Linux Kernel Programming

Lab 06: Memory Management

AUTHOR
Redha Gouicem and Julien Sopena

Task 1: Process monitoring (but better)

In this task, we want to improve the task monitor implemented in the previous labs. The goal is to maintain a history of the samples in memory.

We wish to maintain this list of statistics in memory by using the `struct task_sample` previously introduced:

```
struct task_sample {
    u64 utime;
    u64 stime;
    struct list_head list;
};
```

In addition to the CPU statistics measured at some point in time, we now have a new `list` member to link the samples together.

Question 1

In your `struct task_monitor`, add a member that represents the head of the list, an integer to count the number of elements in the list as well as a `struct mutex` to protect the list from concurrent accesses. This mutex can be initialized with the macro `mutex_init`.

Question 2

Instead of periodically printing the CPU statistics, your *kthread* should now periodically **store** the statistics. To do so, implement the `int save_sample(void)` function that will be called every second by your *kthread*. This function will allocate a new `struct task_sample`, initialize it with your `get_sample` function, and add it to the sample list in your `struct task_monitor`.

> ⚠ **Warning**
>
> Don't forget to use the mutex when accessing your list! At the end of this lab, this list may be accessed concurrently by the *kthread* as well as by user accesses through the *sysfs*.

Question 3

We now want to change our *sysfs* interface so that it prints the statistics recorded by the *kthread* instead of getting a sample.

Modify the `taskmonitor_show` function that you implemented in the previous lab so that it prints the last samples stored in your `struct task_monitor`. The output should look like this:

```
[root@pnl-tp ~]# cat /sys/kernel/taskmonitor
pid 131 usr 595727581840 sys 463141635510
pid 131 usr 595184668908 sys 462660705879
pid 131 usr 594615820174 sys 462205838086
pid 131 usr 594012109211 sys 461785961240
pid 131 usr 593449309875 sys 461325105600
pid 131 usr 592882507384 sys 460871265522
pid 131 usr 592316693613 sys 460413404313
pid 131 usr 591758860260 sys 459947605897
```

> ⚠ **Warning**
>
> The *sysfs* limits the size of the communication buffer to `PAGE_SIZE` bytes. Be careful to print the last samples without exceeding this buffer size!

Question 4

In your module exit function, free all the memory used by your module.

Question 5

In addition to CPU usage, we also want to record the memory use of the monitored task. This information is found in the `struct mm_struct mm` member of the task.

Modify your `struct task_sample` and your `save_sample()` function to also record the size of the memory mapped for this process (total, stack, data).

Task 2: Memory reclaiming with the shrinker

Your module periodically allocates memory and only frees it when unloaded. After a while, your module's memory footprint will become significant, potentially using up all the memory available on your system, which will lead to a crash. It is therefore necessary to regularly give back some memory to the system.

Question 1

We will use the *shrinker* API to allow the kernel to reclaim memory from our module when needed. Open the `include/linux/shrinker.h` file and analyze the shrinker API. Which functions do you need to implement? What are their purpose?

Question 2

Implement the required functions to use the shrinker API and use the `register_shrinker` and `unregister_shrinker` function to activate/deactivate it for your module.

> 💡 **Tip**
>
> This API is used in multiple locations in the kernel. Check source files that use it to understand how it works and how to use it.

Question 3

Check that your *shrinker* works. You can do this be creating a situation with memory pressure, for example by increasing the sampling frequency of your task monitor. Additionally, you can generate more memory pressure by reading all the files in your system like this:

```
find /usr /var -type f -print0 | xargs -0 cat > /dev/null
```

If it is not enough, you can also reduce the amount of memory available in your VM by changing the QEMU options, *e.g.*, 128M.

Task 3: Efficient memory management with slabs

Question 1

In your module, print the size of your `struct task_sample` using `sizeof` and the `ksize()` function. Why is the size returned by `ksize()` larger than the one from `sizeof`?

Question 2

To optimize the memory efficiency of our task monitor, let's use the *slab layer* and create our own cache of samples. This way, less memory will be wasted by allocating objects larger than necessary from the `kmalloc` slabs.

Create a `struct kmem_cache` that will manage the allocations of your `struct task_sample` objects. You can use the `KMEM_CACHE()` function to initialize your cache, and use the `kmem_cache_alloc()` and `kmem_cache_free` functions instead of `kmalloc()` and `kfree()` to manage your samples.

> **Tip**
>
> This API is used in multiple locations in the kernel. Check source files that use it to understand how it works and how to use it.

Task 4: Preallocated memory with mempools

In case of memory pressure, it might become impossible to allocate new samples. One solution is to use a preallocated memory pool that is guaranteed to be available for new allocations. You can do this with the *mempool* API.

Question 1

Check the *mempool* API in `include/linux/mempool.h`. What types and functions do you need to use and implement? Keep in mind that you are implementing this memory pool on top of your slab cache.

Question 2

Modify your code to use a memory pool on top of your slab cache.

> **Tip**
>
> This API is used in multiple locations in the kernel. Check source files that use it to understand how it works and how to use it.

Task 5: Object management with reference counters

In the current version of our task monitor, a sample cannot be manipulated without acquiring the mutex first. Indeed, if you are holding a pointer to a sample, *e.g., to print it*, you need to hold the mutex in order to avoid some other component to free the sample, *e.g., the shrinker*. While the mutex is **needed** to protect the list itself, we could avoid it for simple sample manipulation. This is possible by using reference counters to manage when a sample can effectively be freed.

Question 1

The kernel provides a reference counter API called *kref*. Check out this API in `include/linux/kref.h`. What functions do you need to implement to use reference counters on your samples?

Question 2

Modify your code to manage your samples with reference counters. With this change, you will not need to hold the mutex to manipulate individual samples anymore.

To exercise your modification, change your implementation of the `save_sample()` function so that it returns a reference to the newly allocated sample. Your *kthread* should also print the sample after that. Manage your reference counter properly.

> **Caution**
>
> The mutex is still required if you manipulate the list itself.

Question 3

How many references should you have for a sample just before it is printed by the *kthread*? Why?

Task 6: Exporting monitoring log to user space

Exporting the monitoring log on the console, *i.e.,* with `printk`, is costly performance wise and pollutes the system journal, even more so if you increase the sampling frequency. It is thus preferable to only print the logged information only when necessary.

Previously, you used the *sysfs* interface to export this data to user space. This is fine when displaying a small quantity of data, as the *sysfs* limits communication to `PAGE_SIZE` bytes. However, now that we keep the history of samples in memory, we may need to read more than `PAGE_SIZE` bytes of data from user space.

In this task, we will use the *debugfs* that does not exhibit this size limitation.

Question 1

Check out the *debugfs* documentation in `Documentation/filesystems/debugfs.rst`. What is the difference between *debugfs* and other pseudo file systems such as *procfs* and *sysfs*? What functions do you need to use to create and destroy a *debugfs* files and directories?

Question 2

We want to export the full history of samples in `/sys/kernel/debug/taskmonitor`. We need to define a `struct file_operations` with the proper handlers for our file. Since our data is a list of sequential samples, we can use the *seq_file* API provided by the kernel, since it is particularly suited for this type of data.

Analyze the documentation of this API in `Documentation/filesystems/seq_file.rst`. Which structure do you need to implement to use this API? Which functions in the `struct file_operations` do you need to implement?

Question 3

Modify your task monitor module to export the history of samples to user space through the *seq_file* API and the *debugfs* in `/sys/kernel/debug/taskmonitor`.

> **Tip**
>
> As usual, you can find examples of usage of *seq_files* in the kernel source. For example, the memory leak detector *kmemleak* uses a *seq_file* to display information to user space (`mm/kmemleak.c`). This can be a good starting point for you. But be careful, *kmemleak* is a complex module! Don't get lost in the complexity and focus **only** on the *seq_file* implementation!

Task 7: Monitoring multiple processes

Your module is currently quite limited, as it can only record the statistics of a single process at a time. We now want to be able to monitor multiple processes at the same time.

To do so, we will make the *debugfs* file writable, so that if you write a PID to that file, it will be added to a list of monitored PIDs. For example, to start monitoring the process with PID 312:

```
        echo 312 > /sys/kernel/debug/taskmonitor
```

Conversely, to disable the monitoring of this process:

```
        echo -312 > /sys/kernel/debug/taskmonitor
```

Question 1

Modify your `struct task_monitor` to be able to chain multiple instances. Instead of having a single global `struct task_monitor`, you will now have a global list of such structures `struct list_head tasks`.

Question 2

Modify your `monitor_fn()` function so that it records the statistics of all the processes monitored in your new list.

Question 3

Add a `write()` function to your *debugfs* `struct file_operations` to handle the addition/deletion of a monitored process, following the API shown at the beginning of this task.

> **Tip**
>
> You can get some inspiration from the `write()` function of the *kmemleak* module.