Laboratory 2: "Dynamic memory allocation in Linux kernel space"
(one week)

Arkadiusz Chrobot, PhD
Karol Tomaszewski PhD

March 8, 2024

Contents

Introduction

This guide provides information on how to use dynamic memory in the Linux kernel. Chapter 1 provides general information about the mechanisms of dynamic memory management in the Linux kernel, which are briefly called the *allocators*. Chapter 2 contains a description and how to use the zone allocator interface. Chapter 3 contains information about the slab allocator interface and examples of using it. Chapter 4 describes other ways of allocating and releasing memory, which, although associated with both allocators, cannot be clearly qualified as part of one of them. The document ends with a chapter with tasks for self-implementation.

1. Memory allocators

The Linux kernel has allocators that allow dynamic management of both its memory and memory belonging to user processes. The most important of these are the zone allocator and the slab allocator. The first one is a low-level mechanism that allocates memory based on the *buddy system* algorithm, which registers physically continuous areas of memory and allocates it with portions whose size is expressed by the power of 2 multiplied by the page size. Thus, the basic memory unit for the *buddy system* algorithm is *memory page*. The name of the allocator comes from the fact that physical memory in Linux can be divided into zones for which this allocator maintains separate records of free areas. Memory division into zones depends on the hardware platform. For example, on 32-bit pc computers, memory is typically divided into three zones:

1. DMA allocations, the area where buffers can be created for peripheral devices using DMA transmission,
2. *normal* allocations, the area from which memory is allocated for typical needs of kernel and the user's processes,
3. *highmem* allocations, the area that is not directly addressed by the kernel.

There is only one zone for the Raspberry Pi 2 computer — the normal allocations zone. The operating system kernel must often allocate and release memory for the various data structures it uses. Typical solutions used for these operations are too inefficient and low performance. Therefore, Linux uses a slab allocator, which creates memory caches[1], which are specific stores of data structures used by the kernel. This allocator uses the zone allocator services to create memory areas filled with structures of a given type, e.g. process descriptors, memory or buffers for the network subsystem. These areas are called slices (slabs) and together they form a cache. They are created before other kernel subsystems request the allocation of even a single structure needed for their operation. Thanks to this, creating a new structure is based on passing the pointer to the existing structure placed in the memory slab, to the subsystem

[1] Please do not confuse them with the processor's hardware caches.

that requested it. The release of structure consists in marking this structure as available for subsequent allocations. In addition, the slab allocator treats all structures as objects and allows them to be initialized using a special function that works as a constructor[2]. There is also a special type of memory cache, which is called *memory pool*. It ensures that the collection of free structures is not used up and the allocation of a new structure will always be possible. They are used in critical allocations, that means the ones that should never fail.

2. Zone allocator

In the Linux kernel, there are five macros and functions defined in the `linux/gfp.h` header file, which are the interface of the zone allocator, i.e. they allow the allocation and release of memory areas through it.

`unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)` - This function allocates a 2^{order} memory area and returns its beginning address. If the assignment fails, it returns zero. The `order` parameter is an exponent of the power specifying the number of pages allocated, e.g. to allocate one page you must pass the number `0` through this parameter. The first parameter is used to pass so-called type marker. It defines what actions can be performed during the allocation and from which zone it will be made. This guide will use the marker for normal kernel memory allocations, i.e., `GFP_KERNEL`

`__get_free_page(gfp_mask)` - A macro that allows to allocate only one page. As an argument, it takes a type marker and it returns a page address, or zero if the assignment fails.

`unsigned long get_zeroed_page(gfp_t gfp_mask)` - This function assigns a single page whose content is zeroed. It is used to make allocations for user space. The return value is the page address or zero if the assignment fails.

`struct page *alloc_pages(gfp_t gfp_mask, unsigned int order)` - This function is an `inline` function and takes the same arguments as `__get_free_pages()`, but instead of the address of the allocated area, it returns the address of the `struct page` structure. Each such structure is associated with a single frame in the computer's physical memory. To get the address of the page residing in it, use the `page_address` macro, which will be described below.

`alloc_page(gfp_mask)` - This macro takes the same argument as `__get_free_page`, but returns the `struct page` address instead of the page address directly.

There may be zones in the physical Linux memory that contain pages without a permanent virtual address. However, these pages reside in memory frames, and each frame has a `struct page` stucture associated with it. Having the pointer on such a structure, it is possible to refer to the page contained in the frame associated with it and obtain its address using the `page_address()` macro. It is defined in the `linux/mm.h` header file and takes the `struct page` structure address as argument. It returns the page address afterwards. There are also macros: `IS_ERR` and `PTR_ERR` associated with the `alloc_pages()` and `alloc_page`. Both take as an argument the addresses returned by the mentioned function (macro). The `IS_ERR` macro checks if these addresses indicate a memory allocation failure. If so, it returns a non-zero value, and if not, then the macro returns zero. The `PTR_ERR` macro allows to specify the exception code for the wrong address that appeared when attempting to allocate memory. The memory allocated by the macros and functions described above can be released using:

`free_page(addr)` - releases memory allocated with `__get_free_page()` or `get_zeroed_page()`. Through the `addr` parameter the address of the page to be released is passed.

`void free_pages(unsigned long addr, unsigned int order)` - releases memory allocated by `__get_free_pages()`. As the first call argument it takes the address of the area to be released, and as the second call argument it takes the exponent of the power of 2 determining the number of pages to be released.

[2]Earlier versions of the kernel also allowed to define a destructor, but developers did not take advantage of this possibility and eventually it was removed.

`__free_page(page)` - releases memory allocated by `alloc_page`. As a call argument, it takes the `struct page` structure address.

`void __free_pages(struct page *page, unsigned int order)` - releases memory allocated by `alloc_pages()`. The first call argument is the `struct page` structure address, and the second call argument is the same as for `free_pages()`.

Listing 1 contains the code of the module demonstrating the use of the zone allocator. The memory is allocated in the module initializing function and released in the cleaning function.

Listing 1: Using a zone allocator

```
#include<linux/module.h>
#include<linux/gfp.h>
#include<linux/mm.h>

#define ORDER 2

static struct page *page_pointer, *pages_pointer;
static unsigned long int single_page_address, zeroed_page_address,
                    multiple_pages_address;


static int __init mallocmod_init(void)
{
        pages_pointer = alloc_pages(GFP_KERNEL,ORDER);
        if(IS_ERR(pages_pointer)) {
                pr_alert("Error allocating %u pages: %ld!\n",1<<ORDER,
                                PTR_ERR(pages_pointer));
                return -ENOMEM;
        }
        pr_notice("Pages address: %p\n",page_address(pages_pointer));

        page_pointer = alloc_page(GFP_KERNEL);
        if(IS_ERR(page_pointer)) {
                pr_alert("Error allocating page: %ld!\n",PTR_ERR(page_pointer));
                return -ENOMEM;
        }
        pr_notice("Pages address: %p\n",page_address(page_pointer));

        multiple_pages_address = __get_free_pages(GFP_KERNEL,ORDER);
        if(!multiple_pages_address) {
                pr_alert("Error allocating %u pages!\n",1<<ORDER);
                return -ENOMEM;
        }
        pr_notice("Pages address: %lx\n",multiple_pages_address);

        single_page_address = __get_free_page(GFP_KERNEL);
        if(!single_page_address) {
                pr_alert("Error allocating page!\n");
                return -ENOMEM;
        }
        pr_notice("Page address: %lx\n",single_page_address);

        zeroed_page_address = get_zeroed_page(GFP_KERNEL);
        if(!zeroed_page_address) {
```

```
                pr_alert("Error allocating zeroed page!\n");
                return -ENOMEM;
        }
        pr_notice("Page address: %lx\n",zeroed_page_address);

        return 0;
}

static void __exit mallocmod_exit(void)
{
        if(zeroed_page_address)
                free_page(zeroed_page_address);
        if(single_page_address)
                free_page(single_page_address);
        if(multiple_pages_address)
                free_pages(multiple_pages_address,ORDER);
        if(page_pointer)
                __free_page(page_pointer);
        if(pages_pointer)
                __free_pages(pages_pointer,ORDER);
}

module_init(mallocmod_init);
module_exit(mallocmod_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
MODULE_DESCRIPTION("A module demonstrating the usage of zone allocator.");
MODULE_VERSION("1.0");
```

The exponent of the power specifying the number of pages to be allocated for `alloc_pages()` and `get_pages()` calls was determined using the `ORDER` constant variable. If the assignment succeeds, the address of the allocated area is displayed, and in case of failure, the exception code is displayed and the function initializing the module return error specified by the expression `-ENOMEM`. The `ENOMEM` constant variable is foreseen for exceptions caused by insufficient memory or allocation failure. It should be noted that the memory allocated by the zone allocator is physically continuous.

3. Slab allocator

The slab allocator uses two mechanisms to allocate the structures needed for other kernel subsystems - the cache memory and memory pools. The API of both of them will be described later in this chapter.

3.1. Cache memory

This part of the guide describes the functions that are used to manage cache memory and to allocate and release structures from those caches. They are all defined in the `linux/slatb.h` file.

`struct kmem_cache *kmem_cache_create(const char *, size_t, size_t, unsigned long, void (*)(void *))`
- this function creates a new cache and returns a pointer to its descriptor, i.e. `struct kmem_cache`. It takes five call arguments to invoke. The first is a string containing the human-readable name of the cache memory. It is presented in the `/proc/slabinfo` file containing statistical data of all caches. The second argument is the size of a single object (structure) stored in the memory cache. The third argument is the value of offset of the first object relative to the beginning of the slab. Usually its value is `0`. The fourth argument contains flags. Developers have several flags at their disposal that can be passed individually or logically summed:

`SLAB_HWCACHE_ALIGN` - the flag aligns all objects in the slabs to the line size in the hardware cache. This setting increases the performance of the slab allocator, but also increases memory usage. It is recommended to use it only for caches of objects that must be accessible quickly.

`SLAB_POISON` - causes the slabs to be filled with a specific value to facilitate the recognition and interception of attempts to access uninitialized memory.

`SLAB_RED_ZONE` - setting this flag makes it easier to detect „buffer overrun" errors.

`SLAB_PANIC` - setting this flag causes the object allocation failure to end in a critical kernel error - *kernel panic*.

`SLAB_CACHE_DMA` - instructs the allocator to allocate memory for slabs in the DMA zone.

The last argument is a pointer to a function that does not return any value, but takes one parameter of type `void *`. This function plays the role of a constructor, i.e. it initializes every structure created in the memory cache. The programmer does not have to define it. If he doesn't want to do this, he can pass `NULL` or simply `0` as the last parameter of the `kmem_cache_create()` function call.

`void *kmem_cache_alloc(struct kmem_cache *, gfp_t flags)` - this function is used to allocate a single structure (object) from the cache. It returns a pointer of type `void *`. Its correctness can be checked using the `is_err` macro, and the possible exception code can be checked with the `ptr_err` macro. The function accepts two call arguments: the cache pointer to a cache memory that should be used for allocation and the type from m to allocate and the type marker.

`void kmem_cache_free(struct kmem_cache *, void *)` - this function releases the structure (object) allocated from the cache memory to which the pointer is passed as its first call argument. The pointer to the released structure is passed as a second call argument.

`void kmem_cache_destroy(struct kmem_cache *)` - this function removes the cache memory to which the pointer was passed to it as a call argument.

Listing 2 contains the source code of the module, which creates a cache memory for sample structures and allocates and then releases a single structure of this type.

Listing 2: Using the slab allocator - cache memory

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

static struct example_struct {
        unsigned int id;
        char example_string[10];
} *example_struct_pointer;

static struct kmem_cache *example_cachep;

static void example_constructor(void *argument)
{
        static unsigned int id;
        static char test_string[] = "Test";
        struct example_struct *example = (struct example_struct *)argument;
        example->id = id;
        strcpy(example->example_string,test_string);
        id++;
}

void print_example_struct(struct example_struct *example)
{
```

```
24          pr_notice("Example struct id: %u\n",example->id);
25          pr_notice("Example string field content: %s\n",example->example_string);
26  }
27
28  static int __init slabmod_init(void)
29  {
30          example_cachep = kmem_cache_create("example cache",
31                          sizeof(struct example_struct),0,
32                          SLAB_HWCACHE_ALIGN|SLAB_POISON|SLAB_RED_ZONE,
33                          example_constructor);
34          if(IS_ERR(example_cachep)) {
35                  pr_alert("Error creating cache: %ld\n",PTR_ERR(example_cachep));
36                  return -ENOMEM;
37          }
38
39          example_struct_pointer = (struct example_struct *)
40                  kmem_cache_alloc(example_cachep,GFP_KERNEL);
41          if(IS_ERR(example_struct_pointer)) {
42                  pr_alert("Error allocating form cache: %ld\n",
43                                  PTR_ERR(example_struct_pointer));
44                  kmem_cache_destroy(example_cachep);
45                  return -ENOMEM;
46          }
47
48          return 0;
49  }
50
51  static void __exit slabmod_exit(void)
52  {
53          if(example_cachep) {
54                  if(example_struct_pointer) {
55                          print_example_struct(example_struct_pointer);
56                          kmem_cache_free(example_cachep,example_struct_pointer);
57                  }
58                  kmem_cache_destroy(example_cachep);
59          }
60  }
61
62  module_init(slabmod_init);
63  module_exit(slabmod_exit);
64
65  MODULE_LICENSE("GPL");
66  MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
67  MODULE_DESCRIPTION("A module demonstrating the use of the slab allocator.");
68  MODULE_VERSION("1.0");
```

Lines 5-8 define the type of sample structure for which the module will create the cache memory. In addition, an indicator for this structure was declared in row 8. In line 10 there is a definition of a pointer to the cache memory. By the convention adopted by the Linux kernel programmers, all such pointers have a name ending in „ep”. The `example_constructor()` function (lines 12-20) is responsible for initializing each structure in the local cache. Such a structure (object) is passed to it by the `argument` parameter, which is a pointer of type `void *`, which in line 16 is cast onto a local pointer of type `struct example`. In addition, this function has two static local variables. The first is of type `int` and the second is an array of characters. Making the variable `id` local allows to increase its value during subsequent function calls. As a result, the `id` fields of subsequent structures will receive values that are consecutive natural

numbers. From the other hand the `example_string` field will receive the same value, i.e. the string „`Test`”, copied from the `test_string` variable. This variable is static for a another reason than the variable `id`. **The Linux kernel has a stack size of two pages. In the case of PCs, it is 8 KiB, so you need to use this memory sparingly. It is important to avoid creating large local variables or create them as static variables.** The `print_example()` function defined on lines 22-26 is used to put the contents of the sample structure into the kernel buffer. The rest of the code contains calls to the functions described earlier. Please note that the module first creates the cache memory and then the sample structure, but they are released in the reverse order - first the structure (object) is released and then the cache memory. Please also note the handling of exception in `kmem_cache_alloc()` function. If the allocation fails, not only is the allocation error signalled, but the cache memory is released as well.

3.2. Memory pools

There are two functions for managing memory pools that become available in the kernel module when the `linux/mempool.h` header file is included. Those are:

`mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn, mempool_free_t *free_fn, void *pool_data)` - this function returns a pointer to the created memory pool (of type `mempool_t`). If the allocation fails, this can be detected using the `IS_ERR` macro and the exception code can be obtained from this pointer using `PTR_ERR` macro. The function takes four call arguments: a value specifying the minimum number of objects that must always be available, a pointer to the function that allocates the object, a pointer to the function releasing the object, and a pointer to the memory area from which the pool can be created. Usually the last argument is a pointer to the cache memory. It is not necessary to define custom allocation and release functions. Two pointers named `mempool_alloc_slab` and `mempool_free_slab` are defined, which indicate ready to use allocation and release functions named `mempool_alloc()` and `mempool_free()`, respectively. It is enough to pass them as the second and the third call argument of the `mempool_create()` function to use the functions they indicate.

`void mempool_destroy(mempool_t *pool)` - this function deletes the memory pool to which the pointer is passed as the function call argument.

Listing 3 contains a module illustrating how to use a memory pool.

Listing 3: Using the slab allocator - memory pool

```
1  #include<linux/module.h>
2  #include<linux/slab.h>
3  #include<linux/mempool.h>
4  #include<linux/string.h>
5
6  #define MINIMUM_MEMPOOL_OBJECTS 10
7
8  static struct example_struct {
9          unsigned int id;
10         char example_string[10];
11 } *example_struct_pointer;
12
13 static struct kmem_cache *example_cachep;
14 static mempool_t *mempool_pointer;
15
16 static void example_constructor(void *argument)
17 {
18         static unsigned int id;
19         static char test_string[] = "Test";
20         struct example_struct *example = (struct example_struct *)argument;
```

```
        example->id = id;
        strcpy(example->example_string,test_string);
        id++;
}

void print_example_struct(struct example_struct *example)
{
        pr_notice("Example struct id: %u\n",example->id);
        pr_notice("Example string field content: %s\n",example->example_string);
}

static int __init slabmod_init(void)
{
        example_cachep = kmem_cache_create("example cache",
                sizeof(struct example_struct),0,
                SLAB_HWCACHE_ALIGN|SLAB_POISON|SLAB_RED_ZONE,
                example_constructor);
        if(IS_ERR(example_cachep)) {
                pr_alert("Error creating cache: %ld\n",PTR_ERR(example_cachep));
                return -ENOMEM;
        }

        mempool_pointer = mempool_create(MINIMUM_MEMPOOL_OBJECTS,
                        mempool_alloc_slab,mempool_free_slab,example_cachep);
        if(IS_ERR(mempool_pointer)) {
                pr_alert("Error creating cache: %ld\n",PTR_ERR(mempool_pointer));
                kmem_cache_destroy(example_cachep);
                return -1;
        }

        example_struct_pointer = (struct example_struct *)
                                mempool_alloc(mempool_pointer,GFP_KERNEL);
        if(IS_ERR(example_struct_pointer)) {
                pr_alert("Error allocating form cache: %ld\n",
                                        PTR_ERR(example_struct_pointer));
                mempool_destroy(mempool_pointer);
                kmem_cache_destroy(example_cachep);
                return -ENOMEM;
        }

        return 0;
}

static void __exit slabmod_exit(void)
{
        if(example_cachep) {
                if(mempool_pointer) {
                        if(example_struct_pointer) {
                                print_example_struct(example_struct_pointer);
                                mempool_free(example_struct_pointer,mempool_pointer);
                        }
                        mempool_destroy(mempool_pointer);
                }
                kmem_cache_destroy(example_cachep);
        }
```

```
76 }
77
78 module_init(slabmod_init);
79 module_exit(slabmod_exit);
80
81 MODULE_LICENSE("GPL");
82 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
83 MODULE_DESCRIPTION("A module demonstrating the use of a mempool.");
84 MODULE_VERSION("1.0");
```

As you can easily tell, this module is a converted version of the one that demonstrated the use of memory cache. The elements that have been added are creating a memory pool based on previously created memory cache and deleting it. Please note the calls to the `mempool_alloc()` function (line 52) and `mempool_free()` function. The first takes two arguments: a pointer to the memory pool from which the object is to be allocated and a type marker, and it returns a pointer of type `void *`. The second function returns nothing, but takes the pointer to the structure to be released and a pointer to the memory pool.

4. Other methods of allocating and releasing memory

There are four more functions that are used to dynamically manage memory in the kernel space. They are described in Table 1.

Prototype	**Description**
`void *kmalloc(size_t size, gfp_t flags)`	This function is available after including the `linux/slab.h` header file in the code. It allocates physically continuous memory. The first call argument is the size of the created area, and the second call argument is the type marker. The function returns a pointer to the allocated area or exception information, which can be handled using the `IS_ERR` and `PTR_ERR` macros. Because this function uses a zone allocator, the allocated area is always a multiple of the page size, but the programmer should use only as much of that area as he specified in the first call argument of the function.
`void kfree(const void *)`	This function is used to release the memory allocated with `kmalloc()` and only with this function this memory can it be released.
`void *vmalloc(unsigned long size)`	This is a function that allocates virtually continuous, but not necessarily physically continuous, memory. It can be used after including the `linux/vmalloc.h` header file in the code. It accepts one call argument, which is the amount of memory to be allocated. The return value is the address of the allocated memory or an error value that can be interpreted using the `IS_ERR` and `PTR_ERR` macros.
`void vfree(const void *addr)`	This function releases the memory allocated with `vmalloc()` and only this function should be used to release such memory. It does not return any value, but accepts one call argument which is the pointer to the released memory area.

Table 1: Other functions related to dynamic memory management in the kernel space

Physically continuous memory areas are mainly required to create buffers by device drivers that use DMA transmission. However, in kernel code, memory allocated with `kmalloc()` is more likely to be found than `vmalloc()`, because access to it is more efficient. Access to memory allocated by `vmalloc()` requires address translation using a page table. The `vmalloc()` function is therefore more commonly used in code related to memory allocation for user space. Listing 4 contains the module code that allocates and releases memory with the use of functions described in Table 1.

Listing 4: The use of `kmalloc()` and `vmalloc()`

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

#define NUMBER_OF_ELEMENTS 20
#define MEMORY_SIZE NUMBER_OF_ELEMENTS*sizeof(int)

static int *first_array, *second_array;


static int __init mallocmod_init(void)
{
        first_array = (int *)vmalloc(MEMORY_SIZE);
        if(IS_ERR(first_array)) {
                pr_alert("Error allocating memory with the use of vmaloc(): %ld\n",
                PTR_ERR(first_array));
                return -ENOMEM;
        }
        pr_notice("The address in first_array: %p\n",first_array);

        second_array = (int *)kmalloc(MEMORY_SIZE,GFP_KERNEL);
        if(IS_ERR(second_array)) {
                pr_alert("Error allocating memory with the use of vmaloc(): %ld\n",
                PTR_ERR(second_array));
                return -ENOMEM;
        }
        pr_notice("The address in second_array: %p\n",second_array);

        return 0;
}

static void __exit mallocmod_exit(void)
{
        int i;

        if(second_array) {
                pr_notice("second_array content:\n");
                for(i=0;i<NUMBER_OF_ELEMENTS;i++)
                        pr_notice("%d ",second_array[i]);
                pr_notice("\n");
                kfree(second_array);
        }

        if(first_array) {
                pr_notice("first_array content:\n");
                for(i=0;i<NUMBER_OF_ELEMENTS;i++)
                        pr_notice("%d ",first_array[i]);
```

```
48                 pr_notice("\n");
49                 vfree(first_array);
50         }
51 }
52
53 module_init(mallocmod_init);
54 module_exit(mallocmod_exit);
55
56 MODULE_LICENSE("GPL");
57 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
58 MODULE_DESCRIPTION("A module demonstrating the usage of different kernel\
59         memory allocators.");
60 MODULE_VERSION("1.0");
```

Exercises

1. [3 points] Change the module from Listing 2 so that it assigns five sample structures and writes their contents to the kernel buffer. Check by examining the printed `id` values of these structures to see if the order in which the objects are allocated from the memory slab is the same as the order of their initialization.

2. [5 points] Write a module in which you will create a stack in the constructor, and print its contents and delete it in the destructor.

3. [7 points] Write a module in which you will create a doubly-linked list in the constructor, and in the destructor print its contents in both directions and delete this list.

4. [3 points] Make the changes described in the first task to the module from Listing 3.

5. [5 points] Write a module in which you will create a FIFO queue in the constructor, and print its contents and delete it in the destructor.

6. [7 points] Write a module in which you will create a cyclic, singly-list in the constructor, and print its contents and delete it in the destructor. The cyclic, singly-linked list is a singly-linked list, whose nodes are linked in a circle, and no pointer is empty.