Article SaaKM

Dubois Swan Gouicem Redha Pires Baptiste Sopena Julien

August 20, 2025

1 Abstract

[lkd]

2 Introduction

CPU scheduling is a fundamental aspect of operating systems and plays a critical part regarding performances and security. With the increasing complexity of hardware and the diversity of workloads servers have to run, writing schedulers is tedious task.

3 Background

The scheduler is a critical part of an operating system (OS) that manages the processes executions. It is required to take into account the hardware it runs on and it might need to adapt to a lot of different workloads. It is also tighly coupled with other parts of the OS like the memory management subsystem (especially regarding NUMA architectures), it therefore require a lot of knowledge to write and maintain a scheduler. When it comes to the Linux kernel, the Earliest Eligible Virtual Deadline First (EEVDF) scheduler is the default general purpose scheduler since v6.6. It is composed of 7000+ LoC alone and depends on a significant amount of C headers an4d kenel APIs. Because of its genericity it implements a lot of heuristics that can be hard to isolate and understand. It also causes a loss in performances in order to adapt to various workloads. That is why there is an effort in the industry and academia to offer kernel developpers a way to write and test schedulers easily*[ghost,schedext,plugsched]*. This need motivated and allowed the successful merge in the kernel of *sched_ext* in the Linux version v6.12.
In this section, we will go through the current options that are available to kernel developer to write and test schedulers in the linux kernel. First, we will see the *sched_class* API of the Linux kernel, then we will take a look at Google proposition, ghOSt[1] that delegate scheduling decisions to user space. Then, we will see *plugsched* a solution proposed by Alibaba, finally we will see *sched_ext* the latest solution proposed by Meta that has been merged into the Linux kernel.*[ajouter un passage sur la liste des sched class]*

The *sched_class* API *[tableau comme CSD avec +/- de chaque solutions ?]*
Linux scheduler subsystem offers an API to write scheduler through its *sched_class* structure. It contains a set of function pointers that has to be implemented by the schedulers and we will be called in the right place by the scheduler core. You still need to modify some datastructures of the kernel like the *task_struct* (it represents a task) and the *rq* (it represents a runqueue) to store data of your scheduler. This already represents a lot of work because you need to understand how the core of the scheduler (6000+ LoC) works, when and how to take locks and all of the other synchronization mecanisms used (RCU, memory syncrhonizations, ...).
Once you have implemented yout scheduler, you need to modify the linking file *vmlinux.lds.h* to add your brand new scheduling class. This is what will allow the kernel to find it and register it with the others. The last step is to compile and boot your kernel, and only then, you will be able to test and debug it.

In short, writing a scheduler using the *sched_class* API is a long and tedious process that requires a wide and deep knowledge of the Linux kernel. *[réecrire + restructurer un peu cette partie avec schedma sched class + avantages et inconvénients]*

ghOSt - Userspace scheduling In order to match the evolution of their needs, Google needed a way to write, test and deploy schedulers easily. That is the main motivation behind ghOSt[1]. It delegates scheduling decisions to userspace and allows to deploy new schedulers or features without having to recompile and deploy kernel images. ghOst architecture is devided into two parts. The first one lies in the kernel, using a *sched_class* to implement the core of ghOSt and to define a rich API exposed to the userspace. The second part is executed in userspace and is composed of *agents* which are scheduling policies. The two parts communicate through *message queues* that allows the kernel to notify the agents of a thread state change, a scheduling tick. The agents now have all the information needed to make scheduling decisions and can commit *transactions* via shared memory. *[presentation resultats + limitations]*

plugsched - Live scheduler update Plugsched takes another approach to give users a way to tweak and modify the scheduler. It leverage the modularization of the Linux kernel by isolating the scheduler code into a *Linux kernel module* that users can then modify, recompile and insert it into a running kernel. To do so, they rewrite the code sections of threads to redirect modified functions. *[on est forcés de de passer tous les threads dans la nouvelle politique]*

sched_ext *sched_ext* is a proposition by Meta that has been merged into the Linux kernel mainline in v6.12. It offers a way to write schedulers as *eBPF* programs. They rely on the *eBPF* verifier to provide safety and security. The eBPF API allows to communicate with userspace program quite easily through *maps*. In this case, it is used to delegate some scheduling work to the userspace, such as in the scheduler *scx_rusty* where they offload all of the load balancing logic to a userspace program written in Rust. This solution comes with limitations as you need to learn the eBPF API (which is still less time consuming than the core of the Linux scheduler) and you can be limited by the eBPF verifier.

As we have seen in this section, the need for a way to write, test and easily deploy new schedulers in the Linux kernel is a pressing issue. Each of the presented solutions have their own advantages and drawbacks

4 Scheduler as a Kernel Module

We now present *Scheduler as a Kernel Module* (SaaKM), a framework that allows kernel developers to write schedulers as Linux kernel modules. Our main goal is to provide a way to write, test and deploy schedulers easily. To do so, we hide the complexity of the core scheduler (synchronization mecanisms, complex API) behind a set of functions corresponding to scheduling events that each policy must implement. We are also capable to have multiple policies loaded at the same time, allowing each applications to chose the scheduler that best fits its needs.

Design We rely on the *sched_class* API presented above to implement a minimal core scheduler (less than 1500 LoC) that will only be used to register policies and call callbacks at the right places. To not interfer with the existing schedulers such as EEVDF or sched_ext, we place our scheduling class right before the idle one.
Each policy must implement a set of functions that map to specific scheduling events (e.g wakeup, scheduling tick, ...) and then register itself to our scheduling class. To not interfer with the existing schedulers such as EEVDF or sched_ext, we place our scheduling class right before the idle one.

There are two types of events. The *thread events* and *core events*. Table 1 shows some examples of these events and their descriptions. We devided the selection of a CPU and the actual enqueuing in the runqueue in two steps. This is needed when a task is newly created task is woken up for the first time. In a first time, it locks the task and calls `new_prepare` which returns the CPU. At this point, our scheduler sees the task for the first time, it needs to initialize and allocated metadata for it. Only then, the runqueue of the CPU is locked and `new_place` is called to actually enqueue the task. Thanks to this architecture, we are able to hide the complexity of the syncrhonization mechanisms to the user.

Function	**Description**
Thread events	
`new_prepare(p)`	Return CPU id where `p` should run
`new_place(task, core)`	Insert `p` into `core` runqueue
Core events	
`schedule(core)`	Called when a task must be elected
`newly_idle(core)`	Called right after `schedule` if there is no task to run

Table 1: Some `saakm_module_routines` functions and their description.

5 Evaluation

References

[1] Jack Tigar Humphries et al. "ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling". In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. Ed. by Robbert van Renesse and Nickolai Zeldovich. ACM, 2021, pp. 588–604. DOI: `10.1145/3477132.3483542`. URL: `https://doi.org/10.1145/3477132.3483542`.