E0-253: Operating Systems Project 1

Building a Task Scheduler

Spring 2023

Background

The scheduler is one of the most critical components of the modern-day operating system. The task scheduler, sometimes called process scheduler, is the part of the kernel that decides which task to run next. It is responsible for best using system resources to guarantee that multiple tasks are executed simultaneously. This makes it a core component of any multitasking operating system. There have been numerous process schedulers throughout the Linux Kernel's lifetime, from the very first scheduler (v0.01) to the O(1) scheduler (v2.6.0) to the completely fair scheduler (v2.6.23) [1]. This assignment tasks you to build a new scheduler and integrate it with the existing Linux kernel. The scheduler you will be building was proposed by Con Kolivas in 2007, called the Rotating Staircase Deadline Scheduler (RSDL) [1]. Unfortunately for Kolivas, his patch never made it to the Linux kernel, but his ideas have been incorporated into the present-day scheduler.

Assignment Summary

RSDL is an O(1) scheduler, guarantees fairness, bounded latency, no starvation, and achieves good interactivity. RSDL primarily uses two arrays, called active and expired, where each array maintains a list of runqueues (see Figure 1). A runqueue can be viewed as a queue of processes waiting to be scheduled at a particular priority level. The size of the arrays (or equivalently the number of runqueues in each array) is based on the number of priority levels supported by the kernel. In the Linux kernel, nice values range from -19 to 20 and hence the active and expired arrays contain 40 runqueues each.

The scheduler schedules tasks in order from the highest priority runqueue to lowest priority runqueue of the active array. Each task is assigned a fixed quota and once the task has exhausted its quota in a runqueue, it is moved down to the tail of the next lower priority runqueue. In addition to per-task quota, each runqueue is also assigned a maximum quota. If a runqueue has exhausted this maximum quota, its remaining tasks are all moved to the tail of the next lower priority runqueue. If there are no runnable tasks in the runqueue, the scheduler moves on to scheduling tasks from the next runqueue. Thus, high-priority tasks automatically get more CPU time as they run multiple times while rolling down the staircase, while low-priority tasks get less CPU time as they start lower in the stairs.

When a task completes its quotas in all the runqueues, it is sent to the expired array at the position of its original priority. An epoch is said to be complete when all runqueues' quota gets exhausted or there are no runnable processes left in the system. When this happens, the expired and active lists are interchanged, and all processes are reset to their original run queues with full quotas.

In your implementation, you will assign a fixed quota of 5 time units to each task in a runqueue irrespective of the priority level. Further, per-runqueue quota would be 20ms. Note that a time unit is defined as a "scheduler tick". You need to consider several cases while implementing the scheduler:

- Currently scheduled task has completed its quota: In this case, the task is shifted to the next run queue, with a fresh quota of 5 time units.
- Currently scheduled task sleeps or gets blocked and has quota remaining: In this case, the task is shifted to the end of the current run queue, and its quota remains unchanged.
- Currently active runqueues' quota is exhausted: In this case, all processes are shifted, in order, to the tail of the next run queue, and their new quota becomes their remaining quota plus 5 time units.

[1] Versions in brackets denote when these schedulers were integrated into the mainline Linux kernel code

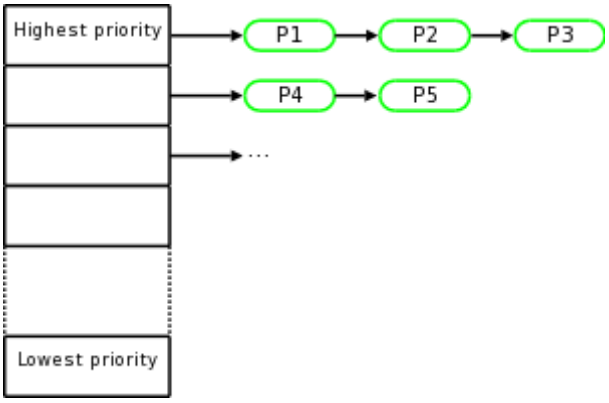

Figure 1: Data structure used in RSDL

Some important points to note:

1. Whenever a new process enters the system in an epoch, it is appended to the currently processing run queue, with a quota of 5 time units.
2. Even if the system has processes which are not runnable (sleeping or blocked) and have quota remaining, and there are some run queues that have quota remaining, the epoch will stand completed, and these tasks too will be reset to their original priority positions, with initial quota values.
3. If (a) the runqueue quota gets over but some processes in that queue have remaining quotas, or (b) there are no runnable processes in that run queue, then these processes are moved in order to the tail of the next lower run queue, but with new quota value equal to remaining quota plus 5 time units.
4. Scheduling always happens in a round robin manner in every run queue.

Setup

You will build the kernel driver in Linux version 6.0.19. Download it as follows:

wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.0.19.tar.xz

Also, make sure that you boot your virtual machine with 4 cores.

We will be posting other project related files (if any) here.

Assignment 0 (Due date: 30/01/2023)

You will be doing this project on a single CPU core.

Kernel Space

- Implement a system call that will isolate CPU 0 such that no processes (not even background or system processes) use that core.

User Space

- Next, you need to schedule two processes with highest and lowest priorities, such that both of them run on CPU 0.

In a sense, this assignment helps you create a test bed for the main objective.

Deliverable

A single kernel patch that contains all your changes and the modified user space file (main.c), which would be provided by us. The kernel patch must include your changes from assignment 0.

Assignment 1 (Due date: 14/02/2023)

In this part of the project, you would be implementing the RSDL scheduler for scheduling processes on the isolated core. You would be implementing the RSDL scheduler with 40 priority levels. Thus, you are required to:

- Implement the code for the RSDL scheduler within the kernel.
- Make sure that all processes running on CPU 0 are scheduled using the RSDL scheduler.

Deliverable

A single kernel patch that contains all your changes and the user space file(s) you used to test out your implementation. The kernel patch must include your changes from assignment 0 and 1.

Additional details regarding assignment 1 (if any) would be posted on Teams.

How to generate a patch using git:

- First, stage all modified files for a commit using git add.
- Second, prepare a commit signed with your email id, using git commit -s -m.
- Third, fetch the commit id using git log (this will be the top-most commit id, say XXX).
- Finally, prepare a patch from your commit using git format-patch.

```
$ git add include/linux/abc.h
$ git add mm/xyz.c
$ git commit -s -m "os assignment-1"
$ git log
$ git format-patch XXX -1
```

References

[1] https://lore.kernel.org/lkml/200703050104.09603.kernel@kolivas.org/T/

[2] Watch this youtube video (Minutes 31 to 46)

[3] https://ck.fandom.com/wiki/RSDL

[4] Understanding the Linux Kernel, 3rd Edition, By Daniel P. Bovet, Marco Cesati. (Chapter 7)

[5] sched(7) — Linux manual page