory.
- With no clean interfaces for memory allocation, code was duplicated in many places.

The slab allocator solves those problems and dramatically reduces overall system complexity. In fact, when the slab allocator was integrated into Solaris, it resulted in a net reduction of 3,000 lines of code because we could centralize a great deal of the memory allocation code and could remove a lot of the duplicated memory allocator functions from the clients of the memory allocator.

The slab allocator is significantly faster than the SVR4 allocator it replaced. Table 6-7 shows some of the performance measurements that were made when the slab allocator was first introduced.

Table 6-7 Performance Comparison of the Slab Allocator

Operation	SVR4	Slab
Average time to allocate and free	9.4 µs	3.8 µs
Total fragmentation (wasted memory)	46%	14%
Kenbus benchmark performance (number of scripts executed per second)	199	233

The slab allocator provides substantial additional functionality, including the following:

- General-purpose, variable-sized memory object allocation
- A central interface for memory allocation, which simplifies clients of the allocator and reduces duplicated allocation code
- Very fast allocation/deallocation of objects
- Low fragmentation / small allocator footprint
- Full debugging and auditing capability
- Coloring to optimize use of CPU caches
- Per-processor caching of memory objects to reduce contention
- A configurable back-end memory allocator to allocate objects other than regular wired-down memory

The slab allocator uses the term *object* to describe a single memory allocation unit, *cache* to refer to a pool of like objects, and *slab* to refer to a group of objects that reside within the cache. Each object type has one cache, which is constructed from one or more slabs. Figure 6.4 shows the relationship between objects, slabs, and the cache. The example shows 3-Kbyte memory objects within a cache, backed by 8-Kbyte pages.

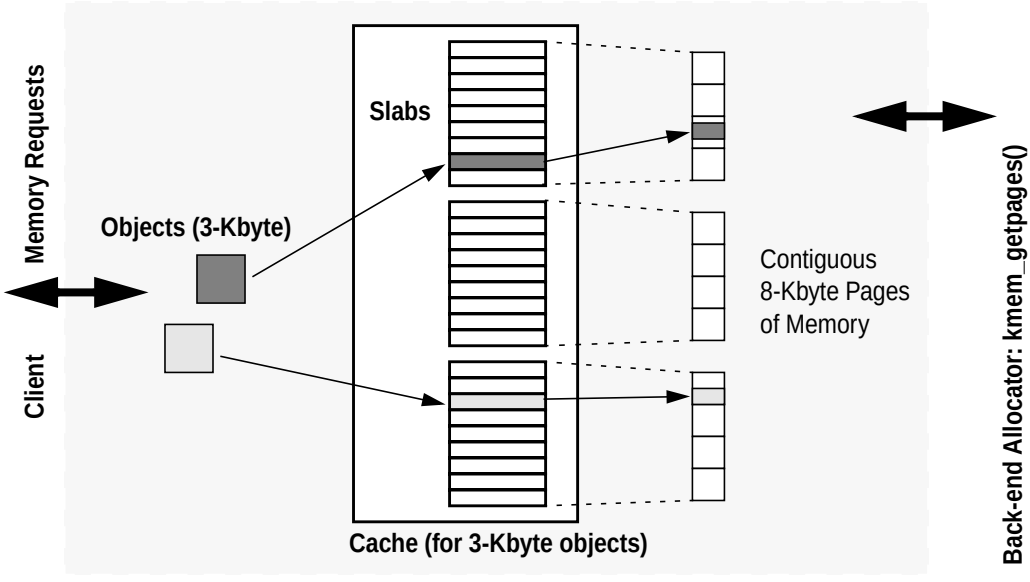

Figure 6.4 Objects, Caches, Slabs, and Pages of Memory

The slab allocator solves many of the fragmentation issues by grouping different-sized memory objects into separate caches, where each object cache has its own object size and characteristics. Grouping the memory objects into caches of similar size allows the allocator to minimize the amount of free space within each cache by neatly packing objects into slabs, where each slab in the cache represents a contiguous group of pages. Since we have one cache per object type, we would expect to see many caches active at once in the Solaris kernel. For example, we should expect to see one cache with 440 byte objects for UFS inodes, another cache of 56 byte objects for file structures, another cache of 872 bytes for LWP structures, and several other caches.

The allocator has a logical front end and back end. Objects are allocated from the front end, and slabs are allocated from pages supplied by the back-end page allocator. This approach allows the slab allocator to be used for more than regular wired-down memory; in fact, the allocator can allocate almost any type of memory object. The allocator is, however, primarily used to allocate memory objects from physical pages by using `kmem_getpages` as the back-end allocator.

Caches are created with `kmem_cache_create()`, once for each type of memory object. Caches are generally created during subsystem initialization, for example, in the `init` routine of a loadable driver. Similarly, caches are destroyed with the `kmem_cache_destroy()` function. Caches are named by a string provided as an argument, to allow friendlier statistics and tags for debugging. Once a cache is created, objects can be created within the cache with `kmem_cache_alloc()`, which

creates one object of the size associated with the cache from which the object is created. Objects are returned to the cache with `kmem_cache_free()`.

6.2.4.2 Object Caching

The slab allocator makes use of the fact that most of the time objects are heavily allocated and deallocated, and many of the slab allocator's benefits arise from resolving the issues surrounding allocation and deallocation. The allocator tries to defer most of the real work associated with allocation and deallocation until it is really necessary, by keeping the objects alive until memory needs to be returned to the back end. It does this by telling the allocator what the object is being used for, so that the allocator remains in control of the object's true state.

So, what do we really mean by keeping the object *alive*? If we look at what a subsystem uses memory objects for, we find that a memory object typically consists of two common components: the header or description of what resides within the object and associated locks; and the actual payload that resides within the object. A subsystem typically allocates memory for the object, constructs the object in some way (writes a header inside the object or adds it to a list), and then creates any locks required to synchronize access to the object. The subsystem then uses the object. When finished with the object, the subsystem must deconstruct the object, release locks, and then return the memory to the allocator. In short, a subsystem typically allocates, constructs, uses, deallocates, and then frees the object.

If the object is being created and destroyed often, then a great deal of work is expended constructing and deconstructing the object. The slab allocator does away with this extra work by caching the object in its constructed form. When the client asks for a new object, the allocator simply creates or finds an available constructed object. When the client returns an object, the allocator does nothing other than mark the object as free, leaving all of the constructed data (header information and locks) intact. The object can be reused by the client subsystem without the allocator needing to construct or deconstruct—the construction and deconstruction is only done when the cache needs to grow or shrink. Deconstruction is deferred until the allocator needs to free memory back to the back-end allocator.

To allow the slab allocator to take ownership of constructing and deconstructing objects, the client subsystem must provide a constructor and destructor method. This service allows the allocator to construct new objects as required and then to deconstruct objects later, asynchronously to the client's memory requests. The `kmem_cache_create()` interface supports this feature by providing a constructor and destructor function as part of the create request.

The slab allocator also allows slab caches to be created with no constructor or destructor, to allow simple allocation and deallocation of simple raw memory objects.

The slab allocator moves a lot of the complexity out of the clients and centralizes memory allocation and deallocation policies. At some points, the allocator may need to shrink a cache as a result of being notified of a memory shortage by the

VM system. At this time, the allocator can free all unused objects by calling the destructor for each object that is marked free and then returning unused slabs to the back-end allocator. A further callback interface is provided in each cache so that the allocator can let the client subsystem know about the memory pressure. This callback is optionally supplied when the cache is created and is simply a function that the client implements to return, by means of `kmem_cache_free()`, as many objects to the cache as possible.

A good example is a file system, which uses objects to store the inodes. The slab allocator manages inode objects; the cache management, construction, and deconstruction of inodes are handed over to the slab allocator. The file system simply asks the slab allocator for a "new inode" each time it requires one. For example, a file system could call the slab allocator to create a slab cache, as shown below.

```
inode_cache = kmem_cache_create("inode_cache",
        sizeof (struct inode), 0, inode_cache_constructor,
        inode_cache_destructor, inode_cache_reclaim,
        NULL, NULL, 0);

struct inode *inode = kmem_cache_alloc(inode_cache, 0);
```

The example shows that we create a cache named `inode_cache`, with objects of the size of an inode, no alignment enforcement, a constructor and a destructor function, and a reclaim function. The back-end memory allocator is specified as `NULL`, which by default allocates physical pages from the `segkmem` page allocator.

We can see from the statistics exported by the slab allocator that the UFS file system uses a similar mechanism to allocate its inodes. We use the `netstat -k` function to dump the statistics. (We discuss allocator statistics in more detail in "Slab Allocator Statistics" on page 270.)

```
# netstat -k ufs_inode_cache
ufs_inode_cache:
buf_size 440 align 8 chunk_size 440 slab_size 8192 alloc 20248589
alloc_fail 0 free 20770500 depot_alloc 657344 depot_free 678433
depot_contention 85 global_alloc 602986 global_free 578089
buf_constructed 0 buf_avail 7971 buf_inuse 24897 buf_total 32868
buf_max 41076 slab_create 2802 slab_destroy 976 memory_class 0
hash_size 0 hash_lookup_depth 0 hash_rescale 0 full_magazines 0
empty_magazines 0 magazine_size 31 alloc_from_cpu0 9583811
free_to_cpu0 10344474 buf_avail_cpu0 0 alloc_from_cpu1 9404448
free_to_cpu1 9169504 buf_avail_cpu1 0
```

The allocator interfaces are shown in Table 6-8.

Table 6-8 Solaris 7 Slab Allocator Interfaces from <sys/kmem.h>

Function	Description
kmem_cache_create()	Creates a new slab cache with the supplied *name*, aligning objects on the boundary supplied with alignment. The constructor, destructor, and reclaim functions are optional and can be supplied as NULL. An argument can be provided to the constructor with *arg*. The back-end memory allocator can also be specified or supplied as NULL. If a NULL back-end allocator is supplied, then the default allocator, kmem_getpages(), is used. Flags can supplied as be KMC_NOTOUCH, KMC_NODEBUG, KMC_NOMAGAZINE, and KMC_NOHASH.
kmem_cache_destroy()	Destroys the cache referenced by *cp*.
kmem_cache_alloc()	Allocates one object from the cache referenced by *cp*. Flags can be supplied as either KM_SLEEP or KM_NOSLEEP.
kmem_cache_free()	Returns the buffer *buf* to the cache referenced by *cp*.
kmem_cache_stat()	Returns a named statistic about a particular cache that matches the string name. Finds a name by looking at the kstat slab cache names with netstat -k.

Caches are created with the kmem_cache_create() function, which can optionally supply callbacks for construction, deletion, and cache reclaim notifications. The callback functions are described in Table 6-9.

Table 6-9 Slab Allocator Callback Interfaces from <sys/kmem.h>

Function	Description
constructor()	Initializes the object *buf*. The arguments *arg* and *flag* are those provided during kmem_cache_create().
destructor()	Destroys the object *buf*. The argument *arg* is that provided during kmem_cache_create().
reclaim()	Where possible, returns objects to the cache. The argument is that provided during kmem_cache_create().

6.2.4.3 General-Purpose Allocations

In addition to object-based memory allocation, the slab allocator provides backward-compatible, general-purpose memory allocation routines. These routines allocate arbitrary-length memory by providing a method to `malloc()`. The slab allocator maintains a list of various-sized slabs to accommodate `kmem_alloc()` requests and simply converts the `kmem_alloc()` request into a request for an object from the nearest-sized cache. The sizes of the caches used for `kmem_alloc()` are named `kmem_alloc_`*n*, where *n* is the size of the objects within the cache (see Section 6.2.4.9, "Slab Allocator Statistics," on page 270). The functions are shown in Table 6-10.

Table 6-10 General-Purpose Memory Allocation

Function	Description
`kmem_alloc()`	Allocates *`size`* bytes of memory. Flags can be either `KM_SLEEP` or `KM_NOSLEEP`.
`kmem_zalloc()`	Allocates *`size`* bytes of zeroed memory. Flags can be either `KM_SLEEP` or `KM_NOSLEEP`.
`kmem_free()`	Returns to the allocator the buffer pointed to by *`buf`* and *`size`*.

6.2.4.4 Slab Allocator Implementation

The slab allocator implements the allocation and management of objects to the front-end clients, using memory provided by the back-end allocator. In our introduction to the slab allocator, we discussed in some detail the virtual allocation units: the object and the slab. The slab allocator implements several internal layers to provide efficient allocation of objects from slabs. The extra internal layers reduce the amount of contention between allocation requests from multiple threads, which ultimately allows the allocator to provide good scalability on large SMP systems.

Figure 6.5 shows the internal layers of the slab allocator. The additional layers provide a cache of allocated objects for each CPU, so a thread can allocate an object from a local per-CPU object cache without having to hold a lock on the global slab cache. For example, if two threads both want to allocate an inode object from the inode cache, then the first thread's allocation request would hold a lock on the inode cache and would block the second thread until the first thread has its object allocated. The per-cpu cache layers overcome this blocking with an object cache per CPU to try to avoid the contention between two concurrent requests. Each CPU has its own short-term cache of objects, which reduces the amount of time that each request needs to go down into the global slab cache.

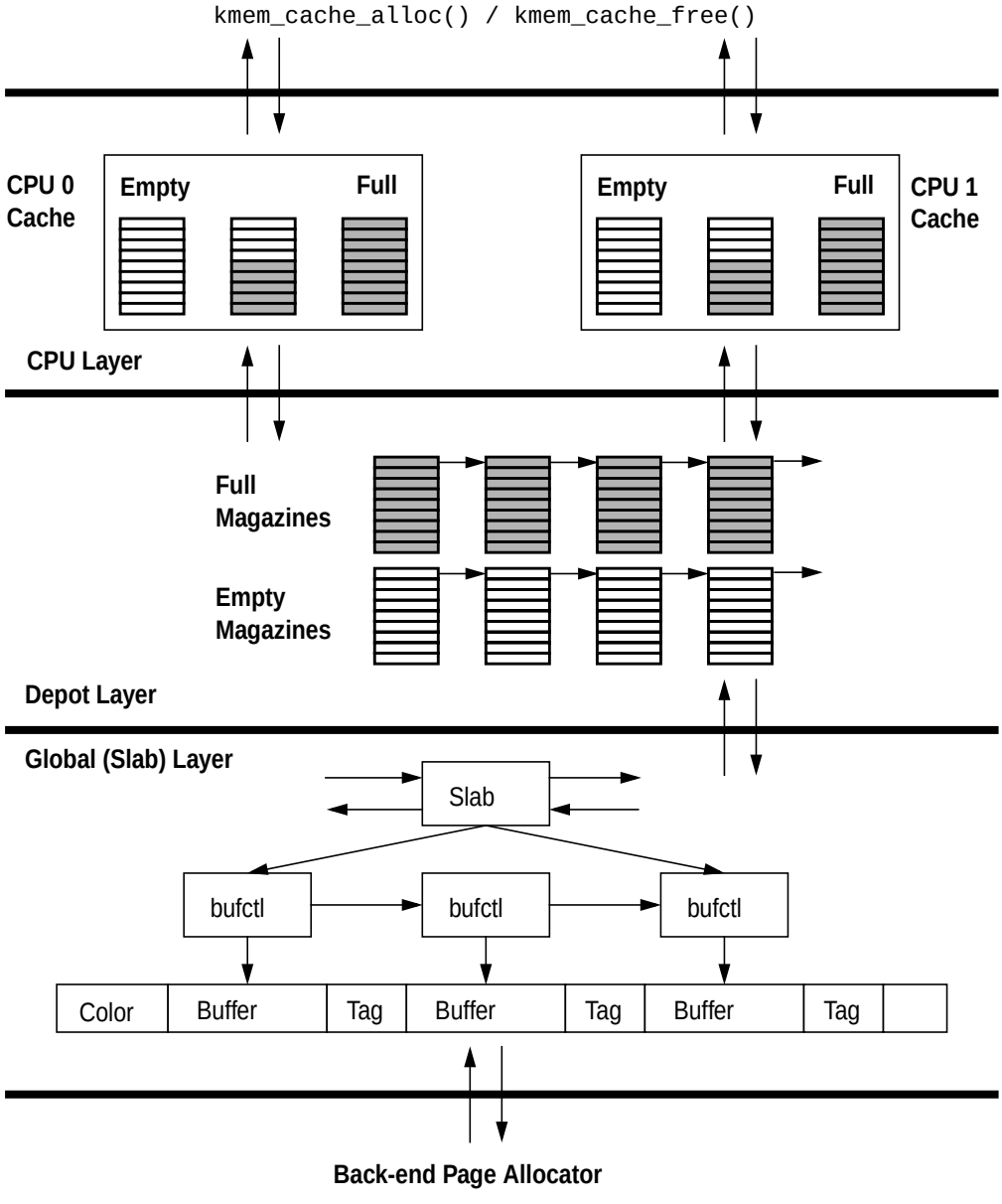

Figure 6.5 Slab Allocator Internal Implementation

The layers shown in Figure 6.5 are separated into the slab layer, the depot layer, and the CPU layer. The upper two layers (which together are known as the magazine layer) are caches of allocated groups of objects and use a military analogy of allocating rifle rounds from magazines. Each per-CPU cache has magazines of allocated objects and can allocate objects (rounds) from its own magazines without having to bother the lower layers. The CPU layer needs to allocate objects from the lower (depot) layer only when its magazines are empty. The depot layer refills magazines from the slab layer by assembling objects, which may reside in many different slabs, into full magazines.

6.2.4.5 The CPU Layer

The CPU layer caches groups of objects to minimize the number of times that an allocation will need to go down to the lower layers. This means that we can satisfy the majority of allocation requests without having to hold any global locks, thus dramatically improving the scalability of the allocator.

Continuing the military analogy: three magazines of objects are kept in the CPU layer to satisfy allocation and deallocation requests—a full, a half-allocated, and an empty magazine are on hand. Objects are allocated from the half-empty magazine, and until the magazine is empty, all allocations are simply satisfied from the magazine. When the magazine empties, an empty magazine is returned to the magazine layer, and objects are allocated from the full magazine that was already available at the CPU layer. The CPU layer keeps the empty and full magazine on hand to prevent the magazine layer from having to construct and deconstruct magazines when on a full or empty magazine boundary. If a client rapidly allocates and deallocates objects when the magazine is on a boundary, then the CPU layer can simply use its full and empty magazines to service the requests, rather than having the magazine layer deconstruct and reconstruct new magazines at each request. The magazine model allows the allocator to guarantee that it can satisfy at least a magazine size of rounds without having to go to the depot layer.

6.2.4.6 The Depot Layer

The depot layer assembles groups of objects into magazines. Unlike a slab, a magazine's objects are not necessarily allocated from contiguous memory; rather, a magazine contains a series of pointers to objects within slabs.

The number of rounds per magazine for each cache changes dynamically, depending on the amount of contention that occurs at the depot layer. The more rounds per magazine, the lower the depot contention, but more memory is consumed. Each range of object sizes has an upper and lower magazine size. Table 6-11 shows the magazine size range for each object size.

Table 6-11 Magazine Sizes

Object Size Range	Minimum Magazine Size	Maximum Magazine Size
0–63	15	143
64–127	7	95
128–255	3	47
256–511	1	31
512–1023	1	15
1024–2047	1	7
2048–16383	1	3
16384–	1	1

A slab allocator maintenance thread is scheduled every 15 seconds (controlled by the tunable `kmem_update_interval`) to recalculate the magazine sizes. If significant contention has occurred at the depot level, then the magazine size is bumped up. Refer to Table 6-12 on page 269 for the parameters that control magazine resizing.

6.2.4.7 The Global (Slab) Layer

The global slab layer allocates slabs of objects from contiguous pages of physical memory and hands them up to the magazine layer for allocation. The global slab layer is used only when the upper layers need to allocate or deallocate entire slabs of objects to refill their magazines.

The slab is the primary unit of allocation in the slab layer. When the allocator needs to grow a cache, it acquires an entire slab of objects. When the allocator wants to shrink a cache, it returns unused memory to the back end by deallocating a complete slab. A slab consists of one or more pages of virtually contiguous memory carved up into equal-sized chunks, with a reference count indicating how many of those chunks have been allocated.

The contents of each slab are managed by a `kmem_slab` data structure that maintains the slab's linkage in the cache, its reference count, and its list of free buffers. In turn, each buffer in the slab is managed by a `kmem_bufctl` structure that holds the freelist linkage, the buffer address, and a back-pointer to the controlling slab.

For objects smaller than 1/8th of a page, the slab allocator builds a slab by allocating a page, placing the slab data at the end, and dividing the rest into equal-sized buffers. Each buffer serves as its own `kmem_bufctl` while on the freelist. Only the linkage is actually needed, since everything else is computable. These are essential optimizations for small buffers; otherwise, we would end up allocating almost as much memory for `kmem_bufctl` as for the buffers themselves. The free-list linkage resides at the end of the buffer, rather than the beginning, to facilitate debugging. This location is driven by the empirical observation that the beginning of a data structure is typically more active than the end. If a

buffer is modified after being freed, the problem is easier to diagnose if the heap structure (free-list linkage) is still intact. The allocator reserves an additional word for constructed objects so that the linkage does not overwrite any constructed state.

For objects greater than 1/8th of a page, a different scheme is used. Allocating objects from within a page-sized slab is efficient for small objects but not for large ones. The reason for the inefficiency of large-object allocation is that we could fit only one 4-Kbyte buffer on an 8-Kbyte page—the embedded slab control data takes up a few bytes, and two 4-Kbyte buffers would need just over 8 Kbytes. For large objects, we allocate a separate slab management structure from a separate pool of memory (another slab allocator cache, the `kmem_slab_cache`). We also allocate a buffer control structure for each page in the cache from another cache, the `kmem_bufctl_cache`. The `slab`/`bufctl`/`buffer` structures are shown in the slab layer in Figure 6.5 on page 266.

The slab layer solves another common memory allocation problem by implementing slab coloring. If memory objects all start at a common offset (e.g., at 512-byte boundaries), then accessing data at the start of each object could result in the same cache line being used for all of the objects. The issues are similar to those discussed in "The Page Scanner" on page 220. To overcome the cache line problem, the allocator applies an offset to the start of each slab, so that buffers within the slab start at a different offset. This approach is also shown in Figure 6.5 on page 266 by the color offset segment that resides at the start of each memory allocation unit before the actual buffer. Slab coloring results in much better cache utilization and more evenly balanced memory loading.

6.2.4.8 Slab Cache Parameters

The slab allocator parameters are shown in Table 6-12 for reference only. We recommend that none of these values be changed.

Table 6-12 Kernel Memory Allocator Parameters

Parameter	Description	2.7 Def.
`kmem_reap_interval`	The number of ticks after which the slab allocator update thread will run.	15000 (15s)
`kmem_depot_contention`	If the number of times depot contention occurred since the last time the update thread ran is greater than this value, then the magazine size is increased.	3
`kmem_reapahead`	If the amount of free memory falls below `cachefree + kmem_reapahead`, then the slab allocator will give back as many slabs as possible to the back-end page allocator.	0

6.2.4.9 Slab Allocator Statistics

Two forms of slab allocator statistics are available: global statistics and per-cache statistics. The global statistics are available through the `crash` utility and display a summary of the entire cache list managed by the allocator.

```
# crash
dumpfile = /dev/mem, namelist = /dev/ksyms, outfile = stdout
> kmastat
                         buf    buf    buf    memory     #allocations
cache name              size avail total    in use     succeed fail
----------             ----- ----- ----- --------     ------- ----
kmem_magazine_1           16   483   508     8192        6664    0   Magazine
kmem_magazine_3           32  1123  1270    40960       55225    0   Caches
kmem_magazine_7           64   584   762    49152       62794    0
kmem_magazine_15         128   709   945   122880      194764    0
kmem_magazine_31         256    58    62    16384       24915    0
kmem_magazine_47         384     0     0        0           0    0
kmem_magazine_63         512     0     0        0           0    0
kmem_magazine_95         768     0     0        0           0    0
kmem_magazine_143       1152     0     0        0           0    0   Slab &
kmem_slab_cache           56   308  2159   139264       22146    0   Bufctl
kmem_bufctl_cache         32  2129  6096   196608       54870    0   Caches
kmem_bufctl_audit_cache  184    24 16464  3211264       16440    0
kmem_pagectl_cache        32   102   254     8192      406134    0
kmem_alloc_8               8  9888 31527   253952   115432346    0
kmem_alloc_16             16  7642 18288   294912   374733170    0
kmem_alloc_24             24  4432 11187   270336    30957233    0   kmem_alloc()
        .                                                           caches
        .
kmem_alloc_12288       12288     2     4    49152         660    0
kmem_alloc_16384       16384     0    42   688128        1845    0
        .
        .                                                           Streams
streams_mblk              64  3988  5969   385024    31405446    0   Data Blocks
streams_dblk_32          128   795  1134   147456    72553829    0
streams_dblk_64          160   716  1650   270336   196660790    0
        .
        .
streams_dblk_8096       8192    17    17   139264   356266482    0
streams_dblk_12192     12288     8     8    98304    14848223    0
streams_dblk_esb          96     0     0        0      406326    0
stream_head_cache        328    68   648   221184      492256    0
queue_cache              456   109  1513   729088     1237000    0
syncq_cache              120    48    67     8192         373    0
qband_cache               64   125   635    40960        1303    0
linkinfo_cache            48   156   169     8192          90    0
strevent_cache            48   153   169     8192     5442622    0
as_cache                 120    45   201    24576      158778    0
seg_skiplist_cache        32   540  1524    49152     1151455    0
anon_cache                48  1055 71825  3481600     7926946    0
anonmap_cache             48   551  4563   221184     5805027    0
segvn_cache               88   686  6992   622592     9969087    0
flk_edges                 48     0     0        0           1    0
physio_buf_cache         224     0     0        0    98535107    0
snode_cache              240    39   594   147456     1457746    0
ufs_inode_cache          440  8304 32868 14958592    20249920    0
```

The `kmastat` command shows summary information for each statistic and a systemwide summary at the end. The columns are shown in Table 6-13.

Table 6-13 kmastat Columns

Parameter	Description
Cache name	The name of the cache, as supplied during `kmem_cache_create()`.
`buf_size`	The size of each object within the cache in bytes.
`buf_avail`	The number of free objects in the cache.
`buf_total`	The total number of objects in the cache.
Memory in use	The amount of physical memory consumed by the cache in bytes.
Allocations succeeded	The number of allocations that succeeded.
Allocations failed	The number of allocations that failed. These are likely to be allocations that specified `KM_NOSLEEP` during memory pressure.

A more detailed version of the per-cache statistics is exported by the `kstat` mechanism. You can use the `netstat -k` command to display the cache statistics, which are described in Table 6-14.

```
# netstat -k ufs_inode_cache
ufs_inode_cache:
buf_size 440 align 8 chunk_size 440 slab_size 8192 alloc 20248589
alloc_fail 0 free 20770500 depot_alloc 657344 depot_free 678433
depot_contention 85 global_alloc 602986 global_free 578089
buf_constructed 0 buf_avail 7971 buf_inuse 24897 buf_total 32868
buf_max 41076 slab_create 2802 slab_destroy 976 memory_class 0
hash_size 0 hash_lookup_depth 0 hash_rescale 0 full_magazines 0
empty_magazines 0 magazine_size 31 alloc_from_cpu0 9583811
free_to_cpu0 10344474 buf_avail_cpu0 0 alloc_from_cpu1 9404448
free_to_cpu1 9169504 buf_avail_cpu1 0
```

Table 6-14 Slab Allocator Per-Cache Statistics

Parameter	Description
`buf_size`	The size of each object within the cache in bytes.
`align`	The alignment boundary for objects within the cache.
`chunk_size`	The allocation unit for the cache in bytes.
`slab_size`	The size of each slab within the cache in bytes.
`alloc`	The number of object allocations that succeeded.
`alloc_fail`	The number of object allocations that failed. (Should be zero!).

Table 6-14 Slab Allocator Per-Cache Statistics (Continued)

Parameter	Description
free	The number of objects that were freed.
depot_alloc	The number of times a magazine was allocated in the depot layer.
depot_free	The number of times a magazine was freed to the depot layer.
depot_contention	The number of times a depot layer allocation was blocked because another thread was in the depot layer.
global_alloc	The number of times an allocation was made at the global layer.
global_free	The number of times an allocation was freed at the global layer.
buf_constructed	Zero or the same as buf_avail.
buf_avail	The number of free objects in the cache.
buf_inuse	The number of objects used by the client.
buf_total	The total number of objects in the cache.
buf_max	The maximum number of objects the cache has reached.
slab_create	The number of slabs created.
slab_destroy	The number of slabs destroyed.
memory_class	The ID of the back-end memory allocator.
hash_size	Buffer hash lookup statistics.
hash_lookup_depth	Buffer hash lookup statistics.
hash_rescale	Buffer hash lookup statistics.
full_magazines	The number of full magazines.
empty_magazines	The number of empty magazines.
magazine_size	The size of the magazine.
alloc_from_cpu*N*	Object allocations from CPU *N*.
free_to_cpu*N*	Objects freed to CPU *N*.
buf_avail_cpu*N*	Objects available to CPU *N*.

6.2.4.10 Slab Allocator Tracing

The slab allocator includes a general-purpose allocation tracing facility that tracks the allocation history of objects. The facility is switched off by default and can be enabled by setting of the system variable kmem_flags. The tracing facility captures the stack and history of allocations into a slab cache, named as the name of the cache being traced, with .DEBUG appended to it. Audit tracing can be enabled by the following:

- Setting `kmem_flags` to indicate the type of tracing desired, usually 0x1F to indicate all tracing
- Booting the system with `kadb -d` and setting `kmem_flags` before startup

The following simple example shows how to trace a cache that is created on a large system, after the flags have been set. To enable tracing on all caches, the system must be booted with `kadb` and the `kmem_flags` variable set. The steps for such booting are shown below.

```
ok boot kadb -d
Resetting ...

Sun Ultra 1 UPA/SBus (UltraSPARC 167MHz), No Keyboard
OpenBoot 3.1, 128 MB memory installed, Serial #8788108.
Ethernet address 8:0:20:86:18:8c, Host ID: 8086188c.


Rebooting with command: boot kadb -d
Boot device: /sbus/SUNW,fas@e,8800000/sd@0,0  File and args: kadb -d
kadb: <return>
kadb[0]: kmem_flags/D
kmem_flags:
kmem_flags:     0
kadb[0]: kmem_flags/W 0x1f
kmem_flags:     0x0             =       0x1f
kadb[0]: :c

SunOS Release 5.7 Version Generic 64-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved.
\
```

Note that the total number of allocations traced will be limited by the size of the audit cache parameters, shown in Table 6-12 on page 269. Table 6-15 shows the parameters that control kernel memory debugging.

Table 6-15 Kernel Memory Debugging Parameters

Parameter	Description	2.7 Def.
`kmem_flags`	Set this to select the mode of kernel memory debugging. Set to 0x1F to enable all debugging, or set the logical AND of the following: 0x1 Transaction auditing 0x2 deadbeef checking 0x4 red-zone checking 0x8 freed buffer content logging	0
`kmem_log_size`	Maximum amount of memory to use for slab allocator audit tracing.	2% of mem.
`kmem_content_maxsave`	The maximum number of bytes to log in each entry.	256