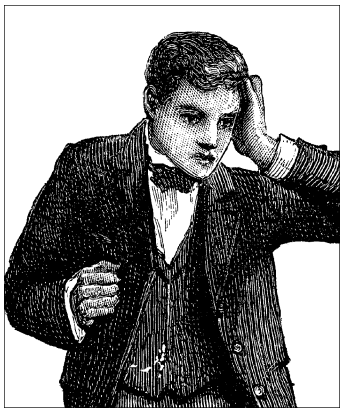

CHAPTER SEVEN

GETTING HOLD OF 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 making use of memory in device drivers and at how to make the best use of your system's memory resources. We will 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 will defer it to "Memory Management in Linux" in Chapter 13.

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 it 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

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-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. While the current process sleeps, the kernel takes proper action to retrieve a memory page, 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, task queues, 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 will fail.

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/mm.h>`: individual flags are prefixed with a double underscore, like `__GFP_DMA`; collections of flags lack the prefix and are sometimes called *allocation priorities*.

`GFP_KERNEL`
: Normal allocation of kernel memory. May sleep.

`GFP_BUFFER`
: Used in managing the buffer cache, this priority allows the allocator to sleep. It differs from `GFP_KERNEL` in that fewer attempts will be made to free memory by flushing dirty pages to disk; the purpose here is to avoid deadlocks when the I/O subsystems themselves need memory.

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

`GFP_USER`
: Used to allocate memory on behalf of the user. It may sleep, and is a low-priority request.

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

`__GFP_DMA`
: This flag requests memory usable in DMA data transfers to/from devices. Its exact meaning is platform dependent, and the flag can be OR'd to either `GFP_KERNEL` or `GFP_ATOMIC`.

`__GFP_HIGHMEM`
: The flag requests high memory, a platform-dependent feature that has no effect on platforms that don't support it. It is part of the `GFP_HIGHUSER` mask and has little use elsewhere.

Memory zones

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

Version 2.4 of the kernel knows about 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 equivalent) will fall into this abstraction.

DMA-capable memory is the only memory that can be involved in DMA data transfers with peripheral devices. This restriction arises when the address bus used to connect peripheral devices to the processor is limited with respect to the address bus used to access RAM. For example, on the x86, devices that plug into the ISA bus can only address memory from 0 to 16 MB. Other platforms have similar needs, although usually less stringent than the ISA one.*

High memory is memory that requires special handling to be accessed. It made its appearance in kernel memory management when support for the Pentium II Virtual Memory Extension was implemented during 2.3 development to access up to 64 GB of physical memory. High memory is a concept that only applies to the x86 and SPARC platforms, and the two implementations are different.

Whenever a new page is allocated to fulfill the *kmalloc* 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 is searched; if `__GFP_HIGHMEM` is set, then all three zones are used to search a free page.

If the platform has no concept of high memory or it has been disabled in the kernel configuration, `__GFP_HIGHMEM` is defined as `0` and has no effect.

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 13.

---

* It's interesting to note that the limit is only in force for the ISA bus; an x86 device that plugs into the PCI bus can perform DMA with all *normal* memory.

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 than 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. After all, the implementation can change—as it did in the 2.1.38 kernel—without affecting the interface seen by the rest of the kernel.

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 minimum memory that *kmalloc* handles is as big as 32 or 64, depending on the page size used by the current architecture.

The data sizes available are generally powers of two. In the 2.0 kernel, the available sizes were actually slightly less than a power of two, due to control flags added by the management system. If you keep this fact in mind, you'll use memory more efficiently. For example, if you need a buffer of about 2000 bytes and run Linux 2.0, you're better off asking for 2000 bytes, rather than 2048. Requesting exactly a power of two is the worst possible case with any kernel older than 2.1.38—the kernel will allocate twice as much as you requested. This is why *scull* used 4000 bytes per quantum instead of 4096.

You can find the exact values used for the allocation blocks in *mm/kmalloc.c* (with the 2.0 kernel) or `mm/slab.c` (in current kernels), but remember that they can change again without notice. The trick of allocating less than 4 KB works well for *scull* with all 2.*x* kernels, but it's not guaranteed to be optimal in the future.

In any case, the maximum size that can be allocated by *kmalloc* is 128 KB—slightly less with 2.0 kernels. If you need more than a few kilobytes, however, there are better ways than *kmalloc* to obtain memory, as outlined next.

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 this sort of *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 ISDN drivers in Linux 2.4 use caches.

Linux memory caches have a type of `kmem_cache_t` and 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 will be cached. The maximum length for the name is 20 characters, including the trailing terminator.

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. You would not usually need to set this flag.

`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 is usually a good choice.

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

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 option will succeed only if all objects allocated from the cache have been returned to it. A module should thus 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. There is even a kernel configuration option that enables the collection of extra statistical information, but at a noticeable runtime cost. Cache 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. The sample module refuses to compile with version 2.0 of the kernel because memory caches were not there, as explained in "Backward Compatibility" later in the chapter.

scullc is a complete example that can be used to make tests. It differs from *scull* only in a few lines of code. 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]);
kfree(dptr->data);
```

To support use of `scullc_cache`, these few lines are included in the file at proper places:

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

    /* init_module: 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) {
        result = -ENOMEM;
        goto fail_malloc2;
    }

    /* cleanup_module: release the cache of our quanta */
    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.

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 will be introduced later, in "The mmap Device Operation" in Chapter 13.

To allocate pages, the following functions are available:

get_zeroed_page
: Returns a pointer to a new page and fills the page with zeros.

__get_free_page
: Similar to *get_zeroed_page*, but doesn't clear the page.

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

__get_dma_pages
: Similar to *get_free_pages*, but guarantees that the allocated memory is DMA capable. If you use version 2.2 or later of the kernel, you can simply use *__get_free_pages* and pass the `__GFP_DMA` flag; if you want backward compatibility with 2.0, you need to call this function instead.

The prototypes for the functions follow:

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

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 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., $\log_2 N$). 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 will fail. The maximum value of `order` was 5 in Linux 2.0 (corresponding to 32 pages) and 9 with later versions (corresponding to 512 pages: 2 MB on most platforms). Anyway, the bigger `order` is, the more likely it is that the allocation will fail.

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 than you allocated, the memory map will become corrupted and the system will get 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.

It has been said that if you want to live dangerously, you can assume that neither *kmalloc* nor the underlying *get_free_pages* will ever fail when called with a priority of `GFP_KERNEL`. This is *almost* true, but not completely: small, memory-limited systems can still run into trouble. A driver writer ignores the possibility of allocation failures at his or her peril (or that of his or her users).

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 will start 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 nonetheless be careful, 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).

A scull Using Whole Pages: scullp

In order to test page allocation for real, the *scullp* module is released 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 and can be specified at either compile time 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*, instead, 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 four megabytes 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 *__get_free_page* is that the page is 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'll discuss this kind of operation in "The mmap Device Operation" in Chapter 13, where we show how *scullp* offers memory mapping, something that *scull* cannot offer.

vmalloc and Friends

The next memory allocation function that we'll show you is *vmalloc*, which allocates a contiguous memory region in the *virtual* address space. Although the pages are not necessarily consecutive in physical memory (each page is retrieved with a separate call to *__get_free_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`.

The prototypes of the function and its relatives (*ioremap*, which is not strictly an allocation function, will be discussed shortly) 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 (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.

* 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.

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 greater than addresses that *kmalloc* addresses. On other platforms (for example, MIPS and IA-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 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 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*, after *insmod* has relocated the code. In this way, the module appears to be loaded into contiguous memory. You can verify, by looking in */proc/ksyms*, that kernel symbols exported by modules lie in a different memory range than 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*. Note that the return value from *ioremap* cannot be safely dereferenced on all platforms; instead, functions like *readb* should be used. See "Directly Mapped Memory" in Chapter 8for the details.

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 "The PCI Interface" in Chapter 15.

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 Using I/O Memory, in Chapter 8. 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.

There is almost no limit to how much memory *vmalloc* can allocate and *ioremap* can make accessible, although *vmalloc* refuses to allocate more memory than the amount of physical RAM, in order to detect common errors or typos made by programmers. You should remember, however, that requesting too much memory with *vmalloc* leads to the same problems as it does with *kmalloc*.

Both *ioremap* and *vmalloc* are page oriented (they work by modifying the page tables); thus the relocated or allocated size is rounded up to the nearest page boundary. In addition, the implementation of *ioremap* found in Linux 2.0 won't even consider remapping a physical address that doesn't start at a page boundary. Newer kernels allow that 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 at interrupt time because internally it uses `kmalloc(GFP_KERNEL)` to acquire storage for the page tables, and thus could sleep. This shouldn't be a problem—if the use of *__get_free_page* isn't good enough for an interrupt handler, then 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*. The following snapshots were taken on two different systems:

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

Device 0: qset 500, order 0, sz 1048576
  item at e00000003e641b40, qset at e000000025c60000
       0:e00000003007c000
       1:e000000024778000
salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1048576
  item at e0000000303699c0, qset at e000000025c87000
       0:a000000000034000
       1:a000000000078000
salma% uname -m
ia64

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

Device 0: qset 500, order 0, sz 1048576
  item at c4184780, qset at c71c4800
       0:c262b000
       1:c2193000
rudo%  cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1048576
  item at c4184b80, qset at c71c4000
       0:c881a000
       1:c882b000
rudo% uname -m
i686
```

The values show two different behaviors. On IA-64, physical addresses and virtual addresses are mapped to completely different address ranges (0xE and 0xA), whereas on x86 computers *vmalloc* returns virtual addresses just above the mapping used for physical memory.

Boot-Time Allocation

If you really need a huge buffer of physically contiguous memory, you need to allocate it by requesting memory at boot time. 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.

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.

One noticeable problem with boot-time allocation is that it is not a feasible option for the average user: being only available for code linked in the kernel image, a device driver using this kind of allocation can only be installed or replaced by rebuilding the kernel and rebooting the computer. Fortunately, there are a pair of workarounds to this problem, which we introduce soon.

Even though we won't suggest allocating memory at boot time, it's something worth mentioning because it used to be the only way to allocate a DMA-capable buffer in the first Linux versions, before `__GFP_DMA` was introduced.

Acquiring a Dedicated Buffer at Boot Time

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.

With version 2.4 of the kernel, this kind of 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. They allocate either low or normal memory (see the discussion of memory zones earlier in this chapter). Normal allocation returns memory addresses that are above `MAX_DMA_ADDRESS`; low memory is at addresses lower than that value.

This interface was introduced in version 2.3.23 of the kernel. Earlier versions used a less refined interface, similar to the one described in Unix books. Basically, the initialization functions of several kernel subsystems received two `unsigned long` arguments, which represented the current bounds of the free memory area. Each such function could steal part of this area, returning the new lower bound. A driver allocating memory at boot time, therefore, was able to steal consecutive memory from the linear array of available RAM.

The main problem with this older mechanism of managing boot-time allocation requests was that not all initialization functions could modify the lower memory bound, so writing a driver needing such allocation usually implied providing users with a kernel patch. On the other hand, *alloc_bootmem* can be called by the initialization function of any kernel subsystem, provided it is performed at boot time.

This way of allocating memory has several disadvantages, not the least being the inability to ever free the buffer. After a driver has taken some memory, it has no way of returning it to the pool of free pages; the pool is created after all the physical allocation has taken place, and we don't recommend hacking the data structures internal to memory management. On the other hand, the advantage of this technique is that it makes available an area of consecutive physical memory that is suitable for DMA. This is currently the only safe way in the standard kernel to allocate a buffer of more than 32 consecutive pages, because the maximum value of `order` that is accepted by *get_free_pages* is 5. If, however, you need many pages and they don't have to be physically contiguous, *vmalloc* is by far the best function to use.

If you are going to resort to grabbing memory at boot time, you must modify *init/main.c* in the kernel sources. You'll find more about *main.c* in Chapter 16.

Note that this "allocation" can be performed only in multiples of the page size, though the number of pages doesn't have to be a power of two.

The bigphysarea Patch

Another approach that can be used to make large, contiguous memory regions available to drivers is to apply the *bigphysarea* patch. This unofficial patch has been floating around the Net for years; it is so renowned and useful that some distributions apply it to the kernel images they install by default. The patch basically allocates memory at boot time and makes it available to device drivers at runtime. You'll need to pass a command-line option to the kernel to specify the amount of memory that must be reserved at boot time.

The patch is currently maintained at *http://www.polyware.nl/~middelink/En/hob-v4l.html*. It includes its own documentation that describes the allocation interface it offers to device drivers. The Zoran 36120 frame grabber driver, part of the 2.4 kernel (in *drivers/char/zr36120.c*) uses the *bigphysarea* extension if it is available, and is thus a good example of how the interface is used.

Reserving High RAM Addresses

The last option for allocating contiguous memory areas, and possibly the easiest, is reserving a memory area at the *end* of physical memory (whereas *bigphysarea* reserves it at the beginning of physical memory). To this aim, you need to pass a command-line option to the kernel to limit the amount of memory being managed. For example, one of your authors uses `mem=126M` to reserve 2 megabytes in a system that actually has 128 megabytes of RAM. Later, at runtime, this memory can be allocated and used by device drivers.

The *allocator* module, part of the sample code released on the O'Reilly FTP site, offers an allocation interface to manage any high memory not used by the Linux kernel. The module is described in more detail in "Do-it-yourself allocation" in Chapter 13.

The advantage of *allocator* over the *bigphysarea* patch is that there's no need to modify official kernel sources. The disadvantage is that you must change the command-line option to the kernel whenever you change the amount of RAM in the system. Another disadvantage, which makes *allocator* unsuitable in some situations is that high memory cannot be used for some tasks, such as DMA buffers for ISA devices.

Backward Compatibility

The Linux memory management subsystem has changed dramatically since the 2.0 kernel came out. Happily, however, the changes to its programming interface have been much smaller and easier to deal with.

kmalloc and *kfree* have remained essentially constant between Linux 2.0 and 2.4. Access to high memory, and thus the `__GFP_HIGHMEM` flag, was added starting with kernel 2.3.23; *sysdep.h* fills the gaps and allows for 2.4 semantics to be used in 2.2 and 2.0.

The lookaside cache functions were introduced in Linux 2.1.23, and were simply not available in the 2.0 kernel. Code that must be portable back to Linux 2.0 should stick with *kmalloc* and *kfree*. Moreover, *kmem_destroy_cache* was introduced during 2.3 development and has only been backported to 2.2 as of 2.2.18. For this reason *scullc* refuses to compile with a 2.2 kernel older than that.

__get_free_pages in Linux 2.0 had a third, integer argument called `dma`; it served the same function that the `__GFP_DMA` flag serves in modern kernels but it was not merged in the `flags` argument. To address the problem, *sysdep.h* passes 0 as the third argument to the 2.0 function. If you want to request DMA pages and be backward compatible with 2.0, you need to call *get_dma_pages* instead of using `__GFP_DMA`.

vmalloc and *vfree* are unchanged across all 2.*x* kernels. However, the *ioremap* function was called *vremap* in the 2.0 days, and there was no *iounmap*. Instead, an I/O mapping obtained with *vremap* would be freed with *vfree*. Also, the header `<linux/vmalloc.h>` didn't exist in 2.0; the functions were declared by `<linux/mm.h>` instead. As usual, *sysdep.h* makes 2.4 code work with earlier kernels; it also includes `<linux/vmalloc.h>` if `<linux/mm.h>` is included, thus hiding this difference as well.

Quick Reference

The functions and symbols related to memory allocation follow.

```
#include <linux/malloc.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_KERNEL
GFP_ATOMIC
__GFP_DMA
__GFP_HIGHMEM
```
kmalloc flags. `__GFP_DMA` and `__GFP_HIGHMEM` are flags that can be OR'd to either `GFP_KERNEL` or `GFP_ATOMIC`.

```
#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.

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long
      order);
unsigned long __get_dma_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). *__get_dma_pages* is only a compatibility macro in Linux 2.2 and later (you can use `__GFP_DMA` instead).

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```
These functions release page-oriented allocations.

```
#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);
```
These functions 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/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);
```
Only with version 2.4 of the kernel, memory can be allocated at boot time using these functions. The facility can only be used by drivers directly linked in the kernel image.