eBPF, plugin and Rust

1. Userspace Rust

Userspace Rust

By userspace Rust, Rust code that runs as a wrapper of the eBPF. For this part any eBPF C code can be used as the kernel part, the lottery scheduler is chosen here.

```
build.rs                -> build script
Cargo.toml
/src
-- bpf_intf.rs          -> shared by C headers
-- bpf_skel.rs          -> skeleton bindings
-- main.rs              -> userspace program
-- stats.rs             -> stats collector
-- /bpf
   -- scx_base.bpf.c       -> eBPF C program
   -- scx_base.h           -> eBPF C header
```

Userspace Rust

Rust uses an entirely different build system to build the eBPF.

```rust
scx_cargo::BpfBuilder::new()
    .unwrap()
    .enable_intf("src/bpf/scx_base.h", "bpf_intf.rs")
    .enable_skel("src/bpf/scx_base.bpf.c", "bpf")
    .build()
    .unwrap();
```

Advantages of Rust

Rust here serves simply as a loader of the eBPF program, and a collector of statistics, and it does have some advantages compared to C.

- Better ecosystem:
  - Use existing packages for argument parsing, data structures, etc.
  - e.g. Using `circular buffer`, `serde` and `csv` for more flexible logging.

Advantages of Rust

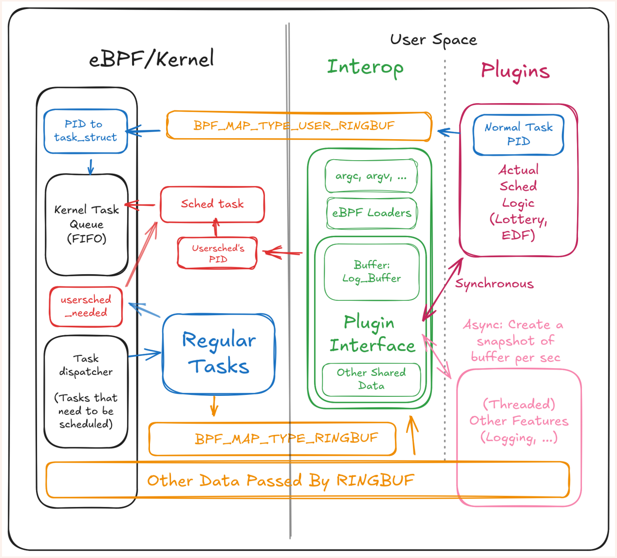
- Better safety and error handling, better atomic variables and strict memory orders all built in, e.g. `Arc{AtomicBool}`, `Ordering::SeqCst`.
- Better build system, without the conflict of header files, especially preprocessor macros.
- `scx_stats` contains endpoint for statistics collection, and can be easily extended. (Here this is unused.)

Disadvantages of Rust

- New language which is hard for beginners (like me).
    - Traits, lifetimes, ownership is all touched upon by scx package.
    - Async Rust is even harder with its complicated concepts on ownership and lifetimes.
- Not necessarily safe, as it uses unsafe code to interface C pointers.

2. Interlude: Plugins in C

Plugin Structure



A plugin structure is used to hide away all the complexity of writing anything in eBPF C with restraints.

Basic Ideas

Essentially, to implement a scheduler policy, we need:

- A `task_struct` to provide the context of the task to be scheduled.

A scheduler policy only need to provide:

- A pid of the process that it decides to run.

Basic Ideas

Essentially, to implement a scheduler policy, we need:

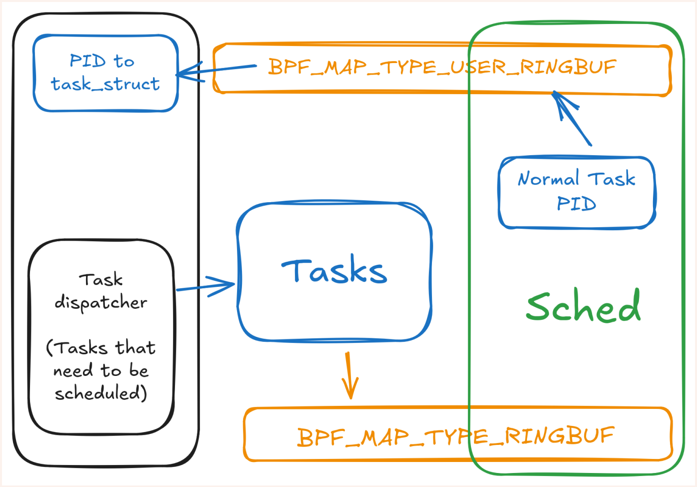
- A `task_struct` to provide the context of the task to be scheduled.

A scheduler policy only need to provide:

- A pid of the process that it decides to run.

This, combined with the zero-copy `BPF_MAP_TYPE_RINGBUF` and `BPF_MAP_TYPE_USER_RINGBUF` maps, allows us to efficiently pass data between kernel and userspace.

Basic Ideas


PID to task_struct
BPF_MAP_TYPE_USER_RINGBUF
Normal Task PID
Tasks
Sched
Task dispatcher (Tasks that need to be scheduled)
BPF_MAP_TYPE_RINGBUF

Issues: Overhead

Although we still need to cross boundaries when we need the userspace to schedule processes, but the overhead of crossing the boundary is much smaller.

Still, to schedule a process we still need to cross the boundary twice and call the userspace functions.

Issues: Overhead

Although we still need to cross boundaries when we need the userspace to schedule processes, but the overhead of crossing the boundary is much smaller.

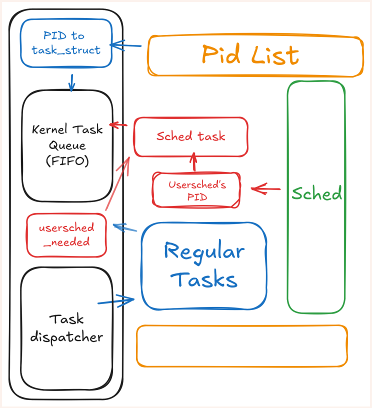
Still, to schedule a process we still need to cross the boundary twice and call the userspace functions.

To minimize this problem this we can batch the scheduling decisions in the userspace by:

- Using a FIFO queue in eBPF to run scheduling decisions.
- Make the userspace supply an array of n pids to be scheduled.
- Under high load, the ratio of scheduler by actual content is roughly $1/n$.

Issues: Deadlock

Another issue is that if the userspace scheduler is not scheduled to run, it cannot make decisions, causing deadlock.



Inspired by scx_userland scheduler, we make these design:

- The userspace will pass its PID to the eBPF scheduler.
- The eBPF scheduler will always push that PID to the scheduling queue rather to userspace.
- Atomic flag is used to notify when there is other task to be scheduled.

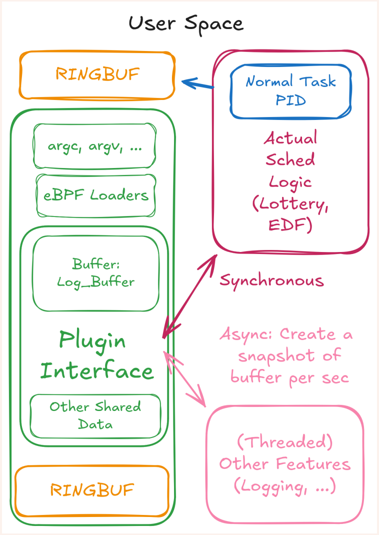
Issue: Logging

In order to observe the behavior of the scheduler, logging is needed. We may also need other features that has other functions.

Run everything in synchronous, and the scheduler will blow up. I/O and other things that we need to run is much more expensive then scheduling.

As we are already this deep in the rabbit hole we decided to use plugin and dynamic libraries to allow runtime loading.

Issues: Logging



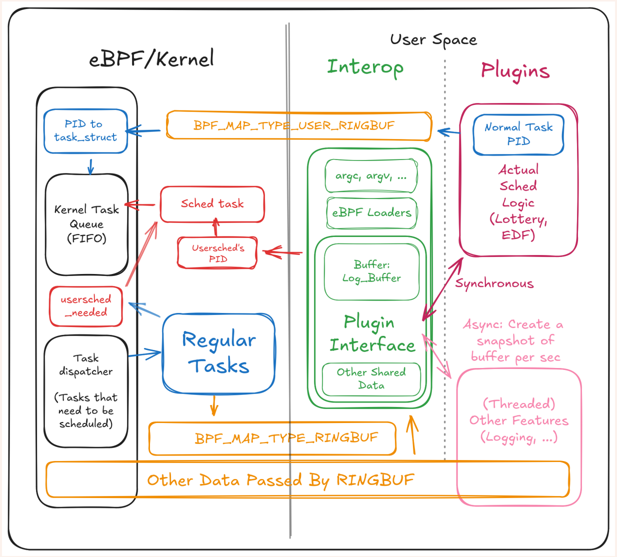
- Only one scheduler will be run, and it is the only plugin to run in sync.
- Other plugins have its own thread spanned by the plugin interface.
- The scheduler can log its decisions into a circular buffer.
- After a time interval, the async plugins receive the head and tail to the circular buffer.

Issues: Logging

While we designed the circular buffer to have 500000 entries to store huge loads, we also introduced a *debug* mode:

- In debug mode, the circular buffer will be alloc'ed 1GB in memory, the exact size of a huge transparent page. This currently holds more than 10 million scheduling decisions.
- The logging plugins will not operate regularly.
- Instead, it will only operate on quit of the scheduler plugin.
- This allows us to have a complete log of the scheduling decisions in extended interval.

Final structure



Thus we have discussed all the structure relevant to the plugin system, and can turn back to this picture.

Lottery

```
compute_0  Sched: 54130
compute_1  Sched: 51487
compute_2  Sched: 48143
compute_3  Sched: 45032
compute_4  Sched: 40560
compute_5  Sched: 36630
compute_6  Sched: 32557
compute_7  Sched: 28394
compute_8  Sched: 24328
compute_9  Sched: 20886
compute_10 Sched: 17503
compute_11 Sched: 14884
compute_12 Sched: 12187
compute_13 Sched: 10212
compute_14 Sched: 8246
compute_15 Sched: 6665
compute_16 Sched: 5374
compute_17 Sched: 4367
compute_18 Sched: 3435
compute_19 Sched: 2815
```

```
compute_20 Sched: 2317
compute_21 Sched: 1837
compute_22 Sched: 1474
compute_23 Sched: 1217
compute_24 Sched: 931
compute_25 Sched: 761
compute_26 Sched: 592
compute_27 Sched: 453
compute_28 Sched: 376
compute_29 Sched: 308
compute_30 Sched: 244
compute_31 Sched: 168
compute_32 Sched: 156
compute_33 Sched: 129
compute_34 Sched: 136
compute_35 Sched: 83
compute_36 Sched: 71
compute_37 Sched: 61
compute_38 Sched: 40
compute_39 Sched: 39
```

Reflection

The final structure of the plugin system is good for:

- Simpler implementation of a scheduler policy.

```c
struct scx_scheduler_plugin_info scx_plugin_info = {
    .info = {.capability = sched,},
    .init = fifo_init,
    .exit = fifo_exit,
    .enqueue = fifo_enqueue,
    .dispatch = fifo_dispatch,
}
```

- Plugin everything, slightly modified my teammates' pressure tester C++ and compiled to a dynamic library, and now the test load is also a plugin.

Reflection

The final structure of the plugin system is flawed for:

- There still lacks a command line capability to specify plugins to load.
- There still lacks a fallback mechanism when no scheduler plugin is loaded. Also bug-prone plugins will stuck the whole computer. *(Pass four null pointers as functions?)*
- The interface(header) exposed to plugins is currently a bit messy and needs refactoring.

That said, these problems exist because of time constraints, and can be solved in future work.

Reflection

Aside from before problems, there are still some questions:

- In systems with multi CPU it needs fine grained policy and interface.
- How to implement both the core eBPF and the plugin interface for things like:
    - CPU affinity
    - Cache locality
    - NUMA nodes
    - ...

Reflection

Aside from before problems, there are still some questions:

- In systems with multi CPU it needs fine grained policy and interface.
- How to implement both the core eBPF and the plugin interface for things like:
  - CPU affinity
  - Cache locality
  - NUMA nodes
  - ...
- The aim is to have a scalable and flexible scheduler that has a very reasonable default but gives the flexibility for user to implement specific policies.

Maybe it is time to turn back to Rust for help from existing crates?

3. Rust Scheduler

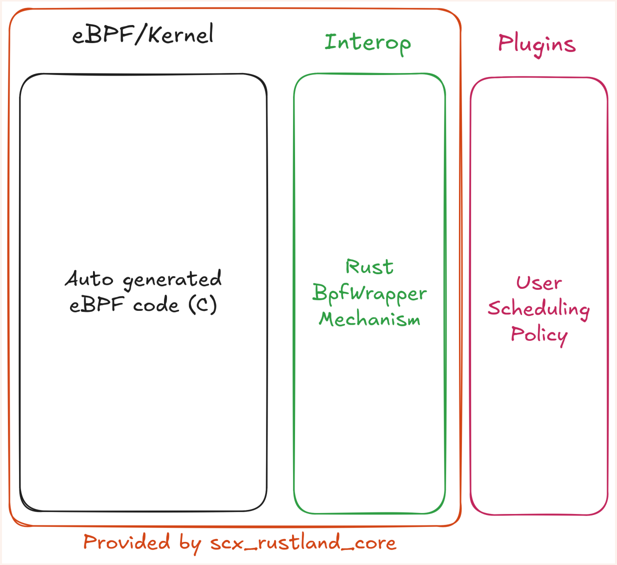
Scheduler in Rust

We can turn our attention to writing the entire scheduler in Rust.

This uses the crate `scx_rustland_core` and `scx_utils` to provide the eBPF bindings and the plugin interface.

The main function of these crates is to auto-generate eBPF code and corresponding Rust bindings to let us focus on writing the scheduling policy only.

A Top-Down Approach


eBPF/Kernel
Interop
Plugins
Auto generated eBPF code (C)
Rust BpfWrapper Mechanism
User Scheduling Policy
Provided by scx_rustland_core

Instead from the bottom up like in the last part, we start from the top.

Scheduler in Rust

These generated codes can be modified, but we use them more as a reasonable default for our time constraints.

We will introduce it in three aspects, compared to my previous C plugin implementation:

- How I used this as a basis to build the scheduler in Rust.
- How the abstraction is designed in Rust.
- The similarities in implementation of eBPF.

Scheduler

We start with a Scheduler object:

```rust
struct Scheduler<'a> {
    bpf: BpfScheduler<'a>,  // BPF connector
    tasks: BTreeSet<Task>,  // tasks ordered by deadline
    slice_ns: u64,          // Default time slice (in ns)
}
```

and a schedule method:

```rust
fn schedule(&mut self) {
    self.enqueue_all_from_bpf();
    self.dispatch_task_to_bpf(8);
    // Notify the dispatcher if there are still
    // pending tasks to be processed.
    self.bpf.notify_complete(self.tasks.len() as u64);
}
```

Some Psuedocode

```
fn enqueue_all_from_bpf(&mut self) {
    let timestamp = Self::now();
    loop { match self.bpf.dequeue_task() {
        Ok(Some(task_in)) =>
            // Preserve the previous valid deadline
            let deadline = if task_in.vtime != 0 {
                task_in.vtime
            } else { Self::niceness_to_time()};
            self.tasks.insert(Task);

        Ok(None) => ...
        Err(err) => ...
        }
    }
}
```

More Psuedocode

```
// dispatch some tasks to ebpf
fn dispatch_task_to_bpf(&mut self, num: u64) {
    let Some(task) = self.tasks.pop_first() else {
        break;
    };
    let mut dispatched_task =
        DispatchedTask::new(&task.qtask);

    dispatched_task.slice_ns = ...
    dispatched_task.vtime = task.deadline;
    dispatched_task.cpu = ...

    // Send the task to the BPF dispatcher.
    if self.bpf.dispatch_task(&dispatched_task).is_err()
        // Dispatching failed: reinsert the task
        self.tasks.insert(task); break;
}
```

EDF

```
compute_0  Sched: 54034
compute_1  Sched: 52112
compute_2  Sched: 51831
compute_3  Sched: 45332
compute_4  Sched: 50409
compute_5  Sched: 47692
compute_6  Sched: 41756
compute_7  Sched: 37856
compute_8  Sched: 32610
compute_9  Sched: 25936
compute_10 Sched: 21448
compute_11 Sched: 16813
compute_12 Sched: 13327
compute_13 Sched: 10580
compute_14 Sched: 8318
compute_15 Sched: 6160
compute_16 Sched: 4944
compute_17 Sched: 4523
compute_18 Sched: 3201
compute_19 Sched: 3052
```

```
compute_20 Sched: 2473
compute_21 Sched: 2018
compute_22 Sched: 1428
compute_23 Sched: 1405
compute_24 Sched: 1129
compute_25 Sched: 933
compute_26 Sched: 766
compute_27 Sched: 642
compute_28 Sched: 527
compute_29 Sched: 402
compute_30 Sched: 396
compute_31 Sched: 273
compute_32 Sched: 281
compute_33 Sched: 186
compute_34 Sched: 154
compute_35 Sched: 171
compute_36 Sched: 123
compute_37 Sched: 105
compute_38 Sched: 105
compute_39 Sched: 69
```

scx_rustland_core's implementation

Now we can turn to the implementation of `scx_rustland_core`'s bpf.rs and see how it abstracts away the eBPF code.
Similarly, it also uses objects to represent the BPF scheduler.

scx_rustland_core's implementation

```rust
pub struct BpfScheduler<'cb> {
    pub skel: BpfSkel<'cb>,
        // low level BPF skeleton
    shutdown: Arc<AtomicBool>,
        // Atomic flag for shutdown
    queued: libbpf_rs::RingBuffer<'cb>,
    dispatched: libbpf_rs::UserRingBuffer,
    struct_ops: Option<libbpf_rs::Link>,
        // Low level BPF methods
}
```

Abstraction Layer: Task Reception

Task Dequeue Abstraction

```rust
pub fn dequeue_task(&mut self) -> Result<Option<QueuedTask>, i32>
```

Task Dispatch Abstraction

```rust
pub fn dispatch_task(&mut self, task: &DispatchedTask)
    -> Result<(), libbpf_rs::Error>
```

Abstraction Layer: Counter Management

Type-Safe Counter Access

```rust
pub fn nr_queued_mut(&mut self) -> &mut u64
pub fn nr_scheduled_mut(&mut self) -> &mut u64
pub fn nr_user_dispatches_mut(&mut self) -> &mut u64
// ... other counter methods
```

- Type-safe counter access without manual offset calculations
- Direct mapping to BPF memory regions
- Eliminates pointer arithmetic and memory safety risks
- Automatic synchronization with kernel space

Abstraction Layer: Completion Notification

Completion Notification Abstraction

```
pub fn notify_complete(&mut self, nr_pending: u64)
```

- Notifies BPF that userspace scheduler completed scheduling cycle
- Updates count of pending tasks
- Prevents unnecessary BPF busy-looping

eBPF implementation details

Now we can turn our attention to the eBPF implementation details, which is also in C.

The eBPF implementation in here, while much more detailed, on a higher perspective is similar to that in C plugin:

- Both uses two ringbuf to pass data between kernel and userspace.
- Both mostly passes structs to userspace and gets information back.

Differences: Wakeup Logic

Multi-Level Prevention Strategy

1. Atomic Flag Notification:

```
volatile u32 usersched_needed;        // atomic flag
set_usersched_needed();               // atomic OR
test_and_clear_usersched_needed();    // atomic AND
```

2. Heartbeat Timer Mechanism:

```
usersched_timer_fn();  // checks every USERSCHED_TIMER_NS (1 sec)
if (scheduler inactive > 1 sec) {
    set_usersched_needed();  // wake scheduler
    kick CPU;
}
```

Differences: Wakeup Logic

3. Dedicated DSQ for Scheduler:

```
#define SCHED_DSQ (MAX_CPUS + 1)  // dedicated queue
// In rustland_enqueue(): place scheduler task in SCHED_DSQ
// In rustland_dispatch(): only dispatch from SCHED_DSQ if needed
```

4. Comprehensive Wakeup Check:

```
bool usersched_has_pending_tasks() {
    return atomic_flag(usersched_needed) ||
           nr_scheduled > 0 ||
           bpf_ringbuf_query(&queued, BPF_RB_AVAIL_DATA);
}
```

Result: Scheduler only wakes when there is actual work; reduces unnecessary context switches.

Personal thoughts on Rust

- It is still very dependent on C eBPF to get the **mechanism** running in kernel space.
- But through this plugin method we can make actual **policy** in any way (language, method, framework) we like.

Personal thoughts on Rust

- The ecosystem/build system of Rust makes it possible to import existing frameworks quickly and easily, which is the core advantages compared to C.
- And it has second best ecosystem and community only to C, an advantage over other languages.

Personal thoughts on Rust

- Along with other modern features it is easy to play around, build and extend without reinventing the wheel for everything like in C.
  - Quickly try new policies
  - Use different data structures
  - Different logging methods
  - ...
- That said Rust is indeed hard to learn.
- To understand eBPF it is better to at least write a working plugin architecture from the ground up like what I did in C plugin, and then turn to higher-level abstractions.

Thank you!

Appendix: Agile Development

We tried our best to follow agile development practices, communication guidelines, etc.
We also used git hooks for commit checking to ensure atomic commit.
We only broke a few times mainly due to copying infrastructure code around.

Appendix: Contribution

The contribution on gitea is not proportional, as I did a lot of work that was not strictly required.
Some code was copied over and over in different parts of the project, especially in the C plugin part.
Rust configuration and copied build scripts also took a some commits.