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.

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 **??**.
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]*

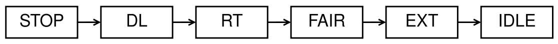

Fig. 1. Linux scheduling classes overview

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. Design

In this section we present the design of SaaKM, our framework for writing schedulers as Linux kernel modules. First we present the goals of our approach, then we detail the design architeture.

A. Design goals

Writing schedulers in Linux is a tedious task so one of our main goals is to provide a simpler way to do that. The need to have a deep and wide knowledge of the Linux kernel can be a barrier also so we want to hide as much complexity as possible. Finally, testing and debugging a scheduler is also a criteria.

Ease of Use 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 The current process of debuging and testing a scheduler class require to recompile and redeploy your kernel image each time you make a modification. This can be quite time consuming and slow down the development process. That is why our solution must not be quick and easy to test without the need reboot your machine.

B. Design overview

SaaKM is an API to write schedulers as Linux kernel modules. It exposes a minimal set of exported functions to the modules that can be used to allocate, manage and destroy scheduling policies and data. Figure **??** shows how SaAKM integrates itself with the current scheduling classes. We position ourselves right before the idle class to not disrupt higher priority classes that other threads runs.

All of the SaaKM disgn relies on a structure composed of handlers that each policy must implement. Each handler maps to a scheduling event. Events are devided into two categories. Thread events correspond to all scheduling events related to a thread (e.g threak is created, waking up, ...). On the other hand, core events are related to the CPU cores (e.g scheduling tick, core becoming idle, ...). This distinction makes it clear for the user to know what it should do, hiding the complexity of the scheduling class API where a single function can be called from multiple paths (i.e enqueue_task). Table II shows an excerpt of those events.

Furthermore, this division allows us to hide the complexity of the syucrhonization mecanisms of the core scheduler. As multicore and NUMA architectures are the quite common now, the scheduler must synchroniza data across cores quite often as two core may access the same data. It protects these data through the usage of locks, RCU and low level synchronization mecanisms like memory barriers. Thanks to our design, the user does not need to worry about these and assume that it has the right locks on the data whenever it is needed.**note:***[réecrire cette partie]*

To increase the flexibility of our solution, we support to have multiple policies loaded at the same time. This allows users to select the scheduling policy that is tailored for their workload.**note:***[completer]*

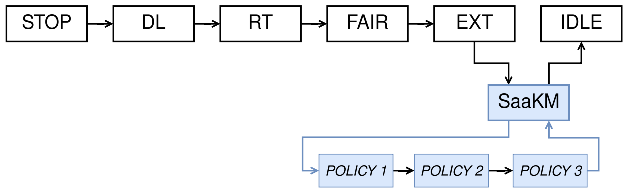

Fig. 2. SaaKM scheduling class with three policies loaded (refaire pour rendre plus visible)

C. Implementation

We implement SaaKM on Linux v6.12, the latest longterm release. In this section we will go through the implementation details.

We implement a minimal (less than 1500 LoC) scheduling class that implements all of the scheduling functions required by the core scheduler. We place ourselves between the ext and idle scheduling classes as shown in Figure 2 where there are three policies loaded. We define a structure `saakm_module_routines` that is composed of all the handler a policy must implement. Table II shows an excerpt of these handlers.

Each policy must allocate and populate a strcture with its own handlers and they will be called by our scheduling class at the right places. Each handlers has at least one argument that is a pointer to its corresponding policy structure. Then depending on the event type of the handler (thread or core), it will have either a pointer to a structure `process_event` or `core_event` that contains the data related to the event.

Runqueues management We provide a minimal runqueue definition that policies uses and includes in their own runqueues as a structure field. This allows to delegate the insertion/deletion and mandatory runqueue metadata fields to be handled by SaaKM. We provide three runqueue types. A FIFO, LIST and RED-BLACK tree. You must provide an ordering function for each type of runqueue, except for the FIFO that will always insert the task at the end of the queue. We provide these three types as they are the most common ones but thanks to its design it can be extended. Policies are required to initialize their runqueues before registering themselves.

Thread states We define our own thread states to make to not interfer with the core scheduler. We define 7 states summurized in Figure I. We have one special state, SAAKM_READY_TICK that allows policy to notify SaaKM that it needs to trigger a reschedule. With this states, we are able to check for wrongful transitions and provide debug feedback to the user at runtime. This ease the debugging and testing process. **note:***[peut etre faire un diagramme d'etat qui montre la machine à etat et que c'est ça qui nous permet de detecter les erreurs ? mieux que tableau ?]*

Thread Migration To handle thread migration that occurs because of affinity change or a `exec` rebalance (a newly created thread is being run on a diffrent CPU than its parent), we introduce a new flag, OUSTED. This flags allows us to know that we are being dequeued becuase of those reason so we can simulate a fast block/unblock sequence to remove the thread from its current runqueue and insert it into the new one.

Load balancing A central and critical part of scheduling is the load balancing. It consist of trying to distribute the load accross CPU cores according to some metrics. To do so, you may need to access data belonging to other cores. Locks are therefore needed to ensure the correctness of the data. We identified two types of load balancing. Periodic load balancing occurs at a fixed time interval and idle balancing. We define a new software interrupt, SCHED_SOFTIRQ_IPANEMA with a handler that calls policies `balancing_select` function. Policies can them chose at which time interval they want to do load balancing. We also raise our interrupt when the fair scheduling kicks a CPU to do the NOHZ balancing.**note:***[est ce qu'on doit garder cette feature avec EXT ?]*
Policies should also be able to do load balacing when a CPU is becoming idle. Whenever that happens, we call the policy handler to let it perform it.

Hardware topology To allow policies to properly make load balancing, we export a per-cpu variable, `topology_levels` that represent the underlying hardware toplogy. To do so, we rely on the already existing scheduling domains that are build at initialization of the scheduler subsystem. Policy can then take educated decisions based on the underlying hardware.

Multi-policy support SaaKM supports to have multiple policies loaded at the same time. When it registers, it is given an unique incremental id that is used by application to select their policy. We then maintain them in a linked list ordered by insertion order. When a new thread must be elected, we go through the list of policies starting from the head of the list. The first policy to return a non NULL thread stops the search and the thread is elected. For load balancing, we also go through the full list to make sure that all policies can perform their load balance.

TABLE I
SAAKM THREAD STATES

State	Meaning
`SAAKM_NOT_QUEUED`	Not in a runqueue
`SAAKM_MIGRATING`	Being migrated
`SAAKM_RUNNING`	Running on a CPU
`SAAKM_READY`	Ready to run and in a runqueue
`SAAKM_READY_TICK`	Became ready from a *tick* event
`SAAKM_BLOCKED`	Blocked and cannot run
`SAAKM_TERMINATED`	Dead

D. SaaKM workflow

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 II 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. **note:***[ancienne version, bouger des phrases ailleurs]*

TABLE II
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`
Policy management events	
`init()`	Called to initialize policy

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 / ajout de prio entre policy ?

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

VI. CONCLUSION

Through this work we show that