Advanced Operating Systems Report 2

SCOTT Harrison Jay

November 3, 2019

Version 5.2.1 of the Linux Kernel was used for this report. No code changes were required to make the examples compile.

1 Low-level memory allocator

1.1 Source Code

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/gfp.h>

#define PRINT_PREF "[LOWLEVEL]: "
#define PAGES_ORDER_REQUESTED 3
#define INTS_IN_PAGE (PAGE_SIZE / sizeof(int))

unsigned long virt_addr;

static int __init my_mod_init(void)
{
    int *int_array;
    int i;

    printk(PRINT_PREF "Entering module.\n");

    virt_addr = __get_free_pages(GFP_KERNEL, PAGES_ORDER_REQUESTED);
    if (!virt_addr)
    {
        printk(PRINT_PREF "Error in allocation\n");
        return -1;
    }

    int_array = (int *)virt_addr;
    for (i = 0; i < INTS_IN_PAGE; i++)
        int_array[i] = i;

    for (i = 0; i < INTS_IN_PAGE; i++)
        printk(PRINT_PREF "array[%d] = %d\n", i, int_array[i]);

    return 0;
}

static void __exit my_mod_exit(void)
{
    free_pages(virt_addr, PAGES_ORDER_REQUESTED);
```

```
39      printk(PRINT_PREF "Exiting module.\n");
40  }
41
42  module_init(my_mod_init);
43  module_exit(my_mod_exit);
```

1.2 Output

Some lines have been omitted

```
<4>[  702.793410] wk2_mod1: loading out-of-tree module taints kernel.
<4>[  702.798554] wk2_mod1: module license 'unspecified' taints kernel.
<4>[  702.802509] Disabling lock debugging due to kernel taint
<4>[  702.816654] [LOWLEVEL]: Entering module.
<4>[  702.819594] [LOWLEVEL]: array[0} = 0
<4>[  702.821737] [LOWLEVEL]: array[1} = 1
<4>[  702.823952] [LOWLEVEL]: array[2} = 2
<4>[  702.825938] [LOWLEVEL]: array[3} = 3
<4>[  702.827956] [LOWLEVEL]: array[4} = 4
<4>[  702.830012] [LOWLEVEL]: array[5} = 5
<4>[  702.832293] [LOWLEVEL]: array[6} = 6
<4>[  702.834574] [LOWLEVEL]: array[7} = 7
<4>[  702.836308] [LOWLEVEL]: array[8} = 8
<4>[  702.837822] [LOWLEVEL]: array[9} = 9
<4>[  702.839427] [LOWLEVEL]: array[10} = 10

...

<4>[  703.519830] [LOWLEVEL]: array[1014} = 1014
<4>[  703.519965] [LOWLEVEL]: array[1015} = 1015
<4>[  703.520058] [LOWLEVEL]: array[1016} = 1016
<4>[  703.520174] [LOWLEVEL]: array[1017} = 1017
<4>[  703.520307] [LOWLEVEL]: array[1018} = 1018
<4>[  703.520508] [LOWLEVEL]: array[1019} = 1019
<4>[  703.520610] [LOWLEVEL]: array[1020} = 1020
<4>[  703.520739] [LOWLEVEL]: array[1021} = 1021
<4>[  703.520918] [LOWLEVEL]: array[1022} = 1022
<4>[  703.521049] [LOWLEVEL]: array[1023} = 1023
```

In summary, this module simply loads an array in low memory with 1024 integers in sequence, and prints them out in order. 1024 is simply the number of integers that fit inside a 4096 byte = 4KB page.

2 `kmalloc()` and `vmalloc()`

2.1 Source Code

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/init.h>
4  #include <linux/slab.h>
5
6  #define PRINT_PREF "[KMALLOC_TEST]: "
7
8  static int __init my_mod_init(void)
9  {
10         unsigned long i;
```

```
11          void *ptr;
12
13          printk(PRINT_PREF "Entering module.\n");
14
15          for (i=1;;i*=2) {
16                  ptr = kmalloc(i, GFP_KERNEL);
17                  if (!ptr) {
18                          printk(PRINT_PREF "could not allocate %lu bytes\n", i);
19                          break;
20                  }
21                  kfree(ptr);
22          }
23
24          return 0;
25  }
26
27  static void __exit my_mod_exit(void)
28  {
29          printk(KERN_INFO "Exiting module.\n");
30  }
31
32  module_init(my_mod_init);
33  module_exit(my_mod_exit);
34
35  MODULE_LICENSE("GPL");
```

2.2 Output

```
<4>[   57.462247] wk2_mod2: loading out-of-tree module taints kernel.
<4>[   57.479160] [KMALLOC_TEST]: Entering module.
<4>[   57.483857] WARNING: CPU: 0 PID: 57 at /root/linux-kernel-module-cheat/
    submodules/linux/mm/page_alloc.c:4640 __alloc_pages_nodemask+0x16e/0xf90
<4>[   57.491921] Modules linked in: wk2_mod2(O+)
<4>[   57.495541] CPU: 0 PID: 57 Comm: insmod Tainted: G           O      5.2.1-
    dirty #1
<4>[   57.500599] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel
    -1.12.1-0-ga5cab58e9a3f-prebuilt.qemu.org 04/01/2014
<4>[   57.505531] RIP: 0010:__alloc_pages_nodemask+0x16e/0xf90
<4>[   57.508441] Code: 48 8b 75 d0 65 48 33 34 25 28 00 00 00 0f 85 49 0d 00 00
    48 8d 65 d8 5b 41 5c 41 5d 41 5e 41 5f 5d c3 81 e7 00 20 00 00 75 02 <0f> 0b
    31 c0 eb d0 48 89 da 44 89 e6 48 c7 c7 40 18 0d 82 e8 fa 55
<4>[   57.516637] RSP: 0018:ffffc900000e3a90 EFLAGS: 00000246
<4>[   57.518991] RAX: 0000000000000000 RBX: 0000000000800000 RCX:
    0000000000000000
<4>[   57.522108] RDX: 0000000000000000 RSI: 000000000000000b RDI:
    0000000000000000
<4>[   57.525109] RBP: ffffc900000e3b88 R08: ffffffff820d0d90 R09:
    0000000000000000
<4>[   57.528525] R10: ffffea0000370000 R11: 0000000000000000 R12: 0000000000000
    cc0
<4>[   57.531706] R13: 000000000000000b R14: 0000000000800000 R15:
    ffffc900000e3e10
<4>[   57.533974] FS:  00007ffff7ff3740(0000) GS:ffff88800f600000(0000) knlGS
    :0000000000000000
<4>[   57.534192] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
<4>[   57.534375] CR2: 00000000006a2008 CR3: 000000000e126000 CR4: 00000000007406
    f0
<4>[   57.534636] PKRU: 55555554
<4>[   57.534776] Call Trace:
<4>[   57.535613]  ? vprintk_emit+0xdb/0x240
```

```
<4>[   57.536011]  ? trace_hardirqs_on+0x38/0xe0
<4>[   57.536137]  ? __free_pages_ok+0x36a/0x570
<4>[   57.536405]  ? __free_pages+0x18/0x30
<4>[   57.536582]  kmalloc_order+0x1c/0x40
<4>[   57.542072]  kmalloc_order_trace+0x24/0xa0
<4>[   57.545372]  ? 0xffffffffc0005000
<4>[   57.547464]  __kmalloc+0x11c/0x180
<4>[   57.549864]  ? __free_pages+0x18/0x30
<4>[   57.552094]  ? 0xffffffffc0005000
<4>[   57.554517]  my_mod_init+0x28/0x1000 [wk2_mod2]
<4>[   57.557294]  do_one_initcall+0x53/0x210
<4>[   57.559182]  ? kmem_cache_alloc_trace+0x32/0x140
<4>[   57.561536]  ? do_init_module+0x27/0x220
<4>[   57.563857]  do_init_module+0x5f/0x220
<4>[   57.566462]  load_module+0x1ff2/0x2530
<4>[   57.568489]  ? kernel_read+0x31/0x50
<4>[   57.570102]  __se_sys_finit_module+0xd0/0x110
<4>[   57.572109]  ? __se_sys_finit_module+0xd0/0x110
<4>[   57.574526]  __x64_sys_finit_module+0x1a/0x20
<4>[   57.576614]  do_syscall_64+0x6d/0x370
<4>[   57.578364]  ? trace_hardirqs_off_thunk+0x1a/0x2e
<4>[   57.580357]  entry_SYSCALL_64_after_hwframe+0x49/0xbe
<4>[   57.583218] RIP: 0033:0x7ffff78f9fb9
<4>[   57.585471] Code: 00 f3 c3 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 40 00 48 89
    f8 48 89 f7 48 89 d6 48 89 ca 4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d
    01 f0 ff ff 73 01 c3 48 8b 0d bf 1e 2c 00 f7 d8 64 89 01 48
<4>[   57.593157] RSP: 002b:00007fffffffeb78 EFLAGS: 00000206 ORIG_RAX:
    0000000000000139
<4>[   57.595904] RAX: ffffffffffffffda RBX: 0000000000000000 RCX: 00007
    ffff78f9fb9
<4>[   57.596182] RDX: 0000000000000000 RSI: 00000000006a2260 RDI:
    0000000000000003
<4>[   57.596432] RBP: 00000000006a2260 R08: 0000000000000001 R09: 00007
    fffffffec00
<4>[   57.596688] R10: 0000000000000003 R11: 0000000000000206 R12:
    0000000000000003
<4>[   57.599907] R13: 00007fffffffef27 R14: 0000000000000000 R15:
    0000000000000000
<4>[   57.600402] ---[ end trace 4458ea2b02adf02e ]---
<4>[   57.603993] [KMALLOC_TEST]: could not allocate 8388608 bytes
```

In summary, this module uses `kmalloc` to request exponentially (powers of 2) larger regions of memory each time. It finally fails once a 8388608 byte = 8MB region is requested. A debug call trace is shown for us.

3 Slab layer

3.1 Source Code

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/init.h>
4  #include <linux/slab.h>
5
6  #define PRINT_PREF "[SLAB_TEST]"
7
8  struct my_struct {
9      int int_param;
10     long long_param;
```

```
};

static int __init my_mod_init(void) {
    int ret = 0;
    struct my_struct *ptr1, *ptr2;
    struct kmem_cache *my_cache;

    printk(PRINT_PREF "Entering module.\n");

    my_cache = kmem_cache_create("pierre-cache", sizeof(struct my_struct), 0, 0,
        NULL);

    if (!my_cache)
        return -1;

    ptr1 = kmem_cache_alloc(my_cache, GFP_KERNEL);

    if(!ptr1){
        ret = -ENOMEM;
        goto destroy_cache;
    }

    ptr2 = kmem_cache_alloc(my_cache, GFP_KERNEL);
    if(!ptr2){
        ret = -ENOMEM;
        goto freeptr1;
    }

    ptr1->int_param = 42;
    ptr1->long_param = 42;
    ptr2->int_param =43;
    ptr2->long_param = 43;

    printk(PRINT_PREF "ptr1 = {%d, %ld} ; ptr2 = {%d, %ld}\n",
    ptr1->int_param, ptr1->long_param, ptr2->int_param, ptr2->long_param);

    kmem_cache_free(my_cache, ptr2);

    freeptr1: kmem_cache_free(my_cache, ptr1);
    destroy_cache: kmem_cache_destroy(my_cache);

    return ret;
}

static void __exit my_mod_exit(void) {
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

3.2 Output

```
<4>[  114.429857] [SLAB_TEST]Entering module.
<4>[  114.434032] [SLAB_TEST]ptr1 = {42, 42} ; ptr2 = {43, 43}
```

In summary, this module uses the slab layer to create a cache, where two instances of a simple

struct are allocated. The data is store and then printed to verify. The cache is then freed and destroyed.

4 High memory allocation

4.1 Source Code

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/gfp.h>
#include <linux/highmem.h>

#define PRINT_PREF "[HIGHMEM]: "
#define INTS_IN_PAGE (PAGE_SIZE/sizeof(int))

static int __init my_mod_init(void)
{
    struct page *my_page;
    void *my_ptr;
    int i, *int_array;

    printk(PRINT_PREF "Entering module.\n");

    my_page = alloc_page(GFP_HIGHUSER);

    if(!my_page)
        return -1;

    my_ptr = kmap(my_page);
    int_array = (int *)my_ptr;

    for(i=0; i<INTS_IN_PAGE; i++) {
        int_array[i] = i;
        printk(PRINT_PREF "array[%d] = %d\n",i ,int_array[i]);
    }

    kunmap(my_page);
    __free_pages(my_page, 0);

    return 0;
}

static void __exit my_mod_exit(void)
{
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
```

4.2 Output

Some lines have been omitted

```
<4>[  139.705359] wk2_mod4: module license 'unspecified' taints kernel.
<4>[  139.709694] Disabling lock debugging due to kernel taint
<4>[  139.717272] [HIGHMEM]: Entering module.
```

```
<4>[  139.720341] [HIGHMEM]: array[0] = 0
<4>[  139.722949] [HIGHMEM]: array[1] = 1
<4>[  139.725556] [HIGHMEM]: array[2] = 2
<4>[  139.728977] [HIGHMEM]: array[3] = 3
<4>[  139.732241] [HIGHMEM]: array[4] = 4
<4>[  139.734286] [HIGHMEM]: array[5] = 5
<4>[  139.736453] [HIGHMEM]: array[6] = 6
<4>[  139.738479] [HIGHMEM]: array[7] = 7
<4>[  139.740676] [HIGHMEM]: array[8] = 8
<4>[  139.742703] [HIGHMEM]: array[9] = 9
<4>[  139.744317] [HIGHMEM]: array[10] = 10

...

<4>[  140.401773] [HIGHMEM]: array[1014] = 1014
<4>[  140.401923] [HIGHMEM]: array[1015] = 1015
<4>[  140.402084] [HIGHMEM]: array[1016] = 1016
<4>[  140.402245] [HIGHMEM]: array[1017] = 1017
<4>[  140.402394] [HIGHMEM]: array[1018] = 1018
<4>[  140.402543] [HIGHMEM]: array[1019] = 1019
<4>[  140.402692] [HIGHMEM]: array[1020] = 1020
<4>[  140.402840] [HIGHMEM]: array[1021] = 1021
<4>[  140.403066] [HIGHMEM]: array[1022] = 1022
<4>[  140.403217] [HIGHMEM]: array[1023] = 1023
```

In summary, this module does the same thing as the first module, except high memory is used. The output is identical other than the print prefix.

5 Per-CPU allocation (static)

5.1 Source Code

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/percpu.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/smp.h>

#define PRINT_PREF "[PERCPU]: "

struct task_struct *thread1, *thread2, *thread3;
DEFINE_PER_CPU(int, my_var);

static int thread_function(void *data)
{
    while (!kthread_should_stop()) {
        int cpu;
        get_cpu_var(my_var)++;
        cpu = smp_processor_id();
        printk("cpu[%d] = %d\n", cpu, get_cpu_var(my_var));
        put_cpu_ptr(my_var);
        msleep(500);
    }
    do_exit(0);
}

```

```c
static int __init my_mod_init(void)
{
    int cpu;

    printk(PRINT_PREF "Entering module.\n");

    for (cpu = 0; cpu < NR_CPUS; cpu++) {
        per_cpu(my_var, cpu) = 0;
    }

    wmb();

    thread1 = kthread_run(thread_function, NULL, "percpu-thread1");
    thread2 = kthread_run(thread_function, NULL, "percpu-thread2");
    thread3 = kthread_run(thread_function, NULL, "percpu-thread3");

    return 0;
}

static void __exit my_mod_exit(void)
{
    kthread_stop(thread1);
    kthread_stop(thread2);
    kthread_stop(thread3);
    printk(KERN_INFO "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

5.2 Output

End of output omitted

```
<4>[   31.561946] wk2_mod5: loading out-of-tree module taints kernel.
<4>[   31.691225] [PERCPU]: Entering module.
<4>[   31.755443] cpu[3] = 1
<4>[   31.762124] cpu[5] = 1
<4>[   31.776228] cpu[6] = 1
<4>[   32.313486] cpu[6] = 2
<4>[   32.314273] cpu[3] = 2
<4>[   32.316744] cpu[5] = 2
<4>[   32.828822] cpu[6] = 3
<4>[   32.829300] cpu[3] = 3
<4>[   32.831268] cpu[5] = 3
<4>[   33.344530] cpu[6] = 4
<4>[   33.345019] cpu[3] = 4
<4>[   33.346981] cpu[5] = 4
<4>[   33.856259] cpu[6] = 5
<4>[   33.856718] cpu[3] = 5
<4>[   33.858583] cpu[5] = 5
<4>[   34.376222] cpu[6] = 6
<4>[   34.376868] cpu[3] = 6
<4>[   34.378353] cpu[5] = 6
<4>[   34.888333] cpu[6] = 7
<4>[   34.888774] cpu[3] = 7
<4>[   34.890475] cpu[5] = 7
<4>[   35.403942] cpu[6] = 8
```

```
<4>[   35.404361] cpu[3] = 8
<4>[   35.406265] cpu[5] = 8
<4>[   36.027699] cpu[6] = 9
<4>[   36.028370] cpu[3] = 9
<4>[   36.030386] cpu[5] = 9
<4>[   36.543484] cpu[6] = 10
<4>[   36.543932] cpu[3] = 10
<4>[   36.546059] cpu[5] = 10

...
```

In summary, this module creates 3 threads which run on different CPU cores (this was done on an emulated 16 core system) which simply count upwards. It is static, which means the per-CPU data structures are created at compile time. We can see the scheduler simply rotates control through each thread fairly in order and so we see 3 copies of each number in ascending order.

6 Per-CPU allocation (dynamic)

6.1 Source Code

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/percpu.h>
#include <linux/kthread.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/smp.h>

#define PRINT_PREF "[PERCPU]: "

struct task_struct *thread1, *thread2, *thread3;
void *my_var2;

static int thread_function(void *data)
{
    while (!kthread_should_stop()) {
        int *local_ptr, cpu;
        local_ptr = get_cpu_ptr(my_var2);
        cpu = smp_processor_id();
        (*local_ptr)++;
        printk("cpu[%d] = %d\n", cpu, *local_ptr);
        put_cpu_ptr(my_var2);
        msleep(500);
    }
    do_exit(0);
}

static int __init my_mod_init(void)
{
    int *local_ptr;
    int cpu;
    printk(PRINT_PREF "Entering module.\n");

    my_var2 = alloc_percpu(int);
    if(!my_var2)
        return -1;

    for (cpu = 0; cpu < NR_CPUS; cpu++) {
```

```
40          local_ptr = per_cpu_ptr(my_var2, cpu);
41          *local_ptr = 0;
42          put_cpu();
43      }
44
45      wmb();
46
47      thread1 = kthread_run(thread_function, NULL, "percpu-thread1");
48      thread2 = kthread_run(thread_function, NULL, "percpu-thread2");
49      thread3 = kthread_run(thread_function, NULL, "percpu-thread3");
50
51      return 0;
52  }
53
54  static void __exit my_mod_exit(void)
55  {
56      kthread_stop(thread1);
57      kthread_stop(thread2);
58      kthread_stop(thread3);
59
60      free_percpu(my_var2);
61
62      printk(KERN_INFO "Exiting module.\n");
63  }
64
65  module_init(my_mod_init);
66  module_exit(my_mod_exit);
67
68  MODULE_LICENSE("GPL");
```

6.2 Output

End of output omitted

```
<4>[   29.183164] wk2_mod6: loading out-of-tree module taints kernel.
<4>[   29.266853] [PERCPU]: Entering module.
<4>[   29.289986] cpu[3] = 1
<4>[   29.298591] cpu[7] = 1
<4>[   29.319484] cpu[8] = 1
<4>[   29.890434] cpu[3] = 2
<4>[   29.891693] cpu[7] = 2
<4>[   29.924039] cpu[8] = 2
<4>[   30.401978] cpu[3] = 3
<4>[   30.404319] cpu[7] = 3
<4>[   30.435906] cpu[8] = 3
<4>[   30.922239] cpu[3] = 4
<4>[   30.923689] cpu[7] = 4
<4>[   30.956148] cpu[8] = 4
<4>[   31.446983] cpu[3] = 5
<4>[   31.448676] cpu[7] = 5
<4>[   31.484294] cpu[8] = 5
<4>[   31.964317] cpu[3] = 6
<4>[   31.994849] cpu[7] = 6
<4>[   31.996809] cpu[8] = 6
<4>[   32.474795] cpu[3] = 7
<4>[   32.507405] cpu[7] = 7
<4>[   32.509810] cpu[8] = 7
<4>[   33.002813] cpu[3] = 8
<4>[   33.035174] cpu[7] = 8
<4>[   33.037358] cpu[8] = 8
```

```
<4>[   33.518703] cpu[3] = 9
<4>[   33.559138] cpu[7] = 9
<4>[   33.560775] cpu[8] = 9
<4>[   34.047759] cpu[3] = 10
<4>[   34.079040] cpu[7] = 10
<4>[   34.080964] cpu[8] = 10

...
```

In summary, this module does the same thing as the previous except dynamically. What this means is the per-CPU data structures are created at runtime. The output is the same other than the fact that the 3 assigned CPU cores are different this time.