Paravirt Scheduling

3rd iteration

Vineeth Pillai (Google) <vineeth@bitbyteword.org>
Joel Fernandes (Google) <joel@fernandes.org>

LINUX PLUMBERS CONFERENCE | Vienna, Austria
Sept. 18-20, 2024

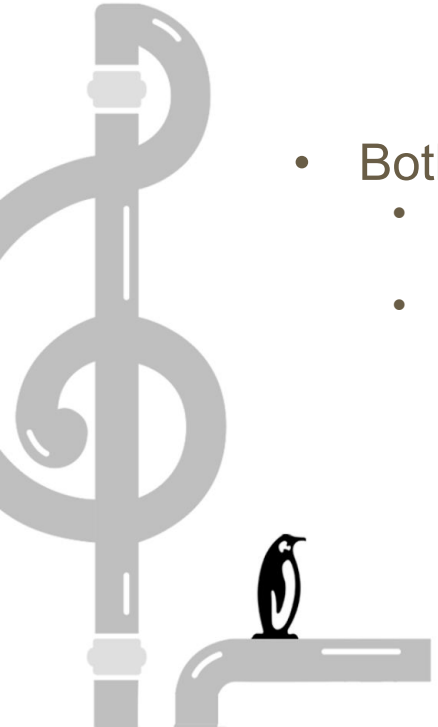
Agenda

- What and Why?
- Some History
- Design and Implementation
- Some Numbers
- What next

Motivation

- Double scheduling: Host schedules vcpu threads and the guest schedules the tasks running inside the guest

- Both schedulers are unaware of the other
  - Hosts schedules vcpu threads without knowing what's being run on the vcpu
  - Guest schedules tasks without knowing where the vcpu is running physically



Motivation (Contd…)

- Mostly, vCPUs are regular CFS tasks in the host and does not get to run in a timely fashion when the host is experiencing load
  - Host scheduler tries to be fair and doesn't know about the priority requirements

- This can cause issues with latencies, power consumption, resource utilization etc.

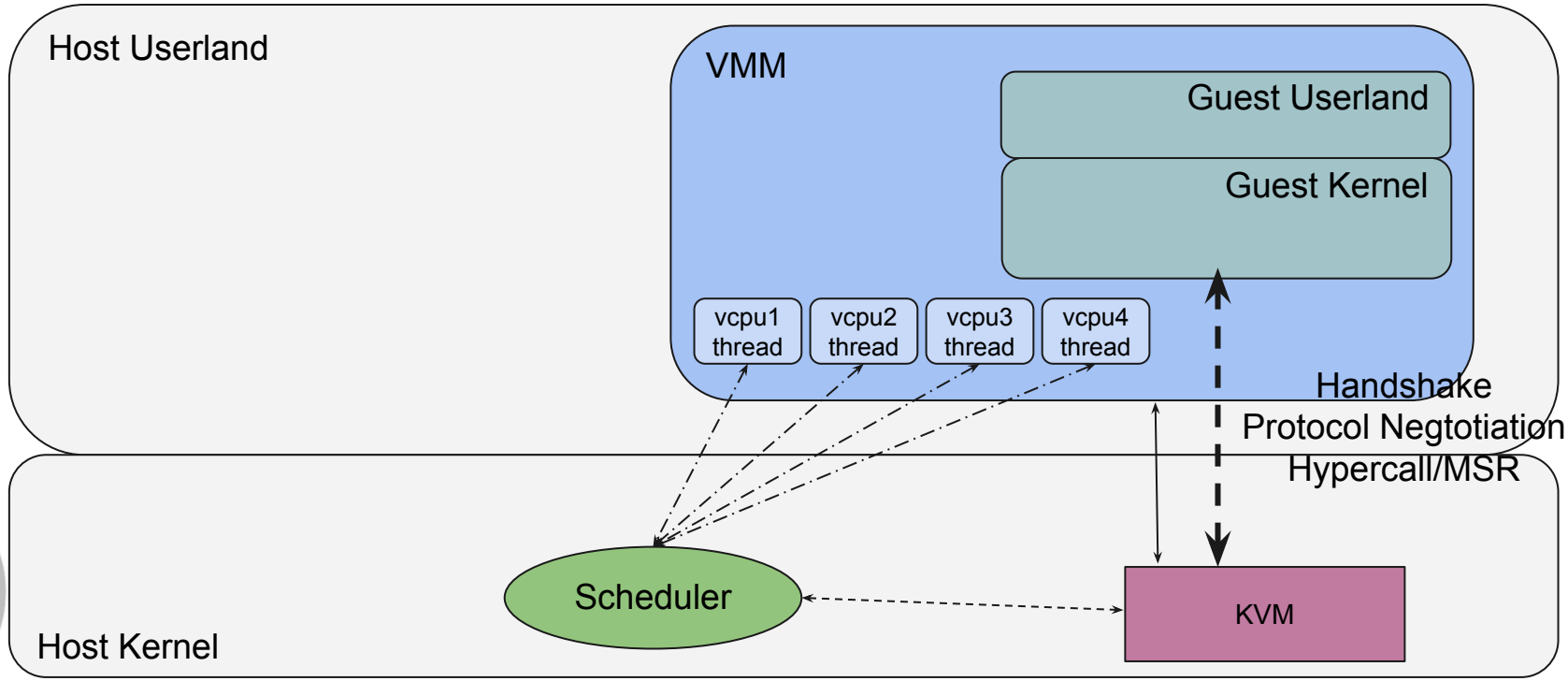
Paravirt Scheduling: Concepts

- Efficient task scheduling decisions based on scheduling information shared between the guest and the host.
- Information shared via shared memory between guest and host.
- General framework in kernel to share the memory and guest/host negotiation
- Scheduling policies could be implemented as a kernel module or bpf program both in the guest and host.

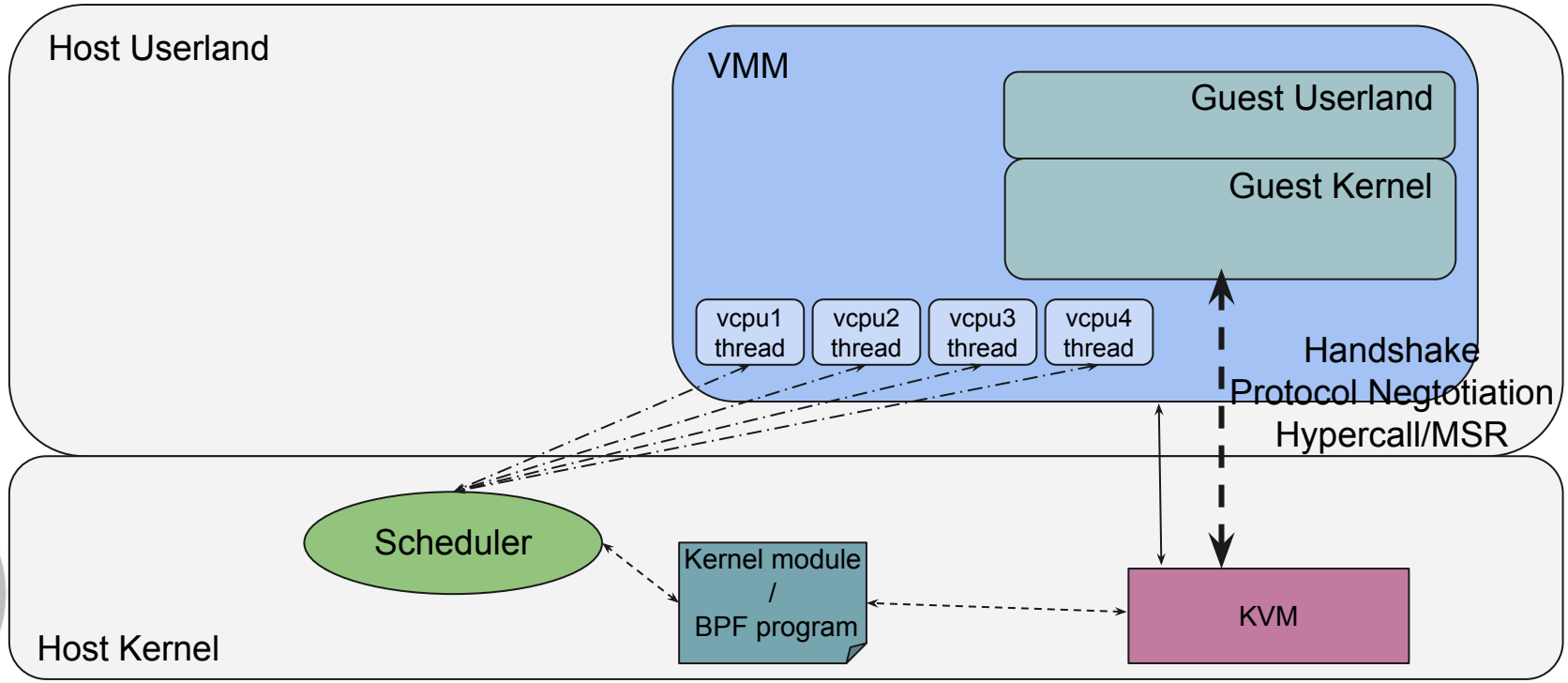
Paravirt Scheduling: History

- V1: https://lkml.org/lkml/2023/12/13/1789
  - Kvm does most of the heavy lifting: handshake, policies, scheduling decision
  - Upstream was against having all these logic in kvm
- V2: https://lwn.net/Articles/968242/
  - Kvm does the handshake with the guest
  - Policy and scheduling decision designed to be implemented separately as a kernel module or a BPF program
  - module/BPF registers to kvm for receiving callbacks on interested events
  - Kvm maintainers is not favorable with the idea of having the handshake also in kernel
- V3
  - Handshake and negotiation logic in the VMM
  - Policies and scheduling decisions in a eBPF program (loaded by the VMM)
    - Could be a kernel module as well

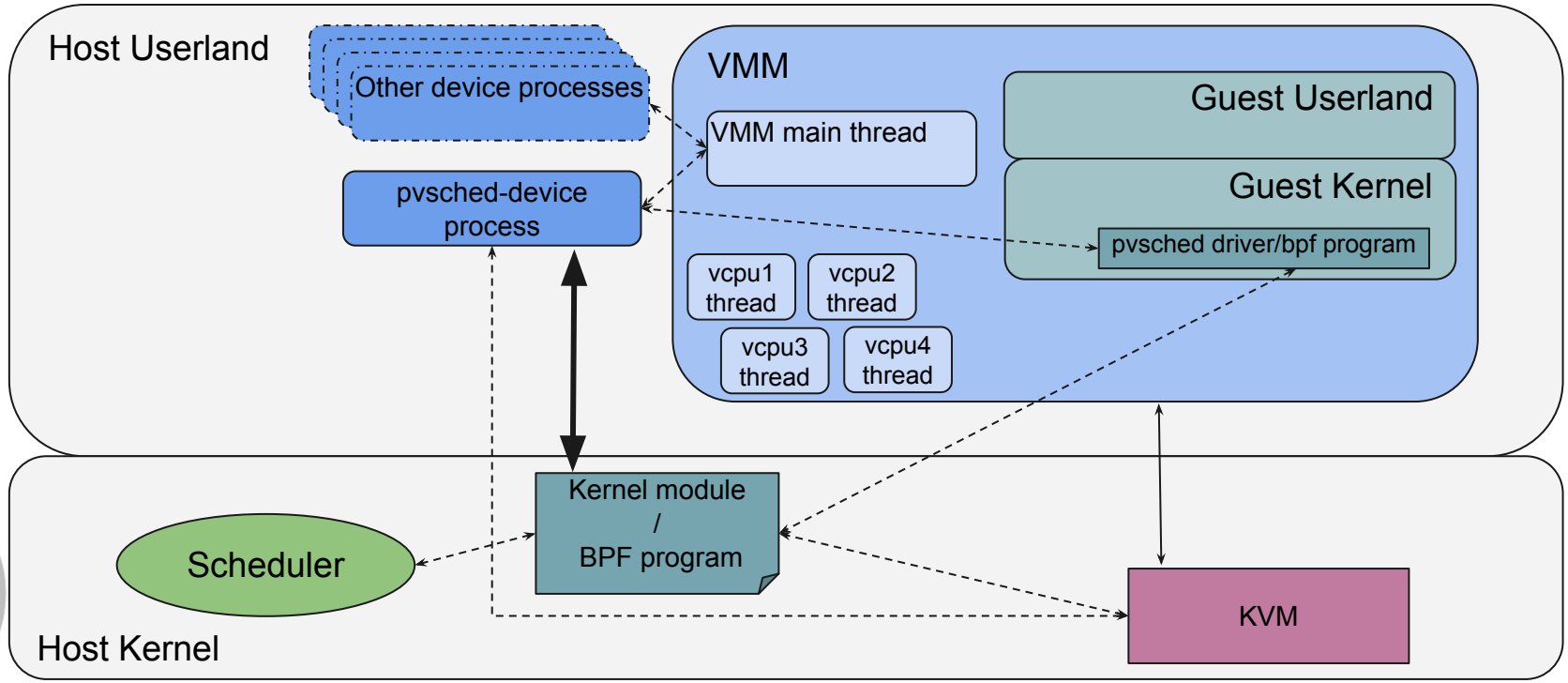
Paravirt Scheduling: V1



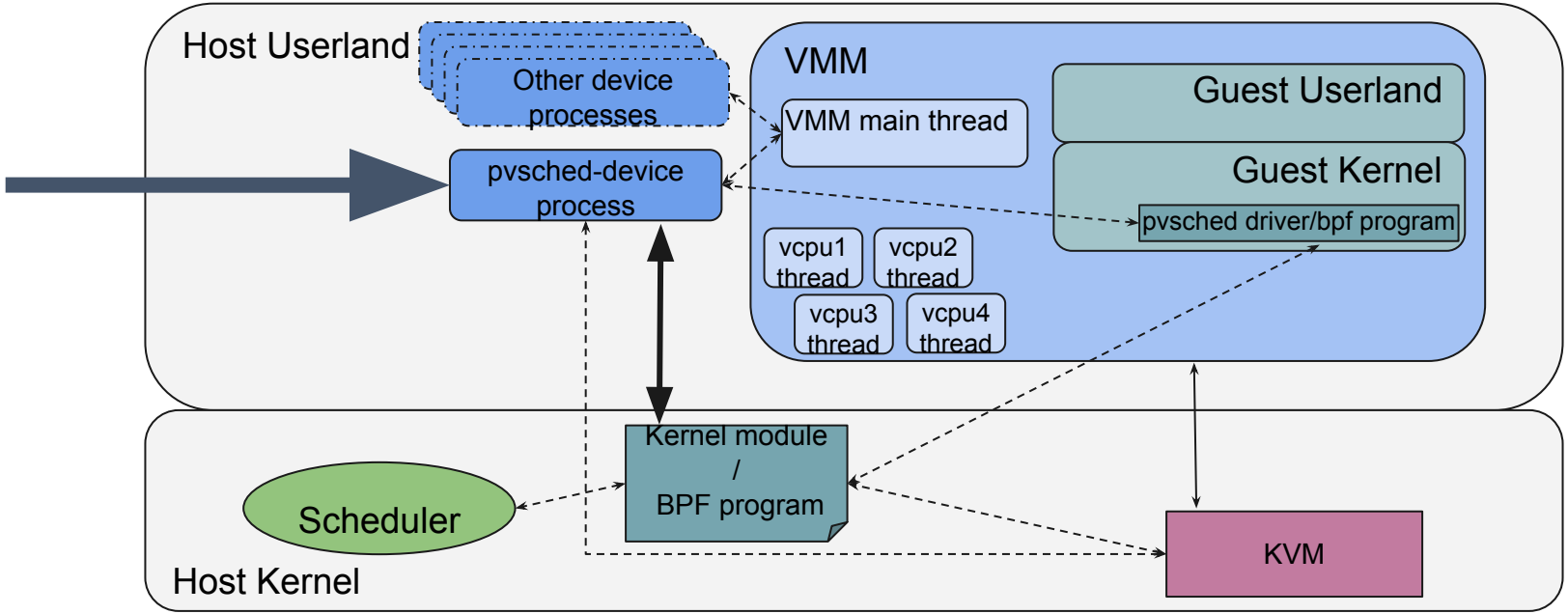
Paravirt Scheduling: v2



Paravirt Scheduling: v3



Paravirt Scheduling: pvsched device



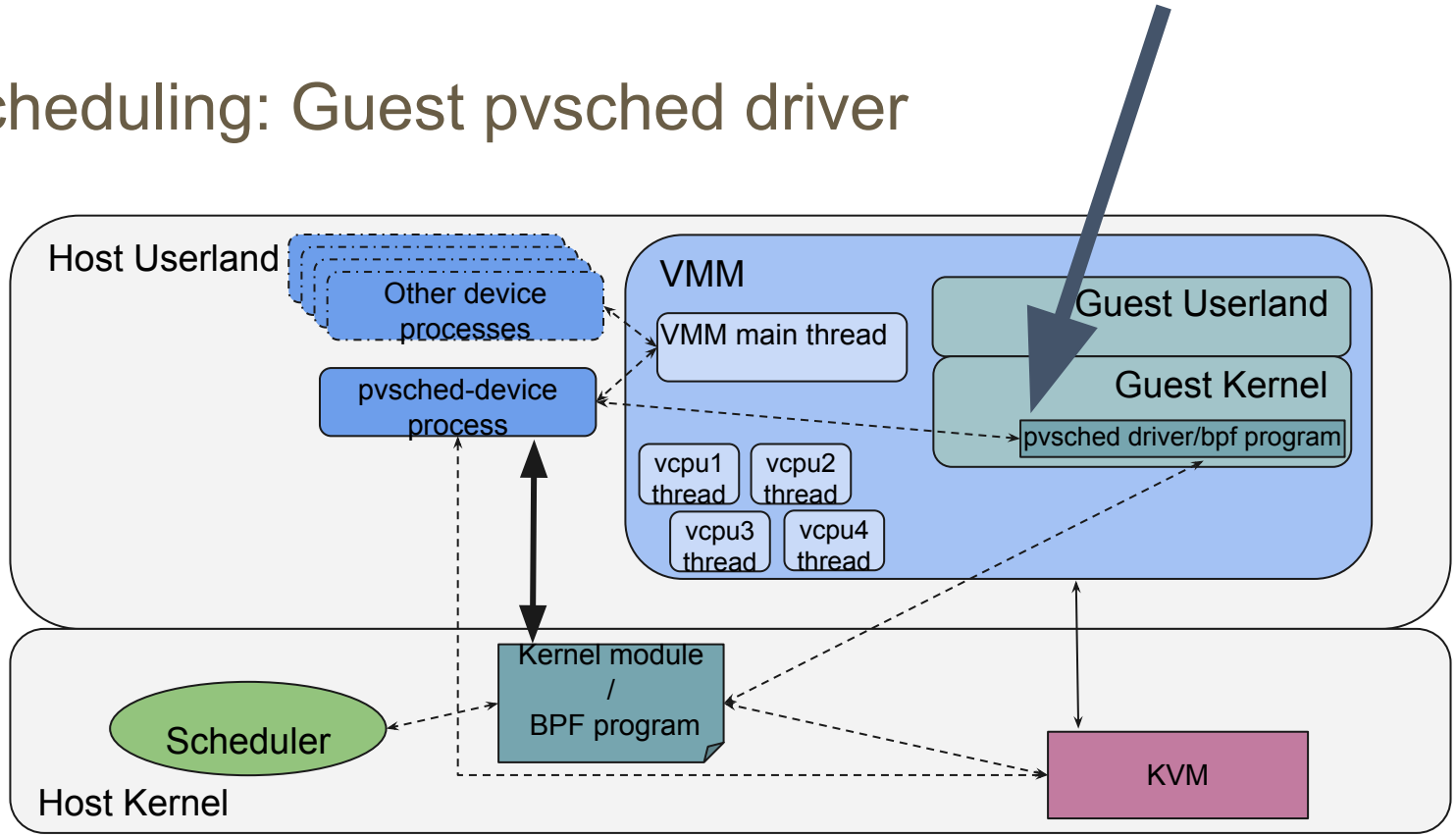
Paravirt Scheduling: pvsched device

- Responsible for handshake with the guest
- Exposes a pci device to the guest
  - A BAR writable by guest. Guest writes the base address of the shared memory region
- Could be extended in future for including feature and policy negotiations

Paravirt Scheduling: pvsched device

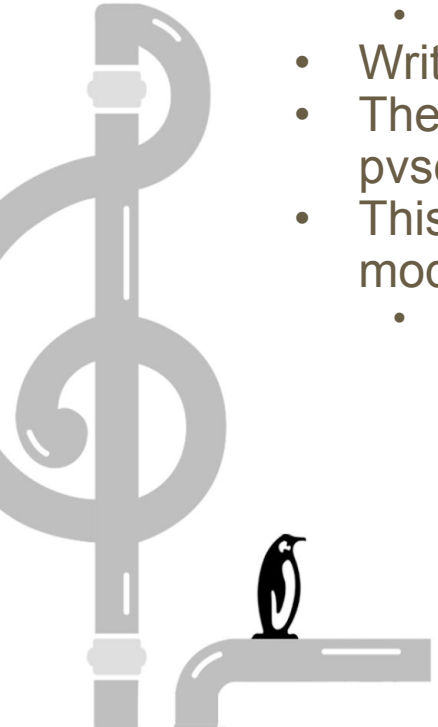
- Prototype implemented in crosvm(VMM written in rust for chromeos)
  - aya-rs: https://aya-rs.dev/
- Pvsched device runs as a separate sandboxed process
- Receives the shared memory GPA from guest and converts to Host Virtual Address in the process address space
- Loads the kernel module or ebpf program that implements the policy and passes details(vcpu id, pid, shared memory address etc)
  - Kernel module: ioctl to pass the information
  - eBPF program: BPF_MAP_TYPE_HASH (indexed by vcpu pid)

Paravirt Scheduling: Guest pvsched driver



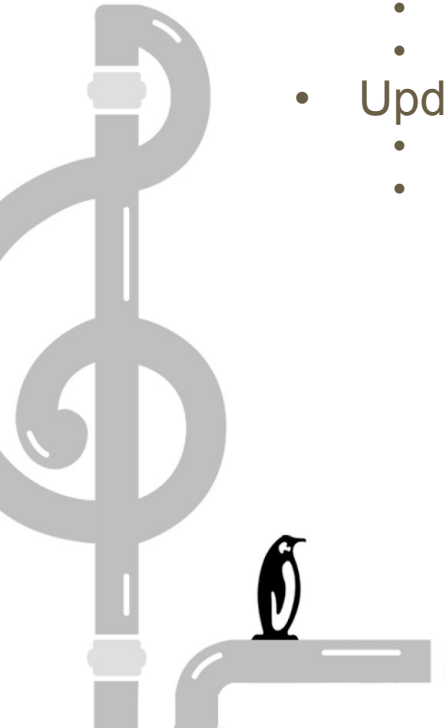
Paravirt Scheduling: Guest pvsched driver

- Kernel mode driver binds to the pvsched device
- Allocates a page for the shared memory
  - Contains an array of structures one per-vcpu
- Writes the shared memory GPA to the BAR exposed by the device
- The prototype has the shared memory updation logic inbuilt in the pvsched driver
- This could be separated out and implemented as another kernel module or a BPF program
  - Cater to specialized use cases.

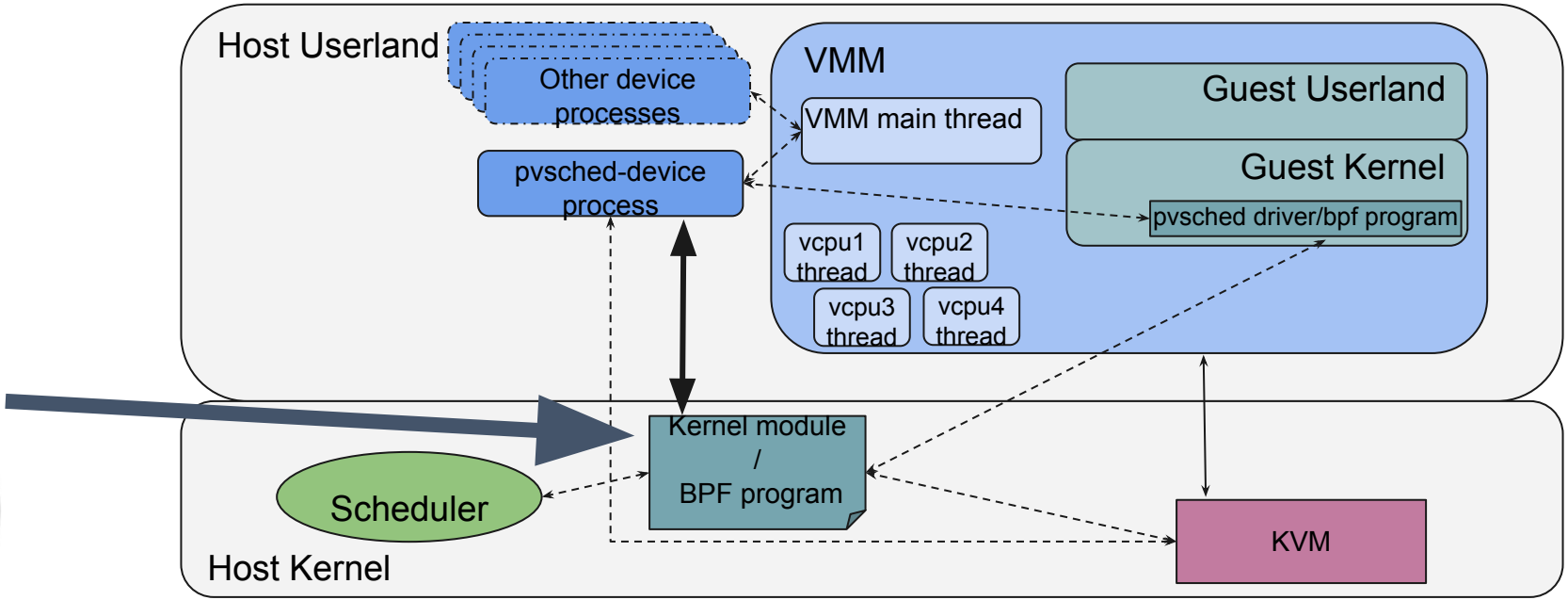


Paravirt Scheduling: Guest pvsched driver

- Registers to scheduler and kernel critical section trace points
  - sched_switch, sched_wake*
  - {nmi/irq/softirq)_{entry/exit}, preempt_{disable/enable}
  - exit_to_user
- Updates the shared memory on trace point callbacks.
  - On kernel critical sections, set a flag in the shared memory
  - Update the shared memory with priority(policy, nice, rt_prio) of the process that is woken up /switched to.



Paravirt Scheduling: scheduling policy

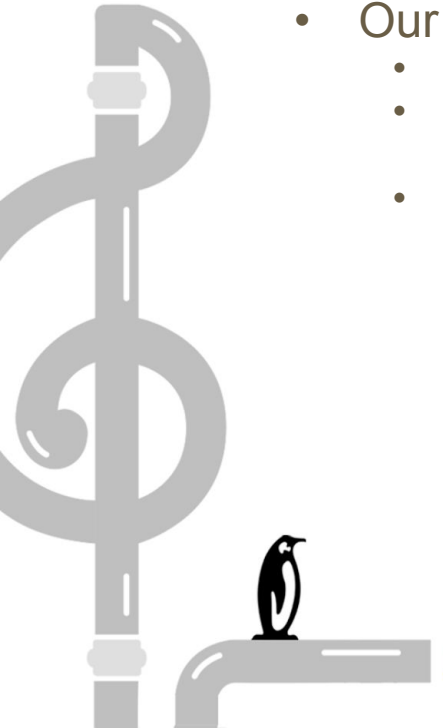


Paravirt Scheduling: scheduling policy

- Could be implemented as a kernel module or eBPF program
- Retrieves the VM and vcpu details from the VMM(nr_vcpus, vcpu_id, vcpu_pid etc)
- Registers for VM event callbacks(VMENTER, VMEXIT, VCPU_HALT, INJ_INTR, …)
- Enforces the scheduling policies on receiving the callbacks

Paravirt Scheduling: scheduling policy

- Policy can be custom implemented based on the use case and requirements.
- Our prototype uses a simple policy to minimize latency
  - Boost the vcpu priority on latency sensitive workloads in the guest
  - 1:1 translation of linux scheduling parameters from guest task to vcpu task in the host.
  - Implements vcpu throttling for misbehaving vcpus



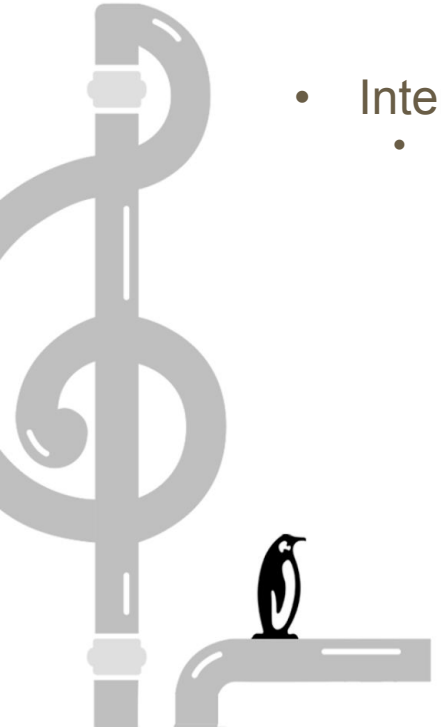
Paravirt Scheduling: Latency mitigation Policy

- VMEXIT
  - Get the requested scheduling parameters from VM
  - Check if the vcpu needs to be throttled based on how long it has been boosted
  - Check if the vcpu needs to be unthrottled based on how long it has been throttled
  - Cap the scheduling priority based on the limits set by admin
  - If VM is not throttled, apply the scheduling priority (sched_setattr)

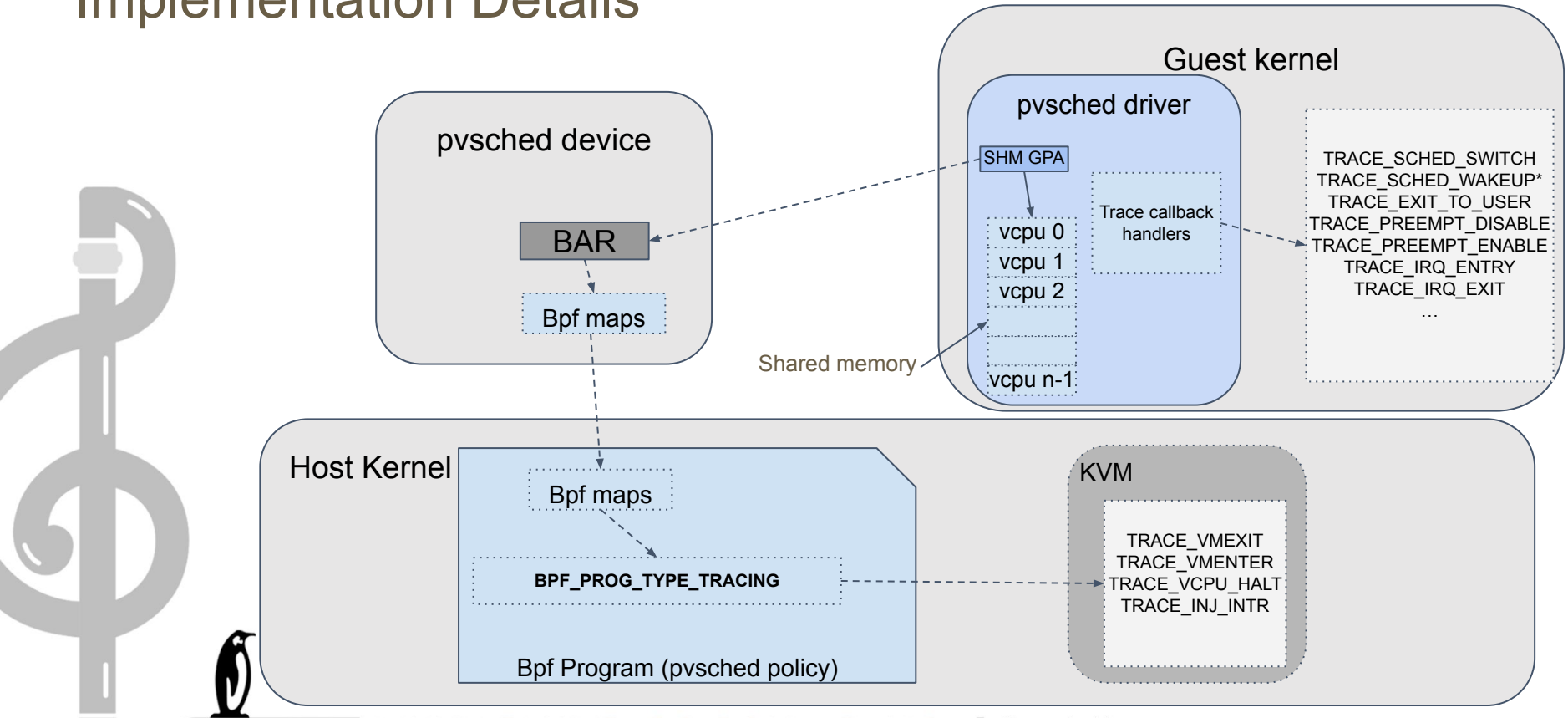
- VMENTER
  - Do the time accounting for boosting, throttling

Paravirt Scheduling: Latency mitigation Policy

- VCPU_HALT
  - Boost the vcpu thread: going to be scheduled out and it will be woken up only on latency sensitive events like interrupts. Guest will request unboost as soon as it is done with priority work loads.
- Interrupt Injection
  - Boost the vcpu thread.



Implementation Details



Implementation: Shared Memory

```
struct pvsched_header {
        __u32 vcpu_id;
        __u32 vcpu_pid;
};

union vcpu_sched {
        struct {
                struct pvsched_header header;
                struct pvsched_guest_area guest_area;
                struct pvsched_host_area host_area;
        };
        __u32 pad[4];
};
```

Implementation: Shared Memory

```
typedef enum {
        PVSCHED_KERNCS_NMI = 0x1,
        PVSCHED_KERNCS_HARDIRQ = 0x2,
        PVSCHED_KERNCS_SOFTIRQ = 0x4,
        PVSCHED_KERNCS_PREEMPT_DISABLED = 0x8
} pvsched_kerncs_t ;
```

```
struct pvsched_guest_area {
        /* In kernel critical section? */
        __u8 kern_cs;
        /* Guest requested sched policy */
        __u8 sched_policy;
        /* Guest requested nice value if CFS */
        __s8 nice;
        /* Guest requested  RT priority */
        __u8 rt_prio;
};
```

Implementation: Shared Memory

```
typedef enum {
        VCPU_STATUS_VMENTERED = 1,
        VCPU_STATUS_VMEXITED = 2,
        VCPU_STATUS_HALTED = 3,
        /*
         * Interrupt could be injected from any of the
         * above states and hence is represented by the
         * last 4 bits.
         */
        VCPU_STATUS_INTR_INJECTED = 0xF0
} pvsched_vcpu_status_t;
```

```
struct pvsched_host_area {
        /*
         * State of the vcpu.
         */
        __u8 vcpu_status;
        /*
         * Sched params set by host before VMENTER
         */
        __u8 sched_policy;
        __u8 nice;
        __u8 rt_prio;
};
```

Implementation: pvsched policy(bpf program)

BPF_MAP_TYPE HASH stores the vcpu specific information

```
struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __uint(max_entries, 100);
        __uint(map_flags, 0);
        __type(key, unsigned int);
        __type(value, struct pvsched_info);
} VCPU_INFO SEC(".maps");
```

Implementation: pvsched policy(bpf program)

```
typedef enum {
        VCPU_ACTION_NONE,
        VCPU_ACTION_BOOSTED,
        VCPU_ACTION_THROTTLED,
} vcpu_action_t;
```

```
struct pvsched_info {
        //struct bpf_timer timer;
        //struct bpf_spin_lock lock;
        vcpu_action_t   action;
        u64 start_ns;
        u64 duration_ns;
};
```

Implementation: pvsched policy(eBPF program)

```
SEC("tp_btf/paravirt_vmentry")
int vmentry_cb(u64 *ctx) {
        return process_pvsched_cbs(ctx, PVSCHED_TP_VCPU_VMENTRY);
}

SEC("tp_btf/paravirt_vmexit")
int vmexit_cb(u64 *ctx) {
        return process_pvsched_cbs(ctx, PVSCHED_TP_VCPU_VMEXIT);
}
SEC("tp_btf/paravirt_vcpu_halt")
int vcpu_halt_cb(u64 *ctx) {
        return process_pvsched_cbs(ctx, PVSCHED_TP_VCPU_HALT);
}
SEC("tp_btf/paravirt_vcpu_inject_intr")
int vcpu_inj_intr_cb(u64 *ctx) {
        return process_pvsched_cbs(ctx, PVSCHED_TP_VCPU_INJ_INTR);
}
```

Implementation: eBPF program & Shared mem

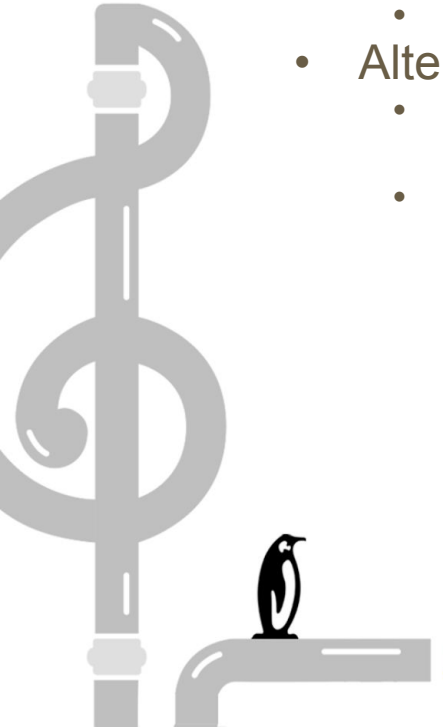
- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user

Implementation: eBPF program & Shared mem

- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user
    - Trace point callback can happen in non-process context

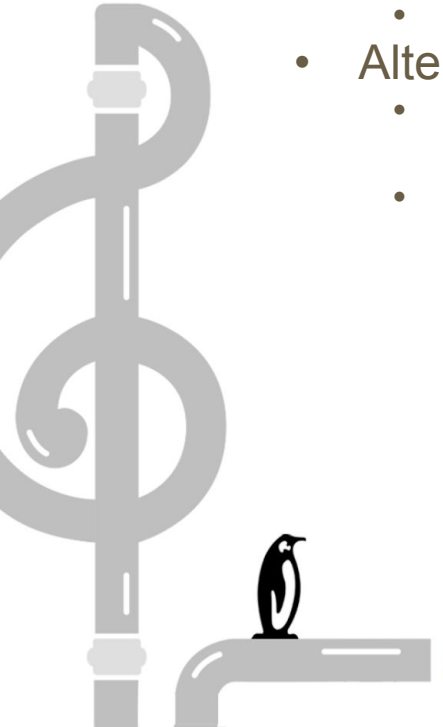
Implementation: eBPF program & Shared mem

- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user
    - Trace point callback can happen in non-process context
  - bpf_probe_read_kernel



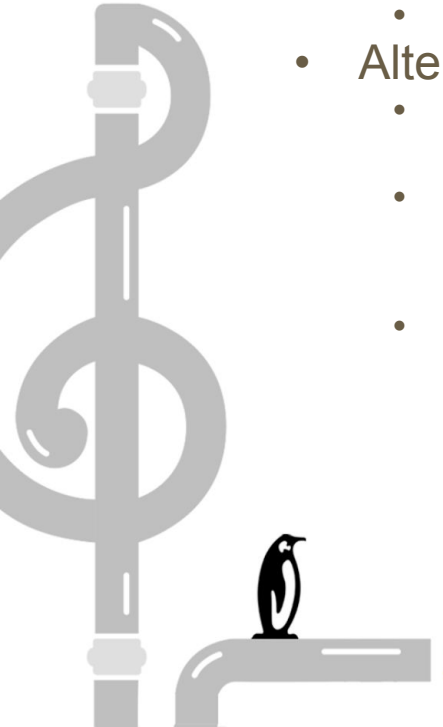
Implementation: eBPF program & Shared mem

- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user
    - Trace point callback can happen in non-process context
  - bpf_probe_read_kernel
    - Need a bpf mechanism to map process Virtual Address space to kernel
    - No kernel address write support.



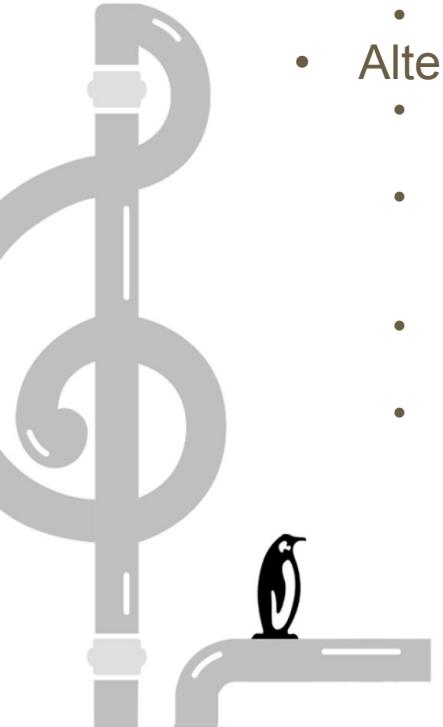
Implementation: eBPF program & Shared mem

- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user
    - Trace point callback can happen in non-process context
  - bpf_probe_read_kernel
    - Need a bpf mechanism to map process Virtual Address space to kernel
    - No kernel address write support
  - kptr
    - Write support?



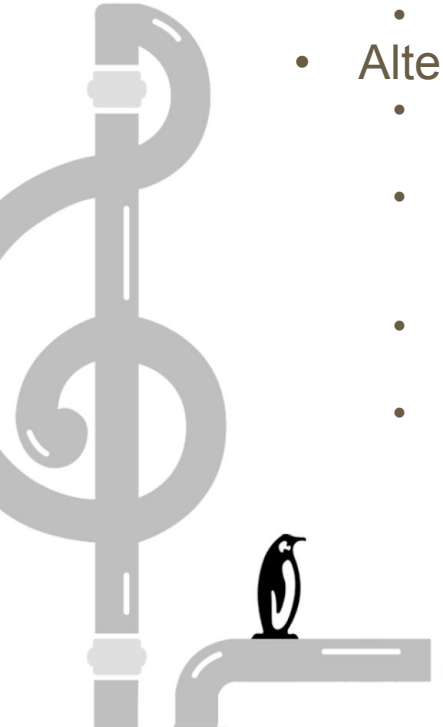
Implementation: eBPF program & Shared mem

- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user
    - Trace point callback can happen in non-process context
  - bpf_probe_read_kernel
    - Need a bpf mechanism to map process Virtual Address space to kernel
    - No kernel address write support
  - kptr
    - Write support?
  - BPF_MAP_TYPE_TASK_STORAGE



Implementation: eBPF program & Shared mem

- Ideally, VMM should be able to share the shared memory Virtual address with the eBPF program
  - Due to ebpf safety constraints, this is not possible easily.
- Alternatives
  - bpf_probe_{read/write}_user
    - Trace point callback can happen in non-process context
  - bpf_probe_read_kernel
    - Need a bpf mechanism to map process Virtual Address space to kernel
    - No kernel address write support
  - kptr
    - Write support?
  - BPF_MAP_TYPE_TASK_STORAGE
    - Share user memory through task local storage: https://lore.kernel.org/bpf/20240816191213.35573-4-thinker.li@gmail.com/T/



Implementation: eBPF Program & Shared mem

- Our Prototype uses a HACK!
  - Modify struct task_struct to include the kernel address and page
  - Use sched_setscheduler to set these fields
  - VMM calls set_scheduler for vcpu tasks
  - BPF kfunc for set/get

```
struct task_struct {
        …
        void            *pvsched_shm_addr;
        struct page      *pvsched_shm_page;
        …
};
```

```
static int __sched_setscheduler(struct task_struct *p,
                                const struct sched_attr *attr,
                                bool user, bool pi)
{
        ...
        ..

        if (attr->sched_flags & SCHED_FLAG_PVSCHED_SHM_ADDR) {
                return sched_set_pvsched_shm_addr(p,
                                attr->sched_pvsched_shm_uaddr);
        }
```

Implementation: eBPF Shared mem kfunc

```
BTF_ID_LIST(bpf_vcpu_sched)
BTF_ID(union, vcpu_sched)

const struct bpf_func_proto bpf_get_pvsched_proto = {
        .func           = bpf_get_pvsched,
        .gpl_only       = false,
        .ret_type       = RET_PTR_TO_BTF_ID_OR_NULL,
        .ret_btf_id     = &bpf_vcpu_sched[0],
        .arg1_type      = ARG_ANYTHING,
};
```

Implementation: eBPF and scheduler

- We need to call into scheduler to modify the scheduling parameters of vcpu tasks.
- Implemented kfunc

```
const struct bpf_func_proto bpf_process_pvsched_params_proto = {
        .func           = bpf_process_pvsched_params,
        .gpl_only       = false,
        .ret_type       = RET_INTEGER,
        .arg1_type      = ARG_ANYTHING,
        .arg2_type      = ARG_ANYTHING,
        .arg3_type      = ARG_ANYTHING,
        .arg4_type      = ARG_ANYTHING,
};
```

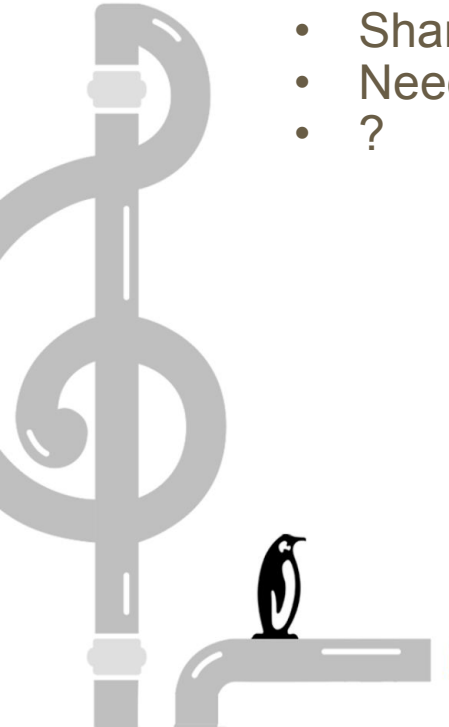
```
BPF_CALL_4(bpf_process_pvsched_params, s32, pid_nr,
                u32, sched_policy, s32, nice, u32, rt_prio)
{
        struct sched_attr _attr = {
                .size = sizeof(_attr),
                .sched_policy = sched_policy,
                .sched_nice = nice,
                .sched_priority = rt_prio,
        };

        return __bpf_process_pvsched(pid_nr, &_attr);
}
```

Implementation: pvsched tunables

- Implemented as BPF maps and VMM can adjust these tunables
  - Scheduling priority for vcpu task when guest is in a kernel critical section or host is injecting interrupt.
  - Maximum boost time allowed for a vcpu task before it is throttled
  - Throttle duration
  - Debug levels

Implementation: eBPF limitations

- Timers and spinlocks not possible for trace points.
- Shared memory access is a bit tricky without firsthand bpf support!
- Needs scheduler kfunc/helpers
- ?



Some performance numbers

- Quick synthetic test with cyclictest on host and guest
  - observe latency impact on guest
  - observe if any regression on host
- Tested on an idle host and busy host
  - Busy host simulated by stress-ng

```
cyclictest --mlockall -q -p 8 --policy=rr
```

```
stress-ng --cpu 2 --iomix 1 --vm 1 --vm-bytes 128M --fork 1
```

- Host System
  - Intel(R) Pentium(R) Silver N6000 @ 1.10GHz
  - 8GB RAM (LPDDR4-2933)
  - Fedora 40 (6.10.10)

Some performance numbers (Idle Host)

Guest: Average latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	60	65	63	64
1000	71	74	72	70

Host: Average latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	29	28	25	30
1000	72	70	71	70

Guest: Max latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	102	76	48	70
1000	160	124	120	120

Host: Max latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	26	34	26	22
1000	90	80	70	85

Legend:
Vanilla: Host kernel: 6.10 guest kernel: 6.10
v1: Host and guest kernels with the v1 patches
v3-kmod: v3 with policy implemented in kernel module
V3-bpf: v3 with Policy implemented in bpf

Some performance numbers (Busy Host)

Guest: Average latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	680	685	680	684
1000	600	600	600	610

Host: Average latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	653	668	641	685
1000	760	750	754	740

Guest: Max latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	9335	7500	7467	7480
1000	14128	12700	12648	12700

Host: Max latency in micro seconds

Interval(us)	vanilla	v1	v3-kmod	v3-bpf
500	8049	8018	8099	8067
1000	12524	12549	12495	12421

Legend:
Vanilla: Host kernel: 6.10 guest kernel: 6.10
v1: Host and guest kernels with the v1 patches
v3-kmod: v3 with policy implemented in kernel module
V3-bpf: v3 with Policy implemented in bpf

Paravirt Scheduling v3: Future

- Upstream collaboration to get the eBPF support like kfunc/helpers for scheduler integration
- Upstream collaboration to get shared memory access support
  - A new map to share data directly between guest and host?
- Spinlock and timer support
  - Use bpf struct_ops instead of trace points?
- Customizability
  - Generalize the shared memory area (union vcpu_sched) to support more use cases
  - Protocols, versioning etc.
- Guest side BPF?
  - sched_ext in guest?

Paravirt Scheduling v3: Future

- Use sched events rather than kvm events
  - For the latency mitigation, we are only concerned about a vcpu being preempted and vcpus getting to vcpu as soon as woken up.
  - So, why not hook to sched_switch and sched_wakeup instead of VMEXIT/VMENTER?
- Still, need to hook to interrupt injection path
  - vcpu may be scheduled out between interrupt injection and VMENTER, if not boosted
  - modify kvm_vcpu_kick() to hint the scheduler about the reason for the kick - interrupt, ipi, …

Paravirt Scheduling v3: Conclusion

- Generic framework for aiding custom policies to be implemented as a kernel module or bpf program
- Shows similar results as v1
- KVM/Hypervisor depency almost removed completely
  - Mostly addition of trace points.
- Minimal changes in kernel
  - Mostly trace points

Danke!