hcHyper

陈岳 清华大学计算机科学与技术系

Overview

hcHyper: https://github.com/cylindrical2002/hcHyper

hcHyper 项目实现了基于 ArceOS 提供的硬件支持下的 x86(Intel) ， ARM ， RISC-V 三大架构的 CPU 和 Memory 的虚拟化支持。CPU 支持部分实现在 hypervisor/crates/hypercraft 中，Memory 部分实现在 hypervisor/crates 下的 guest_page_table 和 guest_page_table_entry 中 。

目前能够支持在含 KVM 的 Intel PC 中运行 Hypervisor 虚拟化 NimbOS 操作系统，同时支持在 QEMU 中运行 ARM 和 RISC-V 两种架构下的 Hypervisor ，分别能够支持 NimbOS 操作系统和 Linux 操作系统。

Predessor

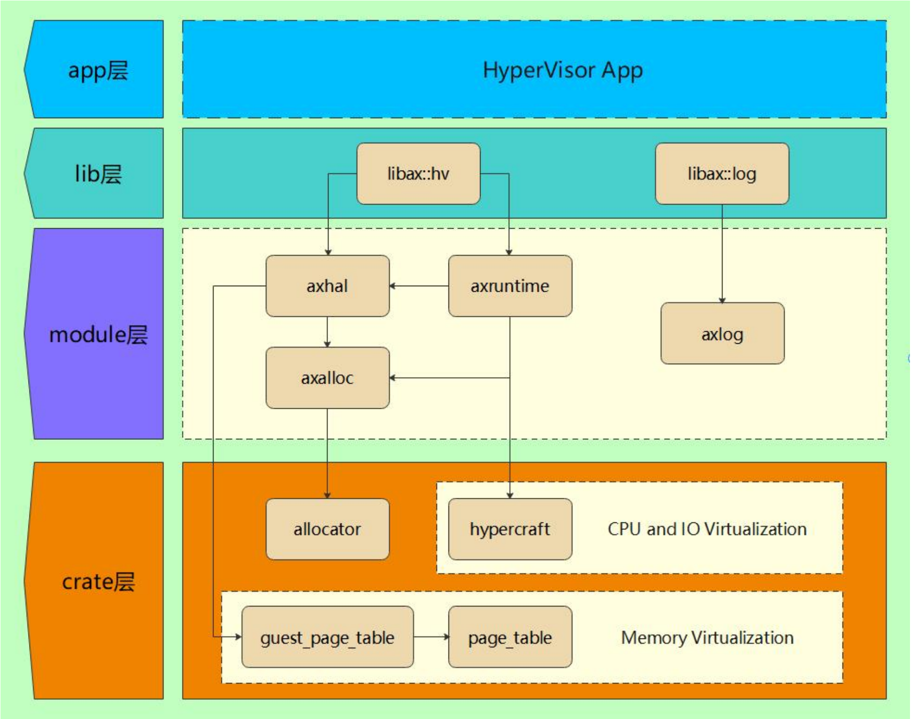
hcHyper 仓库的三大前身分别是 RVM-Tutorial ，rHyper ，hypercraft

RVM-Tutorial 是一个用 RUST 语言编写的 Type-1 Hypervisor ，支持在 x86(Intel) 的 PC 上运行，并支持 x86(Intel) 下的 CPU 和 Memory 的虚拟化，能够支持 NimbOS 操作系统。

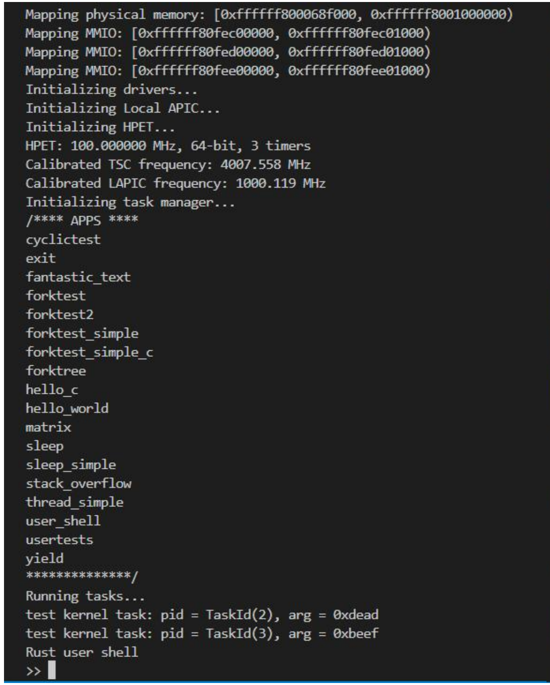
rHyper 是一个用 RUST 语言编写的 Type-1 Hypervisor ，支持在 QEMU 上运行，并支持 ARM-v8 下的 CPU 、 Memory 、IO 的虚拟化，能够支持 NimbOS \ Linux \ rCore \ ArceOS 操作系统，并且能够支持多核虚拟化。

hypercraft 是一个用 RUST 语言编写的 Type-2 Hypervisor ，支持在 QEMU 上运行，并支持 RISC-V 下的 CPU 、 Memory 、IO 的虚拟化，能够支持 NimbOS 操作系统。是首个结合 ArceOS 框架的操作系统。

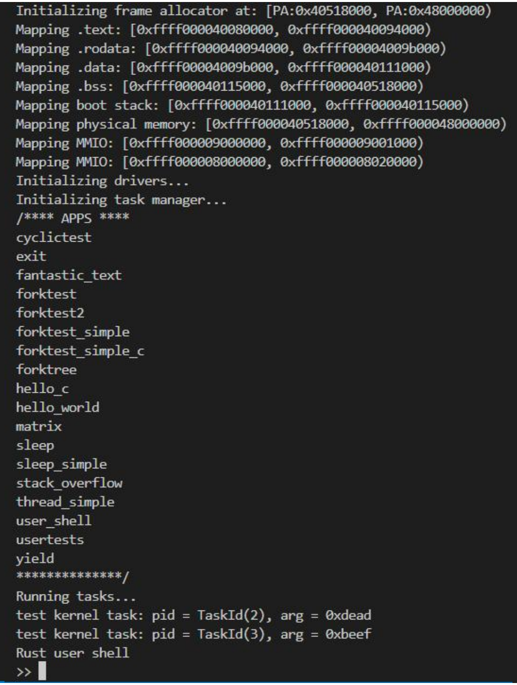
Design


app层
HyperVisor App
lib层
libax::hv
libax::log
module层
axhal
axruntime
axlog
axalloc
crate层
allocator
hypercraft
CPU and IO Virtualization
guest_page_table
page_table
Memory Virtualization

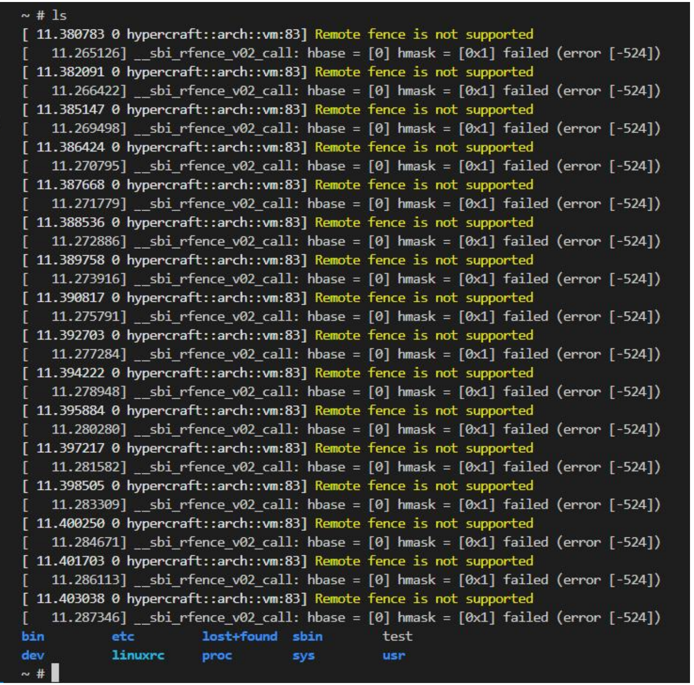
Result



```
Mapping physical memory: [0xffffff800068f000, 0xffffff8001000000)
Mapping MMIO: [0xffffff80fec00000, 0xffffff80fec01000)
Mapping MMIO: [0xffffff80fed00000, 0xffffff80fed01000)
Mapping MMIO: [0xffffff80fee00000, 0xffffff80fee01000)
Initializing drivers...
Initializing Local APIC...
Initializing HPET...
HPET: 100.000000 MHz, 64-bit, 3 timers
Calibrated TSC frequency: 4007.558 MHz
Calibrated LAPIC frequency: 1000.119 MHz
Initializing task manager...
/**** APPS ****
cyclictest
exit
fantastic_text
forktest
forktest2
forktest_simple
forktest_simple_c
forktree
hello_c
hello_world
matrix
sleep
sleep_simple
stack_overflow
thread_simple
user_shell
usertests
yield
**************/
Running tasks...
test kernel task: pid = TaskId(2), arg = 0xdead
test kernel task: pid = TaskId(3), arg = 0xbeef
Rust user shell
>> 
```



```
Initializing frame allocator at: [PA:0x40518000, PA:0x48000000)
Mapping .text: [0xffff000040080000, 0xffff000040094000)
Mapping .rodata: [0xffff000040094000, 0xffff00004009b000)
Mapping .data: [0xffff00004009b000, 0xffff000040111000)
Mapping .bss: [0xffff000040115000, 0xffff000040518000)
Mapping boot stack: [0xffff000040111000, 0xffff000040115000)
Mapping physical memory: [0xffff000040518000, 0xffff000048000000)
Mapping MMIO: [0xffff000009000000, 0xffff000009001000)
Mapping MMIO: [0xffff000008000000, 0xffff000008020000)
Initializing drivers...
Initializing task manager...
/**** APPS ****
cyclictest
exit
fantastic_text
forktest
forktest2
forktest_simple
forktest_simple_c
forktree
hello_c
hello_world
matrix
sleep
sleep_simple
stack_overflow
thread_simple
user_shell
usertests
yield
**************/
Running tasks...
test kernel task: pid = TaskId(2), arg = 0xdead
test kernel task: pid = TaskId(3), arg = 0xbeef
Rust user shell
>> 
```



```
~ # ls
[ 11.380783 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.265126] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.382091 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.266422] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.385147 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.269498] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.386424 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.270795] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.387668 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.271779] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.388536 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.272886] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.389758 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.273916] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.390817 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.275791] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.392703 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.277284] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.394222 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.278948] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.395884 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.280280] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.397217 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.281582] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.398505 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.283309] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.400250 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.284671] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.401703 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.286113] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
[ 11.403038 0 hypercraft::arch::vm:83] Remote fence is not supported
[   11.287346] __sbi_rfence_v02_call: hbase = [0] hmask = [0x1] failed (error [-524])
bin       etc       lost+found  sbin      test
dev       linuxrc   proc        sys       usr
~ # 
```

CPU Virtualization Basic

我们可以先温习一下有关于 CPU 虚拟化技术的知识，这些知识对于三大架构，乃至于所有的不模拟硬件的 Hypervisor 软件都适用。

hypervisor init：初始化一个 hypervisor ，包括初始化运行 hypervisor 的 CPU 基本启动信息，以及 hypervisor 本身的设置。

CPU allocation：分配 Guest 执行的 VCPU ，并分配对应的 PerCPU

VCPU：由 hypervisor 虚拟出来的，运行 guest 软件所需的每 CPU 的私有状态。类似于传统 OS 中的线程，一个 guest OS 可具有多个 vCPU，vCPU 数量与物理 CPU 数量无关。

PerCPU：hypervisor 管理的所有物理 CPU。

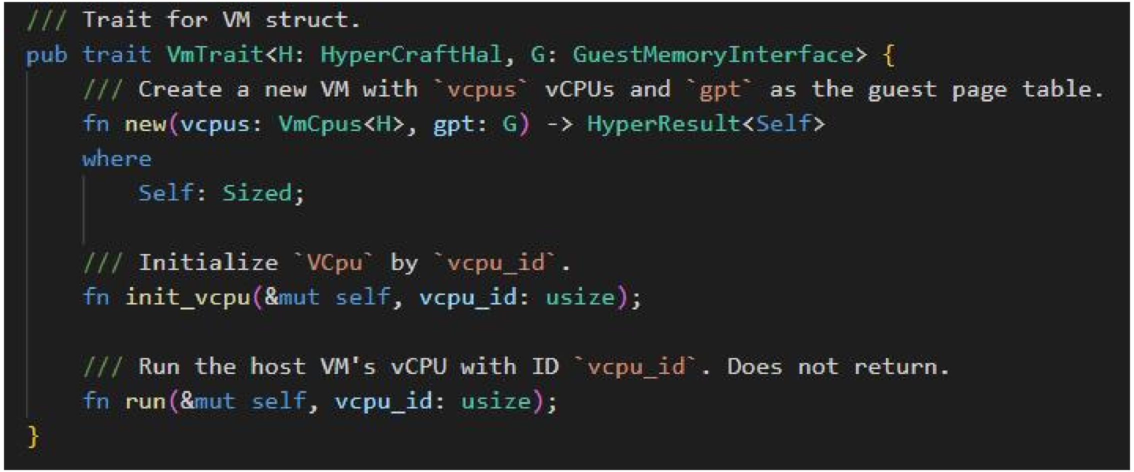
Memory allocation：分配 Guest 执行时的所有 VCPU 中的内存空间以及对应的真实物理内存。并建立 guest_page_table 保存对应的页表指针等等。

VM entry：从 host 模式切换到 guest 模式，开始执行 guest 软件的代码。

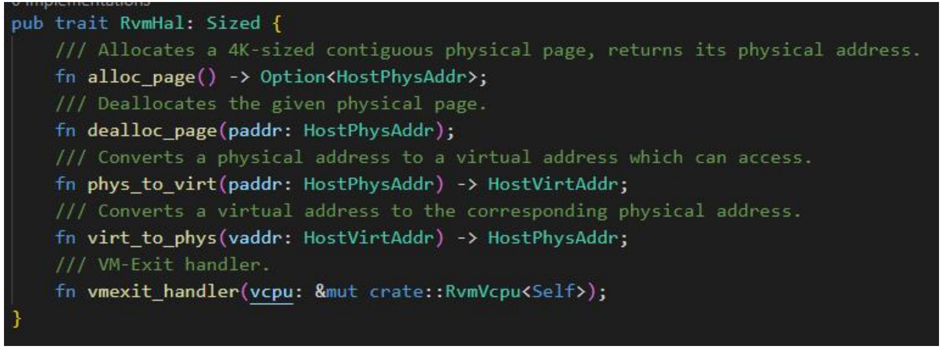
VM exit：从 guest 模式切换回 host 模式，开始执行 hypervisor 的代码。

Device virtualization：虚拟化 guest 软件执行的设备，包含直通，模拟等几种方式。

CPU Virtualization Basic

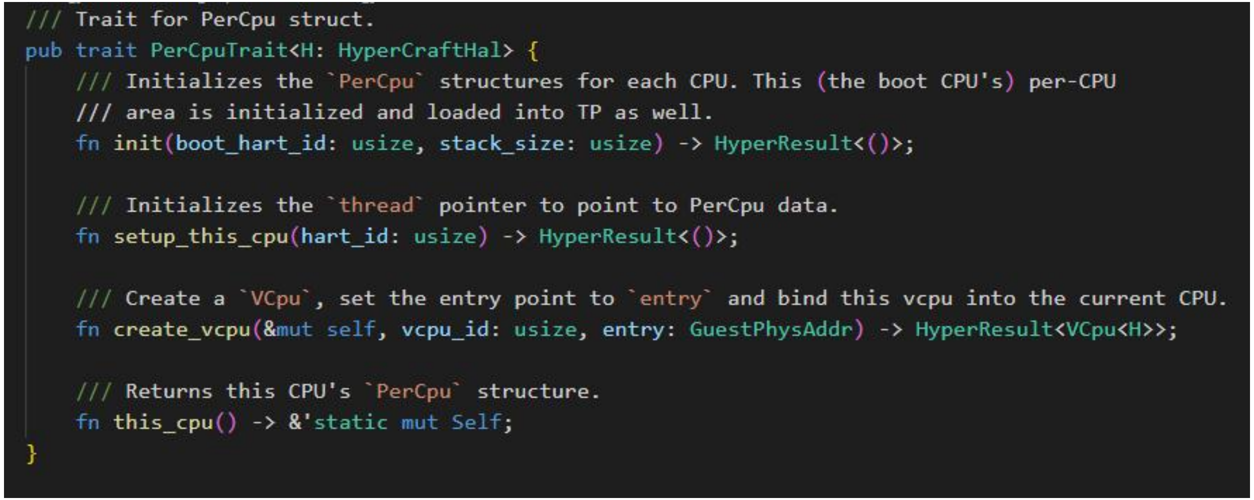


```rust
/// Trait for VM struct.
pub trait VmTrait<H: HyperCraftHal, G: GuestMemoryInterface> {
    /// Create a new VM with `vcpus` vCPUs and `gpt` as the guest page table.
    fn new(vcpus: VmCpus<H>, gpt: G) -> HyperResult<Self>
    where
        Self: Sized;

    /// Initialize `VCpu` by `vcpu_id`.
    fn init_vcpu(&mut self, vcpu_id: usize);

    /// Run the host VM's vCPU with ID `vcpu_id`. Does not return.
    fn run(&mut self, vcpu_id: usize);
}
```



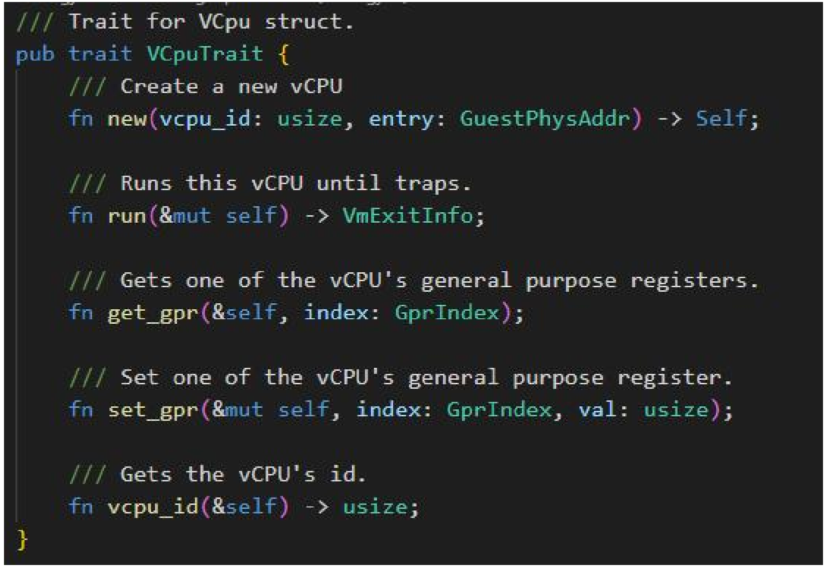
```rust
0 implementations
pub trait RvmHal: Sized {
    /// Allocates a 4K-sized contiguous physical page, returns its physical address.
    fn alloc_page() -> Option<HostPhysAddr>;
    /// Deallocates the given physical page.
    fn dealloc_page(paddr: HostPhysAddr);
    /// Converts a physical address to a virtual address which can access.
    fn phys_to_virt(paddr: HostPhysAddr) -> HostVirtAddr;
    /// Converts a virtual address to the corresponding physical address.
    fn virt_to_phys(vaddr: HostVirtAddr) -> HostPhysAddr;
    /// VM-Exit handler.
    fn vmexit_handler(vcpu: &mut crate::RvmVcpu<Self>);
}
```

PerCPU



```rust
/// Trait for PerCpu struct.
pub trait PerCpuTrait<H: HyperCraftHal> {
    /// Initializes the `PerCpu` structures for each CPU. This (the boot CPU's) per-CPU
    /// area is initialized and loaded into TP as well.
    fn init(boot_hart_id: usize, stack_size: usize) -> HyperResult<()>;

    /// Initializes the `thread` pointer to point to PerCpu data.
    fn setup_this_cpu(hart_id: usize) -> HyperResult<()>;

    /// Create a `VCpu`, set the entry point to `entry` and bind this vcpu into the current CPU.
    fn create_vcpu(&mut self, vcpu_id: usize, entry: GuestPhysAddr) -> HyperResult<VCpu<H>>;

    /// Returns this CPU's `PerCpu` structure.
    fn this_cpu() -> &'static mut Self;
}
```

VCPU



```rust
/// Trait for VCpu struct.
pub trait VCpuTrait {
    /// Create a new vCPU
    fn new(vcpu_id: usize, entry: GuestPhysAddr) -> Self;

    /// Runs this vCPU until traps.
    fn run(&mut self) -> VmExitInfo;

    /// Gets one of the vCPU's general purpose registers.
    fn get_gpr(&self, index: GprIndex);

    /// Set one of the vCPU's general purpose register.
    fn set_gpr(&mut self, index: GprIndex, val: usize);

    /// Gets the vCPU's id.
    fn vcpu_id(&self) -> usize;
}
```

X86 Init

1. x86 的基本启动设置

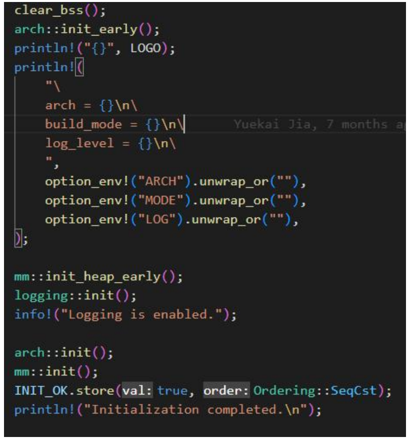
1. 检查硬件的虚拟化支持

1. 初始化 VMXON 空间

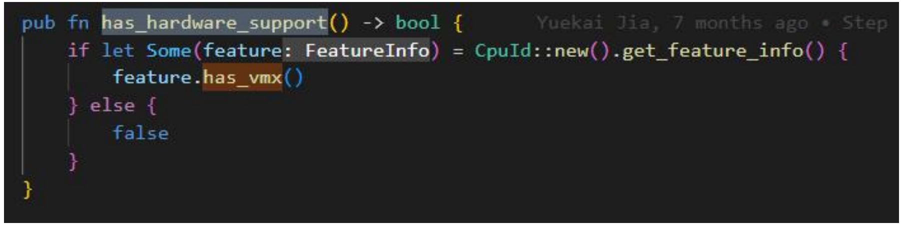
1. 执行 VMXON

1. 初始化 VMCS

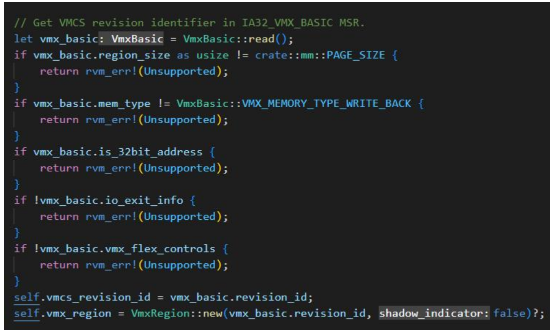
X86 Init



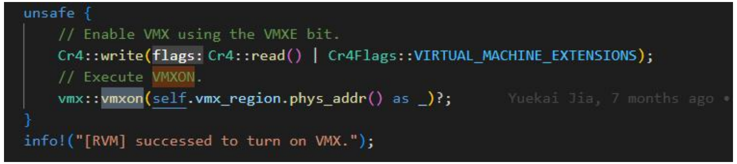
```
clear_bss();
arch::init_early();
println!("{}", LOGO);
println!(
    "\
    arch = {}\n\
    build_mode = {}\n\          Yuekai Jia, 7 months a
    log_level = {}\n\
    ",
    option_env!("ARCH").unwrap_or(""),
    option_env!("MODE").unwrap_or(""),
    option_env!("LOG").unwrap_or(""),
);

mm::init_heap_early();
logging::init();
info!("Logging is enabled.");

arch::init();
mm::init();
INIT_OK.store(val: true, order: Ordering::SeqCst);
println!("Initialization completed.\n");
```



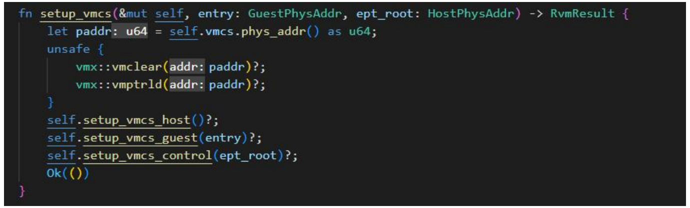
```
pub fn has_hardware_support() -> bool {      Yuekai Jia, 7 months ago • Step
    if let Some(feature: FeatureInfo) = CpuId::new().get_feature_info() {
        feature.has_vmx()
    } else {
        false
    }
}
```



```
// Get VMCS revision identifier in IA32_VMX_BASIC MSR.
let vmx_basic: VmxBasic = VmxBasic::read();
if vmx_basic.region_size as usize != crate::mm::PAGE_SIZE {
    return rvm_err!(Unsupported);
}
if vmx_basic.mem_type != VmxBasic::VMX_MEMORY_TYPE_WRITE_BACK {
    return rvm_err!(Unsupported);
}
if vmx_basic.is_32bit_address {
    return rvm_err!(Unsupported);
}
if !vmx_basic.io_exit_info {
    return rvm_err!(Unsupported);
}
if !vmx_basic.vmx_flex_controls {
    return rvm_err!(Unsupported);
}
self.vmcs_revision_id = vmx_basic.revision_id;
self.vmx_region = VmxRegion::new(vmx_basic.revision_id, shadow_indicator: false)?;
```



```
unsafe {
    // Enable VMX using the VMXE bit.
    Cr4::write(flags: Cr4::read() | Cr4Flags::VIRTUAL_MACHINE_EXTENSIONS);
    // Execute VMXON.
    vmx::vmxon(self.vmx_region.phys_addr() as _)?;      Yuekai Jia, 7 months ago •
}
info!("[RVM] successed to turn on VMX.");
```



```
fn setup_vmcs(&mut self, entry: GuestPhysAddr, ept_root: HostPhysAddr) -> RvmResult {
    let paddr: u64 = self.vmcs.phys_addr() as u64;
    unsafe {
        vmx::vmclear(addr: paddr)?;
        vmx::vmptrld(addr: paddr)?;
    }
    self.setup_vmcs_host()?;
    self.setup_vmcs_guest(entry)?;
    self.setup_vmcs_control(ept_root)?;
    Ok(())
}
```

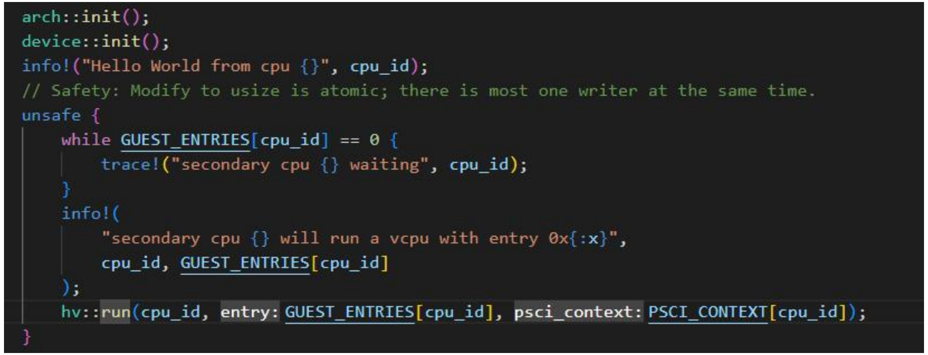
ARM Init

1. ARM 的基本启动设置

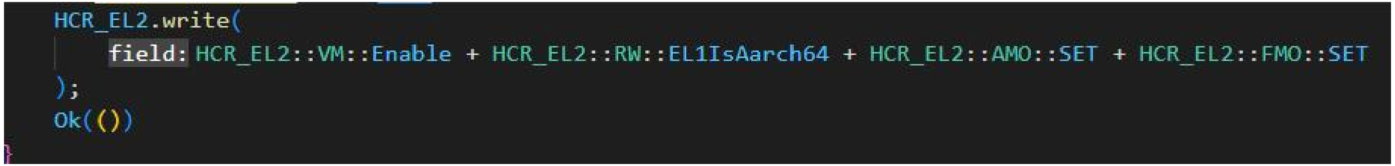
1. 检查硬件的虚拟化支持

1. 设置 HCR_EL2 寄存器

ARM Init



```
arch::init();
device::init();
info!("Hello World from cpu {}", cpu_id);
// Safety: Modify to usize is atomic; there is most one writer at the same time.
unsafe {
    while GUEST_ENTRIES[cpu_id] == 0 {
        trace!("secondary cpu {} waiting", cpu_id);
    }
    info!(
        "secondary cpu {} will run a vcpu with entry 0x{:x}",
        cpu_id, GUEST_ENTRIES[cpu_id]
    );
    hv::run(cpu_id, entry: GUEST_ENTRIES[cpu_id], psci_context: PSCI_CONTEXT[cpu_id]);
}
```



```
    HCR_EL2.write(
        field: HCR_EL2::VM::Enable + HCR_EL2::RW::EL1IsAarch64 + HCR_EL2::AMO::SET + HCR_EL2::FMO::SET
    );
    Ok(())
}
```

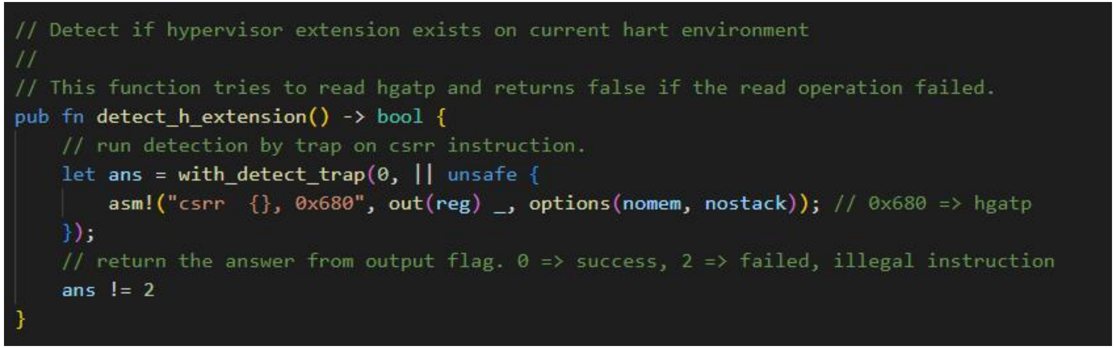
RISC-V Init

1. **RISC-V** 的基本启动设置

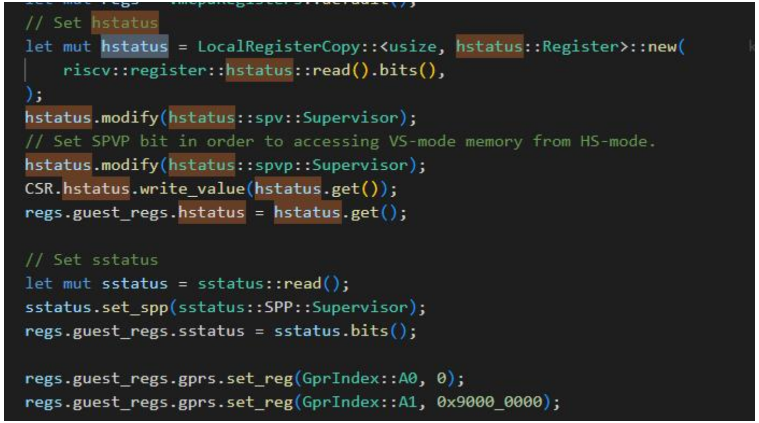
1. 检查硬件的虚拟化支持

1. 初始化 **hypervisor** 寄存器

RISC-V Init

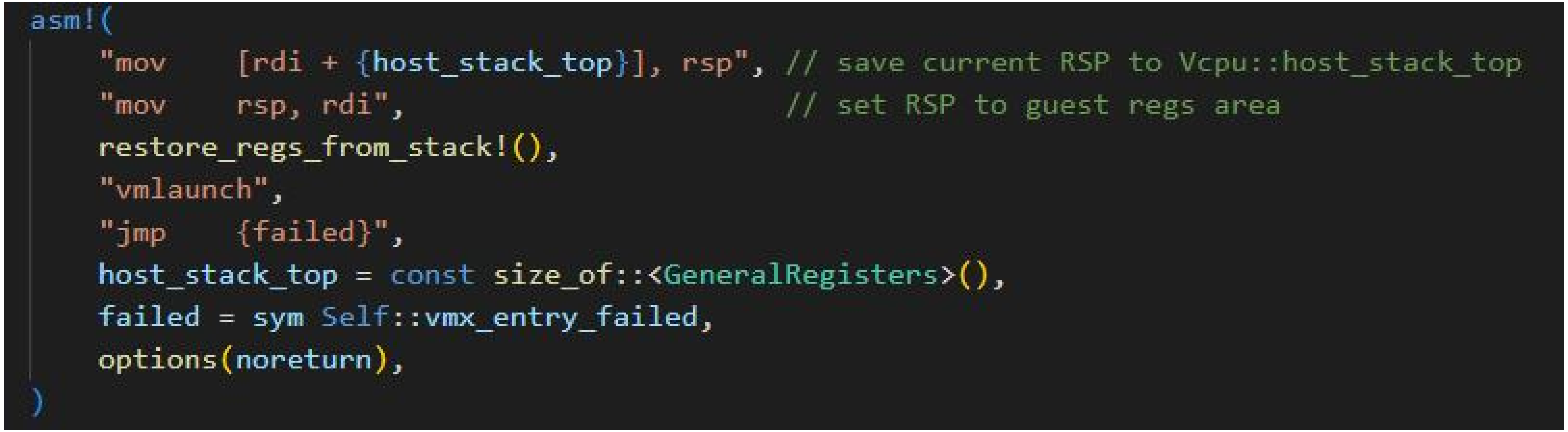


```rust
// Detect if hypervisor extension exists on current hart environment
//
// This function tries to read hgatp and returns false if the read operation failed.
pub fn detect_h_extension() -> bool {
    // run detection by trap on csrr instruction.
    let ans = with_detect_trap(0, || unsafe {
        asm!("csrr  {}, 0x680", out(reg) _, options(nomem, nostack)); // 0x680 => hgatp
    });
    // return the answer from output flag. 0 => success, 2 => failed, illegal instruction
    ans != 2
}
```



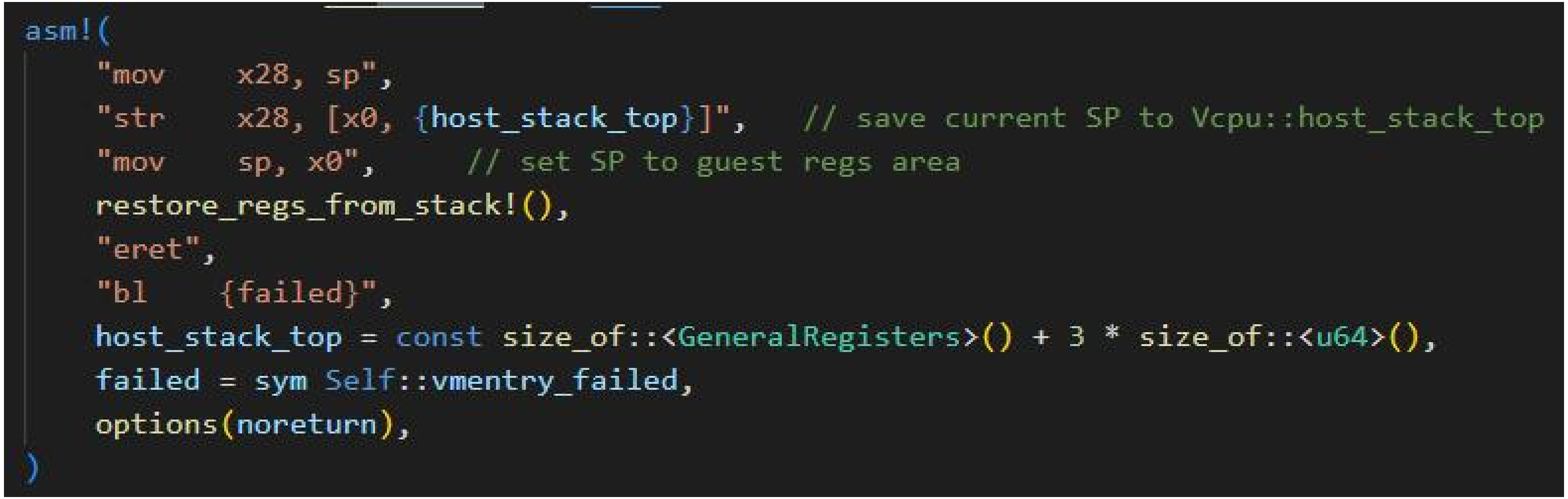
```rust
let mut regs = VmCpuRegisters::default();
// Set hstatus
let mut hstatus = LocalRegisterCopy::<usize, hstatus::Register>::new(
    riscv::register::hstatus::read().bits(),
);
hstatus.modify(hstatus::spv::Supervisor);
// Set SPVP bit in order to accessing VS-mode memory from HS-mode.
hstatus.modify(hstatus::spvp::Supervisor);
CSR.hstatus.write_value(hstatus.get());
regs.guest_regs.hstatus = hstatus.get();

// Set sstatus
let mut sstatus = sstatus::read();
sstatus.set_spp(sstatus::SPP::Supervisor);
regs.guest_regs.sstatus = sstatus.bits();

regs.guest_regs.gprs.set_reg(GprIndex::A0, 0);
regs.guest_regs.gprs.set_reg(GprIndex::A1, 0x9000_0000);
```

x86 launch



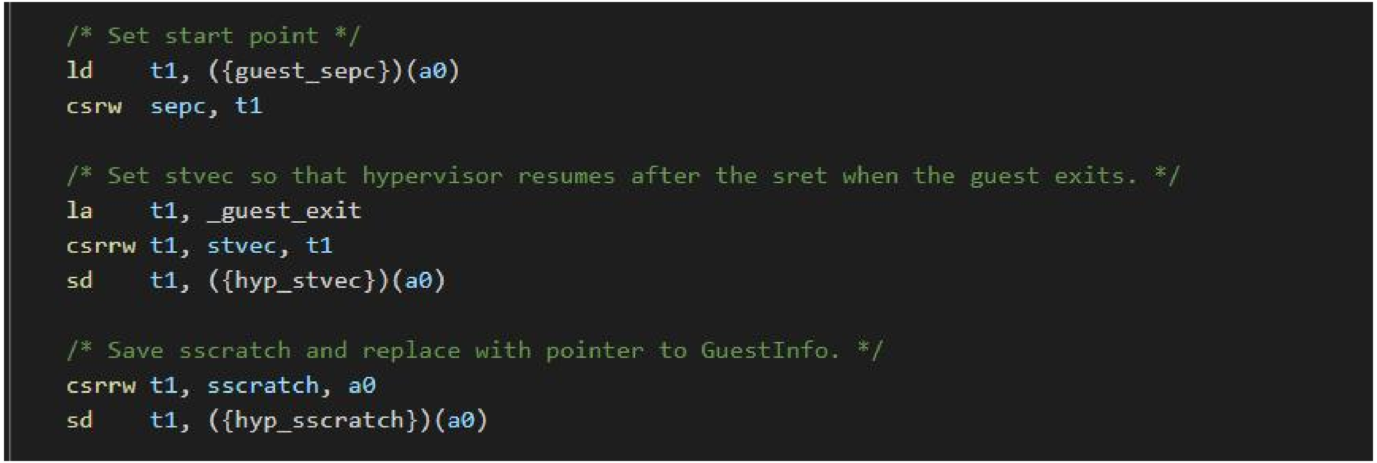
```
asm!(
    "mov    [rdi + {host_stack_top}], rsp", // save current RSP to Vcpu::host_stack_top
    "mov    rsp, rdi",                      // set RSP to guest regs area
    restore_regs_from_stack!(),
    "vmlaunch",
    "jmp    {failed}",
    host_stack_top = const size_of::<GeneralRegisters>(),
    failed = sym Self::vmx_entry_failed,
    options(noreturn),
)
```

arm launch

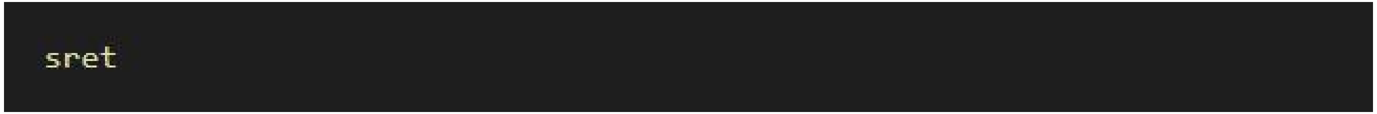


```
asm!(
    "mov    x28, sp",
    "str    x28, [x0, {host_stack_top}]",   // save current SP to Vcpu::host_stack_top
    "mov    sp, x0",     // set SP to guest regs area
    restore_regs_from_stack!(),
    "eret",
    "bl    {failed}",
    host_stack_top = const size_of::<GeneralRegisters>() + 3 * size_of::<u64>(),
    failed = sym Self::vmentry_failed,
    options(noreturn),
)
```

risc-v launch



```
/* Set start point */
ld    t1, ({guest_sepc})(a0)
csrw  sepc, t1

/* Set stvec so that hypervisor resumes after the sret when the guest exits. */
la    t1, _guest_exit
csrrw t1, stvec, t1
sd    t1, ({hyp_stvec})(a0)

/* Save sscratch and replace with pointer to GuestInfo. */
csrrw t1, sscratch, a0
sd    t1, ({hyp_sscratch})(a0)
```



```
sret
```

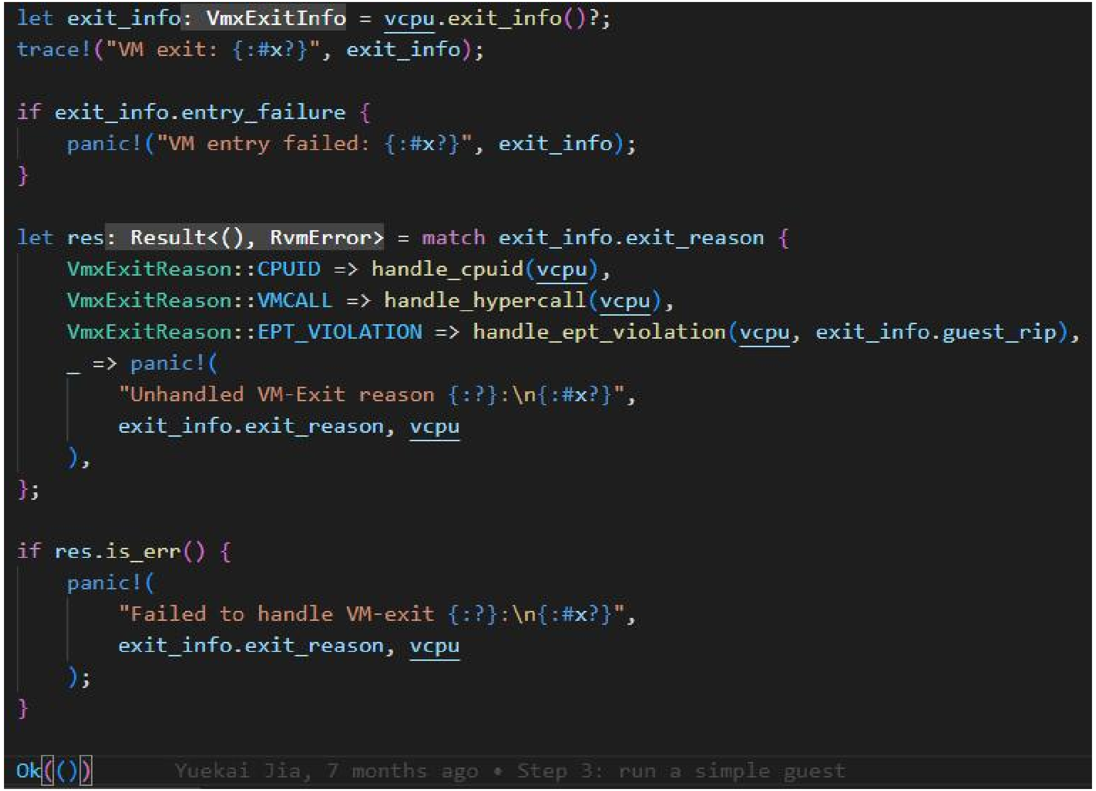
x86 exit-handler

1. CPUID 指令的处理

1. VMCALL 对 hypervisor 的调用

1. EPT VIOLATION

x86 exit-handler



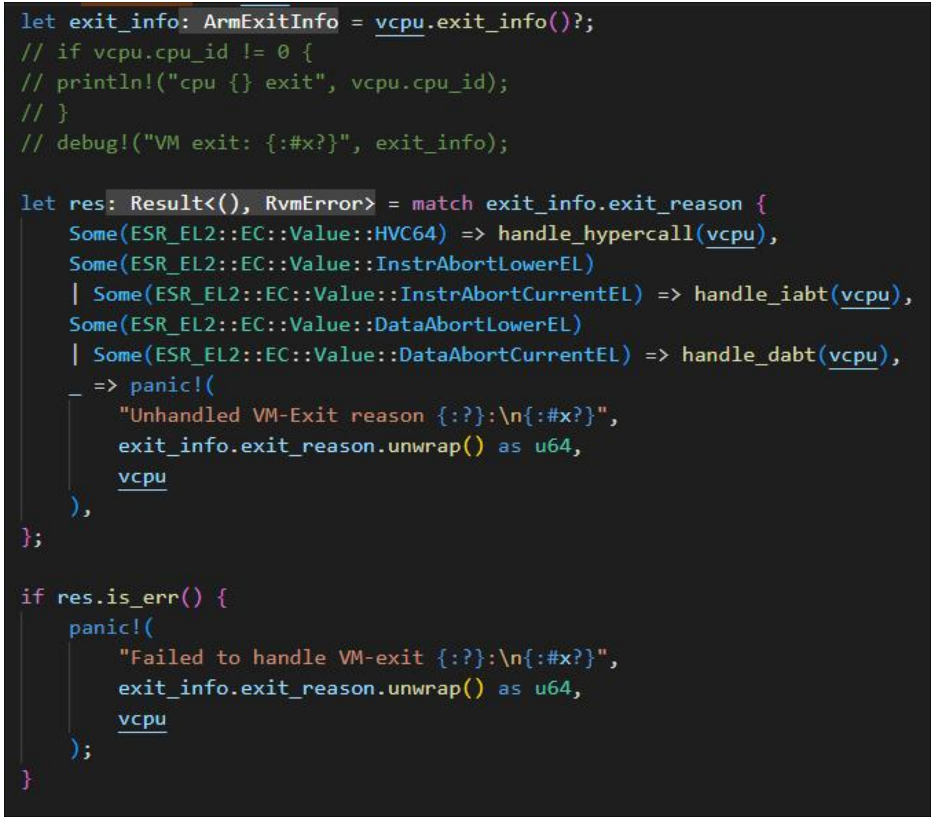
```
let exit_info: VmxExitInfo = vcpu.exit_info()?;
trace!("VM exit: {:#x?}", exit_info);

if exit_info.entry_failure {
    panic!("VM entry failed: {:#x?}", exit_info);
}

let res: Result<(), RvmError> = match exit_info.exit_reason {
    VmxExitReason::CPUID => handle_cpuid(vcpu),
    VmxExitReason::VMCALL => handle_hypercall(vcpu),
    VmxExitReason::EPT_VIOLATION => handle_ept_violation(vcpu, exit_info.guest_rip),
    _ => panic!(
        "Unhandled VM-Exit reason {:?}:\n{:#x?}",
        exit_info.exit_reason, vcpu
    ),
};

if res.is_err() {
    panic!(
        "Failed to handle VM-exit {:?}:\n{:#x?}",
        exit_info.exit_reason, vcpu
    );
}

Ok(())      Yuekai Jia, 7 months ago • Step 3: run a simple guest
```

arm exit-handler

1. hypercall HVC64 指令

1. DataAbort \ InstAbort

arm exit-handler



```rust
let exit_info: ArmExitInfo = vcpu.exit_info()?;
// if vcpu.cpu_id != 0 {
// println!("cpu {} exit", vcpu.cpu_id);
// }
// debug!("VM exit: {:#x?}", exit_info);

let res: Result<(), RvmError> = match exit_info.exit_reason {
    Some(ESR_EL2::EC::Value::HVC64) => handle_hypercall(vcpu),
    Some(ESR_EL2::EC::Value::InstrAbortLowerEL)
    | Some(ESR_EL2::EC::Value::InstrAbortCurrentEL) => handle_iabt(vcpu),
    Some(ESR_EL2::EC::Value::DataAbortLowerEL)
    | Some(ESR_EL2::EC::Value::DataAbortCurrentEL) => handle_dabt(vcpu),
    _ => panic!(
        "Unhandled VM-Exit reason {:?}:\n{:#x?}",
        exit_info.exit_reason.unwrap() as u64,
        vcpu
    ),
};

if res.is_err() {
    panic!(
        "Failed to handle VM-exit {:?}:\n{:#x?}",
        exit_info.exit_reason.unwrap() as u64,
        vcpu
    );
}
```

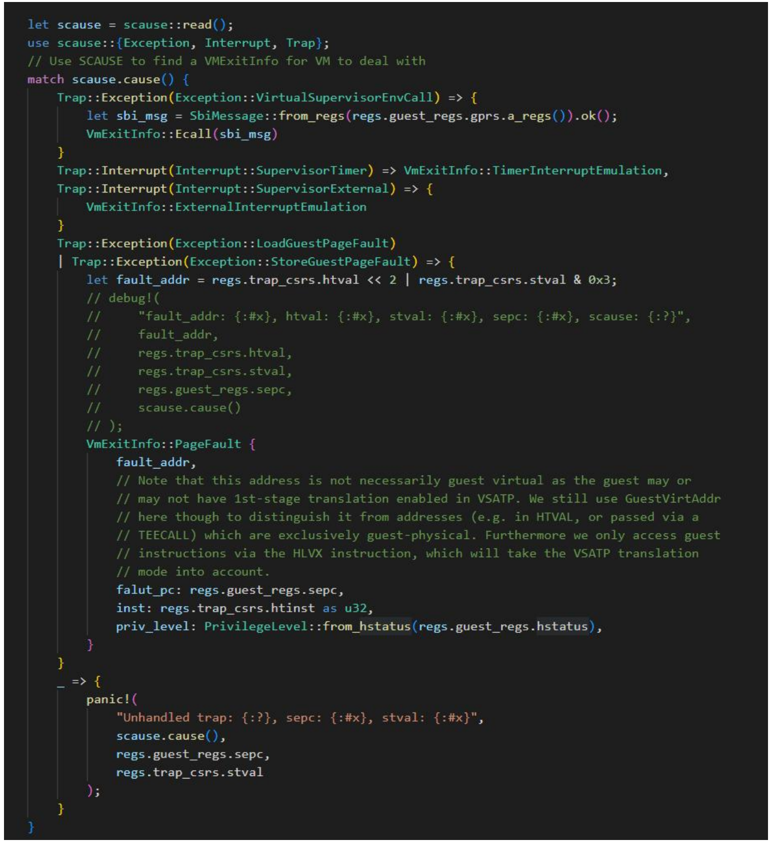
risc-v exit-handler

1. hypercall ECALL 指令

1. PAGEFAULT

1. TIMER GuestOS 时钟中断

risc-v exit-handler



```rust
let scause = scause::read();
use scause::{Exception, Interrupt, Trap};
// Use SCAUSE to find a VMExitInfo for VM to deal with
match scause.cause() {
    Trap::Exception(Exception::VirtualSupervisorEnvCall) => {
        let sbi_msg = SbiMessage::from_regs(regs.guest_regs.gprs.a_regs()).ok();
        VmExitInfo::Ecall(sbi_msg)
    }
    Trap::Interrupt(Interrupt::SupervisorTimer) => VmExitInfo::TimerInterruptEmulation,
    Trap::Interrupt(Interrupt::SupervisorExternal) => {
        VmExitInfo::ExternalInterruptEmulation
    }
    Trap::Exception(Exception::LoadGuestPageFault)
    | Trap::Exception(Exception::StoreGuestPageFault) => {
        let fault_addr = regs.trap_csrs.htval << 2 | regs.trap_csrs.stval & 0x3;
        // debug!(
        //     "fault_addr: {:#x}, htval: {:#x}, stval: {:#x}, sepc: {:#x}, scause: {:?}",
        //     fault_addr,
        //     regs.trap_csrs.htval,
        //     regs.trap_csrs.stval,
        //     regs.guest_regs.sepc,
        //     scause.cause()
        // );
        VmExitInfo::PageFault {
            fault_addr,
            // Note that this address is not necessarily guest virtual as the guest may or
            // may not have 1st-stage translation enabled in VSATP. We still use GuestVirtAddr
            // here though to distinguish it from addresses (e.g. in HTVAL, or passed via a
            // TEECALL) which are exclusively guest-physical. Furthermore we only access guest
            // instructions via the HLVX instruction, which will take the VSATP translation
            // mode into account.
            falut_pc: regs.guest_regs.sepc,
            inst: regs.trap_csrs.htinst as u32,
            priv_level: PrivilegeLevel::from_hstatus(regs.guest_regs.hstatus),
        }
    }
    _ => {
        panic!(
            "Unhandled trap: {:?}, sepc: {:#x}, stval: {:#x}",
            scause.cause(),
            regs.guest_regs.sepc,
            regs.trap_csrs.stval
        );
    }
}
```

Memory Virtualization Basic

我们同样先温习一下有关于 Memory 虚拟化技术的知识，这些知识对于三大架构，乃至于所有的不模拟硬件的 Hypervisor 软件都适用。

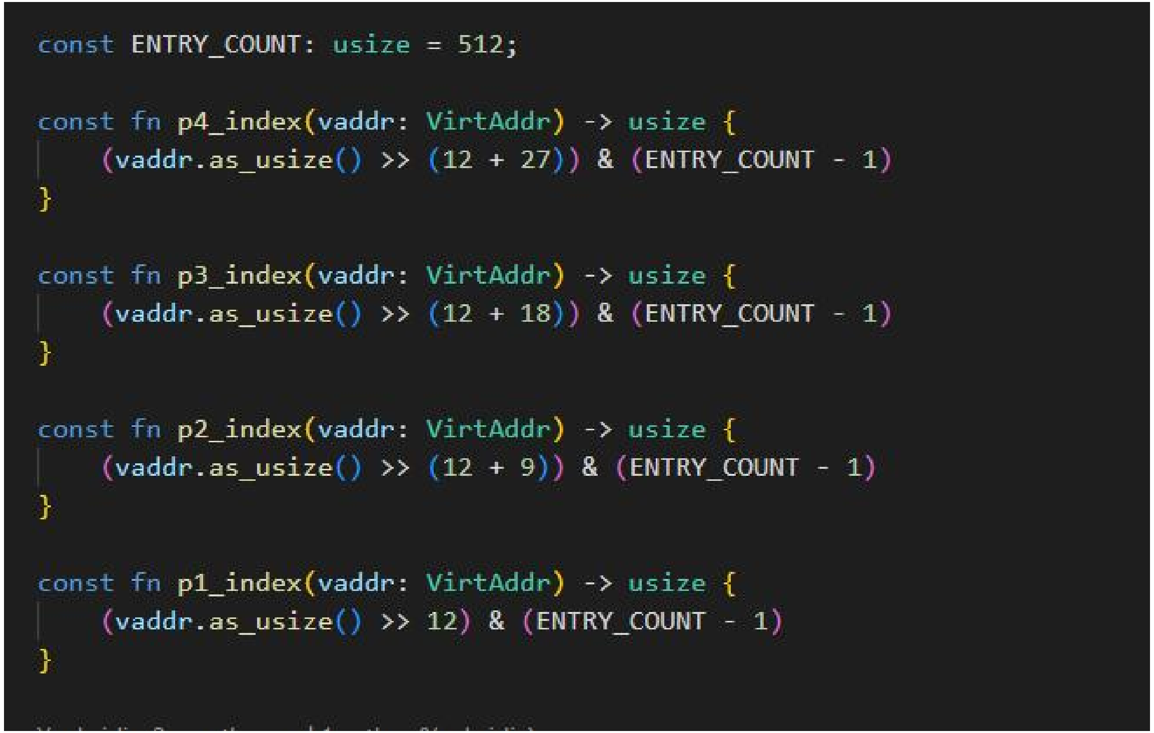
Memory allocation: 分配 Guest 执行时的所有 VCPU 中的内存空间以及对应的真实物理内存。并建立 guest_page_table 保存对应的页。表指针等等

guest page table: 支持 guest physaddr 到真实的物理地址的地址转换的页表

guest physaddr: 用户认为自己所在的地址

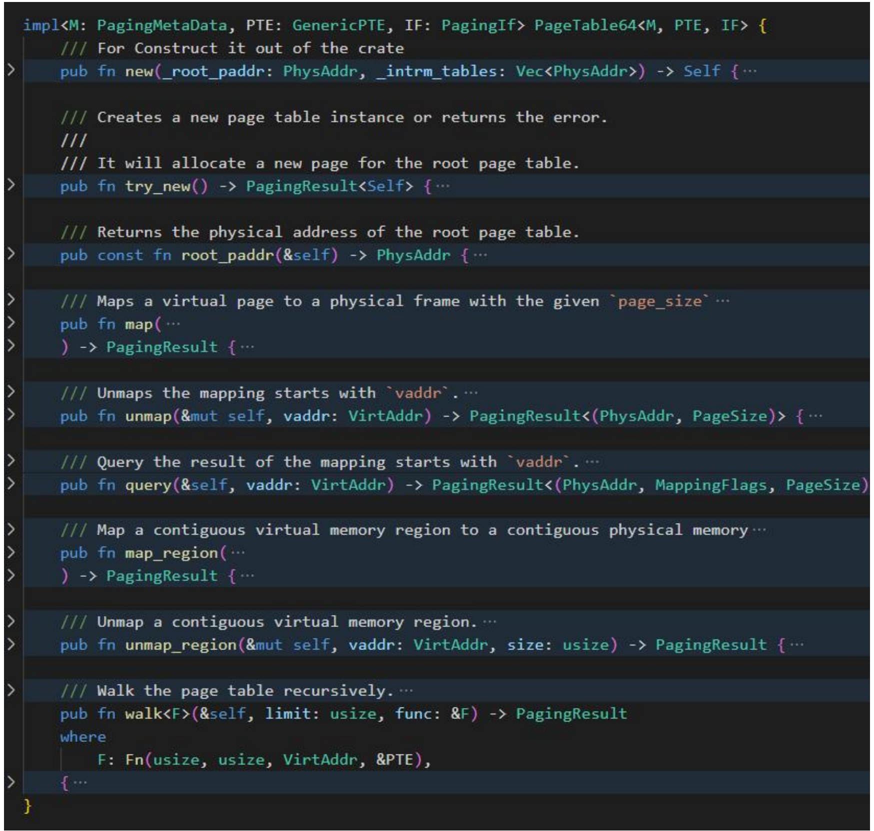
two stage translation: 二级页表转换，即 guest app 的 guest virtaddr 向 guest os 的 guest physaddr 再向真实的物理地址的转换。

ArceOS PageTable



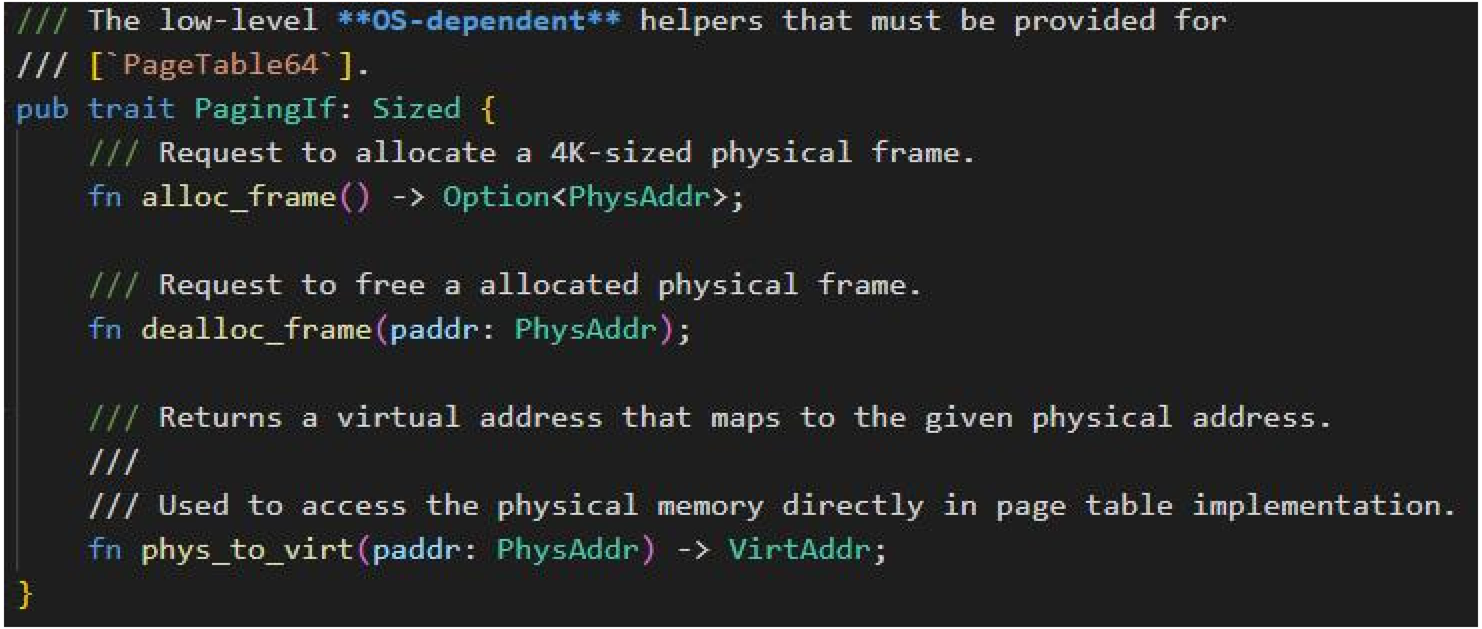
```
const ENTRY_COUNT: usize = 512;

const fn p4_index(vaddr: VirtAddr) -> usize {
    (vaddr.as_usize() >> (12 + 27)) & (ENTRY_COUNT - 1)
}

const fn p3_index(vaddr: VirtAddr) -> usize {
    (vaddr.as_usize() >> (12 + 18)) & (ENTRY_COUNT - 1)
}

const fn p2_index(vaddr: VirtAddr) -> usize {
    (vaddr.as_usize() >> (12 + 9)) & (ENTRY_COUNT - 1)
}

const fn p1_index(vaddr: VirtAddr) -> usize {
    (vaddr.as_usize() >> 12) & (ENTRY_COUNT - 1)
}
```

ArceOS PageTable



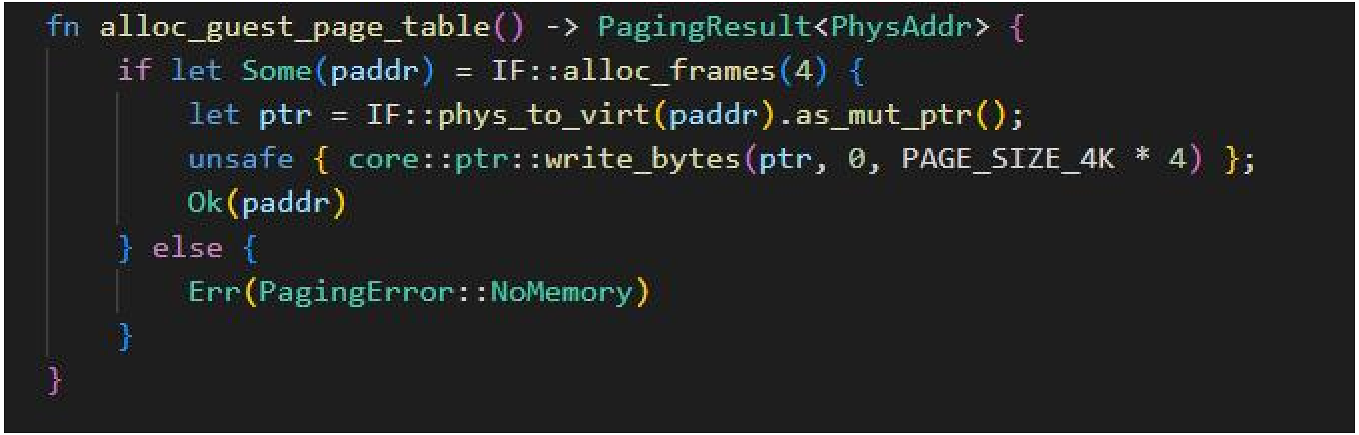
```
impl<M: PagingMetaData, PTE: GenericPTE, IF: PagingIf> PageTable64<M, PTE, IF> {
    /// For Construct it out of the crate
    pub fn new(_root_paddr: PhysAddr, _intrm_tables: Vec<PhysAddr>) -> Self {⋯

    /// Creates a new page table instance or returns the error.
    ///
    /// It will allocate a new page for the root page table.
    pub fn try_new() -> PagingResult<Self> {⋯

    /// Returns the physical address of the root page table.
    pub const fn root_paddr(&self) -> PhysAddr {⋯

    /// Maps a virtual page to a physical frame with the given `page_size`⋯
    pub fn map(⋯
    ) -> PagingResult {⋯

    /// Unmaps the mapping starts with `vaddr`.⋯
    pub fn unmap(&mut self, vaddr: VirtAddr) -> PagingResult<(PhysAddr, PageSize)> {⋯

    /// Query the result of the mapping starts with `vaddr`.⋯
    pub fn query(&self, vaddr: VirtAddr) -> PagingResult<(PhysAddr, MappingFlags, PageSize)

    /// Map a contiguous virtual memory region to a contiguous physical memory⋯
    pub fn map_region(⋯
    ) -> PagingResult {⋯

    /// Unmap a contiguous virtual memory region.⋯
    pub fn unmap_region(&mut self, vaddr: VirtAddr, size: usize) -> PagingResult {⋯

    /// Walk the page table recursively.⋯
    pub fn walk<F>(&self, limit: usize, func: &F) -> PagingResult
    where
        F: Fn(usize, usize, VirtAddr, &PTE),
    {⋯
}
```

ArceOS PageTable



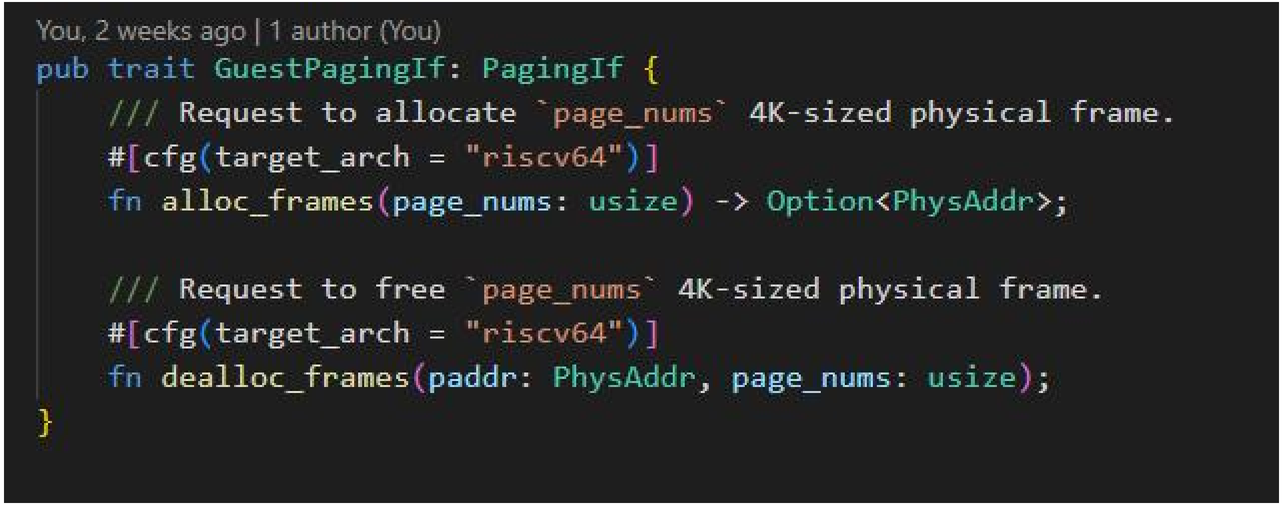
```rust
/// The low-level **OS-dependent** helpers that must be provided for
/// [`PageTable64`].
pub trait PagingIf: Sized {
    /// Request to allocate a 4K-sized physical frame.
    fn alloc_frame() -> Option<PhysAddr>;

    /// Request to free a allocated physical frame.
    fn dealloc_frame(paddr: PhysAddr);

    /// Returns a virtual address that maps to the given physical address.
    ///
    /// Used to access the physical memory directly in page table implementation.
    fn phys_to_virt(paddr: PhysAddr) -> VirtAddr;
}
```

Guest PageTable



```
fn alloc_guest_page_table() -> PagingResult<PhysAddr> {
    if let Some(paddr) = IF::alloc_frames(4) {
        let ptr = IF::phys_to_virt(paddr).as_mut_ptr();
        unsafe { core::ptr::write_bytes(ptr, 0, PAGE_SIZE_4K * 4) };
        Ok(paddr)
    } else {
        Err(PagingError::NoMemory)
    }
}
```

Guest PageTable



You, 2 weeks ago | 1 author (You)

```rust
pub trait GuestPagingIf: PagingIf {
    /// Request to allocate `page_nums` 4K-sized physical frame.
    #[cfg(target_arch = "riscv64")]
    fn alloc_frames(page_nums: usize) -> Option<PhysAddr>;

    /// Request to free `page_nums` 4K-sized physical frame.
    #[cfg(target_arch = "riscv64")]
    fn dealloc_frames(paddr: PhysAddr, page_nums: usize);
}
```