SaaKM : Schedulers as Kernel Module*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Baptiste Pires
LIP6
Sorbonne Université
Paris France
baptiste.pires@lip6.fr

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

Abstract*—*todo:[abstract]

***Index Terms*—kernel, scheduler, module**

I. 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. Linux provide a general purpose scheduler, Earliest Eligible Virtual Deadline First (EEVDF) that is designed to handle those constraints. However, due to its genericity, it is not tailored for specific workloads and causes performances losses. If you want to tweak it to better fit your workloads, you need to have a deep understanding of the core scheduler. Futhermore, the process to update, compile, deploy and test the new version is time consuming. That is why there are efforts to provide kernel developers ways to write, test and deploy schedulers faster.

II. Motivation and Background

The scheduler is a critical part of an Operating System (OS) that manages processes execution. It has to take into account the hardware it runs on but also have to adapt to a lot of different workloads. Currently, the general purpose scheduler of Linux is Earliest Eligible Virtual Deadline First (EEVDF) that has been introduced in v6.6. Being the default scheduler, it has to support a wide variety of workloads (e.g video encoding, web services, ...) but this genericity comes with a cost. Firstly, il is composed of 7000+ LOC and depends heavily on other kernel APIs like the memory management subsystem, the synchronization mechanisms (e.g locks, RCU, ...). This make the development and maintenance quite complex for the kernel developers as you need to put a lot of time and effort to be able to understand the code. Being a core part of the kernel, modifying it requires you to recompile and redeploy your kernel, making the developemnt and debuging a time consuming process. That's why there are been efforts to provide kernel developers ways to write, test, debug and deploy schedulers easily.

Identify applicable funding agency here. If none, delete this.

Ease of Usenote:*[renommer ?]* An ideal solution would not require kernel developers to master a new langage nor a new subsystem of the kernel as it would defeat one the purpose of the solution, that is the accessibility part. By the nature of kernel development, they must already be knowledgeable in C and some of the kernel APIs and subsystems. Linux kernel modules answer perfectly to this need.

Performance The solution must incur the lowest overhead possible. As the scheduler is a critiacal part regarding performances, even a small overhead can cause significant performances degradation. We must keep our solution as ligthweiht as possible, giving the users the choice to add features that can cause this overhead if they want to.

Testing and Debugging

note:*[ces trois points à bouger dans Saakm ?]*

There has been an effort in recent years to provide such a solution. We can devide those in two main categories, the schedulers executing in userspace and the ones executing in kernel space.

A. Userspace schedulers

Userspace schedulers are executed outside of the kernel. They require a library to be linked against that will handle the communication with the kernel.

Google proposed ghOSt [**?**] that is a framework to delegate scheduling decisions to userspace and deploy new scheduling policies without having to recompile and deploy kernel images. It is composed of two parts, the first one lies in the kernel and implement ghOSt core and define a rich API. The second part is executed in userspace and is composed of

agents. Those agents then communicate with the core through message queues that allow the kernel to notify the agents of scheduling events, such as a thread state change. Agents can then commit transactions via shared memory.

B. Kernel schedulers

***sched_class* API** 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 your 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. Schedulers are sorted by priority order as seen in Figure 1.
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. **note:***[réecrire + restructurer un peu cette partie avec schedma sched class + avantages et inconvénients]*

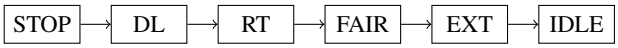

Fig. 1. Linux scheduling classes per priority order

a) 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. Once you have modified the extracted code, you can insert it. They do a data update during the insertion to migrate the data from previous scheduler version to the new one.**note:***[reformuler, on est forcés de de passer tous les threads dans la nouvelle politique]*

b) Enoki: Enoki exposes

c) 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.

There are others way to modify the scheduler in the kernel. kpatch [] provide a way to live patch a running kernel but is restricted to function level.

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 summarized in Table ?.

III. Scheduler as a Kernel Module

We 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 scheduling 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.

A. Design and Implementation

We implement a minimal (less than 1500 LoC) scheduling class that implements all of the scheduling functions requiered by the core scheduler. We place ourselves after the ext policy and before the idle one to not disturb higher priority scheduling classes (e.g RR, fair). We expose an API for the Linux kernel Modules (LKM) to enable them to register as scheduling policies. We store the policies in a linked list sorted by insertion date. Each policy is assigned a id that is unique and used by applications to select the policy they want to use. Figure ? shows the general architecture of SaaKM.

To register a policy, a module must implement a set of functions defined in the structure `saakm_module_routines`. It is composed of handlers that map to scheduling events. Table I shows an excerpt of those events. Events are devided into two categories. Thread events consist of all events related to a thread (e.g thread is waking up, a new thread is created, ...) and core events that are related to core management (e.g core becoming idle, scheduling tick, ...). This distinction allow user to exactly know the path it is. For instance, take the `enqueue_task` function from the `sched_class` structure. X ***todo:****[est-ce que c'est une bonne idée comme example ?]* paths lead to its call. For instance, when a thread is waking up and need to be enqueued back on a runqueue or if a thread was just created. We hide this complexity behind the event and call the handlers at the right places. This way, the user does not need to differentiate paths, it just have to worry about the event currently happening.

Synchronizations In order to hide the complexity of the synchronization mechanisms, we split the selection of a CPU and the actual enqueuing in two steps. The first one

(`new_prepare` and `unblock_prepare`) must returns the id of the CPU on which the thread should run. Then comes the second step (*new_place* and `unblock_place`) where the actual enqueuing takes place. It frees the user from knowing which locks are currently held and which ones it is allowed to take. It is needed for instance when a thread is created. **note:***[completer et réécrire]*

Thread states We overload thread states with ours to make it easier for the user to manage**note:***[?]*. As we have the full control over our states, this allows us to check for wrongful transitions and provide debug feedback to the users. This ease the process of development and debugging.

Runqueues mamagement Each policy must allocate its runqueues ...

Thread migration To handle thread migration, we introduce a new flag, OUSTED. This allows to discriminate between a dequeue coming from a blocking event from a migration. We then simulate a block/unblock sequence to let the policies remove the thread from its runqueue and insert it into the new one.

note:*[parler des etats des threads (et verifications via la machine à état), de comment on passe des données via les structures *_event, peut etre un exemple du workflow (insertion usage etc), parler de la gestion des runqueues, la migration avec le flag OUSTED, gestion du load balancing,]*

B.

TABLE I
EXTRACT OF `SAAKM_MODULE_ROUTINES` FUNCTIONS

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`

IV. EVALUATION

- Definir les métriques
- Definir ce à quoi on se compare
- Présenter l'env et les benchmarks et les motiver
- Présentation et analyse des résultats

V. FUTURE WORKS

Scheduling de policy ?

Rajouter une API pour faciliter la communication avec userspaces vu que c'est en vogue ?

VI. CONCLUSION

Through this work we show that