Lab 5: Kernel Memory Management

The Linux kernel offers several approaches to memory management, including a variety of memory allocation and deallocation facilities for use in kernel space, which offer different trade-offs between memory utilization and performance. In today's studio you will explore three of them: page-level management, (physically contiguous) object-level management, and slab allocation.

This lab assignment, you will:

1. Allocate and deallocate memory for different numbers of objects, using three different kernel memory management approaches
2. Measure the latency for doing that with each approach

You will need Raspberry Pi (or a Linux machine) to complete the lab.

Make sure that the name of each person who worked on these exercises is listed in the first answer, and make sure you number each of your responses so it is easy to match your responses with each exercise.

This lab has 8 points in total.

Submission:

When finished, ***submit your report in the PDF form to Moodle***. In addition to the report, please also submit the `page_alloc.c`, `kmem_alloc.c` and `cache_alloc.c` files.

Required Exercises:

0. As the answer to the first exercise, list the names of the team members working on this lab.

1. Create a kernel module called `page_alloc` that takes a single unsigned integer parameter called `objnum`, which defaults to 2,000 if no value is provided for it. The module's initialization function should create a single kernel thread that is pinned to one core (which one to use is at your discretion) and that simply outputs a message to the system log (i.e., using printk) that contains the value of `objnum` (which will be the number of objects for which memory will be allocated and then deallocated).

Compile your module, and load it and check if the message appears in the system log. Then unload your module and as the answer to this exercise take the **screenshots of two different outputs** of your module executions where one does not input any number and one set the `objnum` as 100 at the module's initialization.

Hint: Modules can take command line arguments, but not with the argc/argv you might be used to. Therefore, you first need to study how to *Passing Command Line Arguments to a Module*.

2. In a terminal window, run the following command that gives the size of a kernel memory page, in bytes:

```
getconf PAGE_SIZE
```

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 that says how large that struct is (in bytes). You can use `sizeof` and the `struct` type to do this.

As the answer to this exercise, write down (1) how large each kernel memory page is, (2) how large the struct type you declared is, and (3) how many instances of that struct type would fit within a single kernel memory page.

3. Inside the module's thread function, after the initial messages are sent to the system log, use an appropriate clock to obtain the current time, and store that value in a local variable. Immediately after that, use the `alloc_pages` function with flag `GFP_KERNEL` to allocate enough memory pages to contain the number of structs given in the module parameter - i.e., the smallest power of 2 that would be sufficient based on your calculation of the number of objects per page in the previous exercise. Immediately after that, use the `__free_pages` function to deallocate those pages.

Hint: You should use the `PAGE_SIZE` macro to calculate the correct order used in `alloc_pages`. Also note that you cannot use the user space C math library in kernel space. Therefore, you need to study what are the available math functions in Linux kernel API.

Immediately after that, use the same clock as before to obtain the current time, and print a message to the system log with how much time all the memory allocation and deallocation took.

Compile and then load and unload your module on your Raspberry Pi, with values 1,000 and then 2,000 and then 4,000. As the answer to this exercise please record how much time was spent on memory allocation and deallocation for each of those cases.

4. Make a copy of your kernel module code, in a new file named `kmem_alloc.c`. In that file, replace the page-level memory allocation and deallocation approach with a loop that runs as many times as the value given in the module parameter, and in each iteration of the loop (1) uses `kmalloc` to allocate just enough memory for an instance of the struct type and then (2) immediately uses `kfree` to deallocate that memory.

Compile and then load and unload your new module with values 1,000 and then 2,000 and then 4,000. As the answer to this exercise please record how much time was spent on memory allocation and deallocation for each of those cases.

5. Copy your kernel module kmem_alloc.c into a new file named cache_alloc.c, and change the memory allocation and deallocation scheme to use slab allocation. Specifically, your new module's init function should use kmem_cache_create to create a new cache for the module's struct type, and store a pointer to it in a global variable, and your new module's exit function should use kmem_cache_destroy to destroy the cache. Then inside the loop within the thread function of your new module, replace the calls to kmalloc and kfree with appropriate calls to kmem_cache_alloc and kmem_cache_free respectively, instead.

Compile and then load and unload your new module with values 1,000 and then 2,000 and then 4,000. As the answer to this exercise please record how much time was spent on memory allocation and deallocation for each of those cases.

6. Compare and contrast the results you saw for each of the module parameter values, and for each of the memory allocation and deallocation approaches. As the answer to this exercise, please describe briefly any trends you saw as the number of objects to allocate memory for increased, and/or as the memory allocation approach changed.

Documentation:

This webpage can be useful: https://notes.shichao.io/lkd/ch12/