Operating Systems 2

Memory Management in Linux

Arkadiusz Chrobot

Department of Computer Science

May 4, 2020

Outline

Introduction

The *Memory Management* subsystem is one of the most complex parts of the kernel. Its detailed description is given by Mel Gorman in his book “Understanding Linux Virtual Memory Manager”. This book is freely available, but today is a little outdated. The complexity of the memory management subsystem is caused by the fact, that Linux supports many hardware platforms, with miscellaneous memory systems. Some of them, like the x86 computer system family apply segmentation, others like the Alpha processors-based systems do not used this technique at all. Most of contemporary computer systems implement the virtual memory. There are however some embedded or real-time systems for which this solution is too resource-consuming or to unpredictable. The multiprocessor systems may use the UMA or NUMA organization of RAM. All those differences have to be addressed by the kernel programmers.

Low-Level Memory Management

The mainline Linux kernel uses paging as the most basic memory management system which is supported by most hardware platforms for which Linux is available. Kernel pages are not swappable, and for swapping the user pages Linux applies the PFRA (*Page Frame Reclaiming Algorithm*) which is a modified *Second Chance Algorithm* that strives to retain a pool of free frames. The algorithm is implemented in the *kwsap* thread, responsible for page swapping. The kernel uses the *multilevel page table*. Since the 64-bit hardware platforms have become more common, the table has 4 levels: The *Page Global Directory*, the *Page Upper Directory*, the *Page Middle Directory* and finally the *Page Table*. In some hardware platforms, like the 32-bit x86 family, the Page Global Directory and Page Upper Directory have only one entry. The virtual address has five parts. The first four, starting from the most significant bit, identify entries in, respectively, Page Global Directory, Page Upper Directory, Page Middle Directory and Page Table (see Figure 1 for reference). The last part — offset — identifies a byte inside a page.

Pages Table

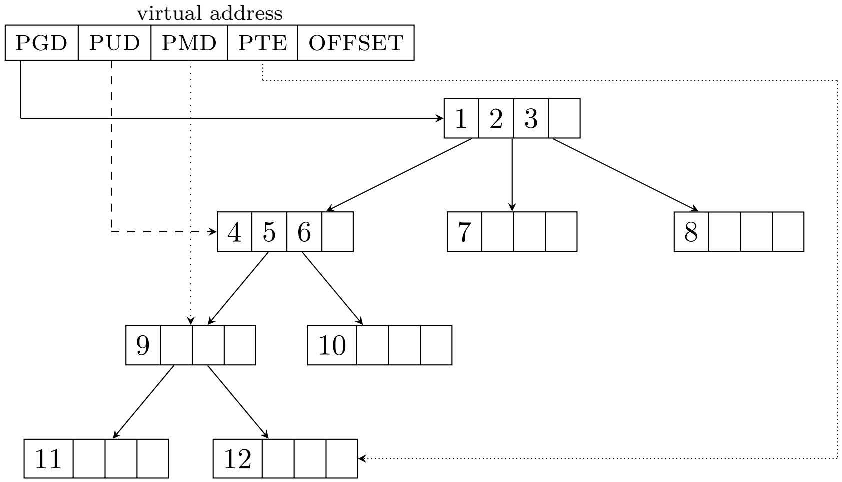

Figure 1 : Multilevel Page Table

Segmentation

In case of the 32-bit x86 hardware platforms the Linux kernel aside from pages uses also segments, but only for their memory protection system which aids a similar system for pages. Six segment descriptors are applied: the kernel code segment descriptor, the kernel data segment descriptor, the user code segment descriptor, the user data segment descriptor (TSS), the task segment descriptor and the local table descriptor (LTD). The first four cover the whole virtual memory (4 GiB). They only difference among them is in the memory access permissions. The TTS is used in process scheduling and its usage is very reduced. The LDT is needed only by MS Windows emulators, like WINE.

Zones

With each frame the Linux kernel associates a structure of the `struct page` type, which stores data about a page that occupies the frame. Among those data are reference counter, flags that describe the state of the page, address of a descriptor of the virtual address space where the page is mapped and the virtual address of the page. In case of some hardware platforms, most notably the x86 family, not all pages can be treated equally. That is why they are divided into zones. In the 32-bit x86 hardware platforms the following zones can be found: `ZONE_DMA`, `ZONE_NORMAL` and `ZONE_HIGHMEM`. The `ZONE_DMA` contains pages used for DMA transmissions by the ISA (*Industrial Standard Architecture*) bus based devices, that can only access the first 16 MiB of RAM (the bus is 24-bits wide). Moreover, this memory has to be physically continuous, because this devices cannot use the virtual memory. This zone exists only for historical reasons. The `ZONE_NORMAL` contains regular pages.

Zones

The Figure 2 will be helpful in explaining the need for `ZONE_HIGHMEM`. Starting from 2.6 series the Linux kernel splits the virtual address space into two parts in the 3 : 1 ratio[1] The first part is for user-space processes the second is for the kernel. The address that separates those two parts is defined as the `PAGE_OFFSET` constant. In case of 32-bit x86 hardware platforms the total virtual address space is 4 GiB (2^{32} B). It means that there are 3 GiB available to user processes and 1 GiB for the kernel. The virtual memory has to be mapped to the physical memory (the RAM). It's not a problem if the RAM size is equal of less than 1 GiB. When there is more RAM, than the kernel wouldn't be able to address it entirely. To remove this obstacle the kernel virtual memory address space is splat into two parts in such hardware platforms. The first part has the size of 896 MiB and contains both the `ZONE_DMA` and `ZONE_NORMAL`.

[1] Earlier it splat the address space in 2 : 2 ratio, just like MS Windows systems.

Zones

The virtual addresses from those zones are converted to physical addresses and in the other way by subtracting or adding the value of the `PAGE_OFFSET` constant. The last 128 MiB are pages without fixed virtual addresses. Those addresses can be assigned to them on demand, with a uses of a special array. That way they the kernel can map them to any part of the RAM that is above the 1 GiB limit. The three zones are used in the 32-bit x86 hardware platforms[2]. In some other platforms there is no need for all of them. The 64-bit platforms usually use `ZONE_DMA`, `ZONE_NORMAL` and `ZONE_DMA32`. The last contains pages that are used in DMA transmissions by the 32-bit PCI bus based devices, that can address only 4 GiB of RAM. As for now there is no need for the `ZONE_HIGHMEM` in such hardware platforms. Moreover, they do not use the whole 64 bits in addresses. They usually make use of 52 or even 48 of them, to reduce the gap between the user-space and kernel-space virtual memory.

[2]There is also another called `ZONE_MOVABLE`, but it won't be discussed here.

Zone HIGHMEM

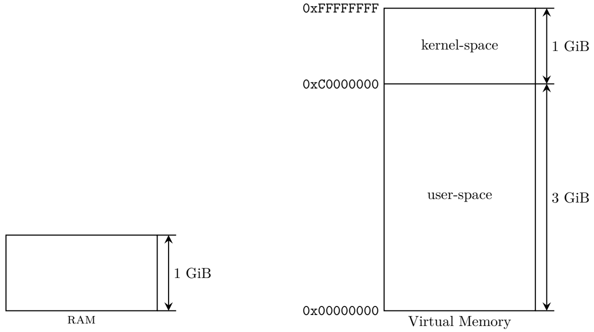

Figure 2 : Zone HIGHMEM Explanation

Zone HIGHMEM

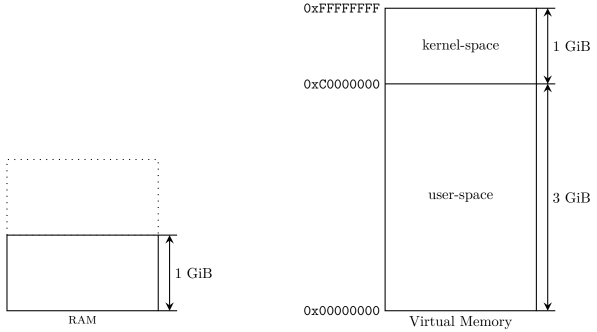

Figure 2 : Zone HIGHMEM Explanation

Zone HIGHMEM

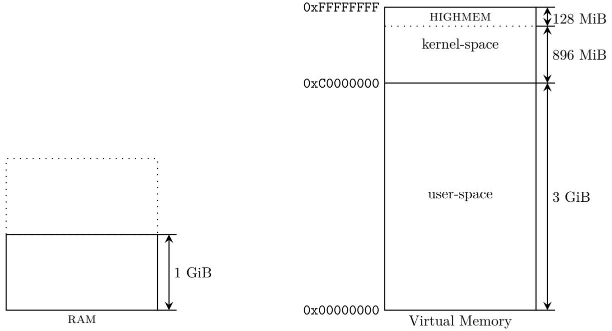

Figure 2 : Zone HIGHMEM Explanation

Zone Management

The kernel associates with every zone a structure of the `struct zone` type. Those are relatively big variables that stores such data, like the name of the zone and the number of free pages that are available in the zone[3]. The names are strings of characters ended with the `'\0'` character: “DMA”, “DMA32”, “Normal”, “HighMem”. Each structure is protected by a spin lock. The lock protects only data inside the variable, not the content of the pages in the zone. For the normal allocations the kernel by default allocates the memory from the normal zone, provided there are some free pages. Otherwise it uses the DMA zone or the HIGHMEM, if the former is also empty. If the zone is specified in the kernel memory allocator call, then the memory has to allocated from this zone.

[3]As it was stated before, the kernel tries to keep as many of them as possible.

The Low-Level Allocator

The Linux kernel has a low-level memory allocator that allocates physically continuous memory areas, which are needed for DMA transmissions by some devices and are helpful in reducing the frequency of TLB updates. It utilizes the *buddy memory system* for such allocations. It means that the allocator keeps for each of the zones a list of free adjoining memory frames, that form areas of the memory which sizes can be expressed as a power of two. For example if the kernel needs two pages of physically continuous memory and the only available area has four pages, then the allocator splits the area in two, each with two pages. One is allocated and the other is classified as a free area of the size of two pages. If the allocated pages are then freed, then the allocator merges the two adjoining areas into one free area of the size of four pages.

The Low-Level Allocator API

The low-level allocator has an API that consists of the following functions and macros used for allocating memory:

`alloc_pages(gfp_mask, order)` allocates 2^{order} pages and returns an address of the `struct page` structure that refers to the first of them,

`alloc_page(gfp_mask)` allocates a single page and returns an address of the `strut page` structure that refers to it,

`get_zeroed_page(gfp_mask)` allocates a single pages filled with zeros and returns its virtual address (used for allocating pages for user-space processes,

`__get_free_page(gfp_mask)` allocates a single page and returns its virtual address,

`__get_free_pages(gfp_mask, order)` allocates 2^{order} pages and returns the virtual address of the first of them.

The Low-Level Allocator API

If the structure of the `struct page` is available for the page, then its address may be acquired with the use of the `page_address()` function. The `gpf_mask` parameter is explained latter in this lecture. The allocated memory can be freed with the use of the following functions and macros:

`void __free_pages(struct page *page, unsigned int order)`
: releases the group of 2^{order} pages identified by the address of the `struct page` structure associated with the first of them,

`void free_pages(unsigned long addr, unsigned int order)`
: releases the group of 2^{order} pages identified by the virtual address of the first of them,

`free_page(addr)` releases one page identified by its virtual address,

`__free_page(page)` releases one page identified by the address of its `struct page` structure.

The `kmalloc()` function

The result of each operation of allocating memory should be verified. Starting with the version 2.6.31 the Linux kernel provides a mechanism for detecting memory leaks.
If a physically continuous memory region of arbitrary size is needed then the `kmalloc()` function can be used for allocating it. Its prototype is as follows:

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

The function allocates memory area of the size specified by the `size` parameter or bigger, never smaller. If it fails to allocate the memory it returns `NULL`. To free the memory the `kfree()` function has to be used. The prototype of the function is as follows:

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

The function checks only if its argument is `NULL`, so its up to the programmer to provide the correct input for that function. The `gfp_mask` parameter is used for passing flags which define the character of the allocation. They are divided into three categories: action modifiers, zone modifiers and type flags.

Type flags

The action modifiers define what actions the allocator can take while allocating the memory. The zone modifiers define the zone that has to be used for allocating the memory. Finally, the type flags are bitwise sums of some action and zone modifiers and are frequently used as arguments for allocator functions. Some of the are:

`GFP_ATOMIC` means a high priority allocation, that cannot sleep; usually used in the interrupt context,

`GFP_NOWAIT` similar to `GFP_ATOMIC`, but the memory pools are used in the allocation, to reduce the probability that it fails,

`GFP_NOIO` means that sleeping during the allocation is possible, but no block I/O operation can be performed; used by the code that performs block I/O operations to avoid deadlocks,

`GFP_NOFS` sleeping and I/O operations are possible during the memory allocation, as long as the operations do not involve file system,

Type flags

`GFP_KERNEL`	regular allocation of the memory for the kernel,
`GFP_USER`	regular allocation of the memory for the user-space process, sleeping is possible,
`GFP_HIGHUSER`	similar to `GFP_USER` but the memory is allocated in the high memory zone,
`GFP_DMA`	the memory is allocated in the DMA zone.

The `vmalloc()` and `vfree()` functions

If the allocated memory doesn't have to be physically continuous then the `vmalloc()` function can be used. Its prototype is as follows:

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

The allocated memory is freed with the use of the `vfree()` function of the following prototype:

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

Slab Allocator

The kernel often allocates and frees memory for various data structures that it needs. Such operations are time-consuming, so it is a good idea to created a buffer of such structures when the system is booting. Whenever such a structure would be needed the kernel would just take it from the buffer and then return it when it is no longer needed. This is the concept behind the *Slab Allocator* that was invented by Jeff Bonwick from SUN Microsystem and implemented for the first time in SunOS 5.4 operating system. Later it was adopted for Linux kernel.

Slab Allocator

The slab allocator tries to address the following issues:

- buffering the often used data structures is beneficial,
- frequent allocations and deallocations result in external memory fragmentation on pages/frames level, so the memory for buffers is physically continuous,
- allocating and freeing buffered data structures is fast,
- making a part of a buffer CPU-specific eliminates the need for locking when the allocations take a place in multiprocessor environment,
- coloring can be applied to prevent mapping multiple stored structures to the same cache line of CPU,
- in the NUMA systems the memory allocations can be performed on the same node, that initialized them.

Slab Allocator Implementation

In the slab allocator the data structure buffers are called caches. The Linux kernel creates two types of such caches: *general purpose* and dedicated. The first ones are used only by the allocator, the second caches are used for storing a specific data structure. For example there is one such a dedicated cache for process descriptors. Usually the names of the caches indicate what type of data structures it stores, for example: `task_struct_cachep`. The cache is built from slabs, which usually consists of many pages. Each slab stores a number of data structures, that are called *object* is slab allocator terminology. The slabs can be full, empty and partially empty. The objects are allocated from the partially empty slabs. If there are not available, then empty slabs are used. Each cache is represented by a structure of the `kmem_cache` type and each slab has its own descriptor, which is a structure of the `struct slab` type. The descriptors are stored in the general purpose caches or directly in the slabs. Slabs are created and destroyed by the slab allocator with the use of `kmem_getpages()` and `kmem_freepages()` functions.

Slab Allocator API

The programmer can create a new dedicated cache with the help of the `kmem_cache_create()` function. It takes five arguments. The first one is the name of the cache. The second one is the size of a single object. The third one defines the offset of the first object within a slab — usually it is zero. The fourth argument is a flags that defines the characteristics of the cache. It can be zero, a single value or a bitwise sum of several flags. The last argument is an address of a function which is a constructor for the object — it initializes the object when it is taken from the cache. If the function is not required then the `NULL` value can be passed as this argument. The `kmem_cache_create()` had another argument in earlier kernel versions, which was an address of a destructor — a function which cleaned the object when it was returned to the cache. It was not used, so the kernel programmers decided to remove it. The cache can be destroyed with the help of the `kmem_cache_destroy()` function.

Slab Allocator API

To allocate objects from the cache the `kmem_cache_alloc()` function can be used. Objects are deallocated with the help of the `kmem_cache_free()` function. For a more detailed description of the slab allocator and the low-level allocators APIs please refer to the second laboratory instruction.

Memory Pools

Memory pools are a special type of caches that are managed by the slab allocator. The pools assure that there will be available free memory, for critical allocations that cannot fail. Each memory pool is represented by a variable of the `mempool_t` type and it can be created with the help of the `mempool_create()` function, which takes four arguments. The first one is the minimal number of free objects that the pool should have. The second and third arguments are addresses to the functions that allocate and deallocate memory from the pool. The last argument is a pointer to the memory area where the pool should be created. Usually it is a cache. The programmer can provide her or his own functions that allocate and free objects, or the `mempool_alloc()` and `mempool_free()` can be used instead. Those subroutines use other slab allocator functions. The pool can be resized with the help of `mempool_resize()` or be destroyed with the use of `mempool_destroy()` function. More detailed description of the memory pools API is also available in the second laboratory instruction.

Slab Allocator Replacements

The slab allocator is ineffective for the embedded systems. For such hardware platforms it has been replaced with the *slob* (*Simple Linked List of Blocks*) since 2.6.23 version of the kernel. Also starting from this version another replacement of the slab allocator is available, but this time it is for the MPP (*Massively Parallel Processing*) systems. The replacement is called *slub* allocator and it uses single structures of the `struct page` type to describe groups of frames. It allows the MPP systems to save a lot of RAM.

NUMA Systems

Linux supports systems based on the *Non-Uniform Memory Access* (NUMA) architecture. In the UMA systems the kernel assumes that the memory is linearly addressed and that it belongs to a single NUMA node. The memory however don't have to be continuous, it may contain small gaps in the address space. For the 64-bit x86 hardware platforms the kernel can be compiled with an option enabled that allows it to emulate a NUMA system. This option is useful for testing software for such systems. For the true NUMA systems are available two compilation options. The first one is called `DISCONTIGMEM` and it enables the basic support for a noncontinuous memory of the NUMA architecture, which can also be applied to the memory of UMA architecture with big gaps in the address space. The second option is called `SPARSEMEM` and it enables experimental support of the NUMA architecture, which offers additional features, but it can be unstable and it shouldn't be used in production environments. Each NUMA node has its own set of zones and its own `kswapd` daemon.

Miscellaneous Memory Management Issues

The size of the kernel process stack is only two pages (8 KiB for computers based on x86 CPUs and 16 KiB for hardware platforms based on Alpha CPUs). It means that the memory within this stack should be carefully allocated. The stack is used by system calls, regular kernel functions, interrupt handlers and many other parts of the kernel code. The programmers should avoid using large data structures as local variables for functions. If it is necessary then those structures should be declared with the use of the `static` keyword. There is kernel compiling option that enables reducing the size of the kernel process stack to only one page. It can be beneficial for the MPP systems. In that case the interrupt and softirq handlers get its own stack also of the size of a single page.

Miscellaneous Memory Management Issues

The high memory zone pages don't have fixed virtual addresses. Such a page can be mapped to the virtual addresses space with the help of the `kmap()` function. This mapping can be removed with the use of the `kunmap()` function. If the mapping has to be done or removed in the interrupt context then the `kmap_atomic()` and `kunmap_atomic()` functions can be applied.

Questions

?

The End

Thank You for Your attention!