

Studio #11: Kernel Memory Management

In this studio, you will:

1. Allocate and deallocate memory for different numbers of objects, using the kernel-level page allocator.
2. Use address translation structures to manipulate the memory allocated via the kernel-level page allocator.

Required Exercises

1. List the names of the people who worked together on this studio.

Ans: This studio is finished by Xingjian Xuanyuan alone.

2. Compile your module, choose any positive value for `nr_structs` and load the module on your Raspberry Pi and check that the message appears in the system log. Now, inside your module, declare a `struct` type that contains an array of 8 unsigned integers. In your module's thread function, print a second message to the system log. Recompile your module and load it on your Raspberry Pi and show `dmesg` traces that print out these values.

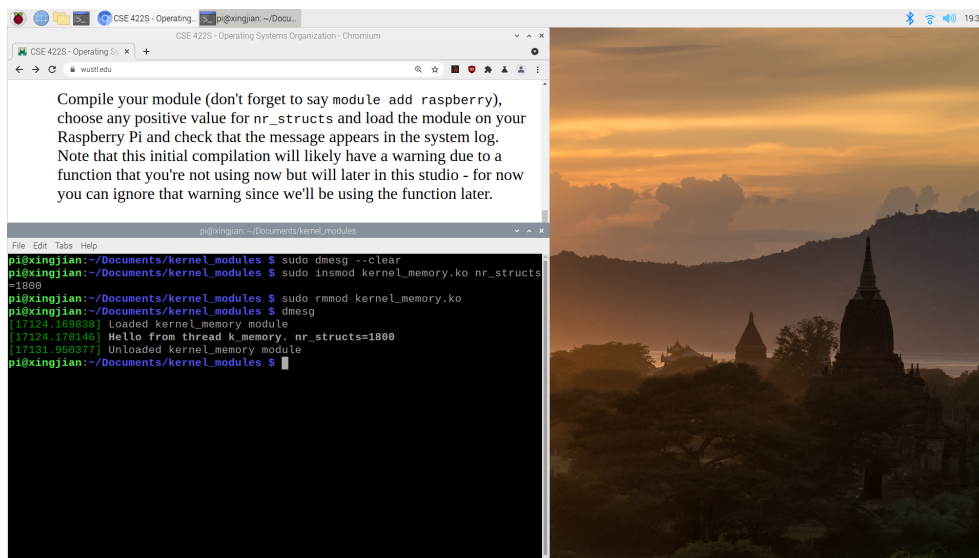


Figure 1: Results from the initial compilation of `kernel_memory.c`

Ans: Note that the `0644` parameter specified in `module_param` means that owner can read and write, group can read, everyone else can read the value of `nr_structs`. The `printk` function:

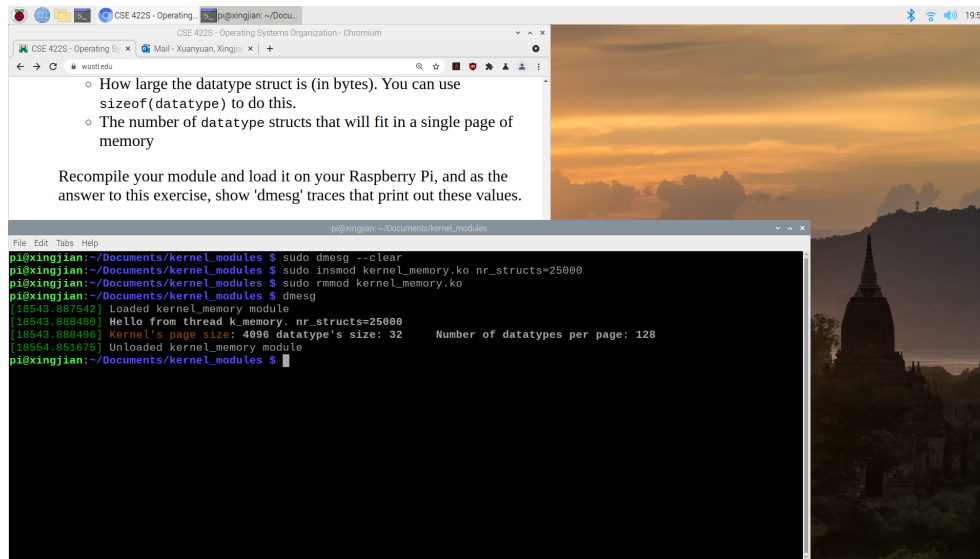


Figure 2: Second system log message added

```
printk("Kernel's page size: %lu\tdatatype's size: %zu\tNumber of
↳ datatypes per page: %lu\n", PAGE_SIZE, sizeof(datatype),
↳ PAGE_SIZE/sizeof(datatype));
```

has been added to the thread function.

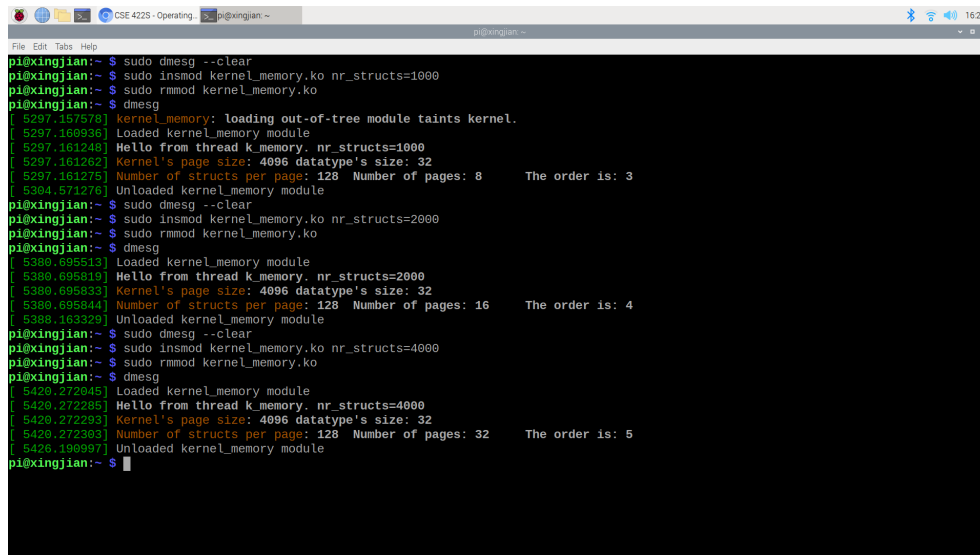
3. Inside your kernel thread function, you are going to allocate memory that can hold the user-configurable number (`nr_structs`) of datatype structs. Modify your module to include three new static unsigned int variables, `nr_pages`, `order`, and `nr_structs_per_page` that your kernel thread function should calculate based on the user-specified `nr_structs`. Modify your kernel thread to print out the value of these three variables. Explain what the function `my_get_order` does and show the relevant `dmesg` output when the module is loaded with 1000, 2000, 4000 as `nr_structs`.

Ans: For this exercise, `nr_pages` is calculated by:

```
nr_pages = nr_structs / nr_structs_per_page;
if (nr_structs % nr_structs_per_page > 0)
    nr_pages++;
```

so that enough pages will be allocated for all datatype structs. This value is then passed to the `my_get_order` function to calculate $\log_2 nr_pages$. Inside the `my_get_order` function, we want to calculate the order by counting the times of 1-bit right shift needed to reduce the value of `nr_pages` to zero. The value of `nr_pages` is checked: if `nr_pages == 0`, then the order is *zero*; if `nr_pages` is a power of 2, `value & (value - 1)` should equal zero, and the value of `nr_pages` must be decreased by one before going to the while loop. As shown by Figure 3, the order values are 3, 4, and 5 for the values of `nr_structs` specified as 1000, 2000, and 4000, respectively. The GFP flags control the allocators behavior. They tell what memory zones can be used, how hard the allocator should try to find free memory, whether the memory can be accessed by the userspace, etc. GFP_KERNEL is performed on behalf of a process running in kernel space. Using GFP_KERNEL means that `kmalloc`

can put the current process to sleep waiting for a page when called in low-memory situations. While the current process sleeps, the kernel takes proper action to locate some free memory, either by flushing buffers to disk or by swapping out memory from a user process.



```

pi@xingjian:~$ sudo dmesg --clear
pi@xingjian:~$ sudo insmod kernel_memory.ko nr_structs=1000
pi@xingjian:~$ sudo rmmod kernel_memory.ko
pi@xingjian:~$ dmesg
[ 5297.157578] kernel_memory: loading out-of-tree module taints kernel.
[ 5297.160936] Loaded kernel_memory module
[ 5297.161248] Hello from thread k_memory. nr_structs=1000
[ 5297.161262] Kernel's page size: 4096 datatype's size: 32
[ 5297.161275] Number of structs per page: 128 Number of pages: 8      The order is: 3
[ 5304.571276] Unloaded kernel_memory module
pi@xingjian:~$ sudo dmesg --clear
pi@xingjian:~$ sudo insmod kernel_memory.ko nr_structs=2000
pi@xingjian:~$ sudo rmmod kernel_memory.ko
pi@xingjian:~$ dmesg
[ 5380.695513] Loaded kernel_memory module
[ 5380.695819] Hello from thread k_memory. nr_structs=2000
[ 5380.695833] Kernel's page size: 4096 datatype's size: 32
[ 5380.695844] Number of structs per page: 128 Number of pages: 16     The order is: 4
[ 5380.163329] Unloaded kernel_memory module
pi@xingjian:~$ sudo dmesg --clear
pi@xingjian:~$ sudo insmod kernel_memory.ko nr_structs=4000
pi@xingjian:~$ sudo rmmod kernel_memory.ko
pi@xingjian:~$ dmesg
[ 5420.272045] Loaded kernel_memory module
[ 5420.272285] Hello from thread k_memory. nr_structs=4000
[ 5420.272293] Kernel's page size: 4096 datatype's size: 32
[ 5420.272303] Number of structs per page: 128 Number of pages: 32    The order is: 5
[ 5426.190997] Unloaded kernel_memory module
pi@xingjian:~$

```

Figure 3: Numbers of pages and their orders calculated

4. Because code must always access memory via its virtual address, and we currently only have `struct page *` to identify the memory, we must perform the following operations before accessing the memory:

- Convert from `struct page *` to a page frame number
- Convert from a page frame number to a physical memory address
- Convert from a physical memory address to a virtual memory address

Now we are going to have your module's thread function iterate through each struct in each physical memory page and set each integer value to a specific number. Finally, also modify the inner k loop to print out the value of every element for which both j and k are zero. Compile and then load and unload your module on your Raspberry Pi with the value of 200 and please show the values printed in `dmesg`.

Ans: My implementation of the above requirements is presented here:

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
#include <linux/gfp.h>
#include <linux/slab.h>
#include <linux/list.h>
#include <linux/time.h>

```

```

pi@xingjian:~/Documents/kernel_modules $ sudo dmesg --clear
pi@xingjian:~/Documents/kernel_modules $ sudo insmod kernel_memory.ko nr_structs=200
pi@xingjian:~/Documents/kernel_modules $ sudo rmmod kernel_memory.ko
pi@xingjian:~/Documents/kernel_modules $ dmesg
[ 7617.530092] Loaded kernel_memory module
[ 7617.530326] Hello from thread k_memory. nr_structs=200
[ 7617.530341] Kernel's page size: 4096 datatype's size: 32
[ 7617.530355] Number of structs per page: 128 Number of pages: 2      The order is: 1
[ 7617.530373] The first element in the 0th page is 0
[ 7617.530396] The first element in the 1th page is 1024
[ 7625.850463] Unloaded kernel_memory module
pi@xingjian:~/Documents/kernel_modules $

```

Figure 4: The value of the first element of each allocated page

```

#include <linux/kthread.h>
#include <linux/mm.h>

#include <asm/uaccess.h>

#define ARR_SIZE 8
typedef struct datatype_t {
    unsigned int array[ARR_SIZE];
} datatype;
static struct datatype_t * dt = NULL;

static uint nr_structs = 2000;
module_param(nr_structs, uint, 0644);

static struct task_struct * kthread = NULL;
static struct page * pages = NULL;

static unsigned int nr_structs_per_page;
static unsigned int nr_pages;
static unsigned int order;

static unsigned int
my_get_order(unsigned int value)
{
    unsigned int shifts = 0;

    if (!value)
        return 0;

```

```

    if (!(value & (value - 1)))
        value--;

    while (value > 0) {
        value >>= 1;
        shifts++;
    }

    return shifts;
}

static int
thread_fn(void * data)
{
    int i, j, k;
    datatype * this_struct = NULL;

    printk(KERN_INFO "Hello from thread %s. nr_structs=%u\n",
        ↪ current->comm, nr_structs);

    nr_structs_per_page = PAGE_SIZE / sizeof(datatype);
    printk(KERN_INFO "Kernel's page size: %lu\tdatatype's size:
        ↪ %zu\n", PAGE_SIZE, sizeof(datatype));

    nr_pages = nr_structs / nr_structs_per_page;
    if (nr_structs % nr_structs_per_page > 0)
        nr_pages++;
    order = my_get_order(nr_pages);
    printk(KERN_INFO "Number of structs per page: %u\tNumber of
        ↪ pages: %u\tThe order is: %u\n", nr_structs_per_page,
        ↪ nr_pages, order);
    pages = alloc_pages(GFP_KERNEL, order);
    if (pages == NULL) {
        printk(KERN_ERR "alloc_pages() failed!\n");
        return PTR_ERR(pages);
    }

    for (i = 0; i < nr_pages; i++) {
        dt = (datatype *)__va(PFN_PHYS(page_to_pfn(pages + i)));
        for (j = 0; j < nr_structs_per_page; j++) {
            this_struct = dt + j;
            for (k = 0; k < ARR_SIZE; k++) {
                this_struct->array[k] = i *
                    ↪ nr_structs_per_page * ARR_SIZE + j *
                    ↪ ARR_SIZE + k;
                if (j == 0 && k == 0) {

```

```

                                printk(KERN_INFO "The first
                                ↪ element in the %dth page is
                                ↪ %u\n", i,
                                ↪ this_struct->array[k]);
                                }
                                }
                                }

    while (!kthread_should_stop()) {
        schedule();
    }

    return 0;
}

static int
kernel_memory_init(void)
{
    printk(KERN_INFO "Loaded kernel_memory module\n");

    kthread = kthread_create(thread_fn, NULL, "k_memory");
    if (IS_ERR(kthread)) {
        printk(KERN_ERR "Failed to create kernel thread\n");
        return PTR_ERR(kthread);
    }

    wake_up_process(kthread);

    return 0;
}

static void
kernel_memory_exit(void)
{
    kthread_stop(kthread);
    printk(KERN_INFO "Unloaded kernel_memory module\n");
}

module_init(kernel_memory_init);
module_exit(kernel_memory_exit);

MODULE_LICENSE ("GPL");

```

5. After invoking `kthread_stop()` to break the kernel thread out of its `schedule()` loop, perform the same address translation steps as in the thread function. In the body of the inner k loop, simply ensure that each value matches its expected value. Now after the loop, invoke `--free-pages(struct page`

`*, unsigned int order);` to free the pages allocated by the kernel thread. If all values match their expected values, print out a success message. Run your program with values of 1000, 10000. and 50000, and show the output in dmesg.

```

pi@xingjian: ~/Documents/kernel_modules
File Edit Tabs Help
pi@xingjian:~/Documents/kernel_modules $ sftp x.xingjian@shell.cec.wustl.edu
x.xingjian@shell.cec.wustl.edu's password:
Connected to x.xingjian@shell.cec.wustl.edu.
sftp> cd /project/scratch01/compile/x.xingjian/modules
sftp> get kernel_memory.ko
Fetching /project/scratch01/compile/x.xingjian/modules/kernel_memory.ko to kernel_memory.ko 100% 7045 552.7KB/s 00:00
sftp> quit
pi@xingjian:~/Documents/kernel_modules $ sudo dmesg --clear
pi@xingjian:~/Documents/kernel_modules $ sudo insmod kernel_memory.ko nr_structs=1000
pi@xingjian:~/Documents/kernel_modules $ sudo rmmod kernel_memory.ko
pi@xingjian:~/Documents/kernel_modules $ dmesg
[ 8829.515448] Loaded kernel_memory module
[ 8829.515709] Hello from thread k_memory. nr_structs=1000
[ 8829.515721] Kernel's page size: 4096 datatype's size: 32
[ 8829.515732] Number of structs per page: 128 Number of pages: 8 The order is: 3
[ 8835.008332] Success!
[ 8835.008405] Unloaded kernel_memory module
pi@xingjian:~/Documents/kernel_modules $ sudo dmesg --clear
pi@xingjian:~/Documents/kernel_modules $ sudo insmod kernel_memory.ko nr_structs=10000
pi@xingjian:~/Documents/kernel_modules $ sudo rmmod kernel_memory.ko
pi@xingjian:~/Documents/kernel_modules $ dmesg
[ 8864.386238] Loaded kernel_memory module
[ 8864.386251] Hello from thread k_memory. nr_structs=10000
[ 8864.386264] Kernel's page size: 4096 datatype's size: 32
[ 8864.386264] Number of structs per page: 128 Number of pages: 79 The order is: 7
[ 8870.109767] Success!
[ 8870.109840] Unloaded kernel_memory module
pi@xingjian:~/Documents/kernel_modules $ sudo dmesg --clear
pi@xingjian:~/Documents/kernel_modules $ sudo insmod kernel_memory.ko nr_structs=50000
pi@xingjian:~/Documents/kernel_modules $ sudo rmmod kernel_memory.ko
pi@xingjian:~/Documents/kernel_modules $ dmesg
[ 8897.617866] Loaded kernel_memory module
[ 8897.620361] Hello from thread k_memory. nr_structs=50000
[ 8897.620362] Kernel's page size: 4096 datatype's size: 32
[ 8897.620407] Number of structs per page: 128 Number of pages: 391 The order is: 9
[ 8904.478489] Success!
[ 8904.478584] Unloaded kernel_memory module

```

Figure 5: Values are correct

Ans: My thread function is modified as in

```

static int
thread_fn(void * data)
{
    int i, j, k;
    datatype * this_struct = NULL;

    printk("Hello from thread %s. nr_structs=%u\n", current->comm,
        nr_structs);

    nr_structs_per_page = PAGE_SIZE / sizeof(datatype);
    printk("Kernel's page size: %u\tdatatype's size: %zu\n",
        PAGE_SIZE, sizeof(datatype));

    nr_pages = nr_structs / nr_structs_per_page;
    if (nr_structs % nr_structs_per_page > 0)
        nr_pages++;
    order = my_get_order(nr_pages);
    printk("Number of structs per page: %u\tNumber of pages: %u\tThe
        order is: %u\n", nr_structs_per_page, nr_pages, order);
    pages = alloc_pages(GFP_KERNEL, order);
    if (pages == NULL) {
        printk(KERN_ALERT "alloc_pages() failed!\n");
        return -1;
    }
}

```

```
for (i = 0; i < nr_pages; i++) {
    dt = (datatype *)__va(PFN_PHYS(page_to_pfn(pages + i)));
    for (j = 0; j < nr_structs_per_page; j++) {
        this_struct = dt + j;
        for (k = 0; k < ARR_SIZE; k++) {
            this_struct->array[k] = i *
                ↪ nr_structs_per_page * ARR_SIZE + j *
                ↪ ARR_SIZE + k;
            //if (j == 0 && k == 0) {
                //printf(KERN_INFO "The first
                ↪ element in the %dth page is
                ↪ %u\n", i,
                ↪ this_struct->array[k]);
            //}
        }
    }
}

while (!kthread_should_stop()) {
    schedule();
}

for (i = 0; i < nr_pages; i++) {
    dt = (datatype *)__va(PFN_PHYS(page_to_pfn(pages + i)));
    for (j = 0; j < nr_structs_per_page; j++) {
        this_struct = dt + j;
        for (k = 0; k < ARR_SIZE; k++) {
            if (this_struct->array[k] != (i *
                ↪ nr_structs_per_page * ARR_SIZE + j *
                ↪ ARR_SIZE + k)) {
                printk(KERN_ERR "The element
                ↪ array[%d] in the %dth
                ↪ datatype of the %dth page has
                ↪ wrong value!\n", k, j, i);
            }
        }
    }
}

printk(KERN_DEBUG "Success!\n");
__free_pages(pages, order);

return 0;
}
```