Toolbox

Kernel Korner

In this article, Robert offers a refresher on kernel memory allocation and how it has changed for the 2.

by Robert Love

Unfortunately for kernel developers, allocating memory in the kernel is not as simple as allocating memory in user space. A number of factors contribute to the complication, among them:

- The kernel is limited to about 1GB of virtual and physical memory.
- The kernel's memory is not pageable.
- The kernel usually wants physically contiguous memory.
- Often, the kernel must allocate the memory without sleeping.
- Mistakes in the kernel have a much higher price than they do elsewhere.

Although easy access to an abundance of memory certainly is not a luxury to the kernel, a little understanding of the issues can go a long way toward making the process relatively painless.

A General-Purpose Allocator

The general interface for allocating memory inside of the kernel is kmalloc():

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

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

It should look familiar—it is pretty much the same as user space's malloc(), after all—except that it takes a second argument, flags. Let's ignore flags for a second and see what we recognize. First off, size is the same here as in malloc()'s—it specifies the size in bytes of the allocation. Upon successful return, kmalloc() returns a pointer to size bytes of memory. The alignment of the allocated memory is suitable for storage of and access to any type of object. As with malloc(), kmalloc() can fail, and you must check its return value against NULL. Let's look at an example:

```
struct falcon *p;

p = kmalloc(sizeof (struct falcon), GFP_KERNEL);
if (!p)
  /* the allocation failed - handle appropriately */
```

Flags
The flags field controls the behavior of memory allocation. We can divide flags into three groups: action modifiers, zone modifiers and types. Action modifiers tell the kernel how to allocate memory. They specify, for example, whether the kernel can sleep (that is, whether the call to kmalloc() can block) in order to satisfy the allocation. Zone modifiers, on the other hand, tell the kernel from where the request should be satisfied. For example, some requests may need to be satisfied from memory that hardware can access through direct memory access (DMA). Finally, type flags specify a type of allocation. They group together relevant action and zone modifiers into a single mnemonic. In general, instead of specifying multiple action and zone modifiers, you specify a single type flag.

Table 1 is a listing of the action modifiers, and Table 2 is a listing of the zone modifiers. Many different flags can be used; allocating memory in the kernel is nontrivial. It is possible to control many aspects of memory allocation in the kernel. Your code should use the type flags and not the individual action and zone modifiers. The two most common flags are GFP_ATOMIC and GFP_KERNEL. Nearly all of your kernel memory allocations should specify one of these two flags.

**Table 1. Action Modifiers**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__GFP_COLD</td>
<td>The kernel should use cache cold pages.</td>
</tr>
<tr>
<td>__GFP_FS</td>
<td>The kernel can start filesystem I/O.</td>
</tr>
<tr>
<td>__GFP_HIGH</td>
<td>The kernel can access emergency pools.</td>
</tr>
<tr>
<td>__GFP_IO</td>
<td>The kernel can start disk I/O.</td>
</tr>
<tr>
<td>__GFP_NOFAIL</td>
<td>The kernel can repeat the allocation.</td>
</tr>
<tr>
<td>__GFP_NORETRY</td>
<td>The kernel does not retry if the allocation fails.</td>
</tr>
<tr>
<td>__GFP_NOWARN</td>
<td>The kernel does not print failure warnings.</td>
</tr>
<tr>
<td>__GFP_REPEAT</td>
<td>The kernel repeats the allocation if it fails.</td>
</tr>
<tr>
<td>__GFP_WAIT</td>
<td>The kernel can sleep.</td>
</tr>
</tbody>
</table>

**Table 2. Zone Modifiers**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__GFP_DMA</td>
<td>Allocate only DMA-capable memory.</td>
</tr>
<tr>
<td>No flag</td>
<td>Allocate from wherever available.</td>
</tr>
</tbody>
</table>

The GFP_ATOMIC flag instructs the memory allocator never to block. Use this flag in situations where it cannot sleep—where it must remain atomic—such as interrupt handlers, bottom halves and process context code that is holding a lock. Because the kernel cannot block the allocation and try to free up sufficient memory to satisfy the request, an allocation specifying GFP_ATOMIC has a lesser chance of succeeding than one that does not. Nonetheless, if your current context is incapable of sleeping, it is your only choice. Using GFP_ATOMIC is simple:

```c
struct wolf *p;
p = kmalloc(sizeof (struct wolf), GFP_ATOMIC);
if (!p)
    /* error */
```

Conversely, the GFP_KERNEL flag specifies a normal kernel allocation. Use this flag in code executing in
process context without any locks. A call to kmalloc() with this flag can sleep; thus, you must use this flag only when it is safe to do so. The kernel utilizes the ability to sleep in order to free memory, if needed. Therefore, allocations that specify this flag have a greater chance of succeeding. If insufficient memory is available, for example, the kernel can block the requesting code and swap some inactive pages to disk, shrink the in-memory caches, write out buffers and so on.

Sometimes, as when writing an ISA device driver, you need to ensure that the memory allocated is capable of undergoing DMA. For ISA devices, this is memory in the first 16MB of physical memory. To ensure that the kernel allocates from this specific memory, use the GFP_DMA flag. Generally, you would use this flag in conjunction with either GFP_ATOMIC or GFP_KERNEL; you can combine flags with a binary OR operation. For example, to instruct the kernel to allocate DMA-capable memory and to sleep if needed, do:

```c
char *buf;
/* we want DMA-capable memory, * and we can sleep if needed */
buf = kmalloc(BUF_LEN, GFP_DMA | GFP_KERNEL);
if (!buf)
    /* error */
```

Table 3 is a listing of the type flags, and Table 4 shows to which type flag each action and zone modifier equates. The header `<linux/gfp.h>` defines all of the flags.

### Table 3. Types

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>The allocation is high-priority and does not sleep. This is the flag to use in interrupt handlers, bottom halves and other situations where you cannot sleep.</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>This is an allocation of DMA-capable memory. Device drivers that need DMA-capable memory use this flag.</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep.</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>This allocation might block and might initiate disk I/O, but it does not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>This allocation might block, but it does not initiate block I/O. This is the flag to use in block layer code when you cannot start more block I/O.</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.</td>
</tr>
</tbody>
</table>

### Table 4. Composition of the Type Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>__GFP_HIGH</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>Flag</td>
<td>Value</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>GFP_DMA</td>
</tr>
</tbody>
</table>

**Returning Memory**

When you are finished accessing the memory allocated via kmalloc(), you must return it to the kernel. This job is done using kfree(), which is the counterpart to user space's free() library call. The prototype for kfree() is:

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

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

kfree()'s usage is identical to the user-space variant. Assume `p` is a pointer to a block of memory obtained via kmalloc(). The following command, then, would free that block and return the memory to the kernel:

```c
kfree(p);
```

As with free() in user space, calling kfree() on a block of memory that already has been freed or on a pointer that is not an address returned from kmalloc() is a bug, and it can result in memory corruption. Always balance allocations and frees to ensure that kfree() is called exactly once on the correct pointer. Calling kfree() on NULL is checked for explicitly and is safe, although it is not necessarily a sensible idea.

Let's look at the full allocation and freeing cycle:

```c
struct sausage *s;

s = kmalloc(sizeof (struct sausage), GFP_KERNEL);
if (!s)
    return -ENOMEM;
/* ... */

kfree(s);
```

**Allocating from Virtual Memory**

The kmalloc() function returns physically and therefore virtually contiguous memory. This is a contrast to user space's malloc() function, which returns virtually but not necessarily physically contiguous memory. Physically contiguous memory has two primary benefits. First, many hardware devices cannot address virtual memory. Therefore, in order for them to be able to access a block of memory, the block must exist as a physically contiguous chunk of memory. Second, a physically contiguous block of memory can use a single large page mapping. This minimizes the translation lookaside buffer (TLB) overhead of addressing the memory, as only a single TLB entry is required.

Allocating physically contiguous memory has one downside: it is often hard to find physically contiguous blocks of memory, especially for large allocations. Allocating memory that is only virtually contiguous has a much larger chance of success. If you do not need physically contiguous memory, use vmalloc():

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

void * vmalloc(unsigned long size);
```
You then return memory obtained with vmalloc() to the system by using vfree():

```c
#include <linux/vmalloc.h>
void vfree(void *addr);
```

Here again, vfree()'s usage is identical to user space's malloc() and free() functions:

```c
struct black_bear *p;
p = vmalloc(sizeof (struct black_bear));
if (!p)
  /* error */
/* ... */
vfree(p);
```

In this particular case, vmalloc() might sleep.

Many allocations in the kernel can use vmalloc(), because few allocations need to appear contiguous to hardware devices. If you are allocating memory that only software accesses, such as data associated with a user process, there is no need for the memory to be physically contiguous. Nonetheless, few allocations in the kernel use vmalloc(). Most choose to use kmalloc(), even if it's not needed, partly for historical and partly for performance reasons. Because the TLB overhead for physically contiguous pages is reduced greatly, the performance gains often are well appreciated. Despite this, if you need to allocate tens of megabytes of memory in the kernel, vmalloc() is your best option.

**A Small Fixed-Size Stack**

Unlike user-space processes, code executing in the kernel has neither a large nor a dynamically growing stack. Instead, each process in the kernel has a small fixed-size stack. The exact size of the stack is architecture-dependent. Most architectures allocate two pages for the stack, so the stack is 8KB on 32-bit machines.

Because of the small stack, allocations that are large, automatic and on-the-stack are discouraged. Indeed, you never should see anything such as this in kernel code:

```c
#define BUF_LEN 2048
void rabbit_function(void)
{
  char buf[BUF_LEN];
  /* ... */
}
```

Instead, the following is preferred:

```c
#define BUF_LEN 2048
void rabbit_function(void)
{
  char *buf;
  buf = kmalloc(BUF_LEN, GFP_KERNEL);
  if (!buf)
    /* error! */
```
You also seldom see the equivalent of this stack in user space, because there is rarely a reason to perform a
dynamic memory allocation when you know the allocation size at the time you write the code. In the kernel,
however, you should use dynamic memory any time the allocation size is larger than a handful of bytes or so.
This helps prevent stack overflow, which ruins everyone's day.

**Conclusion**

With a little understanding, getting a hold of memory in the kernel is demystified and not too much more
difficult to do than it is in user space. A few simple rules of thumb can go a long way:

- Decide whether you can sleep (that is, whether the call to kmalloc() can block). If you are in an
  interrupt handler, in a bottom half, or if you hold a lock, you cannot. If you are in process context and
do not hold a lock, you probably can.

- If you can sleep, specify GFP_KERNEL.

- If you cannot sleep, specify GFP_ATOMIC.

- If you need DMA-capable memory (for example, for an ISA or broken PCI device), specify
  GFP_DMA.

- Always check for and handle a NULL return value from kmalloc().

- Do not leak memory; make sure you call kfree() somewhere.

- Ensure that you do not race and call kfree() multiple times and that you never access a block of
  memory after you free it.

**Resources**

For more information, check out these files in your kernel source tree.

- include/linux/gfp.h: home of the allocation flags.
- include/linux/slab.h: definitions of kmalloc(), et al.
- mm/page_alloc.c: page allocation functions.
- mm/slab.c: implementation of kmalloc(), et al.

Robert Love ([rml@tech9.net](mailto:rml@tech9.net)) is a kernel hacker at MontaVista Software and a student at the University of
Florida. He is the author of *Linux Kernel Development*. Robert enjoys fine wine and lives in Gainesville,
Florida.