A Walkthrough on some recent KVM performance improvements

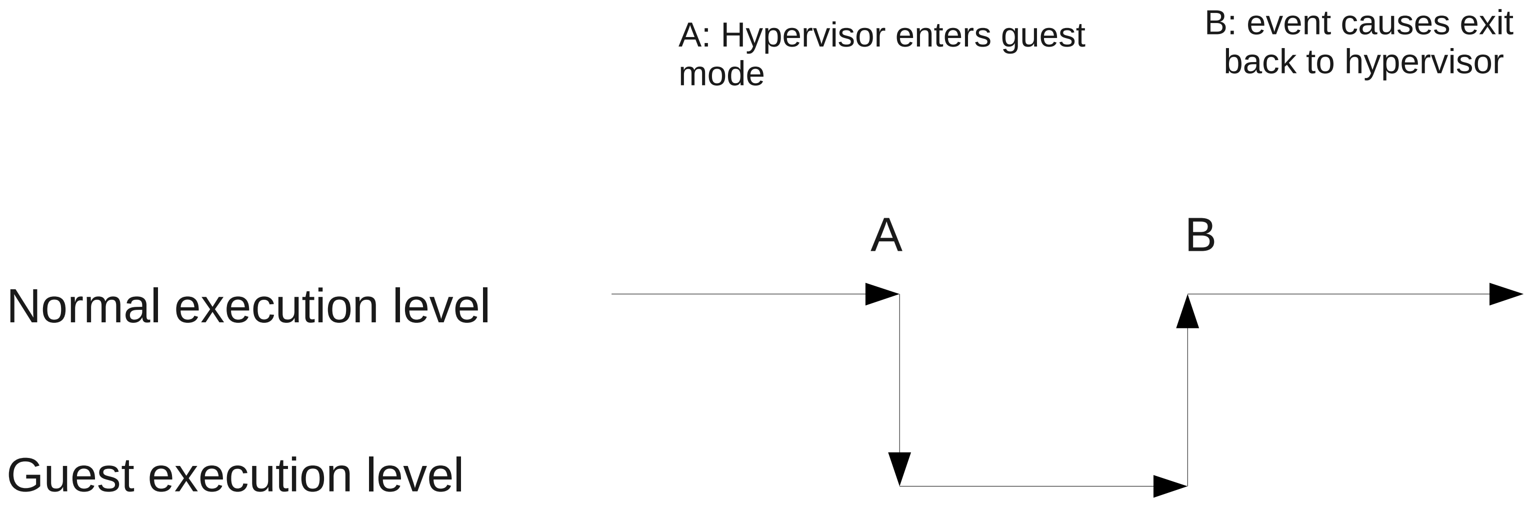
Marcelo Tosatti, Red Hat
KVM Forum 2010

Introduction

- overview of mode switch overhead
- description of recent kvm improvements

X86 hardware assisted virtualization

- Special processor mode which allows guest code to run natively most of the time, except for certain operations which trap back to the hypervisor.


A: Hypervisor enters guest mode
B: event causes exit back to hypervisor
A
B
Normal execution level
Guest execution level

Mode switch is expensive

1) save host context
2) vm entry
3) vm resume
4) restore host context

Solutions:

➔ Reduce number of exits (eg. virtio, TDP)
➔ Reduce processing time of an exit

PIO write exit timings, i7 2.6GHz

0) out		
1) host acquires control	400 cycles	(+400)
2) exit handler runs	800 cycles	(+400)
3) kernel pio search ends	1800 cycles	(+1000)
4) all host state restored (MSRs, FPU)	3800 cycles	(+2000)
5) qemu returns to kernel	6700 cycles	(+3000)
6) guest state reloaded (MSRs, FPU)	8500 cycles	(+1800)
7) guest entry	9600 cycles	(+1100)
8) back to guest mode	10400 cycles	(+800)

APIC accesses

- xAPIC accessed via memory mapped IO.
- On access a vmexit is generated (pagefault / apic access).
- KVM has to decode instruction, and translate virtual address.
- EOI is most frequent exit for common workloads.

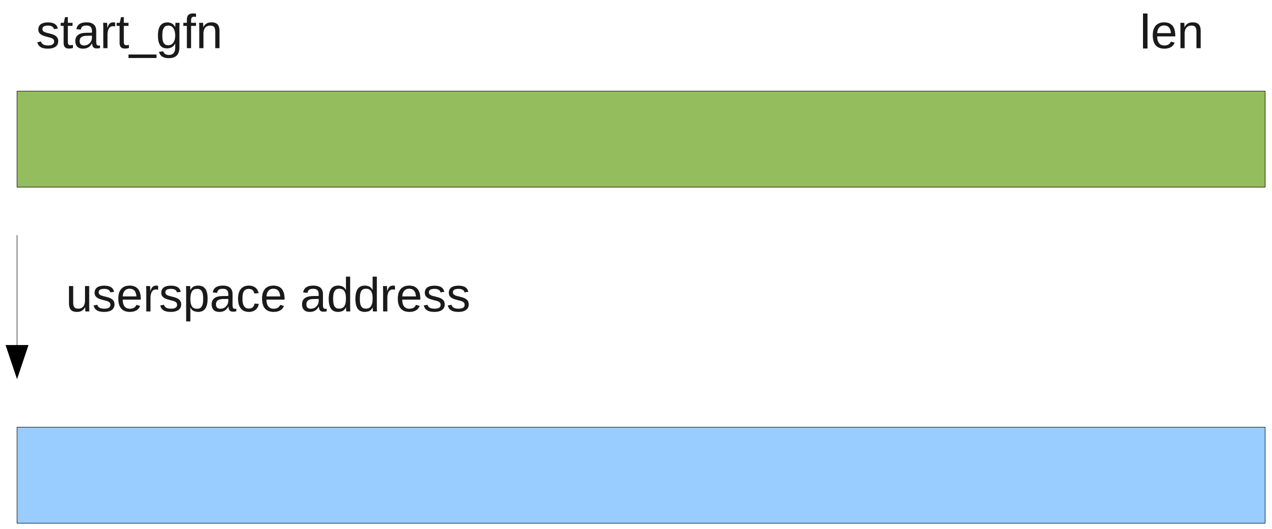
x2APIC

- Next generation APIC.
- RDMSR / WRMSR (read/write model specific register) as the interface.
- No need to decode instruction, address and value available in registers.

x2APIC performance

- EOI 4677 cycles vs 2366 cycles (approx 100% improvement).

KVM memslots



KVM in-kernel device emulation

- Certain devices emulated in-kernel for performance reasons (APIC, IO-APIC, PIT).
- Registered ranges in the PIO or memory space.

slots_lock

- ➔ Memslots and in-kernel device ranges protected by a read-write semaphore.
- ➔ Taken on every exit.
- ➔ Cacheline bouncing.

RCU

1) Atomically publish new structure.

2) Wait for grace period.

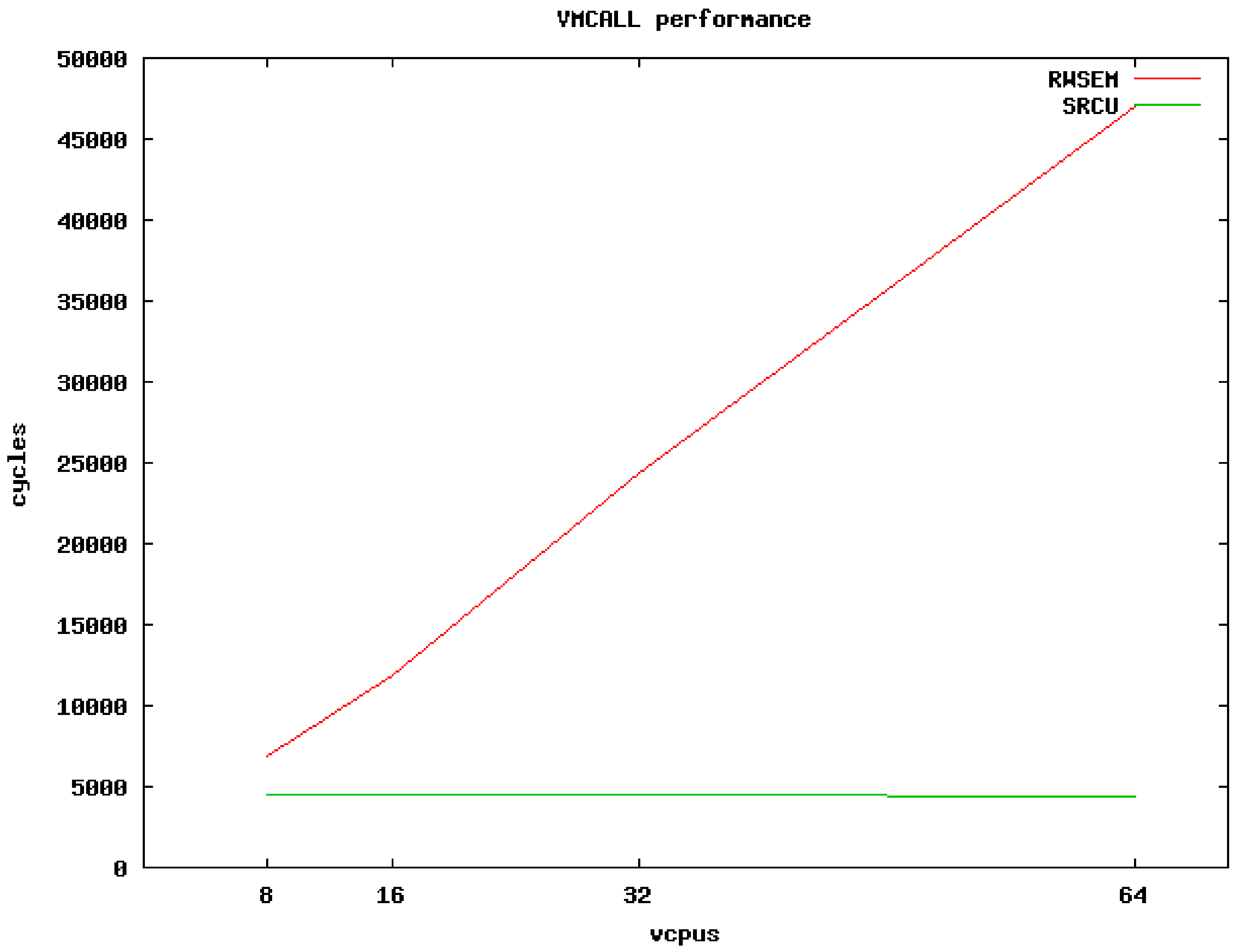
3) Free old structure.

➔ SRCU is similar, but grace period requires verifying potential readers left critical section.

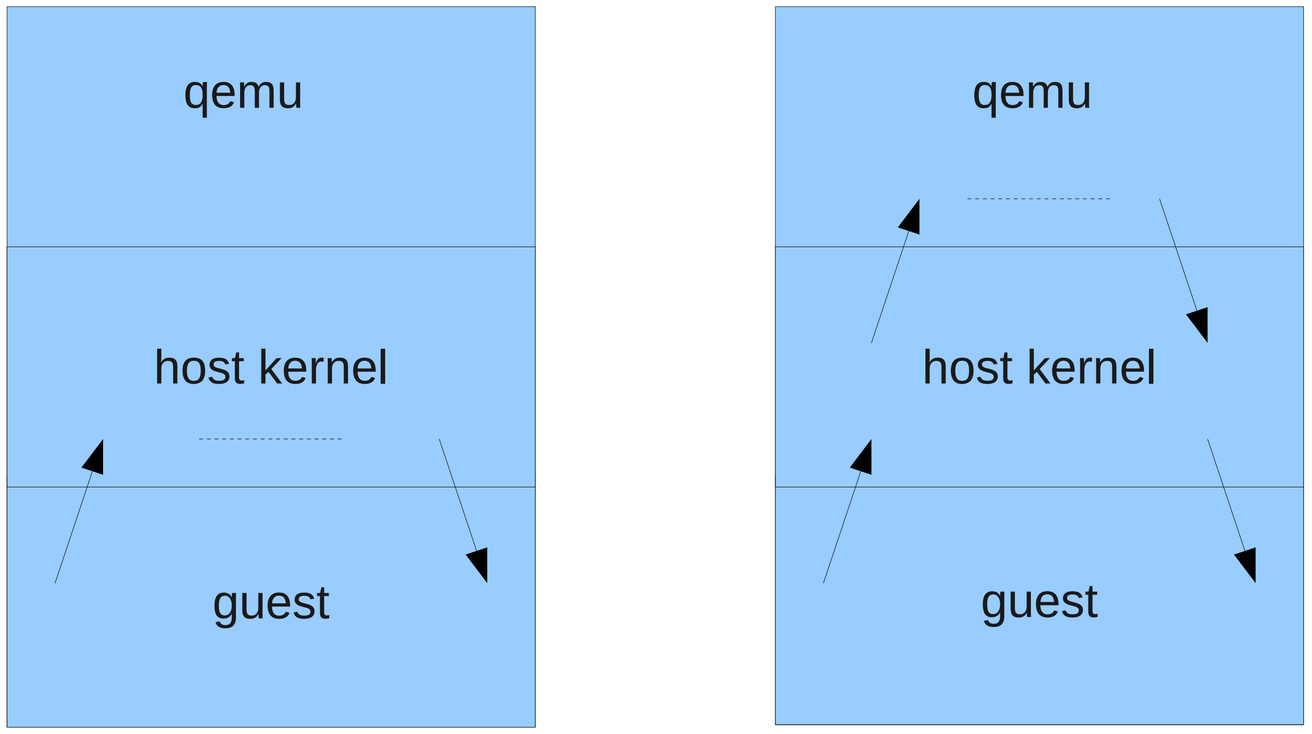
SRCU for slots_lock

- Good candidate for SRCU.
- But grace period is slow, 10ms (SYSLINUX using VGABIOS to draw text).
- synchronize_srcu_expedited, to force context switch.
- IRQ injection path also converted to RCU.

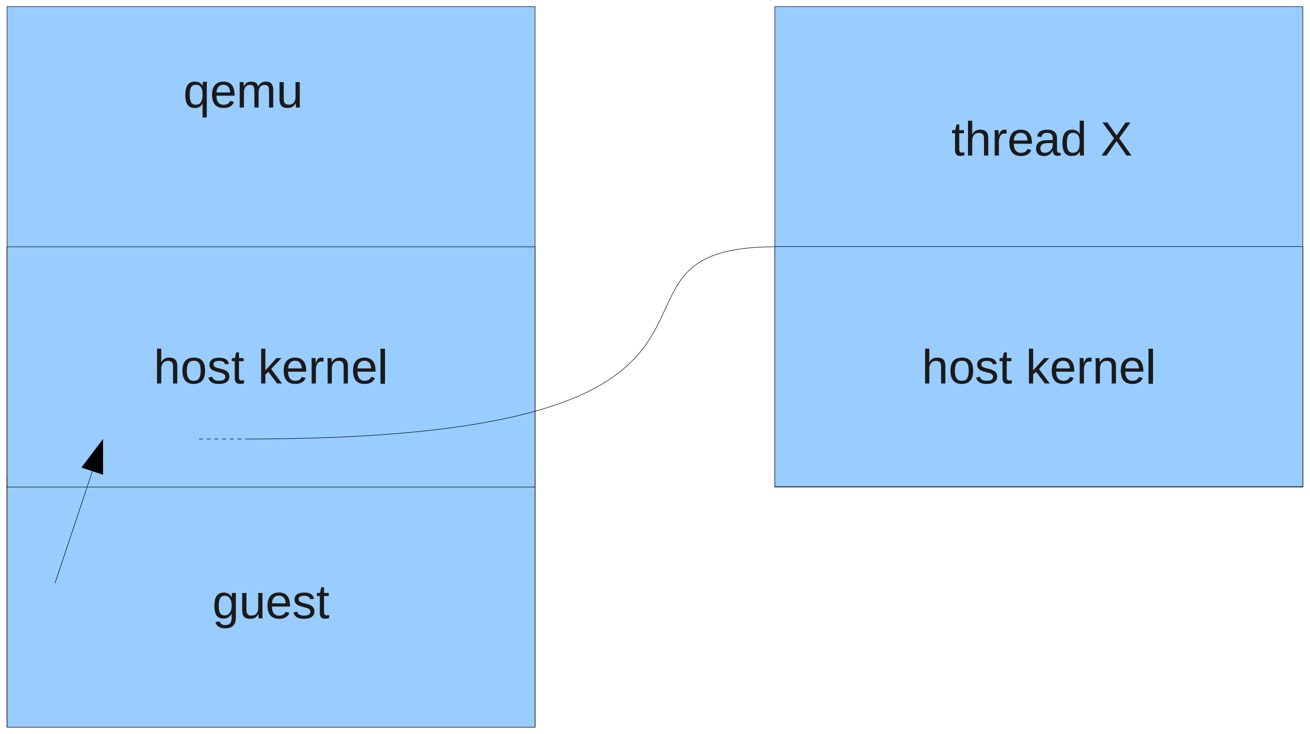
SRCU results



Lightweight / heavy weight exit



Sched notifiers



Heavy weight exit on context switch

Fast syscall and swapgs

- ➔ System calls traditionally implemented using software interrupt on x86.
- ➔ Fast system call use dedicated MSRs:
  - SYSCALL/RET, RIP <- MSR_IA32_LSTAR.
  - SYSENTER/EXIT, MSR_SYSENTER_{CS,EIP,ESP}.
- ➔ swapgs:
  - GS <- MSR_KERNELGSBASE

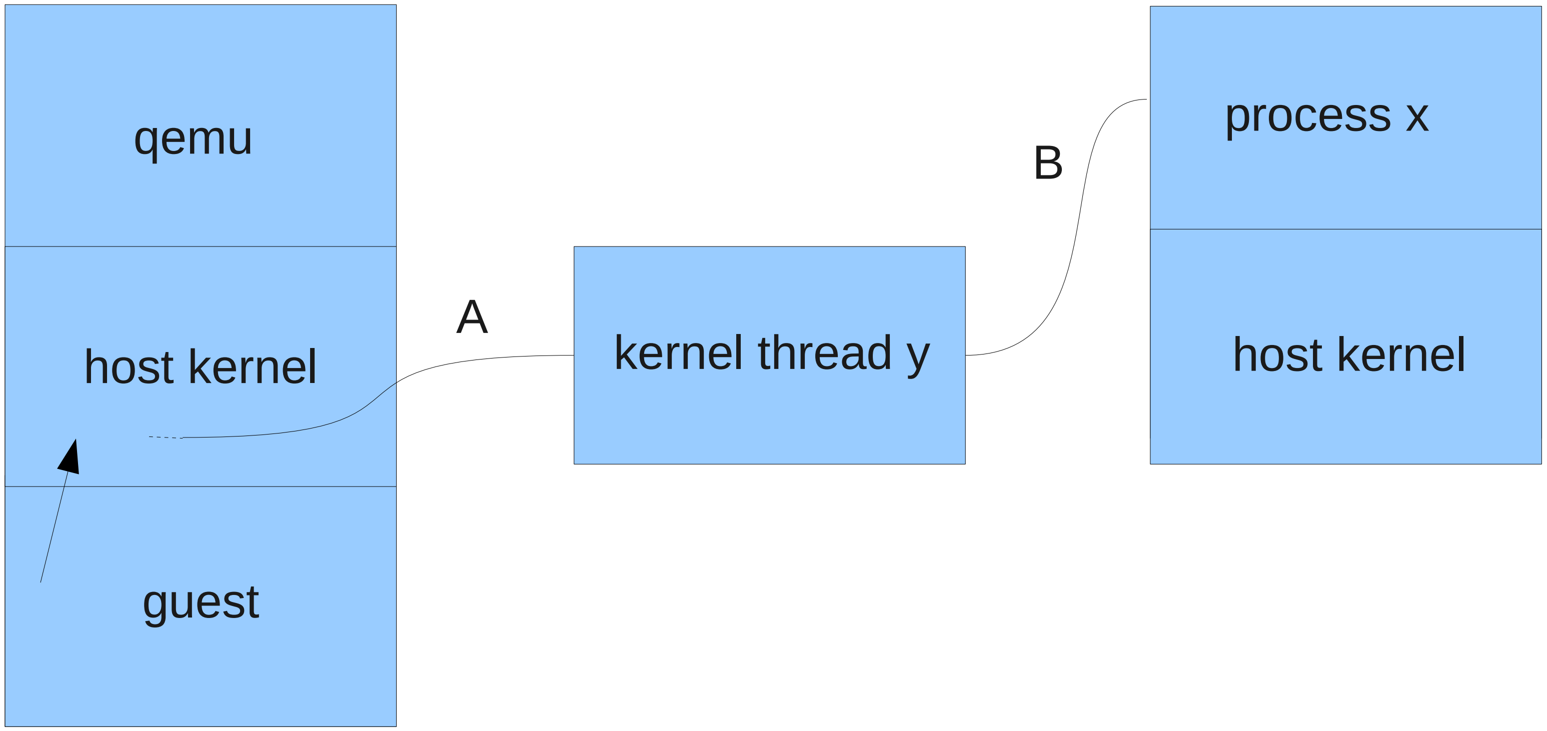
Fast syscall MSRs and KVM

- ➔ Guests have direct access to these MSRs.
- ➔ Not automatically saved/restored by hardware (* except SYSENTER_{CS,ESP,EIP} on VMX).
- ➔ On context switch or return to userspace, KVM must restore host values.

Last minute MSR restoration

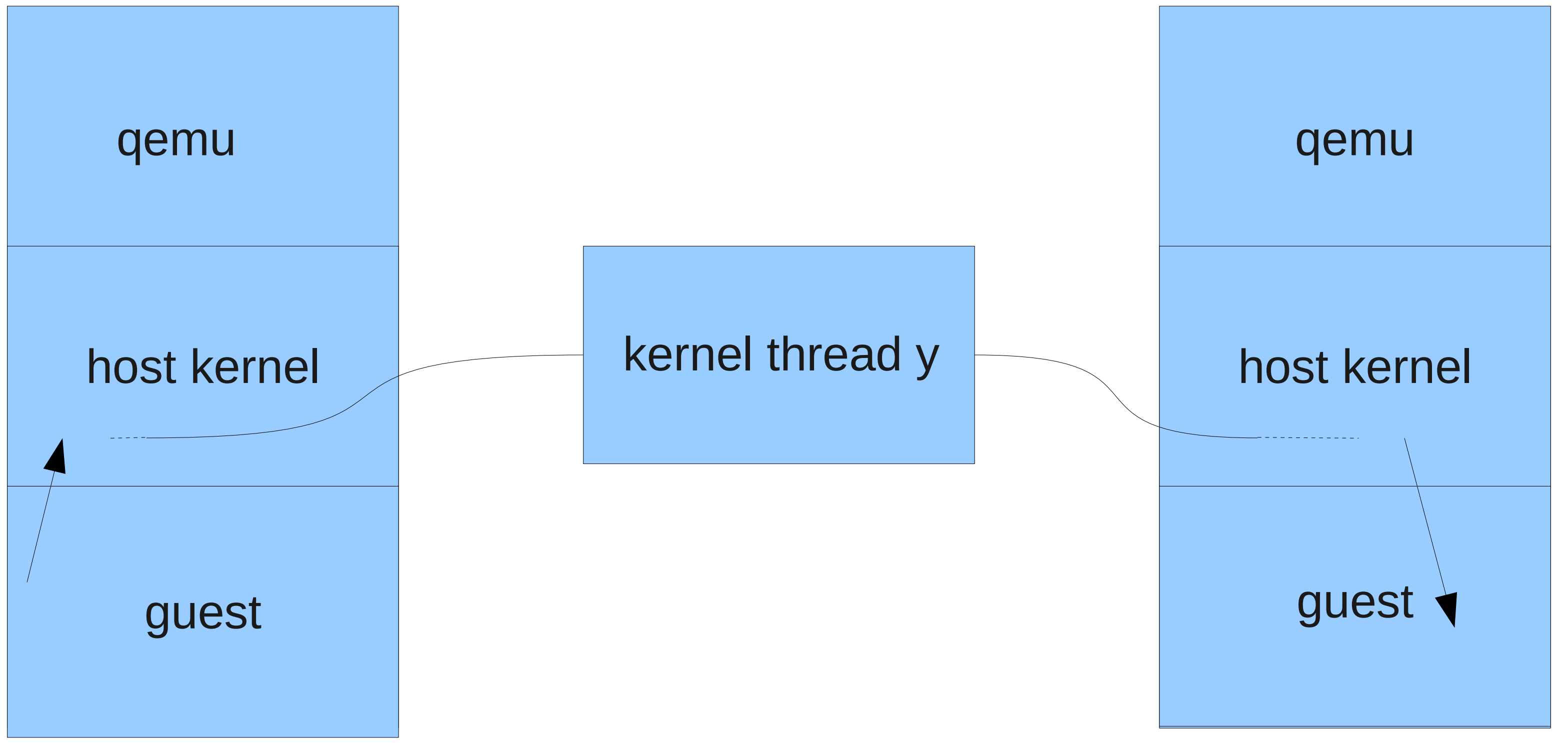
- ➔ Host kernel does not make use of such MSRs.
- ➔ User return notifiers allow distinction between kernel and userspace context switch.

User return notifiers



MSR restoration moved from point A to point B

User return notifiers



No need to save/restore MSRs on guest -> kernel thread -> guest switch

Last minute MSR restoration

- Improves guest -> idle thread -> guest switches by about 2000 cycles.
- Avoids save/restore when guest/host values are equal.

X86's FPU

- Set of registers and instructions for operations on floating point numbers.
- FPU state: 8 data registers and 7 control registers.

X86's lazy FPU switching

- Expensive on every context switch / not always used.
- CR0.TS (Task Switched) allows FPU state to be saved/restored on demand. If set, FPU instructions generate #NM exception.
- If a given task used the FPU during its time slice, OS can save FPU state on context switch.

KVM FPU handling

- If a vcpus CR0.TS flag is set, KVM intercepts #NM exception, and enters guest with host FPU state.
- When guest uses FPU, #NM exception is triggered, handler in KVM loads guest state and disables #NM interception.
- If TS clear, guest FPU loaded unconditionally.
- Host FPU reloaded on heavy exit.
- Unconditional CR0.TS trap, suboptimal.

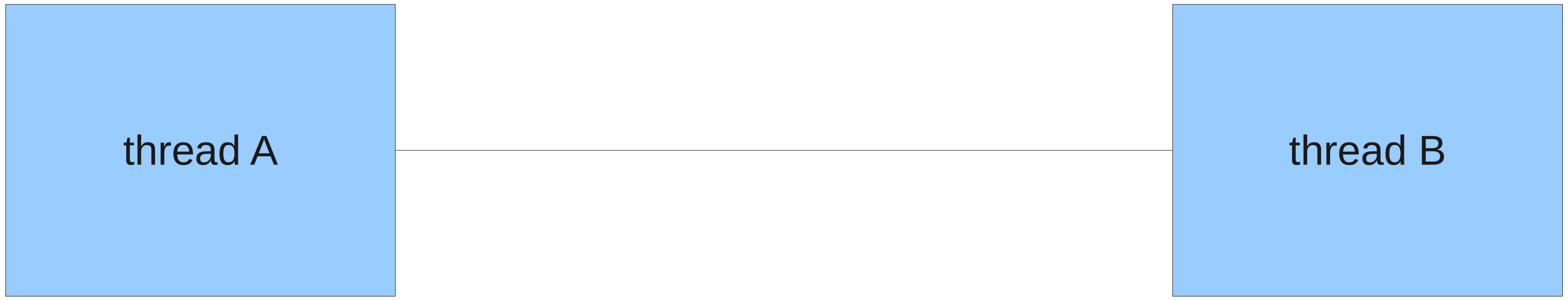
Direct guest access to CR0.TS

- If guest FPU is loaded, CR0.TS value is of no interest.
- Allow direct CR0.TS access.
- Task switching between processes that use FPU in guest require no exit.

Direct CR0.TS numbers

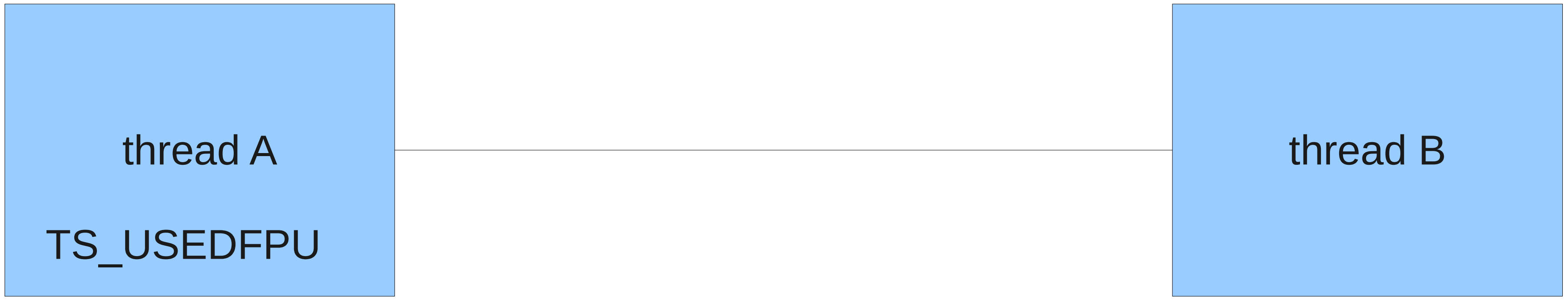
- fbench modified to context switch.
- 4 simultaneous processes on 1 vcpu guest.
- 200k vs 100k exits per second.

Linux lazy FPU



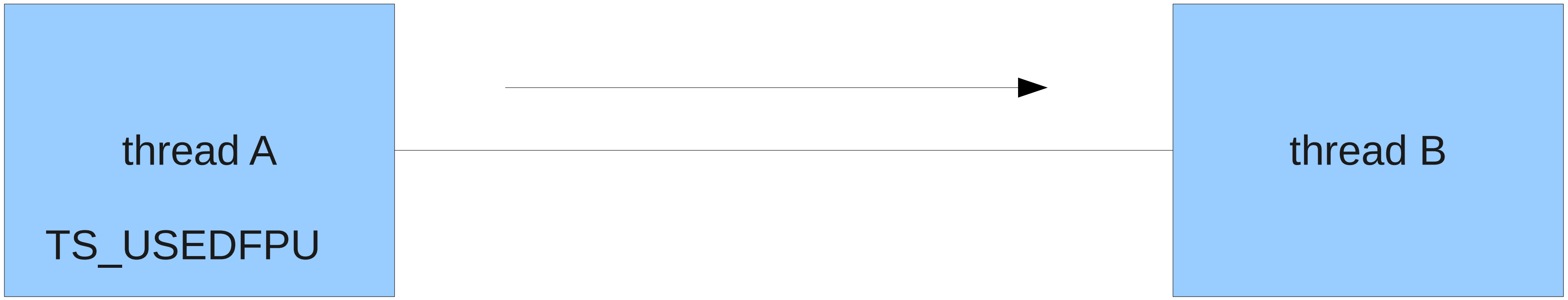
1) thread A issues FPU instruction

Linux lazy FPU



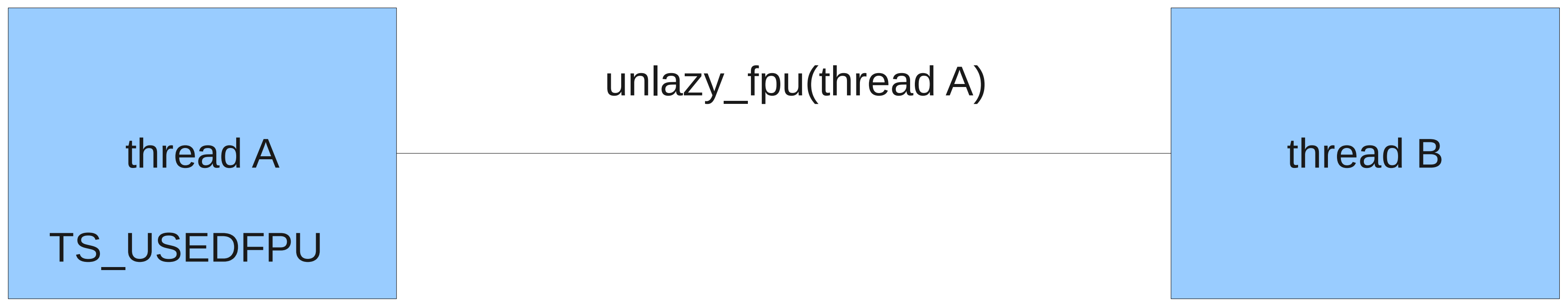
1) thread A issues FPU instruction
2) #NM exception triggers, handler loads task FPU state from memory to registers, sets task->status TS_USEDFPU bit

Linux lazy FPU



1) thread A issues FPU instruction
2) #NM exception triggers, handler loads task FPU state from memory to registers, sets task->status TS_USEDFPU bit
3) scheduler begins switch from thread A to thread B

Linux lazy FPU



1) thread A issues FPU instruction
2) #NM exception triggers, handler loads task FPU state from memory to registers, sets task->status TS_USEDFPU bit
3) scheduler begins switch from thread A to thread B
4) unlazy_fpu(thread A) sees TS_USEDFPU bit, saves FPU registers to thread A state.

Linux lazy FPU for qemu -> guest switch

- Unconditional saving of hosts FPU state on guest FPU load (and restore on heavy exit / preemption).
- Switch to unlazy_fpu.
- Host FPU saved/restored on demand, only when qemu thread has made use of FPU.

Optimization TODO's

- In-kernel device search
- Instruction emulation
- What else can be shared??

Questions / comments ?