

Memory management :: inside the kernel

Contributed by Cereal
Monday, 05 September 2005

This is the second article about memory management by the Linux kernel. The first article took you through the memory in the user space. This second article will explain how memory management is done inside the kernel itself. I recommend reading the first article before starting on this one.

Now we know how the kernel manages memory from the first article, let's have a look how the kernel allows you to allocate and free memory inside the kernel.

{mosgoogle_right}

=> Allocating and Freeing pages

Getting free pages

The kernel provides one low-level mechanism for requesting memory, and several methods to access the allocated memory. All these interfaces are defined inside `< linux/gfp.h >` and work with a page-sized granularity. The master or core function is:

```
struct page * alloc_pages( unsigned int gfp_mask, unsigned int order )
```

This will allocate physical pages and returns a pointer to the first pages page structure. When an error occurs NULL is returned. The next function we will need is a function to convert a given page to its logical address:

```
void * page_address( struct page *page )
```

This returns a pointer to the logical address where the physical page resides. There is a handy function in the kernel, if you not really need the struct page, you can use this function:

```
unsigned long __get_free_pages( unsigned int gfp_mask, unsigned int order )
```

The above function works the same as `alloc_page()`, except this function directly returns the logical address of the first requested page. Because the pages are contiguous, the next pages will simply follow from the first.

For getting only 1 page, there are two functions implemented as wrappers to save you some typing:

```
struct page * alloc_page( unsigned int gfp_mask )
```

```
unsigned long __gfp_free_page( unsigned int gfp_mask )
```

These functions work the same as the others but pass zero for the order, and $2^0=1$, so 1 page is returned.

There is one special function declared, this function is used to get a free page filled with zeros.

```
unsigned long get_zeroed_page( unsigned int gfp_mask )
```

A lot of people will think "what can be the usage of this function?"; Well it's pretty easy, when you allocate a page that is given to the user-space it might "randomly" contain sensitive data. All data will be zeroed with this function, so it's safe to give this page to the user space, this way the system security isn't compromised.

Freeing Allocated Pages

When you no longer need the pages you have allocated, you need to free them, so the kernel knows when it can use these pages for another process.

The kernel defines 3 functions for this:

```
void __free_pages( struct page *pages, unsigned int order )
```

```
void free_pages( unsigned long addr, unsigned int order )
```

```
void free_page( unsigned long addr )
```

The first function is used when you actually have the struct page, the last 2 functions are used to free a page based on its address.

When freeing pages, you must be careful to only free the pages you allocated. Passing a wrong struct page or a wrong address or even an incorrect order could result in a memory corruption. Remember the kernel trusts itself. Unlike user-space, the kernel happily hangs itself if you ask it.

Example code

```
unsigned long allocated_pages;
```

```
allocated_pages = __get_free_pages( GFP_KERNEL, 2 );
```

```
if ( !allocated_pages )
```

```
/* An error ocured, handle it */
```

```
/* allocated_pages is now the address to the first page off the 4 allocated pages */
```

```
/* Use the pages for what you need it */
```

```
free_pages( allocated_pages, 3 );
```

```
/* Or pages are now given free, so we no longer should use the allocated_pages long */
```

=> Allocating and freeing bytes

kmalloc() and kfree()

This function works very similar to the user-space malloc() function; with the exception that kmalloc takes a second argument, the flag argument. The kmalloc() function is a simple interface for obtaining kernel memory in byte-sized chunks. If you need a whole page the previously discussed interface might be a better choice. For most kernel allocations kmalloc() is the preferred interface. The kmalloc() function is declared in < linux/slab.h >:

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

This function will return a pointer to a region of memory that is at least size bytes in length. The region of memory allocated is physically contiguous. On error the function will return NULL. Kernel memory allocation will always succeed, unless there is an insufficient amount off memory available.

The opposite of `kmalloc()` is `kfree()`, which is declared in the same file.

```
void kfree( const void *ptr )
```

Calling this function to free memory that is not previously allocated with `kmalloc()` or calling it on memory that is already freed will result in bad things. Like in the user space, be careful to balance your allocations with your de-allocations to prevent memory leaks and other bugs. Calling `kfree(NULL)` is explicitly checked for and safe.

```
vmalloc() and vfree()
```

The `vmalloc()` function is almost the same as the `kmalloc()` function, though there is only 1 big difference, as you know `kmalloc()` allocates memory that is physically contiguous, `vmalloc()` allocates memory that is virtual contiguous and not necessarily physically contiguous. This is the same as `malloc()` does inside the user-space, `malloc()` allocates pages that are contiguous within the virtual address space of the processor, but it is not guaranteed that they are actually contiguous in physical RAM. The `vmalloc()` function allocates pages that are virtual contiguous, it does this by allocating potentially noncontiguous chunks of physical memory and "fixing up" the page tables to map the memory into a contiguous chunk of the logical address space.

Mostly it is only a hardware device that requires physically contiguous memory allocations. Hardware devices live on the other side of the memory management unit and, thus, do not understand virtual addresses. Blocks of memory used only by software are fine using memory that is only virtually contiguous. Software will never know the difference. All memory appears to the kernel as logically contiguous.

Despite the fact that physically contiguous memory is only required in certain cases, most kernel code uses `kmalloc()` and not `vmalloc()` to obtain memory. Primarily, this is for performance. The `vmalloc()` function must specifically set up the page table entries. Worse, pages obtained by `vmalloc()` must be mapped by their individual pages, which results in much greater TLB (translation lookaside buffer) thrashing than when using direct mapped memory. Because of these concerns, `vmalloc` is only used when absolutely necessary, typically to obtain very large regions of memory. For example, modules that are dynamically loaded into the kernel, will be loaded into memory created by `vmalloc()`.

The `vmalloc()` function is declared in `< linux/vmalloc.h >` and defined in `mm/vmalloc.c`.

```
void * vmalloc(unsigned long size)
```

The function will return a pointer to at least `size` bytes of virtually contiguous memory. On error the function will return `NULL`. The function might sleep and therefore it cannot be called from an interrupt context or other situations where blocking is not permissible.

To free the allocation obtained by `vmalloc()` you can use the `vfree` function.

```
void vfree( void *addr )
```

This function can also sleep. It will free the block of memory beginning at addr that is previously allocated via vmalloc().

=> High memory mappings

By definition, pages in the high memory might not be permanently mapped into the kernels address space. Thus, pages obtained via alloc_pages() with the `__GFP_HIGHMEM` flag might not have a logical address.

On the x86 architecture, all physical memory beyond the 896MB mark is high memory and is not permanently and certainly not automatically mapped into the kernels address space, despite x86 processors being capable of physically addressing up the 4GB of physical RAM. Once allocated, these pages must be mapped into the kernels logical address space. On x86, pages in high memory are mapped somewhere between 3 and 4 GB mark.

Permanent

This is a main function to map a given page into the kernels address space:

```
void *kmap( struct page *page )
```

This function will work on both high and low memory. If the page structure belongs to a page in low memory, the pages virtual address is simply returned. If a page resides in high memory, a permanently mapping is created and the address is returned.

The number of permanently mapped pages is limited, so high memory should be unmapped when no longer needed. This can be done by calling this can be done by calling the next function:

```
void kunmap( struct page *page )
```

Temporary

When a mapping is needed but the context cant sleep, the kernel provides temporary or atomic mappings. These are a set of reserved permanent mappings that can hold a mapping on-the-fly. The kernel can atomically map a high memory page into one of the reserved mappings. A temporary mapping can be used in places that cannot sleep, such as interrupt handlers. A temporary mapping can be setup with the following function:

```
void *kmap_atomic( struct page *page, enum km_type type )
```

The type parameter describes the purpose of the temporary mapping. These types are defined in `<asm/kmap_types.h>`.

This function does not block and thus can be used in interrupt context and other places that cannot reschedule. It also disables kernel preemption; which is needed because the mappings are unique to each processor.

The mapping can be undone by the following function:

```
void kunmap_atomic( void *kvaddr, enum km_type type )
```

This function also does not block. In fact, in many architectures it does not do anything at all except enable kernel preemption, because temporary mapping is only valid until the next temporary mapping. Therefore the kernel can just forget about the `kmap_atomic` mapping and `kunmap_atomic()` does not need to do anything special. The next atomic mapping will then simply overwrite the previous one.

=> The slab layer

The slab layer is something very complex in the Linux kernel, and therefore I'm not going to deep into the working of it, I'm only going to try and explain what it does, and how it will do its job.

Allocating and freeing data structures is one of the most common operations inside any kernel. To make the frequent allocations and deallocations a bit easier, programmers often use free lists. A free list contains a block of available, already allocated, data structures. When code requires a new instance of a data structure, it can grab one of the structures off the free list instead of allocating a sufficient amount of memory and setting it up for the data structure. Later on, when the data structure is no longer needed, it is returned to the free list instead of deallocated. In this sense, the free list acts as an object cache, caching a frequently used type of object.

One of the main problems with free lists in the kernel is that there exists no global control. When available memory is low, there is no way for the kernel to communicate to every free list that it should shrink the size of its cache to free up memory. In fact, the kernel has no understanding of the random free lists at all. To remedy this, and to consolidate code, the Linux kernel provides the slab layer or slab allocator. The slab layer acts as a generic data structure-caching layer.

The slab layer was "invented" by Sun and first introduced in Sun Microsystems SunOS 5.4 operating system. The Linux slab layer shares the same name and the same basic design.

The slab layer attempts to fill some basic tasks:

- Data structures that are allocated and freed often need to be cached.
- If the cache is made per-processor, allocations and frees can be performed without an SMP lock.
- Improve performance during allocation and deallocation of frequently used objects.
- *[Illegible]*.

-Cereal from neworder.box.sk