CHAPTER 8

Allocating Memory

Thus far, we have used *kmalloc* and *kfree* for the allocation and freeing of memory. The Linux kernel offers a richer set of memory allocation primitives, however. In this chapter, we look at other ways of using memory in device drivers and how to optimize your system's memory resources. We do not get into how the different architectures actually administer memory. Modules are not involved in issues of segmentation, paging, and so on, since the kernel offers a unified memory management interface to the drivers. In addition, we won't describe the internal details of memory management in this chapter, but defer it to Chapter 15.

The Real Story of kmalloc

The *kmalloc* allocation engine is a powerful tool and easily learned because of its similarity to *malloc*. The function is fast (unless it blocks) and doesn't clear the memory it obtains; the allocated region still holds its previous content.* The allocated region is also contiguous in physical memory. In the next few sections, we talk in detail about *kmalloc*, so you can compare it with the memory allocation techniques that we discuss later.

The Flags Argument

Remember that the prototype for *kmalloc* is:

```
#include <linux/slab.h>

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

* Among other things, this implies that you should explicitly clear any memory that might be exposed to user space or written to a device; otherwise, you risk disclosing information that should be kept private.

The first argument to *kmalloc* is the size of the block to be allocated. The second argument, the allocation flags, is much more interesting, because it controls the behavior of *kmalloc* in a number of ways.

The most commonly used flag, GFP_KERNEL, means that the allocation (internally performed by calling, eventually, *__get_free_pages*, which is the source of the GFP_ prefix) is performed on behalf of a process running in kernel space. In other words, this means that the calling function is executing a system call on behalf of a process. Using GFP_KERNEL means that *kmalloc* can put the current process to sleep waiting for a page when called in low-memory situations. A function that allocates memory using GFP_KERNEL must, therefore, be reentrant and cannot be running in atomic context. While the current process sleeps, the kernel takes proper action to locate some free memory, either by flushing buffers to disk or by swapping out memory from a user process.

GFP_KERNEL isn't always the right allocation flag to use; sometimes *kmalloc* is called from outside a process's context. This type of call can happen, for instance, in interrupt handlers, tasklets, and kernel timers. In this case, the current process should not be put to sleep, and the driver should use a flag of GFP_ATOMIC instead. The kernel normally tries to keep some free pages around in order to fulfill atomic allocation. When GFP_ATOMIC is used, *kmalloc* can use even the last free page. If that last page does not exist, however, the allocation fails.

Other flags can be used in place of or in addition to GFP_KERNEL and GFP_ATOMIC, although those two cover most of the needs of device drivers. All the flags are defined in *<linux/gfp.h>*, and individual flags are prefixed with a double underscore, such as __GFP_DMA. In addition, there are symbols that represent frequently used combinations of flags; these lack the prefix and are sometimes called *allocation priorities*. The latter include:

GFP_ATOMIC
: Used to allocate memory from interrupt handlers and other code outside of a process context. Never sleeps.

GFP_KERNEL
: Normal allocation of kernel memory. May sleep.

GFP_USER
: Used to allocate memory for user-space pages; it may sleep.

GFP_HIGHUSER
: Like GFP_USER, but allocates from high memory, if any. High memory is described in the next subsection.

GFP_NOIO
GFP_NOFS
: These flags function like GFP_KERNEL, but they add restrictions on what the kernel can do to satisfy the request. A GFP_NOFS allocation is not allowed to perform

any filesystem calls, while GFP_NOIO disallows the initiation of any I/O at all. They are used primarily in the filesystem and virtual memory code where an allocation may be allowed to sleep, but recursive filesystem calls would be a bad idea.

The allocation flags listed above can be augmented by an ORing in any of the following flags, which change how the allocation is carried out:

__GFP_DMA
: This flag requests allocation to happen in the DMA-capable memory zone. The exact meaning is platform-dependent and is explained in the following section.

__GFP_HIGHMEM
: This flag indicates that the allocated memory may be located in high memory.

__GFP_COLD
: Normally, the memory allocator tries to return "cache warm" pages—pages that are likely to be found in the processor cache. Instead, this flag requests a "cold" page, which has not been used in some time. It is useful for allocating pages for DMA reads, where presence in the processor cache is not useful. See the section "Direct Memory Access" in Chapter 1 for a full discussion of how to allocate DMA buffers.

__GFP_NOWARN
: This rarely used flag prevents the kernel from issuing warnings (with *printk*) when an allocation cannot be satisfied.

__GFP_HIGH
: This flag marks a high-priority request, which is allowed to consume even the last pages of memory set aside by the kernel for emergencies.

__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
: These flags modify how the allocator behaves when it has difficulty satisfying an allocation. __GFP_REPEAT means "try a little harder" by repeating the attempt—but the allocation can still fail. The __GFP_NOFAIL flag tells the allocator never to fail; it works as hard as needed to satisfy the request. Use of __GFP_NOFAIL is very strongly discouraged; there will probably never be a valid reason to use it in a device driver. Finally, __GFP_NORETRY tells the allocator to give up immediately if the requested memory is not available.

Memory zones

Both __GFP_DMA and __GFP_HIGHMEM have a platform-dependent role, although their use is valid for all platforms.

The Linux kernel knows about a minimum of three *memory zones*: DMA-capable memory, normal memory, and high memory. While allocation normally happens in

the *normal* zone, setting either of the bits just mentioned requires memory to be allocated from a different zone. The idea is that every computer platform that must know about special memory ranges (instead of considering all RAM equivalents) will fall into this abstraction.

DMA-capable memory is memory that lives in a preferential address range, where peripherals can perform DMA access. On most sane platforms, all memory lives in this zone. On the x86, the DMA zone is used for the first 16 MB of RAM, where legacy ISA devices can perform DMA; PCI devices have no such limit.

High memory is a mechanism used to allow access to (relatively) large amounts of memory on 32-bit platforms. This memory cannot be directly accessed from the kernel without first setting up a special mapping and is generally harder to work with. If your driver uses large amounts of memory, however, it will work better on large systems if it can use high memory. See the section "High and Low Memory" in Chapter 1 for a detailed description of how high memory works and how to use it.

Whenever a new page is allocated to fulfill a memory allocation request, the kernel builds a list of zones that can be used in the search. If `__GFP_DMA` is specified, only the DMA zone is searched: if no memory is available at low addresses, allocation fails. If no special flag is present, both normal and DMA memory are searched; if `__GFP_HIGHMEM` is set, all three zones are used to search a free page. (Note, however, that *kmalloc* cannot allocate high memory.)

The situation is more complicated on nonuniform memory access (NUMA) systems. As a general rule, the allocator attempts to locate memory local to the processor performing the allocation, although there are ways of changing that behavior.

The mechanism behind memory zones is implemented in *mm/page_alloc.c*, while initialization of the zone resides in platform-specific files, usually in *mm/init.c* within the *arch* tree. We'll revisit these topics in Chapter 15.

The Size Argument

The kernel manages the system's *physical* memory, which is available only in page-sized chunks. As a result, *kmalloc* looks rather different from a typical user-space *malloc* implementation. A simple, heap-oriented allocation technique would quickly run into trouble; it would have a hard time working around the page boundaries. Thus, the kernel uses a special page-oriented allocation technique to get the best use from the system's RAM.

Linux handles memory allocation by creating a set of pools of memory objects of fixed sizes. Allocation requests are handled by going to a pool that holds sufficiently large objects and handing an entire memory chunk back to the requester. The memory management scheme is quite complex, and the details of it are not normally all that interesting to device driver writers.

The one thing driver developers should keep in mind, though, is that the kernel can allocate only certain predefined, fixed-size byte arrays. If you ask for an arbitrary amount of memory, you're likely to get slightly more than you asked for, up to twice as much. Also, programmers should remember that the smallest allocation that *kmalloc* can handle is as big as 32 or 64 bytes, depending on the page size used by the system's architecture.

There is an upper limit to the size of memory chunks that can be allocated by *kmalloc*. That limit varies depending on architecture and kernel configuration options. If your code is to be completely portable, it cannot count on being able to allocate anything larger than 128 KB. If you need more than a few kilobytes, however, there are better ways than *kmalloc* to obtain memory, which we describe later in this chapter.

Lookaside Caches

A device driver often ends up allocating many objects of the same size, over and over. Given that the kernel already maintains a set of memory pools of objects that are all the same size, why not add some special pools for these high-volume objects? In fact, the kernel does implement a facility to create this sort of pool, which is often called a *lookaside cache*. Device drivers normally do not exhibit the sort of memory behavior that justifies using a lookaside cache, but there can be exceptions; the USB and SCSI drivers in Linux 2.6 use caches.

The cache manager in the Linux kernel is sometimes called the "slab allocator." For that reason, its functions and types are declared in *<linux/slab.h>*. The slab allocator implements caches that have a type of `kmem_cache_t`; they are created with a call to *kmem_cache_create*:

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size,
                                size_t offset,
                                unsigned long flags,
                                void (*constructor)(void *, kmem_cache_t *,
                                                    unsigned long flags),
                                void (*destructor)(void *, kmem_cache_t *,
                                                   unsigned long flags));
```

The function creates a new cache object that can host any number of memory areas all of the same size, specified by the `size` argument. The `name` argument is associated with this cache and functions as housekeeping information usable in tracking problems; usually, it is set to the name of the type of structure that is cached. The cache keeps a pointer to the name, rather than copying it, so the driver should pass in a pointer to a name in static storage (usually the name is just a literal string). The name cannot contain blanks.

The `offset` is the offset of the first object in the page; it can be used to ensure a particular alignment for the allocated objects, but you most likely will use `0` to request

the default value. `flags` controls how allocation is done and is a bit mask of the following flags:

`SLAB_NO_REAP`
: Setting this flag protects the cache from being reduced when the system is looking for memory. Setting this flag is normally a bad idea; it is important to avoid restricting the memory allocator's freedom of action unnecessarily.

`SLAB_HWCACHE_ALIGN`
: This flag requires each data object to be aligned to a cache line; actual alignment depends on the cache layout of the host platform. This option can be a good choice if your cache contains items that are frequently accessed on SMP machines. The padding required to achieve cache line alignment can end up wasting significant amounts of memory, however.

`SLAB_CACHE_DMA`
: This flag requires each data object to be allocated in the DMA memory zone.

There is also a set of flags that can be used during the debugging of cache allocations; see *mm/slab.c* for the details. Usually, however, these flags are set globally via a kernel configuration option on systems used for development.

The `constructor` and `destructor` arguments to the function are optional functions (but there can be no destructor without a constructor); the former can be used to initialize newly allocated objects, and the latter can be used to "clean up" objects prior to their memory being released back to the system as a whole.

Constructors and destructors can be useful, but there are a few constraints that you should keep in mind. A constructor is called when the memory for a set of objects is allocated; because that memory may hold several objects, the constructor may be called multiple times. You cannot assume that the constructor will be called as an immediate effect of allocating an object. Similarly, destructors can be called at some unknown future time, not immediately after an object has been freed. Constructors and destructors may or may not be allowed to sleep, according to whether they are passed the `SLAB_CTOR_ATOMIC` flag (where `CTOR` is short for *constructor*).

For convenience, a programmer can use the same function for both the constructor and destructor; the slab allocator always passes the `SLAB_CTOR_CONSTRUCTOR` flag when the callee is a constructor.

Once a cache of objects is created, you can allocate objects from it by calling *kmem_cache_alloc*:

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

Here, the `cache` argument is the cache you have created previously; the flags are the same as you would pass to *kmalloc* and are consulted if *kmem_cache_alloc* needs to go out and allocate more memory itself.

To free an object, use *kmem_cache_free*:

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

When driver code is finished with the cache, typically when the module is unloaded, it should free its cache as follows:

```
int kmem_cache_destroy(kmem_cache_t *cache);
```

The destroy operation succeeds only if all objects allocated from the cache have been returned to it. Therefore, a module should check the return status from *kmem_cache_destroy*; a failure indicates some sort of memory leak within the module (since some of the objects have been dropped).

One side benefit to using lookaside caches is that the kernel maintains statistics on cache usage. These statistics may be obtained from */proc/slabinfo*.

A scull Based on the Slab Caches: scullc

Time for an example. *scullc* is a cut-down version of the *scull* module that implements only the bare device—the persistent memory region. Unlike *scull*, which uses *kmalloc*, *scullc* uses memory caches. The size of the quantum can be modified at compile time and at load time, but not at runtime—that would require creating a new memory cache, and we didn't want to deal with these unneeded details.

scullc is a complete example that can be used to try out the slab allocator. It differs from *scull* only in a few lines of code. First, we must declare our own slab cache:

```
/* declare one cache pointer: use it for all devices */
kmem_cache_t *scullc_cache;
```

The creation of the slab cache is handled (at module load time) in this way:

```
/* scullc_init: create a cache for our quanta */
scullc_cache = kmem_cache_create("scullc", scullc_quantum,
        0, SLAB_HWCACHE_ALIGN, NULL, NULL); /* no ctor/dtor */
if (!scullc_cache) {
    scullc_cleanup( );
    return -ENOMEM;
}
```

This is how it allocates memory quanta:

```
/* Allocate a quantum using the memory cache */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmem_cache_alloc(scullc_cache, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, scullc_quantum);
}
```

And these lines release memory:

```
for (i = 0; i < qset; i++)
if (dptr->data[i])
        kmem_cache_free(scullc_cache, dptr->data[i]);
```

Finally, at module unload time, we have to return the cache to the system:

```
/* scullc_cleanup: release the cache of our quanta */
if (scullc_cache)
    kmem_cache_destroy(scullc_cache);
```

The main differences in passing from *scull* to *scullc* are a slight speed improvement and better memory use. Since quanta are allocated from a pool of memory fragments of exactly the right size, their placement in memory is as dense as possible, as opposed to *scull* quanta, which bring in an unpredictable memory fragmentation.

Memory Pools

There are places in the kernel where memory allocations cannot be allowed to fail. As a way of guaranteeing allocations in those situations, the kernel developers created an abstraction known as a *memory pool* (or "mempool"). A memory pool is really just a form of a lookaside cache that tries to always keep a list of free memory around for use in emergencies.

A memory pool has a type of `mempool_t` (defined in *<linux/mempool.h>*); you can create one with *mempool_create*:

```
mempool_t *mempool_create(int min_nr,
                          mempool_alloc_t *alloc_fn,
                          mempool_free_t *free_fn,
                          void *pool_data);
```

The `min_nr` argument is the minimum number of allocated objects that the pool should always keep around. The actual allocation and freeing of objects is handled by `alloc_fn` and `free_fn`, which have these prototypes:

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

The final parameter to *mempool_create* (`pool_data`) is passed to `alloc_fn` and `free_fn`.

If need be, you can write special-purpose functions to handle memory allocations for mempools. Usually, however, you just want to let the kernel slab allocator handle that task for you. There are two functions (*mempool_alloc_slab* and *mempool_free_slab*) that perform the impedance matching between the memory pool allocation prototypes and *kmem_cache_alloc* and *kmem_cache_free*. Thus, code that sets up memory pools often looks like the following:

```
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM,
                      mempool_alloc_slab, mempool_free_slab,
                      cache);
```

Once the pool has been created, objects can be allocated and freed with:

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

When the mempool is created, the allocation function will be called enough times to create a pool of preallocated objects. Thereafter, calls to *mempool_alloc* attempt to acquire additional objects from the allocation function; should that allocation fail, one of the preallocated objects (if any remain) is returned. When an object is freed with *mempool_free*, it is kept in the pool if the number of preallocated objects is currently below the minimum; otherwise, it is to be returned to the system.

A mempool can be resized with:

```
int mempool_resize(mempool_t *pool, int new_min_nr, int gfp_mask);
```

This call, if successful, resizes the pool to have at least `new_min_nr` objects.

If you no longer need a memory pool, return it to the system with:

```
void mempool_destroy(mempool_t *pool);
```

You must return all allocated objects before destroying the mempool, or a kernel oops results.

If you are considering using a mempool in your driver, please keep one thing in mind: mempools allocate a chunk of memory that sits in a list, idle and unavailable for any real use. It is easy to consume a great deal of memory with mempools. In almost every case, the preferred alternative is to do without the mempool and simply deal with the possibility of allocation failures instead. If there is any way for your driver to respond to an allocation failure in a way that does not endanger the integrity of the system, do things that way. Use of mempools in driver code should be rare.

get_free_page and Friends

If a module needs to allocate big chunks of memory, it is usually better to use a page-oriented technique. Requesting whole pages also has other advantages, which are introduced in Chapter 15.

To allocate pages, the following functions are available:

`get_zeroed_page(unsigned int flags);`
: Returns a pointer to a new page and fills the page with zeros.

`__get_free_page(unsigned int flags);`
: Similar to *get_zeroed_page*, but doesn't clear the page.

`__get_free_pages(unsigned int flags, unsigned int order);`
: Allocates and returns a pointer to the first byte of a memory area that is potentially several (physically contiguous) pages long but doesn't zero the area.

The `flags` argument works in the same way as with *kmalloc*; usually either `GFP_KERNEL` or `GFP_ATOMIC` is used, perhaps with the addition of the `__GFP_DMA` flag (for memory that can be used for ISA direct-memory-access operations) or `__GFP_HIGHMEM` when

high memory can be used.* `order` is the base-two logarithm of the number of pages you are requesting or freeing (i.e., log2*N*). For example, `order` is `0` if you want one page and `3` if you request eight pages. If `order` is too big (no contiguous area of that size is available), the page allocation fails. The *get_order* function, which takes an integer argument, can be used to extract the order from a size (that must be a power of two) for the hosting platform. The maximum allowed value for `order` is `10` or `11` (corresponding to 1024 or 2048 pages), depending on the architecture. The chances of an order-10 allocation succeeding on anything other than a freshly booted system with a lot of memory are small, however.

If you are curious, */proc/buddyinfo* tells you how many blocks of each order are available for each memory zone on the system.

When a program is done with the pages, it can free them with one of the following functions. The first function is a macro that falls back on the second:

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

If you try to free a different number of pages from what you allocated, the memory map becomes corrupted, and the system gets in trouble at a later time.

It's worth stressing that *__get_free_pages* and the other functions can be called at any time, subject to the same rules we saw for *kmalloc*. The functions can fail to allocate memory in certain circumstances, particularly when `GFP_ATOMIC` is used. Therefore, the program calling these allocation functions must be prepared to handle an allocation failure.

Although `kmalloc(GFP_KERNEL)` sometimes fails when there is no available memory, the kernel does its best to fulfill allocation requests. Therefore, it's easy to degrade system responsiveness by allocating too much memory. For example, you can bring the computer down by pushing too much data into a *scull* device; the system starts crawling while it tries to swap out as much as possible in order to fulfill the *kmalloc* request. Since every resource is being sucked up by the growing device, the computer is soon rendered unusable; at that point, you can no longer even start a new process to try to deal with the problem. We don't address this issue in *scull*, since it is just a sample module and not a real tool to put into a multiuser system. As a programmer, you must be careful nonetheless, because a module is privileged code and can open new security holes in the system (the most likely is a denial-of-service hole like the one just outlined).

* Although *alloc_pages* (described shortly) should really be used for allocating high-memory pages, for reasons we can't really get into until Chapter 15.

A scull Using Whole Pages: scullp

In order to test page allocation for real, we have released the *scullp* module together with other sample code. It is a reduced *scull*, just like *scullc* introduced earlier.

Memory quanta allocated by *scullp* are whole pages or page sets: the `scullp_order` variable defaults to `0` but can be changed at either compile or load time.

The following lines show how it allocates memory:

```
/* Here's the allocation of a single quantum */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)__get_free_pages(GFP_KERNEL, dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

The code to deallocate memory in *scullp* looks like this:

```
/* This code frees a whole quantum-set */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        free_pages((unsigned long)(dptr->data[i]),
                dptr->order);
```

At the user level, the perceived difference is primarily a speed improvement and better memory use, because there is no internal fragmentation of memory. We ran some tests copying 4 MB from *scull0* to *scull1* and then from *scullp0* to *scullp1*; the results showed a slight improvement in kernel-space processor usage.

The performance improvement is not dramatic, because *kmalloc* is designed to be fast. The main advantage of page-level allocation isn't actually speed, but rather more efficient memory usage. Allocating by pages wastes no memory, whereas using *kmalloc* wastes an unpredictable amount of memory because of allocation granularity.

But the biggest advantage of the *__get_free_page* functions is that the pages obtained are completely yours, and you could, in theory, assemble the pages into a linear area by appropriate tweaking of the page tables. For example, you can allow a user process to *mmap* memory areas obtained as single unrelated pages. We discuss this kind of operation in Chapter 15, where we show how *scullp* offers memory mapping, something that *scull* cannot offer.

The alloc_pages Interface

For completeness, we introduce another interface for memory allocation, even though we will not be prepared to use it until after Chapter 15. For now, suffice it to say that `struct page` is an internal kernel structure that describes a page of memory. As we will see, there are many places in the kernel where it is necessary to work with

page structures; they are especially useful in any situation where you might be dealing with high memory, which does not have a constant address in kernel space.

The *real* core of the Linux page allocator is a function called *alloc_pages_node*:

```
struct page *alloc_pages_node(int nid, unsigned int flags,
                              unsigned int order);
```

This function also has two variants (which are simply macros); these are the versions that you will most likely use:

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

The core function, *alloc_pages_node*, takes three arguments. `nid` is the NUMA node ID* whose memory should be allocated, `flags` is the usual `GFP_` allocation flags, and `order` is the size of the allocation. The return value is a pointer to the first of (possibly many) `page` structures describing the allocated memory, or, as usual, `NULL` on failure.

alloc_pages simplifies the situation by allocating the memory on the current NUMA node (it calls *alloc_pages_node* with the return value from *numa_node_id* as the `nid` parameter). And, of course, *alloc_page* omits the `order` parameter and allocates a single page.

To release pages allocated in this manner, you should use one of the following:

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

If you have specific knowledge of whether a single page's contents are likely to be resident in the processor cache, you should communicate that to the kernel with *free_hot_page* (for cache-resident pages) or *free_cold_page*. This information helps the memory allocator optimize its use of memory across the system.

vmalloc and Friends

The next memory allocation function that we show you is *vmalloc*, which allocates a contiguous memory region in the *virtual* address space. Although the pages are not consecutive in physical memory (each page is retrieved with a separate call to *alloc_page*), the kernel sees them as a contiguous range of addresses. *vmalloc* returns `0` (the `NULL` address) if an error occurs, otherwise, it returns a pointer to a linear memory area of size at least `size`.

* NUMA (nonuniform memory access) computers are multiprocessor systems where memory is "local" to specific groups of processors ("nodes"). Access to local memory is faster than access to nonlocal memory. On such systems, allocating memory on the correct node is important. Driver authors do not normally have to worry about NUMA issues, however.

We describe *vmalloc* here because it is one of the fundamental Linux memory allocation mechanisms. We should note, however, that use of *vmalloc* is discouraged in most situations. Memory obtained from *vmalloc* is slightly less efficient to work with, and, on some architectures, the amount of address space set aside for *vmalloc* is relatively small. Code that uses *vmalloc* is likely to get a chilly reception if submitted for inclusion in the kernel. If possible, you should work directly with individual pages rather than trying to smooth things over with *vmalloc*.

That said, let's see how *vmalloc* works. The prototypes of the function and its relatives (*ioremap*, which is not strictly an allocation function, is discussed later in this section) are as follows:

```
#include <linux/vmalloc.h>

void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

It's worth stressing that memory addresses returned by *kmalloc* and *_get_free_pages* are also virtual addresses. Their actual value is still massaged by the MMU (the memory management unit, usually part of the CPU) before it is used to address physical memory.* *vmalloc* is not different in how it uses the hardware, but rather in how the kernel performs the allocation task.

The (virtual) address range used by *kmalloc* and *__get_free_pages* features a one-to-one mapping to physical memory, possibly shifted by a constant `PAGE_OFFSET` value; the functions don't need to modify the page tables for that address range. The address range used by *vmalloc* and *ioremap*, on the other hand, is completely synthetic, and each allocation builds the (virtual) memory area by suitably setting up the page tables.

This difference can be perceived by comparing the pointers returned by the allocation functions. On some platforms (for example, the x86), addresses returned by *vmalloc* are just beyond the addresses that *kmalloc* uses. On other platforms (for example, MIPS, IA-64, and x86_64), they belong to a completely different address range. Addresses available for *vmalloc* are in the range from `VMALLOC_START` to `VMALLOC_END`. Both symbols are defined in *<asm/pgtable.h>*.

Addresses allocated by *vmalloc* can't be used outside of the microprocessor, because they make sense only on top of the processor's MMU. When a driver needs a real physical address (such as a DMA address, used by peripheral hardware to drive the system's bus), you can't easily use *vmalloc*. The right time to call *vmalloc* is when

* Actually, some architectures define ranges of "virtual" addresses as reserved to address physical memory. When this happens, the Linux kernel takes advantage of the feature, and both the kernel and *__get_free_pages* addresses lie in one of those memory ranges. The difference is transparent to device drivers and other code that is not directly involved with the memory-management kernel subsystem.

you are allocating memory for a large sequential buffer that exists only in software. It's important to note that *vmalloc* has more overhead than *__get_free_pages*, because it must both retrieve the memory and build the page tables. Therefore, it doesn't make sense to call *vmalloc* to allocate just one page.

An example of a function in the kernel that uses *vmalloc* is the *create_module* system call, which uses *vmalloc* to get space for the module being created. Code and data of the module are later copied to the allocated space using *copy_from_user*. In this way, the module appears to be loaded into contiguous memory. You can verify, by looking in */proc/kallsyms*, that kernel symbols exported by modules lie in a different memory range from symbols exported by the kernel proper.

Memory allocated with *vmalloc* is released by *vfree*, in the same way that *kfree* releases memory allocated by *kmalloc*.

Like *vmalloc*, *ioremap* builds new page tables; unlike *vmalloc*, however, it doesn't actually allocate any memory. The return value of *ioremap* is a special virtual address that can be used to access the specified physical address range; the virtual address obtained is eventually released by calling *iounmap*.

ioremap is most useful for mapping the (physical) address of a PCI buffer to (virtual) kernel space. For example, it can be used to access the frame buffer of a PCI video device; such buffers are usually mapped at high physical addresses, outside of the address range for which the kernel builds page tables at boot time. PCI issues are explained in more detail in Chapter 12.

It's worth noting that for the sake of portability, you should not directly access addresses returned by *ioremap* as if they were pointers to memory. Rather, you should always use *readb* and the other I/O functions introduced in Chapter 9. This requirement applies because some platforms, such as the Alpha, are unable to directly map PCI memory regions to the processor address space because of differences between PCI specs and Alpha processors in how data is transferred.

Both *ioremap* and *vmalloc* are page oriented (they work by modifying the page tables); consequently, the relocated or allocated size is rounded up to the nearest page boundary. *ioremap* simulates an unaligned mapping by "rounding down" the address to be remapped and by returning an offset into the first remapped page.

One minor drawback of *vmalloc* is that it can't be used in atomic context because, internally, it uses *kmalloc(GFP_KERNEL)* to acquire storage for the page tables, and therefore could sleep. This shouldn't be a problem—if the use of *__get_free_page* isn't good enough for an interrupt handler, the software design needs some cleaning up.

A scull Using Virtual Addresses: scullv

Sample code using *vmalloc* is provided in the *scullv* module. Like *scullp*, this module is a stripped-down version of *scull* that uses a different allocation function to obtain space for the device to store data.

The module allocates memory 16 pages at a time. The allocation is done in large chunks to achieve better performance than *scullp* and to show something that takes too long with other allocation techniques to be feasible. Allocating more than one page with *__get_free_pages* is failure prone, and even when it succeeds, it can be slow. As we saw earlier, *vmalloc* is faster than other functions in allocating several pages, but somewhat slower when retrieving a single page, because of the overhead of page-table building. *scullv* is designed like *scullp*. `order` specifies the "order" of each allocation and defaults to 4. The only difference between *scullv* and *scullp* is in allocation management. These lines use *vmalloc* to obtain new memory:

```
/* Allocate a quantum using virtual addresses */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)vmalloc(PAGE_SIZE << dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

and these lines release memory:

```
/* Release the quantum-set */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        vfree(dptr->data[i]);
```

If you compile both modules with debugging enabled, you can look at their data allocation by reading the files they create in */proc*. This snapshot was taken on an x86_64 system:

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem
Device 0: qset 500, order 0, sz 1535135
  item at 000001001847da58, qset at 000001001db4c000
       0:1001db56000
       1:1003d1c7000

salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1535135
  item at 000001001847da58, qset at 0000010013dea000
       0:ffffff0001177000
       1:ffffff0001188000
```

The following output, instead, came from an x86 system:

```
rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem

Device 0: qset 500, order 0, sz 1535135
  item at ccf80e00, qset at cf7b9800
       0:ccc58000
       1:cccdd000

rudo%  cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1535135
  item at cfab4800, qset at cf8e4000
       0:d087a000
       1:d08d2000
```

The values show two different behaviors. On x86_64, physical addresses and virtual addresses are mapped to completely different address ranges (0x100 and 0xffffff00), while on x86 computers, *vmalloc* returns virtual addresses just above the mapping used for physical memory.

Per-CPU Variables

Per-CPU variables are an interesting 2.6 kernel feature. When you create a per-CPU variable, each processor on the system gets its own copy of that variable. This may seem like a strange thing to want to do, but it has its advantages. Access to per-CPU variables requires (almost) no locking, because each processor works with its own copy. Per-CPU variables can also remain in their respective processors' caches, which leads to significantly better performance for frequently updated quantities.

A good example of per-CPU variable use can be found in the networking subsystem. The kernel maintains no end of counters tracking how many of each type of packet was received; these counters can be updated thousands of times per second. Rather than deal with the caching and locking issues, the networking developers put the statistics counters into per-CPU variables. Updates are now lockless and fast. On the rare occasion that user space requests to see the values of the counters, it is a simple matter to add up each processor's version and return the total.

The declarations for per-CPU variables can be found in *<linux/percpu.h>*. To create a per-CPU variable at compile time, use this macro:

```
DEFINE_PER_CPU(type, name);
```

If the variable (to be called `name`) is an array, include the dimension information with the `type`. Thus, a per-CPU array of three integers would be created with:

```
DEFINE_PER_CPU(int[3], my_percpu_array);
```

Per-CPU variables can be manipulated without explicit locking—almost. Remember that the 2.6 kernel is preemptible; it would not do for a processor to be preempted in

the middle of a critical section that modifies a per-CPU variable. It also would not be good if your process were to be moved to another processor in the middle of a per-CPU variable access. For this reason, you must explicitly use the *get_cpu_var* macro to access the current processor's copy of a given variable, and call *put_cpu_var* when you are done. The call to *get_cpu_var* returns an lvalue for the current processor's version of the variable and disables preemption. Since an lvalue is returned, it can be assigned to or operated on directly. For example, one counter in the networking code is incremented with these two statements:

```
get_cpu_var(sockets_in_use)++;
put_cpu_var(sockets_in_use);
```

You can access another processor's copy of the variable with:

```
per_cpu(variable, int cpu_id);
```

If you write code that involves processors reaching into each other's per-CPU variables, you, of course, have to implement a locking scheme that makes that access safe.

Dynamically allocated per-CPU variables are also possible. These variables can be allocated with:

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
```

In most cases, *alloc_percpu* does the job; you can call *__alloc_percpu* in cases where a particular alignment is required. In either case, a per-CPU variable can be returned to the system with *free_percpu*. Access to a dynamically allocated per-CPU variable is done via *per_cpu_ptr*:

```
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

This macro returns a pointer to the version of `per_cpu_var` corresponding to the given `cpu_id`. If you are simply reading another CPU's version of the variable, you can dereference that pointer and be done with it. If, however, you are manipulating the current processor's version, you probably need to ensure that you cannot be moved out of that processor first. If the entirety of your access to the per-CPU variable happens with a spinlock held, all is well. Usually, however, you need to use *get_cpu* to block preemption while working with the variable. Thus, code using dynamic per-CPU variables tends to look like this:

```
int cpu;

cpu = get_cpu( )
ptr = per_cpu_ptr(per_cpu_var, cpu);
/* work with ptr */
put_cpu( );
```

When using compile-time per-CPU variables, the *get_cpu_var* and *put_cpu_var* macros take care of these details. Dynamic per-CPU variables require more explicit protection.

Per-CPU variables can be exported to modules, but you must use a special version of the macros:

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

To access such a variable within a module, declare it with:

```
DECLARE_PER_CPU(type, name);
```

The use of *DECLARE_PER_CPU* (instead of *DEFINE_PER_CPU*) tells the compiler that an external reference is being made.

If you want to use per-CPU variables to create a simple integer counter, take a look at the canned implementation in *<linux/percpu_counter.h>*. Finally, note that some architectures have a limited amount of address space available for per-CPU variables. If you create per-CPU variables in your code, you should try to keep them small.

Obtaining Large Buffers

As we have noted in previous sections, allocations of large, contiguous memory buffers are prone to failure. System memory fragments over time, and chances are that a truly large region of memory will simply not be available. Since there are usually ways of getting the job done without huge buffers, the kernel developers have not put a high priority on making large allocations work. Before you try to obtain a large memory area, you should really consider the alternatives. By far the best way of performing large I/O operations is through scatter/gather operations, which we discuss in the section "Scatter-gather mappings" in Chapter 1.

Acquiring a Dedicated Buffer at Boot Time

If you really need a huge buffer of physically contiguous memory, the best approach is often to allocate it by requesting memory at boot time. Allocation at boot time is the only way to retrieve consecutive memory pages while bypassing the limits imposed by *__get_free_pages* on the buffer size, both in terms of maximum allowed size and limited choice of sizes. Allocating memory at boot time is a "dirty" technique, because it bypasses all memory management policies by reserving a private memory pool. This technique is inelegant and inflexible, but it is also the least prone to failure. Needless to say, a module can't allocate memory at boot time; only drivers directly linked to the kernel can do that.

One noticeable problem with boot-time allocation is that it is not a feasible option for the average user, since this mechanism is available only for code linked in the kernel image. A device driver using this kind of allocation can be installed or replaced only by rebuilding the kernel and rebooting the computer.

When the kernel is booted, it gains access to all the physical memory available in the system. It then initializes each of its subsystems by calling that subsystem's initialization function, allowing initialization code to allocate a memory buffer for private use by reducing the amount of RAM left for normal system operation.

Boot-time memory allocation is performed by calling one of these functions:

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

The functions allocate either whole pages (if they end with _pages) or non-page-aligned memory areas. The allocated memory may be high memory unless one of the _low versions is used. If you are allocating this buffer for a device driver, you probably want to use it for DMA operations, and that is not always possible with high memory; thus, you probably want to use one of the _low variants.

It is rare to free memory allocated at boot time; you will almost certainly be unable to get it back later if you want it. There is an interface to free this memory, however:

```
void free_bootmem(unsigned long addr, unsigned long size);
```

Note that partial pages freed in this manner are not returned to the system—but, if you are using this technique, you have probably allocated a fair number of whole pages to begin with.

If you must use boot-time allocation, you need to link your driver directly into the kernel. See the files in the kernel source under *Documentation/kbuild* for more information on how this should be done.

Quick Reference

The functions and symbols related to memory allocation are:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

The most frequently used interface to memory allocation.

```
#include <linux/mm.h>
GFP_USER
GFP_KERNEL
GFP_NOFS
GFP_NOIO
GFP_ATOMIC
```

Flags that control how memory allocations are performed, from the least restrictive to the most. The GFP_USER and GFP_KERNEL priorities allow the current process

to be put to sleep to satisfy the request. GFP_NOFS and GFP_NOIO disable filesystem operations and all I/O operations, respectively, while GFP_ATOMIC allocations cannot sleep at all.

__GFP_DMA
__GFP_HIGHMEM
__GFP_COLD
__GFP_NOWARN
__GFP_HIGH
__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY

These flags modify the kernel's behavior when allocating memory.

```
#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset,
  unsigned long flags, constructor( ), destructor( ));
int kmem_cache_destroy(kmem_cache_t *cache);
```

Create and destroy a slab cache. The cache can be used to allocate several objects of the same size.

SLAB_NO_REAP
SLAB_HWCACHE_ALIGN
SLAB_CACHE_DMA

Flags that can be specified while creating a cache.

SLAB_CTOR_ATOMIC
SLAB_CTOR_CONSTRUCTOR

Flags that the allocator can pass to the constructor and the destructor functions.

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

Allocate and release a single object from the cache.

/proc/slabinfo

A virtual file containing statistics on slab cache usage.

```
#include <linux/mempool.h>
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t
  *free_fn, void *data);
void mempool_destroy(mempool_t *pool);
```

Functions for the creation of memory pools, which try to avoid memory allocation failures by keeping an "emergency list" of allocated items.

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

Functions for allocating items from (and returning them to) memory pools.

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long order);
```

The page-oriented allocation functions. *get_zeroed_page* returns a single, zero-filled page. All the other versions of the call do not initialize the contents of the returned page(s).

```
int get_order(unsigned long size);
```

Returns the allocation order associated to *size* in the current platform, according to `PAGE_SIZE`. The argument must be a power of two, and the return value is at least `0`.

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

Functions that release page-oriented allocations.

```
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

All variants of the lowest-level page allocator in the Linux kernel.

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

Various ways of freeing pages allocated with one of the forms of *alloc_page*.

```
#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
```

Functions that allocate or free a contiguous *virtual* address space. *ioremap* accesses physical memory through virtual addresses, while *vmalloc* allocates free pages. Regions mapped with *ioremap* are freed with *iounmap*, while pages obtained from *vmalloc* are released with *vfree*.

```
#include <linux/percpu.h>
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(type, name);
```

Macros that define and declare per-CPU variables.

```
per_cpu(variable, int cpu_id)
get_cpu_var(variable)
put_cpu_var(variable)
```

Macros that provide access to statically declared per-CPU variables.

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(void *variable);
```

Functions that perform runtime allocation and freeing of per-CPU variables.

```
int get_cpu( );
void put_cpu( );
per_cpu_ptr(void *variable, int cpu_id)
```

get_cpu obtains a reference to the current processor (therefore, preventing preemption and movement to another processor) and returns the ID number of the processor; *put_cpu* returns that reference. To access a dynamically allocated per-CPU variable, use *per_cpu_ptr* with the ID of the CPU whose version should be accessed. Manipulations of the current CPU's version of a dynamic, per-CPU variable should probably be surrounded by calls to *get_cpu* and *put_cpu*.

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
void free_bootmem(unsigned long addr, unsigned long size);
```

Functions (which can be used only by drivers directly linked into the kernel) that perform allocation and freeing of memory at system bootstrap time.