Homework 3 - Reducing Page Faults and Cooperative Scheduler.

Due: December 7th (Part A), December 20th (Part B, C), 2023, 11:55pm

GitHub page: https://github.com/SudarsunKannan/CS519/blob/master/homework3 Your goal would be to reduce page fault time and implement simple cooperative scheduling in the Linux OS.

Linux 4.17 kernel source code can be found here

`https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=linux-4.17.y`

Part A - Reducing page fault time for your shared memory application (20 points, soft deadline December 7th) In this part, using your shared memory implementation developed in project 1, you will aim to reduce the cost of page faults. You will extend the system call you added in project 2 to increase the number of pages allocated during each page faults, but for shared memory pages. Therefore, in this part, you will attempt to to reduce the cost of page fault handling cost. Every time a page fault occurs, instead of allocating only one page, you are required to allocate two or more pages after a page fault. For example, one page for the actual faulting address (say, addr X) and the next page (say, addr X + 4096). For this, you will modify the OS virtual memory fault handler. The page faults for shared memory are handled in mm/memory.c source file. You can vary the number of pages allocated during a page fault by passing that as a parameter for your custom system call.

Report Results: Document and report the results of your experiments. Highlight any observed differences in performance when varying the number of pages allocated during each page fault. Discuss for what number of pages, multi-page allocation is not useful.

Reference 1 Reference 2

Part B - Understanding the Linux Scheduler CFS scheduler (15 points, December 20th) In the class, we discussed the basics of the CFS scheduler. In this project, we will aim to understand the code internals of the CFS scheduler (part B), identify ideas to improve the CFS scheduler (part C), and test our proposed scheduling mechanism on shared memory project from Part A. The project will be split into two parts, given its complexity. You will be graded for each part independently. You will have 2 independent deadlines.

In the first part of the project (easier), you will try to understand the code of the CFS scheduler and write a detailed description of:

- How are tasks added to a per-CPU runqueue and which file maintains this information??
- How are tasks scheduled on a CPU and which functions handle this?
- How are tasks de-scheduled/preempted from a CPU?
- How are queues re-balanced?
- What are the data structures used for managing run-queues?
- How the scheduler converts application specified nice values to priority.
- How are the statistics of a task and process-level statistics collected by the scheduler, and how are they exposed to userspace?
- The deliverable for the first part of the project will be a detailed description (i.e., code commentary) of the Linux scheduler code, which includes the functions that are called and a description of what these functions do. You will also discuss cases when the CFS scheduler fails to capture application idleness. HINT: Application idleness includes any CPU core underutilized by an application.

Part C - Changing CFS scheduler to be Co-operative (65 points, December 20th)

1. Add a New System Call: Introduce a new system call that allows application threads to convey their inactivity to the scheduler. This system call should provide a mechanism for a thread to notify the scheduler that it is not actively using CPUs.

2. Handle the System Call in the Scheduler: Modify the scheduler to handle the newly added system call. When the system call is invoked, the scheduler should collect information about the application, specifically using the TGID (Thread Group ID) from the task_struct.

Here a simple incomplete system call. One suggestion is that it could be implemented in kernel/sched/core.c but not necessarily this needs to be in this file.

```
syscall_set_inactive(....)
{
    struct task_struct *task = current;  // Get the current task_struct
    // Mark the task as inactive in scheduler data structures
    task->inactive = 1;  // Assume a flag 'inactive' in the task_struct

    // ... (Additional logic for cooperative scheduling)

    return 0;  // Success
}

//Feel free to add your own signatures to these functions
static inline void check_preempt_inactive(struct task_struct *p, struct rq *rq, int flags)
{
    // Check if the task is inactive, and adjust scheduling accordingly
    if (p->inactive) {
        // Adjust scheduler behavior for inactive tasks
        // ... (Your logic for cooperative scheduling)
    } else {
        // Standard preempt logic for active tasks
        check_preempt_wakeup(p, rq, flags);
    }
}
```

3. Design a Multi-threaded Benchmark: Develop a simple multi-threaded benchmark that engages all available cores by continuously spinning on a while() loop. Within this loop, the application should perform either no operations or dummy operations, such as simple vector addition.

4. Implement Cooperative Scheduling: When the application is deemed inactive, execute the system call to inform the operating system about its reduced priority. This step involves the application signaling the OS to lower its priority, allowing other active applications to be prioritized over it.

5. Ensure NICE Values Remain Unchanged: Emphasize that NICE values for applications should not be altered during this process. The focus is on introducing cooperative scheduling without affecting the NICE parameters set for individual applications.

6. Test with Shared Memory Matrix Application: Utilize the designed multi-threaded benchmark alongside the shared memory matrix application from Project 1. Ensure that the shared memory matrix application performs actual work. Run both applications simultaneously, activating cooperative scheduling for the benchmark with the while loop. Evaluate the performance impact of the shared memory matrix application when it runs standalone.

7. Report Results: Document and report the results of your experiments. Highlight any observed differences in performance when cooperative scheduling is enabled for the multi-threaded benchmark compared to the traditional behavior. If possible, draw comparisons with the yield() system call provided by Linux and discuss any creative insights or unique findings. For any queries or additional support, feel free to reach out via email or through the designated platform, Piazza

Resources

In this class, we will use the QEMU-based virtual machine to test the modified OS. QEMU VMs run either on a bare-metal OS or even inside another virtual machine. More details on QEMU can be obtained [here] (https://www.qemu.org/).

For students new to QEMU or hacking kernel, we have created a set of instructions about how to compile a custom kernel (OS) and how to run the OS using QEMU. Detailed step-by-step instructions can be found here.

`https://github.com/SudarsunKannan/CS519`

Computing Resource

If you need access to a development environment and cannot use your laptop, please send me an email (sudarsun.kannan@cs.rutgers.edu).

Starting Early

This a significant-but-essential homework for understanding the basics of OS virtual memory. Please start working on this homework early. If you have questions, make sure to ask them during office hours.