Linux Virt Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

CHAPTER

ONE

KVM

1.1 The Definitive KVM (Kernel-based Virtual Machine) API Documentation

1.1.1 1. General description

The kvm API is a set of ioctls that are issued to control various aspects of a virtual machine. The ioctls belong to the following classes:

- System ioctls: These query and set global attributes which affect the whole kvm subsystem. In addition a system ioctl is used to create virtual machines.
- VM ioctls: These query and set attributes that affect an entire virtual machine, for example memory layout. In addition a VM ioctl is used to create virtual cpus (vcpus) and devices.

  VM ioctls must be issued from the same process (address space) that was used to create the VM.
- vcpu ioctls: These query and set attributes that control the operation of a single virtual cpu.

  vcpu ioctls should be issued from the same thread that was used to create the vcpu, except for asynchronous vcpu ioctl that are marked as such in the documentation. Otherwise, the first ioctl after switching threads could see a performance impact.
- device ioctls: These query and set attributes that control the operation of a single device.

  device ioctls must be issued from the same process (address space) that was used to create the VM.

1.1.2 2. File descriptors

The kvm API is centered around file descriptors. An initial open("/dev/kvm") obtains a handle to the kvm subsystem; this handle can be used to issue system ioctls. A KVM_CREATE_VM ioctl on this handle will create a VM file descriptor which can be used to issue VM ioctls. A KVM_CREATE_VCPU or KVM_CREATE_DEVICE ioctl on a VM fd will create a virtual cpu or device and return a file descriptor pointing to the new resource. Finally, ioctls on a vcpu or device fd can be used to control the vcpu or device. For vcpus, this includes the important task of actually running guest code.

In general file descriptors can be migrated among processes by means of fork() and the SCM_RIGHTS facility of unix domain socket. These kinds of tricks are explicitly not supported

by kvm. While they will not cause harm to the host, their actual behavior is not guaranteed by the API. See ”General description” for details on the ioctl usage model that is supported by KVM.

It is important to note that although VM ioctls may only be issued from the process that created the VM, a VM's lifecycle is associated with its file descriptor, not its creator (process). In other words, the VM and its resources, *including the associated address space*, are not freed until the last reference to the VM's file descriptor has been released. For example, if fork() is issued after ioctl(KVM_CREATE_VM), the VM will not be freed until both the parent (original) process and its child have put their references to the VM's file descriptor.

Because a VM's resources are not freed until the last reference to its file descriptor is released, creating additional references to a VM via fork(), dup(), etc... without careful consideration is strongly discouraged and may have unwanted side effects, e.g. memory allocated by and on behalf of the VM's process may not be freed/unaccounted when the VM is shut down.

1.1.3 3. Extensions

As of Linux 2.6.22, the KVM ABI has been stabilized: no backward incompatible change are allowed. However, there is an extension facility that allows backward-compatible extensions to the API to be queried and used.

The extension mechanism is not based on the Linux version number. Instead, kvm defines extension identifiers and a facility to query whether a particular extension identifier is available. If it is, a set of ioctls is available for application use.

1.1.4 4. API description

This section describes ioctls that can be used to control kvm guests. For each ioctl, the following information is provided along with a description:

Capability:
: which KVM extension provides this ioctl. Can be 'basic', which means that is will be provided by any kernel that supports API version 12 (see section 4.1), a KVM_CAP_xyz constant, which means availability needs to be checked with KVM_CHECK_EXTENSION (see section 4.4), or 'none' which means that while not all kernels support this ioctl, there's no capability bit to check its availability: for kernels that don't support the ioctl, the ioctl returns -ENOTTY.

Architectures:
: which instruction set architectures provide this ioctl. x86 includes both i386 and x86_64.

Type:
: system, vm, or vcpu.

Parameters:
: what parameters are accepted by the ioctl.

Returns:
: the return value. General error numbers (EBADF, ENOMEM, EINVAL) are not detailed, but errors with specific meanings are.

4.1 KVM_GET_API_VERSION

Capability
basic

Architectures
all

Type
system ioctl

Parameters
none

Returns
the constant KVM_API_VERSION (=12)

This identifies the API version as the stable kvm API. It is not expected that this number will change. However, Linux 2.6.20 and 2.6.21 report earlier versions; these are not documented and not supported. Applications should refuse to run if KVM_GET_API_VERSION returns a value other than 12. If this check passes, all ioctls described as 'basic' will be available.

4.2 KVM_CREATE_VM

Capability
basic

Architectures
all

Type
system ioctl

Parameters
machine type identifier (KVM_VM_*)

Returns
a VM fd that can be used to control the new virtual machine.

The new VM has no virtual cpus and no memory. You probably want to use 0 as machine type.

X86:

Supported X86 VM types can be queried via KVM_CAP_VM_TYPES.

S390:

In order to create user controlled virtual machines on S390, check KVM_CAP_S390_UCONTROL and use the flag KVM_VM_S390_UCONTROL as privileged user (CAP_SYS_ADMIN).

MIPS:

To use hardware assisted virtualization on MIPS (VZ ASE) rather than the default trap & emulate implementation (which changes the virtual memory layout to fit in user mode), check KVM_CAP_MIPS_VZ and use the flag KVM_VM_MIPS_VZ.

ARM64:

On arm64, the physical address size for a VM (IPA Size limit) is limited to 40bits by default. The limit can be configured if the host supports the extension KVM_CAP_ARM_VM_IPA_SIZE. When supported, use KVM_VM_TYPE_ARM_IPA_SIZE(IPA_Bits) to set the size in the machine type identifier, where IPA_Bits is the maximum width of any physical address used by the VM. The IPA_Bits is encoded in bits[7-0] of the machine type identifier.

e.g, to configure a guest to use 48bit physical address size:

```
vm_fd = ioctl(dev_fd, KVM_CREATE_VM, KVM_VM_TYPE_ARM_IPA_SIZE(48));
```

The requested size (IPA_Bits) must be:

0	Implies default size, 40bits (for backward compatibility)
N	Implies N bits, where N is a positive integer such that, 32 <= N <= Host_IPA_Limit

Host_IPA_Limit is the maximum possible value for IPA_Bits on the host and is dependent on the CPU capability and the kernel configuration. The limit can be retrieved using KVM_CAP_ARM_VM_IPA_SIZE of the KVM_CHECK_EXTENSION ioctl() at run-time.

Creation of the VM will fail if the requested IPA size (whether it is implicit or explicit) is unsupported on the host.

Please note that configuring the IPA size does not affect the capability exposed by the guest CPUs in ID_AA64MMFR0_EL1[PARange]. It only affects size of the address translated by the stage2 level (guest physical to host physical address translations).

4.3 KVM_GET_MSR_INDEX_LIST, KVM_GET_MSR_FEATURE_INDEX_LIST

Capability
basic, KVM_CAP_GET_MSR_FEATURES for KVM_GET_MSR_FEATURE_INDEX_LIST

Architectures
x86

Type
system ioctl

Parameters
struct kvm_msr_list (in/out)

Returns
0 on success; -1 on error

Errors:

EFAULT	the msr index list cannot be read from or written to
E2BIG	the msr index list is too big to fit in the array specified by the user.

```
struct kvm_msr_list {
      __u32 nmsrs; /* number of msrs in entries */
      __u32 indices[0];
};
```

The user fills in the size of the indices array in nmsrs, and in return kvm adjusts nmsrs to reflect the actual number of msrs and fills in the indices array with their numbers.

KVM_GET_MSR_INDEX_LIST returns the guest msrs that are supported. The list varies by kvm version and host processor, but does not change otherwise.

Note: if kvm indicates supports MCE (KVM_CAP_MCE), then the MCE bank MSRs are not returned in the MSR list, as different vcpus can have a different number of banks, as set via the KVM_X86_SETUP_MCE ioctl.

KVM_GET_MSR_FEATURE_INDEX_LIST returns the list of MSRs that can be passed to the KVM_GET_MSRS system ioctl. This lets userspace probe host capabilities and processor features that are exposed via MSRs (e.g., VMX capabilities). This list also varies by kvm version and host processor, but does not change otherwise.

4.4 KVM_CHECK_EXTENSION

Capability
basic, KVM_CAP_CHECK_EXTENSION_VM for vm ioctl

Architectures
all

Type
system ioctl, vm ioctl

Parameters
extension identifier (KVM_CAP_*)

Returns
0 if unsupported; 1 (or some other positive integer) if supported

The API allows the application to query about extensions to the core kvm API. Userspace passes an extension identifier (an integer) and receives an integer that describes the extension availability. Generally 0 means no and 1 means yes, but some extensions may report additional information in the integer return value.

Based on their initialization different VMs may have different capabilities. It is thus encouraged to use the vm ioctl to query for capabilities (available with KVM_CAP_CHECK_EXTENSION_VM on the vm fd)

4.5 KVM_GET_VCPU_MMAP_SIZE

Capability
basic

Architectures
all

Type
system ioctl

Parameters
none

Returns
size of vcpu mmap area, in bytes

The KVM_RUN ioctl (cf.) communicates with userspace via a shared memory region. This ioctl returns the size of that region. See the KVM_RUN documentation for details.

Besides the size of the KVM_RUN communication region, other areas of the VCPU file descriptor can be mmap-ed, including:

- if KVM_CAP_COALESCED_MMIO is available, a page at KVM_COALESCED_MMIO_PAGE_OFFSET * PAGE_SIZE; for historical reasons, this page is included in the result of KVM_GET_VCPU_MMAP_SIZE. KVM_CAP_COALESCED_MMIO is not documented yet.
- if KVM_CAP_DIRTY_LOG_RING is available, a number of pages at KVM_DIRTY_LOG_PAGE_OFFSET * PAGE_SIZE. For more information on KVM_CAP_DIRTY_LOG_RING, see section 8.3.

4.7 KVM_CREATE_VCPU

Capability
basic

Architectures
all

Type
vm ioctl

Parameters
vcpu id (apic id on x86)

Returns
vcpu fd on success, -1 on error

This API adds a vcpu to a virtual machine. No more than max_vcpus may be added. The vcpu id is an integer in the range [0, max_vcpu_id).

The recommended max_vcpus value can be retrieved using the KVM_CAP_NR_VCPUS of the KVM_CHECK_EXTENSION ioctl() at run-time. The maximum possible value for max_vcpus can be retrieved using the KVM_CAP_MAX_VCPUS of the KVM_CHECK_EXTENSION ioctl() at run-time.

If the KVM_CAP_NR_VCPUS does not exist, you should assume that max_vcpus is 4 cpus max. If the KVM_CAP_MAX_VCPUS does not exist, you should assume that max_vcpus is same as the value returned from KVM_CAP_NR_VCPUS.

The maximum possible value for max_vcpu_id can be retrieved using the KVM_CAP_MAX_VCPU_ID of the KVM_CHECK_EXTENSION ioctl() at run-time.

If the KVM_CAP_MAX_VCPU_ID does not exist, you should assume that max_vcpu_id is the same as the value returned from KVM_CAP_MAX_VCPUS.

On powerpc using book3s_hv mode, the vcpus are mapped onto virtual threads in one or more virtual CPU cores. (This is because the hardware requires all the hardware threads in a CPU core to be in the same partition.) The KVM_CAP_PPC_SMT capability indicates the number of vcpus per virtual core (vcore). The vcore id is obtained by dividing the vcpu id by the number of vcpus per vcore. The vcpus in a given vcore will always be in the same physical core as each other (though that might be a different physical core from time to time). Userspace can control the threading (SMT) mode of the guest by its allocation of vcpu ids. For example, if userspace wants single-threaded guest vcpus, it should make all vcpu ids be a multiple of the number of vcpus per vcore.

For virtual cpus that have been created with S390 user controlled virtual machines, the resulting vcpu fd can be memory mapped at page offset KVM_S390_SIE_PAGE_OFFSET in order to obtain a memory map of the virtual cpu's hardware control block.

4.8 KVM_GET_DIRTY_LOG (vm ioctl)

Capability
basic

Architectures
all

Type
vm ioctl

Parameters
struct kvm_dirty_log (in/out)

Returns
0 on success, -1 on error

```
/* for KVM_GET_DIRTY_LOG */
struct kvm_dirty_log {
      __u32 slot;
      __u32 padding;
```

```
      union {
              void __user *dirty_bitmap; /* one bit per page */
              __u64 padding;
      };
};
```

Given a memory slot, return a bitmap containing any pages dirtied since the last call to this ioctl. Bit 0 is the first page in the memory slot. Ensure the entire structure is cleared to avoid padding issues.

If KVM_CAP_MULTI_ADDRESS_SPACE is available, bits 16-31 of slot field specifies the address space for which you want to return the dirty bitmap. See KVM_SET_USER_MEMORY_REGION for details on the usage of slot field.

The bits in the dirty bitmap are cleared before the ioctl returns, unless KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2 is enabled. For more information, see the description of the capability.

Note that the Xen shared info page, if configured, shall always be assumed to be dirty. KVM will not explicitly mark it such.

4.10 KVM_RUN

Capability
basic

Architectures
all

Type
vcpu ioctl

Parameters
none

Returns
0 on success, -1 on error

Errors:

EINTR	an unmasked signal is pending
ENOEXEC	the vcpu hasn't been initialized or the guest tried to execute instructions from device memory (arm64)
ENOSYS	data abort outside memslots with no syndrome info and KVM_CAP_ARM_NISV_TO_USER not enabled (arm64)
EPERM	SVE feature set but not finalized (arm64)

This ioctl is used to run a guest virtual cpu. While there are no explicit parameters, there is an implicit parameter block that can be obtained by mmap()ing the vcpu fd at offset 0, with the size given by KVM_GET_VCPU_MMAP_SIZE. The parameter block is formatted as a 'struct kvm_run' (see below).

4.11 KVM_GET_REGS

Capability
basic

Architectures
all except arm64

Type
vcpu ioctl

Parameters
struct kvm_regs (out)

Returns
0 on success, -1 on error

Reads the general purpose registers from the vcpu.

```
/* x86 */
struct kvm_regs {
      /* out (KVM_GET_REGS) / in (KVM_SET_REGS) */
      __u64 rax, rbx, rcx, rdx;
      __u64 rsi, rdi, rsp, rbp;
      __u64 r8,  r9,  r10, r11;
      __u64 r12, r13, r14, r15;
      __u64 rip, rflags;
};

/* mips */
struct kvm_regs {
      /* out (KVM_GET_REGS) / in (KVM_SET_REGS) */
      __u64 gpr[32];
      __u64 hi;
      __u64 lo;
      __u64 pc;
};

/* LoongArch */
struct kvm_regs {
      /* out (KVM_GET_REGS) / in (KVM_SET_REGS) */
      unsigned long gpr[32];
      unsigned long pc;
};
```

4.12 KVM_SET_REGS

Capability
basic

Architectures
all except arm64

Type
vcpu ioctl

Parameters
struct kvm_regs (in)

Returns
0 on success, -1 on error

Writes the general purpose registers into the vcpu.

See KVM_GET_REGS for the data structure.

4.13 KVM_GET_SREGS

Capability
basic

Architectures
x86, ppc

Type
vcpu ioctl

Parameters
struct kvm_sregs (out)

Returns
0 on success, -1 on error

Reads special registers from the vcpu.

```
/* x86 */
struct kvm_sregs {
      struct kvm_segment cs, ds, es, fs, gs, ss;
      struct kvm_segment tr, ldt;
      struct kvm_dtable gdt, idt;
      __u64 cr0, cr2, cr3, cr4, cr8;
      __u64 efer;
      __u64 apic_base;
      __u64 interrupt_bitmap[(KVM_NR_INTERRUPTS + 63) / 64];
};

/* ppc -- see arch/powerpc/include/uapi/asm/kvm.h */
```

interrupt_bitmap is a bitmap of pending external interrupts. At most one bit may be set. This interrupt has been acknowledged by the APIC but not yet injected into the cpu core.

4.14 KVM_SET_SREGS

Capability
basic

Architectures
x86, ppc

Type
vcpu ioctl

Parameters
struct kvm_sregs (in)

Returns
0 on success, -1 on error

Writes special registers into the vcpu. See KVM_GET_SREGS for the data structures.

4.15 KVM_TRANSLATE

Capability
basic

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_translation (in/out)

Returns
0 on success, -1 on error

Translates a virtual address according to the vcpu's current address translation mode.

```
struct kvm_translation {
      /* in */
      __u64 linear_address;

      /* out */
      __u64 physical_address;
      __u8  valid;
      __u8  writeable;
      __u8  usermode;
      __u8  pad[5];
};
```

4.16 KVM_INTERRUPT

Capability
basic

Architectures
x86, ppc, mips, riscv, loongarch

Type
vcpu ioctl

Parameters
struct kvm_interrupt (in)

Returns
0 on success, negative on failure.

Queues a hardware interrupt vector to be injected.

```
/* for KVM_INTERRUPT */
struct kvm_interrupt {
      /* in */
      __u32 irq;
};
```

X86:

Returns

0	on success,
-EEXIST	if an interrupt is already enqueued
-EINVAL	the irq number is invalid
-ENXIO	if the PIC is in the kernel
-EFAULT	if the pointer is invalid

Note 'irq' is an interrupt vector, not an interrupt pin or line. This ioctl is useful if the in-kernel PIC is not used.

PPC:

Queues an external interrupt to be injected. This ioctl is overloaded with 3 different irq values:

a) KVM_INTERRUPT_SET

This injects an edge type external interrupt into the guest once it's ready to receive interrupts. When injected, the interrupt is done.

b) KVM_INTERRUPT_UNSET

This unsets any pending interrupt.

Only available with KVM_CAP_PPC_UNSET_IRQ.

c) KVM_INTERRUPT_SET_LEVEL

This injects a level type external interrupt into the guest context. The interrupt stays pending until a specific ioctl with KVM_INTERRUPT_UNSET is triggered.

Only available with KVM_CAP_PPC_IRQ_LEVEL.

Note that any value for 'irq' other than the ones stated above is invalid and incurs unexpected behavior.

This is an asynchronous vcpu ioctl and can be invoked from any thread.

MIPS:

Queues an external interrupt to be injected into the virtual CPU. A negative interrupt number dequeues the interrupt.

This is an asynchronous vcpu ioctl and can be invoked from any thread.

RISC-V:

Queues an external interrupt to be injected into the virtual CPU. This ioctl is overloaded with 2 different irq values:

a) KVM_INTERRUPT_SET

This sets external interrupt for a virtual CPU and it will receive once it is ready.

b) KVM_INTERRUPT_UNSET

This clears pending external interrupt for a virtual CPU.

This is an asynchronous vcpu ioctl and can be invoked from any thread.

LOONGARCH:

Queues an external interrupt to be injected into the virtual CPU. A negative interrupt number dequeues the interrupt.

This is an asynchronous vcpu ioctl and can be invoked from any thread.

4.18 KVM_GET_MSRS

Capability
basic (vcpu), KVM_CAP_GET_MSR_FEATURES (system)

Architectures
x86

Type
system ioctl, vcpu ioctl

Parameters
struct kvm_msrs (in/out)

Returns
number of msrs successfully returned; -1 on error

When used as a system ioctl: Reads the values of MSR-based features that are available for the VM. This is similar to KVM_GET_SUPPORTED_CPUID, but it returns MSR indices and values. The list of msr-based features can be obtained using KVM_GET_MSR_FEATURE_INDEX_LIST in a system ioctl.

When used as a vcpu ioctl: Reads model-specific registers from the vcpu. Supported msr indices can be obtained using KVM_GET_MSR_INDEX_LIST in a system ioctl.

```
struct kvm_msrs {
      __u32 nmsrs; /* number of msrs in entries */
      __u32 pad;

      struct kvm_msr_entry entries[0];
};

struct kvm_msr_entry {
      __u32 index;
      __u32 reserved;
      __u64 data;
};
```

Application code should set the 'nmsrs' member (which indicates the size of the entries array) and the 'index' member of each array entry. kvm will fill in the 'data' member.

4.19 KVM_SET_MSRS

Capability
basic

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_msrs (in)

Returns
number of msrs successfully set (see below), -1 on error

Writes model-specific registers to the vcpu. See KVM_GET_MSRS for the data structures.

Application code should set the 'nmsrs' member (which indicates the size of the entries array), and the 'index' and 'data' members of each array entry.

It tries to set the MSRs in array entries[] one by one. If setting an MSR fails, e.g., due to setting reserved bits, the MSR isn't supported/emulated by KVM, etc..., it stops processing the MSR list and returns the number of MSRs that have been set successfully.

4.20 KVM_SET_CPUID

Capability
basic

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_cpuid (in)

Returns
0 on success, -1 on error

Defines the vcpu responses to the cpuid instruction. Applications should use the KVM_SET_CPUID2 ioctl if available.

Caveat emptor:

- If this IOCTL fails, KVM gives no guarantees that previous valid CPUID configuration (if there is) is not corrupted. Userspace can get a copy of the resulting CPUID configuration through KVM_GET_CPUID2 in case.
- Using KVM_SET_CPUID{,2} after KVM_RUN, i.e. changing the guest vCPU model after running the guest, may cause guest instability.
- Using heterogeneous CPUID configurations, modulo APIC IDs, topology, etc... may cause guest instability.

```
struct kvm_cpuid_entry {
      __u32 function;
      __u32 eax;
      __u32 ebx;
      __u32 ecx;
      __u32 edx;
      __u32 padding;
};

/* for KVM_SET_CPUID */
struct kvm_cpuid {
      __u32 nent;
      __u32 padding;
      struct kvm_cpuid_entry entries[0];
};
```

4.21 KVM_SET_SIGNAL_MASK

Capability
basic

Architectures
all

Type
vcpu ioctl

Parameters
struct kvm_signal_mask (in)

Returns
0 on success, -1 on error

Defines which signals are blocked during execution of KVM_RUN. This signal mask temporarily overrides the threads signal mask. Any unblocked signal received (except SIGKILL and SIGSTOP, which retain their traditional behaviour) will cause KVM_RUN to return with -EINTR.

Note the signal will only be delivered if not blocked by the original signal mask.

```
/* for KVM_SET_SIGNAL_MASK */
struct kvm_signal_mask {
      __u32 len;
      __u8  sigset[0];
};
```

4.22 KVM_GET_FPU

Capability
basic

Architectures
x86, loongarch

Type
vcpu ioctl

Parameters
struct kvm_fpu (out)

Returns
0 on success, -1 on error

Reads the floating point state from the vcpu.

```
/* x86: for KVM_GET_FPU and KVM_SET_FPU */
struct kvm_fpu {
      __u8  fpr[8][16];
      __u16 fcw;
      __u16 fsw;
      __u8  ftwx;  /* in fxsave format */
      __u8  pad1;
      __u16 last_opcode;
```

```
      __u64 last_ip;
      __u64 last_dp;
      __u8  xmm[16][16];
      __u32 mxcsr;
      __u32 pad2;
};

/* LoongArch: for KVM_GET_FPU and KVM_SET_FPU */
struct kvm_fpu {
      __u32 fcsr;
      __u64 fcc;
      struct kvm_fpureg {
              __u64 val64[4];
      }fpr[32];
};
```

4.23 KVM_SET_FPU

Capability
basic

Architectures
x86, loongarch

Type
vcpu ioctl

Parameters
struct kvm_fpu (in)

Returns
0 on success, -1 on error

Writes the floating point state to the vcpu.

```
/* x86: for KVM_GET_FPU and KVM_SET_FPU */
struct kvm_fpu {
      __u8  fpr[8][16];
      __u16 fcw;
      __u16 fsw;
      __u8  ftwx;  /* in fxsave format */
      __u8  pad1;
      __u16 last_opcode;
      __u64 last_ip;
      __u64 last_dp;
      __u8  xmm[16][16];
      __u32 mxcsr;
      __u32 pad2;
};

/* LoongArch: for KVM_GET_FPU and KVM_SET_FPU */
struct kvm_fpu {
```

```
      __u32 fcsr;
      __u64 fcc;
      struct kvm_fpureg {
              __u64 val64[4];
      }fpr[32];
};
```

4.24 KVM_CREATE_IRQCHIP

Capability
KVM_CAP_IRQCHIP, KVM_CAP_S390_IRQCHIP (s390)

Architectures
x86, arm64, s390

Type
vm ioctl

Parameters
none

Returns
0 on success, -1 on error

Creates an interrupt controller model in the kernel. On x86, creates a virtual ioapic, a virtual PIC (two PICs, nested), and sets up future vcpus to have a local APIC. IRQ routing for GSIs 0-15 is set to both PIC and IOAPIC; GSI 16-23 only go to the IOAPIC. On arm64, a GICv2 is created. Any other GIC versions require the usage of KVM_CREATE_DEVICE, which also supports creating a GICv2. Using KVM_CREATE_DEVICE is preferred over KVM_CREATE_IRQCHIP for GICv2. On s390, a dummy irq routing table is created.

Note that on s390 the KVM_CAP_S390_IRQCHIP vm capability needs to be enabled before KVM_CREATE_IRQCHIP can be used.

4.25 KVM_IRQ_LINE

Capability
KVM_CAP_IRQCHIP

Architectures
x86, arm64

Type
vm ioctl

Parameters
struct kvm_irq_level

Returns
0 on success, -1 on error

Sets the level of a GSI input to the interrupt controller model in the kernel. On some architectures it is required that an interrupt controller model has been previously created with

KVM_CREATE_IRQCHIP. Note that edge-triggered interrupts require the level to be set to 1 and then back to 0.

On real hardware, interrupt pins can be active-low or active-high. This does not matter for the level field of struct kvm_irq_level: 1 always means active (asserted), 0 means inactive (de-asserted).

x86 allows the operating system to program the interrupt polarity (active-low/active-high) for level-triggered interrupts, and KVM used to consider the polarity. However, due to bitrot in the handling of active-low interrupts, the above convention is now valid on x86 too. This is signaled by KVM_CAP_X86_IOAPIC_POLARITY_IGNORED. Userspace should not present interrupts to the guest as active-low unless this capability is present (or unless it is not using the in-kernel irqchip, of course).

arm64 can signal an interrupt either at the CPU level, or at the in-kernel irqchip (GIC), and for in-kernel irqchip can tell the GIC to use PPIs designated for specific cpus. The irq field is interpreted like this:

```
bits:  |  31 ... 28  | 27 ... 24 | 23  ... 16 | 15 ... 0 |
field: | vcpu2_index | irq_type  | vcpu_index |  irq_id  |
```

The irq_type field has the following values:

- **irq_type[0]:**
  out-of-kernel GIC: irq_id 0 is IRQ, irq_id 1 is FIQ
- **irq_type[1]:**
  in-kernel GIC: SPI, irq_id between 32 and 1019 (incl.) (the vcpu_index field is ignored)
- **irq_type[2]:**
  in-kernel GIC: PPI, irq_id between 16 and 31 (incl.)

(The irq_id field thus corresponds nicely to the IRQ ID in the ARM GIC specs)

In both cases, level is used to assert/deassert the line.

When KVM_CAP_ARM_IRQ_LINE_LAYOUT_2 is supported, the target vcpu is identified as (256 * vcpu2_index + vcpu_index). Otherwise, vcpu2_index must be zero.

Note that on arm64, the KVM_CAP_IRQCHIP capability only conditions injection of interrupts for the in-kernel irqchip. KVM_IRQ_LINE can always be used for a userspace interrupt controller.

```
struct kvm_irq_level {
      union {
              __u32 irq;     /* GSI */
              __s32 status;  /* not used for KVM_IRQ_LEVEL */
      };
      __u32 level;           /* 0 or 1 */
};
```

4.26 KVM_GET_IRQCHIP

Capability
KVM_CAP_IRQCHIP

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_irqchip (in/out)

Returns
0 on success, -1 on error

Reads the state of a kernel interrupt controller created with KVM_CREATE_IRQCHIP into a buffer provided by the caller.

```
struct kvm_irqchip {
      __u32 chip_id;  /* 0 = PIC1, 1 = PIC2, 2 = IOAPIC */
      __u32 pad;
      union {
              char dummy[512];  /* reserving space */
              struct kvm_pic_state pic;
              struct kvm_ioapic_state ioapic;
      } chip;
};
```

4.27 KVM_SET_IRQCHIP

Capability
KVM_CAP_IRQCHIP

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_irqchip (in)

Returns
0 on success, -1 on error

Sets the state of a kernel interrupt controller created with KVM_CREATE_IRQCHIP from a buffer provided by the caller.

```
struct kvm_irqchip {
      __u32 chip_id;  /* 0 = PIC1, 1 = PIC2, 2 = IOAPIC */
      __u32 pad;
      union {
              char dummy[512];  /* reserving space */
```

```
                struct kvm_pic_state pic;
                struct kvm_ioapic_state ioapic;
        } chip;
};
```

4.28 KVM_XEN_HVM_CONFIG

Capability
KVM_CAP_XEN_HVM

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_xen_hvm_config (in)

Returns
0 on success, -1 on error

Sets the MSR that the Xen HVM guest uses to initialize its hypercall page, and provides the starting address and size of the hypercall blobs in userspace. When the guest writes the MSR, kvm copies one page of a blob (32- or 64-bit, depending on the vcpu mode) to guest memory.

```
struct kvm_xen_hvm_config {
        __u32 flags;
        __u32 msr;
        __u64 blob_addr_32;
        __u64 blob_addr_64;
        __u8 blob_size_32;
        __u8 blob_size_64;
        __u8 pad2[30];
};
```

If certain flags are returned from the KVM_CAP_XEN_HVM check, they may be set in the flags field of this ioctl:

The KVM_XEN_HVM_CONFIG_INTERCEPT_HCALL flag requests KVM to generate the contents of the hypercall page automatically; hypercalls will be intercepted and passed to userspace through KVM_EXIT_XEN. In this case, all of the blob size and address fields must be zero.

The KVM_XEN_HVM_CONFIG_EVTCHN_SEND flag indicates to KVM that userspace will always use the KVM_XEN_HVM_EVTCHN_SEND ioctl to deliver event channel interrupts rather than manipulating the guest's shared_info structures directly. This, in turn, may allow KVM to enable features such as intercepting the SCHEDOP_poll hypercall to accelerate PV spinlock operation for the guest. Userspace may still use the ioctl to deliver events if it was advertised, even if userspace does not send this indication that it will always do so

No other flags are currently valid in the struct kvm_xen_hvm_config.

4.29 KVM_GET_CLOCK

Capability
KVM_CAP_ADJUST_CLOCK

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_clock_data (out)

Returns
0 on success, -1 on error

Gets the current timestamp of kvmclock as seen by the current guest. In conjunction with KVM_SET_CLOCK, it is used to ensure monotonicity on scenarios such as migration.

When KVM_CAP_ADJUST_CLOCK is passed to KVM_CHECK_EXTENSION, it returns the set of bits that KVM can return in struct kvm_clock_data's flag member.

The following flags are defined:

KVM_CLOCK_TSC_STABLE
If set, the returned value is the exact kvmclock value seen by all VCPUs at the instant when KVM_GET_CLOCK was called. If clear, the returned value is simply CLOCK_MONOTONIC plus a constant offset; the offset can be modified with KVM_SET_CLOCK. KVM will try to make all VCPUs follow this clock, but the exact value read by each VCPU could differ, because the host TSC is not stable.

KVM_CLOCK_REALTIME
If set, the *realtime* field in the kvm_clock_data structure is populated with the value of the host's real time clocksource at the instant when KVM_GET_CLOCK was called. If clear, the *realtime* field does not contain a value.

KVM_CLOCK_HOST_TSC
If set, the *host_tsc* field in the kvm_clock_data structure is populated with the value of the host's timestamp counter (TSC) at the instant when KVM_GET_CLOCK was called. If clear, the *host_tsc* field does not contain a value.

```
struct kvm_clock_data {
      __u64 clock;  /* kvmclock current value */
      __u32 flags;
      __u32 pad0;
      __u64 realtime;
      __u64 host_tsc;
      __u32 pad[4];
};
```

4.30 KVM_SET_CLOCK

Capability
KVM_CAP_ADJUST_CLOCK

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_clock_data (in)

Returns
0 on success, -1 on error

Sets the current timestamp of kvmclock to the value specified in its parameter. In conjunction with KVM_GET_CLOCK, it is used to ensure monotonicity on scenarios such as migration.

The following flags can be passed:

KVM_CLOCK_REALTIME
If set, KVM will compare the value of the *realtime* field with the value of the host's real time clocksource at the instant when KVM_SET_CLOCK was called. The difference in elapsed time is added to the final kvmclock value that will be provided to guests.

Other flags returned by `KVM_GET_CLOCK` are accepted but ignored.

```
struct kvm_clock_data {
      __u64 clock;  /* kvmclock current value */
      __u32 flags;
      __u32 pad0;
      __u64 realtime;
      __u64 host_tsc;
      __u32 pad[4];
};
```

4.31 KVM_GET_VCPU_EVENTS

Capability
KVM_CAP_VCPU_EVENTS

Extended by
KVM_CAP_INTR_SHADOW

Architectures
x86, arm64

Type
vcpu ioctl

Parameters
struct kvm_vcpu_events (out)

Returns
0 on success, -1 on error

X86:

Gets currently pending exceptions, interrupts, and NMIs as well as related states of the vcpu.

```
struct kvm_vcpu_events {
      struct {
              __u8 injected;
              __u8 nr;
              __u8 has_error_code;
              __u8 pending;
              __u32 error_code;
      } exception;
      struct {
              __u8 injected;
              __u8 nr;
              __u8 soft;
              __u8 shadow;
      } interrupt;
      struct {
              __u8 injected;
              __u8 pending;
              __u8 masked;
              __u8 pad;
      } nmi;
      __u32 sipi_vector;
      __u32 flags;
      struct {
              __u8 smm;
              __u8 pending;
              __u8 smm_inside_nmi;
              __u8 latched_init;
      } smi;
      __u8 reserved[27];
      __u8 exception_has_payload;
      __u64 exception_payload;
};
```

The following bits are defined in the flags field:

- KVM_VCPUEVENT_VALID_SHADOW may be set to signal that interrupt.shadow contains a valid state.
- KVM_VCPUEVENT_VALID_SMM may be set to signal that smi contains a valid state.
- KVM_VCPUEVENT_VALID_PAYLOAD may be set to signal that the exception_has_payload, exception_payload, and exception.pending fields contain a valid state. This bit will be set whenever KVM_CAP_EXCEPTION_PAYLOAD is enabled.
- KVM_VCPUEVENT_VALID_TRIPLE_FAULT may be set to signal that the triple_fault_pending field contains a valid state. This bit will be set whenever KVM_CAP_X86_TRIPLE_FAULT_EVENT is enabled.

ARM64:

If the guest accesses a device that is being emulated by the host kernel in such a way that a real device would generate a physical SError, KVM may make a virtual SError pending for that VCPU. This system error interrupt remains pending until the guest takes the exception by unmasking PSTATE.A.

Running the VCPU may cause it to take a pending SError, or make an access that causes an SError to become pending. The event's description is only valid while the VPCU is not running.

This API provides a way to read and write the pending 'event' state that is not visible to the guest. To save, restore or migrate a VCPU the struct representing the state can be read then written using this GET/SET API, along with the other guest-visible registers. It is not possible to 'cancel' an SError that has been made pending.

A device being emulated in user-space may also wish to generate an SError. To do this the events structure can be populated by user-space. The current state should be read first, to ensure no existing SError is pending. If an existing SError is pending, the architecture's 'Multiple SError interrupts' rules should be followed. (2.5.3 of DDI0587.a ”ARM Reliability, Availability, and Serviceability (RAS) Specification”).

SError exceptions always have an ESR value. Some CPUs have the ability to specify what the virtual SError's ESR value should be. These systems will advertise KVM_CAP_ARM_INJECT_SERROR_ESR. In this case exception.has_esr will always have a non-zero value when read, and the agent making an SError pending should specify the ISS field in the lower 24 bits of exception.serror_esr. If the system supports KVM_CAP_ARM_INJECT_SERROR_ESR, but user-space sets the events with exception.has_esr as zero, KVM will choose an ESR.

Specifying exception.has_esr on a system that does not support it will return -EINVAL. Setting anything other than the lower 24bits of exception.serror_esr will return -EINVAL.

It is not possible to read back a pending external abort (injected via KVM_SET_VCPU_EVENTS or otherwise) because such an exception is always delivered directly to the virtual CPU).

```
struct kvm_vcpu_events {
      struct {
              __u8 serror_pending;
              __u8 serror_has_esr;
              __u8 ext_dabt_pending;
              /* Align it to 8 bytes */
              __u8 pad[5];
              __u64 serror_esr;
      } exception;
      __u32 reserved[12];
};
```

4.32 KVM_SET_VCPU_EVENTS

Capability
KVM_CAP_VCPU_EVENTS

Extended by
KVM_CAP_INTR_SHADOW

Architectures
x86, arm64

Type
vcpu ioctl

Parameters
struct kvm_vcpu_events (in)

Returns
0 on success, -1 on error

X86:

Set pending exceptions, interrupts, and NMIs as well as related states of the vcpu.

See KVM_GET_VCPU_EVENTS for the data structure.

Fields that may be modified asynchronously by running VCPUs can be excluded from the update. These fields are nmi.pending, sipi_vector, smi.smm, smi.pending. Keep the corresponding bits in the flags field cleared to suppress overwriting the current in-kernel state. The bits are:

KVM_VCPUEVENT_VALID_NMI_PENDING	transfer nmi.pending to the kernel
KVM_VCPUEVENT_VALID_SIPI_VECTOR	transfer sipi_vector
KVM_VCPUEVENT_VALID_SMM	transfer the smi sub-struct.

If KVM_CAP_INTR_SHADOW is available, KVM_VCPUEVENT_VALID_SHADOW can be set in the flags field to signal that interrupt.shadow contains a valid state and shall be written into the VCPU.

KVM_VCPUEVENT_VALID_SMM can only be set if KVM_CAP_X86_SMM is available.

If KVM_CAP_EXCEPTION_PAYLOAD is enabled, KVM_VCPUEVENT_VALID_PAYLOAD can be set in the flags field to signal that the exception_has_payload, exception_payload, and exception.pending fields contain a valid state and shall be written into the VCPU.

If KVM_CAP_X86_TRIPLE_FAULT_EVENT is enabled, KVM_VCPUEVENT_VALID_TRIPLE_FAULT can be set in flags field to signal that the triple_fault field contains a valid state and shall be written into the VCPU.

ARM64:

User space may need to inject several types of events to the guest.

Set the pending SError exception state for this VCPU. It is not possible to 'cancel' an Serror that has been made pending.

If the guest performed an access to I/O memory which could not be handled by userspace, for example because of missing instruction syndrome decode information or because there is no device mapped at the accessed IPA, then userspace can ask the kernel to inject an external abort using the address from the exiting fault on the VCPU. It is a programming error to set ext_dabt_pending after an exit which was not either KVM_EXIT_MMIO or KVM_EXIT_ARM_NISV. This feature is only available if the system supports KVM_CAP_ARM_INJECT_EXT_DABT. This is a helper which provides commonality in how userspace reports accesses for the above cases to guests, across different userspace implementations. Nevertheless, userspace can still emulate all Arm exceptions by manipulating individual registers using the KVM_SET_ONE_REG API.

See KVM_GET_VCPU_EVENTS for the data structure.

4.33 KVM_GET_DEBUGREGS

Capability
KVM_CAP_DEBUGREGS

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_debugregs (out)

Returns
0 on success, -1 on error

Reads debug registers from the vcpu.

```
struct kvm_debugregs {
      __u64 db[4];
      __u64 dr6;
      __u64 dr7;
      __u64 flags;
      __u64 reserved[9];
};
```

4.34 KVM_SET_DEBUGREGS

Capability
KVM_CAP_DEBUGREGS

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_debugregs (in)

Returns
0 on success, -1 on error

Writes debug registers into the vcpu.

See KVM_GET_DEBUGREGS for the data structure. The flags field is unused yet and must be cleared on entry.

4.35 KVM_SET_USER_MEMORY_REGION

Capability
KVM_CAP_USER_MEMORY

Architectures
all

Type
vm ioctl

Parameters
struct kvm_userspace_memory_region (in)

Returns
0 on success, -1 on error

```
struct kvm_userspace_memory_region {
      __u32 slot;
      __u32 flags;
      __u64 guest_phys_addr;
      __u64 memory_size; /* bytes */
      __u64 userspace_addr; /* start of the userspace allocated memory */
};

/* for kvm_userspace_memory_region::flags */
#define KVM_MEM_LOG_DIRTY_PAGES       (1UL << 0)
#define KVM_MEM_READONLY      (1UL << 1)
```

This ioctl allows the user to create, modify or delete a guest physical memory slot. Bits 0-15 of "slot" specify the slot id and this value should be less than the maximum number of user memory slots supported per VM. The maximum allowed slots can be queried using KVM_CAP_NR_MEMSLOTS. Slots may not overlap in guest physical address space.

If KVM_CAP_MULTI_ADDRESS_SPACE is available, bits 16-31 of "slot" specifies the address space which is being modified. They must be less than the value that KVM_CHECK_EXTENSION returns for the KVM_CAP_MULTI_ADDRESS_SPACE capability. Slots in separate address spaces are unrelated; the restriction on overlapping slots only applies within each address space.

Deleting a slot is done by passing zero for memory_size. When changing an existing slot, it may be moved in the guest physical memory space, or its flags may be modified, but it may not be resized.

Memory for the region is taken starting at the address denoted by the field userspace_addr, which must point at user addressable memory for the entire memory slot size. Any object may back this memory, including anonymous memory, ordinary files, and hugetlbfs.

On architectures that support a form of address tagging, userspace_addr must be an untagged address.

It is recommended that the lower 21 bits of guest_phys_addr and userspace_addr be identical. This allows large pages in the guest to be backed by large pages in the host.

The flags field supports two flags: KVM_MEM_LOG_DIRTY_PAGES and KVM_MEM_READONLY. The former can be set to instruct KVM to keep track of writes to memory within the slot. See KVM_GET_DIRTY_LOG ioctl to know how to use it. The latter can be set, if KVM_CAP_READONLY_MEM capability allows it, to make a new slot read-only. In this case, writes to this memory will be posted to userspace as KVM_EXIT_MMIO exits.

When the KVM_CAP_SYNC_MMU capability is available, changes in the backing of the memory region are automatically reflected into the guest. For example, an mmap() that affects the region will be made visible immediately. Another example is madvise(MADV_DROP).

Note: On arm64, a write generated by the page-table walker (to update the Access and Dirty flags, for example) never results in a KVM_EXIT_MMIO exit when the slot has the KVM_MEM_READONLY flag. This is because KVM cannot provide the data that would be written by the page-table walker, making it impossible to emulate the access. Instead, an abort (data abort if the cause of the page-table update was a load or a store, instruction abort if it was an instruction fetch) is injected in the guest.

4.36 KVM_SET_TSS_ADDR

Capability
KVM_CAP_SET_TSS_ADDR

Architectures
x86

Type
vm ioctl

Parameters
unsigned long tss_address (in)

Returns
0 on success, -1 on error

This ioctl defines the physical address of a three-page region in the guest physical address space. The region must be within the first 4GB of the guest physical address space and must

not conflict with any memory slot or any mmio address. The guest may malfunction if it accesses this memory region.

This ioctl is required on Intel-based hosts. This is needed on Intel hardware because of a quirk in the virtualization implementation (see the internals documentation when it pops into existence).

4.37 KVM_ENABLE_CAP

Capability
KVM_CAP_ENABLE_CAP

Architectures
mips, ppc, s390, x86, loongarch

Type
vcpu ioctl

Parameters
struct kvm_enable_cap (in)

Returns
0 on success; -1 on error

Capability
KVM_CAP_ENABLE_CAP_VM

Architectures
all

Type
vm ioctl

Parameters
struct kvm_enable_cap (in)

Returns
0 on success; -1 on error

Note: Not all extensions are enabled by default. Using this ioctl the application can enable an extension, making it available to the guest.

On systems that do not support this ioctl, it always fails. On systems that do support it, it only works for extensions that are supported for enablement.

To check if a capability can be enabled, the KVM_CHECK_EXTENSION ioctl should be used.

```
struct kvm_enable_cap {
     /* in */
     __u32 cap;
```

The capability that is supposed to get enabled.

```
__u32 flags;
```

A bitfield indicating future enhancements. Has to be 0 for now.

```
__u64 args[4];
```

Arguments for enabling a feature. If a feature needs initial values to function properly, this is the place to put them.

```
    __u8  pad[64];
};
```

The vcpu ioctl should be used for vcpu-specific capabilities, the vm ioctl for vm-wide capabilities.

4.38 KVM_GET_MP_STATE

Capability
KVM_CAP_MP_STATE

Architectures
x86, s390, arm64, riscv, loongarch

Type
vcpu ioctl

Parameters
struct kvm_mp_state (out)

Returns
0 on success; -1 on error

```
struct kvm_mp_state {
     __u32 mp_state;
};
```

Returns the vcpu's current ”multiprocessing state” (though also valid on uniprocessor guests).

Possible values are:

KVM_MP_STATE_RUNNABLE	the vcpu is currently running [x86,arm64,riscv,loongarch]
KVM_MP_STATE_UNINITIALIZED	the vcpu is an application processor (AP) which has not yet received an INIT signal [x86]
KVM_MP_STATE_INIT_RECEIVED	the vcpu has received an INIT signal, and is now ready for a SIPI [x86]
KVM_MP_STATE_HALTED	the vcpu has executed a HLT instruction and is waiting for an interrupt [x86]
KVM_MP_STATE_SIPI_RECEIVED	the vcpu has just received a SIPI (vector accessible via KVM_GET_VCPU_EVENTS) [x86]
KVM_MP_STATE_STOPPED	the vcpu is stopped [s390,arm64,riscv]
KVM_MP_STATE_CHECK_STOP	the vcpu is in a special error state [s390]
KVM_MP_STATE_OPERATING	the vcpu is operating (running or halted) [s390]
KVM_MP_STATE_LOAD	the vcpu is in a special load/startup state [s390]
KVM_MP_STATE_SUSPENDED	the vcpu is in a suspend state and is waiting for a wakeup event [arm64]

On x86, this ioctl is only useful after KVM_CREATE_IRQCHIP. Without an in-kernel irqchip, the multiprocessing state must be maintained by userspace on these architectures.

For arm64:

If a vCPU is in the KVM_MP_STATE_SUSPENDED state, KVM will emulate the architectural execution of a WFI instruction.

If a wakeup event is recognized, KVM will exit to userspace with a KVM_SYSTEM_EVENT exit, where the event type is KVM_SYSTEM_EVENT_WAKEUP. If userspace wants to honor the wakeup, it must set the vCPU's MP state to KVM_MP_STATE_RUNNABLE. If it does not, KVM will continue to await a wakeup event in subsequent calls to KVM_RUN.

Warning: If userspace intends to keep the vCPU in a SUSPENDED state, it is strongly recommended that userspace take action to suppress the wakeup event (such as masking an interrupt). Otherwise, subsequent calls to KVM_RUN will immediately exit with a KVM_SYSTEM_EVENT_WAKEUP event and inadvertently waste CPU cycles.

Additionally, if userspace takes action to suppress a wakeup event, it is strongly recommended that it also restores the vCPU to its original state when the vCPU is made RUNNABLE again. For example, if userspace masked a pending interrupt to suppress the wakeup, the interrupt should be unmasked before returning control to the guest.

For riscv:

The only states that are valid are KVM_MP_STATE_STOPPED and KVM_MP_STATE_RUNNABLE which reflect if the vcpu is paused or not.

On LoongArch, only the KVM_MP_STATE_RUNNABLE state is used to reflect whether the vcpu is runnable.

4.39 KVM_SET_MP_STATE

Capability
KVM_CAP_MP_STATE

Architectures
x86, s390, arm64, riscv, loongarch

Type
vcpu ioctl

Parameters
struct kvm_mp_state (in)

Returns
0 on success; -1 on error

Sets the vcpu's current ”multiprocessing state”; see KVM_GET_MP_STATE for arguments.

On x86, this ioctl is only useful after KVM_CREATE_IRQCHIP. Without an in-kernel irqchip, the multiprocessing state must be maintained by userspace on these architectures.

For arm64/riscv:

The only states that are valid are KVM_MP_STATE_STOPPED and KVM_MP_STATE_RUNNABLE which reflect if the vcpu should be paused or not.

On LoongArch, only the KVM_MP_STATE_RUNNABLE state is used to reflect whether the vcpu is runnable.

4.40 KVM_SET_IDENTITY_MAP_ADDR

Capability
KVM_CAP_SET_IDENTITY_MAP_ADDR

Architectures
x86

Type
vm ioctl

Parameters
unsigned long identity (in)

Returns
0 on success, -1 on error

This ioctl defines the physical address of a one-page region in the guest physical address space. The region must be within the first 4GB of the guest physical address space and must not conflict with any memory slot or any mmio address. The guest may malfunction if it accesses this memory region.

Setting the address to 0 will result in resetting the address to its default (0xfffbc000).

This ioctl is required on Intel-based hosts. This is needed on Intel hardware because of a quirk in the virtualization implementation (see the internals documentation when it pops into existence).

Fails if any VCPU has already been created.

4.41 KVM_SET_BOOT_CPU_ID

Capability
KVM_CAP_SET_BOOT_CPU_ID

Architectures
x86

Type
vm ioctl

Parameters
unsigned long vcpu_id

Returns
0 on success, -1 on error

Define which vcpu is the Bootstrap Processor (BSP). Values are the same as the vcpu id in KVM_CREATE_VCPU. If this ioctl is not called, the default is vcpu 0. This ioctl has to be called before vcpu creation, otherwise it will return EBUSY error.

4.42 KVM_GET_XSAVE

Capability
KVM_CAP_XSAVE

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xsave (out)

Returns
0 on success, -1 on error

```
struct kvm_xsave {
      __u32 region[1024];
      __u32 extra[0];
};
```

This ioctl would copy current vcpu's xsave struct to the userspace.

4.43 KVM_SET_XSAVE

Capability
KVM_CAP_XSAVE and KVM_CAP_XSAVE2

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xsave (in)

Returns
0 on success, -1 on error

```
struct kvm_xsave {
      __u32 region[1024];
      __u32 extra[0];
};
```

This ioctl would copy userspace's xsave struct to the kernel. It copies as many bytes as are returned by KVM_CHECK_EXTENSION(KVM_CAP_XSAVE2), when invoked on the vm file descriptor. The size value returned by KVM_CHECK_EXTENSION(KVM_CAP_XSAVE2) will always be at least 4096. Currently, it is only greater than 4096 if a dynamic feature has been enabled with `arch_prctl()`, but this may change in the future.

The offsets of the state save areas in struct kvm_xsave follow the contents of CPUID leaf 0xD on the host.

4.44 KVM_GET_XCRS

Capability
KVM_CAP_XCRS

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xcrs (out)

Returns
0 on success, -1 on error

```
struct kvm_xcr {
      __u32 xcr;
      __u32 reserved;
      __u64 value;
};

struct kvm_xcrs {
```

```
      __u32 nr_xcrs;
      __u32 flags;
      struct kvm_xcr xcrs[KVM_MAX_XCRS];
      __u64 padding[16];
};
```

This ioctl would copy current vcpu's xcrs to the userspace.

4.45 KVM_SET_XCRS

Capability
KVM_CAP_XCRS

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xcrs (in)

Returns
0 on success, -1 on error

```
struct kvm_xcr {
      __u32 xcr;
      __u32 reserved;
      __u64 value;
};

struct kvm_xcrs {
      __u32 nr_xcrs;
      __u32 flags;
      struct kvm_xcr xcrs[KVM_MAX_XCRS];
      __u64 padding[16];
};
```

This ioctl would set vcpu's xcr to the value userspace specified.

4.46 KVM_GET_SUPPORTED_CPUID

Capability
KVM_CAP_EXT_CPUID

Architectures
x86

Type
system ioctl

Parameters
struct kvm_cpuid2 (in/out)

Returns

0 on success, -1 on error

```
struct kvm_cpuid2 {
      __u32 nent;
      __u32 padding;
      struct kvm_cpuid_entry2 entries[0];
};

#define KVM_CPUID_FLAG_SIGNIFCANT_INDEX               BIT(0)
#define KVM_CPUID_FLAG_STATEFUL_FUNC          BIT(1) /* deprecated */
#define KVM_CPUID_FLAG_STATE_READ_NEXT                BIT(2) /* deprecated */

struct kvm_cpuid_entry2 {
      __u32 function;
      __u32 index;
      __u32 flags;
      __u32 eax;
      __u32 ebx;
      __u32 ecx;
      __u32 edx;
      __u32 padding[3];
};
```

This ioctl returns x86 cpuid features which are supported by both the hardware and kvm in its default configuration. Userspace can use the information returned by this ioctl to construct cpuid information (for KVM_SET_CPUID2) that is consistent with hardware, kernel, and userspace capabilities, and with user requirements (for example, the user may wish to constrain cpuid to emulate older hardware, or for feature consistency across a cluster).

Dynamically-enabled feature bits need to be requested with `arch_prctl()` before calling this ioctl. Feature bits that have not been requested are excluded from the result.

Note that certain capabilities, such as KVM_CAP_X86_DISABLE_EXITS, may expose cpuid features (e.g. MONITOR) which are not supported by kvm in its default configuration. If userspace enables such capabilities, it is responsible for modifying the results of this ioctl appropriately.

Userspace invokes KVM_GET_SUPPORTED_CPUID by passing a kvm_cpuid2 structure with the 'nent' field indicating the number of entries in the variable-size array 'entries'. If the number of entries is too low to describe the cpu capabilities, an error (E2BIG) is returned. If the number is too high, the 'nent' field is adjusted and an error (ENOMEM) is returned. If the number is just right, the 'nent' field is adjusted to the number of valid entries in the 'entries' array, which is then filled.

The entries returned are the host cpuid as returned by the cpuid instruction, with unknown or unsupported features masked out. Some features (for example, x2apic), may not be present in the host cpu, but are exposed by kvm if it can emulate them efficiently. The fields in each entry are defined as follows:

function:

the eax value used to obtain the entry

index:

the ecx value used to obtain the entry (for entries that are affected by ecx)

flags:

an OR of zero or more of the following:

KVM_CPUID_FLAG_SIGNIFCANT_INDEX:
if the index field is valid

eax, ebx, ecx, edx:
the values returned by the cpuid instruction for this function/index combination

The TSC deadline timer feature (CPUID leaf 1, ecx[24]) is always returned as false, since the feature depends on KVM_CREATE_IRQCHIP for local APIC support. Instead it is reported via:

```
ioctl(KVM_CHECK_EXTENSION, KVM_CAP_TSC_DEADLINE_TIMER)
```

if that returns true and you use KVM_CREATE_IRQCHIP, or if you emulate the feature in userspace, then you can enable the feature for KVM_SET_CPUID2.

4.47 KVM_PPC_GET_PVINFO

Capability
KVM_CAP_PPC_GET_PVINFO

Architectures
ppc

Type
vm ioctl

Parameters
struct kvm_ppc_pvinfo (out)

Returns
0 on success, !0 on error

```
struct kvm_ppc_pvinfo {
      __u32 flags;
      __u32 hcall[4];
      __u8  pad[108];
};
```

This ioctl fetches PV specific information that need to be passed to the guest using the device tree or other means from vm context.

The hcall array defines 4 instructions that make up a hypercall.

If any additional field gets added to this structure later on, a bit for that additional piece of information will be set in the flags bitmap.

The flags bitmap is defined as:

```
/* the host supports the ePAPR idle hcall
#define KVM_PPC_PVINFO_FLAGS_EV_IDLE   (1<<0)
```

4.52 KVM_SET_GSI_ROUTING

Capability
KVM_CAP_IRQ_ROUTING

Architectures
x86 s390 arm64

Type
vm ioctl

Parameters
struct kvm_irq_routing (in)

Returns
0 on success, -1 on error

Sets the GSI routing table entries, overwriting any previously set entries.

On arm64, GSI routing has the following limitation:

- GSI routing does not apply to KVM_IRQ_LINE but only to KVM_IRQFD.

```
struct kvm_irq_routing {
      __u32 nr;
      __u32 flags;
      struct kvm_irq_routing_entry entries[0];
};
```

No flags are specified so far, the corresponding field must be set to zero.

```
struct kvm_irq_routing_entry {
      __u32 gsi;
      __u32 type;
      __u32 flags;
      __u32 pad;
      union {
              struct kvm_irq_routing_irqchip irqchip;
              struct kvm_irq_routing_msi msi;
              struct kvm_irq_routing_s390_adapter adapter;
              struct kvm_irq_routing_hv_sint hv_sint;
              struct kvm_irq_routing_xen_evtchn xen_evtchn;
              __u32 pad[8];
      } u;
};

/* gsi routing entry types */
#define KVM_IRQ_ROUTING_IRQCHIP 1
#define KVM_IRQ_ROUTING_MSI 2
#define KVM_IRQ_ROUTING_S390_ADAPTER 3
#define KVM_IRQ_ROUTING_HV_SINT 4
#define KVM_IRQ_ROUTING_XEN_EVTCHN 5
```

flags:

- KVM_MSI_VALID_DEVID: used along with KVM_IRQ_ROUTING_MSI routing entry type, specifies that the devid field contains a valid value. The per-VM KVM_CAP_MSI_DEVID capability advertises the requirement to provide the device ID. If this capability is not available, userspace should never set the KVM_MSI_VALID_DEVID flag as the ioctl might fail.
- zero otherwise

```
struct kvm_irq_routing_irqchip {
      __u32 irqchip;
      __u32 pin;
};

struct kvm_irq_routing_msi {
      __u32 address_lo;
      __u32 address_hi;
      __u32 data;
      union {
              __u32 pad;
              __u32 devid;
      };
};
```

If KVM_MSI_VALID_DEVID is set, devid contains a unique device identifier for the device that wrote the MSI message. For PCI, this is usually a BFD identifier in the lower 16 bits.

On x86, address_hi is ignored unless the KVM_X2APIC_API_USE_32BIT_IDS feature of KVM_CAP_X2APIC_API capability is enabled. If it is enabled, address_hi bits 31-8 provide bits 31-8 of the destination id. Bits 7-0 of address_hi must be zero.

```
struct kvm_irq_routing_s390_adapter {
      __u64 ind_addr;
      __u64 summary_addr;
      __u64 ind_offset;
      __u32 summary_offset;
      __u32 adapter_id;
};

struct kvm_irq_routing_hv_sint {
      __u32 vcpu;
      __u32 sint;
};

struct kvm_irq_routing_xen_evtchn {
      __u32 port;
      __u32 vcpu;
      __u32 priority;
};
```

When KVM_CAP_XEN_HVM includes the KVM_XEN_HVM_CONFIG_EVTCHN_2LEVEL bit in its indication of supported features, routing to Xen event channels is supported. Although the priority field is present, only the value KVM_XEN_HVM_CONFIG_EVTCHN_2LEVEL is supported, which means delivery by 2 level event channels. FIFO event channel support may be added in

the future.

4.55 KVM_SET_TSC_KHZ

Capability
KVM_CAP_TSC_CONTROL / KVM_CAP_VM_TSC_CONTROL

Architectures
x86

Type
vcpu ioctl / vm ioctl

Parameters
virtual tsc_khz

Returns
0 on success, -1 on error

Specifies the tsc frequency for the virtual machine. The unit of the frequency is KHz.

If the KVM_CAP_VM_TSC_CONTROL capability is advertised, this can also be used as a vm ioctl to set the initial tsc frequency of subsequently created vCPUs.

4.56 KVM_GET_TSC_KHZ

Capability
KVM_CAP_GET_TSC_KHZ / KVM_CAP_VM_TSC_CONTROL

Architectures
x86

Type
vcpu ioctl / vm ioctl

Parameters
none

Returns
virtual tsc-khz on success, negative value on error

Returns the tsc frequency of the guest. The unit of the return value is KHz. If the host has unstable tsc this ioctl returns -EIO instead as an error.

4.57 KVM_GET_LAPIC

Capability
KVM_CAP_IRQCHIP

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_lapic_state (out)

Returns
0 on success, -1 on error

```
#define KVM_APIC_REG_SIZE 0x400
struct kvm_lapic_state {
      char regs[KVM_APIC_REG_SIZE];
};
```

Reads the Local APIC registers and copies them into the input argument. The data format and layout are the same as documented in the architecture manual.

If KVM_X2APIC_API_USE_32BIT_IDS feature of KVM_CAP_X2APIC_API is enabled, then the format of APIC_ID register depends on the APIC mode (reported by MSR_IA32_APICBASE) of its VCPU. x2APIC stores APIC ID in the APIC_ID register (bytes 32-35). xAPIC only allows an 8-bit APIC ID which is stored in bits 31-24 of the APIC register, or equivalently in byte 35 of struct kvm_lapic_state's regs field. KVM_GET_LAPIC must then be called after MSR_IA32_APICBASE has been set with KVM_SET_MSR.

If KVM_X2APIC_API_USE_32BIT_IDS feature is disabled, struct kvm_lapic_state always uses xAPIC format.

4.58 KVM_SET_LAPIC

Capability
KVM_CAP_IRQCHIP

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_lapic_state (in)

Returns
0 on success, -1 on error

```
#define KVM_APIC_REG_SIZE 0x400
struct kvm_lapic_state {
      char regs[KVM_APIC_REG_SIZE];
};
```

Copies the input argument into the Local APIC registers. The data format and layout are the same as documented in the architecture manual.

The format of the APIC ID register (bytes 32-35 of struct kvm_lapic_state's regs field) depends on the state of the KVM_CAP_X2APIC_API capability. See the note in KVM_GET_LAPIC.

4.59 KVM_IOEVENTFD

Capability
KVM_CAP_IOEVENTFD

Architectures
all

Type
vm ioctl

Parameters
struct kvm_ioeventfd (in)

Returns
0 on success, !0 on error

This ioctl attaches or detaches an ioeventfd to a legal pio/mmio address within the guest. A guest write in the registered address will signal the provided event instead of triggering an exit.

```
struct kvm_ioeventfd {
      __u64 datamatch;
      __u64 addr;        /* legal pio/mmio address */
      __u32 len;         /* 0, 1, 2, 4, or 8 bytes    */
      __s32 fd;
      __u32 flags;
      __u8  pad[36];
};
```

For the special case of virtio-ccw devices on s390, the ioevent is matched to a subchannel/virtqueue tuple instead.

The following flags are defined:

```
#define KVM_IOEVENTFD_FLAG_DATAMATCH (1 << kvm_ioeventfd_flag_nr_datamatch)
#define KVM_IOEVENTFD_FLAG_PIO       (1 << kvm_ioeventfd_flag_nr_pio)
#define KVM_IOEVENTFD_FLAG_DEASSIGN  (1 << kvm_ioeventfd_flag_nr_deassign)
#define KVM_IOEVENTFD_FLAG_VIRTIO_CCW_NOTIFY \
      (1 << kvm_ioeventfd_flag_nr_virtio_ccw_notify)
```

If datamatch flag is set, the event will be signaled only if the written value to the registered address is equal to datamatch in struct kvm_ioeventfd.

For virtio-ccw devices, addr contains the subchannel id and datamatch the virtqueue index.

With KVM_CAP_IOEVENTFD_ANY_LENGTH, a zero length ioeventfd is allowed, and the kernel will ignore the length of guest write and may get a faster vmexit. The speedup may only apply to specific architectures, but the ioeventfd will work anyway.

4.60 KVM_DIRTY_TLB

Capability
KVM_CAP_SW_TLB

Architectures
ppc

Type
vcpu ioctl

Parameters
struct kvm_dirty_tlb (in)

Returns
0 on success, -1 on error

```
struct kvm_dirty_tlb {
      __u64 bitmap;
      __u32 num_dirty;
};
```

This must be called whenever userspace has changed an entry in the shared TLB, prior to calling KVM_RUN on the associated vcpu.

The ”bitmap” field is the userspace address of an array. This array consists of a number of bits, equal to the total number of TLB entries as determined by the last successful call to KVM_CONFIG_TLB, rounded up to the nearest multiple of 64.

Each bit corresponds to one TLB entry, ordered the same as in the shared TLB array.

The array is little-endian: the bit 0 is the least significant bit of the first byte, bit 8 is the least significant bit of the second byte, etc. This avoids any complications with differing word sizes.

The ”num_dirty” field is a performance hint for KVM to determine whether it should skip processing the bitmap and just invalidate everything. It must be set to the number of set bits in the bitmap.

4.62 KVM_CREATE_SPAPR_TCE

Capability
KVM_CAP_SPAPR_TCE

Architectures
powerpc

Type
vm ioctl

Parameters
struct kvm_create_spapr_tce (in)

Returns
file descriptor for manipulating the created TCE table

This creates a virtual TCE (translation control entry) table, which is an IOMMU for PAPR-style virtual I/O. It is used to translate logical addresses used in virtual I/O into guest physical addresses, and provides a scatter/gather capability for PAPR virtual I/O.

```
/* for KVM_CAP_SPAPR_TCE */
struct kvm_create_spapr_tce {
      __u64 liobn;
      __u32 window_size;
};
```

The liobn field gives the logical IO bus number for which to create a TCE table. The window_size field specifies the size of the DMA window which this TCE table will translate - the table will contain one 64 bit TCE entry for every 4kiB of the DMA window.

When the guest issues an H_PUT_TCE hcall on a liobn for which a TCE table has been created using this ioctl(), the kernel will handle it in real mode, updating the TCE table. H_PUT_TCE calls for other liobns will cause a vm exit and must be handled by userspace.

The return value is a file descriptor which can be passed to mmap(2) to map the created TCE table into userspace. This lets userspace read the entries written by kernel-handled H_PUT_TCE calls, and also lets userspace update the TCE table directly which is useful in some circumstances.

4.63 KVM_ALLOCATE_RMA

Capability
KVM_CAP_PPC_RMA

Architectures
powerpc

Type
vm ioctl

Parameters
struct kvm_allocate_rma (out)

Returns
file descriptor for mapping the allocated RMA

This allocates a Real Mode Area (RMA) from the pool allocated at boot time by the kernel. An RMA is a physically-contiguous, aligned region of memory used on older POWER processors to provide the memory which will be accessed by real-mode (MMU off) accesses in a KVM guest. POWER processors support a set of sizes for the RMA that usually includes 64MB, 128MB, 256MB and some larger powers of two.

```
/* for KVM_ALLOCATE_RMA */
struct kvm_allocate_rma {
      __u64 rma_size;
};
```

The return value is a file descriptor which can be passed to mmap(2) to map the allocated RMA into userspace. The mapped area can then be passed to the KVM_SET_USER_MEMORY_REGION ioctl to establish it as the RMA for a virtual machine. The

size of the RMA in bytes (which is fixed at host kernel boot time) is returned in the rma_size field of the argument structure.

The KVM_CAP_PPC_RMA capability is 1 or 2 if the KVM_ALLOCATE_RMA ioctl is supported; 2 if the processor requires all virtual machines to have an RMA, or 1 if the processor can use an RMA but doesn't require it, because it supports the Virtual RMA (VRMA) facility.

4.64 KVM_NMI

Capability
KVM_CAP_USER_NMI

Architectures
x86

Type
vcpu ioctl

Parameters
none

Returns
0 on success, -1 on error

Queues an NMI on the thread's vcpu. Note this is well defined only when KVM_CREATE_IRQCHIP has not been called, since this is an interface between the virtual cpu core and virtual local APIC. After KVM_CREATE_IRQCHIP has been called, this interface is completely emulated within the kernel.

To use this to emulate the LINT1 input with KVM_CREATE_IRQCHIP, use the following algorithm:

- pause the vcpu
- read the local APIC's state (KVM_GET_LAPIC)
- check whether changing LINT1 will queue an NMI (see the LVT entry for LINT1)
- if so, issue KVM_NMI
- resume the vcpu

Some guests configure the LINT1 NMI input to cause a panic, aiding in debugging.

4.65 KVM_S390_UCAS_MAP

Capability
KVM_CAP_S390_UCONTROL

Architectures
s390

Type
vcpu ioctl

Parameters
struct kvm_s390_ucas_mapping (in)

Returns
0 in case of success

The parameter is defined like this:

```
struct kvm_s390_ucas_mapping {
        __u64 user_addr;
        __u64 vcpu_addr;
        __u64 length;
};
```

This ioctl maps the memory at ”user_addr” with the length ”length” to the vcpu's address space starting at ”vcpu_addr”. All parameters need to be aligned by 1 megabyte.

4.66 KVM_S390_UCAS_UNMAP

Capability
KVM_CAP_S390_UCONTROL

Architectures
s390

Type
vcpu ioctl

Parameters
struct kvm_s390_ucas_mapping (in)

Returns
0 in case of success

The parameter is defined like this:

```
struct kvm_s390_ucas_mapping {
        __u64 user_addr;
        __u64 vcpu_addr;
        __u64 length;
};
```

This ioctl unmaps the memory in the vcpu's address space starting at ”vcpu_addr” with the length ”length”. The field ”user_addr” is ignored. All parameters need to be aligned by 1 megabyte.

4.67 KVM_S390_VCPU_FAULT

Capability
KVM_CAP_S390_UCONTROL

Architectures
s390

Type
vcpu ioctl

Parameters
vcpu absolute address (in)

Returns
0 in case of success

This call creates a page table entry on the virtual cpu's address space (for user controlled virtual machines) or the virtual machine's address space (for regular virtual machines). This only works for minor faults, thus it's recommended to access subject memory page via the user page table upfront. This is useful to handle validity intercepts for user controlled virtual machines to fault in the virtual cpu's lowcore pages prior to calling the KVM_RUN ioctl.

4.68 KVM_SET_ONE_REG

Capability
KVM_CAP_ONE_REG

Architectures
all

Type
vcpu ioctl

Parameters
struct kvm_one_reg (in)

Returns
0 on success, negative value on failure

Errors:

ENOENT	no such register
EINVAL	invalid register ID, or no such register or used with VMs in protected virtualization mode on s390
EPERM	(arm64) register access not allowed before vcpu finalization
EBUSY	(riscv) changing register value not allowed after the vcpu has run at least once

(These error codes are indicative only: do not rely on a specific error code being returned in a specific situation.)

```
 struct kvm_one_reg {
      __u64 id;
      __u64 addr;
};
```

Using this ioctl, a single vcpu register can be set to a specific value defined by user space with the passed in struct kvm_one_reg, where id refers to the register identifier as described below and addr is a pointer to a variable with the respective size. There can be architecture agnostic and architecture specific registers. Each have their own range of operation and their own constants and width. To keep track of the implemented registers, find a list below:

Arch	Register	Width (bits)
PPC	KVM_REG_PPC_HIOR	64
PPC	KVM_REG_PPC_IAC1	64
PPC	KVM_REG_PPC_IAC2	64
PPC	KVM_REG_PPC_IAC3	64
PPC	KVM_REG_PPC_IAC4	64
PPC	KVM_REG_PPC_DAC1	64
PPC	KVM_REG_PPC_DAC2	64
PPC	KVM_REG_PPC_DABR	64
PPC	KVM_REG_PPC_DSCR	64
PPC	KVM_REG_PPC_PURR	64
PPC	KVM_REG_PPC_SPURR	64
PPC	KVM_REG_PPC_DAR	64
PPC	KVM_REG_PPC_DSISR	32
PPC	KVM_REG_PPC_AMR	64
PPC	KVM_REG_PPC_UAMOR	64
PPC	KVM_REG_PPC_MMCR0	64
PPC	KVM_REG_PPC_MMCR1	64
PPC	KVM_REG_PPC_MMCRA	64
PPC	KVM_REG_PPC_MMCR2	64
PPC	KVM_REG_PPC_MMCRS	64
PPC	KVM_REG_PPC_MMCR3	64
PPC	KVM_REG_PPC_SIAR	64
PPC	KVM_REG_PPC_SDAR	64
PPC	KVM_REG_PPC_SIER	64
PPC	KVM_REG_PPC_SIER2	64
PPC	KVM_REG_PPC_SIER3	64
PPC	KVM_REG_PPC_PMC1	32
PPC	KVM_REG_PPC_PMC2	32
PPC	KVM_REG_PPC_PMC3	32
PPC	KVM_REG_PPC_PMC4	32
PPC	KVM_REG_PPC_PMC5	32
PPC	KVM_REG_PPC_PMC6	32
PPC	KVM_REG_PPC_PMC7	32
PPC	KVM_REG_PPC_PMC8	32
PPC	KVM_REG_PPC_FPR0	64
...		
PPC	KVM_REG_PPC_FPR31	64
PPC	KVM_REG_PPC_VR0	128
...		
PPC	KVM_REG_PPC_VR31	128
PPC	KVM_REG_PPC_VSR0	128
...		
PPC	KVM_REG_PPC_VSR31	128
PPC	KVM_REG_PPC_FPSCR	64
PPC	KVM_REG_PPC_VSCR	32
PPC	KVM_REG_PPC_VPA_ADDR	64
PPC	KVM_REG_PPC_VPA_SLB	128
PPC	KVM_REG_PPC_VPA_DTL	128

continues on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
PPC	KVM_REG_PPC_EPCR	32
PPC	KVM_REG_PPC_EPR	32
PPC	KVM_REG_PPC_TCR	32
PPC	KVM_REG_PPC_TSR	32
PPC	KVM_REG_PPC_OR_TSR	32
PPC	KVM_REG_PPC_CLEAR_TSR	32
PPC	KVM_REG_PPC_MAS0	32
PPC	KVM_REG_PPC_MAS1	32
PPC	KVM_REG_PPC_MAS2	64
PPC	KVM_REG_PPC_MAS7_3	64
PPC	KVM_REG_PPC_MAS4	32
PPC	KVM_REG_PPC_MAS6	32
PPC	KVM_REG_PPC_MMUCFG	32
PPC	KVM_REG_PPC_TLB0CFG	32
PPC	KVM_REG_PPC_TLB1CFG	32
PPC	KVM_REG_PPC_TLB2CFG	32
PPC	KVM_REG_PPC_TLB3CFG	32
PPC	KVM_REG_PPC_TLB0PS	32
PPC	KVM_REG_PPC_TLB1PS	32
PPC	KVM_REG_PPC_TLB2PS	32
PPC	KVM_REG_PPC_TLB3PS	32
PPC	KVM_REG_PPC_EPTCFG	32
PPC	KVM_REG_PPC_ICP_STATE	64
PPC	KVM_REG_PPC_VP_STATE	128
PPC	KVM_REG_PPC_TB_OFFSET	64
PPC	KVM_REG_PPC_SPMC1	32
PPC	KVM_REG_PPC_SPMC2	32
PPC	KVM_REG_PPC_IAMR	64
PPC	KVM_REG_PPC_TFHAR	64
PPC	KVM_REG_PPC_TFIAR	64
PPC	KVM_REG_PPC_TEXASR	64
PPC	KVM_REG_PPC_FSCR	64
PPC	KVM_REG_PPC_PSPB	32
PPC	KVM_REG_PPC_EBBHR	64
PPC	KVM_REG_PPC_EBBRR	64
PPC	KVM_REG_PPC_BESCR	64
PPC	KVM_REG_PPC_TAR	64
PPC	KVM_REG_PPC_DPDES	64
PPC	KVM_REG_PPC_DAWR	64
PPC	KVM_REG_PPC_DAWRX	64
PPC	KVM_REG_PPC_CIABR	64
PPC	KVM_REG_PPC_IC	64
PPC	KVM_REG_PPC_VTB	64
PPC	KVM_REG_PPC_CSIGR	64
PPC	KVM_REG_PPC_TACR	64
PPC	KVM_REG_PPC_TCSCR	64
PPC	KVM_REG_PPC_PID	64

continues on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
PPC	KVM_REG_PPC_ACOP	64
PPC	KVM_REG_PPC_VRSAVE	32
PPC	KVM_REG_PPC_LPCR	32
PPC	KVM_REG_PPC_LPCR_64	64
PPC	KVM_REG_PPC_PPR	64
PPC	KVM_REG_PPC_ARCH_COMPAT	32
PPC	KVM_REG_PPC_DABRX	32
PPC	KVM_REG_PPC_WORT	64
PPC	KVM_REG_PPC_SPRG9	64
PPC	KVM_REG_PPC_DBSR	32
PPC	KVM_REG_PPC_TIDR	64
PPC	KVM_REG_PPC_PSSCR	64
PPC	KVM_REG_PPC_DEC_EXPIRY	64
PPC	KVM_REG_PPC_PTCR	64
PPC	KVM_REG_PPC_DAWR1	64
PPC	KVM_REG_PPC_DAWRX1	64
PPC	KVM_REG_PPC_TM_GPR0	64
...		
PPC	KVM_REG_PPC_TM_GPR31	64
PPC	KVM_REG_PPC_TM_VSR0	128
...		
PPC	KVM_REG_PPC_TM_VSR63	128
PPC	KVM_REG_PPC_TM_CR	64
PPC	KVM_REG_PPC_TM_LR	64
PPC	KVM_REG_PPC_TM_CTR	64
PPC	KVM_REG_PPC_TM_FPSCR	64
PPC	KVM_REG_PPC_TM_AMR	64
PPC	KVM_REG_PPC_TM_PPR	64
PPC	KVM_REG_PPC_TM_VRSAVE	64
PPC	KVM_REG_PPC_TM_VSCR	32
PPC	KVM_REG_PPC_TM_DSCR	64
PPC	KVM_REG_PPC_TM_TAR	64
PPC	KVM_REG_PPC_TM_XER	64
MIPS	KVM_REG_MIPS_R0	64
...		
MIPS	KVM_REG_MIPS_R31	64
MIPS	KVM_REG_MIPS_HI	64
MIPS	KVM_REG_MIPS_LO	64
MIPS	KVM_REG_MIPS_PC	64
MIPS	KVM_REG_MIPS_CP0_INDEX	32
MIPS	KVM_REG_MIPS_CP0_ENTRYLO0	64
MIPS	KVM_REG_MIPS_CP0_ENTRYLO1	64
MIPS	KVM_REG_MIPS_CP0_CONTEXT	64
MIPS	KVM_REG_MIPS_CP0_CONTEXTCONFIG	32
MIPS	KVM_REG_MIPS_CP0_USERLOCAL	64
MIPS	KVM_REG_MIPS_CP0_XCONTEXTCONFIG	64
MIPS	KVM_REG_MIPS_CP0_PAGEMASK	32

continues on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
MIPS	KVM_REG_MIPS_CP0_PAGEGRAIN	32
MIPS	KVM_REG_MIPS_CP0_SEGCTL0	64
MIPS	KVM_REG_MIPS_CP0_SEGCTL1	64
MIPS	KVM_REG_MIPS_CP0_SEGCTL2	64
MIPS	KVM_REG_MIPS_CP0_PWBASE	64
MIPS	KVM_REG_MIPS_CP0_PWFIELD	64
MIPS	KVM_REG_MIPS_CP0_PWSIZE	64
MIPS	KVM_REG_MIPS_CP0_WIRED	32
MIPS	KVM_REG_MIPS_CP0_PWCTL	32
MIPS	KVM_REG_MIPS_CP0_HWRENA	32
MIPS	KVM_REG_MIPS_CP0_BADVADDR	64
MIPS	KVM_REG_MIPS_CP0_BADINSTR	32
MIPS	KVM_REG_MIPS_CP0_BADINSTRP	32
MIPS	KVM_REG_MIPS_CP0_COUNT	32
MIPS	KVM_REG_MIPS_CP0_ENTRYHI	64
MIPS	KVM_REG_MIPS_CP0_COMPARE	32
MIPS	KVM_REG_MIPS_CP0_STATUS	32
MIPS	KVM_REG_MIPS_CP0_INTCTL	32
MIPS	KVM_REG_MIPS_CP0_CAUSE	32
MIPS	KVM_REG_MIPS_CP0_EPC	64
MIPS	KVM_REG_MIPS_CP0_PRID	32
MIPS	KVM_REG_MIPS_CP0_EBASE	64
MIPS	KVM_REG_MIPS_CP0_CONFIG	32
MIPS	KVM_REG_MIPS_CP0_CONFIG1	32
MIPS	KVM_REG_MIPS_CP0_CONFIG2	32
MIPS	KVM_REG_MIPS_CP0_CONFIG3	32
MIPS	KVM_REG_MIPS_CP0_CONFIG4	32
MIPS	KVM_REG_MIPS_CP0_CONFIG5	32
MIPS	KVM_REG_MIPS_CP0_CONFIG7	32
MIPS	KVM_REG_MIPS_CP0_XCONTEXT	64
MIPS	KVM_REG_MIPS_CP0_ERROREPC	64
MIPS	KVM_REG_MIPS_CP0_KSCRATCH1	64
MIPS	KVM_REG_MIPS_CP0_KSCRATCH2	64
MIPS	KVM_REG_MIPS_CP0_KSCRATCH3	64
MIPS	KVM_REG_MIPS_CP0_KSCRATCH4	64
MIPS	KVM_REG_MIPS_CP0_KSCRATCH5	64
MIPS	KVM_REG_MIPS_CP0_KSCRATCH6	64
MIPS	KVM_REG_MIPS_CP0_MAAR(0..63)	64
MIPS	KVM_REG_MIPS_COUNT_CTL	64
MIPS	KVM_REG_MIPS_COUNT_RESUME	64
MIPS	KVM_REG_MIPS_COUNT_HZ	64
MIPS	KVM_REG_MIPS_FPR_32(0..31)	32
MIPS	KVM_REG_MIPS_FPR_64(0..31)	64
MIPS	KVM_REG_MIPS_VEC_128(0..31)	128
MIPS	KVM_REG_MIPS_FCR_IR	32
MIPS	KVM_REG_MIPS_FCR_CSR	32
MIPS	KVM_REG_MIPS_MSA_IR	32

continues on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
MIPS	KVM_REG_MIPS_MSA_CSR	32

ARM registers are mapped using the lower 32 bits. The upper 16 of that is the register group type, or coprocessor number:

ARM core registers have the following id bit patterns:

```
0x4020 0000 0010 <index into the kvm_regs struct:16>
```

ARM 32-bit CP15 registers have the following id bit patterns:

```
0x4020 0000 000F <zero:1> <crn:4> <crm:4> <opc1:4> <opc2:3>
```

ARM 64-bit CP15 registers have the following id bit patterns:

```
0x4030 0000 000F <zero:1> <zero:4> <crm:4> <opc1:4> <zero:3>
```

ARM CCSIDR registers are demultiplexed by CSSELR value:

```
0x4020 0000 0011 00 <csselr:8>
```

ARM 32-bit VFP control registers have the following id bit patterns:

```
0x4020 0000 0012 1 <regno:12>
```

ARM 64-bit FP registers have the following id bit patterns:

```
0x4030 0000 0012 0 <regno:12>
```

ARM firmware pseudo-registers have the following bit pattern:

```
0x4030 0000 0014 <regno:16>
```

arm64 registers are mapped using the lower 32 bits. The upper 16 of that is the register group type, or coprocessor number:

arm64 core/FP-SIMD registers have the following id bit patterns. Note that the size of the access is variable, as the kvm_regs structure contains elements ranging from 32 to 128 bits. The index is a 32bit value in the kvm_regs structure seen as a 32bit array:

```
0x60x0 0000 0010 <index into the kvm_regs struct:16>
```

Specifically:

Encoding	Register	Bits	kvm_regs member
0x6030 0000 0010 0000	X0	64	regs.regs[0]
0x6030 0000 0010 0002	X1	64	regs.regs[1]
...			
0x6030 0000 0010 003c	X30	64	regs.regs[30]
0x6030 0000 0010 003e	SP	64	regs.sp
0x6030 0000 0010 0040	PC	64	regs.pc
0x6030 0000 0010 0042	PSTATE	64	regs.pstate
0x6030 0000 0010 0044	SP_EL1	64	sp_el1
0x6030 0000 0010 0046	ELR_EL1	64	elr_el1
0x6030 0000 0010 0048	SPSR_EL1	64	spsr[KVM_SPSR_EL1] (alias SPSR_SVC)
0x6030 0000 0010 004a	SPSR_ABT	64	spsr[KVM_SPSR_ABT]
0x6030 0000 0010 004c	SPSR_UND	64	spsr[KVM_SPSR_UND]
0x6030 0000 0010 004e	SPSR_IRQ	64	spsr[KVM_SPSR_IRQ]
0x6060 0000 0010 0050	SPSR_FIQ	64	spsr[KVM_SPSR_FIQ]
0x6040 0000 0010 0054	V0	128	fp_regs.vregs[0][1]
0x6040 0000 0010 0058	V1	128	fp_regs.vregs[1]Page 54, 1
...			
0x6040 0000 0010 00d0	V31	128	fp_regs.vregs[31][1]
0x6020 0000 0010 00d4	FPSR	32	fp_regs.fpsr
0x6020 0000 0010 00d5	FPCR	32	fp_regs.fpcr

arm64 CCSIDR registers are demultiplexed by CSSELR value:

```
0x6020 0000 0011 00 <csselr:8>
```

arm64 system registers have the following id bit patterns:

```
0x6030 0000 0013 <op0:2> <op1:3> <crn:4> <crm:4> <op2:3>
```

Warning: Two system register IDs do not follow the specified pattern. These are KVM_REG_ARM_TIMER_CVAL and KVM_REG_ARM_TIMER_CNT, which map to system registers CNTV_CVAL_EL0 and CNTVCT_EL0 respectively. These two had their values accidentally swapped, which means TIMER_CVAL is derived from the register encoding for CNTVCT_EL0 and TIMER_CNT is derived from the register encoding for CNTV_CVAL_EL0. As this is API, it must remain this way.

arm64 firmware pseudo-registers have the following bit pattern:

```
0x6030 0000 0014 <regno:16>
```

arm64 SVE registers have the following bit patterns:

```
0x6080 0000 0015 00 <n:5> <slice:5>   Zn bits[2048*slice + 2047 : 2048*slice]
0x6050 0000 0015 04 <n:4> <slice:5>   Pn bits[256*slice + 255 : 256*slice]
```

[1] These encodings are not accepted for SVE-enabled vcpus. See KVM_ARM_VCPU_INIT.

The equivalent register content can be accessed via bits [127:0] of the corresponding SVE Zn registers instead for vcpus that have SVE enabled (see below).

```
0x6050 0000 0015 060 <slice:5>        FFR bits[256*slice + 255 : 256*slice]
0x6060 0000 0015 ffff                 KVM_REG_ARM64_SVE_VLS pseudo-register
```

Access to register IDs where 2048 * slice >= 128 * max_vq will fail with ENOENT. max_vq is the vcpu's maximum supported vector length in 128-bit quadwords: see[2] below.

These registers are only accessible on vcpus for which SVE is enabled. See KVM_ARM_VCPU_INIT for details.

In addition, except for KVM_REG_ARM64_SVE_VLS, these registers are not accessible until the vcpu's SVE configuration has been finalized using KVM_ARM_VCPU_FINALIZE(KVM_ARM_VCPU_SVE). See KVM_ARM_VCPU_INIT and KVM_ARM_VCPU_FINALIZE for more information about this procedure.

KVM_REG_ARM64_SVE_VLS is a pseudo-register that allows the set of vector lengths supported by the vcpu to be discovered and configured by userspace. When transferred to or from user memory via KVM_GET_ONE_REG or KVM_SET_ONE_REG, the value of this register is of type __u64[KVM_ARM64_SVE_VLS_WORDS], and encodes the set of vector lengths as follows:

```
__u64 vector_lengths[KVM_ARM64_SVE_VLS_WORDS];

if (vq >= SVE_VQ_MIN && vq <= SVE_VQ_MAX &&
    ((vector_lengths[(vq - KVM_ARM64_SVE_VQ_MIN) / 64] >>
              ((vq - KVM_ARM64_SVE_VQ_MIN) % 64)) & 1))
      /* Vector length vq * 16 bytes supported */
else
      /* Vector length vq * 16 bytes not supported */
```

(See Documentation/arch/arm64/sve.rst for an explanation of the ”vq” nomenclature.)

KVM_REG_ARM64_SVE_VLS is only accessible after KVM_ARM_VCPU_INIT. KVM_ARM_VCPU_INIT initialises it to the best set of vector lengths that the host supports.

Userspace may subsequently modify it if desired until the vcpu's SVE configuration is finalized using KVM_ARM_VCPU_FINALIZE(KVM_ARM_VCPU_SVE).

Apart from simply removing all vector lengths from the host set that exceed some value, support for arbitrarily chosen sets of vector lengths is hardware-dependent and may not be available. Attempting to configure an invalid set of vector lengths via KVM_SET_ONE_REG will fail with EINVAL.

After the vcpu's SVE configuration is finalized, further attempts to write this register will fail with EPERM.

arm64 bitmap feature firmware pseudo-registers have the following bit pattern:

```
0x6030 0000 0016 <regno:16>
```

The bitmap feature firmware registers exposes the hypercall services that are available for userspace to configure. The set bits corresponds to the services that are available for the guests to access. By default, KVM sets all the supported bits during VM initialization. The userspace can discover the available services via KVM_GET_ONE_REG, and write back the bitmap corresponding to the features that it wishes guests to see via KVM_SET_ONE_REG.

[2] The maximum value vq for which the above condition is true is max_vq. This is the maximum vector length available to the guest on this vcpu, and determines which register slices are visible through this ioctl interface.

Note: These registers are immutable once any of the vCPUs of the VM has run at least once. A KVM_SET_ONE_REG in such a scenario will return a -EBUSY to userspace.

(See *ARM Hypercall Interface* for more details.)

MIPS registers are mapped using the lower 32 bits. The upper 16 of that is the register group type:

MIPS core registers (see above) have the following id bit patterns:

```
0x7030 0000 0000 <reg:16>
```

MIPS CP0 registers (see KVM_REG_MIPS_CP0_* above) have the following id bit patterns depending on whether they're 32-bit or 64-bit registers:

```
0x7020 0000 0001 00 <reg:5> <sel:3>   (32-bit)
0x7030 0000 0001 00 <reg:5> <sel:3>   (64-bit)
```

Note: KVM_REG_MIPS_CP0_ENTRYLO0 and KVM_REG_MIPS_CP0_ENTRYLO1 are the MIPS64 versions of the EntryLo registers regardless of the word size of the host hardware, host kernel, guest, and whether XPA is present in the guest, i.e. with the RI and XI bits (if they exist) in bits 63 and 62 respectively, and the PFNX field starting at bit 30.

MIPS MAARs (see KVM_REG_MIPS_CP0_MAAR(*) above) have the following id bit patterns:

```
0x7030 0000 0001 01 <reg:8>
```

MIPS KVM control registers (see above) have the following id bit patterns:

```
0x7030 0000 0002 <reg:16>
```

MIPS FPU registers (see KVM_REG_MIPS_FPR_{32,64}() above) have the following id bit patterns depending on the size of the register being accessed. They are always accessed according to the current guest FPU mode (Status.FR and Config5.FRE), i.e. as the guest would see them, and they become unpredictable if the guest FPU mode is changed. MIPS SIMD Architecture (MSA) vector registers (see KVM_REG_MIPS_VEC_128() above) have similar patterns as they overlap the FPU registers:

```
0x7020 0000 0003 00 <0:3> <reg:5> (32-bit FPU registers)
0x7030 0000 0003 00 <0:3> <reg:5> (64-bit FPU registers)
0x7040 0000 0003 00 <0:3> <reg:5> (128-bit MSA vector registers)
```

MIPS FPU control registers (see KVM_REG_MIPS_FCR_{IR,CSR} above) have the following id bit patterns:

```
0x7020 0000 0003 01 <0:3> <reg:5>
```

MIPS MSA control registers (see KVM_REG_MIPS_MSA_{IR,CSR} above) have the following id bit patterns:

```
0x7020 0000 0003 02 <0:3> <reg:5>
```

RISC-V registers are mapped using the lower 32 bits. The upper 8 bits of that is the register group type.

RISC-V config registers are meant for configuring a Guest VCPU and it has the following id bit patterns:

```
0x8020 0000 01 <index into the kvm_riscv_config struct:24> (32bit Host)
0x8030 0000 01 <index into the kvm_riscv_config struct:24> (64bit Host)
```

Following are the RISC-V config registers:

Encoding	Register	Description
0x80x0 0000 0100 0000	isa	ISA feature bitmap of Guest VCPU

The isa config register can be read anytime but can only be written before a Guest VCPU runs. It will have ISA feature bits matching underlying host set by default.

RISC-V core registers represent the general execution state of a Guest VCPU and it has the following id bit patterns:

```
0x8020 0000 02 <index into the kvm_riscv_core struct:24> (32bit Host)
0x8030 0000 02 <index into the kvm_riscv_core struct:24> (64bit Host)
```

Following are the RISC-V core registers:

Encoding	Register	Description
0x80x0 0000 0200 0000	regs.pc	Program counter
0x80x0 0000 0200 0001	regs.ra	Return address
0x80x0 0000 0200 0002	regs.sp	Stack pointer
0x80x0 0000 0200 0003	regs.gp	Global pointer
0x80x0 0000 0200 0004	regs.tp	Task pointer
0x80x0 0000 0200 0005	regs.t0	Caller saved register 0
0x80x0 0000 0200 0006	regs.t1	Caller saved register 1
0x80x0 0000 0200 0007	regs.t2	Caller saved register 2
0x80x0 0000 0200 0008	regs.s0	Callee saved register 0
0x80x0 0000 0200 0009	regs.s1	Callee saved register 1
0x80x0 0000 0200 000a	regs.a0	Function argument (or return value) 0
0x80x0 0000 0200 000b	regs.a1	Function argument (or return value) 1
0x80x0 0000 0200 000c	regs.a2	Function argument 2
0x80x0 0000 0200 000d	regs.a3	Function argument 3
0x80x0 0000 0200 000e	regs.a4	Function argument 4
0x80x0 0000 0200 000f	regs.a5	Function argument 5
0x80x0 0000 0200 0010	regs.a6	Function argument 6
0x80x0 0000 0200 0011	regs.a7	Function argument 7
0x80x0 0000 0200 0012	regs.s2	Callee saved register 2
0x80x0 0000 0200 0013	regs.s3	Callee saved register 3
0x80x0 0000 0200 0014	regs.s4	Callee saved register 4
0x80x0 0000 0200 0015	regs.s5	Callee saved register 5
0x80x0 0000 0200 0016	regs.s6	Callee saved register 6
0x80x0 0000 0200 0017	regs.s7	Callee saved register 7
0x80x0 0000 0200 0018	regs.s8	Callee saved register 8
0x80x0 0000 0200 0019	regs.s9	Callee saved register 9
0x80x0 0000 0200 001a	regs.s10	Callee saved register 10

continues on next page

Table 2 – continued from previous page

Encoding	Register	Description
0x80x0 0000 0200 001b	regs.s11	Callee saved register 11
0x80x0 0000 0200 001c	regs.t3	Caller saved register 3
0x80x0 0000 0200 001d	regs.t4	Caller saved register 4
0x80x0 0000 0200 001e	regs.t5	Caller saved register 5
0x80x0 0000 0200 001f	regs.t6	Caller saved register 6
0x80x0 0000 0200 0020	mode	Privilege mode (1 = S-mode or 0 = U-mode)

RISC-V csr registers represent the supervisor mode control/status registers of a Guest VCPU and it has the following id bit patterns:

```
0x8020 0000 03 <index into the kvm_riscv_csr struct:24> (32bit Host)
0x8030 0000 03 <index into the kvm_riscv_csr struct:24> (64bit Host)
```

Following are the RISC-V csr registers:

Encoding	Register	Description
0x80x0 0000 0300 0000	sstatus	Supervisor status
0x80x0 0000 0300 0001	sie	Supervisor interrupt enable
0x80x0 0000 0300 0002	stvec	Supervisor trap vector base
0x80x0 0000 0300 0003	sscratch	Supervisor scratch register
0x80x0 0000 0300 0004	sepc	Supervisor exception program counter
0x80x0 0000 0300 0005	scause	Supervisor trap cause
0x80x0 0000 0300 0006	stval	Supervisor bad address or instruction
0x80x0 0000 0300 0007	sip	Supervisor interrupt pending
0x80x0 0000 0300 0008	satp	Supervisor address translation and protection

RISC-V timer registers represent the timer state of a Guest VCPU and it has the following id bit patterns:

```
0x8030 0000 04 <index into the kvm_riscv_timer struct:24>
```

Following are the RISC-V timer registers:

Encoding	Register	Description
0x8030 0000 0400 0000	frequency	Time base frequency (read-only)
0x8030 0000 0400 0001	time	Time value visible to Guest
0x8030 0000 0400 0002	compare	Time compare programmed by Guest
0x8030 0000 0400 0003	state	Time compare state (1 = ON or 0 = OFF)

RISC-V F-extension registers represent the single precision floating point state of a Guest VCPU and it has the following id bit patterns:

```
0x8020 0000 05 <index into the __riscv_f_ext_state struct:24>
```

Following are the RISC-V F-extension registers:

Encoding	Register	Description
0x8020 0000 0500 0000	f[0]	Floating point register 0
...		
0x8020 0000 0500 001f	f[31]	Floating point register 31
0x8020 0000 0500 0020	fcsr	Floating point control and status register

RISC-V D-extension registers represent the double precision floating point state of a Guest VCPU and it has the following id bit patterns:

```
0x8020 0000 06 <index into the __riscv_d_ext_state struct:24> (fcsr)
0x8030 0000 06 <index into the __riscv_d_ext_state struct:24> (non-fcsr)
```

Following are the RISC-V D-extension registers:

Encoding	Register	Description
0x8030 0000 0600 0000	f[0]	Floating point register 0
...		
0x8030 0000 0600 001f	f[31]	Floating point register 31
0x8020 0000 0600 0020	fcsr	Floating point control and status register

LoongArch registers are mapped using the lower 32 bits. The upper 16 bits of that is the register group type.

LoongArch csr registers are used to control guest cpu or get status of guest cpu, and they have the following id bit patterns:

```
0x9030 0000 0001 00 <reg:5> <sel:3>   (64-bit)
```

LoongArch KVM control registers are used to implement some new defined functions such as set vcpu counter or reset vcpu, and they have the following id bit patterns:

```
0x9030 0000 0002 <reg:16>
```

4.69 KVM_GET_ONE_REG

Capability
KVM_CAP_ONE_REG

Architectures
all

Type
vcpu ioctl

Parameters
struct kvm_one_reg (in and out)

Returns
0 on success, negative value on failure

Errors include:

ENOENT	no such register
EINVAL	invalid register ID, or no such register or used with VMs in protected virtualization mode on s390
EPERM	(arm64) register access not allowed before vcpu finalization

(These error codes are indicative only: do not rely on a specific error code being returned in a specific situation.)

This ioctl allows to receive the value of a single register implemented in a vcpu. The register to read is indicated by the ”id” field of the kvm_one_reg struct passed in. On success, the register value can be found at the memory location pointed to by ”addr”.

The list of registers accessible using this interface is identical to the list in 4.68.

4.70 KVM_KVMCLOCK_CTRL

Capability
KVM_CAP_KVMCLOCK_CTRL

Architectures
Any that implement pvclocks (currently x86 only)

Type
vcpu ioctl

Parameters
None

Returns
0 on success, -1 on error

This ioctl sets a flag accessible to the guest indicating that the specified vCPU has been paused by the host userspace.

The host will set a flag in the pvclock structure that is checked from the soft lockup watchdog. The flag is part of the pvclock structure that is shared between guest and host, specifically the second bit of the flags field of the pvclock_vcpu_time_info structure. It will be set exclusively by the host and read/cleared exclusively by the guest. The guest operation of checking and clearing the flag must be an atomic operation so load-link/store-conditional, or equivalent must be used. There are two cases where the guest will clear the flag: when the soft lockup watchdog timer resets itself or when a soft lockup is detected. This ioctl can be called any time after pausing the vcpu, but before it is resumed.

4.71 KVM_SIGNAL_MSI

Capability
KVM_CAP_SIGNAL_MSI

Architectures
x86 arm64

Type
vm ioctl

Parameters
struct kvm_msi (in)

Returns
>0 on delivery, 0 if guest blocked the MSI, and -1 on error

Directly inject a MSI message. Only valid with in-kernel irqchip that handles MSI messages.

```
struct kvm_msi {
      __u32 address_lo;
      __u32 address_hi;
      __u32 data;
      __u32 flags;
      __u32 devid;
      __u8  pad[12];
};
```

flags:
KVM_MSI_VALID_DEVID: devid contains a valid value. The per-VM KVM_CAP_MSI_DEVID capability advertises the requirement to provide the device ID. If this capability is not available, userspace should never set the KVM_MSI_VALID_DEVID flag as the ioctl might fail.

If KVM_MSI_VALID_DEVID is set, devid contains a unique device identifier for the device that wrote the MSI message. For PCI, this is usually a BFD identifier in the lower 16 bits.

On x86, address_hi is ignored unless the KVM_X2APIC_API_USE_32BIT_IDS feature of KVM_CAP_X2APIC_API capability is enabled. If it is enabled, address_hi bits 31-8 provide bits 31-8 of the destination id. Bits 7-0 of address_hi must be zero.

4.71 KVM_CREATE_PIT2

Capability
KVM_CAP_PIT2

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_pit_config (in)

Returns
0 on success, -1 on error

Creates an in-kernel device model for the i8254 PIT. This call is only valid after enabling in-kernel irqchip support via KVM_CREATE_IRQCHIP. The following parameters have to be passed:

```
struct kvm_pit_config {
      __u32 flags;
      __u32 pad[15];
};
```

Valid flags are:

```
#define KVM_PIT_SPEAKER_DUMMY     1 /* emulate speaker port stub */
```

PIT timer interrupts may use a per-VM kernel thread for injection. If it exists, this thread will have a name of the following pattern:

```
kvm-pit/<owner-process-pid>
```

When running a guest with elevated priorities, the scheduling parameters of this thread may have to be adjusted accordingly.

This IOCTL replaces the obsolete KVM_CREATE_PIT.

4.72 KVM_GET_PIT2

Capability
KVM_CAP_PIT_STATE2

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_pit_state2 (out)

Returns
0 on success, -1 on error

Retrieves the state of the in-kernel PIT model. Only valid after KVM_CREATE_PIT2. The state is returned in the following structure:

```
struct kvm_pit_state2 {
      struct kvm_pit_channel_state channels[3];
      __u32 flags;
      __u32 reserved[9];
};
```

Valid flags are:

```
/* disable PIT in HPET legacy mode */
#define KVM_PIT_FLAGS_HPET_LEGACY     0x00000001
/* speaker port data bit enabled */
#define KVM_PIT_FLAGS_SPEAKER_DATA_ON 0x00000002
```

This IOCTL replaces the obsolete KVM_GET_PIT.

4.73 KVM_SET_PIT2

Capability
KVM_CAP_PIT_STATE2

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_pit_state2 (in)

Returns
0 on success, -1 on error

Sets the state of the in-kernel PIT model. Only valid after KVM_CREATE_PIT2. See KVM_GET_PIT2 for details on struct kvm_pit_state2.

This IOCTL replaces the obsolete KVM_SET_PIT.

4.74 KVM_PPC_GET_SMMU_INFO

Capability
KVM_CAP_PPC_GET_SMMU_INFO

Architectures
powerpc

Type
vm ioctl

Parameters
None

Returns
0 on success, -1 on error

This populates and returns a structure describing the features of the "Server" class MMU emulation supported by KVM. This can in turn be used by userspace to generate the appropriate device-tree properties for the guest operating system.

The structure contains some global information, followed by an array of supported segment page sizes:

```
struct kvm_ppc_smmu_info {
       __u64 flags;
```

```
        __u32 slb_size;
        __u32 pad;
        struct kvm_ppc_one_seg_page_size sps[KVM_PPC_PAGE_SIZES_MAX_SZ];
};
```

The supported flags are:

- **KVM_PPC_PAGE_SIZES_REAL:**
  When that flag is set, guest page sizes must ”fit” the backing store page sizes. When not set, any page size in the list can be used regardless of how they are backed by userspace.
- **KVM_PPC_1T_SEGMENTS**
  The emulated MMU supports 1T segments in addition to the standard 256M ones.
- **KVM_PPC_NO_HASH**
  This flag indicates that HPT guests are not supported by KVM, thus all guests must use radix MMU mode.

The ”slb_size” field indicates how many SLB entries are supported

The ”sps” array contains 8 entries indicating the supported base page sizes for a segment in increasing order. Each entry is defined as follow:

```
struct kvm_ppc_one_seg_page_size {
     __u32 page_shift;       /* Base page shift of segment (or 0) */
     __u32 slb_enc;          /* SLB encoding for BookS */
     struct kvm_ppc_one_page_size enc[KVM_PPC_PAGE_SIZES_MAX_SZ];
};
```

An entry with a ”page_shift” of 0 is unused. Because the array is organized in increasing order, a lookup can stop when encountering such an entry.

The ”slb_enc” field provides the encoding to use in the SLB for the page size. The bits are in positions such as the value can directly be OR'ed into the ”vsid” argument of the slbmte instruction.

The ”enc” array is a list which for each of those segment base page size provides the list of supported actual page sizes (which can be only larger or equal to the base page size), along with the corresponding encoding in the hash PTE. Similarly, the array is 8 entries sorted by increasing sizes and an entry with a ”0” shift is an empty entry and a terminator:

```
struct kvm_ppc_one_page_size {
     __u32 page_shift;       /* Page shift (or 0) */
     __u32 pte_enc;          /* Encoding in the HPTE (>>12) */
};
```

The ”pte_enc” field provides a value that can OR'ed into the hash PTE's RPN field (ie, it needs to be shifted left by 12 to OR it into the hash PTE second double word).

4.75 KVM_IRQFD

Capability
KVM_CAP_IRQFD

Architectures
x86 s390 arm64

Type
vm ioctl

Parameters
struct kvm_irqfd (in)

Returns
0 on success, -1 on error

Allows setting an eventfd to directly trigger a guest interrupt. kvm_irqfd.fd specifies the file descriptor to use as the eventfd and kvm_irqfd.gsi specifies the irqchip pin toggled by this event. When an event is triggered on the eventfd, an interrupt is injected into the guest using the specified gsi pin. The irqfd is removed using the KVM_IRQFD_FLAG_DEASSIGN flag, specifying both kvm_irqfd.fd and kvm_irqfd.gsi.

With KVM_CAP_IRQFD_RESAMPLE, KVM_IRQFD supports a de-assert and notify mechanism allowing emulation of level-triggered, irqfd-based interrupts. When KVM_IRQFD_FLAG_RESAMPLE is set the user must pass an additional eventfd in the kvm_irqfd.resamplefd field. When operating in resample mode, posting of an interrupt through kvm_irq.fd asserts the specified gsi in the irqchip. When the irqchip is resampled, such as from an EOI, the gsi is de-asserted and the user is notified via kvm_irqfd.resamplefd. It is the user's responsibility to re-queue the interrupt if the device making use of it still requires service. Note that closing the resamplefd is not sufficient to disable the irqfd. The KVM_IRQFD_FLAG_RESAMPLE is only necessary on assignment and need not be specified with KVM_IRQFD_FLAG_DEASSIGN.

On arm64, gsi routing being supported, the following can happen:

- in case no routing entry is associated to this gsi, injection fails
- in case the gsi is associated to an irqchip routing entry, irqchip.pin + 32 corresponds to the injected SPI ID.
- in case the gsi is associated to an MSI routing entry, the MSI message and device ID are translated into an LPI (support restricted to GICv3 ITS in-kernel emulation).

4.76 KVM_PPC_ALLOCATE_HTAB

Capability
KVM_CAP_PPC_ALLOC_HTAB

Architectures
powerpc

Type
vm ioctl

Parameters
Pointer to u32 containing hash table order (in/out)

Returns
0 on success, -1 on error

This requests the host kernel to allocate an MMU hash table for a guest using the PAPR paravirtualization interface. This only does anything if the kernel is configured to use the Book 3S HV style of virtualization. Otherwise the capability doesn't exist and the ioctl returns an ENOTTY error. The rest of this description assumes Book 3S HV.

There must be no vcpus running when this ioctl is called; if there are, it will do nothing and return an EBUSY error.

The parameter is a pointer to a 32-bit unsigned integer variable containing the order (log base 2) of the desired size of the hash table, which must be between 18 and 46. On successful return from the ioctl, the value will not be changed by the kernel.

If no hash table has been allocated when any vcpu is asked to run (with the KVM_RUN ioctl), the host kernel will allocate a default-sized hash table (16 MB).

If this ioctl is called when a hash table has already been allocated, with a different order from the existing hash table, the existing hash table will be freed and a new one allocated. If this is ioctl is called when a hash table has already been allocated of the same order as specified, the kernel will clear out the existing hash table (zero all HPTEs). In either case, if the guest is using the virtualized real-mode area (VRMA) facility, the kernel will re-create the VMRA HPTEs on the next KVM_RUN of any vcpu.

4.77 KVM_S390_INTERRUPT

Capability
basic

Architectures
s390

Type
vm ioctl, vcpu ioctl

Parameters
struct kvm_s390_interrupt (in)

Returns
0 on success, -1 on error

Allows to inject an interrupt to the guest. Interrupts can be floating (vm ioctl) or per cpu (vcpu ioctl), depending on the interrupt type.

Interrupt parameters are passed via kvm_s390_interrupt:

```
struct kvm_s390_interrupt {
      __u32 type;
      __u32 parm;
      __u64 parm64;
};
```

type can be one of the following:

KVM_S390_SIGP_STOP (vcpu)

- sigp stop; optional flags in parm

KVM_S390_PROGRAM_INT (vcpu)

- program check; code in parm

KVM_S390_SIGP_SET_PREFIX (vcpu)

- sigp set prefix; prefix address in parm

KVM_S390_RESTART (vcpu)

- restart

KVM_S390_INT_CLOCK_COMP (vcpu)

- clock comparator interrupt

KVM_S390_INT_CPU_TIMER (vcpu)

- CPU timer interrupt

KVM_S390_INT_VIRTIO (vm)

- virtio external interrupt; external interrupt parameters in parm and parm64

KVM_S390_INT_SERVICE (vm)

- sclp external interrupt; sclp parameter in parm

KVM_S390_INT_EMERGENCY (vcpu)

- sigp emergency; source cpu in parm

KVM_S390_INT_EXTERNAL_CALL (vcpu)

- sigp external call; source cpu in parm

KVM_S390_INT_IO(ai,cssid,ssid,schid) (vm)

- compound value to indicate an I/O interrupt (ai - adapter interrupt; cssid,ssid,schid - subchannel); I/O interruption parameters in parm (subchannel) and parm64 (intparm, interruption subclass)

KVM_S390_MCHK (vm, vcpu)

- machine check interrupt; cr 14 bits in parm, machine check interrupt code in parm64 (note that machine checks needing further payload are not supported by this ioctl)

This is an asynchronous vcpu ioctl and can be invoked from any thread.

4.78 KVM_PPC_GET_HTAB_FD

Capability
KVM_CAP_PPC_HTAB_FD

Architectures
powerpc

Type
vm ioctl

Parameters
Pointer to struct kvm_get_htab_fd (in)

Returns
file descriptor number (>= 0) on success, -1 on error

This returns a file descriptor that can be used either to read out the entries in the guest's hashed page table (HPT), or to write entries to initialize the HPT. The returned fd can only be written to if the KVM_GET_HTAB_WRITE bit is set in the flags field of the argument, and can only be read if that bit is clear. The argument struct looks like this:

```
/* For KVM_PPC_GET_HTAB_FD */
struct kvm_get_htab_fd {
      __u64   flags;
      __u64   start_index;
      __u64   reserved[2];
};

/* Values for kvm_get_htab_fd.flags */
#define KVM_GET_HTAB_BOLTED_ONLY      ((__u64)0x1)
#define KVM_GET_HTAB_WRITE            ((__u64)0x2)
```

The 'start_index' field gives the index in the HPT of the entry at which to start reading. It is ignored when writing.

Reads on the fd will initially supply information about all ”interesting” HPT entries. Interesting entries are those with the bolted bit set, if the KVM_GET_HTAB_BOLTED_ONLY bit is set, otherwise all entries. When the end of the HPT is reached, the read() will return. If read() is called again on the fd, it will start again from the beginning of the HPT, but will only return HPT entries that have changed since they were last read.

Data read or written is structured as a header (8 bytes) followed by a series of valid HPT entries (16 bytes) each. The header indicates how many valid HPT entries there are and how many invalid entries follow the valid entries. The invalid entries are not represented explicitly in the stream. The header format is:

```
struct kvm_get_htab_header {
      __u32   index;
      __u16   n_valid;
      __u16   n_invalid;
};
```

Writes to the fd create HPT entries starting at the index given in the header; first 'n_valid' valid entries with contents from the data written, then 'n_invalid' invalid entries, invalidating any previously valid entries found.

4.79 KVM_CREATE_DEVICE

Capability
KVM_CAP_DEVICE_CTRL

Architectures
all

Type
vm ioctl

Parameters
struct kvm_create_device (in/out)

Returns
0 on success, -1 on error

Errors:

ENODEV	The device type is unknown or unsupported
EEXIST	Device already created, and this type of device may not be instantiated multiple times

Other error conditions may be defined by individual device types or have their standard meanings.

Creates an emulated device in the kernel. The file descriptor returned in fd can be used with KVM_SET/GET/HAS_DEVICE_ATTR.

If the KVM_CREATE_DEVICE_TEST flag is set, only test whether the device type is supported (not necessarily whether it can be created in the current vm).

Individual devices should not define flags. Attributes should be used for specifying any behavior that is not implied by the device type number.

```
struct kvm_create_device {
      __u32   type;   /* in: KVM_DEV_TYPE_xxx */
      __u32   fd;     /* out: device handle */
      __u32   flags;  /* in: KVM_CREATE_DEVICE_xxx */
};
```

4.80 KVM_SET_DEVICE_ATTR/KVM_GET_DEVICE_ATTR

Capability
KVM_CAP_DEVICE_CTRL, KVM_CAP_VM_ATTRIBUTES for vm device, KVM_CAP_VCPU_ATTRIBUTES for vcpu device KVM_CAP_SYS_ATTRIBUTES for system (/dev/kvm) device (no set)

Architectures
x86, arm64, s390

Type
device ioctl, vm ioctl, vcpu ioctl

Parameters
struct kvm_device_attr

Returns
0 on success, -1 on error

Errors:

ENXIO	The group or attribute is unknown/unsupported for this device or hardware support is missing.
EPERM	The attribute cannot (currently) be accessed this way (e.g. read-only attribute, or attribute that only makes sense when the device is in a different state)

Other error conditions may be defined by individual device types.

Gets/sets a specified piece of device configuration and/or state. The semantics are device-specific. See individual device documentation in the ”devices” directory. As with ONE_REG, the size of the data transferred is defined by the particular attribute.

```
struct kvm_device_attr {
      __u32   flags;          /* no flags currently defined */
      __u32   group;          /* device-defined */
      __u64   attr;           /* group-defined */
      __u64   addr;           /* userspace address of attr data */
};
```

4.81 KVM_HAS_DEVICE_ATTR

Capability
KVM_CAP_DEVICE_CTRL, KVM_CAP_VM_ATTRIBUTES for vm device, KVM_CAP_VCPU_ATTRIBUTES for vcpu device KVM_CAP_SYS_ATTRIBUTES for system (/dev/kvm) device

Type
device ioctl, vm ioctl, vcpu ioctl

Parameters
struct kvm_device_attr

Returns
0 on success, -1 on error

Errors:

ENXIO	The group or attribute is unknown/unsupported for this device or hardware support is missing.

Tests whether a device supports a particular attribute. A successful return indicates the attribute is implemented. It does not necessarily indicate that the attribute can be read or written in the device's current state. ”addr” is ignored.

4.82 KVM_ARM_VCPU_INIT

Capability
basic

Architectures
arm64

Type
vcpu ioctl

Parameters
struct kvm_vcpu_init (in)

Returns
0 on success; -1 on error

Errors:

EINVAL	the target is unknown, or the combination of features is invalid.
ENOENT	a features bit specified is unknown.

This tells KVM what type of CPU to present to the guest, and what optional features it should have. This will cause a reset of the cpu registers to their initial values. If this is not called, KVM_RUN will return ENOEXEC for that vcpu.

The initial values are defined as:

- **Processor state:**
  - AArch64: EL1h, D, A, I and F bits set. All other bits are cleared.
  - AArch32: SVC, A, I and F bits set. All other bits are cleared.
- General Purpose registers, including PC and SP: set to 0
- FPSIMD/NEON registers: set to 0
- SVE registers: set to 0
- System registers: Reset to their architecturally defined values as for a warm reset to EL1 (resp. SVC)

Note that because some registers reflect machine topology, all vcpus should be created before this ioctl is invoked.

Userspace can call this function multiple times for a given vcpu, including after the vcpu has been run. This will reset the vcpu to its initial state. All calls to this function after the initial call must use the same target and same set of feature flags, otherwise EINVAL will be returned.

Possible features:

- KVM_ARM_VCPU_POWER_OFF: Starts the CPU in a power-off state. Depends on KVM_CAP_ARM_PSCI. If not set, the CPU will be powered on and execute guest code when KVM_RUN is called.
- KVM_ARM_VCPU_EL1_32BIT: Starts the CPU in a 32bit mode. Depends on KVM_CAP_ARM_EL1_32BIT (arm64 only).

- KVM_ARM_VCPU_PSCI_0_2: Emulate PSCI v0.2 (or a future revision backward compatible with v0.2) for the CPU. Depends on KVM_CAP_ARM_PSCI_0_2.
- KVM_ARM_VCPU_PMU_V3: Emulate PMUv3 for the CPU. Depends on KVM_CAP_ARM_PMU_V3.
- KVM_ARM_VCPU_PTRAUTH_ADDRESS: Enables Address Pointer authentication for arm64 only. Depends on KVM_CAP_ARM_PTRAUTH_ADDRESS. If KVM_CAP_ARM_PTRAUTH_ADDRESS and KVM_CAP_ARM_PTRAUTH_GENERIC are both present, then both KVM_ARM_VCPU_PTRAUTH_ADDRESS and KVM_ARM_VCPU_PTRAUTH_GENERIC must be requested or neither must be requested.
- KVM_ARM_VCPU_PTRAUTH_GENERIC: Enables Generic Pointer authentication for arm64 only. Depends on KVM_CAP_ARM_PTRAUTH_GENERIC. If KVM_CAP_ARM_PTRAUTH_ADDRESS and KVM_CAP_ARM_PTRAUTH_GENERIC are both present, then both KVM_ARM_VCPU_PTRAUTH_ADDRESS and KVM_ARM_VCPU_PTRAUTH_GENERIC must be requested or neither must be requested.
- KVM_ARM_VCPU_SVE: Enables SVE for the CPU (arm64 only). Depends on KVM_CAP_ARM_SVE. Requires KVM_ARM_VCPU_FINALIZE(KVM_ARM_VCPU_SVE):
  - After KVM_ARM_VCPU_INIT:
    - KVM_REG_ARM64_SVE_VLS may be read using KVM_GET_ONE_REG: the initial value of this pseudo-register indicates the best set of vector lengths possible for a vcpu on this host.
  - Before KVM_ARM_VCPU_FINALIZE(KVM_ARM_VCPU_SVE):
    - KVM_RUN and KVM_GET_REG_LIST are not available;
    - KVM_GET_ONE_REG and KVM_SET_ONE_REG cannot be used to access the scalable architectural SVE registers KVM_REG_ARM64_SVE_ZREG(), KVM_REG_ARM64_SVE_PREG() or KVM_REG_ARM64_SVE_FFR;
    - KVM_REG_ARM64_SVE_VLS may optionally be written using KVM_SET_ONE_REG, to modify the set of vector lengths available for the vcpu.
  - After KVM_ARM_VCPU_FINALIZE(KVM_ARM_VCPU_SVE):
    - the KVM_REG_ARM64_SVE_VLS pseudo-register is immutable, and can no longer be written using KVM_SET_ONE_REG.

4.83 KVM_ARM_PREFERRED_TARGET

Capability
basic

Architectures
arm64

Type
vm ioctl

Parameters
struct kvm_vcpu_init (out)

Returns
0 on success; -1 on error

Errors:

ENODEV	no preferred target available for the host

This queries KVM for preferred CPU target type which can be emulated by KVM on underlying host.

The ioctl returns struct kvm_vcpu_init instance containing information about preferred CPU target type and recommended features for it. The kvm_vcpu_init->features bitmap returned will have feature bits set if the preferred target recommends setting these features, but this is not mandatory.

The information returned by this ioctl can be used to prepare an instance of struct kvm_vcpu_init for KVM_ARM_VCPU_INIT ioctl which will result in VCPU matching underlying host.

4.84 KVM_GET_REG_LIST

Capability
basic

Architectures
arm64, mips, riscv

Type
vcpu ioctl

Parameters
struct kvm_reg_list (in/out)

Returns
0 on success; -1 on error

Errors:

E2BIG	the reg index list is too big to fit in the array specified by the user (the number required will be written into n).

```
struct kvm_reg_list {
      __u64 n; /* number of registers in reg[] */
      __u64 reg[0];
};
```

This ioctl returns the guest registers that are supported for the KVM_GET_ONE_REG/KVM_SET_ONE_REG calls.

4.85 KVM_ARM_SET_DEVICE_ADDR (deprecated)

Capability
KVM_CAP_ARM_SET_DEVICE_ADDR

Architectures
arm64

Type
vm ioctl

Parameters
struct kvm_arm_device_address (in)

Returns
0 on success, -1 on error

Errors:

ENODEV	The device id is unknown
ENXIO	Device not supported on current system
EEXIST	Address already set
E2BIG	Address outside guest physical address space
EBUSY	Address overlaps with other device range

```
struct kvm_arm_device_addr {
      __u64 id;
      __u64 addr;
};
```

Specify a device address in the guest's physical address space where guests can access emulated or directly exposed devices, which the host kernel needs to know about. The id field is an architecture specific identifier for a specific device.

arm64 divides the id field into two parts, a device id and an address type id specific to the individual device:

```
bits:  | 63        ...       32 | 31    ...    16 | 15    ...    0 |
field: |        0x00000000      |     device id   |  addr type id  |
```

arm64 currently only require this when using the in-kernel GIC support for the hardware VGIC features, using KVM_ARM_DEVICE_VGIC_V2 as the device id. When setting the base address for the guest's mapping of the VGIC virtual CPU and distributor interface, the ioctl must be called after calling KVM_CREATE_IRQCHIP, but before calling KVM_RUN on any of the VCPUs. Calling this ioctl twice for any of the base addresses will return -EEXIST.

Note, this IOCTL is deprecated and the more flexible SET/GET_DEVICE_ATTR API should be used instead.

4.86 KVM_PPC_RTAS_DEFINE_TOKEN

Capability
KVM_CAP_PPC_RTAS

Architectures
ppc

Type
vm ioctl

Parameters
struct kvm_rtas_token_args

Returns
0 on success, -1 on error

Defines a token value for a RTAS (Run Time Abstraction Services) service in order to allow it to be handled in the kernel. The argument struct gives the name of the service, which must be the name of a service that has a kernel-side implementation. If the token value is non-zero, it will be associated with that service, and subsequent RTAS calls by the guest specifying that token will be handled by the kernel. If the token value is 0, then any token associated with the service will be forgotten, and subsequent RTAS calls by the guest for that service will be passed to userspace to be handled.

4.87 KVM_SET_GUEST_DEBUG

Capability
KVM_CAP_SET_GUEST_DEBUG

Architectures
x86, s390, ppc, arm64

Type
vcpu ioctl

Parameters
struct kvm_guest_debug (in)

Returns
0 on success; -1 on error

```
struct kvm_guest_debug {
     __u32 control;
     __u32 pad;
     struct kvm_guest_debug_arch arch;
};
```

Set up the processor specific debug registers and configure vcpu for handling guest debug events. There are two parts to the structure, the first a control bitfield indicates the type of debug events to handle when running. Common control bits are:

- KVM_GUESTDBG_ENABLE: guest debugging is enabled
- KVM_GUESTDBG_SINGLESTEP: the next run should single-step

The top 16 bits of the control field are architecture specific control flags which can include the following:

- KVM_GUESTDBG_USE_SW_BP: using software breakpoints [x86, arm64]
- KVM_GUESTDBG_USE_HW_BP: using hardware breakpoints [x86, s390]
- KVM_GUESTDBG_USE_HW: using hardware debug events [arm64]
- KVM_GUESTDBG_INJECT_DB: inject DB type exception [x86]
- KVM_GUESTDBG_INJECT_BP: inject BP type exception [x86]
- KVM_GUESTDBG_EXIT_PENDING: trigger an immediate guest exit [s390]
- KVM_GUESTDBG_BLOCKIRQ: avoid injecting interrupts/NMI/SMI [x86]

For example KVM_GUESTDBG_USE_SW_BP indicates that software breakpoints are enabled in memory so we need to ensure breakpoint exceptions are correctly trapped and the KVM run loop exits at the breakpoint and not running off into the normal guest vector. For KVM_GUESTDBG_USE_HW_BP we need to ensure the guest vCPUs architecture specific registers are updated to the correct (supplied) values.

The second part of the structure is architecture specific and typically contains a set of debug registers.

For arm64 the number of debug registers is implementation defined and can be determined by querying the KVM_CAP_GUEST_DEBUG_HW_BPS and KVM_CAP_GUEST_DEBUG_HW_WPS capabilities which return a positive number indicating the number of supported registers.

For ppc, the KVM_CAP_PPC_GUEST_DEBUG_SSTEP capability indicates whether the single-step debug event (KVM_GUESTDBG_SINGLESTEP) is supported.

Also when supported, KVM_CAP_SET_GUEST_DEBUG2 capability indicates the supported KVM_GUESTDBG_* bits in the control field.

When debug events exit the main run loop with the reason KVM_EXIT_DEBUG with the kvm_debug_exit_arch part of the kvm_run structure containing architecture specific debug information.

4.88 KVM_GET_EMULATED_CPUID

Capability
KVM_CAP_EXT_EMUL_CPUID

Architectures
x86

Type
system ioctl

Parameters
struct kvm_cpuid2 (in/out)

Returns
0 on success, -1 on error

```
struct kvm_cpuid2 {
      __u32 nent;
```

```
      __u32 flags;
      struct kvm_cpuid_entry2 entries[0];
};
```

The member 'flags' is used for passing flags from userspace.

```
#define KVM_CPUID_FLAG_SIGNIFCANT_INDEX               BIT(0)
#define KVM_CPUID_FLAG_STATEFUL_FUNC          BIT(1) /* deprecated */
#define KVM_CPUID_FLAG_STATE_READ_NEXT                BIT(2) /* deprecated */

struct kvm_cpuid_entry2 {
      __u32 function;
      __u32 index;
      __u32 flags;
      __u32 eax;
      __u32 ebx;
      __u32 ecx;
      __u32 edx;
      __u32 padding[3];
};
```

This ioctl returns x86 cpuid features which are emulated by kvm.Userspace can use the information returned by this ioctl to query which features are emulated by kvm instead of being present natively.

Userspace invokes KVM_GET_EMULATED_CPUID by passing a kvm_cpuid2 structure with the 'nent' field indicating the number of entries in the variable-size array 'entries'. If the number of entries is too low to describe the cpu capabilities, an error (E2BIG) is returned. If the number is too high, the 'nent' field is adjusted and an error (ENOMEM) is returned. If the number is just right, the 'nent' field is adjusted to the number of valid entries in the 'entries' array, which is then filled.

The entries returned are the set CPUID bits of the respective features which kvm emulates, as returned by the CPUID instruction, with unknown or unsupported feature bits cleared.

Features like x2apic, for example, may not be present in the host cpu but are exposed by kvm in KVM_GET_SUPPORTED_CPUID because they can be emulated efficiently and thus not included here.

The fields in each entry are defined as follows:

function:
the eax value used to obtain the entry

index:
the ecx value used to obtain the entry (for entries that are affected by ecx)

flags:

an OR of zero or more of the following:

KVM_CPUID_FLAG_SIGNIFCANT_INDEX:
if the index field is valid

eax, ebx, ecx, edx:

the values returned by the cpuid instruction for this function/index combination

4.89 KVM_S390_MEM_OP

Capability
KVM_CAP_S390_MEM_OP, KVM_CAP_S390_PROTECTED, KVM_CAP_S390_MEM_OP_EXTENSION

Architectures
s390

Type
vm ioctl, vcpu ioctl

Parameters
struct kvm_s390_mem_op (in)

Returns
= 0 on success, < 0 on generic error (e.g. -EFAULT or -ENOMEM), 16 bit program exception code if the access causes such an exception

Read or write data from/to the VM's memory. The KVM_CAP_S390_MEM_OP_EXTENSION capability specifies what functionality is supported.

Parameters are specified via the following structure:

```
struct kvm_s390_mem_op {
      __u64 gaddr;            /* the guest address */
      __u64 flags;            /* flags */
      __u32 size;             /* amount of bytes */
      __u32 op;               /* type of operation */
      __u64 buf;              /* buffer in userspace */
      union {
              struct {
                      __u8 ar;        /* the access register number */
                      __u8 key;       /* access key, ignored if flag unset */
                      __u8 pad1[6];   /* ignored */
                      __u64 old_addr; /* ignored if flag unset */
              };
              __u32 sida_offset; /* offset into the sida */
              __u8 reserved[32]; /* ignored */
      };
};
```

The start address of the memory region has to be specified in the ”gaddr” field, and the length of the region in the ”size” field (which must not be 0). The maximum value for ”size” can be obtained by checking the KVM_CAP_S390_MEM_OP capability. ”buf” is the buffer supplied by the userspace application where the read data should be written to for a read access, or where the data that should be written is stored for a write access. The ”reserved” field is meant for future extensions. Reserved and unused values are ignored. Future extension that add members must introduce new flags.

The type of operation is specified in the ”op” field. Flags modifying their behavior can be set in the ”flags” field. Undefined flag bits must be set to 0.

Possible operations are:

- KVM_S390_MEMOP_LOGICAL_READ
- KVM_S390_MEMOP_LOGICAL_WRITE
- KVM_S390_MEMOP_ABSOLUTE_READ
- KVM_S390_MEMOP_ABSOLUTE_WRITE
- KVM_S390_MEMOP_SIDA_READ
- KVM_S390_MEMOP_SIDA_WRITE
- KVM_S390_MEMOP_ABSOLUTE_CMPXCHG

Logical read/write:

Access logical memory, i.e. translate the given guest address to an absolute address given the state of the VCPU and use the absolute address as target of the access. ”ar” designates the access register number to be used; the valid range is 0..15. Logical accesses are permitted for the VCPU ioctl only. Logical accesses are permitted for non-protected guests only.

Supported flags:

- KVM_S390_MEMOP_F_CHECK_ONLY
- KVM_S390_MEMOP_F_INJECT_EXCEPTION
- KVM_S390_MEMOP_F_SKEY_PROTECTION

The KVM_S390_MEMOP_F_CHECK_ONLY flag can be set to check whether the corresponding memory access would cause an access exception; however, no actual access to the data in memory at the destination is performed. In this case, ”buf” is unused and can be NULL.

In case an access exception occurred during the access (or would occur in case of KVM_S390_MEMOP_F_CHECK_ONLY), the ioctl returns a positive error number indicating the type of exception. This exception is also raised directly at the corresponding VCPU if the flag KVM_S390_MEMOP_F_INJECT_EXCEPTION is set. On protection exceptions, unless specified otherwise, the injected translation-exception identifier (TEID) indicates suppression.

If the KVM_S390_MEMOP_F_SKEY_PROTECTION flag is set, storage key protection is also in effect and may cause exceptions if accesses are prohibited given the access key designated by ”key”; the valid range is 0..15. KVM_S390_MEMOP_F_SKEY_PROTECTION is available if KVM_CAP_S390_MEM_OP_EXTENSION is > 0. Since the accessed memory may span multiple pages and those pages might have different storage keys, it is possible that a protection exception occurs after memory has been modified. In this case, if the exception is injected, the TEID does not indicate suppression.

Absolute read/write:

Access absolute memory. This operation is intended to be used with the KVM_S390_MEMOP_F_SKEY_PROTECTION flag, to allow accessing memory and performing the checks required for storage key protection as one operation (as opposed to user space getting the storage keys, performing the checks, and accessing memory thereafter, which could lead to a delay between check and access). Absolute accesses are permitted for the VM ioctl if KVM_CAP_S390_MEM_OP_EXTENSION has the KVM_S390_MEMOP_EXTENSION_CAP_BASE bit set. Currently absolute accesses are not permitted for VCPU ioctls. Absolute accesses are permitted for non-protected guests only.

Supported flags:

- `KVM_S390_MEMOP_F_CHECK_ONLY`
- `KVM_S390_MEMOP_F_SKEY_PROTECTION`

The semantics of the flags common with logical accesses are as for logical accesses.

Absolute cmpxchg:

Perform cmpxchg on absolute guest memory. Intended for use with the KVM_S390_MEMOP_F_SKEY_PROTECTION flag. Instead of doing an unconditional write, the access occurs only if the target location contains the value pointed to by ”old_addr”. This is performed as an atomic cmpxchg with the length specified by the ”size” parameter. ”size” must be a power of two up to and including 16. If the exchange did not take place because the target value doesn't match the old value, the value ”old_addr” points to is replaced by the target value. User space can tell if an exchange took place by checking if this replacement occurred. The cmpxchg op is permitted for the VM ioctl if KVM_CAP_S390_MEM_OP_EXTENSION has flag KVM_S390_MEMOP_EXTENSION_CAP_CMPXCHG set.

Supported flags:

- `KVM_S390_MEMOP_F_SKEY_PROTECTION`

SIDA read/write:

Access the secure instruction data area which contains memory operands necessary for instruction emulation for protected guests. SIDA accesses are available if the KVM_CAP_S390_PROTECTED capability is available. SIDA accesses are permitted for the VCPU ioctl only. SIDA accesses are permitted for protected guests only.

No flags are supported.

4.90 KVM_S390_GET_SKEYS

Capability
KVM_CAP_S390_SKEYS

Architectures
s390

Type
vm ioctl

Parameters
struct kvm_s390_skeys

Returns
0 on success, KVM_S390_GET_SKEYS_NONE if guest is not using storage keys, negative value on error

This ioctl is used to get guest storage key values on the s390 architecture. The ioctl takes parameters via the kvm_s390_skeys struct:

```
struct kvm_s390_skeys {
      __u64 start_gfn;
      __u64 count;
      __u64 skeydata_addr;
      __u32 flags;
      __u32 reserved[9];
};
```

The start_gfn field is the number of the first guest frame whose storage keys you want to get.

The count field is the number of consecutive frames (starting from start_gfn) whose storage keys to get. The count field must be at least 1 and the maximum allowed value is defined as KVM_S390_SKEYS_MAX. Values outside this range will cause the ioctl to return -EINVAL.

The skeydata_addr field is the address to a buffer large enough to hold count bytes. This buffer will be filled with storage key data by the ioctl.

4.91 KVM_S390_SET_SKEYS

Capability
KVM_CAP_S390_SKEYS

Architectures
s390

Type
vm ioctl

Parameters
struct kvm_s390_skeys

Returns
0 on success, negative value on error

This ioctl is used to set guest storage key values on the s390 architecture. The ioctl takes parameters via the kvm_s390_skeys struct. See section on KVM_S390_GET_SKEYS for struct definition.

The start_gfn field is the number of the first guest frame whose storage keys you want to set.

The count field is the number of consecutive frames (starting from start_gfn) whose storage keys to get. The count field must be at least 1 and the maximum allowed value is defined as KVM_S390_SKEYS_MAX. Values outside this range will cause the ioctl to return -EINVAL.

The skeydata_addr field is the address to a buffer containing count bytes of storage keys. Each byte in the buffer will be set as the storage key for a single frame starting at start_gfn for count frames.

Note: If any architecturally invalid key value is found in the given data then the ioctl will return -EINVAL.

4.92 KVM_S390_IRQ

Capability
KVM_CAP_S390_INJECT_IRQ

Architectures
s390

Type
vcpu ioctl

Parameters
struct kvm_s390_irq (in)

Returns
0 on success, -1 on error

Errors:

EINVAL	interrupt type is invalid type is KVM_S390_SIGP_STOP and flag parameter is invalid value, type is KVM_S390_INT_EXTERNAL_CALL and code is bigger than the maximum of VCPUs
EBUSY	type is KVM_S390_SIGP_SET_PREFIX and vcpu is not stopped, type is KVM_S390_SIGP_STOP and a stop irq is already pending, type is KVM_S390_INT_EXTERNAL_CALL and an external call interrupt is already pending

Allows to inject an interrupt to the guest.

Using struct kvm_s390_irq as a parameter allows to inject additional payload which is not possible via KVM_S390_INTERRUPT.

Interrupt parameters are passed via kvm_s390_irq:

```
struct kvm_s390_irq {
      __u64 type;
      union {
              struct kvm_s390_io_info io;
```

```
                struct kvm_s390_ext_info ext;
                struct kvm_s390_pgm_info pgm;
                struct kvm_s390_emerg_info emerg;
                struct kvm_s390_extcall_info extcall;
                struct kvm_s390_prefix_info prefix;
                struct kvm_s390_stop_info stop;
                struct kvm_s390_mchk_info mchk;
                char reserved[64];
        } u;
};
```

type can be one of the following:

- KVM_S390_SIGP_STOP - sigp stop; parameter in .stop
- KVM_S390_PROGRAM_INT - program check; parameters in .pgm
- KVM_S390_SIGP_SET_PREFIX - sigp set prefix; parameters in .prefix
- KVM_S390_RESTART - restart; no parameters
- KVM_S390_INT_CLOCK_COMP - clock comparator interrupt; no parameters
- KVM_S390_INT_CPU_TIMER - CPU timer interrupt; no parameters
- KVM_S390_INT_EMERGENCY - sigp emergency; parameters in .emerg
- KVM_S390_INT_EXTERNAL_CALL - sigp external call; parameters in .extcall
- KVM_S390_MCHK - machine check interrupt; parameters in .mchk

This is an asynchronous vcpu ioctl and can be invoked from any thread.

4.94 KVM_S390_GET_IRQ_STATE

Capability
KVM_CAP_S390_IRQ_STATE

Architectures
s390

Type
vcpu ioctl

Parameters
struct kvm_s390_irq_state (out)

Returns
>= number of bytes copied into buffer, -EINVAL if buffer size is 0, -ENOBUFS if buffer size is too small to fit all pending interrupts, -EFAULT if the buffer address was invalid

This ioctl allows userspace to retrieve the complete state of all currently pending interrupts in a single buffer. Use cases include migration and introspection. The parameter structure contains the address of a userspace buffer and its length:

```
struct kvm_s390_irq_state {
      __u64 buf;
      __u32 flags;        /* will stay unused for compatibility reasons */
      __u32 len;
      __u32 reserved[4];  /* will stay unused for compatibility reasons */
};
```

Userspace passes in the above struct and for each pending interrupt a struct kvm_s390_irq is copied to the provided buffer.

The structure contains a flags and a reserved field for future extensions. As the kernel never checked for flags == 0 and QEMU never pre-zeroed flags and reserved, these fields can not be used in the future without breaking compatibility.

If -ENOBUFS is returned the buffer provided was too small and userspace may retry with a bigger buffer.

4.95 KVM_S390_SET_IRQ_STATE

Capability
KVM_CAP_S390_IRQ_STATE

Architectures
s390

Type
vcpu ioctl

Parameters
struct kvm_s390_irq_state (in)

Returns
0 on success, -EFAULT if the buffer address was invalid, -EINVAL for an invalid buffer length (see below), -EBUSY if there were already interrupts pending, errors occurring when actually injecting the interrupt. See KVM_S390_IRQ.

This ioctl allows userspace to set the complete state of all cpu-local interrupts currently pending for the vcpu. It is intended for restoring interrupt state after a migration. The input parameter is a userspace buffer containing a struct kvm_s390_irq_state:

```
struct kvm_s390_irq_state {
      __u64 buf;
      __u32 flags;        /* will stay unused for compatibility reasons */
      __u32 len;
      __u32 reserved[4];  /* will stay unused for compatibility reasons */
};
```

The restrictions for flags and reserved apply as well. (see KVM_S390_GET_IRQ_STATE)

The userspace memory referenced by buf contains a struct kvm_s390_irq for each interrupt to be injected into the guest. If one of the interrupts could not be injected for some reason the ioctl aborts.

len must be a multiple of sizeof(struct kvm_s390_irq). It must be > 0 and it must not exceed (max_vcpus + 32) * sizeof(struct kvm_s390_irq), which is the maximum number of possibly

pending cpu-local interrupts.

4.96 KVM_SMI

Capability
KVM_CAP_X86_SMM

Architectures
x86

Type
vcpu ioctl

Parameters
none

Returns
0 on success, -1 on error

Queues an SMI on the thread's vcpu.

4.97 KVM_X86_SET_MSR_FILTER

Capability
KVM_CAP_X86_MSR_FILTER

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_msr_filter

Returns
0 on success, < 0 on error

```
struct kvm_msr_filter_range {
#define KVM_MSR_FILTER_READ  (1 << 0)
#define KVM_MSR_FILTER_WRITE (1 << 1)
      __u32 flags;
      __u32 nmsrs; /* number of msrs in bitmap */
      __u32 base;  /* MSR index the bitmap starts at */
      __u8 *bitmap; /* a 1 bit allows the operations in flags, 0 denies */
};

#define KVM_MSR_FILTER_MAX_RANGES 16
struct kvm_msr_filter {
#define KVM_MSR_FILTER_DEFAULT_ALLOW (0 << 0)
#define KVM_MSR_FILTER_DEFAULT_DENY  (1 << 0)
      __u32 flags;
      struct kvm_msr_filter_range ranges[KVM_MSR_FILTER_MAX_RANGES];
};
```

flags values for `struct kvm_msr_filter_range`:

`KVM_MSR_FILTER_READ`

Filter read accesses to MSRs using the given bitmap. A 0 in the bitmap indicates that read accesses should be denied, while a 1 indicates that a read for a particular MSR should be allowed regardless of the default filter action.

`KVM_MSR_FILTER_WRITE`

Filter write accesses to MSRs using the given bitmap. A 0 in the bitmap indicates that write accesses should be denied, while a 1 indicates that a write for a particular MSR should be allowed regardless of the default filter action.

flags values for `struct kvm_msr_filter`:

`KVM_MSR_FILTER_DEFAULT_ALLOW`

If no filter range matches an MSR index that is getting accessed, KVM will allow accesses to all MSRs by default.

`KVM_MSR_FILTER_DEFAULT_DENY`

If no filter range matches an MSR index that is getting accessed, KVM will deny accesses to all MSRs by default.

This ioctl allows userspace to define up to 16 bitmaps of MSR ranges to deny guest MSR accesses that would normally be allowed by KVM. If an MSR is not covered by a specific range, the ”default” filtering behavior applies. Each bitmap range covers MSRs from [base .. base+nmsrs).

If an MSR access is denied by userspace, the resulting KVM behavior depends on whether or not KVM_CAP_X86_USER_SPACE_MSR's KVM_MSR_EXIT_REASON_FILTER is enabled. If KVM_MSR_EXIT_REASON_FILTER is enabled, KVM will exit to userspace on denied accesses, i.e. userspace effectively intercepts the MSR access. If KVM_MSR_EXIT_REASON_FILTER is not enabled, KVM will inject a #GP into the guest on denied accesses.

If an MSR access is allowed by userspace, KVM will emulate and/or virtualize the access in accordance with the vCPU model. Note, KVM may still ultimately inject a #GP if an access is allowed by userspace, e.g. if KVM doesn't support the MSR, or to follow architectural behavior for the MSR.

By default, KVM operates in KVM_MSR_FILTER_DEFAULT_ALLOW mode with no MSR range filters.

Calling this ioctl with an empty set of ranges (all nmsrs == 0) disables MSR filtering. In that mode, `KVM_MSR_FILTER_DEFAULT_DENY` is invalid and causes an error.

Warning: MSR accesses as part of nested VM-Enter/VM-Exit are not filtered. This includes both writes to individual VMCS fields and reads/writes through the MSR lists pointed to by the VMCS.

x2APIC MSR accesses cannot be filtered (KVM silently ignores filters that cover any x2APIC MSRs).

Note, invoking this ioctl while a vCPU is running is inherently racy. However, KVM does guarantee that vCPUs will see either the previous filter or the new filter, e.g. MSRs with identical

settings in both the old and new filter will have deterministic behavior.

Similarly, if userspace wishes to intercept on denied accesses, KVM_MSR_EXIT_REASON_FILTER must be enabled before activating any filters, and left enabled until after all filters are deactivated. Failure to do so may result in KVM injecting a #GP instead of exiting to userspace.

4.98 KVM_CREATE_SPAPR_TCE_64

Capability
KVM_CAP_SPAPR_TCE_64

Architectures
powerpc

Type
vm ioctl

Parameters
struct kvm_create_spapr_tce_64 (in)

Returns
file descriptor for manipulating the created TCE table

This is an extension for KVM_CAP_SPAPR_TCE which only supports 32bit windows, described in 4.62 KVM_CREATE_SPAPR_TCE

This capability uses extended struct in ioctl interface:

```
/* for KVM_CAP_SPAPR_TCE_64 */
struct kvm_create_spapr_tce_64 {
      __u64 liobn;
      __u32 page_shift;
      __u32 flags;
      __u64 offset;   /* in pages */
      __u64 size;     /* in pages */
};
```

The aim of extension is to support an additional bigger DMA window with a variable page size. KVM_CREATE_SPAPR_TCE_64 receives a 64bit window size, an IOMMU page shift and a bus offset of the corresponding DMA window, @size and @offset are numbers of IOMMU pages.

@flags are not used at the moment.

The rest of functionality is identical to KVM_CREATE_SPAPR_TCE.

4.99 KVM_REINJECT_CONTROL

Capability
KVM_CAP_REINJECT_CONTROL

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_reinject_control (in)

Returns
0 on success, -EFAULT if struct kvm_reinject_control cannot be read, -ENXIO if KVM_CREATE_PIT or KVM_CREATE_PIT2 didn't succeed earlier.

i8254 (PIT) has two modes, reinject and !reinject. The default is reinject, where KVM queues elapsed i8254 ticks and monitors completion of interrupt from vector(s) that i8254 injects. Reinject mode dequeues a tick and injects its interrupt whenever there isn't a pending interrupt from i8254. !reinject mode injects an interrupt as soon as a tick arrives.

```
struct kvm_reinject_control {
      __u8 pit_reinject;
      __u8 reserved[31];
};
```

pit_reinject = 0 (!reinject mode) is recommended, unless running an old operating system that uses the PIT for timing (e.g. Linux 2.4.x).

4.100 KVM_PPC_CONFIGURE_V3_MMU

Capability
KVM_CAP_PPC_RADIX_MMU or KVM_CAP_PPC_HASH_MMU_V3

Architectures
ppc

Type
vm ioctl

Parameters
struct kvm_ppc_mmuv3_cfg (in)

Returns
0 on success, -EFAULT if struct kvm_ppc_mmuv3_cfg cannot be read, -EINVAL if the configuration is invalid

This ioctl controls whether the guest will use radix or HPT (hashed page table) translation, and sets the pointer to the process table for the guest.

```
struct kvm_ppc_mmuv3_cfg {
      __u64   flags;
      __u64   process_table;
};
```

There are two bits that can be set in flags; KVM_PPC_MMUV3_RADIX and KVM_PPC_MMUV3_GTSE. KVM_PPC_MMUV3_RADIX, if set, configures the guest to use radix tree translation, and if clear, to use HPT translation. KVM_PPC_MMUV3_GTSE, if set and if KVM permits it, configures the guest to be able to use the global TLB and SLB invalidation instructions; if clear, the guest may not use these instructions.

The process_table field specifies the address and size of the guest process table, which is in the guest's space. This field is formatted as the second doubleword of the partition table entry, as defined in the Power ISA V3.00, Book III section 5.7.6.1.

4.101 KVM_PPC_GET_RMMU_INFO

Capability
KVM_CAP_PPC_RADIX_MMU

Architectures
ppc

Type
vm ioctl

Parameters
struct kvm_ppc_rmmu_info (out)

Returns
0 on success, -EFAULT if struct kvm_ppc_rmmu_info cannot be written, -EINVAL if no useful information can be returned

This ioctl returns a structure containing two things: (a) a list containing supported radix tree geometries, and (b) a list that maps page sizes to put in the ”AP” (actual page size) field for the tlbie (TLB invalidate entry) instruction.

```
struct kvm_ppc_rmmu_info {
      struct kvm_ppc_radix_geom {
              __u8    page_shift;
              __u8    level_bits[4];
              __u8    pad[3];
      }       geometries[8];
      __u32   ap_encodings[8];
};
```

The geometries[] field gives up to 8 supported geometries for the radix page table, in terms of the log base 2 of the smallest page size, and the number of bits indexed at each level of the tree, from the PTE level up to the PGD level in that order. Any unused entries will have 0 in the page_shift field.

The ap_encodings gives the supported page sizes and their AP field encodings, encoded with the AP value in the top 3 bits and the log base 2 of the page size in the bottom 6 bits.

4.102 KVM_PPC_RESIZE_HPT_PREPARE

Capability
KVM_CAP_SPAPR_RESIZE_HPT

Architectures
powerpc

Type
vm ioctl

Parameters
struct kvm_ppc_resize_hpt (in)

Returns
0 on successful completion, >0 if a new HPT is being prepared, the value is an estimated number of milliseconds until preparation is complete, -EFAULT if struct kvm_reinject_control cannot be read, -EINVAL if the supplied shift or flags are invalid, -ENOMEM if unable to allocate the new HPT,

Used to implement the PAPR extension for runtime resizing of a guest's Hashed Page Table (HPT). Specifically this starts, stops or monitors the preparation of a new potential HPT for the guest, essentially implementing the H_RESIZE_HPT_PREPARE hypercall.

```
struct kvm_ppc_resize_hpt {
      __u64 flags;
      __u32 shift;
      __u32 pad;
};
```

If called with shift > 0 when there is no pending HPT for the guest, this begins preparation of a new pending HPT of size 2^(shift) bytes. It then returns a positive integer with the estimated number of milliseconds until preparation is complete.

If called when there is a pending HPT whose size does not match that requested in the parameters, discards the existing pending HPT and creates a new one as above.

If called when there is a pending HPT of the size requested, will:

- If preparation of the pending HPT is already complete, return 0
- If preparation of the pending HPT has failed, return an error code, then discard the pending HPT.
- If preparation of the pending HPT is still in progress, return an estimated number of milliseconds until preparation is complete.

If called with shift == 0, discards any currently pending HPT and returns 0 (i.e. cancels any in-progress preparation).

flags is reserved for future expansion, currently setting any bits in flags will result in an -EINVAL.

Normally this will be called repeatedly with the same parameters until it returns <= 0. The first call will initiate preparation, subsequent ones will monitor preparation until it completes or fails.

4.103 KVM_PPC_RESIZE_HPT_COMMIT

Capability
KVM_CAP_SPAPR_RESIZE_HPT

Architectures
powerpc

Type
vm ioctl

Parameters
struct kvm_ppc_resize_hpt (in)

Returns
0 on successful completion, -EFAULT if struct kvm_reinject_control cannot be read, -EINVAL if the supplied shift or flags are invalid, -ENXIO is there is no pending HPT, or the pending HPT doesn't have the requested size, -EBUSY if the pending HPT is not fully prepared, -ENOSPC if there was a hash collision when moving existing HPT entries to the new HPT, -EIO on other error conditions

Used to implement the PAPR extension for runtime resizing of a guest's Hashed Page Table (HPT). Specifically this requests that the guest be transferred to working with the new HPT, essentially implementing the H_RESIZE_HPT_COMMIT hypercall.

```
struct kvm_ppc_resize_hpt {
      __u64 flags;
      __u32 shift;
      __u32 pad;
};
```

This should only be called after KVM_PPC_RESIZE_HPT_PREPARE has returned 0 with the same parameters. In other cases KVM_PPC_RESIZE_HPT_COMMIT will return an error (usually -ENXIO or -EBUSY, though others may be possible if the preparation was started, but failed).

This will have undefined effects on the guest if it has not already placed itself in a quiescent state where no vcpu will make MMU enabled memory accesses.

On successful completion, the pending HPT will become the guest's active HPT and the previous HPT will be discarded.

On failure, the guest will still be operating on its previous HPT.

4.104 KVM_X86_GET_MCE_CAP_SUPPORTED

Capability
KVM_CAP_MCE

Architectures
x86

Type
system ioctl

Parameters
u64 mce_cap (out)

Returns
0 on success, -1 on error

Returns supported MCE capabilities. The u64 mce_cap parameter has the same format as the MSR_IA32_MCG_CAP register. Supported capabilities will have the corresponding bits set.

4.105 KVM_X86_SETUP_MCE

Capability
KVM_CAP_MCE

Architectures
x86

Type
vcpu ioctl

Parameters
u64 mcg_cap (in)

Returns
0 on success, -EFAULT if u64 mcg_cap cannot be read, -EINVAL if the requested number of banks is invalid, -EINVAL if requested MCE capability is not supported.

Initializes MCE support for use. The u64 mcg_cap parameter has the same format as the MSR_IA32_MCG_CAP register and specifies which capabilities should be enabled. The maximum supported number of error-reporting banks can be retrieved when checking for KVM_CAP_MCE. The supported capabilities can be retrieved with KVM_X86_GET_MCE_CAP_SUPPORTED.

4.106 KVM_X86_SET_MCE

Capability
KVM_CAP_MCE

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_x86_mce (in)

Returns
0 on success, -EFAULT if struct kvm_x86_mce cannot be read, -EINVAL if the bank number is invalid, -EINVAL if VAL bit is not set in status field.

Inject a machine check error (MCE) into the guest. The input parameter is:

```
struct kvm_x86_mce {
      __u64 status;
      __u64 addr;
      __u64 misc;
      __u64 mcg_status;
```

```
        __u8 bank;
        __u8 pad1[7];
        __u64 pad2[3];
};
```

If the MCE being reported is an uncorrected error, KVM will inject it as an MCE exception into the guest. If the guest MCG_STATUS register reports that an MCE is in progress, KVM causes an KVM_EXIT_SHUTDOWN vmexit.

Otherwise, if the MCE is a corrected error, KVM will just store it in the corresponding bank (provided this bank is not holding a previously reported uncorrected error).

4.107 KVM_S390_GET_CMMA_BITS

Capability
KVM_CAP_S390_CMMA_MIGRATION

Architectures
s390

Type
vm ioctl

Parameters
struct kvm_s390_cmma_log (in, out)

Returns
0 on success, a negative value on error

Errors:

ENOMEM	not enough memory can be allocated to complete the task
ENXIO	if CMMA is not enabled
EINVAL	if KVM_S390_CMMA_PEEK is not set but migration mode was not enabled
EINVAL	if KVM_S390_CMMA_PEEK is not set but dirty tracking has been disabled (and thus migration mode was automatically disabled)
EFAULT	if the userspace address is invalid or if no page table is present for the addresses (e.g. when using hugepages).

This ioctl is used to get the values of the CMMA bits on the s390 architecture. It is meant to be used in two scenarios:

- During live migration to save the CMMA values. Live migration needs to be enabled via the KVM_REQ_START_MIGRATION VM property.
- To non-destructively peek at the CMMA values, with the flag KVM_S390_CMMA_PEEK set.

The ioctl takes parameters via the kvm_s390_cmma_log struct. The desired values are written to a buffer whose location is indicated via the ”values” member in the kvm_s390_cmma_log struct. The values in the input struct are also updated as needed.

Each CMMA value takes up one byte.

```
struct kvm_s390_cmma_log {
      __u64 start_gfn;
      __u32 count;
      __u32 flags;
      union {
              __u64 remaining;
              __u64 mask;
      };
      __u64 values;
};
```

start_gfn is the number of the first guest frame whose CMMA values are to be retrieved,

count is the length of the buffer in bytes,

values points to the buffer where the result will be written to.

If count is greater than KVM_S390_SKEYS_MAX, then it is considered to be KVM_S390_SKEYS_MAX. KVM_S390_SKEYS_MAX is re-used for consistency with other ioctls.

The result is written in the buffer pointed to by the field values, and the values of the input parameter are updated as follows.

Depending on the flags, different actions are performed. The only supported flag so far is KVM_S390_CMMA_PEEK.

The default behaviour if KVM_S390_CMMA_PEEK is not set is: start_gfn will indicate the first page frame whose CMMA bits were dirty. It is not necessarily the same as the one passed as input, as clean pages are skipped.

count will indicate the number of bytes actually written in the buffer. It can (and very often will) be smaller than the input value, since the buffer is only filled until 16 bytes of clean values are found (which are then not copied in the buffer). Since a CMMA migration block needs the base address and the length, for a total of 16 bytes, we will send back some clean data if there is some dirty data afterwards, as long as the size of the clean data does not exceed the size of the header. This allows to minimize the amount of data to be saved or transferred over the network at the expense of more roundtrips to userspace. The next invocation of the ioctl will skip over all the clean values, saving potentially more than just the 16 bytes we found.

If KVM_S390_CMMA_PEEK is set: the existing storage attributes are read even when not in migration mode, and no other action is performed;

the output start_gfn will be equal to the input start_gfn,

the output count will be equal to the input count, except if the end of memory has been reached.

In both cases: the field ”remaining” will indicate the total number of dirty CMMA values still remaining, or 0 if KVM_S390_CMMA_PEEK is set and migration mode is not enabled.

mask is unused.

values points to the userspace buffer where the result will be stored.

4.108 KVM_S390_SET_CMMA_BITS

Capability
KVM_CAP_S390_CMMA_MIGRATION

Architectures
s390

Type
vm ioctl

Parameters
struct kvm_s390_cmma_log (in)

Returns
0 on success, a negative value on error

This ioctl is used to set the values of the CMMA bits on the s390 architecture. It is meant to be used during live migration to restore the CMMA values, but there are no restrictions on its use. The ioctl takes parameters via the kvm_s390_cmma_values struct. Each CMMA value takes up one byte.

```
struct kvm_s390_cmma_log {
      __u64 start_gfn;
      __u32 count;
      __u32 flags;
      union {
              __u64 remaining;
              __u64 mask;
      };
      __u64 values;
};
```

start_gfn indicates the starting guest frame number,

count indicates how many values are to be considered in the buffer,

flags is not used and must be 0.

mask indicates which PGSTE bits are to be considered.

remaining is not used.

values points to the buffer in userspace where to store the values.

This ioctl can fail with -ENOMEM if not enough memory can be allocated to complete the task, with -ENXIO if CMMA is not enabled, with -EINVAL if the count field is too large (e.g. more than KVM_S390_CMMA_SIZE_MAX) or if the flags field was not 0, with -EFAULT if the userspace address is invalid, if invalid pages are written to (e.g. after the end of memory) or if no page table is present for the addresses (e.g. when using hugepages).

4.109 KVM_PPC_GET_CPU_CHAR

Capability
KVM_CAP_PPC_GET_CPU_CHAR

Architectures
powerpc

Type
vm ioctl

Parameters
struct kvm_ppc_cpu_char (out)

Returns
0 on successful completion, -EFAULT if struct kvm_ppc_cpu_char cannot be written

This ioctl gives userspace information about certain characteristics of the CPU relating to speculative execution of instructions and possible information leakage resulting from speculative execution (see CVE-2017-5715, CVE-2017-5753 and CVE-2017-5754). The information is returned in struct kvm_ppc_cpu_char, which looks like this:

```
struct kvm_ppc_cpu_char {
      __u64   character;              /* characteristics of the CPU */
      __u64   behaviour;              /* recommended software behaviour */
      __u64   character_mask;         /* valid bits in character */
      __u64   behaviour_mask;         /* valid bits in behaviour */
};
```

For extensibility, the character_mask and behaviour_mask fields indicate which bits of character and behaviour have been filled in by the kernel. If the set of defined bits is extended in future then userspace will be able to tell whether it is running on a kernel that knows about the new bits.

The character field describes attributes of the CPU which can help with preventing inadvertent information disclosure - specifically, whether there is an instruction to flash-invalidate the L1 data cache (ori 30,30,0 or mtspr SPRN_TRIG2,rN), whether the L1 data cache is set to a mode where entries can only be used by the thread that created them, whether the bcctr[l] instruction prevents speculation, and whether a speculation barrier instruction (ori 31,31,0) is provided.

The behaviour field describes actions that software should take to prevent inadvertent information disclosure, and thus describes which vulnerabilities the hardware is subject to; specifically whether the L1 data cache should be flushed when returning to user mode from the kernel, and whether a speculation barrier should be placed between an array bounds check and the array access.

These fields use the same bit definitions as the new H_GET_CPU_CHARACTERISTICS hypercall.

4.110 KVM_MEMORY_ENCRYPT_OP

Capability
basic

Architectures
x86

Type
vm

Parameters
an opaque platform specific structure (in/out)

Returns
0 on success; -1 on error

If the platform supports creating encrypted VMs then this ioctl can be used for issuing platform-specific memory encryption commands to manage those encrypted VMs.

Currently, this ioctl is used for issuing Secure Encrypted Virtualization (SEV) commands on AMD Processors. The SEV commands are defined in *Secure Encrypted Virtualization (SEV)*.

4.111 KVM_MEMORY_ENCRYPT_REG_REGION

Capability
basic

Architectures
x86

Type
system

Parameters
struct kvm_enc_region (in)

Returns
0 on success; -1 on error

This ioctl can be used to register a guest memory region which may contain encrypted data (e.g. guest RAM, SMRAM etc).

It is used in the SEV-enabled guest. When encryption is enabled, a guest memory region may contain encrypted data. The SEV memory encryption engine uses a tweak such that two identical plaintext pages, each at different locations will have differing ciphertexts. So swapping or moving ciphertext of those pages will not result in plaintext being swapped. So relocating (or migrating) physical backing pages for the SEV guest will require some additional steps.

Note: The current SEV key management spec does not provide commands to swap or migrate (move) ciphertext pages. Hence, for now we pin the guest memory region registered with the ioctl.

4.112 KVM_MEMORY_ENCRYPT_UNREG_REGION

Capability
basic

Architectures
x86

Type
system

Parameters
struct kvm_enc_region (in)

Returns
0 on success; -1 on error

This ioctl can be used to unregister the guest memory region registered with KVM_MEMORY_ENCRYPT_REG_REGION ioctl above.

4.113 KVM_HYPERV_EVENTFD

Capability
KVM_CAP_HYPERV_EVENTFD

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_hyperv_eventfd (in)

This ioctl (un)registers an eventfd to receive notifications from the guest on the specified Hyper-V connection id through the SIGNAL_EVENT hypercall, without causing a user exit. SIGNAL_EVENT hypercall with non-zero event flag number (bits 24-31) still triggers a KVM_EXIT_HYPERV_HCALL user exit.

```
struct kvm_hyperv_eventfd {
      __u32 conn_id;
      __s32 fd;
      __u32 flags;
      __u32 padding[3];
};
```

The conn_id field should fit within 24 bits:

```
#define KVM_HYPERV_CONN_ID_MASK                0x00ffffff
```

The acceptable values for the flags field are:

```
#define KVM_HYPERV_EVENTFD_DEASSIGN   (1 << 0)
```

Returns
0 on success, -EINVAL if conn_id or flags is outside the allowed range, -ENOENT

on deassign if the conn_id isn't registered, -EEXIST on assign if the conn_id is already registered

4.114 KVM_GET_NESTED_STATE

Capability
KVM_CAP_NESTED_STATE

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_nested_state (in/out)

Returns
0 on success, -1 on error

Errors:

E2BIG	the total state size exceeds the value of 'size' specified by the user; the size required will be written into size.

```
struct kvm_nested_state {
      __u16 flags;
      __u16 format;
      __u32 size;

      union {
              struct kvm_vmx_nested_state_hdr vmx;
              struct kvm_svm_nested_state_hdr svm;

              /* Pad the header to 128 bytes.  */
              __u8 pad[120];
      } hdr;

      union {
              struct kvm_vmx_nested_state_data vmx[0];
              struct kvm_svm_nested_state_data svm[0];
      } data;
};

#define KVM_STATE_NESTED_GUEST_MODE           0x00000001
#define KVM_STATE_NESTED_RUN_PENDING          0x00000002
#define KVM_STATE_NESTED_EVMCS                0x00000004

#define KVM_STATE_NESTED_FORMAT_VMX           0
#define KVM_STATE_NESTED_FORMAT_SVM           1
```

```
#define KVM_STATE_NESTED_VMX_VMCS_SIZE        0x1000

#define KVM_STATE_NESTED_VMX_SMM_GUEST_MODE   0x00000001
#define KVM_STATE_NESTED_VMX_SMM_VMXON        0x00000002

#define KVM_STATE_VMX_PREEMPTION_TIMER_DEADLINE 0x00000001

struct kvm_vmx_nested_state_hdr {
      __u64 vmxon_pa;
      __u64 vmcs12_pa;

      struct {
              __u16 flags;
      } smm;

      __u32 flags;
      __u64 preemption_timer_deadline;
};

struct kvm_vmx_nested_state_data {
      __u8 vmcs12[KVM_STATE_NESTED_VMX_VMCS_SIZE];
      __u8 shadow_vmcs12[KVM_STATE_NESTED_VMX_VMCS_SIZE];
};
```

This ioctl copies the vcpu's nested virtualization state from the kernel to userspace.

The maximum size of the state can be retrieved by passing KVM_CAP_NESTED_STATE to the KVM_CHECK_EXTENSION ioctl().

4.115 KVM_SET_NESTED_STATE

Capability
KVM_CAP_NESTED_STATE

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_nested_state (in)

Returns
0 on success, -1 on error

This copies the vcpu's kvm_nested_state struct from userspace to the kernel. For the definition of struct kvm_nested_state, see KVM_GET_NESTED_STATE.

4.116 KVM_(UN)REGISTER_COALESCED_MMIO

Capability
KVM_CAP_COALESCED_MMIO (for coalesced mmio)
KVM_CAP_COALESCED_PIO (for coalesced pio)

Architectures
all

Type
vm ioctl

Parameters
struct kvm_coalesced_mmio_zone

Returns
0 on success, < 0 on error

Coalesced I/O is a performance optimization that defers hardware register write emulation so that userspace exits are avoided. It is typically used to reduce the overhead of emulating frequently accessed hardware registers.

When a hardware register is configured for coalesced I/O, write accesses do not exit to userspace and their value is recorded in a ring buffer that is shared between kernel and userspace.

Coalesced I/O is used if one or more write accesses to a hardware register can be deferred until a read or a write to another hardware register on the same device. This last access will cause a vmexit and userspace will process accesses from the ring buffer before emulating it. That will avoid exiting to userspace on repeated writes.

Coalesced pio is based on coalesced mmio. There is little difference between coalesced mmio and pio except that coalesced pio records accesses to I/O ports.

4.117 KVM_CLEAR_DIRTY_LOG (vm ioctl)

Capability
KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2

Architectures
x86, arm64, mips

Type
vm ioctl

Parameters
struct kvm_clear_dirty_log (in)

Returns
0 on success, -1 on error

```
/* for KVM_CLEAR_DIRTY_LOG */
struct kvm_clear_dirty_log {
      __u32 slot;
      __u32 num_pages;
      __u64 first_page;
      union {
```

```
                void __user *dirty_bitmap; /* one bit per page */
                __u64 padding;
        };
};
```

The ioctl clears the dirty status of pages in a memory slot, according to the bitmap that is passed in struct kvm_clear_dirty_log's dirty_bitmap field. Bit 0 of the bitmap corresponds to page "first_page" in the memory slot, and num_pages is the size in bits of the input bitmap. first_page must be a multiple of 64; num_pages must also be a multiple of 64 unless first_page + num_pages is the size of the memory slot. For each bit that is set in the input bitmap, the corresponding page is marked "clean" in KVM's dirty bitmap, and dirty tracking is re-enabled for that page (for example via write-protection, or by clearing the dirty bit in a page table entry).

If KVM_CAP_MULTI_ADDRESS_SPACE is available, bits 16-31 of slot field specifies the address space for which you want to clear the dirty status. See KVM_SET_USER_MEMORY_REGION for details on the usage of slot field.

This ioctl is mostly useful when KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2 is enabled; for more information, see the description of the capability. However, it can always be used as long as KVM_CHECK_EXTENSION confirms that KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2 is present.

4.118 KVM_GET_SUPPORTED_HV_CPUID

Capability
KVM_CAP_HYPERV_CPUID (vcpu), KVM_CAP_SYS_HYPERV_CPUID (system)

Architectures
x86

Type
system ioctl, vcpu ioctl

Parameters
struct kvm_cpuid2 (in/out)

Returns
0 on success, -1 on error

```
struct kvm_cpuid2 {
        __u32 nent;
        __u32 padding;
        struct kvm_cpuid_entry2 entries[0];
};

struct kvm_cpuid_entry2 {
        __u32 function;
        __u32 index;
        __u32 flags;
        __u32 eax;
        __u32 ebx;
        __u32 ecx;
        __u32 edx;
```

```
        __u32 padding[3];
};
```

This ioctl returns x86 cpuid features leaves related to Hyper-V emulation in KVM. Userspace can use the information returned by this ioctl to construct cpuid information presented to guests consuming Hyper-V enlightenments (e.g. Windows or Hyper-V guests).

CPUID feature leaves returned by this ioctl are defined by Hyper-V Top Level Functional Specification (TLFS). These leaves can't be obtained with KVM_GET_SUPPORTED_CPUID ioctl because some of them intersect with KVM feature leaves (0x40000000, 0x40000001).

Currently, the following list of CPUID leaves are returned:

- HYPERV_CPUID_VENDOR_AND_MAX_FUNCTIONS
- HYPERV_CPUID_INTERFACE
- HYPERV_CPUID_VERSION
- HYPERV_CPUID_FEATURES
- HYPERV_CPUID_ENLIGHTMENT_INFO
- HYPERV_CPUID_IMPLEMENT_LIMITS
- HYPERV_CPUID_NESTED_FEATURES
- HYPERV_CPUID_SYNDBG_VENDOR_AND_MAX_FUNCTIONS
- HYPERV_CPUID_SYNDBG_INTERFACE
- HYPERV_CPUID_SYNDBG_PLATFORM_CAPABILITIES

Userspace invokes KVM_GET_SUPPORTED_HV_CPUID by passing a kvm_cpuid2 structure with the 'nent' field indicating the number of entries in the variable-size array 'entries'. If the number of entries is too low to describe all Hyper-V feature leaves, an error (E2BIG) is returned. If the number is more or equal to the number of Hyper-V feature leaves, the 'nent' field is adjusted to the number of valid entries in the 'entries' array, which is then filled.

'index' and 'flags' fields in 'struct kvm_cpuid_entry2' are currently reserved, userspace should not expect to get any particular value there.

Note, vcpu version of KVM_GET_SUPPORTED_HV_CPUID is currently deprecated. Unlike system ioctl which exposes all supported feature bits unconditionally, vcpu version has the following quirks:

- HYPERV_CPUID_NESTED_FEATURES leaf and HV_X64_ENLIGHTENED_VMCS_RECOMMENDE feature bit are only exposed when Enlightened VMCS was previously enabled on the corresponding vCPU (KVM_CAP_HYPERV_ENLIGHTENED_VMCS).
- HV_STIMER_DIRECT_MODE_AVAILABLE bit is only exposed with in-kernel LAPIC. (presumes KVM_CREATE_IRQCHIP has already been called).

4.119 KVM_ARM_VCPU_FINALIZE

Architectures
arm64

Type
vcpu ioctl

Parameters
int feature (in)

Returns
0 on success, -1 on error

Errors:

EPERM	feature not enabled, needs configuration, or already finalized
EINVAL	feature unknown or not present

Recognised values for feature:

arm64	KVM_ARM_VCPU_SVE (requires KVM_CAP_ARM_SVE)

Finalizes the configuration of the specified vcpu feature.

The vcpu must already have been initialised, enabling the affected feature, by means of a successful KVM_ARM_VCPU_INIT call with the appropriate flag set in features[].

For affected vcpu features, this is a mandatory step that must be performed before the vcpu is fully usable.

Between KVM_ARM_VCPU_INIT and KVM_ARM_VCPU_FINALIZE, the feature may be configured by use of ioctls such as KVM_SET_ONE_REG. The exact configuration that should be performed and how to do it are feature-dependent.

Other calls that depend on a particular feature being finalized, such as KVM_RUN, KVM_GET_REG_LIST, KVM_GET_ONE_REG and KVM_SET_ONE_REG, will fail with -EPERM unless the feature has already been finalized by means of a KVM_ARM_VCPU_FINALIZE call.

See KVM_ARM_VCPU_INIT for details of vcpu features that require finalization using this ioctl.

4.120 KVM_SET_PMU_EVENT_FILTER

Capability
KVM_CAP_PMU_EVENT_FILTER

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_pmu_event_filter (in)

Returns

0 on success, -1 on error

Errors:

EFAULT	args[0] cannot be accessed
EINVAL	args[0] contains invalid data in the filter or filter events
E2BIG	nevents is too large
EBUSY	not enough memory to allocate the filter

```
struct kvm_pmu_event_filter {
      __u32 action;
      __u32 nevents;
      __u32 fixed_counter_bitmap;
      __u32 flags;
      __u32 pad[4];
      __u64 events[0];
};
```

This ioctl restricts the set of PMU events the guest can program by limiting which event select and unit mask combinations are permitted.

The argument holds a list of filter events which will be allowed or denied.

Filter events only control general purpose counters; fixed purpose counters are controlled by the fixed_counter_bitmap.

Valid values for 'flags':

```
``0``
```

To use this mode, clear the 'flags' field.

In this mode each event will contain an event select + unit mask.

When the guest attempts to program the PMU the guest's event select + unit mask is compared against the filter events to determine whether the guest should have access.

`KVM_PMU_EVENT_FLAG_MASKED_EVENTS`:Capability: KVM_CAP_PMU_EVENT_MASKED_EVENTS

In this mode each filter event will contain an event select, mask, match, and exclude value. To encode a masked event use:

```
KVM_PMU_ENCODE_MASKED_ENTRY()
```

An encoded event will follow this layout:

```
Bits   Description
----   -----------
7:0    event select (low bits)
15:8   umask match
31:16  unused
35:32  event select (high bits)
36:54  unused
```

```
55     exclude bit
63:56  umask mask
```

When the guest attempts to program the PMU, these steps are followed in determining if the guest should have access:

1. Match the event select from the guest against the filter events.
2. If a match is found, match the guest's unit mask to the mask and match values of the included filter events. I.e. (unit mask & mask) == match && !exclude.
3. If a match is found, match the guest's unit mask to the mask and match values of the excluded filter events. I.e. (unit mask & mask) == match && exclude.
4. a. If an included match is found and an excluded match is not found, filter the event.
   b. For everything else, do not filter the event.
5. a. If the event is filtered and it's an allow list, allow the guest to program the event.
   b. If the event is filtered and it's a deny list, do not allow the guest to program the event.

When setting a new pmu event filter, -EINVAL will be returned if any of the unused fields are set or if any of the high bits (35:32) in the event select are set when called on Intel.

Valid values for 'action':

```
#define KVM_PMU_EVENT_ALLOW 0
#define KVM_PMU_EVENT_DENY 1
```

Via this API, KVM userspace can also control the behavior of the VM's fixed counters (if any) by configuring the ”action” and ”fixed_counter_bitmap” fields.

Specifically, KVM follows the following pseudo-code when determining whether to allow the guest FixCtr[i] to count its pre-defined fixed event:

```
FixCtr[i]_is_allowed = (action == ALLOW) && (bitmap & BIT(i)) ||
  (action == DENY) && !(bitmap & BIT(i));
FixCtr[i]_is_denied = !FixCtr[i]_is_allowed;
```

KVM always consumes fixed_counter_bitmap, it's userspace's responsibility to ensure fixed_counter_bitmap is set correctly, e.g. if userspace wants to define a filter that only affects general purpose counters.

Note, the ”events” field also applies to fixed counters' hardcoded event_select and unit_mask values. ”fixed_counter_bitmap” has higher priority than ”events” if there is a contradiction between the two.

4.121 KVM_PPC_SVM_OFF

Capability
basic

Architectures
powerpc

Type
vm ioctl

Parameters
none

Returns
0 on successful completion,

Errors:

EINVAL	if ultravisor failed to terminate the secure guest
ENOMEM	if hypervisor failed to allocate new radix page tables for guest

This ioctl is used to turn off the secure mode of the guest or transition the guest from secure mode to normal mode. This is invoked when the guest is reset. This has no effect if called for a normal guest.

This ioctl issues an ultravisor call to terminate the secure guest, unpins the VPA pages and releases all the device pages that are used to track the secure pages by hypervisor.

4.122 KVM_S390_NORMAL_RESET

Capability
KVM_CAP_S390_VCPU_RESETS

Architectures
s390

Type
vcpu ioctl

Parameters
none

Returns
0

This ioctl resets VCPU registers and control structures according to the cpu reset definition in the POP (Principles Of Operation).

4.123 KVM_S390_INITIAL_RESET

Capability
none

Architectures
s390

Type
vcpu ioctl

Parameters
none

Returns
0

This ioctl resets VCPU registers and control structures according to the initial cpu reset definition in the POP. However, the cpu is not put into ESA mode. This reset is a superset of the normal reset.

4.124 KVM_S390_CLEAR_RESET

Capability
KVM_CAP_S390_VCPU_RESETS

Architectures
s390

Type
vcpu ioctl

Parameters
none

Returns
0

This ioctl resets VCPU registers and control structures according to the clear cpu reset definition in the POP. However, the cpu is not put into ESA mode. This reset is a superset of the initial reset.

4.125 KVM_S390_PV_COMMAND

Capability
KVM_CAP_S390_PROTECTED

Architectures
s390

Type
vm ioctl

Parameters
struct kvm_pv_cmd

Returns
0 on success, < 0 on error

```
struct kvm_pv_cmd {
      __u32 cmd;      /* Command to be executed */
      __u16 rc;       /* Ultravisor return code */
      __u16 rrc;      /* Ultravisor return reason code */
      __u64 data;     /* Data or address */
      __u32 flags;    /* flags for future extensions. Must be 0 for now */
      __u32 reserved[3];
};
```

Ultravisor return codes The Ultravisor return (reason) codes are provided by the kernel if a Ultravisor call has been executed to achieve the results expected by the command. Therefore they are independent of the IOCTL return code. If KVM changes *rc*, its value will always be greater than 0 hence setting it to 0 before issuing a PV command is advised to be able to detect a change of *rc*.

cmd values:

KVM_PV_ENABLE
Allocate memory and register the VM with the Ultravisor, thereby donating memory to the Ultravisor that will become inaccessible to KVM. All existing CPUs are converted to protected ones. After this command has succeeded, any CPU added via hotplug will become protected during its creation as well.

Errors:

EINTR	an unmasked signal is pending

KVM_PV_DISABLE
Deregister the VM from the Ultravisor and reclaim the memory that had been donated to the Ultravisor, making it usable by the kernel again. All registered VCPUs are converted back to non-protected ones. If a previous protected VM had been prepared for asynchronous teardown with KVM_PV_ASYNC_CLEANUP_PREPARE and not subsequently torn down with KVM_PV_ASYNC_CLEANUP_PERFORM, it will be torn down in this call together with the current protected VM.

KVM_PV_VM_SET_SEC_PARMS
Pass the image header from VM memory to the Ultravisor in preparation of image unpacking and verification.

KVM_PV_VM_UNPACK
Unpack (protect and decrypt) a page of the encrypted boot image.

KVM_PV_VM_VERIFY
Verify the integrity of the unpacked image. Only if this succeeds, KVM is allowed to start protected VCPUs.

KVM_PV_INFO

Capability
KVM_CAP_S390_PROTECTED_DUMP

Presents an API that provides Ultravisor related data to userspace via subcommands. len_max is the size of the user space buffer, len_written is KVM's indication of how much

bytes of that buffer were actually written to. len_written can be used to determine the valid fields if more response fields are added in the future.

```
enum pv_cmd_info_id {
   KVM_PV_INFO_VM,
   KVM_PV_INFO_DUMP,
};

struct kvm_s390_pv_info_header {
   __u32 id;
   __u32 len_max;
   __u32 len_written;
   __u32 reserved;
};

struct kvm_s390_pv_info {
   struct kvm_s390_pv_info_header header;
   struct kvm_s390_pv_info_dump dump;
   struct kvm_s390_pv_info_vm vm;
};
```

subcommands:

KVM_PV_INFO_VM

This subcommand provides basic Ultravisor information for PV hosts. These values are likely also exported as files in the sysfs firmware UV query interface but they are more easily available to programs in this API.

The installed calls and feature_indication members provide the installed UV calls and the UV's other feature indications.

The max_* members provide information about the maximum number of PV vcpus, PV guests and PV guest memory size.

```
struct kvm_s390_pv_info_vm {
  __u64 inst_calls_list[4];
  __u64 max_cpus;
  __u64 max_guests;
  __u64 max_guest_addr;
  __u64 feature_indication;
};
```

KVM_PV_INFO_DUMP

This subcommand provides information related to dumping PV guests.

```
struct kvm_s390_pv_info_dump {
  __u64 dump_cpu_buffer_len;
  __u64 dump_config_mem_buffer_per_1m;
  __u64 dump_config_finalize_len;
};
```

KVM_PV_DUMP

Capability

KVM_CAP_S390_PROTECTED_DUMP

Presents an API that provides calls which facilitate dumping a protected VM.

```
struct kvm_s390_pv_dmp {
  __u64 subcmd;
  __u64 buff_addr;
  __u64 buff_len;
  __u64 gaddr;              /* For dump storage state */
};
```

subcommands:

KVM_PV_DUMP_INIT
Initializes the dump process of a protected VM. If this call does not succeed all other subcommands will fail with -EINVAL. This subcommand will return -EINVAL if a dump process has not yet been completed.

Not all PV vms can be dumped, the owner needs to set *dump allowed* PCF bit 34 in the SE header to allow dumping.

KVM_PV_DUMP_CONFIG_STOR_STATE
Stores *buff_len* bytes of tweak component values starting with the 1MB block specified by the absolute guest address (*gaddr*). *buff_len* needs to be *conf_dump_storage_state_len* aligned and at least >= the *conf_dump_storage_state_len* value provided by the dump uv_info data. buff_user might be written to even if an error rc is returned. For instance if we encounter a fault after writing the first page of data.

KVM_PV_DUMP_COMPLETE
If the subcommand succeeds it completes the dump process and lets KVM_PV_DUMP_INIT be called again.

On success *conf_dump_finalize_len* bytes of completion data will be stored to the *buff_addr*. The completion data contains a key derivation seed, IV, tweak nonce and encryption keys as well as an authentication tag all of which are needed to decrypt the dump at a later time.

KVM_PV_ASYNC_CLEANUP_PREPARE

Capability
KVM_CAP_S390_PROTECTED_ASYNC_DISABLE

Prepare the current protected VM for asynchronous teardown. Most resources used by the current protected VM will be set aside for a subsequent asynchronous teardown. The current protected VM will then resume execution immediately as non-protected. There can be at most one protected VM prepared for asynchronous teardown at any time. If a protected VM had already been prepared for teardown without subsequently calling KVM_PV_ASYNC_CLEANUP_PERFORM, this call will fail. In that case, the userspace process should issue a normal KVM_PV_DISABLE. The resources set aside with this call will need to be cleaned up with a subsequent call to KVM_PV_ASYNC_CLEANUP_PERFORM or KVM_PV_DISABLE, otherwise they will be cleaned up when KVM terminates. KVM_PV_ASYNC_CLEANUP_PREPARE can be called again as soon as cleanup starts, i.e. before KVM_PV_ASYNC_CLEANUP_PERFORM finishes.

KVM_PV_ASYNC_CLEANUP_PERFORM

Capability
KVM_CAP_S390_PROTECTED_ASYNC_DISABLE

Tear down the protected VM previously prepared for teardown with KVM_PV_ASYNC_CLEANUP_PREPARE. The resources that had been set aside will be freed during the execution of this command. This PV command should ideally be issued by userspace from a separate thread. If a fatal signal is received (or the process terminates naturally), the command will terminate immediately without completing, and the normal KVM shutdown procedure will take care of cleaning up all remaining protected VMs, including the ones whose teardown was interrupted by process termination.

4.126 KVM_XEN_HVM_SET_ATTR

Capability
KVM_CAP_XEN_HVM / KVM_XEN_HVM_CONFIG_SHARED_INFO

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_xen_hvm_attr

Returns
0 on success, < 0 on error

```
struct kvm_xen_hvm_attr {
      __u16 type;
      __u16 pad[3];
      union {
              __u8 long_mode;
              __u8 vector;
              __u8 runstate_update_flag;
              struct {
                      __u64 gfn;
              } shared_info;
              struct {
                      __u32 send_port;
                      __u32 type; /* EVTCHNSTAT_ipi / EVTCHNSTAT_interdomain */
                      __u32 flags;
                      union {
                              struct {
                                      __u32 port;
                                      __u32 vcpu;
                                      __u32 priority;
                              } port;
                              struct {
                                      __u32 port; /* Zero for eventfd */
                                      __s32 fd;
                              } eventfd;
```

```
                        __u32 padding[4];
                } deliver;
        } evtchn;
        __u32 xen_version;
        __u64 pad[8];
    } u;
};
```

type values:

KVM_XEN_ATTR_TYPE_LONG_MODE
Sets the ABI mode of the VM to 32-bit or 64-bit (long mode). This determines the layout of the shared info pages exposed to the VM.

KVM_XEN_ATTR_TYPE_SHARED_INFO
Sets the guest physical frame number at which the Xen "shared info" page resides. Note that although Xen places vcpu_info for the first 32 vCPUs in the shared_info page, KVM does not automatically do so and instead requires that KVM_XEN_VCPU_ATTR_TYPE_VCPU_INFO be used explicitly even when the vcpu_info for a given vCPU resides at the "default" location in the shared_info page. This is because KVM may not be aware of the Xen CPU id which is used as the index into the vcpu_info[] array, so may know the correct default location.

Note that the shared info page may be constantly written to by KVM; it contains the event channel bitmap used to deliver interrupts to a Xen guest, amongst other things. It is exempt from dirty tracking mechanisms — KVM will not explicitly mark the page as dirty each time an event channel interrupt is delivered to the guest! Thus, userspace should always assume that the designated GFN is dirty if any vCPU has been running or any event channel interrupts can be routed to the guest.

Setting the gfn to KVM_XEN_INVALID_GFN will disable the shared info page.

KVM_XEN_ATTR_TYPE_UPCALL_VECTOR
Sets the exception vector used to deliver Xen event channel upcalls. This is the HVM-wide vector injected directly by the hypervisor (not through the local APIC), typically configured by a guest via HVM_PARAM_CALLBACK_IRQ. This can be disabled again (e.g. for guest SHUTDOWN_soft_reset) by setting it to zero.

KVM_XEN_ATTR_TYPE_EVTCHN
This attribute is available when the KVM_CAP_XEN_HVM ioctl indicates support for KVM_XEN_HVM_CONFIG_EVTCHN_SEND features. It configures an outbound port number for interception of EVTCHNOP_send requests from the guest. A given sending port number may be directed back to a specified vCPU (by APIC ID) / port / priority on the guest, or to trigger events on an eventfd. The vCPU and priority can be changed by setting KVM_XEN_EVTCHN_UPDATE in a subsequent call, but other fields cannot change for a given sending port. A port mapping is removed by using KVM_XEN_EVTCHN_DEASSIGN in the flags field. Passing KVM_XEN_EVTCHN_RESET in the flags field removes all interception of outbound event channels. The values of the flags field are mutually exclusive and cannot be combined as a bitmask.

KVM_XEN_ATTR_TYPE_XEN_VERSION
This attribute is available when the KVM_CAP_XEN_HVM ioctl indicates support for KVM_XEN_HVM_CONFIG_EVTCHN_SEND features. It configures the 32-bit version code returned to the guest when it invokes the XENVER_version call; typically (XEN_MAJOR

<< 16 | XEN_MINOR). PV Xen guests will often use this to as a dummy hypercall to trigger event channel delivery, so responding within the kernel without exiting to userspace is beneficial.

KVM_XEN_ATTR_TYPE_RUNSTATE_UPDATE_FLAG
This attribute is available when the KVM_CAP_XEN_HVM ioctl indicates support for KVM_XEN_HVM_CONFIG_RUNSTATE_UPDATE_FLAG. It enables the XEN_RUNSTATE_UPDATE flag which allows guest vCPUs to safely read other vCPUs' vcpu_runstate_info. Xen guests enable this feature via the VMASST_TYPE_runstate_update_flag of the HYPERVISOR_vm_assist hypercall.

4.127 KVM_XEN_HVM_GET_ATTR

Capability
KVM_CAP_XEN_HVM / KVM_XEN_HVM_CONFIG_SHARED_INFO

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_xen_hvm_attr

Returns
0 on success, < 0 on error

Allows Xen VM attributes to be read. For the structure and types, see KVM_XEN_HVM_SET_ATTR above. The KVM_XEN_ATTR_TYPE_EVTCHN attribute cannot be read.

4.128 KVM_XEN_VCPU_SET_ATTR

Capability
KVM_CAP_XEN_HVM / KVM_XEN_HVM_CONFIG_SHARED_INFO

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xen_vcpu_attr

Returns
0 on success, < 0 on error

```
struct kvm_xen_vcpu_attr {
      __u16 type;
      __u16 pad[3];
      union {
              __u64 gpa;
              __u64 pad[4];
```

```
                struct {
                        __u64 state;
                        __u64 state_entry_time;
                        __u64 time_running;
                        __u64 time_runnable;
                        __u64 time_blocked;
                        __u64 time_offline;
                } runstate;
                __u32 vcpu_id;
                struct {
                        __u32 port;
                        __u32 priority;
                        __u64 expires_ns;
                } timer;
                __u8 vector;
        } u;
};
```

type values:

KVM_XEN_VCPU_ATTR_TYPE_VCPU_INFO
Sets the guest physical address of the vcpu_info for a given vCPU. As with the shared_info page for the VM, the corresponding page may be dirtied at any time if event channel interrupt delivery is enabled, so userspace should always assume that the page is dirty without relying on dirty logging. Setting the gpa to KVM_XEN_INVALID_GPA will disable the vcpu_info.

KVM_XEN_VCPU_ATTR_TYPE_VCPU_TIME_INFO
Sets the guest physical address of an additional pvclock structure for a given vCPU. This is typically used for guest vsyscall support. Setting the gpa to KVM_XEN_INVALID_GPA will disable the structure.

KVM_XEN_VCPU_ATTR_TYPE_RUNSTATE_ADDR
Sets the guest physical address of the vcpu_runstate_info for a given vCPU. This is how a Xen guest tracks CPU state such as steal time. Setting the gpa to KVM_XEN_INVALID_GPA will disable the runstate area.

KVM_XEN_VCPU_ATTR_TYPE_RUNSTATE_CURRENT
Sets the runstate (RUNSTATE_running/_runnable/_blocked/_offline) of the given vCPU from the .u.runstate.state member of the structure. KVM automatically accounts running and runnable time but blocked and offline states are only entered explicitly.

KVM_XEN_VCPU_ATTR_TYPE_RUNSTATE_DATA
Sets all fields of the vCPU runstate data from the .u.runstate member of the structure, including the current runstate. The state_entry_time must equal the sum of the other four times.

KVM_XEN_VCPU_ATTR_TYPE_RUNSTATE_ADJUST
This *adds* the contents of the .u.runstate members of the structure to the corresponding members of the given vCPU's runstate data, thus permitting atomic adjustments to the runstate times. The adjustment to the state_entry_time must equal the sum of the adjustments to the other four times. The state field must be set to -1, or to a valid runstate value (RUNSTATE_running, RUNSTATE_runnable, RUNSTATE_blocked or RUNSTATE_offline) to set the current accounted state as of the adjusted state_entry_time.

KVM_XEN_VCPU_ATTR_TYPE_VCPU_ID
This attribute is available when the KVM_CAP_XEN_HVM ioctl indicates support for KVM_XEN_HVM_CONFIG_EVTCHN_SEND features. It sets the Xen vCPU ID of the given vCPU, to allow timer-related VCPU operations to be intercepted by KVM.

KVM_XEN_VCPU_ATTR_TYPE_TIMER
This attribute is available when the KVM_CAP_XEN_HVM ioctl indicates support for KVM_XEN_HVM_CONFIG_EVTCHN_SEND features. It sets the event channel port/priority for the VIRQ_TIMER of the vCPU, as well as allowing a pending timer to be saved/restored. Setting the timer port to zero disables kernel handling of the singleshot timer.

KVM_XEN_VCPU_ATTR_TYPE_UPCALL_VECTOR
This attribute is available when the KVM_CAP_XEN_HVM ioctl indicates support for KVM_XEN_HVM_CONFIG_EVTCHN_SEND features. It sets the per-vCPU local APIC upcall vector, configured by a Xen guest with the HVMOP_set_evtchn_upcall_vector hypercall. This is typically used by Windows guests, and is distinct from the HVM-wide upcall vector configured with HVM_PARAM_CALLBACK_IRQ. It is disabled by setting the vector to zero.

4.129 KVM_XEN_VCPU_GET_ATTR

Capability
KVM_CAP_XEN_HVM / KVM_XEN_HVM_CONFIG_SHARED_INFO

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xen_vcpu_attr

Returns
0 on success, < 0 on error

Allows Xen vCPU attributes to be read. For the structure and types, see KVM_XEN_VCPU_SET_ATTR above.

The KVM_XEN_VCPU_ATTR_TYPE_RUNSTATE_ADJUST type may not be used with the KVM_XEN_VCPU_GET_ATTR ioctl.

4.130 KVM_ARM_MTE_COPY_TAGS

Capability
KVM_CAP_ARM_MTE

Architectures
arm64

Type
vm ioctl

Parameters
struct kvm_arm_copy_mte_tags

Returns
number of bytes copied, < 0 on error (-EINVAL for incorrect arguments, -EFAULT if memory cannot be accessed).

```
struct kvm_arm_copy_mte_tags {
      __u64 guest_ipa;
      __u64 length;
      void __user *addr;
      __u64 flags;
      __u64 reserved[2];
};
```

Copies Memory Tagging Extension (MTE) tags to/from guest tag memory. The `guest_ipa` and `length` fields must be `PAGE_SIZE` aligned. `length` must not be bigger than 2^31 - PAGE_SIZE bytes. The `addr` field must point to a buffer which the tags will be copied to or from.

`flags` specifies the direction of copy, either `KVM_ARM_TAGS_TO_GUEST` or `KVM_ARM_TAGS_FROM_GUEST`.

The size of the buffer to store the tags is `(length / 16)` bytes (granules in MTE are 16 bytes long). Each byte contains a single tag value. This matches the format of `PTRACE_PEEKMTETAGS` and `PTRACE_POKEMTETAGS`.

If an error occurs before any data is copied then a negative error code is returned. If some tags have been copied before an error occurs then the number of bytes successfully copied is returned. If the call completes successfully then `length` is returned.

4.131 KVM_GET_SREGS2

Capability
KVM_CAP_SREGS2

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_sregs2 (out)

Returns
0 on success, -1 on error

Reads special registers from the vcpu. This ioctl (when supported) replaces the KVM_GET_SREGS.

```
struct kvm_sregs2 {
        /* out (KVM_GET_SREGS2) / in (KVM_SET_SREGS2) */
        struct kvm_segment cs, ds, es, fs, gs, ss;
        struct kvm_segment tr, ldt;
        struct kvm_dtable gdt, idt;
        __u64 cr0, cr2, cr3, cr4, cr8;
        __u64 efer;
        __u64 apic_base;
```

```
        __u64 flags;
        __u64 pdptrs[4];
};
```

flags values for `kvm_sregs2`:

`KVM_SREGS2_FLAGS_PDPTRS_VALID`

Indicates that the struct contains valid PDPTR values.

4.132 KVM_SET_SREGS2

Capability
KVM_CAP_SREGS2

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_sregs2 (in)

Returns
0 on success, -1 on error

Writes special registers into the vcpu. See KVM_GET_SREGS2 for the data structures. This ioctl (when supported) replaces the KVM_SET_SREGS.

4.133 KVM_GET_STATS_FD

Capability
KVM_CAP_STATS_BINARY_FD

Architectures
all

Type
vm ioctl, vcpu ioctl

Parameters
none

Returns
statistics file descriptor on success, < 0 on error

Errors:

ENOMEM	if the fd could not be created due to lack of memory
EMFILE	if the number of opened files exceeds the limit

The returned file descriptor can be used to read VM/vCPU statistics data in binary format. The data in the file descriptor consists of four blocks organized as follows:

Header
id string
Descriptors
Stats Data

Apart from the header starting at offset 0, please be aware that it is not guaranteed that the four blocks are adjacent or in the above order; the offsets of the id, descriptors and data blocks are found in the header. However, all four blocks are aligned to 64 bit offsets in the file and they do not overlap.

All blocks except the data block are immutable. Userspace can read them only one time after retrieving the file descriptor, and then use `pread` or `lseek` to read the statistics repeatedly.

All data is in system endianness.

The format of the header is as follows:

```
struct kvm_stats_header {
        __u32 flags;
        __u32 name_size;
        __u32 num_desc;
        __u32 id_offset;
        __u32 desc_offset;
        __u32 data_offset;
};
```

The `flags` field is not used at the moment. It is always read as 0.

The `name_size` field is the size (in byte) of the statistics name string (including trailing '0') which is contained in the ”id string” block and appended at the end of every descriptor.

The `num_desc` field is the number of descriptors that are included in the descriptor block. (The actual number of values in the data block may be larger, since each descriptor may comprise more than one value).

The `id_offset` field is the offset of the id string from the start of the file indicated by the file descriptor. It is a multiple of 8.

The `desc_offset` field is the offset of the Descriptors block from the start of the file indicated by the file descriptor. It is a multiple of 8.

The `data_offset` field is the offset of the Stats Data block from the start of the file indicated by the file descriptor. It is a multiple of 8.

The id string block contains a string which identifies the file descriptor on which KVM_GET_STATS_FD was invoked. The size of the block, including the trailing `'\0'`, is indicated by the `name_size` field in the header.

The descriptors block is only needed to be read once for the lifetime of the file descriptor contains a sequence of `struct kvm_stats_desc`, each followed by a string of size `name_size`.

```
#define KVM_STATS_TYPE_SHIFT            0
#define KVM_STATS_TYPE_MASK             (0xF << KVM_STATS_TYPE_SHIFT)
#define KVM_STATS_TYPE_CUMULATIVE       (0x0 << KVM_STATS_TYPE_SHIFT)
#define KVM_STATS_TYPE_INSTANT          (0x1 << KVM_STATS_TYPE_SHIFT)
```

```
#define KVM_STATS_TYPE_PEAK             (0x2 << KVM_STATS_TYPE_SHIFT)
#define KVM_STATS_TYPE_LINEAR_HIST      (0x3 << KVM_STATS_TYPE_SHIFT)
#define KVM_STATS_TYPE_LOG_HIST         (0x4 << KVM_STATS_TYPE_SHIFT)
#define KVM_STATS_TYPE_MAX              KVM_STATS_TYPE_LOG_HIST

#define KVM_STATS_UNIT_SHIFT            4
#define KVM_STATS_UNIT_MASK             (0xF << KVM_STATS_UNIT_SHIFT)
#define KVM_STATS_UNIT_NONE             (0x0 << KVM_STATS_UNIT_SHIFT)
#define KVM_STATS_UNIT_BYTES            (0x1 << KVM_STATS_UNIT_SHIFT)
#define KVM_STATS_UNIT_SECONDS          (0x2 << KVM_STATS_UNIT_SHIFT)
#define KVM_STATS_UNIT_CYCLES           (0x3 << KVM_STATS_UNIT_SHIFT)
#define KVM_STATS_UNIT_BOOLEAN          (0x4 << KVM_STATS_UNIT_SHIFT)
#define KVM_STATS_UNIT_MAX              KVM_STATS_UNIT_BOOLEAN

#define KVM_STATS_BASE_SHIFT            8
#define KVM_STATS_BASE_MASK             (0xF << KVM_STATS_BASE_SHIFT)
#define KVM_STATS_BASE_POW10            (0x0 << KVM_STATS_BASE_SHIFT)
#define KVM_STATS_BASE_POW2             (0x1 << KVM_STATS_BASE_SHIFT)
#define KVM_STATS_BASE_MAX              KVM_STATS_BASE_POW2

struct kvm_stats_desc {
        __u32 flags;
        __s16 exponent;
        __u16 size;
        __u32 offset;
        __u32 bucket_size;
        char name[];
};
```

The `flags` field contains the type and unit of the statistics data described by this descriptor. Its endianness is CPU native. The following flags are supported:

Bits 0-3 of `flags` encode the type:

- `KVM_STATS_TYPE_CUMULATIVE` The statistics reports a cumulative count. The value of data can only be increased. Most of the counters used in KVM are of this type. The corresponding `size` field for this type is always 1. All cumulative statistics data are read/write.
- `KVM_STATS_TYPE_INSTANT` The statistics reports an instantaneous value. Its value can be increased or decreased. This type is usually used as a measurement of some resources, like the number of dirty pages, the number of large pages, etc. All instant statistics are read only. The corresponding `size` field for this type is always 1.
- `KVM_STATS_TYPE_PEAK` The statistics data reports a peak value, for example the maximum number of items in a hash table bucket, the longest time waited and so on. The value of data can only be increased. The corresponding `size` field for this type is always 1.
- `KVM_STATS_TYPE_LINEAR_HIST` The statistic is reported as a linear histogram. The number of buckets is specified by the `size` field. The size of buckets is specified by the `hist_param` field. The range of the Nth bucket (1 <= N < `size`) is [`hist_param``*(N-1), ``hist_param``*N), while the range of the last bucket is [``hist_param``*(``size-1), +INF). (+INF means positive infinity value.)
- `KVM_STATS_TYPE_LOG_HIST` The statistic is reported as a logarithmic histogram. The num-

ber of buckets is specified by the `size` field. The range of the first bucket is [0, 1), while the range of the last bucket is [pow(2, `size`-2), +INF). Otherwise, The Nth bucket (1 < N < `size`) covers [pow(2, N-2), pow(2, N-1)).

Bits 4-7 of `flags` encode the unit:

- `KVM_STATS_UNIT_NONE` There is no unit for the value of statistics data. This usually means that the value is a simple counter of an event.
- `KVM_STATS_UNIT_BYTES` It indicates that the statistics data is used to measure memory size, in the unit of Byte, KiByte, MiByte, GiByte, etc. The unit of the data is determined by the `exponent` field in the descriptor.
- `KVM_STATS_UNIT_SECONDS` It indicates that the statistics data is used to measure time or latency.
- `KVM_STATS_UNIT_CYCLES` It indicates that the statistics data is used to measure CPU clock cycles.
- `KVM_STATS_UNIT_BOOLEAN` It indicates that the statistic will always be either 0 or 1. Boolean statistics of ”peak” type will never go back from 1 to 0. Boolean statistics can be linear histograms (with two buckets) but not logarithmic histograms.

Note that, in the case of histograms, the unit applies to the bucket ranges, while the bucket value indicates how many samples fell in the bucket's range.

Bits 8-11 of `flags`, together with `exponent`, encode the scale of the unit:

- `KVM_STATS_BASE_POW10` The scale is based on power of 10. It is used for measurement of time and CPU clock cycles. For example, an exponent of -9 can be used with `KVM_STATS_UNIT_SECONDS` to express that the unit is nanoseconds.
- `KVM_STATS_BASE_POW2` The scale is based on power of 2. It is used for measurement of memory size. For example, an exponent of 20 can be used with `KVM_STATS_UNIT_BYTES` to express that the unit is MiB.

The `size` field is the number of values of this statistics data. Its value is usually 1 for most of simple statistics. 1 means it contains an unsigned 64bit data.

The `offset` field is the offset from the start of Data Block to the start of the corresponding statistics data.

The `bucket_size` field is used as a parameter for histogram statistics data. It is only used by linear histogram statistics data, specifying the size of a bucket in the unit expressed by bits 4-11 of `flags` together with `exponent`.

The `name` field is the name string of the statistics data. The name string starts at the end of `struct kvm_stats_desc`. The maximum length including the trailing `'\0'`, is indicated by `name_size` in the header.

The Stats Data block contains an array of 64-bit values in the same order as the descriptors in Descriptors block.

4.134 KVM_GET_XSAVE2

Capability
KVM_CAP_XSAVE2

Architectures
x86

Type
vcpu ioctl

Parameters
struct kvm_xsave (out)

Returns
0 on success, -1 on error

```
struct kvm_xsave {
      __u32 region[1024];
      __u32 extra[0];
};
```

This ioctl would copy current vcpu's xsave struct to the userspace. It copies as many bytes as are returned by KVM_CHECK_EXTENSION(KVM_CAP_XSAVE2) when invoked on the vm file descriptor. The size value returned by KVM_CHECK_EXTENSION(KVM_CAP_XSAVE2) will always be at least 4096. Currently, it is only greater than 4096 if a dynamic feature has been enabled with `arch_prctl()`, but this may change in the future.

The offsets of the state save areas in struct kvm_xsave follow the contents of CPUID leaf 0xD on the host.

4.135 KVM_XEN_HVM_EVTCHN_SEND

Capability
KVM_CAP_XEN_HVM / KVM_XEN_HVM_CONFIG_EVTCHN_SEND

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_irq_routing_xen_evtchn

Returns
0 on success, < 0 on error

```
struct kvm_irq_routing_xen_evtchn {
     __u32 port;
     __u32 vcpu;
     __u32 priority;
};
```

This ioctl injects an event channel interrupt directly to the guest vCPU.

4.136 KVM_S390_PV_CPU_COMMAND

Capability
KVM_CAP_S390_PROTECTED_DUMP

Architectures
s390

Type
vcpu ioctl

Parameters
none

Returns
0 on success, < 0 on error

This ioctl closely mirrors *KVM_S390_PV_COMMAND* but handles requests for vcpus. It re-uses the kvm_s390_pv_dmp struct and hence also shares the command ids.

command:

KVM_PV_DUMP
Presents an API that provides calls which facilitate dumping a vcpu of a protected VM.

subcommand:

KVM_PV_DUMP_CPU
Provides encrypted dump data like register values. The length of the returned data is provided by uv_info.guest_cpu_stor_len.

4.137 KVM_S390_ZPCI_OP

Capability
KVM_CAP_S390_ZPCI_OP

Architectures
s390

Type
vm ioctl

Parameters
struct kvm_s390_zpci_op (in)

Returns
0 on success, <0 on error

Used to manage hardware-assisted virtualization features for zPCI devices.

Parameters are specified via the following structure:

```
struct kvm_s390_zpci_op {
      /* in */
      __u32 fh;               /* target device */
      __u8  op;               /* operation to perform */
      __u8  pad[3];
      union {
```

```
                /* for KVM_S390_ZPCIOP_REG_AEN */
                struct {
                        __u64 ibv;      /* Guest addr of interrupt bit vector */
                        __u64 sb;       /* Guest addr of summary bit */
                        __u32 flags;
                        __u32 noi;      /* Number of interrupts */
                        __u8 isc;       /* Guest interrupt subclass */
                        __u8 sbo;       /* Offset of guest summary bit vector */
                        __u16 pad;
                } reg_aen;
                __u64 reserved[8];
        } u;
};
```

The type of operation is specified in the ”op” field. KVM_S390_ZPCIOP_REG_AEN is used to register the VM for adapter event notification interpretation, which will allow firmware delivery of adapter events directly to the vm, with KVM providing a backup delivery mechanism; KVM_S390_ZPCIOP_DEREG_AEN is used to subsequently disable interpretation of adapter event notifications.

The target zPCI function must also be specified via the ”fh” field. For the KVM_S390_ZPCIOP_REG_AEN operation, additional information to establish firmware delivery must be provided via the ”reg_aen” struct.

The ”pad” and ”reserved” fields may be used for future extensions and should be set to 0s by userspace.

4.138 KVM_ARM_SET_COUNTER_OFFSET

Capability
KVM_CAP_COUNTER_OFFSET

Architectures
arm64

Type
vm ioctl

Parameters
struct kvm_arm_counter_offset (in)

Returns
0 on success, < 0 on error

This capability indicates that userspace is able to apply a single VM-wide offset to both the virtual and physical counters as viewed by the guest using the KVM_ARM_SET_CNT_OFFSET ioctl and the following data structure:

```
struct kvm_arm_counter_offset {
        __u64 counter_offset;
        __u64 reserved;
};
```

The offset describes a number of counter cycles that are subtracted from both virtual and physical counter views (similar to the effects of the CNTVOFF_EL2 and CNTPOFF_EL2 system registers, but only global). The offset always applies to all vcpus (already created or created after this ioctl) for this VM.

It is userspace's responsibility to compute the offset based, for example, on previous values of the guest counters.

Any value other than 0 for the ”reserved” field may result in an error (-EINVAL) being returned. This ioctl can also return -EBUSY if any vcpu ioctl is issued concurrently.

Note that using this ioctl results in KVM ignoring subsequent userspace writes to the CNTVCT_EL0 and CNTPCT_EL0 registers using the SET_ONE_REG interface. No error will be returned, but the resulting offset will not be applied.

4.139 KVM_ARM_GET_REG_WRITABLE_MASKS

Capability
KVM_CAP_ARM_SUPPORTED_REG_MASK_RANGES

Architectures
arm64

Type
vm ioctl

Parameters
struct reg_mask_range (in/out)

Returns
0 on success, < 0 on error

```
#define KVM_ARM_FEATURE_ID_RANGE        0
#define KVM_ARM_FEATURE_ID_RANGE_SIZE   (3 * 8 * 8)

struct reg_mask_range {
        __u64 addr;             /* Pointer to mask array */
        __u32 range;            /* Requested range */
        __u32 reserved[13];
};
```

This ioctl copies the writable masks for a selected range of registers to userspace.

The `addr` field is a pointer to the destination array where KVM copies the writable masks.

The `range` field indicates the requested range of registers. `KVM_CHECK_EXTENSION` for the `KVM_CAP_ARM_SUPPORTED_REG_MASK_RANGES` capability returns the supported ranges, expressed as a set of flags. Each flag's bit index represents a possible value for the `range` field. All other values are reserved for future use and KVM may return an error.

The `reserved[13]` array is reserved for future use and should be 0, or KVM may return an error.

KVM_ARM_FEATURE_ID_RANGE (0)

The Feature ID range is defined as the AArch64 System register space with op0==3, op1=={0, 1, 3}, CRn==0, CRm=={0-7}, op2=={0-7}.

The mask returned array pointed to by `addr` is indexed by the macro `ARM64_FEATURE_ID_RANGE_IDX(op0, op1, crn, crm, op2)`, allowing userspace to know what fields can be changed for the system register described by `op0, op1, crn, crm, op2`. KVM rejects ID register values that describe a superset of the features supported by the system.

4.140 KVM_SET_USER_MEMORY_REGION2

Capability
KVM_CAP_USER_MEMORY2

Architectures
all

Type
vm ioctl

Parameters
struct kvm_userspace_memory_region2 (in)

Returns
0 on success, -1 on error

KVM_SET_USER_MEMORY_REGION2 is an extension to KVM_SET_USER_MEMORY_REGION that allows mapping guest_memfd memory into a guest. All fields shared with KVM_SET_USER_MEMORY_REGION identically. Userspace can set KVM_MEM_GUEST_MEMFD in flags to have KVM bind the memory region to a given guest_memfd range of [guest_memfd_offset, guest_memfd_offset + memory_size]. The target guest_memfd must point at a file created via KVM_CREATE_GUEST_MEMFD on the current VM, and the target range must not be bound to any other memory region. All standard bounds checks apply (use common sense).

```
struct kvm_userspace_memory_region2 {
      __u32 slot;
      __u32 flags;
      __u64 guest_phys_addr;
      __u64 memory_size; /* bytes */
      __u64 userspace_addr; /* start of the userspace allocated memory */
      __u64 guest_memfd_offset;
      __u32 guest_memfd;
      __u32 pad1;
      __u64 pad2[14];
};
```

A KVM_MEM_GUEST_MEMFD region _must_ have a valid guest_memfd (private memory) and userspace_addr (shared memory). However, ”valid” for userspace_addr simply means that the address itself must be a legal userspace address. The backing mapping for userspace_addr is not required to be valid/populated at the time of KVM_SET_USER_MEMORY_REGION2, e.g. shared memory can be lazily mapped/allocated on-demand.

When mapping a gfn into the guest, KVM selects shared vs. private, i.e consumes userspace_addr vs. guest_memfd, based on the gfn's KVM_MEMORY_ATTRIBUTE_PRIVATE state. At VM creation time, all memory is shared, i.e. the PRIVATE attribute is '0' for all gfns. Userspace can control whether memory is shared/private by toggling KVM_MEMORY_ATTRIBUTE_PRIVATE via KVM_SET_MEMORY_ATTRIBUTES as needed.

4.141 KVM_SET_MEMORY_ATTRIBUTES

Capability
KVM_CAP_MEMORY_ATTRIBUTES

Architectures
x86

Type
vm ioctl

Parameters
struct kvm_memory_attributes (in)

Returns
0 on success, <0 on error

KVM_SET_MEMORY_ATTRIBUTES allows userspace to set memory attributes for a range of guest physical memory.

```
struct kvm_memory_attributes {
      __u64 address;
      __u64 size;
      __u64 attributes;
      __u64 flags;
};

#define KVM_MEMORY_ATTRIBUTE_PRIVATE           (1ULL << 3)
```

The address and size must be page aligned. The supported attributes can be retrieved via ioctl(KVM_CHECK_EXTENSION) on KVM_CAP_MEMORY_ATTRIBUTES. If executed on a VM, KVM_CAP_MEMORY_ATTRIBUTES precisely returns the attributes supported by that VM. If executed at system scope, KVM_CAP_MEMORY_ATTRIBUTES returns all attributes supported by KVM. The only attribute defined at this time is KVM_MEMORY_ATTRIBUTE_PRIVATE, which marks the associated gfn as being guest private memory.

Note, there is no ”get” API. Userspace is responsible for explicitly tracking the state of a gfn/page as needed.

The ”flags” field is reserved for future extensions and must be '0'.

4.142 KVM_CREATE_GUEST_MEMFD

Capability
KVM_CAP_GUEST_MEMFD

Architectures
none

Type
vm ioctl

Parameters
struct kvm_create_guest_memfd(in)

Returns
0 on success, <0 on error

KVM_CREATE_GUEST_MEMFD creates an anonymous file and returns a file descriptor that refers to it. guest_memfd files are roughly analogous to files created via memfd_create(), e.g. guest_memfd files live in RAM, have volatile storage, and are automatically released when the last reference is dropped. Unlike "regular" memfd_create() files, guest_memfd files are bound to their owning virtual machine (see below), cannot be mapped, read, or written by userspace, and cannot be resized (guest_memfd files do however support PUNCH_HOLE).

```
struct kvm_create_guest_memfd {
      __u64 size;
      __u64 flags;
      __u64 reserved[6];
};
```

Conceptually, the inode backing a guest_memfd file represents physical memory, i.e. is coupled to the virtual machine as a thing, not to a "struct kvm". The file itself, which is bound to a "struct kvm", is that instance's view of the underlying memory, e.g. effectively provides the translation of guest addresses to host memory. This allows for use cases where multiple KVM structures are used to manage a single virtual machine, e.g. when performing intrahost migration of a virtual machine.

KVM currently only supports mapping guest_memfd via KVM_SET_USER_MEMORY_REGION2, and more specifically via the guest_memfd and guest_memfd_offset fields in "struct kvm_userspace_memory_region2", where guest_memfd_offset is the offset into the guest_memfd instance. For a given guest_memfd file, there can be at most one mapping per page, i.e. binding multiple memory regions to a single guest_memfd range is not allowed (any number of memory regions can be bound to a single guest_memfd file, but the bound ranges must not overlap).

See KVM_SET_USER_MEMORY_REGION2 for additional details.

1.1.5 5. The kvm_run structure

Application code obtains a pointer to the kvm_run structure by mmap()ing a vcpu fd. From that point, application code can control execution by changing fields in kvm_run prior to calling the KVM_RUN ioctl, and obtain information about the reason KVM_RUN returned by looking up structure members.

```
struct kvm_run {
      /* in */
      __u8 request_interrupt_window;
```

Request that KVM_RUN return when it becomes possible to inject external interrupts into the guest. Useful in conjunction with KVM_INTERRUPT.

```
__u8 immediate_exit;
```

This field is polled once when KVM_RUN starts; if non-zero, KVM_RUN exits immediately, returning -EINTR. In the common scenario where a signal is used to ”kick” a VCPU out of KVM_RUN, this field can be used to avoid usage of KVM_SET_SIGNAL_MASK, which has worse scalability. Rather than blocking the signal outside KVM_RUN, userspace can set up a signal handler that sets run->immediate_exit to a non-zero value.

This field is ignored if KVM_CAP_IMMEDIATE_EXIT is not available.

```
__u8 padding1[6];

/* out */
__u32 exit_reason;
```

When KVM_RUN has returned successfully (return value 0), this informs application code why KVM_RUN has returned. Allowable values for this field are detailed below.

```
__u8 ready_for_interrupt_injection;
```

If request_interrupt_window has been specified, this field indicates an interrupt can be injected now with KVM_INTERRUPT.

```
__u8 if_flag;
```

The value of the current interrupt flag. Only valid if in-kernel local APIC is not used.

```
__u16 flags;
```

More architecture-specific flags detailing state of the VCPU that may affect the device's behavior. Current defined flags:

```
/* x86, set if the VCPU is in system management mode */
#define KVM_RUN_X86_SMM     (1 << 0)
/* x86, set if bus lock detected in VM */
#define KVM_RUN_BUS_LOCK    (1 << 1)
/* arm64, set for KVM_EXIT_DEBUG */
#define KVM_DEBUG_ARCH_HSR_HIGH_VALID  (1 << 0)
```

```
/* in (pre_kvm_run), out (post_kvm_run) */
__u64 cr8;
```

The value of the cr8 register. Only valid if in-kernel local APIC is not used. Both input and output.

```
__u64 apic_base;
```

The value of the APIC BASE msr. Only valid if in-kernel local APIC is not used. Both input and output.

```
union {
        /* KVM_EXIT_UNKNOWN */
        struct {
                __u64 hardware_exit_reason;
        } hw;
```

If exit_reason is KVM_EXIT_UNKNOWN, the vcpu has exited due to unknown reasons. Further architecture-specific information is available in hardware_exit_reason.

```
/* KVM_EXIT_FAIL_ENTRY */
struct {
        __u64 hardware_entry_failure_reason;
        __u32 cpu; /* if KVM_LAST_CPU */
} fail_entry;
```

If exit_reason is KVM_EXIT_FAIL_ENTRY, the vcpu could not be run due to unknown reasons. Further architecture-specific information is available in hardware_entry_failure_reason.

```
/* KVM_EXIT_EXCEPTION */
struct {
        __u32 exception;
        __u32 error_code;
} ex;
```

Unused.

```
              /* KVM_EXIT_IO */
              struct {
#define KVM_EXIT_IO_IN  0
#define KVM_EXIT_IO_OUT 1
                      __u8 direction;
                      __u8 size; /* bytes */
                      __u16 port;
                      __u32 count;
                      __u64 data_offset; /* relative to kvm_run start */
              } io;
```

If exit_reason is KVM_EXIT_IO, then the vcpu has executed a port I/O instruction which could not be satisfied by kvm. data_offset describes where the data is located (KVM_EXIT_IO_OUT) or where kvm expects application code to place the data for the next KVM_RUN invocation (KVM_EXIT_IO_IN). Data format is a packed array.

```
/* KVM_EXIT_DEBUG */
struct {
        struct kvm_debug_exit_arch arch;
} debug;
```

If the exit_reason is KVM_EXIT_DEBUG, then a vcpu is processing a debug event for which architecture specific information is returned.

```
/* KVM_EXIT_MMIO */
struct {
        __u64 phys_addr;
        __u8  data[8];
        __u32 len;
        __u8  is_write;
} mmio;
```

If exit_reason is KVM_EXIT_MMIO, then the vcpu has executed a memory-mapped I/O instruction which could not be satisfied by kvm. The 'data' member contains the written data if 'is_write' is true, and should be filled by application code otherwise.

The 'data' member contains, in its first 'len' bytes, the value as it would appear if the VCPU performed a load or store of the appropriate width directly to the byte array.

Note: For KVM_EXIT_IO, KVM_EXIT_MMIO, KVM_EXIT_OSI, KVM_EXIT_PAPR, KVM_EXIT_XEN, KVM_EXIT_EPR, KVM_EXIT_X86_RDMSR and KVM_EXIT_X86_WRMSR the corresponding operations are complete (and guest state is consistent) only after userspace has re-entered the kernel with KVM_RUN. The kernel side will first finish incomplete operations and then check for pending signals.

The pending state of the operation is not preserved in state which is visible to userspace, thus userspace should ensure that the operation is completed before performing a live migration. Userspace can re-enter the guest with an unmasked signal pending or with the immediate_exit field set to complete pending operations without allowing any further instructions to be executed.

```
/* KVM_EXIT_HYPERCALL */
struct {
        __u64 nr;
        __u64 args[6];
        __u64 ret;
        __u64 flags;
} hypercall;
```

It is strongly recommended that userspace use `KVM_EXIT_IO` (x86) or `KVM_EXIT_MMIO` (all except s390) to implement functionality that requires a guest to interact with host userspace.

Note: KVM_EXIT_IO is significantly faster than KVM_EXIT_MMIO.

For arm64:

SMCCC exits can be enabled depending on the configuration of the SMCCC filter. See the *Generic vm interface* KVM_ARM_SMCCC_FILTER for more details.

nr contains the function ID of the guest's SMCCC call. Userspace is expected to use the KVM_GET_ONE_REG ioctl to retrieve the call parameters from the vCPU's GPRs.

Definition of flags:

- KVM_HYPERCALL_EXIT_SMC: Indicates that the guest used the SMC conduit to initiate the SMCCC call. If this bit is 0 then the guest used the HVC conduit for the SMCCC call.
- KVM_HYPERCALL_EXIT_16BIT: Indicates that the guest used a 16bit instruction to initiate the SMCCC call. If this bit is 0 then the guest used a 32bit instruction. An AArch64 guest always has this bit set to 0.

At the point of exit, PC points to the instruction immediately following the trapping instruction.

```
/* KVM_EXIT_TPR_ACCESS */
struct {
        __u64 rip;
        __u32 is_write;
        __u32 pad;
} tpr_access;
```

To be documented (KVM_TPR_ACCESS_REPORTING).

```
/* KVM_EXIT_S390_SIEIC */
struct {
        __u8 icptcode;
        __u64 mask; /* psw upper half */
        __u64 addr; /* psw lower half */
        __u16 ipa;
        __u32 ipb;
} s390_sieic;
```

s390 specific.

```
              /* KVM_EXIT_S390_RESET */
#define KVM_S390_RESET_POR       1
#define KVM_S390_RESET_CLEAR     2
#define KVM_S390_RESET_SUBSYSTEM 4
#define KVM_S390_RESET_CPU_INIT  8
#define KVM_S390_RESET_IPL       16
              __u64 s390_reset_flags;
```

s390 specific.

```
/* KVM_EXIT_S390_UCONTROL */
struct {
        __u64 trans_exc_code;
        __u32 pgm_code;
} s390_ucontrol;
```

s390 specific. A page fault has occurred for a user controlled virtual machine (KVM_VM_S390_UNCONTROL) on its host page table that cannot be resolved by the kernel. The program code and the translation exception code that were placed in the cpu's lowcore are presented here as defined by the z Architecture Principles of Operation Book in the Chapter for Dynamic Address Translation (DAT)

```
/* KVM_EXIT_DCR */
struct {
        __u32 dcrn;
        __u32 data;
        __u8  is_write;
} dcr;
```

Deprecated - was used for 440 KVM.

```
/* KVM_EXIT_OSI */
struct {
        __u64 gprs[32];
} osi;
```

MOL uses a special hypercall interface it calls 'OSI'. To enable it, we catch hypercalls and exit with this exit struct that contains all the guest gprs.

If exit_reason is KVM_EXIT_OSI, then the vcpu has triggered such a hypercall. Userspace can now handle the hypercall and when it's done modify the gprs as necessary. Upon guest entry all guest GPRs will then be replaced by the values in this struct.

```
/* KVM_EXIT_PAPR_HCALL */
struct {
        __u64 nr;
        __u64 ret;
        __u64 args[9];
} papr_hcall;
```

This is used on 64-bit PowerPC when emulating a pSeries partition, e.g. with the 'pseries' machine type in qemu. It occurs when the guest does a hypercall using the 'sc 1' instruction. The 'nr' field contains the hypercall number (from the guest R3), and 'args' contains the arguments (from the guest R4 - R12). Userspace should put the return code in 'ret' and any extra returned values in args[]. The possible hypercalls are defined in the Power Architecture Platform Requirements (PAPR) document available from www.power.org (free developer registration required to access it).

```
/* KVM_EXIT_S390_TSCH */
struct {
        __u16 subchannel_id;
        __u16 subchannel_nr;
        __u32 io_int_parm;
        __u32 io_int_word;
        __u32 ipb;
        __u8 dequeued;
} s390_tsch;
```

s390 specific. This exit occurs when KVM_CAP_S390_CSS_SUPPORT has been enabled and TEST SUBCHANNEL was intercepted. If dequeued is set, a pending I/O interrupt for the target subchannel has been dequeued and subchannel_id, subchannel_nr, io_int_parm and io_int_word contain the parameters for that interrupt. ipb is needed for instruction parameter decoding.

```
/* KVM_EXIT_EPR */
struct {
        __u32 epr;
} epr;
```

On FSL BookE PowerPC chips, the interrupt controller has a fast patch interrupt acknowledge path to the core. When the core successfully delivers an interrupt, it automatically populates the EPR register with the interrupt vector number and acknowledges the interrupt inside the interrupt controller.

In case the interrupt controller lives in user space, we need to do the interrupt acknowledge cycle through it to fetch the next to be delivered interrupt vector using this exit.

It gets triggered whenever both KVM_CAP_PPC_EPR are enabled and an external interrupt has just been delivered into the guest. User space should put the acknowledged interrupt vector into the 'epr' field.

```
             /* KVM_EXIT_SYSTEM_EVENT */
             struct {
#define KVM_SYSTEM_EVENT_SHUTDOWN       1
#define KVM_SYSTEM_EVENT_RESET          2
#define KVM_SYSTEM_EVENT_CRASH          3
#define KVM_SYSTEM_EVENT_WAKEUP         4
#define KVM_SYSTEM_EVENT_SUSPEND        5
#define KVM_SYSTEM_EVENT_SEV_TERM       6
                     __u32 type;
                     __u32 ndata;
                     __u64 data[16];
             } system_event;
```

If exit_reason is KVM_EXIT_SYSTEM_EVENT then the vcpu has triggered a system-level event using some architecture specific mechanism (hypercall or some special instruction). In case of ARM64, this is triggered using HVC instruction based PSCI call from the vcpu.

The 'type' field describes the system-level event type. Valid values for 'type' are:

- KVM_SYSTEM_EVENT_SHUTDOWN -- the guest has requested a shutdown of the VM. Userspace is not obliged to honour this, and if it does honour this does not need to destroy the VM synchronously (ie it may call KVM_RUN again before shutdown finally occurs).
- KVM_SYSTEM_EVENT_RESET -- the guest has requested a reset of the VM. As with SHUTDOWN, userspace can choose to ignore the request, or to schedule the reset to occur in the future and may call KVM_RUN again.
- KVM_SYSTEM_EVENT_CRASH -- the guest crash occurred and the guest has requested a crash condition maintenance. Userspace can choose to ignore the request, or to gather VM memory core dump and/or reset/shutdown of the VM.
- KVM_SYSTEM_EVENT_SEV_TERM -- an AMD SEV guest requested termination. The guest physical address of the guest's GHCB is stored in *data[0]*.

- KVM_SYSTEM_EVENT_WAKEUP -- the exiting vCPU is in a suspended state and KVM has recognized a wakeup event. Userspace may honor this event by marking the exiting vCPU as runnable, or deny it and call KVM_RUN again.
- KVM_SYSTEM_EVENT_SUSPEND -- the guest has requested a suspension of the VM.

If KVM_CAP_SYSTEM_EVENT_DATA is present, the 'data' field can contain architecture specific information for the system-level event. Only the first *ndata* items (possibly zero) of the data array are valid.

- for arm64, data[0] is set to KVM_SYSTEM_EVENT_RESET_FLAG_PSCI_RESET2 if the guest issued a SYSTEM_RESET2 call according to v1.1 of the PSCI specification.
- for RISC-V, data[0] is set to the value of the second argument of the `sbi_system_reset` call.

Previous versions of Linux defined a *flags* member in this struct. The field is now aliased to *data[0]*. Userspace can assume that it is only written if ndata is greater than 0.

For arm/arm64:

KVM_SYSTEM_EVENT_SUSPEND exits are enabled with the KVM_CAP_ARM_SYSTEM_SUSPEND VM capability. If a guest invokes the PSCI SYSTEM_SUSPEND function, KVM will exit to userspace with this event type.

It is the sole responsibility of userspace to implement the PSCI SYSTEM_SUSPEND call according to ARM DEN0022D.b 5.19 ”SYSTEM_SUSPEND”. KVM does not change the vCPU's state before exiting to userspace, so the call parameters are left in-place in the vCPU registers.

Userspace is _required_ to take action for such an exit. It must either:

- Honor the guest request to suspend the VM. Userspace can request in-kernel emulation of suspension by setting the calling vCPU's state to KVM_MP_STATE_SUSPENDED. Userspace must configure the vCPU's state according to the parameters passed to the PSCI function when the calling vCPU is resumed. See ARM DEN0022D.b 5.19.1 ”Intended use” for details on the function parameters.
- Deny the guest request to suspend the VM. See ARM DEN0022D.b 5.19.2 ”Caller responsibilities” for possible return values.

```
/* KVM_EXIT_IOAPIC_EOI */
struct {
        __u8 vector;
} eoi;
```

Indicates that the VCPU's in-kernel local APIC received an EOI for a level-triggered IOAPIC interrupt. This exit only triggers when the IOAPIC is implemented in userspace (i.e. KVM_CAP_SPLIT_IRQCHIP is enabled); the userspace IOAPIC should process the EOI and retrigger the interrupt if it is still asserted. Vector is the LAPIC interrupt vector for which the EOI was received.

```
              struct kvm_hyperv_exit {
#define KVM_EXIT_HYPERV_SYNIC          1
#define KVM_EXIT_HYPERV_HCALL          2
#define KVM_EXIT_HYPERV_SYNDBG         3
```

```
                __u32 type;
                __u32 pad1;
                union {
                        struct {
                                __u32 msr;
                                __u32 pad2;
                                __u64 control;
                                __u64 evt_page;
                                __u64 msg_page;
                        } synic;
                        struct {
                                __u64 input;
                                __u64 result;
                                __u64 params[2];
                        } hcall;
                        struct {
                                __u32 msr;
                                __u32 pad2;
                                __u64 control;
                                __u64 status;
                                __u64 send_page;
                                __u64 recv_page;
                                __u64 pending_page;
                        } syndbg;
                } u;
        };
        /* KVM_EXIT_HYPERV */
        struct kvm_hyperv_exit hyperv;
```

Indicates that the VCPU exits into userspace to process some tasks related to Hyper-V emulation.

Valid values for 'type' are:

- KVM_EXIT_HYPERV_SYNIC -- synchronously notify user-space about

Hyper-V SynIC state change. Notification is used to remap SynIC event/message pages and to enable/disable SynIC messages/events processing in userspace.

- KVM_EXIT_HYPERV_SYNDBG -- synchronously notify user-space about

Hyper-V Synthetic debugger state change. Notification is used to either update the pending_page location or to send a control command (send the buffer located in send_page or recv a buffer to recv_page).

```
/* KVM_EXIT_ARM_NISV */
struct {
        __u64 esr_iss;
        __u64 fault_ipa;
} arm_nisv;
```

Used on arm64 systems. If a guest accesses memory not in a memslot, KVM will typically return to userspace and ask it to do MMIO emulation on its behalf. However, for certain classes

of instructions, no instruction decode (direction, length of memory access) is provided, and fetching and decoding the instruction from the VM is overly complicated to live in the kernel.

Historically, when this situation occurred, KVM would print a warning and kill the VM. KVM assumed that if the guest accessed non-memslot memory, it was trying to do I/O, which just couldn't be emulated, and the warning message was phrased accordingly. However, what happened more often was that a guest bug caused access outside the guest memory areas which should lead to a more meaningful warning message and an external abort in the guest, if the access did not fall within an I/O window.

Userspace implementations can query for KVM_CAP_ARM_NISV_TO_USER, and enable this capability at VM creation. Once this is done, these types of errors will instead return to userspace with KVM_EXIT_ARM_NISV, with the valid bits from the ESR_EL2 in the esr_iss field, and the faulting IPA in the fault_ipa field. Userspace can either fix up the access if it's actually an I/O access by decoding the instruction from guest memory (if it's very brave) and continue executing the guest, or it can decide to suspend, dump, or restart the guest.

Note that KVM does not skip the faulting instruction as it does for KVM_EXIT_MMIO, but userspace has to emulate any change to the processing state if it decides to decode and emulate the instruction.

```
/* KVM_EXIT_X86_RDMSR / KVM_EXIT_X86_WRMSR */
struct {
        __u8 error; /* user -> kernel */
        __u8 pad[7];
        __u32 reason; /* kernel -> user */
        __u32 index; /* kernel -> user */
        __u64 data; /* kernel <-> user */
} msr;
```

Used on x86 systems. When the VM capability KVM_CAP_X86_USER_SPACE_MSR is enabled, MSR accesses to registers that would invoke a #GP by KVM kernel code may instead trigger a KVM_EXIT_X86_RDMSR exit for reads and KVM_EXIT_X86_WRMSR exit for writes.

The ”reason” field specifies why the MSR interception occurred. Userspace will only receive MSR exits when a particular reason was requested during through ENABLE_CAP. Currently valid exit reasons are:

KVM_MSR_EXIT_REASON_UNKNOWN	access to MSR that is unknown to KVM
KVM_MSR_EXIT_REASON_INVAL	access to invalid MSRs or reserved bits
KVM_MSR_EXIT_REASON_FILTER	access blocked by KVM_X86_SET_MSR_FILTER

For KVM_EXIT_X86_RDMSR, the ”index” field tells userspace which MSR the guest wants to read. To respond to this request with a successful read, userspace writes the respective data into the ”data” field and must continue guest execution to ensure the read data is transferred into guest register state.

If the RDMSR request was unsuccessful, userspace indicates that with a ”1” in the ”error” field. This will inject a #GP into the guest when the VCPU is executed again.

For KVM_EXIT_X86_WRMSR, the ”index” field tells userspace which MSR the guest wants to write. Once finished processing the event, userspace must continue vCPU execution. If the MSR write was unsuccessful, userspace also sets the ”error” field to ”1”.

See KVM_X86_SET_MSR_FILTER for details on the interaction with MSR filtering.

```
                struct kvm_xen_exit {
#define KVM_EXIT_XEN_HCALL          1
                        __u32 type;
                        union {
                                struct {
                                        __u32 longmode;
                                        __u32 cpl;
                                        __u64 input;
                                        __u64 result;
                                        __u64 params[6];
                                } hcall;
                        } u;
                };
                /* KVM_EXIT_XEN */
                struct kvm_hyperv_exit xen;
```

Indicates that the VCPU exits into userspace to process some tasks related to Xen emulation.

Valid values for 'type' are:

- KVM_EXIT_XEN_HCALL -- synchronously notify user-space about Xen hypercall. Userspace is expected to place the hypercall result into the appropriate field before invoking KVM_RUN again.

```
/* KVM_EXIT_RISCV_SBI */
struct {
        unsigned long extension_id;
        unsigned long function_id;
        unsigned long args[6];
        unsigned long ret[2];
} riscv_sbi;
```

If exit reason is KVM_EXIT_RISCV_SBI then it indicates that the VCPU has done a SBI call which is not handled by KVM RISC-V kernel module. The details of the SBI call are available in 'riscv_sbi' member of kvm_run structure. The 'extension_id' field of 'riscv_sbi' represents SBI extension ID whereas the 'function_id' field represents function ID of given SBI extension. The 'args' array field of 'riscv_sbi' represents parameters for the SBI call and 'ret' array field represents return values. The userspace should update the return values of SBI call before resuming the VCPU. For more details on RISC-V SBI spec refer, https://github.com/riscv/riscv-sbi-doc.

```
                /* KVM_EXIT_MEMORY_FAULT */
                struct {
#define KVM_MEMORY_EXIT_FLAG_PRIVATE  (1ULL << 3)
                        __u64 flags;
                        __u64 gpa;
                        __u64 size;
                } memory_fault;
```

KVM_EXIT_MEMORY_FAULT indicates the vCPU has encountered a memory fault that could not be resolved by KVM. The 'gpa' and 'size' (in bytes) describe the guest physical address range [gpa, gpa + size) of the fault. The 'flags' field describes properties of the faulting access that are likely pertinent:

- KVM_MEMORY_EXIT_FLAG_PRIVATE - When set, indicates the memory fault occurred on a private memory access. When clear, indicates the fault occurred on a shared access.

Note! KVM_EXIT_MEMORY_FAULT is unique among all KVM exit reasons in that it accompanies a return code of '-1', not '0'! errno will always be set to EFAULT or EHWPOISON when KVM exits with KVM_EXIT_MEMORY_FAULT, userspace should assume kvm_run.exit_reason is stale/undefined for all other error numbers.

```
  /* KVM_EXIT_NOTIFY */
  struct {
#define KVM_NOTIFY_CONTEXT_INVALID    (1 << 0)
    __u32 flags;
  } notify;
```

Used on x86 systems. When the VM capability KVM_CAP_X86_NOTIFY_VMEXIT is enabled, a VM exit generated if no event window occurs in VM non-root mode for a specified amount of time. Once KVM_X86_NOTIFY_VMEXIT_USER is set when enabling the cap, it would exit to userspace with the exit reason KVM_EXIT_NOTIFY for further handling. The ”flags” field contains more detailed info.

The valid value for 'flags' is:

- KVM_NOTIFY_CONTEXT_INVALID -- the VM context is corrupted and not valid in VMCS. It would run into unknown result if resume the target VM.

```
        /* Fix the size of the union. */
        char padding[256];
};

/*
 * shared registers between kvm and userspace.
 * kvm_valid_regs specifies the register classes set by the host
 * kvm_dirty_regs specified the register classes dirtied by userspace
 * struct kvm_sync_regs is architecture specific, as well as the
 * bits for kvm_valid_regs and kvm_dirty_regs
 */
__u64 kvm_valid_regs;
__u64 kvm_dirty_regs;
union {
        struct kvm_sync_regs regs;
        char padding[SYNC_REGS_SIZE_BYTES];
} s;
```

If KVM_CAP_SYNC_REGS is defined, these fields allow userspace to access certain guest registers without having to call SET/GET_*REGS. Thus we can avoid some system call overhead if userspace has to handle the exit. Userspace can query the validity of the structure by checking kvm_valid_regs for specific bits. These bits are architecture specific and usually define the validity of a groups of registers. (e.g. one bit for general purpose registers)

Please note that the kernel is allowed to use the kvm_run structure as the primary storage for certain register types. Therefore, the kernel may use the values in kvm_run even if the corresponding bit in kvm_dirty_regs is not set.

1.1.6 6. Capabilities that can be enabled on vCPUs

There are certain capabilities that change the behavior of the virtual CPU or the virtual machine when enabled. To enable them, please see section 4.37. Below you can find a list of capabilities and what their effect on the vCPU or the virtual machine is when enabling them.

The following information is provided along with the description:

Architectures:
which instruction set architectures provide this ioctl. x86 includes both i386 and x86_64.

Target:
whether this is a per-vcpu or per-vm capability.

Parameters:
what parameters are accepted by the capability.

Returns:
the return value. General error numbers (EBADF, ENOMEM, EINVAL) are not detailed, but errors with specific meanings are.

6.1 KVM_CAP_PPC_OSI

Architectures
ppc

Target
vcpu

Parameters
none

Returns
0 on success; -1 on error

This capability enables interception of OSI hypercalls that otherwise would be treated as normal system calls to be injected into the guest. OSI hypercalls were invented by Mac-on-Linux to have a standardized communication mechanism between the guest and the host.

When this capability is enabled, KVM_EXIT_OSI can occur.

6.2 KVM_CAP_PPC_PAPR

Architectures
ppc

Target
vcpu

Parameters
none

Returns
0 on success; -1 on error

This capability enables interception of PAPR hypercalls. PAPR hypercalls are done using the hypercall instruction ”sc 1”.

It also sets the guest privilege level to ”supervisor” mode. Usually the guest runs in ”hypervisor” privilege mode with a few missing features.

In addition to the above, it changes the semantics of SDR1. In this mode, the HTAB address part of SDR1 contains an HVA instead of a GPA, as PAPR keeps the HTAB invisible to the guest.

When this capability is enabled, KVM_EXIT_PAPR_HCALL can occur.

6.3 KVM_CAP_SW_TLB

Architectures
ppc

Target
vcpu

Parameters
args[0] is the address of a struct kvm_config_tlb

Returns
0 on success; -1 on error

```
struct kvm_config_tlb {
      __u64 params;
      __u64 array;
      __u32 mmu_type;
      __u32 array_len;
};
```

Configures the virtual CPU's TLB array, establishing a shared memory area between userspace and KVM. The ”params” and ”array” fields are userspace addresses of mmu-type-specific data structures. The ”array_len” field is an safety mechanism, and should be set to the size in bytes of the memory that userspace has reserved for the array. It must be at least the size dictated by ”mmu_type” and ”params”.

While KVM_RUN is active, the shared region is under control of KVM. Its contents are undefined, and any modification by userspace results in boundedly undefined behavior.

On return from KVM_RUN, the shared region will reflect the current state of the guest's TLB. If userspace makes any changes, it must call KVM_DIRTY_TLB to tell KVM which entries have been changed, prior to calling KVM_RUN again on this vcpu.

For mmu types KVM_MMU_FSL_BOOKE_NOHV and KVM_MMU_FSL_BOOKE_HV:

- The ”params” field is of type ”struct kvm_book3e_206_tlb_params”.
- The ”array” field points to an array of type ”struct kvm_book3e_206_tlb_entry”.
- The array consists of all entries in the first TLB, followed by all entries in the second TLB.
- Within a TLB, entries are ordered first by increasing set number. Within a set, entries are ordered by way (increasing ESEL).
- The hash for determining set number in TLB0 is: (MAS2 >> 12) & (num_sets - 1) where ”num_sets” is the tlb_sizes[] value divided by the tlb_ways[] value.

- The tsize field of mas1 shall be set to 4K on TLB0, even though the hardware ignores this value for TLB0.

6.4 KVM_CAP_S390_CSS_SUPPORT

Architectures
s390

Target
vcpu

Parameters
none

Returns
0 on success; -1 on error

This capability enables support for handling of channel I/O instructions.

TEST PENDING INTERRUPTION and the interrupt portion of TEST SUBCHANNEL are handled in-kernel, while the other I/O instructions are passed to userspace.

When this capability is enabled, KVM_EXIT_S390_TSCH will occur on TEST SUBCHANNEL intercepts.

Note that even though this capability is enabled per-vcpu, the complete virtual machine is affected.

6.5 KVM_CAP_PPC_EPR

Architectures
ppc

Target
vcpu

Parameters
args[0] defines whether the proxy facility is active

Returns
0 on success; -1 on error

This capability enables or disables the delivery of interrupts through the external proxy facility.

When enabled (args[0] != 0), every time the guest gets an external interrupt delivered, it automatically exits into user space with a KVM_EXIT_EPR exit to receive the topmost interrupt vector.

When disabled (args[0] == 0), behavior is as if this facility is unsupported.

When this capability is enabled, KVM_EXIT_EPR can occur.

6.6 KVM_CAP_IRQ_MPIC

Architectures
ppc

Parameters
args[0] is the MPIC device fd; args[1] is the MPIC CPU number for this vcpu

This capability connects the vcpu to an in-kernel MPIC device.

6.7 KVM_CAP_IRQ_XICS

Architectures
ppc

Target
vcpu

Parameters
args[0] is the XICS device fd; args[1] is the XICS CPU number (server ID) for this vcpu

This capability connects the vcpu to an in-kernel XICS device.

6.8 KVM_CAP_S390_IRQCHIP

Architectures
s390

Target
vm

Parameters
none

This capability enables the in-kernel irqchip for s390. Please refer to "4.24 KVM_CREATE_IRQCHIP" for details.

6.9 KVM_CAP_MIPS_FPU

Architectures
mips

Target
vcpu

Parameters
args[0] is reserved for future use (should be 0).

This capability allows the use of the host Floating Point Unit by the guest. It allows the Config1.FP bit to be set to enable the FPU in the guest. Once this is done the `KVM_REG_MIPS_FPR_*` and `KVM_REG_MIPS_FCR_*` registers can be accessed (depending on the current guest FPU register mode), and the Status.FR, Config5.FRE bits are accessible via the KVM API and also from the guest, depending on them being supported by the FPU.

6.10 KVM_CAP_MIPS_MSA

Architectures
mips

Target
vcpu

Parameters
args[0] is reserved for future use (should be 0).

This capability allows the use of the MIPS SIMD Architecture (MSA) by the guest. It allows the Config3.MSAP bit to be set to enable the use of MSA by the guest. Once this is done the `KVM_REG_MIPS_VEC_*` and `KVM_REG_MIPS_MSA_*` registers can be accessed, and the Config5.MSAEn bit is accessible via the KVM API and also from the guest.

6.74 KVM_CAP_SYNC_REGS

Architectures
s390, x86

Target
s390: always enabled, x86: vcpu

Parameters
none

Returns
x86: KVM_CHECK_EXTENSION returns a bit-array indicating which register sets are supported (bitfields defined in arch/x86/include/uapi/asm/kvm.h).

As described above in the kvm_sync_regs struct info in section 5 (kvm_run): KVM_CAP_SYNC_REGS "allow[s] userspace to access certain guest registers without having to call SET/GET_*REGS". This reduces overhead by eliminating repeated ioctl calls for setting and/or getting register values. This is particularly important when userspace is making synchronous guest state modifications, e.g. when emulating and/or intercepting instructions in userspace.

For s390 specifics, please refer to the source code.

For x86:

- the register sets to be copied out to kvm_run are selectable by userspace (rather that all sets being copied out for every exit).
- vcpu_events are available in addition to regs and sregs.

For x86, the 'kvm_valid_regs' field of struct kvm_run is overloaded to function as an input bit-array field set by userspace to indicate the specific register sets to be copied out on the next exit.

To indicate when userspace has modified values that should be copied into the vCPU, the all architecture bitarray field, 'kvm_dirty_regs' must be set. This is done using the same bitflags as for the 'kvm_valid_regs' field. If the dirty bit is not set, then the register set values will not be copied into the vCPU even if they've been modified.

Unused bitfields in the bitarrays must be set to zero.

```
struct kvm_sync_regs {
      struct kvm_regs regs;
      struct kvm_sregs sregs;
      struct kvm_vcpu_events events;
};
```

6.75 KVM_CAP_PPC_IRQ_XIVE

Architectures
ppc

Target
vcpu

Parameters
args[0] is the XIVE device fd; args[1] is the XIVE CPU number (server ID) for this vcpu

This capability connects the vcpu to an in-kernel XIVE device.

1.1.7 7. Capabilities that can be enabled on VMs

There are certain capabilities that change the behavior of the virtual machine when enabled. To enable them, please see section 4.37. Below you can find a list of capabilities and what their effect on the VM is when enabling them.

The following information is provided along with the description:

Architectures:
which instruction set architectures provide this ioctl. x86 includes both i386 and x86_64.

Parameters:
what parameters are accepted by the capability.

Returns:
the return value. General error numbers (EBADF, ENOMEM, EINVAL) are not detailed, but errors with specific meanings are.

7.1 KVM_CAP_PPC_ENABLE_HCALL

Architectures
ppc

Parameters
args[0] is the sPAPR hcall number; args[1] is 0 to disable, 1 to enable in-kernel handling

This capability controls whether individual sPAPR hypercalls (hcalls) get handled by the kernel or not. Enabling or disabling in-kernel handling of an hcall is effective across the VM. On creation, an initial set of hcalls are enabled for in-kernel handling, which consists of those hcalls for which in-kernel handlers were implemented before this capability was implemented. If disabled, the kernel will not to attempt to handle the hcall, but will always exit to userspace

to handle it. Note that it may not make sense to enable some and disable others of a group of related hcalls, but KVM does not prevent userspace from doing that.

If the hcall number specified is not one that has an in-kernel implementation, the KVM_ENABLE_CAP ioctl will fail with an EINVAL error.

7.2 KVM_CAP_S390_USER_SIGP

Architectures
s390

Parameters
none

This capability controls which SIGP orders will be handled completely in user space. With this capability enabled, all fast orders will be handled completely in the kernel:

- SENSE
- SENSE RUNNING
- EXTERNAL CALL
- EMERGENCY SIGNAL
- CONDITIONAL EMERGENCY SIGNAL

All other orders will be handled completely in user space.

Only privileged operation exceptions will be checked for in the kernel (or even in the hardware prior to interception). If this capability is not enabled, the old way of handling SIGP orders is used (partially in kernel and user space).

7.3 KVM_CAP_S390_VECTOR_REGISTERS

Architectures
s390

Parameters
none

Returns
0 on success, negative value on error

Allows use of the vector registers introduced with z13 processor, and provides for the synchronization between host and user space. Will return -EINVAL if the machine does not support vectors.

7.4 KVM_CAP_S390_USER_STSI

Architectures
s390

Parameters
none

This capability allows post-handlers for the STSI instruction. After initial handling in the kernel, KVM exits to user space with KVM_EXIT_S390_STSI to allow user space to insert further data.

Before exiting to userspace, kvm handlers should fill in s390_stsi field of vcpu->run:

```
struct {
      __u64 addr;
      __u8 ar;
      __u8 reserved;
      __u8 fc;
      __u8 sel1;
      __u16 sel2;
} s390_stsi;

@addr - guest address of STSI SYSIB
@fc   - function code
@sel1 - selector 1
@sel2 - selector 2
@ar   - access register number
```

KVM handlers should exit to userspace with rc = -EREMOTE.

7.5 KVM_CAP_SPLIT_IRQCHIP

Architectures
x86

Parameters
args[0] - number of routes reserved for userspace IOAPICs

Returns
0 on success, -1 on error

Create a local apic for each processor in the kernel. This can be used instead of KVM_CREATE_IRQCHIP if the userspace VMM wishes to emulate the IOAPIC and PIC (and also the PIT, even though this has to be enabled separately).

This capability also enables in kernel routing of interrupt requests; when KVM_CAP_SPLIT_IRQCHIP only routes of KVM_IRQ_ROUTING_MSI type are used in the IRQ routing table. The first args[0] MSI routes are reserved for the IOAPIC pins. Whenever the LAPIC receives an EOI for these routes, a KVM_EXIT_IOAPIC_EOI vmexit will be reported to userspace.

Fails if VCPU has already been created, or if the irqchip is already in the kernel (i.e. KVM_CREATE_IRQCHIP has already been called).

7.6 KVM_CAP_S390_RI

Architectures
s390

Parameters
none

Allows use of runtime-instrumentation introduced with zEC12 processor. Will return -EINVAL if the machine does not support runtime-instrumentation. Will return -EBUSY if a VCPU has already been created.

7.7 KVM_CAP_X2APIC_API

Architectures
x86

Parameters
args[0] - features that should be enabled

Returns
0 on success, -EINVAL when args[0] contains invalid features

Valid feature flags in args[0] are:

```
#define KVM_X2APIC_API_USE_32BIT_IDS            (1ULL << 0)
#define KVM_X2APIC_API_DISABLE_BROADCAST_QUIRK  (1ULL << 1)
```

Enabling KVM_X2APIC_API_USE_32BIT_IDS changes the behavior of KVM_SET_GSI_ROUTING, KVM_SIGNAL_MSI, KVM_SET_LAPIC, and KVM_GET_LAPIC, allowing the use of 32-bit APIC IDs. See KVM_CAP_X2APIC_API in their respective sections.

KVM_X2APIC_API_DISABLE_BROADCAST_QUIRK must be enabled for x2APIC to work in logical mode or with more than 255 VCPUs. Otherwise, KVM treats 0xff as a broadcast even in x2APIC mode in order to support physical x2APIC without interrupt remapping. This is undesirable in logical mode, where 0xff represents CPUs 0-7 in cluster 0.

7.8 KVM_CAP_S390_USER_INSTR0

Architectures
s390

Parameters
none

With this capability enabled, all illegal instructions 0x0000 (2 bytes) will be intercepted and forwarded to user space. User space can use this mechanism e.g. to realize 2-byte software breakpoints. The kernel will not inject an operating exception for these instructions, user space has to take care of that.

This capability can be enabled dynamically even if VCPUs were already created and are running.

7.9 KVM_CAP_S390_GS

Architectures
s390

Parameters
none

Returns
0 on success; -EINVAL if the machine does not support guarded storage; -EBUSY if a VCPU has already been created.

Allows use of guarded storage for the KVM guest.

7.10 KVM_CAP_S390_AIS

Architectures
s390

Parameters
none

Allow use of adapter-interruption suppression. :Returns: 0 on success; -EBUSY if a VCPU has already been created.

7.11 KVM_CAP_PPC_SMT

Architectures
ppc

Parameters
vsmt_mode, flags

Enabling this capability on a VM provides userspace with a way to set the desired virtual SMT mode (i.e. the number of virtual CPUs per virtual core). The virtual SMT mode, vsmt_mode, must be a power of 2 between 1 and 8. On POWER8, vsmt_mode must also be no greater than the number of threads per subcore for the host. Currently flags must be 0. A successful call to enable this capability will result in vsmt_mode being returned when the KVM_CAP_PPC_SMT capability is subsequently queried for the VM. This capability is only supported by HV KVM, and can only be set before any VCPUs have been created. The KVM_CAP_PPC_SMT_POSSIBLE capability indicates which virtual SMT modes are available.

7.12 KVM_CAP_PPC_FWNMI

Architectures
ppc

Parameters
none

With this capability a machine check exception in the guest address space will cause KVM to exit the guest with NMI exit reason. This enables QEMU to build error log and branch to guest kernel registered machine check handling routine. Without this capability KVM will branch to guests' 0x200 interrupt vector.

7.13 KVM_CAP_X86_DISABLE_EXITS

Architectures
x86

Parameters
args[0] defines which exits are disabled

Returns
0 on success, -EINVAL when args[0] contains invalid exits

Valid bits in args[0] are:

```
#define KVM_X86_DISABLE_EXITS_MWAIT            (1 << 0)
#define KVM_X86_DISABLE_EXITS_HLT              (1 << 1)
#define KVM_X86_DISABLE_EXITS_PAUSE            (1 << 2)
#define KVM_X86_DISABLE_EXITS_CSTATE           (1 << 3)
```

Enabling this capability on a VM provides userspace with a way to no longer intercept some instructions for improved latency in some workloads, and is suggested when vCPUs are associated to dedicated physical CPUs. More bits can be added in the future; userspace can just pass the KVM_CHECK_EXTENSION result to KVM_ENABLE_CAP to disable all such vmexits.

Do not enable KVM_FEATURE_PV_UNHALT if you disable HLT exits.

7.14 KVM_CAP_S390_HPAGE_1M

Architectures
s390

Parameters
none

Returns
0 on success, -EINVAL if hpage module parameter was not set or cmma is enabled, or the VM has the KVM_VM_S390_UCONTROL flag set

With this capability the KVM support for memory backing with 1m pages through hugetlbfs can be enabled for a VM. After the capability is enabled, cmma can't be enabled anymore and pfmfi and the storage key interpretation are disabled. If cmma has already been enabled or the hpage module parameter is not set to 1, -EINVAL is returned.

While it is generally possible to create a huge page backed VM without this capability, the VM will not be able to run.

7.15 KVM_CAP_MSR_PLATFORM_INFO

Architectures
x86

Parameters
args[0] whether feature should be enabled or not

With this capability, a guest may read the MSR_PLATFORM_INFO MSR. Otherwise, a #GP would be raised when the guest tries to access. Currently, this capability does not enable write permissions of this MSR for the guest.

7.16 KVM_CAP_PPC_NESTED_HV

Architectures
ppc

Parameters
none

Returns
0 on success, -EINVAL when the implementation doesn't support nested-HV virtualization.

HV-KVM on POWER9 and later systems allows for ”nested-HV” virtualization, which provides a way for a guest VM to run guests that can run using the CPU's supervisor mode (privileged non-hypervisor state). Enabling this capability on a VM depends on the CPU having the necessary functionality and on the facility being enabled with a kvm-hv module parameter.

7.17 KVM_CAP_EXCEPTION_PAYLOAD

Architectures
x86

Parameters
args[0] whether feature should be enabled or not

With this capability enabled, CR2 will not be modified prior to the emulated VM-exit when L1 intercepts a #PF exception that occurs in L2. Similarly, for kvm-intel only, DR6 will not be modified prior to the emulated VM-exit when L1 intercepts a #DB exception that occurs in L2. As a result, when KVM_GET_VCPU_EVENTS reports a pending #PF (or #DB) exception for L2, exception.has_payload will be set and the faulting address (or the new DR6 bits*) will be reported in the exception_payload field. Similarly, when userspace injects a #PF (or #DB) into L2 using KVM_SET_VCPU_EVENTS, it is expected to set exception.has_payload and to put the faulting address - or the new DR6 bits[3] - in the exception_payload field.

This capability also enables exception.pending in struct kvm_vcpu_events, which allows userspace to distinguish between pending and injected exceptions.

7.18 KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2

Architectures
x86, arm64, mips

Parameters
args[0] whether feature should be enabled or not

Valid flags are:

[3] For the new DR6 bits, note that bit 16 is set iff the #DB exception will clear DR6.RTM.

```
#define KVM_DIRTY_LOG_MANUAL_PROTECT_ENABLE   (1 << 0)
#define KVM_DIRTY_LOG_INITIALLY_SET           (1 << 1)
```

With KVM_DIRTY_LOG_MANUAL_PROTECT_ENABLE is set, KVM_GET_DIRTY_LOG will not automatically clear and write-protect all pages that are returned as dirty. Rather, userspace will have to do this operation separately using KVM_CLEAR_DIRTY_LOG.

At the cost of a slightly more complicated operation, this provides better scalability and responsiveness for two reasons. First, KVM_CLEAR_DIRTY_LOG ioctl can operate on a 64-page granularity rather than requiring to sync a full memslot; this ensures that KVM does not take spinlocks for an extended period of time. Second, in some cases a large amount of time can pass between a call to KVM_GET_DIRTY_LOG and userspace actually using the data in the page. Pages can be modified during this time, which is inefficient for both the guest and userspace: the guest will incur a higher penalty due to write protection faults, while userspace can see false reports of dirty pages. Manual reprotection helps reducing this time, improving guest performance and reducing the number of dirty log false positives.

With KVM_DIRTY_LOG_INITIALLY_SET set, all the bits of the dirty bitmap will be initialized to 1 when created. This also improves performance because dirty logging can be enabled gradually in small chunks on the first call to KVM_CLEAR_DIRTY_LOG. KVM_DIRTY_LOG_INITIALLY_SET depends on KVM_DIRTY_LOG_MANUAL_PROTECT_ENABLE (it is also only available on x86 and arm64 for now).

KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2 was previously available under the name KVM_CAP_MANUAL_DIRTY_LOG_PROTECT, but the implementation had bugs that make it hard or impossible to use it correctly. The availability of KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2 signals that those bugs are fixed. Userspace should not try to use KVM_CAP_MANUAL_DIRTY_LOG_PROTECT.

7.19 KVM_CAP_PPC_SECURE_GUEST

Architectures
ppc

This capability indicates that KVM is running on a host that has ultravisor firmware and thus can support a secure guest. On such a system, a guest can ask the ultravisor to make it a secure guest, one whose memory is inaccessible to the host except for pages which are explicitly requested to be shared with the host. The ultravisor notifies KVM when a guest requests to become a secure guest, and KVM has the opportunity to veto the transition.

If present, this capability can be enabled for a VM, meaning that KVM will allow the transition to secure guest mode. Otherwise KVM will veto the transition.

7.20 KVM_CAP_HALT_POLL

Architectures
all

Target
VM

Parameters
args[0] is the maximum poll time in nanoseconds

Returns
0 on success; -1 on error

KVM_CAP_HALT_POLL overrides the kvm.halt_poll_ns module parameter to set the maximum halt-polling time for all vCPUs in the target VM. This capability can be invoked at any time and any number of times to dynamically change the maximum halt-polling time.

See *The KVM halt polling system* for more information on halt polling.

7.21 KVM_CAP_X86_USER_SPACE_MSR

Architectures
x86

Target
VM

Parameters
args[0] contains the mask of KVM_MSR_EXIT_REASON_* events to report

Returns
0 on success; -1 on error

This capability allows userspace to intercept RDMSR and WRMSR instructions if access to an MSR is denied. By default, KVM injects #GP on denied accesses.

When a guest requests to read or write an MSR, KVM may not implement all MSRs that are relevant to a respective system. It also does not differentiate by CPU type.

To allow more fine grained control over MSR handling, userspace may enable this capability. With it enabled, MSR accesses that match the mask specified in args[0] and would trigger a #GP inside the guest will instead trigger KVM_EXIT_X86_RDMSR and KVM_EXIT_X86_WRMSR exit notifications. Userspace can then implement model specific MSR handling and/or user notifications to inform a user that an MSR was not emulated/virtualized by KVM.

The valid mask flags are:

KVM_MSR_EXIT_REASON_UNKNOWN	intercept accesses to unknown (to KVM) MSRs
KVM_MSR_EXIT_REASON_INVAL	intercept accesses that are architecturally invalid according to the vCPU model and/or mode
KVM_MSR_EXIT_REASON_FILTER	intercept accesses that are denied by userspace via KVM_X86_SET_MSR_FILTER

7.22 KVM_CAP_X86_BUS_LOCK_EXIT

Architectures
x86

Target
VM

Parameters
args[0] defines the policy used when bus locks detected in guest

Returns
0 on success, -EINVAL when args[0] contains invalid bits

Valid bits in args[0] are:

```
#define KVM_BUS_LOCK_DETECTION_OFF      (1 << 0)
#define KVM_BUS_LOCK_DETECTION_EXIT     (1 << 1)
```

Enabling this capability on a VM provides userspace with a way to select a policy to handle the bus locks detected in guest. Userspace can obtain the supported modes from the result of KVM_CHECK_EXTENSION and define it through the KVM_ENABLE_CAP.

KVM_BUS_LOCK_DETECTION_OFF and KVM_BUS_LOCK_DETECTION_EXIT are supported currently and mutually exclusive with each other. More bits can be added in the future.

With KVM_BUS_LOCK_DETECTION_OFF set, bus locks in guest will not cause vm exits so that no additional actions are needed. This is the default mode.

With KVM_BUS_LOCK_DETECTION_EXIT set, vm exits happen when bus lock detected in VM. KVM just exits to userspace when handling them. Userspace can enforce its own throttling or other policy based mitigations.

This capability is aimed to address the thread that VM can exploit bus locks to degree the performance of the whole system. Once the userspace enable this capability and select the KVM_BUS_LOCK_DETECTION_EXIT mode, KVM will set the KVM_RUN_BUS_LOCK flag in vcpu-run->flags field and exit to userspace. Concerning the bus lock vm exit can be preempted by a higher priority VM exit, the exit notifications to userspace can be KVM_EXIT_BUS_LOCK or other reasons. KVM_RUN_BUS_LOCK flag is used to distinguish between them.

7.23 KVM_CAP_PPC_DAWR1

Architectures
ppc

Parameters
none

Returns
0 on success, -EINVAL when CPU doesn't support 2nd DAWR

This capability can be used to check / enable 2nd DAWR feature provided by POWER10 processor.

7.24 KVM_CAP_VM_COPY_ENC_CONTEXT_FROM

Architectures: x86 SEV enabled Type: vm Parameters: args[0] is the fd of the source vm Returns: 0 on success; ENOTTY on error

This capability enables userspace to copy encryption context from the vm indicated by the fd to the vm this is called on.

This is intended to support in-guest workloads scheduled by the host. This allows the in-guest workload to maintain its own NPTs and keeps the two vms from accidentally clobbering each other with interrupts and the like (separate APIC/MSRs/etc).

7.25 KVM_CAP_SGX_ATTRIBUTE

Architectures
x86

Target
VM

Parameters
args[0] is a file handle of a SGX attribute file in securityfs

Returns
0 on success, -EINVAL if the file handle is invalid or if a requested attribute is not supported by KVM.

KVM_CAP_SGX_ATTRIBUTE enables a userspace VMM to grant a VM access to one or more privileged enclave attributes. args[0] must hold a file handle to a valid SGX attribute file corresponding to an attribute that is supported/restricted by KVM (currently only PROVISIONKEY).

The SGX subsystem restricts access to a subset of enclave attributes to provide additional security for an uncompromised kernel, e.g. use of the PROVISIONKEY is restricted to deter malware from using the PROVISIONKEY to obtain a stable system fingerprint. To prevent userspace from circumventing such restrictions by running an enclave in a VM, KVM prevents access to privileged attributes by default.

See Documentation/arch/x86/sgx.rst for more details.

7.26 KVM_CAP_PPC_RPT_INVALIDATE

Capability
KVM_CAP_PPC_RPT_INVALIDATE

Architectures
ppc

Type
vm

This capability indicates that the kernel is capable of handling H_RPT_INVALIDATE hcall.

In order to enable the use of H_RPT_INVALIDATE in the guest, user space might have to advertise it for the guest. For example, IBM pSeries (sPAPR) guest starts using it if "hcall-rpt-invalidate" is present in the "ibm,hypertas-functions" device-tree property.

This capability is enabled for hypervisors on platforms like POWER9 that support radix MMU.

7.27 KVM_CAP_EXIT_ON_EMULATION_FAILURE

Architectures
x86

Parameters
args[0] whether the feature should be enabled or not

When this capability is enabled, an emulation failure will result in an exit to userspace with KVM_INTERNAL_ERROR (except when the emulator was invoked to handle a VMware backdoor instruction). Furthermore, KVM will now provide up to 15 instruction bytes for any exit to userspace resulting from an emulation failure. When these exits to userspace occur use the emulation_failure struct instead of the internal struct. They both have the same layout, but the emulation_failure struct matches the content better. It also explicitly defines the 'flags' field which is used to describe the fields in the struct that are valid (ie: if KVM_INTERNAL_ERROR_EMULATION_FLAG_INSTRUCTION_BYTES is set in the 'flags' field then both 'insn_size' and 'insn_bytes' have valid data in them.)

7.28 KVM_CAP_ARM_MTE

Architectures
arm64

Parameters
none

This capability indicates that KVM (and the hardware) supports exposing the Memory Tagging Extensions (MTE) to the guest. It must also be enabled by the VMM before creating any VCPUs to allow the guest access. Note that MTE is only available to a guest running in AArch64 mode and enabling this capability will cause attempts to create AArch32 VCPUs to fail.

When enabled the guest is able to access tags associated with any memory given to the guest. KVM will ensure that the tags are maintained during swap or hibernation of the host; however the VMM needs to manually save/restore the tags as appropriate if the VM is migrated.

When this capability is enabled all memory in memslots must be mapped as `MAP_ANONYMOUS` or with a RAM-based file mapping (`tmpfs`, `memfd`), attempts to create a memslot with an invalid mmap will result in an -EINVAL return.

When enabled the VMM may make use of the `KVM_ARM_MTE_COPY_TAGS` ioctl to perform a bulk copy of tags to/from the guest.

7.29 KVM_CAP_VM_MOVE_ENC_CONTEXT_FROM

Architectures: x86 SEV enabled Type: vm Parameters: args[0] is the fd of the source vm Returns: 0 on success

This capability enables userspace to migrate the encryption context from the VM indicated by the fd to the VM this is called on.

This is intended to support intra-host migration of VMs between userspace VMMs, upgrading the VMM process without interrupting the guest.

7.30 KVM_CAP_PPC_AIL_MODE_3

Capability
KVM_CAP_PPC_AIL_MODE_3

Architectures
ppc

Type
vm

This capability indicates that the kernel supports the mode 3 setting for the ”Address Translation Mode on Interrupt” aka ”Alternate Interrupt Location” resource that is controlled with the H_SET_MODE hypercall.

This capability allows a guest kernel to use a better-performance mode for handling interrupts and system calls.

7.31 KVM_CAP_DISABLE_QUIRKS2

Capability
KVM_CAP_DISABLE_QUIRKS2

Parameters
args[0] - set of KVM quirks to disable

Architectures
x86

Type
vm

This capability, if enabled, will cause KVM to disable some behavior quirks.

Calling KVM_CHECK_EXTENSION for this capability returns a bitmask of quirks that can be disabled in KVM.

The argument to KVM_ENABLE_CAP for this capability is a bitmask of quirks to disable, and must be a subset of the bitmask returned by KVM_CHECK_EXTENSION.

The valid bits in cap.args[0] are:

KVM_X86_QUIRK_LINT0_REENABLED	By default, the reset value for the LVT LINT0 register is 0x700 (APIC_MODE_EXTINT). When this quirk is disabled, the reset value is 0x10000 (APIC_LVT_MASKED).
KVM_X86_QUIRK_CD_NW_CLEARED	By default, KVM clears CR0.CD and CR0.NW. When this quirk is disabled, KVM does not change the value of CR0.CD and CR0.NW.
KVM_X86_QUIRK_LAPIC_MMIO_HOLE	By default, the MMIO LAPIC interface is available even when configured for x2APIC mode. When this quirk is disabled, KVM disables the MMIO LAPIC interface if the LAPIC is in x2APIC mode.
KVM_X86_QUIRK_OUT_7E_INC_RIP	By default, KVM pre-increments %rip before exiting to userspace for an OUT instruction to port 0x7e. When this quirk is disabled, KVM does not pre-increment %rip before exiting to userspace.
KVM_X86_QUIRK_MISC_ENABLE_NO_MWAIT	When this quirk is disabled, KVM sets CPUID.01H:ECX[bit 3] (MONITOR/MWAIT) if IA32_MISC_ENABLE[bit 18] (MWAIT) is set. Additionally, when this quirk is disabled, KVM clears CPUID.01H:ECX[bit 3] if IA32_MISC_ENABLE[bit 18] is cleared.
KVM_X86_QUIRK_FIX_HYPERCALL_INSN	By default, KVM rewrites guest VMMCALL/VMCALL instructions to match the vendor's hypercall instruction for the system. When this quirk is disabled, KVM will no longer rewrite invalid guest hypercall instructions. Executing the incorrect hypercall instruction will generate a #UD within the guest.
KVM_X86_QUIRK_MWAIT_NEVER_UD_FAULTS	By default, KVM emulates MONITOR/MWAIT (if they are intercepted) as NOPs regardless of whether or not MONITOR/MWAIT are supported according to guest CPUID. When this quirk is disabled and KVM_X86_DISABLE_EXITS_MWAIT is not set (MONITOR/MWAIT are intercepted), KVM will inject a #UD on MONITOR/MWAIT if they're unsupported per guest CPUID. Note, KVM will modify MONITOR/MWAIT support in guest CPUID on writes to MISC_ENABLE if KVM_X86_QUIRK_MISC_ENABLE_NO_MWAIT is disabled.

7.32 KVM_CAP_MAX_VCPU_ID

Architectures
x86

Target
VM

Parameters
args[0] - maximum APIC ID value set for current VM

Returns
0 on success, -EINVAL if args[0] is beyond KVM_MAX_VCPU_IDS supported in KVM or if it has been set.

This capability allows userspace to specify maximum possible APIC ID assigned for current VM session prior to the creation of vCPUs, saving memory for data structures indexed by the APIC ID. Userspace is able to calculate the limit to APIC ID values from designated CPU topology.

The value can be changed only until KVM_ENABLE_CAP is set to a nonzero value or until a vCPU is created. Upon creation of the first vCPU, if the value was set to zero or KVM_ENABLE_CAP was not invoked, KVM uses the return value of KVM_CHECK_EXTENSION(KVM_CAP_MAX_VCPU_ID) as the maximum APIC ID.

7.33 KVM_CAP_X86_NOTIFY_VMEXIT

Architectures
x86

Target
VM

Parameters
args[0] is the value of notify window as well as some flags

Returns
0 on success, -EINVAL if args[0] contains invalid flags or notify VM exit is unsupported.

Bits 63:32 of args[0] are used for notify window. Bits 31:0 of args[0] are for some flags. Valid bits are:

```
#define KVM_X86_NOTIFY_VMEXIT_ENABLED    (1 << 0)
#define KVM_X86_NOTIFY_VMEXIT_USER       (1 << 1)
```

This capability allows userspace to configure the notify VM exit on/off in per-VM scope during VM creation. Notify VM exit is disabled by default. When userspace sets KVM_X86_NOTIFY_VMEXIT_ENABLED bit in args[0], VMM will enable this feature with the notify window provided, which will generate a VM exit if no event window occurs in VM non-root mode for a specified of time (notify window).

If KVM_X86_NOTIFY_VMEXIT_USER is set in args[0], upon notify VM exits happen, KVM would exit to userspace for handling.

This capability is aimed to mitigate the threat that malicious VMs can cause CPU stuck (due to event windows don't open up) and make the CPU unavailable to host or other VMs.

7.34 KVM_CAP_MEMORY_FAULT_INFO

Architectures
x86

Returns
Informational only, -EINVAL on direct KVM_ENABLE_CAP.

The presence of this capability indicates that KVM_RUN will fill kvm_run.memory_fault if KVM cannot resolve a guest page fault VM-Exit, e.g. if there is a valid memslot but no backing VMA for the corresponding host virtual address.

The information in kvm_run.memory_fault is valid if and only if KVM_RUN returns an error with errno=EFAULT or errno=EHWPOISON *and* kvm_run.exit_reason is set to KVM_EXIT_MEMORY_FAULT.

Note: Userspaces which attempt to resolve memory faults so that they can retry KVM_RUN are encouraged to guard against repeatedly receiving the same error/annotated fault.

See KVM_EXIT_MEMORY_FAULT for more information.

1.1.8 8. Other capabilities.

This section lists capabilities that give information about other features of the KVM implementation.

8.1 KVM_CAP_PPC_HWRNG

Architectures
ppc

This capability, if KVM_CHECK_EXTENSION indicates that it is available, means that the kernel has an implementation of the H_RANDOM hypercall backed by a hardware random-number generator. If present, the kernel H_RANDOM handler can be enabled for guest use with the KVM_CAP_PPC_ENABLE_HCALL capability.

8.2 KVM_CAP_HYPERV_SYNIC

Architectures
x86

This capability, if KVM_CHECK_EXTENSION indicates that it is available, means that the kernel has an implementation of the Hyper-V Synthetic interrupt controller(SynIC). Hyper-V SynIC is used to support Windows Hyper-V based guest paravirt drivers(VMBus).

In order to use SynIC, it has to be activated by setting this capability via KVM_ENABLE_CAP ioctl on the vcpu fd. Note that this will disable the use of APIC hardware virtualization even if supported by the CPU, as it's incompatible with SynIC auto-EOI behavior.

8.3 KVM_CAP_PPC_RADIX_MMU

Architectures
ppc

This capability, if KVM_CHECK_EXTENSION indicates that it is available, means that the kernel can support guests using the radix MMU defined in Power ISA V3.00 (as implemented in the POWER9 processor).

8.4 KVM_CAP_PPC_HASH_MMU_V3

Architectures
ppc

This capability, if KVM_CHECK_EXTENSION indicates that it is available, means that the kernel can support guests using the hashed page table MMU defined in Power ISA V3.00 (as implemented in the POWER9 processor), including in-memory segment tables.

8.5 KVM_CAP_MIPS_VZ

Architectures
mips

This capability, if KVM_CHECK_EXTENSION on the main kvm handle indicates that it is available, means that full hardware assisted virtualization capabilities of the hardware are available for use through KVM. An appropriate KVM_VM_MIPS_* type must be passed to KVM_CREATE_VM to create a VM which utilises it.

If KVM_CHECK_EXTENSION on a kvm VM handle indicates that this capability is available, it means that the VM is using full hardware assisted virtualization capabilities of the hardware. This is useful to check after creating a VM with KVM_VM_MIPS_DEFAULT.

The value returned by KVM_CHECK_EXTENSION should be compared against known values (see below). All other values are reserved. This is to allow for the possibility of other hardware assisted virtualization implementations which may be incompatible with the MIPS VZ ASE.

0	The trap & emulate implementation is in use to run guest code in user mode. Guest virtual memory segments are rearranged to fit the guest in the user mode address space.
1	The MIPS VZ ASE is in use, providing full hardware assisted virtualization, including standard guest virtual memory segments.

8.6 KVM_CAP_MIPS_TE

Architectures
mips

This capability, if KVM_CHECK_EXTENSION on the main kvm handle indicates that it is available, means that the trap & emulate implementation is available to run guest code in user mode, even if KVM_CAP_MIPS_VZ indicates that hardware assisted virtualisation is also available. KVM_VM_MIPS_TE (0) must be passed to KVM_CREATE_VM to create a VM which utilises it.

If KVM_CHECK_EXTENSION on a kvm VM handle indicates that this capability is available, it means that the VM is using trap & emulate.

8.7 KVM_CAP_MIPS_64BIT

Architectures
mips

This capability indicates the supported architecture type of the guest, i.e. the supported register and address width.

The values returned when this capability is checked by KVM_CHECK_EXTENSION on a kvm VM handle correspond roughly to the CP0_Config.AT register field, and should be checked specifically against known values (see below). All other values are reserved.

0	MIPS32 or microMIPS32. Both registers and addresses are 32-bits wide. It will only be possible to run 32-bit guest code.
1	MIPS64 or microMIPS64 with access only to 32-bit compatibility segments. Registers are 64-bits wide, but addresses are 32-bits wide. 64-bit guest code may run but cannot access MIPS64 memory segments. It will also be possible to run 32-bit guest code.
2	MIPS64 or microMIPS64 with access to all address segments. Both registers and addresses are 64-bits wide. It will be possible to run 64-bit or 32-bit guest code.

8.9 KVM_CAP_ARM_USER_IRQ

Architectures
arm64

This capability, if KVM_CHECK_EXTENSION indicates that it is available, means that if userspace creates a VM without an in-kernel interrupt controller, it will be notified of changes to the output level of in-kernel emulated devices, which can generate virtual interrupts, presented to the VM. For such VMs, on every return to userspace, the kernel updates the vcpu's run->s.regs.device_irq_level field to represent the actual output level of the device.

Whenever kvm detects a change in the device output level, kvm guarantees at least one return to userspace before running the VM. This exit could either be a KVM_EXIT_INTR or any other exit event, like KVM_EXIT_MMIO. This way, userspace can always sample the device output level and re-compute the state of the userspace interrupt controller. Userspace should always check the state of run->s.regs.device_irq_level on every kvm exit. The value in run->s.regs.device_irq_level can represent both level and edge triggered interrupt signals, depending on the device. Edge triggered interrupt signals will exit to userspace with the bit in run->s.regs.device_irq_level set exactly once per edge signal.

The field run->s.regs.device_irq_level is available independent of run->kvm_valid_regs or run->kvm_dirty_regs bits.

If KVM_CAP_ARM_USER_IRQ is supported, the KVM_CHECK_EXTENSION ioctl returns a number larger than 0 indicating the version of this capability is implemented and thereby which bits in run->s.regs.device_irq_level can signal values.

Currently the following bits are defined for the device_irq_level bitmap:

```
KVM_CAP_ARM_USER_IRQ >= 1:

  KVM_ARM_DEV_EL1_VTIMER -  EL1 virtual timer
  KVM_ARM_DEV_EL1_PTIMER -  EL1 physical timer
  KVM_ARM_DEV_PMU        -  ARM PMU overflow interrupt signal
```

Future versions of kvm may implement additional events. These will get indicated by returning a higher number from KVM_CHECK_EXTENSION and will be listed above.

8.10 KVM_CAP_PPC_SMT_POSSIBLE

Architectures
ppc

Querying this capability returns a bitmap indicating the possible virtual SMT modes that can be set using KVM_CAP_PPC_SMT. If bit N (counting from the right) is set, then a virtual SMT mode of 2^N is available.

8.11 KVM_CAP_HYPERV_SYNIC2

Architectures
x86

This capability enables a newer version of Hyper-V Synthetic interrupt controller (SynIC). The only difference with KVM_CAP_HYPERV_SYNIC is that KVM doesn't clear SynIC message and event flags pages when they are enabled by writing to the respective MSRs.

8.12 KVM_CAP_HYPERV_VP_INDEX

Architectures
x86

This capability indicates that userspace can load HV_X64_MSR_VP_INDEX msr. Its value is used to denote the target vcpu for a SynIC interrupt. For compatibility, KVM initializes this msr to KVM's internal vcpu index. When this capability is absent, userspace can still query this msr's value.

8.13 KVM_CAP_S390_AIS_MIGRATION

Architectures
s390

Parameters
none

This capability indicates if the flic device will be able to get/set the AIS states for migration via the KVM_DEV_FLIC_AISM_ALL attribute and allows to discover this without having to create a flic device.

8.14 KVM_CAP_S390_PSW

Architectures
s390

This capability indicates that the PSW is exposed via the kvm_run structure.

8.15 KVM_CAP_S390_GMAP

Architectures
s390

This capability indicates that the user space memory used as guest mapping can be anywhere in the user memory address space, as long as the memory slots are aligned and sized to a segment (1MB) boundary.

8.16 KVM_CAP_S390_COW

Architectures
s390

This capability indicates that the user space memory used as guest mapping can use copy-on-write semantics as well as dirty pages tracking via read-only page tables.

8.17 KVM_CAP_S390_BPB

Architectures
s390

This capability indicates that kvm will implement the interfaces to handle reset, migration and nested KVM for branch prediction blocking. The stfle facility 82 should not be provided to the guest without this capability.

8.18 KVM_CAP_HYPERV_TLBFLUSH

Architectures
x86

This capability indicates that KVM supports paravirtualized Hyper-V TLB Flush hypercalls: HvFlushVirtualAddressSpace, HvFlushVirtualAddressSpaceEx, HvFlushVirtualAddressList, HvFlushVirtualAddressListEx.

8.19 KVM_CAP_ARM_INJECT_SERROR_ESR

Architectures
arm64

This capability indicates that userspace can specify (via the KVM_SET_VCPU_EVENTS ioctl) the syndrome value reported to the guest when it takes a virtual SError interrupt exception. If KVM advertises this capability, userspace can only specify the ISS field for the ESR syndrome. Other parts of the ESR, such as the EC are generated by the CPU when the exception is taken. If this virtual SError is taken to EL1 using AArch64, this value will be reported in the ISS field of ESR_ELx.

See KVM_CAP_VCPU_EVENTS for more details.

8.20 KVM_CAP_HYPERV_SEND_IPI

Architectures
x86

This capability indicates that KVM supports paravirtualized Hyper-V IPI send hypercalls: HvCallSendSyntheticClusterIpi, HvCallSendSyntheticClusterIpiEx.

8.21 KVM_CAP_HYPERV_DIRECT_TLBFLUSH

Architectures
x86

This capability indicates that KVM running on top of Hyper-V hypervisor enables Direct TLB flush for its guests meaning that TLB flush hypercalls are handled by Level 0 hypervisor (Hyper-V) bypassing KVM. Due to the different ABI for hypercall parameters between Hyper-V and KVM, enabling this capability effectively disables all hypercall handling by KVM (as some KVM hypercall may be mistakenly treated as TLB flush hypercalls by Hyper-V) so userspace should disable KVM identification in CPUID and only exposes Hyper-V identification. In this case, guest thinks it's running on Hyper-V and only use Hyper-V hypercalls.

8.22 KVM_CAP_S390_VCPU_RESETS

Architectures
s390

This capability indicates that the KVM_S390_NORMAL_RESET and KVM_S390_CLEAR_RESET ioctls are available.

8.23 KVM_CAP_S390_PROTECTED

Architectures
s390

This capability indicates that the Ultravisor has been initialized and KVM can therefore start protected VMs. This capability governs the KVM_S390_PV_COMMAND ioctl and the KVM_MP_STATE_LOAD MP_STATE. KVM_SET_MP_STATE can fail for protected guests when the state change is invalid.

8.24 KVM_CAP_STEAL_TIME

Architectures
arm64, x86

This capability indicates that KVM supports steal time accounting. When steal time accounting is supported it may be enabled with architecture-specific interfaces. This capability and the architecture- specific interfaces must be consistent, i.e. if one says the feature is supported, than the other should as well and vice versa. For arm64 see *Generic vcpu interface* "KVM_ARM_VCPU_PVTIME_CTRL". For x86 see *KVM-specific MSRs* "MSR_KVM_STEAL_TIME".

8.25 KVM_CAP_S390_DIAG318

Architectures
s390

This capability enables a guest to set information about its control program (i.e. guest kernel type and version). The information is helpful during system/firmware service events, providing additional data about the guest environments running on the machine.

The information is associated with the DIAGNOSE 0x318 instruction, which sets an 8-byte value consisting of a one-byte Control Program Name Code (CPNC) and a 7-byte Control Program Version Code (CPVC). The CPNC determines what environment the control program is running in (e.g. Linux, z/VM...), and the CPVC is used for information specific to OS (e.g. Linux version, Linux distribution...)

If this capability is available, then the CPNC and CPVC can be synchronized between KVM and userspace via the sync regs mechanism (KVM_SYNC_DIAG318).

8.26 KVM_CAP_X86_USER_SPACE_MSR

Architectures
x86

This capability indicates that KVM supports deflection of MSR reads and writes to user space. It can be enabled on a VM level. If enabled, MSR accesses that would usually trigger a #GP by KVM into the guest will instead get bounced to user space through the KVM_EXIT_X86_RDMSR and KVM_EXIT_X86_WRMSR exit notifications.

8.27 KVM_CAP_X86_MSR_FILTER

Architectures
x86

This capability indicates that KVM supports that accesses to user defined MSRs may be rejected. With this capability exposed, KVM exports new VM ioctl KVM_X86_SET_MSR_FILTER which user space can call to specify bitmaps of MSR ranges that KVM should deny access to.

In combination with KVM_CAP_X86_USER_SPACE_MSR, this allows user space to trap and emulate MSRs that are outside of the scope of KVM as well as limit the attack surface on KVM's MSR emulation code.

8.28 KVM_CAP_ENFORCE_PV_FEATURE_CPUID

Architectures: x86

When enabled, KVM will disable paravirtual features provided to the guest according to the bits in the KVM_CPUID_FEATURES CPUID leaf (0x40000001). Otherwise, a guest may use the paravirtual features regardless of what has actually been exposed through the CPUID leaf.

8.29 KVM_CAP_DIRTY_LOG_RING/KVM_CAP_DIRTY_LOG_RING_ACQ_REL

Architectures
x86, arm64

Parameters
args[0] - size of the dirty log ring

KVM is capable of tracking dirty memory using ring buffers that are mmapped into userspace; there is one dirty ring per vcpu.

The dirty ring is available to userspace as an array of `struct kvm_dirty_gfn`. Each dirty entry is defined as:

```
struct kvm_dirty_gfn {
        __u32 flags;
        __u32 slot; /* as_id | slot_id */
        __u64 offset;
};
```

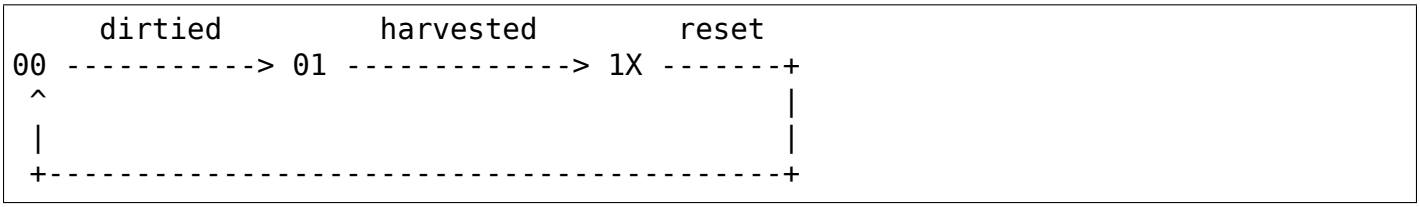
The following values are defined for the flags field to define the current state of the entry:

```
#define KVM_DIRTY_GFN_F_DIRTY           BIT(0)
#define KVM_DIRTY_GFN_F_RESET           BIT(1)
#define KVM_DIRTY_GFN_F_MASK            0x3
```

Userspace should call KVM_ENABLE_CAP ioctl right after KVM_CREATE_VM ioctl to enable this capability for the new guest and set the size of the rings. Enabling the capability is only allowed before creating any vCPU, and the size of the ring must be a power of two. The larger the ring buffer, the less likely the ring is full and the VM is forced to exit to userspace. The optimal size depends on the workload, but it is recommended that it be at least 64 KiB (4096 entries).

Just like for dirty page bitmaps, the buffer tracks writes to all user memory regions for which the KVM_MEM_LOG_DIRTY_PAGES flag was set in KVM_SET_USER_MEMORY_REGION. Once a memory region is registered with the flag set, userspace can start harvesting dirty pages from the ring buffer.

An entry in the ring buffer can be unused (flag bits `00`), dirty (flag bits `01`) or harvested (flag bits `1X`). The state machine for the entry is as follows:



To harvest the dirty pages, userspace accesses the mmapped ring buffer to read the dirty GFNs. If the flags has the DIRTY bit set (at this stage the RESET bit must be cleared), then it means this GFN is a dirty GFN. The userspace should harvest this GFN and mark the flags from state `01b` to `1Xb` (bit 0 will be ignored by KVM, but bit 1 must be set to show that this GFN is harvested and waiting for a reset), and move on to the next GFN. The userspace should continue to do this until the flags of a GFN have the DIRTY bit cleared, meaning that it has harvested all the dirty GFNs that were available.

Note that on weakly ordered architectures, userspace accesses to the ring buffer (and more specifically the 'flags' field) must be ordered, using load-acquire/store-release accessors when available, or any other memory barrier that will ensure this ordering.

It's not necessary for userspace to harvest the all dirty GFNs at once. However it must collect the dirty GFNs in sequence, i.e., the userspace program cannot skip one dirty GFN to collect the one next to it.

After processing one or more entries in the ring buffer, userspace calls the VM ioctl KVM_RESET_DIRTY_RINGS to notify the kernel about it, so that the kernel will reprotect those collected GFNs. Therefore, the ioctl must be called *before* reading the content of the dirty pages.

The dirty ring can get full. When it happens, the KVM_RUN of the vcpu will return with exit reason KVM_EXIT_DIRTY_LOG_FULL.

The dirty ring interface has a major difference comparing to the KVM_GET_DIRTY_LOG interface in that, when reading the dirty ring from userspace, it's still possible that the kernel has not yet flushed the processor's dirty page buffers into the kernel buffer (with dirty bitmaps, the flushing is done by the KVM_GET_DIRTY_LOG ioctl). To achieve that, one needs to kick the vcpu out of KVM_RUN using a signal. The resulting vmexit ensures that all dirty GFNs are flushed to the dirty rings.

NOTE: KVM_CAP_DIRTY_LOG_RING_ACQ_REL is the only capability that should be exposed by weakly ordered architecture, in order to indicate the additional memory ordering requirements imposed on userspace when reading the state of an entry and mutating it from DIRTY to HARVESTED. Architecture with TSO-like ordering (such as x86) are allowed to expose both KVM_CAP_DIRTY_LOG_RING and KVM_CAP_DIRTY_LOG_RING_ACQ_REL to userspace.

After enabling the dirty rings, the userspace needs to detect the capability of KVM_CAP_DIRTY_LOG_RING_WITH_BITMAP to see whether the ring structures can be backed by per-slot bitmaps. With this capability advertised, it means the architecture can dirty guest pages without vcpu/ring context, so that some of the dirty information will still

be maintained in the bitmap structure. KVM_CAP_DIRTY_LOG_RING_WITH_BITMAP can't be enabled if the capability of KVM_CAP_DIRTY_LOG_RING_ACQ_REL hasn't been enabled, or any memslot has been existing.

Note that the bitmap here is only a backup of the ring structure. The use of the ring and bitmap combination is only beneficial if there is only a very small amount of memory that is dirtied out of vcpu/ring context. Otherwise, the stand-alone per-slot bitmap mechanism needs to be considered.

To collect dirty bits in the backup bitmap, userspace can use the same KVM_GET_DIRTY_LOG ioctl. KVM_CLEAR_DIRTY_LOG isn't needed as long as all the generation of the dirty bits is done in a single pass. Collecting the dirty bitmap should be the very last thing that the VMM does before considering the state as complete. VMM needs to ensure that the dirty state is final and avoid missing dirty pages from another ioctl ordered after the bitmap collection.

NOTE: Multiple examples of using the backup bitmap: (1) save vgic/its tables through command KVM_DEV_ARM_{VGIC_GRP_CTRL, ITS_SAVE_TABLES} on KVM device "kvm-arm-vgic-its". (2) restore vgic/its tables through command KVM_DEV_ARM_{VGIC_GRP_CTRL, ITS_RESTORE_TABLES} on KVM device "kvm-arm-vgic-its". VGICv3 LPI pending status is restored. (3) save vgic3 pending table through KVM_DEV_ARM_VGIC_{GRP_CTRL, SAVE_PENDING_TABLES} command on KVM device "kvm-arm-vgic-v3".

8.30 KVM_CAP_XEN_HVM

Architectures
x86

This capability indicates the features that Xen supports for hosting Xen PVHVM guests. Valid flags are:

```
#define KVM_XEN_HVM_CONFIG_HYPERCALL_MSR               (1 << 0)
#define KVM_XEN_HVM_CONFIG_INTERCEPT_HCALL             (1 << 1)
#define KVM_XEN_HVM_CONFIG_SHARED_INFO                 (1 << 2)
#define KVM_XEN_HVM_CONFIG_RUNSTATE                    (1 << 3)
#define KVM_XEN_HVM_CONFIG_EVTCHN_2LEVEL               (1 << 4)
#define KVM_XEN_HVM_CONFIG_EVTCHN_SEND                 (1 << 5)
#define KVM_XEN_HVM_CONFIG_RUNSTATE_UPDATE_FLAG        (1 << 6)
#define KVM_XEN_HVM_CONFIG_PVCLOCK_TSC_UNSTABLE        (1 << 7)
```

The KVM_XEN_HVM_CONFIG_HYPERCALL_MSR flag indicates that the KVM_XEN_HVM_CONFIG ioctl is available, for the guest to set its hypercall page.

If KVM_XEN_HVM_CONFIG_INTERCEPT_HCALL is also set, the same flag may also be provided in the flags to KVM_XEN_HVM_CONFIG, without providing hypercall page contents, to request that KVM generate hypercall page content automatically and also enable interception of guest hypercalls with KVM_EXIT_XEN.

The KVM_XEN_HVM_CONFIG_SHARED_INFO flag indicates the availability of the KVM_XEN_HVM_SET_ATTR, KVM_XEN_HVM_GET_ATTR, KVM_XEN_VCPU_SET_ATTR and KVM_XEN_VCPU_GET_ATTR ioctls, as well as the delivery of exception vectors for event channel upcalls when the evtchn_upcall_pending field of a vcpu's vcpu_info is set.

The KVM_XEN_HVM_CONFIG_RUNSTATE flag indicates that the runstate-related features KVM_XEN_VCPU_ATTR_TYPE_RUNSTATE_ADDR/_CURRENT/_DATA/_ADJUST are supported

by the KVM_XEN_VCPU_SET_ATTR/KVM_XEN_VCPU_GET_ATTR ioctls.

The KVM_XEN_HVM_CONFIG_EVTCHN_2LEVEL flag indicates that IRQ routing entries of the type KVM_IRQ_ROUTING_XEN_EVTCHN are supported, with the priority field set to indicate 2 level event channel delivery.

The KVM_XEN_HVM_CONFIG_EVTCHN_SEND flag indicates that KVM supports injecting event channel events directly into the guest with the KVM_XEN_HVM_EVTCHN_SEND ioctl. It also indicates support for the KVM_XEN_ATTR_TYPE_EVTCHN/XEN_VERSION HVM attributes and the KVM_XEN_VCPU_ATTR_TYPE_VCPU_ID/TIMER/UPCALL_VECTOR vCPU attributes. related to event channel delivery, timers, and the XENVER_version interception.

The KVM_XEN_HVM_CONFIG_RUNSTATE_UPDATE_FLAG flag indicates that KVM supports the KVM_XEN_ATTR_TYPE_RUNSTATE_UPDATE_FLAG attribute in the KVM_XEN_SET_ATTR and KVM_XEN_GET_ATTR ioctls. This controls whether KVM will set the XEN_RUNSTATE_UPDATE flag in guest memory mapped vcpu_runstate_info during updates of the runstate information. Note that versions of KVM which support the RUNSTATE feature above, but not the RUNSTATE_UPDATE_FLAG feature, will always set the XEN_RUNSTATE_UPDATE flag when updating the guest structure, which is perhaps counterintuitive. When this flag is advertised, KVM will behave more correctly, not using the XEN_RUNSTATE_UPDATE flag until/unless specifically enabled (by the guest making the hypercall, causing the VMM to enable the KVM_XEN_ATTR_TYPE_RUNSTATE_UPDATE_FLAG attribute).

The KVM_XEN_HVM_CONFIG_PVCLOCK_TSC_UNSTABLE flag indicates that KVM supports clearing the PVCLOCK_TSC_STABLE_BIT flag in Xen pvclock sources. This will be done when the KVM_CAP_XEN_HVM ioctl sets the KVM_XEN_HVM_CONFIG_PVCLOCK_TSC_UNSTABLE flag.

8.31 KVM_CAP_PPC_MULTITCE

Capability
KVM_CAP_PPC_MULTITCE

Architectures
ppc

Type
vm

This capability means the kernel is capable of handling hypercalls H_PUT_TCE_INDIRECT and H_STUFF_TCE without passing those into the user space. This significantly accelerates DMA operations for PPC KVM guests. User space should expect that its handlers for these hypercalls are not going to be called if user space previously registered LIOBN in KVM (via KVM_CREATE_SPAPR_TCE or similar calls).

In order to enable H_PUT_TCE_INDIRECT and H_STUFF_TCE use in the guest, user space might have to advertise it for the guest. For example, IBM pSeries (sPAPR) guest starts using them if ”hcall-multi-tce” is present in the ”ibm,hypertas-functions” device-tree property.

The hypercalls mentioned above may or may not be processed successfully in the kernel based fast path. If they can not be handled by the kernel, they will get passed on to user space. So user space still has to have an implementation for these despite the in kernel acceleration.

This capability is always enabled.

8.32 KVM_CAP_PTP_KVM

Architectures
arm64

This capability indicates that the KVM virtual PTP service is supported in the host. A VMM can check whether the service is available to the guest on migration.

8.33 KVM_CAP_HYPERV_ENFORCE_CPUID

Architectures: x86

When enabled, KVM will disable emulated Hyper-V features provided to the guest according to the bits Hyper-V CPUID feature leaves. Otherwise, all currently implemented Hyper-V features are provided unconditionally when Hyper-V identification is set in the HYPERV_CPUID_INTERFACE (0x40000001) leaf.

8.34 KVM_CAP_EXIT_HYPERCALL

Capability
KVM_CAP_EXIT_HYPERCALL

Architectures
x86

Type
vm

This capability, if enabled, will cause KVM to exit to userspace with KVM_EXIT_HYPERCALL exit reason to process some hypercalls.

Calling KVM_CHECK_EXTENSION for this capability will return a bitmask of hypercalls that can be configured to exit to userspace. Right now, the only such hypercall is KVM_HC_MAP_GPA_RANGE.

The argument to KVM_ENABLE_CAP is also a bitmask, and must be a subset of the result of KVM_CHECK_EXTENSION. KVM will forward to userspace the hypercalls whose corresponding bit is in the argument, and return ENOSYS for the others.

8.35 KVM_CAP_PMU_CAPABILITY

Capability
KVM_CAP_PMU_CAPABILITY

Architectures
x86

Type
vm

Parameters
arg[0] is bitmask of PMU virtualization capabilities.

Returns
0 on success, -EINVAL when arg[0] contains invalid bits

This capability alters PMU virtualization in KVM.

Calling KVM_CHECK_EXTENSION for this capability returns a bitmask of PMU virtualization capabilities that can be adjusted on a VM.

The argument to KVM_ENABLE_CAP is also a bitmask and selects specific PMU virtualization capabilities to be applied to the VM. This can only be invoked on a VM prior to the creation of VCPUs.

At this time, KVM_PMU_CAP_DISABLE is the only capability. Setting this capability will disable PMU virtualization for that VM. Usermode should adjust CPUID leaf 0xA to reflect that the PMU is disabled.

8.36 KVM_CAP_ARM_SYSTEM_SUSPEND

Capability
KVM_CAP_ARM_SYSTEM_SUSPEND

Architectures
arm64

Type
vm

When enabled, KVM will exit to userspace with KVM_EXIT_SYSTEM_EVENT of type KVM_SYSTEM_EVENT_SUSPEND to process the guest suspend request.

8.37 KVM_CAP_S390_PROTECTED_DUMP

Capability
KVM_CAP_S390_PROTECTED_DUMP

Architectures
s390

Type
vm

This capability indicates that KVM and the Ultravisor support dumping PV guests. The *KVM_PV_DUMP* command is available for the *KVM_S390_PV_COMMAND* ioctl and the *KVM_PV_INFO* command provides dump related UV data. Also the vcpu ioctl *KVM_S390_PV_CPU_COMMAND* is available and supports the *KVM_PV_DUMP_CPU* subcommand.

8.38 KVM_CAP_VM_DISABLE_NX_HUGE_PAGES

Capability
KVM_CAP_VM_DISABLE_NX_HUGE_PAGES

Architectures
x86

Type
vm

Parameters
arg[0] must be 0.

Returns
0 on success, -EPERM if the userspace process does not have CAP_SYS_BOOT, -EINVAL if args[0] is not 0 or any vCPUs have been created.

This capability disables the NX huge pages mitigation for iTLB MULTIHIT.

The capability has no effect if the nx_huge_pages module parameter is not set.

This capability may only be set before any vCPUs are created.

8.39 KVM_CAP_S390_CPU_TOPOLOGY

Capability
KVM_CAP_S390_CPU_TOPOLOGY

Architectures
s390

Type
vm

This capability indicates that KVM will provide the S390 CPU Topology facility which consist of the interpretation of the PTF instruction for the function code 2 along with interception and forwarding of both the PTF instruction with function codes 0 or 1 and the STSI(15,1,x) instruction to the userland hypervisor.

The stfle facility 11, CPU Topology facility, should not be indicated to the guest without this capability.

When this capability is present, KVM provides a new attribute group on vm fd, KVM_S390_VM_CPU_TOPOLOGY. This new attribute allows to get, set or clear the Modified Change Topology Report (MTCR) bit of the SCA through the kvm_device_attr structure.

When getting the Modified Change Topology Report value, the attr->addr must point to a byte where the value will be stored or retrieved from.

8.40 KVM_CAP_ARM_EAGER_SPLIT_CHUNK_SIZE

Capability
KVM_CAP_ARM_EAGER_SPLIT_CHUNK_SIZE

Architectures
arm64

Type
vm

Parameters
arg[0] is the new split chunk size.

Returns
0 on success, -EINVAL if any memslot was already created.

This capability sets the chunk size used in Eager Page Splitting.

Eager Page Splitting improves the performance of dirty-logging (used in live migrations) when guest memory is backed by huge-pages. It avoids splitting huge-pages (into PAGE_SIZE pages) on fault, by doing it eagerly when enabling dirty logging (with the KVM_MEM_LOG_DIRTY_PAGES flag for a memory region), or when using KVM_CLEAR_DIRTY_LOG.

The chunk size specifies how many pages to break at a time, using a single allocation for each chunk. Bigger the chunk size, more pages need to be allocated ahead of time.

The chunk size needs to be a valid block size. The list of acceptable block sizes is exposed in KVM_CAP_ARM_SUPPORTED_BLOCK_SIZES as a 64-bit bitmap (each bit describing a block size). The default value is 0, to disable the eager page splitting.

8.41 KVM_CAP_VM_TYPES

Capability
KVM_CAP_MEMORY_ATTRIBUTES

Architectures
x86

Type
system ioctl

This capability returns a bitmap of support VM types. The 1-setting of bit @n means the VM type with value @n is supported. Possible values of @n are:

```
#define KVM_X86_DEFAULT_VM    0
#define KVM_X86_SW_PROTECTED_VM       1
```

Note, KVM_X86_SW_PROTECTED_VM is currently only for development and testing. Do not use KVM_X86_SW_PROTECTED_VM for "real" VMs, and especially not in production. The behavior and effective ABI for software-protected VMs is unstable.

1.1.9 9. Known KVM API problems

In some cases, KVM's API has some inconsistencies or common pitfalls that userspace need to be aware of. This section details some of these issues.

Most of them are architecture specific, so the section is split by architecture.

9.1. x86

KVM_GET_SUPPORTED_CPUID issues

In general, KVM_GET_SUPPORTED_CPUID is designed so that it is possible to take its result and pass it directly to KVM_SET_CPUID2. This section documents some cases in which that requires some care.

Local APIC features

CPU[EAX=1]:ECX[21] (X2APIC) is reported by KVM_GET_SUPPORTED_CPUID, but it can only be enabled if KVM_CREATE_IRQCHIP or KVM_ENABLE_CAP(KVM_CAP_IRQCHIP_SPLIT) are used to enable in-kernel emulation of the local APIC.

The same is true for the KVM_FEATURE_PV_UNHALT paravirtualized feature.

CPU[EAX=1]:ECX[24] (TSC_DEADLINE) is not reported by KVM_GET_SUPPORTED_CPUID. It can be enabled if KVM_CAP_TSC_DEADLINE_TIMER is present and the kernel has enabled in-kernel emulation of the local APIC.

CPU topology

Several CPUID values include topology information for the host CPU: 0x0b and 0x1f for Intel systems, 0x8000001e for AMD systems. Different versions of KVM return different values for this information and userspace should not rely on it. Currently they return all zeroes.

If userspace wishes to set up a guest topology, it should be careful that the values of these three leaves differ for each CPU. In particular, the APIC ID is found in EDX for all subleaves of 0x0b and 0x1f, and in EAX for 0x8000001e; the latter also encodes the core id and node id in bits 7:0 of EBX and ECX respectively.

Obsolete ioctls and capabilities

KVM_CAP_DISABLE_QUIRKS does not let userspace know which quirks are actually available. Use KVM_CHECK_EXTENSION(KVM_CAP_DISABLE_QUIRKS2) instead if available.

Ordering of KVM_GET_*/KVM_SET_* ioctls

TBD

1.2 Devices

1.2.1 ARM Virtual Interrupt Translation Service (ITS)

Device types supported:
KVM_DEV_TYPE_ARM_VGIC_ITS ARM Interrupt Translation Service Controller

The ITS allows MSI(-X) interrupts to be injected into guests. This extension is optional. Creating a virtual ITS controller also requires a host GICv3 (see arm-vgic-v3.txt), but does not depend on having physical ITS controllers.

There can be multiple ITS controllers per guest, each of them has to have a separate, non-overlapping MMIO region.

Groups

KVM_DEV_ARM_VGIC_GRP_ADDR

Attributes:

KVM_VGIC_ITS_ADDR_TYPE (rw, 64-bit)
Base address in the guest physical address space of the GICv3 ITS control register frame. This address needs to be 64K aligned and the region covers 128K.

Errors:

-E2BIG	Address outside of addressable IPA range
-EINVAL	Incorrectly aligned address
-EEXIST	Address already configured
-EFAULT	Invalid user pointer for attr->addr.
-ENODEV	Incorrect attribute or the ITS is not supported.

KVM_DEV_ARM_VGIC_GRP_CTRL

Attributes:

KVM_DEV_ARM_VGIC_CTRL_INIT
request the initialization of the ITS, no additional parameter in kvm_device_attr.addr.

KVM_DEV_ARM_ITS_CTRL_RESET
reset the ITS, no additional parameter in kvm_device_attr.addr. See ”ITS Reset State” section.

KVM_DEV_ARM_ITS_SAVE_TABLES
save the ITS table data into guest RAM, at the location provisioned by the guest in corresponding registers/table entries. Should userspace require a form of dirty tracking to identify which pages are modified by the saving process, it should use a bitmap even if using another mechanism to track the memory dirtied by the vCPUs.

The layout of the tables in guest memory defines an ABI. The entries are laid out in little endian format as described in the last paragraph.

KVM_DEV_ARM_ITS_RESTORE_TABLES
restore the ITS tables from guest RAM to ITS internal structures.

The GICV3 must be restored before the ITS and all ITS registers but the GITS_CTLR must be restored before restoring the ITS tables.

The GITS_IIDR read-only register must also be restored before calling KVM_DEV_ARM_ITS_RESTORE_TABLES as the IIDR revision field encodes the ABI revision.

The expected ordering when restoring the GICv3/ITS is described in section ”ITS Restore Sequence”.

Errors:

-ENXIO	ITS not properly configured as required prior to setting this attribute
-ENOMEM	Memory shortage when allocating ITS internal data
-EINVAL	Inconsistent restored data
-EFAULT	Invalid guest ram access
-EBUSY	One or more VCPUS are running
-EACCES	The virtual ITS is backed by a physical GICv4 ITS, and the state is not available without GICv4.1

KVM_DEV_ARM_VGIC_GRP_ITS_REGS

Attributes:

The attr field of kvm_device_attr encodes the offset of the ITS register, relative to the ITS control frame base address (ITS_base).

kvm_device_attr.addr points to a __u64 value whatever the width of the addressed register (32/64 bits). 64 bit registers can only be accessed with full length.

Writes to read-only registers are ignored by the kernel except for:

- GITS_CREADR. It must be restored otherwise commands in the queue will be re-executed after restoring CWRITER. GITS_CREADR must be restored before restoring the GITS_CTLR which is likely to enable the ITS. Also it must be restored after GITS_CBASER since a write to GITS_CBASER resets GITS_CREADR.
- GITS_IIDR. The Revision field encodes the table layout ABI revision. In the future we might implement direct injection of virtual LPIs. This will require an upgrade of the table layout and an evolution of the ABI. GITS_IIDR must be restored before calling KVM_DEV_ARM_ITS_RESTORE_TABLES.

For other registers, getting or setting a register has the same effect as reading/writing the register on real hardware.

Errors:

-ENXIO	Offset does not correspond to any supported register
-EFAULT	Invalid user pointer for attr->addr
-EINVAL	Offset is not 64-bit aligned
-EBUSY	one or more VCPUS are running

ITS Restore Sequence:

The following ordering must be followed when restoring the GIC and the ITS:

a) restore all guest memory and create vcpus

b) restore all redistributors

c) provide the ITS base address (KVM_DEV_ARM_VGIC_GRP_ADDR)

d) restore the ITS in the following order:

1. Restore GITS_CBASER
2. Restore all other GITS_ registers, except GITS_CTLR!
3. Load the ITS table data (KVM_DEV_ARM_ITS_RESTORE_TABLES)
4. Restore GITS_CTLR

Then vcpus can be started.

ITS Table ABI REV0:

Revision 0 of the ABI only supports the features of a virtual GICv3, and does not support a virtual GICv4 with support for direct injection of virtual interrupts for nested hypervisors.

The device table and ITT are indexed by the DeviceID and EventID, respectively. The collection table is not indexed by CollectionID, and the entries in the collection are listed in no particular order. All entries are 8 bytes.

Device Table Entry (DTE):

```
bits:     | 63| 62 ... 49 | 48 ... 5 | 4 ... 0 |
values:   | V |   next    | ITT_addr |  Size   |
```

where:

- V indicates whether the entry is valid. If not, other fields are not meaningful.
- next: equals to 0 if this entry is the last one; otherwise it corresponds to the DeviceID offset to the next DTE, capped by 2^14 -1.
- ITT_addr matches bits [51:8] of the ITT address (256 Byte aligned).
- Size specifies the supported number of bits for the EventID, minus one

Collection Table Entry (CTE):

```
bits:     | 63| 62 ..  52  | 51 ... 16 | 15  ...   0 |
values:   | V |    RES0    |  RDBase   |    ICID     |
```

where:

- V indicates whether the entry is valid. If not, other fields are not meaningful.
- RES0: reserved field with Should-Be-Zero-or-Preserved behavior.
- RDBase is the PE number (GICR_TYPER.Processor_Number semantic),
- ICID is the collection ID

Interrupt Translation Entry (ITE):

```
bits:     | 63 ... 48 | 47 ... 16 | 15 ... 0 |
values:   |    next   |   pINTID  |  ICID    |
```

where:

- next: equals to 0 if this entry is the last one; otherwise it corresponds to the EventID offset to the next ITE capped by 2^16 -1.

- pINTID is the physical LPI ID; if zero, it means the entry is not valid and other fields are not meaningful.
- ICID is the collection ID

ITS Reset State:

RESET returns the ITS to the same state that it was when first created and initialized. When the RESET command returns, the following things are guaranteed:

- The ITS is not enabled and quiescent GITS_CTLR.Enabled = 0 .Quiescent=1
- There is no internally cached state
- No collection or device table are used GITS_BASER<n>.Valid = 0
- GITS_CBASER = 0, GITS_CREADR = 0, GITS_CWRITER = 0
- The ABI version is unchanged and remains the one set when the ITS device was first created.

1.2.2 ARM Virtual Generic Interrupt Controller v2 (VGIC)

Device types supported:

- KVM_DEV_TYPE_ARM_VGIC_V2 ARM Generic Interrupt Controller v2.0

Only one VGIC instance may be instantiated through either this API or the legacy KVM_CREATE_IRQCHIP API. The created VGIC will act as the VM interrupt controller, requiring emulated user-space devices to inject interrupts to the VGIC instead of directly to CPUs.

GICv3 implementations with hardware compatibility support allow creating a guest GICv2 through this interface. For information on creating a guest GICv3 device and guest ITS devices, see arm-vgic-v3.txt. It is not possible to create both a GICv3 and GICv2 device on the same VM.

Groups:

KVM_DEV_ARM_VGIC_GRP_ADDR
Attributes:

KVM_VGIC_V2_ADDR_TYPE_DIST (rw, 64-bit)
Base address in the guest physical address space of the GIC distributor register mappings. Only valid for KVM_DEV_TYPE_ARM_VGIC_V2. This address needs to be 4K aligned and the region covers 4 KByte.

KVM_VGIC_V2_ADDR_TYPE_CPU (rw, 64-bit)
Base address in the guest physical address space of the GIC virtual cpu interface register mappings. Only valid for KVM_DEV_TYPE_ARM_VGIC_V2. This address needs to be 4K aligned and the region covers 4 KByte.

Errors:

-E2BIG	Address outside of addressable IPA range
-EINVAL	Incorrectly aligned address
-EEXIST	Address already configured
-ENXIO	The group or attribute is unknown/unsupported for this device or hardware support is missing.
-EFAULT	Invalid user pointer for attr->addr.

KVM_DEV_ARM_VGIC_GRP_DIST_REGS

Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63   ....  40 | 39 ..  32  |  31   ....    0 |
values:   |    reserved   | vcpu_index |      offset     |
```

All distributor regs are (rw, 32-bit)

The offset is relative to the "Distributor base address" as defined in the GICv2 specs. Getting or setting such a register has the same effect as reading or writing the register on the actual hardware from the cpu whose index is specified with the vcpu_index field. Note that most distributor fields are not banked, but return the same value regardless of the vcpu_index used to access the register.

GICD_IIDR.Revision is updated when the KVM implementation of an emulated GICv2 is changed in a way directly observable by the guest or userspace. Userspace should read GICD_IIDR from KVM and write back the read value to confirm its expected behavior is aligned with the KVM implementation. Userspace should set GICD_IIDR before setting any other registers (both KVM_DEV_ARM_VGIC_GRP_DIST_REGS and KVM_DEV_ARM_VGIC_GRP_CPU_REGS) to ensure the expected behavior. Unless GICD_IIDR has been set from userspace, writes to the interrupt group registers (GICD_IGROUPR) are ignored.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	One or more VCPUs are running
-EINVAL	Invalid vcpu_index supplied

KVM_DEV_ARM_VGIC_GRP_CPU_REGS

Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63   ....  40 | 39 ..  32  |  31   ....    0 |
values:   |    reserved   | vcpu_index |      offset     |
```

All CPU interface regs are (rw, 32-bit)

The offset specifies the offset from the "CPU interface base address" as defined in the GICv2 specs. Getting or setting such a register has the same effect as reading or writing the register on the actual hardware.

The Active Priorities Registers APRn are implementation defined, so we set a fixed format for our implementation that fits with the model of a ”GICv2 implementation without the security extensions” which we present to the guest. This interface always exposes four register APR[0-3] describing the maximum possible 128 preemption levels. The semantics of the register indicate if any interrupts in a given preemption level are in the active state by setting the corresponding bit.

Thus, preemption level X has one or more active interrupts if and only if:

APRn[X mod 32] == 0b1, where n = X / 32

Bits for undefined preemption levels are RAZ/WI.

Note that this differs from a CPU's view of the APRs on hardware in which a GIC without the security extensions expose group 0 and group 1 active priorities in separate register groups, whereas we show a combined view similar to GICv2's GICH_APR.

For historical reasons and to provide ABI compatibility with userspace we export the GICC_PMR register in the format of the GICH_VMCR.VMPriMask field in the lower 5 bits of a word, meaning that userspace must always use the lower 5 bits to communicate with the KVM device and must shift the value left by 3 places to obtain the actual priority mask level.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	One or more VCPUs are running
-EINVAL	Invalid vcpu_index supplied

KVM_DEV_ARM_VGIC_GRP_NR_IRQS
Attributes:

A value describing the number of interrupts (SGI, PPI and SPI) for this GIC instance, ranging from 64 to 1024, in increments of 32.

Errors:

-EINVAL	Value set is out of the expected range
-EBUSY	Value has already be set, or GIC has already been initialized with default values.

KVM_DEV_ARM_VGIC_GRP_CTRL
Attributes:

KVM_DEV_ARM_VGIC_CTRL_INIT
request the initialization of the VGIC or ITS, no additional parameter in kvm_device_attr.addr.

Errors:

-ENXIO	VGIC not properly configured as required prior to calling this attribute
-ENODEV	no online VCPU
-ENOMEM	memory shortage when allocating vgic internal data

1.2.3 ARM Virtual Generic Interrupt Controller v3 and later (VGICv3)

Device types supported:

- KVM_DEV_TYPE_ARM_VGIC_V3 ARM Generic Interrupt Controller v3.0

Only one VGIC instance may be instantiated through this API. The created VGIC will act as the VM interrupt controller, requiring emulated user-space devices to inject interrupts to the VGIC instead of directly to CPUs. It is not possible to create both a GICv3 and GICv2 on the same VM.

Creating a guest GICv3 device requires a host GICv3 as well.

Groups:

KVM_DEV_ARM_VGIC_GRP_ADDR
Attributes:

KVM_VGIC_V3_ADDR_TYPE_DIST (rw, 64-bit)
Base address in the guest physical address space of the GICv3 distributor register mappings. Only valid for KVM_DEV_TYPE_ARM_VGIC_V3. This address needs to be 64K aligned and the region covers 64 KByte.

KVM_VGIC_V3_ADDR_TYPE_REDIST (rw, 64-bit)
Base address in the guest physical address space of the GICv3 redistributor register mappings. There are two 64K pages for each VCPU and all of the redistributor pages are contiguous. Only valid for KVM_DEV_TYPE_ARM_VGIC_V3. This address needs to be 64K aligned.

KVM_VGIC_V3_ADDR_TYPE_REDIST_REGION (rw, 64-bit)
The attribute data pointed to by kvm_device_attr.addr is a __u64 value:

```
bits:     | 63   ....  52  |  51   ....   16 | 15 - 12  |11 -␣
↪0
values:   |     count      |       base      |  flags   |␣
↪index
```

- index encodes the unique redistributor region index
- flags: reserved for future use, currently 0
- base field encodes bits [51:16] of the guest physical base address of the first redistributor in the region.
- count encodes the number of redistributors in the region. Must be greater than 0.

There are two 64K pages for each redistributor in the region and redistributors are laid out contiguously within the region. Regions are filled with redistributors in the index order. The sum of all region count fields must be

greater than or equal to the number of VCPUs. Redistributor regions must be registered in the incremental index order, starting from index 0.

The characteristics of a specific redistributor region can be read by presetting the index field in the attr data. Only valid for KVM_DEV_TYPE_ARM_VGIC_V3.

It is invalid to mix calls with KVM_VGIC_V3_ADDR_TYPE_REDIST and KVM_VGIC_V3_ADDR_TYPE_REDIST_REGION attributes.

Note that to obtain reproducible results (the same VCPU being associated with the same redistributor across a save/restore operation), VCPU creation order, redistributor region creation order as well as the respective interleaves of VCPU and region creation MUST be preserved. Any change in either ordering may result in a different vcpu_id/redistributor association, resulting in a VM that will fail to run at restore time.

Errors:

-E2BIG	Address outside of addressable IPA range
-EINVAL	Incorrectly aligned address, bad redistributor region count/index, mixed redistributor region attribute usage
-EEXIST	Address already configured
-ENOENT	Attempt to read the characteristics of a non existing redistributor region
-ENXIO	The group or attribute is unknown/unsupported for this device or hardware support is missing.
-EFAULT	Invalid user pointer for attr->addr.

KVM_DEV_ARM_VGIC_GRP_DIST_REGS, KVM_DEV_ARM_VGIC_GRP_REDIST_REGS

Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63   ....  32  |  31   ....    0 |
values:   |      mpidr     |      offset     |
```

All distributor regs are (rw, 32-bit) and kvm_device_attr.addr points to a __u32 value. 64-bit registers must be accessed by separately accessing the lower and higher word.

Writes to read-only registers are ignored by the kernel.

KVM_DEV_ARM_VGIC_GRP_DIST_REGS accesses the main distributor registers. KVM_DEV_ARM_VGIC_GRP_REDIST_REGS accesses the redistributor of the CPU specified by the mpidr.

The offset is relative to the ”[Re]Distributor base address” as defined in the GICv3/4 specs. Getting or setting such a register has the same effect as reading or writing the register on real hardware, except for the following registers: GICD_STATUSR, GICR_STATUSR, GICD_ISPENDR, GICR_ISPENDR0, GICD_ICPENDR, and GICR_ICPENDR0. These registers behave differently when accessed via this interface compared to their architecturally defined behavior to allow software a full view of the VGIC's internal state.

The mpidr field is used to specify which redistributor is accessed. The mpidr is ignored for the distributor.

The mpidr encoding is based on the affinity information in the architecture defined MPIDR, and the field is encoded as follows:

```
| 63 .... 56 | 55 .... 48 | 47 .... 40 | 39 .... 32 |
|    Aff3    |    Aff2    |    Aff1    |    Aff0    |
```

Note that distributor fields are not banked, but return the same value regardless of the mpidr used to access the register.

GICD_IIDR.Revision is updated when the KVM implementation is changed in a way directly observable by the guest or userspace. Userspace should read GICD_IIDR from KVM and write back the read value to confirm its expected behavior is aligned with the KVM implementation. Userspace should set GICD_IIDR before setting any other registers to ensure the expected behavior.

The GICD_STATUSR and GICR_STATUSR registers are architecturally defined such that a write of a clear bit has no effect, whereas a write with a set bit clears that value. To allow userspace to freely set the values of these two registers, setting the attributes with the register offsets for these two registers simply sets the non-reserved bits to the value written.

Accesses (reads and writes) to the GICD_ISPENDR register region and GICR_ISPENDR0 registers get/set the value of the latched pending state for the interrupts.

This is identical to the value returned by a guest read from ISPENDR for an edge triggered interrupt, but may differ for level triggered interrupts. For edge triggered interrupts, once an interrupt becomes pending (whether because of an edge detected on the input line or because of a guest write to ISPENDR) this state is ”latched”, and only cleared when either the interrupt is activated or when the guest writes to ICPENDR. A level triggered interrupt may be pending either because the level input is held high by a device, or because of a guest write to the ISPENDR register. Only ISPENDR writes are latched; if the device lowers the line level then the interrupt is no longer pending unless the guest also wrote to ISPENDR, and conversely writes to ICPENDR or activations of the interrupt do not clear the pending status if the line level is still being held high. (These rules are documented in the GICv3 specification descriptions of the ICPENDR and ISPENDR registers.) For a level triggered interrupt the value accessed here is that of the latch which is set by ISPENDR and cleared by ICPENDR or interrupt activation, whereas the value returned by a guest read from ISPENDR is the logical OR of the latch value and the input line level.

Raw access to the latch state is provided to userspace so that it can save and restore the entire GIC internal state (which is defined by the combination of the current input line level and the latch state, and cannot be deduced from purely the line level and the value of the ISPENDR registers).

Accesses to GICD_ICPENDR register region and GICR_ICPENDR0 registers have RAZ/WI semantics, meaning that reads always return 0 and writes are always ignored.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	One or more VCPUs are running

KVM_DEV_ARM_VGIC_GRP_CPU_SYSREGS

Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63      ....       32 | 31  ....  16 | 15  ....  0 |
values:   |         mpidr         |      RES     |    instr    |
```

The mpidr field encodes the CPU ID based on the affinity information in the architecture defined MPIDR, and the field is encoded as follows:

```
| 63 .... 56 | 55 .... 48 | 47 .... 40 | 39 .... 32 |
|    Aff3    |    Aff2    |    Aff1    |    Aff0    |
```

The instr field encodes the system register to access based on the fields defined in the A64 instruction set encoding for system register access (RES means the bits are reserved for future use and should be zero):

```
| 15 ... 14 | 13 ... 11 | 10 ... 7 | 6 ... 3 | 2 ... 0 |
|   Op 0    |    Op1    |    CRn   |   CRm   |   Op2   |
```

All system regs accessed through this API are (rw, 64-bit) and kvm_device_attr.addr points to a __u64 value.

KVM_DEV_ARM_VGIC_GRP_CPU_SYSREGS accesses the CPU interface registers for the CPU specified by the mpidr field.

CPU interface registers access is not implemented for AArch32 mode. Error -ENXIO is returned when accessed in AArch32 mode.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	VCPU is running
-EINVAL	Invalid mpidr or register value supplied

KVM_DEV_ARM_VGIC_GRP_NR_IRQS

Attributes:

A value describing the number of interrupts (SGI, PPI and SPI) for this GIC instance, ranging from 64 to 1024, in increments of 32.

kvm_device_attr.addr points to a __u32 value.

Errors:

-EINVAL	Value set is out of the expected range
-EBUSY	Value has already be set.

KVM_DEV_ARM_VGIC_GRP_CTRL
Attributes:

KVM_DEV_ARM_VGIC_CTRL_INIT
request the initialization of the VGIC, no additional parameter in kvm_device_attr.addr. Must be called after all VCPUs have been created.

KVM_DEV_ARM_VGIC_SAVE_PENDING_TABLES
save all LPI pending bits into guest RAM pending tables.

The first kB of the pending table is not altered by this operation.

Errors:

-ENXIO	VGIC not properly configured as required prior to calling this attribute
-ENODEV	no online VCPU
-ENOMEM	memory shortage when allocating vgic internal data
-EFAULT	Invalid guest ram access
-EBUSY	One or more VCPUS are running

KVM_DEV_ARM_VGIC_GRP_LEVEL_INFO
Attributes:

The attr field of kvm_device_attr encodes the following values:

```
bits:     | 63      ....       32 | 31   ....    10 | 9  ....  0␣
↪|
values:   |         mpidr         |      info       |   vINTID  ␣
↪|
```

The vINTID specifies which set of IRQs is reported on.

The info field specifies which information userspace wants to get or set using this interface. Currently we support the following info values:

VGIC_LEVEL_INFO_LINE_LEVEL:
Get/Set the input level of the IRQ line for a set of 32 contiguously numbered interrupts.

vINTID must be a multiple of 32.

kvm_device_attr.addr points to a __u32 value which will contain a bitmap where a set bit means the interrupt level is asserted.

Bit[n] indicates the status for interrupt vINTID + n.

SGIs and any interrupt with a higher ID than the number of interrupts supported, will be RAZ/WI. LPIs are always edge-triggered and are therefore not supported by this interface.

PPIs are reported per VCPU as specified in the mpidr field, and SPIs are reported with the same value regardless of the mpidr specified.

The mpidr field encodes the CPU ID based on the affinity information in the architecture defined MPIDR, and the field is encoded as follows:

```
| 63 .... 56 | 55 .... 48 | 47 .... 40 | 39 .... 32 |
|    Aff3    |    Aff2    |    Aff1    |    Aff0    |
```

Errors:

-EINVAL	vINTID is not multiple of 32 or info field is not VGIC_LEVEL_INFO_LINE_LEVEL

1.2.4 MPIC interrupt controller

Device types supported:

- KVM_DEV_TYPE_FSL_MPIC_20 Freescale MPIC v2.0
- KVM_DEV_TYPE_FSL_MPIC_42 Freescale MPIC v4.2

Only one MPIC instance, of any type, may be instantiated. The created MPIC will act as the system interrupt controller, connecting to each vcpu's interrupt inputs.

Groups:

KVM_DEV_MPIC_GRP_MISC
Attributes:

KVM_DEV_MPIC_BASE_ADDR (rw, 64-bit)
Base address of the 256 KiB MPIC register space. Must be naturally aligned. A value of zero disables the mapping. Reset value is zero.

KVM_DEV_MPIC_GRP_REGISTER (rw, 32-bit)
Access an MPIC register, as if the access were made from the guest. ”attr” is the byte offset into the MPIC register space. Accesses must be 4-byte aligned.

MSIs may be signaled by using this attribute group to write to the relevant MSIIR.

KVM_DEV_MPIC_GRP_IRQ_ACTIVE (rw, 32-bit)
IRQ input line for each standard openpic source. 0 is inactive and 1 is active, regardless of interrupt sense.

For edge-triggered interrupts: Writing 1 is considered an activating edge, and writing 0 is ignored. Reading returns 1 if a previously signaled edge has not been acknowledged, and 0 otherwise.

”attr” is the IRQ number. IRQ numbers for standard sources are the byte offset of the relevant IVPR from EIVPR0, divided by 32.

IRQ Routing:

The MPIC emulation supports IRQ routing. Only a single MPIC device can be instantiated. Once that device has been created, it's available as irqchip id 0.

This irqchip 0 has 256 interrupt pins, which expose the interrupts in the main array of interrupt sources (a.k.a. ”SRC” interrupts).

The numbering is the same as the MPIC device tree binding -- based on the register offset from the beginning of the sources array, without regard to any subdivisions in chip documentation such as ”internal” or ”external” interrupts.

Access to non-SRC interrupts is not implemented through IRQ routing mechanisms.

1.2.5 FLIC (floating interrupt controller)

FLIC handles floating (non per-cpu) interrupts, i.e. I/O, service and some machine check interruptions. All interrupts are stored in a per-vm list of pending interrupts. FLIC performs operations on this list.

Only one FLIC instance may be instantiated.

FLIC provides support to - add interrupts (KVM_DEV_FLIC_ENQUEUE) - inspect currently pending interrupts (KVM_FLIC_GET_ALL_IRQS) - purge all pending floating interrupts (KVM_DEV_FLIC_CLEAR_IRQS) - purge one pending floating I/O interrupt (KVM_DEV_FLIC_CLEAR_IO_IRQ) - enable/disable for the guest transparent async page faults - register and modify adapter interrupt sources (KVM_DEV_FLIC_ADAPTER_*) - modify AIS (adapter-interruption-suppression) mode state (KVM_DEV_FLIC_AISM) - inject adapter interrupts on a specified adapter (KVM_DEV_FLIC_AIRQ_INJECT) - get/set all AIS mode states (KVM_DEV_FLIC_AISM_ALL)

Groups:

KVM_DEV_FLIC_ENQUEUE
Passes a buffer and length into the kernel which are then injected into the list of pending interrupts. attr->addr contains the pointer to the buffer and attr->attr contains the length of the buffer. The format of the data structure kvm_s390_irq as it is copied from userspace is defined in usr/include/linux/kvm.h.

KVM_DEV_FLIC_GET_ALL_IRQS
Copies all floating interrupts into a buffer provided by userspace. When the buffer is too small it returns -ENOMEM, which is the indication for userspace to try again with a bigger buffer.

-ENOBUFS is returned when the allocation of a kernelspace buffer has failed.

-EFAULT is returned when copying data to userspace failed. All interrupts remain pending, i.e. are not deleted from the list of currently pending interrupts. attr->addr contains the userspace address of the buffer into which all interrupt data will be copied. attr->attr contains the size of the buffer in bytes.

KVM_DEV_FLIC_CLEAR_IRQS
Simply deletes all elements from the list of currently pending floating interrupts. No interrupts are injected into the guest.

KVM_DEV_FLIC_CLEAR_IO_IRQ
Deletes one (if any) I/O interrupt for a subchannel identified by the subsystem identification word passed via the buffer specified by attr->addr (address) and attr->attr (length).

KVM_DEV_FLIC_APF_ENABLE
Enables async page faults for the guest. So in case of a major page fault the host is allowed to handle this async and continues the guest.

KVM_DEV_FLIC_APF_DISABLE_WAIT
Disables async page faults for the guest and waits until already pending async page faults are done. This is necessary to trigger a completion interrupt for every init interrupt before migrating the interrupt list.

KVM_DEV_FLIC_ADAPTER_REGISTER

Register an I/O adapter interrupt source. Takes a kvm_s390_io_adapter describing the adapter to register:

```
struct kvm_s390_io_adapter {
        __u32 id;
        __u8 isc;
        __u8 maskable;
        __u8 swap;
        __u8 flags;
};
```

id contains the unique id for the adapter, isc the I/O interruption subclass to use, maskable whether this adapter may be masked (interrupts turned off), swap whether the indicators need to be byte swapped, and flags contains further characteristics of the adapter.

Currently defined values for 'flags' are:

- KVM_S390_ADAPTER_SUPPRESSIBLE: adapter is subject to AIS (adapter-interrupt-suppression) facility. This flag only has an effect if the AIS capability is enabled.

Unknown flag values are ignored.

KVM_DEV_FLIC_ADAPTER_MODIFY

Modifies attributes of an existing I/O adapter interrupt source. Takes a kvm_s390_io_adapter_req specifying the adapter and the operation:

```
struct kvm_s390_io_adapter_req {
        __u32 id;
        __u8 type;
        __u8 mask;
        __u16 pad0;
        __u64 addr;
};
```

id specifies the adapter and type the operation. The supported operations are:

KVM_S390_IO_ADAPTER_MASK
mask or unmask the adapter, as specified in mask

KVM_S390_IO_ADAPTER_MAP
This is now a no-op. The mapping is purely done by the irq route.

KVM_S390_IO_ADAPTER_UNMAP
This is now a no-op. The mapping is purely done by the irq route.

KVM_DEV_FLIC_AISM

modify the adapter-interruption-suppression mode for a given isc if the AIS capability is enabled. Takes a kvm_s390_ais_req describing:

```
struct kvm_s390_ais_req {
        __u8 isc;
```

```
        __u16 mode;
};
```

isc contains the target I/O interruption subclass, mode the target adapter-interruption-suppression mode. The following modes are currently supported:

- KVM_S390_AIS_MODE_ALL: ALL-Interruptions Mode, i.e. airq injection is always allowed;
- KVM_S390_AIS_MODE_SINGLE: SINGLE-Interruption Mode, i.e. airq injection is only allowed once and the following adapter interrupts will be suppressed until the mode is set again to ALL-Interruptions or SINGLE-Interruption mode.

KVM_DEV_FLIC_AIRQ_INJECT
Inject adapter interrupts on a specified adapter. attr->attr contains the unique id for the adapter, which allows for adapter-specific checks and actions. For adapters subject to AIS, handle the airq injection suppression for an isc according to the adapter-interruption-suppression mode on condition that the AIS capability is enabled.

KVM_DEV_FLIC_AISM_ALL
Gets or sets the adapter-interruption-suppression mode for all ISCs. Takes a kvm_s390_ais_all describing:

```
struct kvm_s390_ais_all {
       __u8 simm; /* Single-Interruption-Mode mask */
       __u8 nimm; /* No-Interruption-Mode mask *
};
```

simm contains Single-Interruption-Mode mask for all ISCs, nimm contains No-Interruption-Mode mask for all ISCs. Each bit in simm and nimm corresponds to an ISC (MSB0 bit 0 to ISC 0 and so on). The combination of simm bit and nimm bit presents AIS mode for a ISC.

KVM_DEV_FLIC_AISM_ALL is indicated by KVM_CAP_S390_AIS_MIGRATION.

Note: The KVM_SET_DEVICE_ATTR/KVM_GET_DEVICE_ATTR device ioctls executed on FLIC with an unknown group or attribute gives the error code EINVAL (instead of ENXIO, as specified in the API documentation). It is not possible to conclude that a FLIC operation is unavailable based on the error code resulting from a usage attempt.

Note: The KVM_DEV_FLIC_CLEAR_IO_IRQ ioctl will return EINVAL in case a zero schid is specified.

1.2.6 Generic vcpu interface

The virtual cpu "device" also accepts the ioctls KVM_SET_DEVICE_ATTR, KVM_GET_DEVICE_ATTR, and KVM_HAS_DEVICE_ATTR. The interface uses the same struct kvm_device_attr as other devices, but targets VCPU-wide settings and controls.

The groups and attributes per virtual cpu, if any, are architecture specific.

1. GROUP: KVM_ARM_VCPU_PMU_V3_CTRL

Architectures
ARM64

1.1. ATTRIBUTE: KVM_ARM_VCPU_PMU_V3_IRQ

Parameters
in kvm_device_attr.addr the address for PMU overflow interrupt is a pointer to an int

Returns:

-EBUSY	The PMU overflow interrupt is already set
-EFAULT	Error reading interrupt number
-ENXIO	PMUv3 not supported or the overflow interrupt not set when attempting to get it
-ENODEV	KVM_ARM_VCPU_PMU_V3 feature missing from VCPU
-EINVAL	Invalid PMU overflow interrupt number supplied or trying to set the IRQ number without using an in-kernel irqchip.

A value describing the PMUv3 (Performance Monitor Unit v3) overflow interrupt number for this vcpu. This interrupt could be a PPI or SPI, but the interrupt type must be same for each vcpu. As a PPI, the interrupt number is the same for all vcpus, while as an SPI it must be a separate number per vcpu.

1.2 ATTRIBUTE: KVM_ARM_VCPU_PMU_V3_INIT

Parameters
no additional parameter in kvm_device_attr.addr

Returns:

-EEXIST	Interrupt number already used
-ENODEV	PMUv3 not supported or GIC not initialized
-ENXIO	PMUv3 not supported, missing VCPU feature or interrupt number not set
-EBUSY	PMUv3 already initialized

Request the initialization of the PMUv3. If using the PMUv3 with an in-kernel virtual GIC implementation, this must be done after initializing the in-kernel irqchip.

1.3 ATTRIBUTE: KVM_ARM_VCPU_PMU_V3_FILTER

Parameters

in kvm_device_attr.addr the address for a PMU event filter is a pointer to a struct kvm_pmu_event_filter

Returns

-ENODEV	PMUv3 not supported or GIC not initialized
-ENXIO	PMUv3 not properly configured or in-kernel irqchip not configured as required prior to calling this attribute
-EBUSY	PMUv3 already initialized or a VCPU has already run
-EINVAL	Invalid filter range

Request the installation of a PMU event filter described as follows:

```
struct kvm_pmu_event_filter {
        __u16       base_event;
        __u16       nevents;

#define KVM_PMU_EVENT_ALLOW 0
#define KVM_PMU_EVENT_DENY  1

        __u8        action;
        __u8        pad[3];
};
```

A filter range is defined as the range [@base_event, @base_event + @nevents), together with an @action (KVM_PMU_EVENT_ALLOW or KVM_PMU_EVENT_DENY). The first registered range defines the global policy (global ALLOW if the first @action is DENY, global DENY if the first @action is ALLOW). Multiple ranges can be programmed, and must fit within the event space defined by the PMU architecture (10 bits on ARMv8.0, 16 bits from ARMv8.1 onwards).

Note: ”Cancelling” a filter by registering the opposite action for the same range doesn't change the default action. For example, installing an ALLOW filter for event range [0:10) as the first filter and then applying a DENY action for the same range will leave the whole range as disabled.

Restrictions: Event 0 (SW_INCR) is never filtered, as it doesn't count a hardware event. Filtering event 0x1E (CHAIN) has no effect either, as it isn't strictly speaking an event. Filtering the cycle counter is possible using event 0x11 (CPU_CYCLES).

1.4 ATTRIBUTE: KVM_ARM_VCPU_PMU_V3_SET_PMU

Parameters

in kvm_device_attr.addr the address to an int representing the PMU identifier.

Returns

-EBUSY	PMUv3 already initialized, a VCPU has already run or an event filter has already been set
-EFAULT	Error accessing the PMU identifier
-ENXIO	PMU not found
-ENODEV	PMUv3 not supported or GIC not initialized
-ENOMEM	Could not allocate memory

Request that the VCPU uses the specified hardware PMU when creating guest events for the purpose of PMU emulation. The PMU identifier can be read from the ”type” file for the desired PMU instance under /sys/devices (or, equivalent, /sys/bus/even_source). This attribute is particularly useful on heterogeneous systems where there are at least two CPU PMUs on the system. The PMU that is set for one VCPU will be used by all the other VCPUs. It isn't possible to set a PMU if a PMU event filter is already present.

Note that KVM will not make any attempts to run the VCPU on the physical CPUs associated with the PMU specified by this attribute. This is entirely left to userspace. However, attempting to run the VCPU on a physical CPU not supported by the PMU will fail and KVM_RUN will return with exit_reason = KVM_EXIT_FAIL_ENTRY and populate the fail_entry struct by setting hardare_entry_failure_reason field to KVM_EXIT_FAIL_ENTRY_CPU_UNSUPPORTED and the cpu field to the processor id.

2. GROUP: KVM_ARM_VCPU_TIMER_CTRL

Architectures
ARM64

2.1. ATTRIBUTES: KVM_ARM_VCPU_TIMER_IRQ_VTIMER, KVM_ARM_VCPU_TIMER_IRQ_PTIMER

Parameters
in kvm_device_attr.addr the address for the timer interrupt is a pointer to an int

Returns:

-EINVAL	Invalid timer interrupt number
-EBUSY	One or more VCPUs has already run

A value describing the architected timer interrupt number when connected to an in-kernel virtual GIC. These must be a PPI (16 <= intid < 32). Setting the attribute overrides the default values (see below).

KVM_ARM_VCPU_TIMER_IRQ_VTIMER	The EL1 virtual timer intid (default: 27)
KVM_ARM_VCPU_TIMER_IRQ_PTIMER	The EL1 physical timer intid (default: 30)

Setting the same PPI for different timers will prevent the VCPUs from running. Setting the interrupt number on a VCPU configures all VCPUs created at that time to use the number provided for a given timer, overwriting any previously configured values on other VCPUs. Userspace should configure the interrupt numbers on at least one VCPU after creating all VCPUs and before running any VCPUs.

3. GROUP: KVM_ARM_VCPU_PVTIME_CTRL

Architectures
ARM64

3.1 ATTRIBUTE: KVM_ARM_VCPU_PVTIME_IPA

Parameters
64-bit base address

Returns:

-ENXIO	Stolen time not implemented
-EEXIST	Base address already set for this VCPU
-EINVAL	Base address not 64 byte aligned

Specifies the base address of the stolen time structure for this VCPU. The base address must be 64 byte aligned and exist within a valid guest memory region. See *Paravirtualized time support for arm64* for more information including the layout of the stolen time structure.

4. GROUP: KVM_VCPU_TSC_CTRL

Architectures
x86

4.1 ATTRIBUTE: KVM_VCPU_TSC_OFFSET

Parameters
64-bit unsigned TSC offset

Returns:

-EFAULT	Error reading/writing the provided parameter address.
-ENXIO	Attribute not supported

Specifies the guest's TSC offset relative to the host's TSC. The guest's TSC is then derived by the following equation:

guest_tsc = host_tsc + KVM_VCPU_TSC_OFFSET

This attribute is useful to adjust the guest's TSC on live migration, so that the TSC counts the time during which the VM was paused. The following describes a possible algorithm to use for this purpose.

From the source VMM process:

1. Invoke the KVM_GET_CLOCK ioctl to record the host TSC (tsc_src), kvmclock nanoseconds (guest_src), and host CLOCK_REALTIME nanoseconds (host_src).
2. Read the KVM_VCPU_TSC_OFFSET attribute for every vCPU to record the guest TSC offset (ofs_src[i]).
3. Invoke the KVM_GET_TSC_KHZ ioctl to record the frequency of the guest's TSC (freq).

From the destination VMM process:

4. Invoke the KVM_SET_CLOCK ioctl, providing the source nanoseconds from kvmclock (guest_src) and CLOCK_REALTIME (host_src) in their respective fields. Ensure that the KVM_CLOCK_REALTIME flag is set in the provided structure.

   KVM will advance the VM's kvmclock to account for elapsed time since recording the clock values. Note that this will cause problems in the guest (e.g., timeouts) unless CLOCK_REALTIME is synchronized between the source and destination, and a reasonably short time passes between the source pausing the VMs and the destination executing steps 4-7.

5. Invoke the KVM_GET_CLOCK ioctl to record the host TSC (tsc_dest) and kvmclock nanoseconds (guest_dest).

6. Adjust the guest TSC offsets for every vCPU to account for (1) time elapsed since recording state and (2) difference in TSCs between the source and destination machine:

   ofs_dst[i] = ofs_src[i] -
   (guest_src - guest_dest) * freq + (tsc_src - tsc_dest)

   ("ofs[i] + tsc - guest * freq" is the guest TSC value corresponding to a time of 0 in kvmclock. The above formula ensures that it is the same on the destination as it was on the source).

7. Write the KVM_VCPU_TSC_OFFSET attribute for every vCPU with the respective value derived in the previous step.

1.2.7 VFIO virtual device

Device types supported:

- KVM_DEV_TYPE_VFIO

Only one VFIO instance may be created per VM. The created device tracks VFIO files (group or device) in use by the VM and features of those groups/devices important to the correctness and acceleration of the VM. As groups/devices are enabled and disabled for use by the VM, KVM should be updated about their presence. When registered with KVM, a reference to the VFIO file is held by KVM.

Groups:

KVM_DEV_VFIO_FILE
alias: KVM_DEV_VFIO_GROUP

KVM_DEV_VFIO_FILE attributes:

KVM_DEV_VFIO_FILE_ADD: Add a VFIO file (group/device) to VFIO-KVM device tracking

kvm_device_attr.addr points to an int32_t file descriptor for the VFIO file.

KVM_DEV_VFIO_FILE_DEL: Remove a VFIO file (group/device) from VFIO-KVM device tracking

kvm_device_attr.addr points to an int32_t file descriptor for the VFIO file.

KVM_DEV_VFIO_GROUP (legacy kvm device group restricted to the handling of VFIO group fd):
KVM_DEV_VFIO_GROUP_ADD: same as KVM_DEV_VFIO_FILE_ADD for group fd only

KVM_DEV_VFIO_GROUP_DEL: same as KVM_DEV_VFIO_FILE_DEL for group fd only

KVM_DEV_VFIO_GROUP_SET_SPAPR_TCE: attaches a guest visible TCE table
allocated by sPAPR KVM. kvm_device_attr.addr points to a struct:

```
struct kvm_vfio_spapr_tce {
        __s32   groupfd;
        __s32   tablefd;
};
```

where:

- @groupfd is a file descriptor for a VFIO group;
- @tablefd is a file descriptor for a TCE table allocated via KVM_CREATE_SPAPR_TCE.

The FILE/GROUP_ADD operation above should be invoked prior to accessing the device file descriptor via VFIO_GROUP_GET_DEVICE_FD in order to support drivers which require a kvm pointer to be set in their .open_device() callback. It is the same for device file descriptor via character device open which gets device access via VFIO_DEVICE_BIND_IOMMUFD. For such file descriptors, FILE_ADD should be invoked before VFIO_DEVICE_BIND_IOMMUFD to support the drivers mentioned in prior sentence as well.

1.2.8 Generic vm interface

The virtual machine ”device” also accepts the ioctls KVM_SET_DEVICE_ATTR, KVM_GET_DEVICE_ATTR, and KVM_HAS_DEVICE_ATTR. The interface uses the same struct kvm_device_attr as other devices, but targets VM-wide settings and controls.

The groups and attributes per virtual machine, if any, are architecture specific.

1. GROUP: KVM_S390_VM_MEM_CTRL

Architectures
s390

1.1. ATTRIBUTE: KVM_S390_VM_MEM_ENABLE_CMMA

Parameters
none

Returns
-EBUSY if a vcpu is already defined, otherwise 0

Enables Collaborative Memory Management Assist (CMMA) for the virtual machine.

1.2. ATTRIBUTE: KVM_S390_VM_MEM_CLR_CMMA

Parameters
none

Returns
-EINVAL if CMMA was not enabled; 0 otherwise

Clear the CMMA status for all guest pages, so any pages the guest marked as unused are again used any may not be reclaimed by the host.

1.3. ATTRIBUTE KVM_S390_VM_MEM_LIMIT_SIZE

Parameters
in attr->addr the address for the new limit of guest memory

Returns
-EFAULT if the given address is not accessible; -EINVAL if the virtual machine is of type UCONTROL; -E2BIG if the given guest memory is to big for that machine; -EBUSY if a vcpu is already defined; -ENOMEM if not enough memory is available for a new shadow guest mapping; 0 otherwise.

Allows userspace to query the actual limit and set a new limit for the maximum guest memory size. The limit will be rounded up to 2048 MB, 4096 GB, 8192 TB respectively, as this limit is governed by the number of page table levels. In the case that there is no limit we will set the limit to KVM_S390_NO_MEM_LIMIT (U64_MAX).

2. GROUP: KVM_S390_VM_CPU_MODEL

Architectures
s390

2.1. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE (r/o)

Allows user space to retrieve machine and kvm specific cpu related information:

```
struct kvm_s390_vm_cpu_machine {
     __u64 cpuid;           # CPUID of host
     __u32 ibc;             # IBC level range offered by host
     __u8  pad[4];
     __u64 fac_mask[256];   # set of cpu facilities enabled by KVM
     __u64 fac_list[256];   # set of cpu facilities offered by host
}
```

Parameters
address of buffer to store the machine related cpu data of type struct kvm_s390_vm_cpu_machine*

Returns
-EFAULT if the given address is not accessible from kernel space; -ENOMEM if not enough memory is available to process the ioctl; 0 in case of success.

2.2. ATTRIBUTE: KVM_S390_VM_CPU_PROCESSOR (r/w)

Allows user space to retrieve or request to change cpu related information for a vcpu:

```
struct kvm_s390_vm_cpu_processor {
     __u64 cpuid;           # CPUID currently (to be) used by this vcpu
     __u16 ibc;             # IBC level currently (to be) used by this vcpu
     __u8  pad[6];
     __u64 fac_list[256];   # set of cpu facilities currently (to be) used
                            # by this vcpu
}
```

KVM does not enforce or limit the cpu model data in any form. Take the information retrieved by means of KVM_S390_VM_CPU_MACHINE as hint for reasonable configuration setups. Instruction interceptions triggered by additionally set facility bits that are not handled by KVM need to by implemented in the VM driver code.

Parameters

address of buffer to store/set the processor related cpu data of type struct kvm_s390_vm_cpu_processor*.

Returns

-EBUSY in case 1 or more vcpus are already activated (only in write case); -EFAULT if the given address is not accessible from kernel space; -ENOMEM if not enough memory is available to process the ioctl; 0 in case of success.

2.3. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_FEAT (r/o)

Allows user space to retrieve available cpu features. A feature is available if provided by the hardware and supported by kvm. In theory, cpu features could even be completely emulated by kvm.

```
struct kvm_s390_vm_cpu_feat {
      __u64 feat[16]; # Bitmap (1 = feature available), MSB 0 bit numbering
};
```

Parameters

address of a buffer to load the feature list from.

Returns

-EFAULT if the given address is not accessible from kernel space; 0 in case of success.

2.4. ATTRIBUTE: KVM_S390_VM_CPU_PROCESSOR_FEAT (r/w)

Allows user space to retrieve or change enabled cpu features for all VCPUs of a VM. Features that are not available cannot be enabled.

See *2.3. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_FEAT (r/o)* for a description of the parameter struct.

Parameters
address of a buffer to store/load the feature list from.

Returns
-EFAULT if the given address is not accessible from kernel space; -EINVAL if a cpu feature that is not available is to be enabled; -EBUSY if at least one VCPU has already been defined; 0 in case of success.

2.5. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_SUBFUNC (r/o)

Allows user space to retrieve available cpu subfunctions without any filtering done by a set IBC. These subfunctions are indicated to the guest VCPU via query or ”test bit” subfunctions and used e.g. by cpacf functions, plo and ptff.

A subfunction block is only valid if KVM_S390_VM_CPU_MACHINE contains the STFL(E) bit introducing the affected instruction. If the affected instruction indicates subfunctions via a ”query subfunction”, the response block is contained in the returned struct. If the affected instruction indicates subfunctions via a ”test bit” mechanism, the subfunction codes are contained in the returned struct in MSB 0 bit numbering.

```
struct kvm_s390_vm_cpu_subfunc {
     u8 plo[32];           # always valid (ESA/390 feature)
     u8 ptff[16];          # valid with TOD-clock steering
     u8 kmac[16];          # valid with Message-Security-Assist
     u8 kmc[16];           # valid with Message-Security-Assist
     u8 km[16];            # valid with Message-Security-Assist
     u8 kimd[16];          # valid with Message-Security-Assist
     u8 klmd[16];          # valid with Message-Security-Assist
     u8 pckmo[16];         # valid with Message-Security-Assist-Extension 3
     u8 kmctr[16];         # valid with Message-Security-Assist-Extension 4
     u8 kmf[16];           # valid with Message-Security-Assist-Extension 4
     u8 kmo[16];           # valid with Message-Security-Assist-Extension 4
     u8 pcc[16];           # valid with Message-Security-Assist-Extension 4
     u8 ppno[16];          # valid with Message-Security-Assist-Extension 5
     u8 kma[16];           # valid with Message-Security-Assist-Extension 8
     u8 kdsa[16];          # valid with Message-Security-Assist-Extension 9
     u8 reserved[1792];    # reserved for future instructions
};
```

Parameters
address of a buffer to load the subfunction blocks from.

Returns
-EFAULT if the given address is not accessible from kernel space; 0 in case of success.

2.6. ATTRIBUTE: KVM_S390_VM_CPU_PROCESSOR_SUBFUNC (r/w)

Allows user space to retrieve or change cpu subfunctions to be indicated for all VCPUs of a VM. This attribute will only be available if kernel and hardware support are in place.

The kernel uses the configured subfunction blocks for indication to the guest. A subfunction block will only be used if the associated STFL(E) bit has not been disabled by user space (so the instruction to be queried is actually available for the guest).

As long as no data has been written, a read will fail. The IBC will be used to determine available subfunctions in this case, this will guarantee backward compatibility.

See *2.5. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_SUBFUNC (r/o)* for a description of the parameter struct.

Parameters
address of a buffer to store/load the subfunction blocks from.

Returns
-EFAULT if the given address is not accessible from kernel space; -EINVAL when reading, if there was no write yet; -EBUSY if at least one VCPU has already been defined; 0 in case of success.

3. GROUP: KVM_S390_VM_TOD

Architectures
s390

3.1. ATTRIBUTE: KVM_S390_VM_TOD_HIGH

Allows user space to set/get the TOD clock extension (u8) (superseded by KVM_S390_VM_TOD_EXT).

Parameters
address of a buffer in user space to store the data (u8) to

Returns
-EFAULT if the given address is not accessible from kernel space; -EINVAL if setting the TOD clock extension to != 0 is not supported -EOPNOTSUPP for a PV guest (TOD managed by the ultravisor)

3.2. ATTRIBUTE: KVM_S390_VM_TOD_LOW

Allows user space to set/get bits 0-63 of the TOD clock register as defined in the POP (u64).

Parameters
address of a buffer in user space to store the data (u64) to

Returns
-EFAULT if the given address is not accessible from kernel space -EOPNOTSUPP for a PV guest (TOD managed by the ultravisor)

3.3. ATTRIBUTE: KVM_S390_VM_TOD_EXT

Allows user space to set/get bits 0-63 of the TOD clock register as defined in the POP (u64). If the guest CPU model supports the TOD clock extension (u8), it also allows user space to get/set it. If the guest CPU model does not support it, it is stored as 0 and not allowed to be set to a value != 0.

Parameters
address of a buffer in user space to store the data (kvm_s390_vm_tod_clock) to

Returns
-EFAULT if the given address is not accessible from kernel space; -EINVAL if setting the TOD clock extension to != 0 is not supported -EOPNOTSUPP for a PV guest (TOD managed by the ultravisor)

4. GROUP: KVM_S390_VM_CRYPTO

Architectures
s390

4.1. ATTRIBUTE: KVM_S390_VM_CRYPTO_ENABLE_AES_KW (w/o)

Allows user space to enable aes key wrapping, including generating a new wrapping key.

Parameters
none

Returns
0

4.2. ATTRIBUTE: KVM_S390_VM_CRYPTO_ENABLE_DEA_KW (w/o)

Allows user space to enable dea key wrapping, including generating a new wrapping key.

Parameters
none

Returns
0

4.3. ATTRIBUTE: KVM_S390_VM_CRYPTO_DISABLE_AES_KW (w/o)

Allows user space to disable aes key wrapping, clearing the wrapping key.

Parameters
none

Returns
0

4.4. ATTRIBUTE: KVM_S390_VM_CRYPTO_DISABLE_DEA_KW (w/o)

Allows user space to disable dea key wrapping, clearing the wrapping key.

Parameters
none

Returns
0

5. GROUP: KVM_S390_VM_MIGRATION

Architectures
s390

5.1. ATTRIBUTE: KVM_S390_VM_MIGRATION_STOP (w/o)

Allows userspace to stop migration mode, needed for PGSTE migration. Setting this attribute when migration mode is not active will have no effects.

Parameters
none

Returns
0

5.2. ATTRIBUTE: KVM_S390_VM_MIGRATION_START (w/o)

Allows userspace to start migration mode, needed for PGSTE migration. Setting this attribute when migration mode is already active will have no effects.

Dirty tracking must be enabled on all memslots, else -EINVAL is returned. When dirty tracking is disabled on any memslot, migration mode is automatically stopped.

Parameters
none

Returns
-ENOMEM if there is not enough free memory to start migration mode; -EINVAL if the state of the VM is invalid (e.g. no memory defined); 0 in case of success.

5.3. ATTRIBUTE: KVM_S390_VM_MIGRATION_STATUS (r/o)

Allows userspace to query the status of migration mode.

Parameters
address of a buffer in user space to store the data (u64) to; the data itself is either 0 if migration mode is disabled or 1 if it is enabled

Returns
-EFAULT if the given address is not accessible from kernel space; 0 in case of success.

6. GROUP: KVM_ARM_VM_SMCCC_CTRL

Architectures
arm64

6.1. ATTRIBUTE: KVM_ARM_VM_SMCCC_FILTER (w/o)

Parameters
Pointer to a `struct kvm_smccc_filter`

Returns

EEXIST	Range intersects with a previously inserted or reserved range
EBUSY	A vCPU in the VM has already run
EINVAL	Invalid filter configuration
ENOMEM	Failed to allocate memory for the in-kernel representation of the SMCCC filter

Requests the installation of an SMCCC call filter described as follows:

```
enum kvm_smccc_filter_action {
        KVM_SMCCC_FILTER_HANDLE = 0,
        KVM_SMCCC_FILTER_DENY,
        KVM_SMCCC_FILTER_FWD_TO_USER,
};

struct kvm_smccc_filter {
        __u32 base;
        __u32 nr_functions;
        __u8 action;
        __u8 pad[15];
};
```

The filter is defined as a set of non-overlapping ranges. Each range defines an action to be applied to SMCCC calls within the range. Userspace can insert multiple ranges into the filter by using successive calls to this attribute.

The default configuration of KVM is such that all implemented SMCCC calls are allowed. Thus, the SMCCC filter can be defined sparsely by userspace, only describing ranges that modify the default behavior.

The range expressed by `struct kvm_smccc_filter` is [`base`, `base + nr_functions`). The range is not allowed to wrap, i.e. userspace cannot rely on `base + nr_functions` overflowing.

The SMCCC filter applies to both SMC and HVC calls initiated by the guest. The SMCCC filter gates the in-kernel emulation of SMCCC calls and as such takes effect before other interfaces that interact with SMCCC calls (e.g. hypercall bitmap registers).

Actions:

- `KVM_SMCCC_FILTER_HANDLE`: Allows the guest SMCCC call to be handled in-kernel. It is strongly recommended that userspace *not* explicitly describe the allowed SMCCC call

ranges.

- KVM_SMCCC_FILTER_DENY: Rejects the guest SMCCC call in-kernel and returns to the guest.
- KVM_SMCCC_FILTER_FWD_TO_USER: The guest SMCCC call is forwarded to userspace with an exit reason of KVM_EXIT_HYPERCALL.

The pad field is reserved for future use and must be zero. KVM may return -EINVAL if the field is nonzero.

KVM reserves the 'Arm Architecture Calls' range of function IDs and will reject attempts to define a filter for any portion of these ranges:

Start	End (inclusive)
0x8000_0000	0x8000_FFFF
0xC000_0000	0xC000_FFFF

1.2.9 XICS interrupt controller

Device type supported: KVM_DEV_TYPE_XICS

Groups:

1. **KVM_DEV_XICS_GRP_SOURCES**
   Attributes:

   One per interrupt source, indexed by the source number.

2. **KVM_DEV_XICS_GRP_CTRL**
   Attributes:

   2.1 KVM_DEV_XICS_NR_SERVERS (write only)

   The kvm_device_attr.addr points to a __u32 value which is the number of interrupt server numbers (ie, highest possible vcpu id plus one).

   Errors:

-EINVAL	Value greater than KVM_MAX_VCPU_IDS.
-EFAULT	Invalid user pointer for attr->addr.
-EBUSY	A vcpu is already connected to the device.

This device emulates the XICS (eXternal Interrupt Controller Specification) defined in PAPR. The XICS has a set of interrupt sources, each identified by a 20-bit source number, and a set of Interrupt Control Presentation (ICP) entities, also called ”servers”, each associated with a virtual CPU.

The ICP entities are created by enabling the KVM_CAP_IRQ_ARCH capability for each vcpu, specifying KVM_CAP_IRQ_XICS in args[0] and the interrupt server number (i.e. the vcpu number from the XICS's point of view) in args[1] of the kvm_enable_cap struct. Each ICP has 64 bits of state which can be read and written using the KVM_GET_ONE_REG and KVM_SET_ONE_REG ioctls on the vcpu. The 64 bit state word has the following bitfields, starting at the least-significant end of the word:

- Unused, 16 bits

- Pending interrupt priority, 8 bits Zero is the highest priority, 255 means no interrupt is pending.
- Pending IPI (inter-processor interrupt) priority, 8 bits Zero is the highest priority, 255 means no IPI is pending.
- Pending interrupt source number, 24 bits Zero means no interrupt pending, 2 means an IPI is pending
- Current processor priority, 8 bits Zero is the highest priority, meaning no interrupts can be delivered, and 255 is the lowest priority.

Each source has 64 bits of state that can be read and written using the KVM_GET_DEVICE_ATTR and KVM_SET_DEVICE_ATTR ioctls, specifying the KVM_DEV_XICS_GRP_SOURCES attribute group, with the attribute number being the interrupt source number. The 64 bit state word has the following bitfields, starting from the least-significant end of the word:

- Destination (server number), 32 bits

  This specifies where the interrupt should be sent, and is the interrupt server number specified for the destination vcpu.

- Priority, 8 bits

  This is the priority specified for this interrupt source, where 0 is the highest priority and 255 is the lowest. An interrupt with a priority of 255 will never be delivered.

- Level sensitive flag, 1 bit

  This bit is 1 for a level-sensitive interrupt source, or 0 for edge-sensitive (or MSI).

- Masked flag, 1 bit

  This bit is set to 1 if the interrupt is masked (cannot be delivered regardless of its priority), for example by the ibm,int-off RTAS call, or 0 if it is not masked.

- Pending flag, 1 bit

  This bit is 1 if the source has a pending interrupt, otherwise 0.

Only one XICS instance may be created per VM.

1.2.10 POWER9 eXternal Interrupt Virtualization Engine (XIVE Gen1)

Device types supported:

- KVM_DEV_TYPE_XIVE POWER9 XIVE Interrupt Controller generation 1

This device acts as a VM interrupt controller. It provides the KVM interface to configure the interrupt sources of a VM in the underlying POWER9 XIVE interrupt controller.

Only one XIVE instance may be instantiated. A guest XIVE device requires a POWER9 host and the guest OS should have support for the XIVE native exploitation interrupt mode. If not, it should run using the legacy interrupt mode, referred as XICS (POWER7/8).

- Device Mappings

  The KVM device exposes different MMIO ranges of the XIVE HW which are required for interrupt management. These are exposed to the guest in VMAs populated with a custom VM fault handler.

1. Thread Interrupt Management Area (TIMA)

Each thread has an associated Thread Interrupt Management context composed of a set of registers. These registers let the thread handle priority management and interrupt acknowledgment. The most important are :

- Interrupt Pending Buffer (IPB)
- Current Processor Priority (CPPR)
- Notification Source Register (NSR)

They are exposed to software in four different pages each proposing a view with a different privilege. The first page is for the physical thread context and the second for the hypervisor. Only the third (operating system) and the fourth (user level) are exposed the guest.

2. Event State Buffer (ESB)

Each source is associated with an Event State Buffer (ESB) with either a pair of even/odd pair of pages which provides commands to manage the source: to trigger, to EOI, to turn off the source for instance.

3. Device pass-through

When a device is passed-through into the guest, the source interrupts are from a different HW controller (PHB4) and the ESB pages exposed to the guest should accommodate this change.

The passthru_irq helpers, kvmppc_xive_set_mapped() and kvmppc_xive_clr_mapped() are called when the device HW irqs are mapped into or unmapped from the guest IRQ number space. The KVM device extends these helpers to clear the ESB pages of the guest IRQ number being mapped and then lets the VM fault handler repopulate. The handler will insert the ESB page corresponding to the HW interrupt of the device being passed-through or the initial IPI ESB page if the device has being removed.

The ESB remapping is fully transparent to the guest and the OS device driver. All handling is done within VFIO and the above helpers in KVM-PPC.

- Groups:

1. **KVM_DEV_XIVE_GRP_CTRL**
   Provides global controls on the device

   Attributes:
   1.1 KVM_DEV_XIVE_RESET (write only) Resets the interrupt controller configuration for sources and event queues. To be used by kexec and kdump.

   Errors: none

   1.2 KVM_DEV_XIVE_EQ_SYNC (write only) Sync all the sources and queues and mark the EQ pages dirty. This to make sure that a consistent memory state is captured when migrating the VM.

   Errors: none

   1.3 KVM_DEV_XIVE_NR_SERVERS (write only) The kvm_device_attr.addr points to a __u32 value which is the number of interrupt server numbers (ie, highest possible vcpu id plus one).

   Errors:

-EINVAL	Value greater than KVM_MAX_VCPU_IDS.
-EFAULT	Invalid user pointer for attr->addr.
-EBUSY	A vCPU is already connected to the device.

2. **KVM_DEV_XIVE_GRP_SOURCE (write only)**
   Initializes a new source in the XIVE device and mask it.

   Attributes:
   Interrupt source number (64-bit)

   The kvm_device_attr.addr points to a __u64 value:

```
bits:     | 63   ....  2 |   1   |   0
values:   |    unused    | level | type
```

   - type: 0:MSI 1:LSI
   - level: assertion level in case of an LSI.

   Errors:

-E2BIG	Interrupt source number is out of range
-ENOMEM	Could not create a new source block
-EFAULT	Invalid user pointer for attr->addr.
-ENXIO	Could not allocate underlying HW interrupt

3. **KVM_DEV_XIVE_GRP_SOURCE_CONFIG (write only)**
   Configures source targeting

   Attributes:
   Interrupt source number (64-bit)

   The kvm_device_attr.addr points to a __u64 value:

```
bits:     | 63   ....  33 |  32  | 31 .. 3 |  2 .. 0
values:   |    eisn       | mask |  server | priority
```

   - priority: 0-7 interrupt priority level
   - server: CPU number chosen to handle the interrupt
   - mask: mask flag (unused)
   - eisn: Effective Interrupt Source Number

   Errors:

-ENOENT	Unknown source number
-EINVAL	Not initialized source number
-EINVAL	Invalid priority
-EINVAL	Invalid CPU number.
-EFAULT	Invalid user pointer for attr->addr.
-ENXIO	CPU event queues not configured or configuration of the underlying HW interrupt failed
-EBUSY	No CPU available to serve interrupt

4. **KVM_DEV_XIVE_GRP_EQ_CONFIG (read-write)**
   Configures an event queue of a CPU

   Attributes:
   EQ descriptor identifier (64-bit)

   The EQ descriptor identifier is a tuple (server, priority):

```
bits:     | 63   ....  32 | 31 .. 3 |  2 .. 0
values:   |    unused     |  server | priority
```

   The kvm_device_attr.addr points to:

```
struct kvm_ppc_xive_eq {
    __u32 flags;
    __u32 qshift;
    __u64 qaddr;
    __u32 qtoggle;
    __u32 qindex;
    __u8  pad[40];
};
```

   - **flags: queue flags**

     KVM_XIVE_EQ_ALWAYS_NOTIFY (required)
     forces notification without using the coalescing mechanism provided by the XIVE END ESBs.

   - qshift: queue size (power of 2)
   - qaddr: real address of queue
   - qtoggle: current queue toggle bit
   - qindex: current queue index
   - pad: reserved for future use

   Errors:

-ENOENT	Invalid CPU number
-EINVAL	Invalid priority
-EINVAL	Invalid flags
-EINVAL	Invalid queue size
-EINVAL	Invalid queue address
-EFAULT	Invalid user pointer for attr->addr.
-EIO	Configuration of the underlying HW failed

5. **KVM_DEV_XIVE_GRP_SOURCE_SYNC (write only)**
   Synchronize the source to flush event notifications

   Attributes:
   Interrupt source number (64-bit)

   Errors:

-ENOENT	Unknown source number
-EINVAL	Not initialized source number

- VCPU state

  The XIVE IC maintains VP interrupt state in an internal structure called the NVT. When a VP is not dispatched on a HW processor thread, this structure can be updated by HW if the VP is the target of an event notification.

  It is important for migration to capture the cached IPB from the NVT as it synthesizes the priorities of the pending interrupts. We capture a bit more to report debug information.

  KVM_REG_PPC_VP_STATE (2 * 64bits):

```
bits:     |  63  ....  32  |  31  ....  0  |
values:   |   TIMA word0   |   TIMA word1  |
bits:     | 127       ..........       64  |
values:   |            unused               |
```

- Migration:

  Saving the state of a VM using the XIVE native exploitation mode should follow a specific sequence. When the VM is stopped :

  1. Mask all sources (PQ=01) to stop the flow of events.

  2. Sync the XIVE device with the KVM control KVM_DEV_XIVE_EQ_SYNC to flush any in-flight event notification and to stabilize the EQs. At this stage, the EQ pages are marked dirty to make sure they are transferred in the migration sequence.

  3. Capture the state of the source targeting, the EQs configuration and the state of thread interrupt context registers.

  Restore is similar:

  1. Restore the EQ configuration. As targeting depends on it.
  2. Restore targeting
  3. Restore the thread interrupt contexts
  4. Restore the source states
  5. Let the vCPU run

1.3 ARM

1.3.1 Internal ABI between the kernel and HYP

This file documents the interaction between the Linux kernel and the hypervisor layer when running Linux as a hypervisor (for example KVM). It doesn't cover the interaction of the kernel with the hypervisor when running as a guest (under Xen, KVM or any other hypervisor), or any hypervisor-specific interaction when the kernel is used as a host.

Note: KVM/arm has been removed from the kernel. The API described here is still valid though, as it allows the kernel to kexec when booted at HYP. It can also be used by a hypervisor other than KVM if necessary.

On arm and arm64 (without VHE), the kernel doesn't run in hypervisor mode, but still needs to interact with it, allowing a built-in hypervisor to be either installed or torn down.

In order to achieve this, the kernel must be booted at HYP (arm) or EL2 (arm64), allowing it to install a set of stubs before dropping to SVC/EL1. These stubs are accessible by using a 'hvc #0' instruction, and only act on individual CPUs.

Unless specified otherwise, any built-in hypervisor must implement these functions (see arch/arm{,64}/include/asm/virt.h):

- ```
  r0/x0 = HVC_SET_VECTORS
  r1/x1 = vectors
  ```

  Set HVBAR/VBAR_EL2 to 'vectors' to enable a hypervisor. 'vectors' must be a physical address, and respect the alignment requirements of the architecture. Only implemented by the initial stubs, not by Linux hypervisors.

- ```
  r0/x0 = HVC_RESET_VECTORS
  ```

  Turn HYP/EL2 MMU off, and reset HVBAR/VBAR_EL2 to the initials stubs' exception vector value. This effectively disables an existing hypervisor.

- ```
  r0/x0 = HVC_SOFT_RESTART
  r1/x1 = restart address
  x2 = x0's value when entering the next payload (arm64)
  x3 = x1's value when entering the next payload (arm64)
  x4 = x2's value when entering the next payload (arm64)
  ```

  Mask all exceptions, disable the MMU, clear I+D bits, move the arguments into place (arm64 only), and jump to the restart address while at HYP/EL2. This hypercall is not expected to return to its caller.

- ```
  x0 = HVC_FINALISE_EL2 (arm64 only)
  ```

  Finish configuring EL2 depending on the command-line options, including an attempt to upgrade the kernel's exception level from EL1 to EL2 by enabling the VHE mode. This is conditioned by the CPU supporting VHE, the EL2 MMU being off, and VHE not being disabled by any other means (command line option, for example).

Any other value of r0/x0 triggers a hypervisor-specific handling, which is not documented here.

The return value of a stub hypercall is held by r0/x0, and is 0 on success, and HVC_STUB_ERR on error. A stub hypercall is allowed to clobber any of the caller-saved registers (x0-x18 on arm64, r0-r3 and ip on arm). It is thus recommended to use a function call to perform the hypercall.

1.3.2 ARM Hypercall Interface

KVM handles the hypercall services as requested by the guests. New hypercall services are regularly made available by the ARM specification or by KVM (as vendor services) if they make sense from a virtualization point of view.

This means that a guest booted on two different versions of KVM can observe two different "firmware" revisions. This could cause issues if a given guest is tied to a particular version of a hypercall service, or if a migration causes a different version to be exposed out of the blue to an unsuspecting guest.

In order to remedy this situation, KVM exposes a set of "firmware pseudo-registers" that can be manipulated using the GET/SET_ONE_REG interface. These registers can be saved/restored by userspace, and set to a convenient value as required.

The following registers are defined:

- KVM_REG_ARM_PSCI_VERSION:

  KVM implements the PSCI (Power State Coordination Interface) specification in order to provide services such as CPU on/off, reset and power-off to the guest.

  - Only valid if the vcpu has the KVM_ARM_VCPU_PSCI_0_2 feature set (and thus has already been initialized)
  - Returns the current PSCI version on GET_ONE_REG (defaulting to the highest PSCI version implemented by KVM and compatible with v0.2)
  - Allows any PSCI version implemented by KVM and compatible with v0.2 to be set with SET_ONE_REG
  - Affects the whole VM (even if the register view is per-vcpu)

- **KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1:**
  Holds the state of the firmware support to mitigate CVE-2017-5715, as offered by KVM to the guest via a HVC call. The workaround is described under SMCCC_ARCH_WORKAROUND_1 in [1].

  Accepted values are:

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1_NOT_AVAIL:
  KVM does not offer firmware support for the workaround. The mitigation status for the guest is unknown.

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1_AVAIL:
  The workaround HVC call is available to the guest and required for the mitigation.

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1_NOT_REQUIRED:
  The workaround HVC call is available to the guest, but it is not needed on this VCPU.

- **KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2:**
  Holds the state of the firmware support to mitigate CVE-2018-3639, as offered by KVM to the guest via a HVC call. The workaround is described under SMCCC_ARCH_WORKAROUND_2 in[1].

[1] https://developer.arm.com/-/media/developer/pdf/ARM_DEN_0070A_Firmware_interfaces_for_mitigating_CVE-2017-5715.pdf

Accepted values are:

KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_NOT_AVAIL:
A workaround is not available. KVM does not offer firmware support for the workaround.

KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_UNKNOWN:
The workaround state is unknown. KVM does not offer firmware support for the workaround.

KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_AVAIL:
The workaround is available, and can be disabled by a vCPU. If KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_ENABLED is set, it is active for this vCPU.

KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_NOT_REQUIRED:
The workaround is always active on this vCPU or it is not needed.

Bitmap Feature Firmware Registers

Contrary to the above registers, the following registers exposes the hypercall services in the form of a feature-bitmap to the userspace. This bitmap is translated to the services that are available to the guest. There is a register defined per service call owner and can be accessed via GET/SET_ONE_REG interface.

By default, these registers are set with the upper limit of the features that are supported. This way userspace can discover all the usable hypercall services via GET_ONE_REG. The user-space can write-back the desired bitmap back via SET_ONE_REG. The features for the registers that are untouched, probably because userspace isn't aware of them, will be exposed as is to the guest.

Note that KVM will not allow the userspace to configure the registers anymore once any of the vCPUs has run at least once. Instead, it will return a -EBUSY.

The pseudo-firmware bitmap register are as follows:

- **KVM_REG_ARM_STD_BMAP:**
  Controls the bitmap of the ARM Standard Secure Service Calls.

  The following bits are accepted:

  Bit-0: KVM_REG_ARM_STD_BIT_TRNG_V1_0:
  The bit represents the services offered under v1.0 of ARM True Random Number Generator (TRNG) specification, ARM DEN0098.

- **KVM_REG_ARM_STD_HYP_BMAP:**
  Controls the bitmap of the ARM Standard Hypervisor Service Calls.

  The following bits are accepted:

  Bit-0: KVM_REG_ARM_STD_HYP_BIT_PV_TIME:
  The bit represents the Paravirtualized Time service as represented by ARM DEN0057A.

- **KVM_REG_ARM_VENDOR_HYP_BMAP:**
  Controls the bitmap of the Vendor specific Hypervisor Service Calls.

  The following bits are accepted:

Bit-0: KVM_REG_ARM_VENDOR_HYP_BIT_FUNC_FEAT
The bit represents the ARM_SMCCC_VENDOR_HYP_KVM_FEATURES_FUNC_ID and ARM_SMCCC_VENDOR_HYP_CALL_UID_FUNC_ID function-ids.

Bit-1: KVM_REG_ARM_VENDOR_HYP_BIT_PTP:
The bit represents the Precision Time Protocol KVM service.

Errors:

-ENOENT	Unknown register accessed.
-EBUSY	Attempt a 'write' to the register after the VM has started.
-EINVAL	Invalid bitmap written to the register.

1.3.3 Paravirtualized time support for arm64

Arm specification DEN0057/A defines a standard for paravirtualised time support for AArch64 guests:

https://developer.arm.com/docs/den0057/a

KVM/arm64 implements the stolen time part of this specification by providing some hypervisor service calls to support a paravirtualized guest obtaining a view of the amount of time stolen from its execution.

Two new SMCCC compatible hypercalls are defined:

- PV_TIME_FEATURES: 0xC5000020
- PV_TIME_ST: 0xC5000021

These are only available in the SMC64/HVC64 calling convention as paravirtualized time is not available to 32 bit Arm guests. The existence of the PV_TIME_FEATURES hypercall should be probed using the SMCCC 1.1 ARCH_FEATURES mechanism before calling it.

PV_TIME_FEATURES

Function ID:	(uint32)	0xC5000020
PV_call_id:	(uint32)	The function to query for support. Currently only PV_TIME_ST is supported.
Return value:	(int64)	NOT_SUPPORTED (-1) or SUCCESS (0) if the relevant PV-time feature is supported by the hypervisor.

PV_TIME_ST

Function ID:	(uint32)	0xC5000021
Return value:	(int64)	IPA of the stolen time data structure for this VCPU. On failure: NOT_SUPPORTED (-1)

The IPA returned by PV_TIME_ST should be mapped by the guest as normal memory with inner and outer write back caching attributes, in the inner shareable domain. A total of 16 bytes from the IPA returned are guaranteed to be meaningfully filled by the hypervisor (see structure below).

PV_TIME_ST returns the structure for the calling VCPU.

Stolen Time

The structure pointed to by the PV_TIME_ST hypercall is as follows:

Field	Byte Length	Byte Offset	Description
Revision	4	0	Must be 0 for version 1.0
Attributes	4	4	Must be 0
Stolen time	8	8	Stolen time in unsigned nanoseconds indicating how much time this VCPU thread was involuntarily not running on a physical CPU.

All values in the structure are stored little-endian.

The structure will be updated by the hypervisor prior to scheduling a VCPU. It will be present within a reserved region of the normal memory given to the guest. The guest should not attempt to write into this memory. There is a structure per VCPU of the guest.

It is advisable that one or more 64k pages are set aside for the purpose of these structures and not used for other purposes, this enables the guest to map the region using 64k pages and avoids conflicting attributes with other memory.

For the user space interface see *Documentation/virt/kvm/devices/vcpu.rst*.

1.3.4 PTP_KVM support for arm/arm64

PTP_KVM is used for high precision time sync between host and guests. It relies on transferring the wall clock and counter value from the host to the guest using a KVM-specific hypercall.

- ARM_SMCCC_VENDOR_HYP_KVM_PTP_FUNC_ID: 0x86000001

This hypercall uses the SMC32/HVC32 calling convention:

ARM_SMCCC_VENDOR_HYP_KVM_PTP_FUNC_ID

Function ID:	(uint32)	0x86000001
Arguments:	(uint32)	KVM_PTP_VIRT_COUNTER(0) KVM_PTP_PHYS_COUNTER(1)
Return Values:	(int32) (uint32) (uint32) (uint32) (uint32)	NOT_SUPPORTED(-1) on error, or Upper 32 bits of wall clock time (r0) Lower 32 bits of wall clock time (r1) Upper 32 bits of counter (r2) Lower 32 bits of counter (r3)
Endianness:		No Restrictions.

1.3.5 vCPU feature selection on arm64

KVM/arm64 provides two mechanisms that allow userspace to configure the CPU features presented to the guest.

KVM_ARM_VCPU_INIT

The `KVM_ARM_VCPU_INIT` ioctl accepts a bitmap of feature flags (`struct kvm_vcpu_init::features`). Features enabled by this interface are *opt-in* and may change/extend UAPI. See *4.82 KVM_ARM_VCPU_INIT* for complete documentation of the features controlled by the ioctl.

Otherwise, all CPU features supported by KVM are described by the architected ID registers.

The ID Registers

The Arm architecture specifies a range of *ID Registers* that describe the set of architectural features supported by the CPU implementation. KVM initializes the guest's ID registers to the maximum set of CPU features supported by the system. The ID register values may be VM-scoped in KVM, meaning that the values could be shared for all vCPUs in a VM.

KVM allows userspace to *opt-out* of certain CPU features described by the ID registers by writing values to them via the `KVM_SET_ONE_REG` ioctl. The ID registers are mutable until the VM has started, i.e. userspace has called `KVM_RUN` on at least one vCPU in the VM. Userspace can discover what fields are mutable in the ID registers using the `KVM_ARM_GET_REG_WRITABLE_MASKS`. See the *ioctl documentation* for more details.

Userspace is allowed to *limit* or *mask* CPU features according to the rules outlined by the architecture in DDI0487J.a D19.1.3 'Principles of the ID scheme for fields in ID register'. KVM does not allow ID register values that exceed the capabilities of the system.

> **Warning:** It is **strongly recommended** that userspace modify the ID register values before accessing the rest of the vCPU's CPU register state. KVM may use the ID register values to control feature emulation. Interleaving ID register modification with other system register accesses may lead to unpredictable behavior.

1.4 KVM for s390 systems

1.4.1 The s390 DIAGNOSE call on KVM

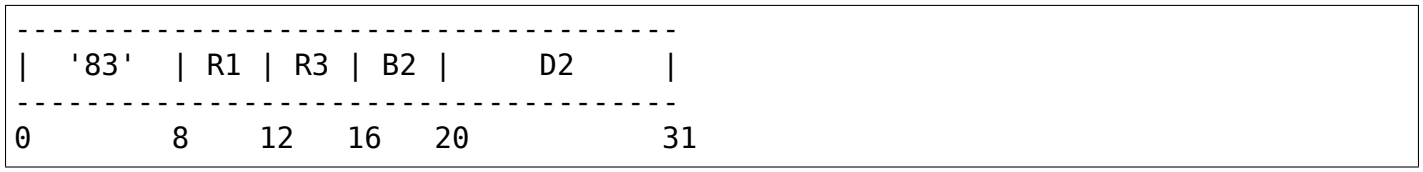
KVM on s390 supports the DIAGNOSE call for making hypercalls, both for native hypercalls and for selected hypercalls found on other s390 hypervisors.

Note that bits are numbered as by the usual s390 convention (most significant bit on the left).

General remarks

DIAGNOSE calls by the guest cause a mandatory intercept. This implies all supported DIAGNOSE calls need to be handled by either KVM or its userspace.

All DIAGNOSE calls supported by KVM use the RS-a format:



The second-operand address (obtained by the base/displacement calculation) is not used to address data. Instead, bits 48-63 of this address specify the function code, and bits 0-47 are ignored.

The supported DIAGNOSE function codes vary by the userspace used. For DIAGNOSE function codes not specific to KVM, please refer to the documentation for the s390 hypervisors defining them.

DIAGNOSE function code 'X'500' - KVM virtio functions

If the function code specifies 0x500, various virtio-related functions are performed.

General register 1 contains the virtio subfunction code. Supported virtio subfunctions depend on KVM's userspace. Generally, userspace provides either s390-virtio (subcodes 0-2) or virtio-ccw (subcode 3).

Upon completion of the DIAGNOSE instruction, general register 2 contains the function's return code, which is either a return code or a subcode specific value.

Subcode 0 - s390-virtio notification and early console printk
: Handled by userspace.

Subcode 1 - s390-virtio reset
: Handled by userspace.

Subcode 2 - s390-virtio set status
: Handled by userspace.

Subcode 3 - virtio-ccw notification
: Handled by either userspace or KVM (ioeventfd case).

  General register 2 contains a subchannel-identification word denoting the subchannel of the virtio-ccw proxy device to be notified.

  General register 3 contains the number of the virtqueue to be notified.

  General register 4 contains a 64bit identifier for KVM usage (the kvm_io_bus cookie). If general register 4 does not contain a valid identifier, it is ignored.

  After completion of the DIAGNOSE call, general register 2 may contain a 64bit identifier (in the kvm_io_bus cookie case), or a negative error value, if an internal error occurred.

  See also the virtio standard for a discussion of this hypercall.

DIAGNOSE function code 'X'501 - KVM breakpoint

If the function code specifies 0x501, breakpoint functions may be performed. This function code is handled by userspace.

This diagnose function code has no subfunctions and uses no parameters.

DIAGNOSE function code 'X'9C - Voluntary Time Slice Yield

General register 1 contains the target CPU address.

In a guest of a hypervisor like LPAR, KVM or z/VM using shared host CPUs, DIAGNOSE with function code 0x9c may improve system performance by yielding the host CPU on which the guest CPU is running to be assigned to another guest CPU, preferably the logical CPU containing the specified target CPU.

DIAG 'X'9C forwarding

The guest may send a DIAGNOSE 0x9c in order to yield to a certain other vcpu. An example is a Linux guest that tries to yield to the vcpu that is currently holding a spinlock, but not running.

However, on the host the real cpu backing the vcpu may itself not be running. Forwarding the DIAGNOSE 0x9c initially sent by the guest to yield to the backing cpu will hopefully cause that cpu, and thus subsequently the guest's vcpu, to be scheduled.

diag9c_forwarding_hz
: KVM kernel parameter allowing to specify the maximum number of DIAGNOSE 0x9c forwarding per second in the purpose of avoiding a DIAGNOSE 0x9c forwarding storm. A value of 0 turns the forwarding off.

1.4.2 s390 (IBM Z) Ultravisor and Protected VMs

Summary

Protected virtual machines (PVM) are KVM VMs that do not allow KVM to access VM state like guest memory or guest registers. Instead, the PVMs are mostly managed by a new entity called Ultravisor (UV). The UV provides an API that can be used by PVMs and KVM to request management actions.

Each guest starts in non-protected mode and then may make a request to transition into protected mode. On transition, KVM registers the guest and its VCPUs with the Ultravisor and prepares everything for running it.

The Ultravisor will secure and decrypt the guest's boot memory (i.e. kernel/initrd). It will safeguard state changes like VCPU starts/stops and injected interrupts while the guest is running.

As access to the guest's state, such as the SIE state description, is normally needed to be able to run a VM, some changes have been made in the behavior of the SIE instruction. A new format 4 state description has been introduced, where some fields have different meanings for a PVM. SIE exits are minimized as much as possible to improve speed and reduce exposed guest state.

Interrupt injection

Interrupt injection is safeguarded by the Ultravisor. As KVM doesn't have access to the VCPUs' lowcores, injection is handled via the format 4 state description.

Machine check, external, IO and restart interruptions each can be injected on SIE entry via a bit in the interrupt injection control field (offset 0x54). If the guest cpu is not enabled for the interrupt at the time of injection, a validity interception is recognized. The format 4 state description contains fields in the interception data block where data associated with the interrupt can be transported.

Program and Service Call exceptions have another layer of safeguarding; they can only be injected for instructions that have been intercepted into KVM. The exceptions need to be a valid outcome of an instruction emulation by KVM, e.g. we can never inject a addressing exception as they are reported by SIE since KVM has no access to the guest memory.

Mask notification interceptions

KVM cannot intercept lctl(g) and lpsw(e) anymore in order to be notified when a PVM enables a certain class of interrupt. As a replacement, two new interception codes have been introduced: One indicating that the contents of CRs 0, 6, or 14 have been changed, indicating different interruption subclasses; and one indicating that PSW bit 13 has been changed, indicating that a machine check intervention was requested and those are now enabled.

Instruction emulation

With the format 4 state description for PVMs, the SIE instruction already interprets more instructions than it does with format 2. It is not able to interpret every instruction, but needs to hand some tasks to KVM; therefore, the SIE and the ultravisor safeguard emulation inputs and outputs.

The control structures associated with SIE provide the Secure Instruction Data Area (SIDA), the Interception Parameters (IP) and the Secure Interception General Register Save Area. Guest GRs and most of the instruction data, such as I/O data structures, are filtered. Instruction data is copied to and from the SIDA when needed. Guest GRs are put into / retrieved from the Secure Interception General Register Save Area.

Only GR values needed to emulate an instruction will be copied into this save area and the real register numbers will be hidden.

The Interception Parameters state description field still contains the bytes of the instruction text, but with pre-set register values instead of the actual ones. I.e. each instruction always uses the same instruction text, in order not to leak guest instruction text. This also implies that the register content that a guest had in r<n> may be in r<m> from the hypervisor's point of view.

The Secure Instruction Data Area contains instruction storage data. Instruction data, i.e. data being referenced by an instruction like the SCCB for sclp, is moved via the SIDA. When an instruction is intercepted, the SIE will only allow data and program interrupts for this instruction to be moved to the guest via the two data areas discussed before. Other data is either ignored or results in validity interceptions.

Instruction emulation interceptions

There are two types of SIE secure instruction intercepts: the normal and the notification type. Normal secure instruction intercepts will make the guest pending for instruction completion of the intercepted instruction type, i.e. on SIE entry it is attempted to complete emulation of the instruction with the data provided by KVM. That might be a program exception or instruction completion.

The notification type intercepts inform KVM about guest environment changes due to guest instruction interpretation. Such an interception is recognized, for example, for the store prefix instruction to provide the new lowcore location. On SIE reentry, any KVM data in the data areas is ignored and execution continues as if the guest instruction had completed. For that reason KVM is not allowed to inject a program interrupt.

Links

KVM Forum 2019 presentation

1.4.3 s390 (IBM Z) Boot/IPL of Protected VMs

Summary

The memory of Protected Virtual Machines (PVMs) is not accessible to I/O or the hypervisor. In those cases where the hypervisor needs to access the memory of a PVM, that memory must be made accessible. Memory made accessible to the hypervisor will be encrypted. See *s390 (IBM Z) Ultravisor and Protected VMs* for details."

On IPL (boot) a small plaintext bootloader is started, which provides information about the encrypted components and necessary metadata to KVM to decrypt the protected virtual machine.

Based on this data, KVM will make the protected virtual machine known to the Ultravisor (UV) and instruct it to secure the memory of the PVM, decrypt the components and verify the data and address list hashes, to ensure integrity. Afterwards KVM can run the PVM via the SIE instruction which the UV will intercept and execute on KVM's behalf.

As the guest image is just like an opaque kernel image that does the switch into PV mode itself, the user can load encrypted guest executables and data via every available method (network, dasd, scsi, direct kernel, ...) without the need to change the boot process.

Diag308

This diagnose instruction is the basic mechanism to handle IPL and related operations for virtual machines. The VM can set and retrieve IPL information blocks, that specify the IPL method/devices and request VM memory and subsystem resets, as well as IPLs.

For PVMs this concept has been extended with new subcodes:

Subcode 8: Set an IPL Information Block of type 5 (information block for PVMs) Subcode 9: Store the saved block in guest memory Subcode 10: Move into Protected Virtualization mode

The new PV load-device-specific-parameters field specifies all data that is necessary to move into PV mode.

- PV Header origin
- PV Header length
- **List of Components composed of**
  - AES-XTS Tweak prefix
  - Origin
  - Size

The PV header contains the keys and hashes, which the UV will use to decrypt and verify the PV, as well as control flags and a start PSW.

The components are for instance an encrypted kernel, kernel parameters and initrd. The components are decrypted by the UV.

After the initial import of the encrypted data, all defined pages will contain the guest content. All non-specified pages will start out as zero pages on first access.

When running in protected virtualization mode, some subcodes will result in exceptions or return error codes.

Subcodes 4 and 7, which specify operations that do not clear the guest memory, will result in specification exceptions. This is because the UV will clear all memory when a secure VM is removed, and therefore non-clearing IPL subcodes are not allowed.

Subcodes 8, 9, 10 will result in specification exceptions. Re-IPL into a protected mode is only possible via a detour into non protected mode.

Keys

Every CEC will have a unique public key to enable tooling to build encrypted images. See s390-tools for the tooling.

1.4.4 s390 (IBM Z) Protected Virtualization dumps

Summary

Dumping a VM is an essential tool for debugging problems inside it. This is especially true when a protected VM runs into trouble as there's no way to access its memory and registers from the outside while it's running.

However when dumping a protected VM we need to maintain its confidentiality until the dump is in the hands of the VM owner who should be the only one capable of analysing it.

The confidentiality of the VM dump is ensured by the Ultravisor who provides an interface to KVM over which encrypted CPU and memory data can be requested. The encryption is based on the Customer Communication Key which is the key that's used to encrypt VM data in a way that the customer is able to decrypt.

Dump process

A dump is done in 3 steps:

Initiation

This step initializes the dump process, generates cryptographic seeds and extracts dump keys with which the VM dump data will be encrypted.

Data gathering

Currently there are two types of data that can be gathered from a VM: the memory and the vcpu state.

The vcpu state contains all the important registers, general, floating point, vector, control and tod/timers of a vcpu. The vcpu dump can contain incomplete data if a vcpu is dumped while an instruction is emulated with help of the hypervisor. This is indicated by a flag bit in the dump data. For the same reason it is very important to not only write out the encrypted vcpu state, but also the unencrypted state from the hypervisor.

The memory state is further divided into the encrypted memory and its metadata comprised of the encryption tweaks and status flags. The encrypted memory can simply be read once it has been exported. The time of the export does not matter as no re-encryption is needed. Memory that has been swapped out and hence was exported can be read from the swap and written to the dump target without need for any special actions.

The tweaks / status flags for the exported pages need to be requested from the Ultravisor.

Finalization

The finalization step will provide the data needed to be able to decrypt the vcpu and memory data and end the dump process. When this step completes successfully a new dump initiation can be started.

1.5 The PPC KVM paravirtual interface

The basic execution principle by which KVM on PowerPC works is to run all kernel space code in PR=1 which is user space. This way we trap all privileged instructions and can emulate them accordingly.

Unfortunately that is also the downfall. There are quite some privileged instructions that needlessly return us to the hypervisor even though they could be handled differently.

This is what the PPC PV interface helps with. It takes privileged instructions and transforms them into unprivileged ones with some help from the hypervisor. This cuts down virtualization costs by about 50% on some of my benchmarks.

The code for that interface can be found in arch/powerpc/kernel/kvm*

1.5.1 Querying for existence

To find out if we're running on KVM or not, we leverage the device tree. When Linux is running on KVM, a node /hypervisor exists. That node contains a compatible property with the value ”linux,kvm”.

Once you determined you're running under a PV capable KVM, you can now use hypercalls as described below.

1.5.2 KVM hypercalls

Inside the device tree's /hypervisor node there's a property called 'hypercall-instructions'. This property contains at most 4 opcodes that make up the hypercall. To call a hypercall, just call these instructions.

The parameters are as follows:

Register	IN	OUT
r0	•	volatile
r3	1st parameter	Return code
r4	2nd parameter	1st output value
r5	3rd parameter	2nd output value
r6	4th parameter	3rd output value
r7	5th parameter	4th output value
r8	6th parameter	5th output value
r9	7th parameter	6th output value
r10	8th parameter	7th output value
r11	hypercall number	8th output value
r12	•	volatile

Hypercall definitions are shared in generic code, so the same hypercall numbers apply for x86 and powerpc alike with the exception that each KVM hypercall also needs to be ORed with the KVM vendor code which is (42 << 16).

Return codes can be as follows:

Code	Meaning
0	Success
12	Hypercall not implemented
<0	Error

1.5.3 The magic page

To enable communication between the hypervisor and guest there is a new shared page that contains parts of supervisor visible register state. The guest can map this shared page using the KVM hypercall KVM_HC_PPC_MAP_MAGIC_PAGE.

With this hypercall issued the guest always gets the magic page mapped at the desired location. The first parameter indicates the effective address when the MMU is enabled. The second parameter indicates the address in real mode, if applicable to the target. For now, we always map the page to -4096. This way we can access it using absolute load and store functions. The following instruction reads the first field of the magic page:

```
ld      rX, -4096(0)
```

The interface is designed to be extensible should there be need later to add additional registers to the magic page. If you add fields to the magic page, also define a new hypercall feature to indicate that the host can give you more registers. Only if the host supports the additional features, make use of them.

The magic page layout is described by struct kvm_vcpu_arch_shared in arch/powerpc/include/uapi/asm/kvm_para.h.

1.5.4 Magic page features

When mapping the magic page using the KVM hypercall KVM_HC_PPC_MAP_MAGIC_PAGE, a second return value is passed to the guest. This second return value contains a bitmap of available features inside the magic page.

The following enhancements to the magic page are currently available:

KVM_MAGIC_FEAT_SR	Maps SR registers r/w in the magic page
KVM_MAGIC_FEAT_MAS0_TO_SPRG7	Maps MASn, ESR, PIR and high SPRGs

For enhanced features in the magic page, please check for the existence of the feature before using them!

1.5.5 Magic page flags

In addition to features that indicate whether a host is capable of a particular feature we also have a channel for a guest to tell the host whether it's capable of something. This is what we call ”flags”.

Flags are passed to the host in the low 12 bits of the Effective Address.

The following flags are currently available for a guest to expose:

MAGIC_PAGE_FLAG_NOT_MAPPED_NX Guest handles NX bits correctly wrt magic page

1.5.6 MSR bits

The MSR contains bits that require hypervisor intervention and bits that do not require direct hypervisor intervention because they only get interpreted when entering the guest or don't have any impact on the hypervisor's behavior.

The following bits are safe to be set inside the guest:

- MSR_EE
- MSR_RI

If any other bit changes in the MSR, please still use mtmsr(d).

1.5.7 Patched instructions

The ”ld” and ”std” instructions are transformed to ”lwz” and ”stw” instructions respectively on 32-bit systems with an added offset of 4 to accommodate for big endianness.

The following is a list of mapping the Linux kernel performs when running as guest. Implementing any of those mappings is optional, as the instruction traps also act on the shared page. So calling privileged instructions still works as before.

From	To
mfmsr rX	ld rX, magic_page->msr
mfsprg rX, 0	ld rX, magic_page->sprg0
mfsprg rX, 1	ld rX, magic_page->sprg1
mfsprg rX, 2	ld rX, magic_page->sprg2
mfsprg rX, 3	ld rX, magic_page->sprg3
mfsrr0 rX	ld rX, magic_page->srr0
mfsrr1 rX	ld rX, magic_page->srr1
mfdar rX	ld rX, magic_page->dar
mfdsisr rX	lwz rX, magic_page->dsisr
mtmsr rX	std rX, magic_page->msr
mtsprg 0, rX	std rX, magic_page->sprg0
mtsprg 1, rX	std rX, magic_page->sprg1
mtsprg 2, rX	std rX, magic_page->sprg2
mtsprg 3, rX	std rX, magic_page->sprg3
mtsrr0 rX	std rX, magic_page->srr0
mtsrr1 rX	std rX, magic_page->srr1
mtdar rX	std rX, magic_page->dar
mtdsisr rX	stw rX, magic_page->dsisr
tlbsync	nop
mtmsrd rX, 0	b <special mtmsr section>
mtmsr rX	b <special mtmsr section>
mtmsrd rX, 1	b <special mtmsrd section>
[Book3S only]	
mtsrin rX, rY	b <special mtsrin section>
[BookE only]	
wrteei [0|1]	b <special wrteei section>

Some instructions require more logic to determine what's going on than a load or store instruc-

tion can deliver. To enable patching of those, we keep some RAM around where we can live translate instructions to. What happens is the following:

1) copy emulation code to memory
2) patch that code to fit the emulated instruction
3) patch that code to return to the original pc + 4
4) patch the original instruction to branch to the new code

That way we can inject an arbitrary amount of code as replacement for a single instruction. This allows us to check for pending interrupts when setting EE=1 for example.

1.5.8 Hypercall ABIs in KVM on PowerPC

1) KVM hypercalls (ePAPR)

These are ePAPR compliant hypercall implementation (mentioned above). Even generic hypercalls are implemented here, like the ePAPR idle hcall. These are available on all targets.

2) PAPR hypercalls

PAPR hypercalls are needed to run server PowerPC PAPR guests (-M pseries in QEMU). These are the same hypercalls that pHyp, the POWER hypervisor, implements. Some of them are handled in the kernel, some are handled in user space. This is only available on book3s_64.

3) OSI hypercalls

Mac-on-Linux is another user of KVM on PowerPC, which has its own hypercall (long before KVM). This is supported to maintain compatibility. All these hypercalls get forwarded to user space. This is only useful on book3s_32, but can be used with book3s_64 as well.

1.6 KVM for x86 systems

1.6.1 Secure Encrypted Virtualization (SEV)

Overview

Secure Encrypted Virtualization (SEV) is a feature found on AMD processors.

SEV is an extension to the AMD-V architecture which supports running virtual machines (VMs) under the control of a hypervisor. When enabled, the memory contents of a VM will be transparently encrypted with a key unique to that VM.

The hypervisor can determine the SEV support through the CPUID instruction. The CPUID function 0x8000001f reports information related to SEV:

```
0x8000001f[eax]:
                Bit[1]  indicates support for SEV
    ...
          [ecx]:
                Bits[31:0]  Number of encrypted guests supported simultaneously
```

If support for SEV is present, MSR 0xc001_0010 (MSR_AMD64_SYSCFG) and MSR 0xc001_0015 (MSR_K7_HWCR) can be used to determine if it can be enabled:

```
0xc001_0010:
        Bit[23]    1 = memory encryption can be enabled
                   0 = memory encryption can not be enabled

0xc001_0015:
        Bit[0]     1 = memory encryption can be enabled
                   0 = memory encryption can not be enabled
```

When SEV support is available, it can be enabled in a specific VM by setting the SEV bit before executing VMRUN.:

```
VMCB[0x90]:
        Bit[1]      1 = SEV is enabled
                    0 = SEV is disabled
```

SEV hardware uses ASIDs to associate a memory encryption key with a VM. Hence, the ASID for the SEV-enabled guests must be from 1 to a maximum value defined in the CPUID 0x8000001f[ecx] field.

SEV Key Management

The SEV guest key management is handled by a separate processor called the AMD Secure Processor (AMD-SP). Firmware running inside the AMD-SP provides a secure key management interface to perform common hypervisor activities such as encrypting bootstrap code, snapshot, migrating and debugging the guest. For more information, see the SEV Key Management spec [api-spec]

The main ioctl to access SEV is KVM_MEMORY_ENCRYPT_OP. If the argument to KVM_MEMORY_ENCRYPT_OP is NULL, the ioctl returns 0 if SEV is enabled and `ENOTTY` if it is disabled (on some older versions of Linux, the ioctl runs normally even with a NULL argument, and therefore will likely return `EFAULT`). If non-NULL, the argument to KVM_MEMORY_ENCRYPT_OP must be a struct kvm_sev_cmd:

```
struct kvm_sev_cmd {
        __u32 id;
        __u64 data;
        __u32 error;
        __u32 sev_fd;
};
```

The `id` field contains the subcommand, and the `data` field points to another struct containing arguments specific to command. The `sev_fd` should point to a file descriptor that is opened on the `/dev/sev` device, if needed (see individual commands).

On output, `error` is zero on success, or an error code. Error codes are defined in `<linux/psp-dev.h>`.

KVM implements the following commands to support common lifecycle events of SEV guests, such as launching, running, snapshotting, migrating and decommissioning.

1. KVM_SEV_INIT

The KVM_SEV_INIT command is used by the hypervisor to initialize the SEV platform context. In a typical workflow, this command should be the first command issued.

The firmware can be initialized either by using its own non-volatile storage or the OS can manage the NV storage for the firmware using the module parameter `init_ex_path`. If the file specified by `init_ex_path` does not exist or is invalid, the OS will create or override the file with output from PSP.

Returns: 0 on success, -negative on error

2. KVM_SEV_LAUNCH_START

The KVM_SEV_LAUNCH_START command is used for creating the memory encryption context. To create the encryption context, user must provide a guest policy, the owner's public Diffie-Hellman (PDH) key and session information.

Parameters: struct kvm_sev_launch_start (in/out)

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_start {
        __u32 handle;           /* if zero then firmware creates a new handle␣
↪*/
        __u32 policy;           /* guest's policy */

        __u64 dh_uaddr;         /* userspace address pointing to the guest␣
↪owner's PDH key */
        __u32 dh_len;

        __u64 session_addr;     /* userspace address which points to the guest␣
↪session information */
        __u32 session_len;
};
```

On success, the 'handle' field contains a new handle and on error, a negative value.

KVM_SEV_LAUNCH_START requires the `sev_fd` field to be valid.

For more details, see SEV spec Section 6.2.

3. KVM_SEV_LAUNCH_UPDATE_DATA

The KVM_SEV_LAUNCH_UPDATE_DATA is used for encrypting a memory region. It also calculates a measurement of the memory contents. The measurement is a signature of the memory contents that can be sent to the guest owner as an attestation that the memory was encrypted correctly by the firmware.

Parameters (in): struct kvm_sev_launch_update_data

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_update {
        __u64 uaddr;    /* userspace address to be encrypted (must be 16-byte␣
↪aligned) */
        __u32 len;      /* length of the data to be encrypted (must be 16-byte␣
↪aligned) */
};
```

For more details, see SEV spec Section 6.3.

4. KVM_SEV_LAUNCH_MEASURE

The KVM_SEV_LAUNCH_MEASURE command is used to retrieve the measurement of the data encrypted by the KVM_SEV_LAUNCH_UPDATE_DATA command. The guest owner may wait to provide the guest with confidential information until it can verify the measurement. Since the guest owner knows the initial contents of the guest at boot, the measurement can be verified by comparing it to what the guest owner expects.

If len is zero on entry, the measurement blob length is written to len and uaddr is unused.

Parameters (in): struct kvm_sev_launch_measure

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_measure {
        __u64 uaddr;    /* where to copy the measurement */
        __u32 len;      /* length of measurement blob */
};
```

For more details on the measurement verification flow, see SEV spec Section 6.4.

5. KVM_SEV_LAUNCH_FINISH

After completion of the launch flow, the KVM_SEV_LAUNCH_FINISH command can be issued to make the guest ready for the execution.

Returns: 0 on success, -negative on error

6. KVM_SEV_GUEST_STATUS

The KVM_SEV_GUEST_STATUS command is used to retrieve status information about a SEV-enabled guest.

Parameters (out): struct kvm_sev_guest_status

Returns: 0 on success, -negative on error

```
struct kvm_sev_guest_status {
        __u32 handle;   /* guest handle */
        __u32 policy;   /* guest policy */
        __u8 state;     /* guest state (see enum below) */
};
```

SEV guest state:

```
enum {
SEV_STATE_INVALID = 0;
SEV_STATE_LAUNCHING,    /* guest is currently being launched */
SEV_STATE_SECRET,       /* guest is being launched and ready to accept the␣
↪ciphertext data */
SEV_STATE_RUNNING,      /* guest is fully launched and running */
SEV_STATE_RECEIVING,    /* guest is being migrated in from another SEV machine␣
↪*/
SEV_STATE_SENDING       /* guest is getting migrated out to another SEV␣
↪machine */
};
```

7. KVM_SEV_DBG_DECRYPT

The KVM_SEV_DEBUG_DECRYPT command can be used by the hypervisor to request the firmware to decrypt the data at the given memory region.

Parameters (in): struct kvm_sev_dbg

Returns: 0 on success, -negative on error

```
struct kvm_sev_dbg {
        __u64 src_uaddr;        /* userspace address of data to decrypt */
        __u64 dst_uaddr;        /* userspace address of destination */
        __u32 len;              /* length of memory region to decrypt */
};
```

The command returns an error if the guest policy does not allow debugging.

8. KVM_SEV_DBG_ENCRYPT

The KVM_SEV_DEBUG_ENCRYPT command can be used by the hypervisor to request the firmware to encrypt the data at the given memory region.

Parameters (in): struct kvm_sev_dbg

Returns: 0 on success, -negative on error

```
struct kvm_sev_dbg {
        __u64 src_uaddr;        /* userspace address of data to encrypt */
        __u64 dst_uaddr;        /* userspace address of destination */
        __u32 len;              /* length of memory region to encrypt */
};
```

The command returns an error if the guest policy does not allow debugging.

9. KVM_SEV_LAUNCH_SECRET

The KVM_SEV_LAUNCH_SECRET command can be used by the hypervisor to inject secret data after the measurement has been validated by the guest owner.

Parameters (in): struct kvm_sev_launch_secret

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_secret {
        __u64 hdr_uaddr;        /* userspace address containing the packet␣
↪header */
        __u32 hdr_len;

        __u64 guest_uaddr;      /* the guest memory region where the secret␣
↪should be injected */
        __u32 guest_len;

        __u64 trans_uaddr;      /* the hypervisor memory region which contains␣
↪the secret */
        __u32 trans_len;
};
```

10. KVM_SEV_GET_ATTESTATION_REPORT

The KVM_SEV_GET_ATTESTATION_REPORT command can be used by the hypervisor to query the attestation report containing the SHA-256 digest of the guest memory and VMSA passed through the KVM_SEV_LAUNCH commands and signed with the PEK. The digest returned by the command should match the digest used by the guest owner with the KVM_SEV_LAUNCH_MEASURE.

If len is zero on entry, the measurement blob length is written to len and uaddr is unused.

Parameters (in): struct kvm_sev_attestation

Returns: 0 on success, -negative on error

```
struct kvm_sev_attestation_report {
        __u8 mnonce[16];        /* A random mnonce that will be placed in the␣
↪report */

        __u64 uaddr;            /* userspace address where the report should␣
↪be copied */
        __u32 len;
};
```

11. KVM_SEV_SEND_START

The KVM_SEV_SEND_START command can be used by the hypervisor to create an outgoing guest encryption context.

If session_len is zero on entry, the length of the guest session information is written to session_len and all other fields are not used.

Parameters (in): struct kvm_sev_send_start

Returns: 0 on success, -negative on error

```
struct kvm_sev_send_start {
        __u32 policy;                 /* guest policy */

        __u64 pdh_cert_uaddr;         /* platform Diffie-Hellman certificate */
        __u32 pdh_cert_len;

        __u64 plat_certs_uaddr;        /* platform certificate chain */
        __u32 plat_certs_len;

        __u64 amd_certs_uaddr;        /* AMD certificate */
        __u32 amd_certs_len;

        __u64 session_uaddr;          /* Guest session information */
        __u32 session_len;
};
```

12. KVM_SEV_SEND_UPDATE_DATA

The KVM_SEV_SEND_UPDATE_DATA command can be used by the hypervisor to encrypt the outgoing guest memory region with the encryption context creating using KVM_SEV_SEND_START.

If hdr_len or trans_len are zero on entry, the length of the packet header and transport region are written to hdr_len and trans_len respectively, and all other fields are not used.

Parameters (in): struct kvm_sev_send_update_data

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_send_update_data {
        __u64 hdr_uaddr;        /* userspace address containing the packet␣
↪header */
        __u32 hdr_len;

        __u64 guest_uaddr;      /* the source memory region to be encrypted */
        __u32 guest_len;

        __u64 trans_uaddr;      /* the destination memory region  */
        __u32 trans_len;
};
```

13. KVM_SEV_SEND_FINISH

After completion of the migration flow, the KVM_SEV_SEND_FINISH command can be issued by the hypervisor to delete the encryption context.

Returns: 0 on success, -negative on error

14. KVM_SEV_SEND_CANCEL

After completion of SEND_START, but before SEND_FINISH, the source VMM can issue the SEND_CANCEL command to stop a migration. This is necessary so that a cancelled migration can restart with a new target later.

Returns: 0 on success, -negative on error

15. KVM_SEV_RECEIVE_START

The KVM_SEV_RECEIVE_START command is used for creating the memory encryption context for an incoming SEV guest. To create the encryption context, the user must provide a guest policy, the platform public Diffie-Hellman (PDH) key and session information.

Parameters: struct kvm_sev_receive_start (in/out)

Returns: 0 on success, -negative on error

```
struct kvm_sev_receive_start {
        __u32 handle;           /* if zero then firmware creates a new handle␣
↪*/
        __u32 policy;           /* guest's policy */

        __u64 pdh_uaddr;        /* userspace address pointing to the PDH key */
        __u32 pdh_len;

        __u64 session_uaddr;    /* userspace address which points to the guest␣
↪session information */
        __u32 session_len;
};
```

On success, the 'handle' field contains a new handle and on error, a negative value.

For more details, see SEV spec Section 6.12.

16. KVM_SEV_RECEIVE_UPDATE_DATA

The KVM_SEV_RECEIVE_UPDATE_DATA command can be used by the hypervisor to copy the incoming buffers into the guest memory region with encryption context created during the KVM_SEV_RECEIVE_START.

Parameters (in): struct kvm_sev_receive_update_data

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_receive_update_data {
        __u64 hdr_uaddr;        /* userspace address containing the packet␣
↪header */
        __u32 hdr_len;

        __u64 guest_uaddr;      /* the destination guest memory region */
        __u32 guest_len;

        __u64 trans_uaddr;      /* the incoming buffer memory region  */
        __u32 trans_len;
};
```

17. KVM_SEV_RECEIVE_FINISH

After completion of the migration flow, the KVM_SEV_RECEIVE_FINISH command can be issued by the hypervisor to make the guest ready for execution.

Returns: 0 on success, -negative on error

References

See [white-paper], [api-spec], [amd-apm] and [kvm-forum] for more info.

1.6.2 KVM CPUID bits

Author
Glauber Costa <glommer@gmail.com>

A guest running on a kvm host, can check some of its features using cpuid. This is not always guaranteed to work, since userspace can mask-out some, or even all KVM-related cpuid features before launching a guest.

KVM cpuid functions are:

function: KVM_CPUID_SIGNATURE (0x40000000)

returns:

```
eax = 0x40000001
ebx = 0x4b4d564b
ecx = 0x564b4d56
edx = 0x4d
```

Note that this value in ebx, ecx and edx corresponds to the string ”KVMKVMKVM”. The value in eax corresponds to the maximum cpuid function present in this leaf, and will be updated if more functions are added in the future. Note also that old hosts set eax value to 0x0. This should be interpreted as if the value was 0x40000001. This function queries the presence of KVM cpuid leafs.

function: define KVM_CPUID_FEATURES (0x40000001)

returns:

```
ebx, ecx
eax = an OR'ed group of (1 << flag)
```

where `flag` is defined as below:

flag	value	meaning
KVM_FEATURE_CLOCKSOURCE	0	kvmclock available at msrs 0x11 and 0x12
KVM_FEATURE_NOP_IO_DELAY	1	not necessary to perform delays on PIO operations
KVM_FEATURE_MMU_OP	2	deprecated
KVM_FEATURE_CLOCKSOURCE2	3	kvmclock available at msrs 0x4b564d00 and 0x4b564d01
KVM_FEATURE_ASYNC_PF	4	async pf can be enabled by writing to msr 0x4b564d02
KVM_FEATURE_STEAL_TIME	5	steal time can be enabled by writing to msr 0x4b564d03
KVM_FEATURE_PV_EOI	6	paravirtualized end of interrupt handler can be enabled by writing to msr 0x4b564d04
KVM_FEATURE_PV_UNHALT	7	guest checks this feature bit before enabling paravirtualized spinlock support
KVM_FEATURE_PV_TLB_FLUSH	9	guest checks this feature bit before enabling paravirtualized tlb flush
KVM_FEATURE_ASYNC_PF_VMEXIT	10	paravirtualized async PF VM EXIT can be enabled by setting bit 2 when writing to msr 0x4b564d02
KVM_FEATURE_PV_SEND_IPI	11	guest checks this feature bit before enabling paravirtualized send IPIs
KVM_FEATURE_POLL_CONTROL	12	host-side polling on HLT can be disabled by writing to msr 0x4b564d05.
KVM_FEATURE_PV_SCHED_YIELD	13	guest checks this feature bit before using paravirtualized sched yield.
KVM_FEATURE_ASYNC_PF_INT	14	guest checks this feature bit before using the second async pf control msr 0x4b564d06 and async pf acknowledgment msr 0x4b564d07.
KVM_FEATURE_MSI_EXT_DEST_ID	15	guest checks this feature bit before using extended destination ID bits in MSI address bits 11-5.
KVM_FEATURE_HC_MAP_GPA_RANGE	16	guest checks this feature bit before using the map gpa range hypercall to notify the page state change
KVM_FEATURE_MIGRATION_CONTROL	17	guest checks this feature bit before using MSR_KVM_MIGRATION_CONTROL
KVM_FEATURE_CLOCKSOURCE_STABLE_BIT	24	host will warn if no guest-side per-cpu warps are expected in kvmclock

```
edx = an OR'ed group of (1 << flag)
```

Where `flag` here is defined as below:

flag	value	meaning
KVM_HINTS_REALTIME	0	guest checks this feature bit to determine that vCPUs are never preempted for an unlimited time allowing optimizations

1.6.3 Known limitations of CPU virtualization

Whenever perfect emulation of a CPU feature is impossible or too hard, KVM has to choose between not implementing the feature at all or introducing behavioral differences between virtual machines and bare metal systems.

This file documents some of the known limitations that KVM has in virtualizing CPU features.

x86

`KVM_GET_SUPPORTED_CPUID` issues

x87 features

Unlike most other CPUID feature bits, CPUID[EAX=7,ECX=0]:EBX[6] (FDP_EXCPTN_ONLY) and CPUID[EAX=7,ECX=0]:EBX]13] (ZERO_FCS_FDS) are clear if the features are present and set if the features are not present.

Clearing these bits in CPUID has no effect on the operation of the guest; if these bits are set on hardware, the features will not be present on any virtual machine that runs on that hardware.

Workaround: It is recommended to always set these bits in guest CPUID. Note however that any software (e.g `WIN87EM.DLL`) expecting these features to be present likely predates these CPUID feature bits, and therefore doesn't know to check for them anyway.

Nested virtualization features

TBD

x2APIC

When KVM_X2APIC_API_USE_32BIT_IDS is enabled, KVM activates a hack/quirk that allows sending events to a single vCPU using its x2APIC ID even if the target vCPU has legacy xAPIC enabled, e.g. to bring up hotplugged vCPUs via INIT-SIPI on VMs with > 255 vCPUs. A side effect of the quirk is that, if multiple vCPUs have the same physical APIC ID, KVM will deliver events targeting that APIC ID only to the vCPU with the lowest vCPU ID. If KVM_X2APIC_API_USE_32BIT_IDS is not enabled, KVM follows x86 architecture when processing interrupts (all vCPUs matching the target APIC ID receive the interrupt).

1.6.4 Linux KVM Hypercall

X86:
KVM Hypercalls have a three-byte sequence of either the vmcall or the vmmcall instruction. The hypervisor can replace it with instructions that are guaranteed to be supported.

Up to four arguments may be passed in rbx, rcx, rdx, and rsi respectively. The hypercall number should be placed in rax and the return value will be placed in rax. No other registers will be clobbered unless explicitly stated by the particular hypercall.

S390:
R2-R7 are used for parameters 1-6. In addition, R1 is used for hypercall number. The return value is written to R2.

S390 uses diagnose instruction as hypercall (0x500) along with hypercall number in R1.

For further information on the S390 diagnose call as supported by KVM, refer to *The s390 DIAGNOSE call on KVM*.

PowerPC:
It uses R3-R10 and hypercall number in R11. R4-R11 are used as output registers. Return value is placed in R3.

KVM hypercalls uses 4 byte opcode, that are patched with 'hypercall-instructions' property inside the device tree's /hypervisor node. For more information refer to *The PPC KVM paravirtual interface*

MIPS:
KVM hypercalls use the HYPCALL instruction with code 0 and the hypercall number in $2 (v0). Up to four arguments may be placed in $4-$7 (a0-a3) and the return value is placed in $2 (v0).

KVM Hypercalls Documentation

The template for each hypercall is: 1. Hypercall name. 2. Architecture(s) 3. Status (deprecated, obsolete, active) 4. Purpose

1. KVM_HC_VAPIC_POLL_IRQ

Architecture
x86

Status
active

Purpose
Trigger guest exit so that the host can check for pending interrupts on reentry.

2. KVM_HC_MMU_OP

Architecture
x86

Status
deprecated.

Purpose
Support MMU operations such as writing to PTE, flushing TLB, release PT.

3. KVM_HC_FEATURES

Architecture
PPC

Status
active

Purpose
Expose hypercall availability to the guest. On x86 platforms, cpuid used to enumerate which hypercalls are available. On PPC, either device tree based lookup (which is also what EPAPR dictates) OR KVM specific enumeration mechanism (which is this hypercall) can be used.

4. KVM_HC_PPC_MAP_MAGIC_PAGE

Architecture
PPC

Status
active

Purpose
To enable communication between the hypervisor and guest there is a shared page that contains parts of supervisor visible register state. The guest can map this shared page to access its supervisor register through memory using this hypercall.

5. KVM_HC_KICK_CPU

Architecture
x86

Status
active

Purpose
Hypercall used to wakeup a vcpu from HLT state

Usage example
A vcpu of a paravirtualized guest that is busywaiting in guest kernel mode for an event to occur (ex: a spinlock to become available) can execute HLT instruction

once it has busy-waited for more than a threshold time-interval. Execution of HLT instruction would cause the hypervisor to put the vcpu to sleep until occurrence of an appropriate event. Another vcpu of the same guest can wakeup the sleeping vcpu by issuing KVM_HC_KICK_CPU hypercall, specifying APIC ID (a1) of the vcpu to be woken up. An additional argument (a0) is used in the hypercall for future use.

6. KVM_HC_CLOCK_PAIRING

Architecture
x86

Status
active

Purpose
Hypercall used to synchronize host and guest clocks.

Usage:

a0: guest physical address where host copies "struct kvm_clock_offset" structure.

a1: clock_type, ATM only KVM_CLOCK_PAIRING_WALLCLOCK (0) is supported (corresponding to the host's CLOCK_REALTIME clock).

```
struct kvm_clock_pairing {
        __s64 sec;
        __s64 nsec;
        __u64 tsc;
        __u32 flags;
        __u32 pad[9];
};
```

Where:

- sec: seconds from clock_type clock.
- nsec: nanoseconds from clock_type clock.
- tsc: guest TSC value used to calculate sec/nsec pair
- flags: flags, unused (0) at the moment.

The hypercall lets a guest compute a precise timestamp across host and guest. The guest can use the returned TSC value to compute the CLOCK_REALTIME for its clock, at the same instant.

Returns KVM_EOPNOTSUPP if the host does not use TSC clocksource, or if clock type is different than KVM_CLOCK_PAIRING_WALLCLOCK.

6. KVM_HC_SEND_IPI

Architecture
x86

Status
active

Purpose
Send IPIs to multiple vCPUs.

- a0: lower part of the bitmap of destination APIC IDs
- a1: higher part of the bitmap of destination APIC IDs
- a2: the lowest APIC ID in bitmap
- a3: APIC ICR

The hypercall lets a guest send multicast IPIs, with at most 128 128 destinations per hypercall in 64-bit mode and 64 vCPUs per hypercall in 32-bit mode. The destinations are represented by a bitmap contained in the first two arguments (a0 and a1). Bit 0 of a0 corresponds to the APIC ID in the third argument (a2), bit 1 corresponds to the APIC ID a2+1, and so on.

Returns the number of CPUs to which the IPIs were delivered successfully.

7. KVM_HC_SCHED_YIELD

Architecture
x86

Status
active

Purpose
Hypercall used to yield if the IPI target vCPU is preempted

a0: destination APIC ID

Usage example
When sending a call-function IPI-many to vCPUs, yield if any of the IPI target vCPUs was preempted.

8. KVM_HC_MAP_GPA_RANGE

Architecture
x86

Status
active

Purpose
Request KVM to map a GPA range with the specified attributes.

a0: the guest physical address of the start page a1: the number of (4kb) pages (must be contiguous in GPA space) a2: attributes

Where 'attributes' :

- bits 3:0 - preferred page size encoding 0 = 4kb, 1 = 2mb, 2 = 1gb, etc...
- bit 4 - plaintext = 0, encrypted = 1
- bits 63:5 - reserved (must be zero)

Implementation note: this hypercall is implemented in userspace via the KVM_CAP_EXIT_HYPERCALL capability. Userspace must enable that capability before advertising KVM_FEATURE_HC_MAP_GPA_RANGE in the guest CPUID. In addition, if the guest supports KVM_FEATURE_MIGRATION_CONTROL, userspace must also set up an MSR filter to process writes to MSR_KVM_MIGRATION_CONTROL.

1.6.5 The x86 kvm shadow mmu

The mmu (in arch/x86/kvm, files mmu.[ch] and paging_tmpl.h) is responsible for presenting a standard x86 mmu to the guest, while translating guest physical addresses to host physical addresses.

The mmu code attempts to satisfy the following requirements:

- **correctness:**
  the guest should not be able to determine that it is running on an emulated mmu except for timing (we attempt to comply with the specification, not emulate the characteristics of a particular implementation such as tlb size)
- **security:**
  the guest must not be able to touch host memory not assigned to it
- **performance:**
  minimize the performance penalty imposed by the mmu
- **scaling:**
  need to scale to large memory and large vcpu guests
- **hardware:**
  support the full range of x86 virtualization hardware
- **integration:**
  Linux memory management code must be in control of guest memory so that swapping, page migration, page merging, transparent hugepages, and similar features work without change
- **dirty tracking:**
  report writes to guest memory to enable live migration and framebuffer-based displays
- **footprint:**
  keep the amount of pinned kernel memory low (most memory should be shrinkable)
- **reliability:**
  avoid multipage or GFP_ATOMIC allocations

Acronyms

pfn	host page frame number
hpa	host physical address
hva	host virtual address
gfn	guest frame number
gpa	guest physical address
gva	guest virtual address
ngpa	nested guest physical address
ngva	nested guest virtual address
pte	page table entry (used also to refer generically to paging structure entries)
gpte	guest pte (referring to gfns)
spte	shadow pte (referring to pfns)
tdp	two dimensional paging (vendor neutral term for NPT and EPT)

Virtual and real hardware supported

The mmu supports first-generation mmu hardware, which allows an atomic switch of the current paging mode and cr3 during guest entry, as well as two-dimensional paging (AMD's NPT and Intel's EPT). The emulated hardware it exposes is the traditional 2/3/4 level x86 mmu, with support for global pages, pae, pse, pse36, cr0.wp, and 1GB pages. Emulated hardware also able to expose NPT capable hardware on NPT capable hosts.

Translation

The primary job of the mmu is to program the processor's mmu to translate addresses for the guest. Different translations are required at different times:

- when guest paging is disabled, we translate guest physical addresses to host physical addresses (gpa->hpa)
- when guest paging is enabled, we translate guest virtual addresses, to guest physical addresses, to host physical addresses (gva->gpa->hpa)
- when the guest launches a guest of its own, we translate nested guest virtual addresses, to nested guest physical addresses, to guest physical addresses, to host physical addresses (ngva->ngpa->gpa->hpa)

The primary challenge is to encode between 1 and 3 translations into hardware that support only 1 (traditional) and 2 (tdp) translations. When the number of required translations matches the hardware, the mmu operates in direct mode; otherwise it operates in shadow mode (see below).

Memory

Guest memory (gpa) is part of the user address space of the process that is using kvm. Userspace defines the translation between guest addresses and user addresses (gpa->hva); note that two gpas may alias to the same hva, but not vice versa.

These hvas may be backed using any method available to the host: anonymous memory, file backed memory, and device memory. Memory might be paged by the host at any time.

Events

The mmu is driven by events, some from the guest, some from the host.

Guest generated events:

- writes to control registers (especially cr3)
- invlpg/invlpga instruction execution
- access to missing or protected translations

Host generated events:

- changes in the gpa->hpa translation (either through gpa->hva changes or through hva->hpa changes)
- memory pressure (the shrinker)

Shadow pages

The principal data structure is the shadow page, 'struct kvm_mmu_page'. A shadow page contains 512 sptes, which can be either leaf or nonleaf sptes. A shadow page may contain a mix of leaf and nonleaf sptes.

A nonleaf spte allows the hardware mmu to reach the leaf pages and is not related to a translation directly. It points to other shadow pages.

A leaf spte corresponds to either one or two translations encoded into one paging structure entry. These are always the lowest level of the translation stack, with optional higher level translations left to NPT/EPT. Leaf ptes point at guest pages.

The following table shows translations encoded by leaf ptes, with higher-level translations in parentheses:

Non-nested guests:

```
nonpaging:      gpa->hpa
paging:         gva->gpa->hpa
paging, tdp:    (gva->)gpa->hpa
```

Nested guests:

```
non-tdp:        ngva->gpa->hpa  (*)
tdp:            (ngva->)ngpa->gpa->hpa
```

```
(*) the guest hypervisor will encode the ngva->gpa translation into␣
↪its page
    tables if npt is not present
```

Shadow pages contain the following information:

role.level:
The level in the shadow paging hierarchy that this shadow page belongs to. 1=4k sptes, 2=2M sptes, 3=1G sptes, etc.

role.direct:
If set, leaf sptes reachable from this page are for a linear range. Examples include real mode translation, large guest pages backed by small host pages, and gpa->hpa translations when NPT or EPT is active. The linear range starts at (gfn << PAGE_SHIFT) and its size is determined by role.level (2MB for first level, 1GB for second level, 0.5TB for third level, 256TB for fourth level) If clear, this page corresponds to a guest page table denoted by the gfn field.

role.quadrant:
When role.has_4_byte_gpte=1, the guest uses 32-bit gptes while the host uses 64-bit sptes. That means a guest page table contains more ptes than the host, so multiple shadow pages are needed to shadow one guest page. For first-level shadow pages, role.quadrant can be 0 or 1 and denotes the first or second 512-gpte block in the guest page table. For second-level page tables, each 32-bit gpte is converted to two 64-bit sptes (since each first-level guest page is shadowed by two first-level shadow pages) so role.quadrant takes values in the range 0..3. Each quadrant maps 1GB virtual address space.

role.access:
Inherited guest access permissions from the parent ptes in the form uwx. Note execute permission is positive, not negative.

role.invalid:
The page is invalid and should not be used. It is a root page that is currently pinned (by a cpu hardware register pointing to it); once it is unpinned it will be destroyed.

role.has_4_byte_gpte:
Reflects the size of the guest PTE for which the page is valid, i.e. '0' if direct map or 64-bit gptes are in use, '1' if 32-bit gptes are in use.

role.efer_nx:
Contains the value of efer.nx for which the page is valid.

role.cr0_wp:
Contains the value of cr0.wp for which the page is valid.

role.smep_andnot_wp:
Contains the value of cr4.smep && !cr0.wp for which the page is valid (pages for which this is true are different from other pages; see the treatment of cr0.wp=0 below).

role.smap_andnot_wp:
Contains the value of cr4.smap && !cr0.wp for which the page is valid (pages for which this is true are different from other pages; see the treatment of cr0.wp=0 below).

role.smm:
Is 1 if the page is valid in system management mode. This field determines which of

the kvm_memslots array was used to build this shadow page; it is also used to go back from a struct kvm_mmu_page to a memslot, through the kvm_memslots_for_spte_role macro and __gfn_to_memslot.

role.ad_disabled:
Is 1 if the MMU instance cannot use A/D bits. EPT did not have A/D bits before Haswell; shadow EPT page tables also cannot use A/D bits if the L1 hypervisor does not enable them.

role.guest_mode:
Indicates the shadow page is created for a nested guest.

role.passthrough:
The page is not backed by a guest page table, but its first entry points to one. This is set if NPT uses 5-level page tables (host CR4.LA57=1) and is shadowing L1's 4-level NPT (L1 CR4.LA57=0).

mmu_valid_gen:
The MMU generation of this page, used to fast zap of all MMU pages within a VM without blocking vCPUs too long. Specifically, KVM updates the per-VM valid MMU generation which causes the mismatch of mmu_valid_gen for each mmu page. This makes all existing MMU pages obsolete. Obsolete pages can't be used. Therefore, vCPUs must load a new, valid root before re-entering the guest. The MMU generation is only ever '0' or '1'. Note, the TDP MMU doesn't use this field as non-root TDP MMU pages are reachable only from their owning root. Thus it suffices for TDP MMU to use role.invalid in root pages to invalidate all MMU pages.

gfn:
Either the guest page table containing the translations shadowed by this page, or the base page frame for linear translations. See role.direct.

spt:
A pageful of 64-bit sptes containing the translations for this page. Accessed by both kvm and hardware. The page pointed to by spt will have its page->private pointing back at the shadow page structure. sptes in spt point either at guest pages, or at lower-level shadow pages. Specifically, if sp1 and sp2 are shadow pages, then sp1->spt[n] may point at __pa(sp2->spt). sp2 will point back at sp1 through parent_pte. The spt array forms a DAG structure with the shadow page as a node, and guest pages as leaves.

shadowed_translation:
An array of 512 shadow translation entries, one for each present pte. Used to perform a reverse map from a pte to a gfn as well as its access permission. When role.direct is set, the shadow_translation array is not allocated. This is because the gfn contained in any element of this array can be calculated from the gfn field when used. In addition, when role.direct is set, KVM does not track access permission for each of the gfn. See role.direct and gfn.

root_count / tdp_mmu_root_count:
root_count is a reference counter for root shadow pages in Shadow MMU. vCPUs elevate the refcount when getting a shadow page that will be used as a root page, i.e. page that will be loaded into hardware directly (CR3, PDPTRs, nCR3 EPTP). Root pages cannot be destroyed while their refcount is non-zero. See role.invalid. tdp_mmu_root_count is similar but exclusively used in TDP MMU as an atomic ref-count.

parent_ptes:
The reverse mapping for the pte/ptes pointing at this page's spt. If parent_ptes bit 0 is zero, only one spte points at this page and parent_ptes points at this single spte, otherwise, there exists multiple sptes pointing at this page and (parent_ptes & ~0x1) points at a data structure with a list of parent sptes.

ptep:
The kernel virtual address of the SPTE that points at this shadow page. Used exclusively by the TDP MMU, this field is a union with parent_ptes.

unsync:
If true, then the translations in this page may not match the guest's translation. This is equivalent to the state of the tlb when a pte is changed but before the tlb entry is flushed. Accordingly, unsync ptes are synchronized when the guest executes invlpg or flushes its tlb by other means. Valid for leaf pages.

unsync_children:
How many sptes in the page point at pages that are unsync (or have unsynchronized children).

unsync_child_bitmap:
A bitmap indicating which sptes in spt point (directly or indirectly) at pages that may be unsynchronized. Used to quickly locate all unsynchronized pages reachable from a given page.

clear_spte_count:
Only present on 32-bit hosts, where a 64-bit spte cannot be written atomically. The reader uses this while running out of the MMU lock to detect in-progress updates and retry them until the writer has finished the write.

write_flooding_count:
A guest may write to a page table many times, causing a lot of emulations if the page needs to be write-protected (see ”Synchronized and unsynchronized pages” below). Leaf pages can be unsynchronized so that they do not trigger frequent emulation, but this is not possible for non-leafs. This field counts the number of emulations since the last time the page table was actually used; if emulation is triggered too frequently on this page, KVM will unmap the page to avoid emulation in the future.

tdp_mmu_page:
Is 1 if the shadow page is a TDP MMU page. This variable is used to bifurcate the control flows for KVM when walking any data structure that may contain pages from both TDP MMU and shadow MMU.

Reverse map

The mmu maintains a reverse mapping whereby all ptes mapping a page can be reached given its gfn. This is used, for example, when swapping out a page.

Synchronized and unsynchronized pages

The guest uses two events to synchronize its tlb and page tables: tlb flushes and page invalidations (invlpg).

A tlb flush means that we need to synchronize all sptes reachable from the guest's cr3. This is expensive, so we keep all guest page tables write protected, and synchronize sptes to gptes when a gpte is written.

A special case is when a guest page table is reachable from the current guest cr3. In this case, the guest is obliged to issue an invlpg instruction before using the translation. We take advantage of that by removing write protection from the guest page, and allowing the guest to modify it freely. We synchronize modified gptes when the guest invokes invlpg. This reduces the amount of emulation we have to do when the guest modifies multiple gptes, or when the a guest page is no longer used as a page table and is used for random guest data.

As a side effect we have to resynchronize all reachable unsynchronized shadow pages on a tlb flush.

Reaction to events

- guest page fault (or npt page fault, or ept violation)

This is the most complicated event. The cause of a page fault can be:

  - a true guest fault (the guest translation won't allow the access) (*)
  - access to a missing translation
  - access to a protected translation - when logging dirty pages, memory is write protected - synchronized shadow pages are write protected (*)
  - access to untranslatable memory (mmio)

  (*) not applicable in direct mode

Handling a page fault is performed as follows:

- if the RSV bit of the error code is set, the page fault is caused by guest accessing MMIO and cached MMIO information is available.
  - walk shadow page table
  - check for valid generation number in the spte (see ”Fast invalidation of MMIO sptes” below)
  - cache the information to vcpu->arch.mmio_gva, vcpu->arch.mmio_access and vcpu->arch.mmio_gfn, and call the emulator
- If both P bit and R/W bit of error code are set, this could possibly be handled as a ”fast page fault” (fixed without taking the MMU lock). See the description in *KVM Lock Overview*.
- if needed, walk the guest page tables to determine the guest translation (gva->gpa or ngpa->gpa)
  - if permissions are insufficient, reflect the fault back to the guest
- determine the host page

  - if this is an mmio request, there is no host page; cache the info to vcpu->arch.mmio_gva, vcpu->arch.mmio_access and vcpu->arch.mmio_gfn

- walk the shadow page table to find the spte for the translation, instantiating missing intermediate page tables as necessary
  - If this is an mmio request, cache the mmio info to the spte and set some reserved bit on the spte (see callers of kvm_mmu_set_mmio_spte_mask)
- try to unsynchronize the page
  - if successful, we can let the guest continue and modify the gpte
- emulate the instruction
  - if failed, unshadow the page and let the guest continue
- update any translations that were modified by the instruction

invlpg handling:

- walk the shadow page hierarchy and drop affected translations
- try to reinstantiate the indicated translation in the hope that the guest will use it in the near future

Guest control register updates:

- mov to cr3
  - look up new shadow roots
  - synchronize newly reachable shadow pages
- mov to cr0/cr4/efer
  - set up mmu context for new paging mode
  - look up new shadow roots
  - synchronize newly reachable shadow pages

Host translation updates:

- mmu notifier called with updated hva
- look up affected sptes through reverse map
- drop (or update) translations

Emulating cr0.wp

If tdp is not enabled, the host must keep cr0.wp=1 so page write protection works for the guest kernel, not guest userspace. When the guest cr0.wp=1, this does not present a problem. However when the guest cr0.wp=0, we cannot map the permissions for gpte.u=1, gpte.w=0 to any spte (the semantics require allowing any guest kernel access plus user read access).

We handle this by mapping the permissions to two possible sptes, depending on fault type:

- kernel write fault: spte.u=0, spte.w=1 (allows full kernel access, disallows user access)
- read fault: spte.u=1, spte.w=0 (allows full read access, disallows kernel write access)

(user write faults generate a #PF)

In the first case there are two additional complications:

- if CR4.SMEP is enabled: since we've turned the page into a kernel page, the kernel may now execute it. We handle this by also setting spte.nx. If we get a user fetch or read fault, we'll change spte.u=1 and spte.nx=gpte.nx back. For this to work, KVM forces EFER.NX to 1 when shadow paging is in use.
- if CR4.SMAP is disabled: since the page has been changed to a kernel page, it can not be reused when CR4.SMAP is enabled. We set CR4.SMAP && !CR0.WP into shadow page's role to avoid this case. Note, here we do not care the case that CR4.SMAP is enabled since KVM will directly inject #PF to guest due to failed permission check.

To prevent an spte that was converted into a kernel page with cr0.wp=0 from being written by the kernel after cr0.wp has changed to 1, we make the value of cr0.wp part of the page role. This means that an spte created with one value of cr0.wp cannot be used when cr0.wp has a different value - it will simply be missed by the shadow page lookup code. A similar issue exists when an spte created with cr0.wp=0 and cr4.smep=0 is used after changing cr4.smep to 1. To avoid this, the value of !cr0.wp && cr4.smep is also made a part of the page role.

Large pages

The mmu supports all combinations of large and small guest and host pages. Supported page sizes include 4k, 2M, 4M, and 1G. 4M pages are treated as two separate 2M pages, on both guest and host, since the mmu always uses PAE paging.

To instantiate a large spte, four constraints must be satisfied:

- the spte must point to a large host page
- the guest pte must be a large pte of at least equivalent size (if tdp is enabled, there is no guest pte and this condition is satisfied)
- if the spte will be writeable, the large page frame may not overlap any write-protected pages
- the guest page must be wholly contained by a single memory slot

To check the last two conditions, the mmu maintains a ->disallow_lpage set of arrays for each memory slot and large page size. Every write protected page causes its disallow_lpage to be incremented, thus preventing instantiation of a large spte. The frames at the end of an unaligned memory slot have artificially inflated ->disallow_lpages so they can never be instantiated.

Fast invalidation of MMIO sptes

As mentioned in ”Reaction to events” above, kvm will cache MMIO information in leaf sptes. When a new memslot is added or an existing memslot is changed, this information may become stale and needs to be invalidated. This also needs to hold the MMU lock while walking all shadow pages, and is made more scalable with a similar technique.

MMIO sptes have a few spare bits, which are used to store a generation number. The global generation number is stored in kvm_memslots(kvm)->generation, and increased whenever guest memory info changes.

When KVM finds an MMIO spte, it checks the generation number of the spte. If the generation number of the spte does not equal the global generation number, it will ignore the cached MMIO information and handle the page fault through the slow path.

Since only 18 bits are used to store generation-number on mmio spte, all pages are zapped when there is an overflow.

Unfortunately, a single memory access might access kvm_memslots(kvm) multiple times, the last one happening when the generation number is retrieved and stored into the MMIO spte. Thus, the MMIO spte might be created based on out-of-date information, but with an up-to-date generation number.

To avoid this, the generation number is incremented again after synchronize_srcu returns; thus, bit 63 of kvm_memslots(kvm)->generation set to 1 only during a memslot update, while some SRCU readers might be using the old copy. We do not want to use an MMIO sptes created with an odd generation number, and we can do this without losing a bit in the MMIO spte. The "update in-progress" bit of the generation is not stored in MMIO spte, and is so is implicitly zero when the generation is extracted out of the spte. If KVM is unlucky and creates an MMIO spte while an update is in-progress, the next access to the spte will always be a cache miss. For example, a subsequent access during the update window will miss due to the in-progress flag diverging, while an access after the update window closes will have a higher generation number (as compared to the spte).

Further reading

- NPT presentation from KVM Forum 2008 https://www.linux-kvm.org/images/c/c8/KvmForum2008%24kdf2008_21.pdf

1.6.6 KVM-specific MSRs

Author
Glauber Costa <glommer@redhat.com>, Red Hat Inc, 2010

KVM makes use of some custom MSRs to service some requests.

Custom MSRs have a range reserved for them, that goes from 0x4b564d00 to 0x4b564dff. There are MSRs outside this area, but they are deprecated and their use is discouraged.

Custom MSR list

The current supported Custom MSR list is:

MSR_KVM_WALL_CLOCK_NEW:
0x4b564d00

data:
4-byte alignment physical address of a memory area which must be in guest RAM. This memory is expected to hold a copy of the following structure:

```
struct pvclock_wall_clock {
       u32   version;
       u32   sec;
```

```
      u32   nsec;
} __attribute__((__packed__));
```

whose data will be filled in by the hypervisor. The hypervisor is only guaranteed to update this data at the moment of MSR write. Users that want to reliably query this information more than once have to write more than once to this MSR. Fields have the following meanings:

version:
guest has to check version before and after grabbing time information and check that they are both equal and even. An odd version indicates an in-progress update.

sec:
number of seconds for wallclock at time of boot.

nsec:
number of nanoseconds for wallclock at time of boot.

In order to get the current wallclock time, the system_time from MSR_KVM_SYSTEM_TIME_NEW needs to be added.

Note that although MSRs are per-CPU entities, the effect of this particular MSR is global.

Availability of this MSR must be checked via bit 3 in 0x4000001 cpuid leaf prior to usage.

MSR_KVM_SYSTEM_TIME_NEW:
0x4b564d01

data:
4-byte aligned physical address of a memory area which must be in guest RAM, plus an enable bit in bit 0. This memory is expected to hold a copy of the following structure:

```
struct pvclock_vcpu_time_info {
      u32   version;
      u32   pad0;
      u64   tsc_timestamp;
      u64   system_time;
      u32   tsc_to_system_mul;
      s8    tsc_shift;
      u8    flags;
      u8    pad[2];
} __attribute__((__packed__)); /* 32 bytes */
```

whose data will be filled in by the hypervisor periodically. Only one write, or registration, is needed for each VCPU. The interval between updates of this structure is arbitrary and implementation-dependent. The hypervisor may update this structure at any time it sees fit until anything with bit0 == 0 is written to it.

Fields have the following meanings:

version:
guest has to check version before and after grabbing time information and check that they are both equal and even. An odd version indicates an in-progress update.

tsc_timestamp:
the tsc value at the current VCPU at the time of the update of this structure. Guests

can subtract this value from current tsc to derive a notion of elapsed time since the structure update.

system_time:
a host notion of monotonic time, including sleep time at the time this structure was last updated. Unit is nanoseconds.

tsc_to_system_mul:
multiplier to be used when converting tsc-related quantity to nanoseconds

tsc_shift:
shift to be used when converting tsc-related quantity to nanoseconds. This shift will ensure that multiplication with tsc_to_system_mul does not overflow. A positive value denotes a left shift, a negative value a right shift.

The conversion from tsc to nanoseconds involves an additional right shift by 32 bits. With this information, guests can derive per-CPU time by doing:

```
time = (current_tsc - tsc_timestamp)
if (tsc_shift >= 0)
        time <<= tsc_shift;
else
        time >>= -tsc_shift;
time = (time * tsc_to_system_mul) >> 32
time = time + system_time
```

flags:
bits in this field indicate extended capabilities coordinated between the guest and the hypervisor. Availability of specific flags has to be checked in 0x40000001 cpuid leaf. Current flags are:

flag bit	cpuid bit	meaning
0	24	time measures taken across multiple cpus are guaranteed to be monotonic
1	N/A	guest vcpu has been paused by the host See 4.70 in api.txt

Availability of this MSR must be checked via bit 3 in 0x4000001 cpuid leaf prior to usage.

MSR_KVM_WALL_CLOCK:
0x11

data and functioning:
same as MSR_KVM_WALL_CLOCK_NEW. Use that instead.

This MSR falls outside the reserved KVM range and may be removed in the future. Its usage is deprecated.

Availability of this MSR must be checked via bit 0 in 0x4000001 cpuid leaf prior to usage.

MSR_KVM_SYSTEM_TIME:
0x12

data and functioning:
same as MSR_KVM_SYSTEM_TIME_NEW. Use that instead.

This MSR falls outside the reserved KVM range and may be removed in the future. Its usage is deprecated.

Availability of this MSR must be checked via bit 0 in 0x4000001 cpuid leaf prior to usage.

The suggested algorithm for detecting kvmclock presence is then:

```
if (!kvm_para_available())    /* refer to cpuid.txt */
        return NON_PRESENT;

flags = cpuid_eax(0x40000001);
if (flags & 3) {
        msr_kvm_system_time = MSR_KVM_SYSTEM_TIME_NEW;
        msr_kvm_wall_clock = MSR_KVM_WALL_CLOCK_NEW;
        return PRESENT;
} else if (flags & 0) {
        msr_kvm_system_time = MSR_KVM_SYSTEM_TIME;
        msr_kvm_wall_clock = MSR_KVM_WALL_CLOCK;
        return PRESENT;
} else
        return NON_PRESENT;
```

MSR_KVM_ASYNC_PF_EN:
0x4b564d02

data:
Asynchronous page fault (APF) control MSR.

Bits 63-6 hold 64-byte aligned physical address of a 64 byte memory area which must be in guest RAM and must be zeroed. This memory is expected to hold a copy of the following structure:

```
struct kvm_vcpu_pv_apf_data {
      /* Used for 'page not present' events delivered via #PF */
      __u32 flags;

      /* Used for 'page ready' events delivered via interrupt notification␣
↪*/
      __u32 token;

      __u8 pad[56];
      __u32 enabled;
};
```

Bits 5-4 of the MSR are reserved and should be zero. Bit 0 is set to 1 when asynchronous page faults are enabled on the vcpu, 0 when disabled. Bit 1 is 1 if asynchronous page faults can be injected when vcpu is in cpl == 0. Bit 2 is 1 if asynchronous page faults are delivered to L1 as #PF vmexits. Bit 2 can be set only if KVM_FEATURE_ASYNC_PF_VMEXIT is present in CPUID. Bit 3 enables interrupt based delivery of 'page ready' events. Bit 3 can only be set if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

'Page not present' events are currently always delivered as synthetic #PF exception. During delivery of these events APF CR2 register contains a token that will be used to notify the guest when missing page becomes available. Also, to make it possible to distinguish between real #PF and APF, first 4 bytes of 64 byte memory location ('flags') will be written to by the hypervisor at the time of injection. Only first bit of 'flags' is currently supported, when set, it indicates that the guest is dealing with asynchronous 'page not present' event.

If during a page fault APF 'flags' is '0' it means that this is regular page fault. Guest is supposed to clear 'flags' when it is done handling #PF exception so the next event can be delivered.

Note, since APF 'page not present' events use the same exception vector as regular page fault, guest must reset 'flags' to '0' before it does something that can generate normal page fault.

Bytes 5-7 of 64 byte memory location ('token') will be written to by the hypervisor at the time of APF 'page ready' event injection. The content of these bytes is a token which was previously delivered as 'page not present' event. The event indicates the page in now available. Guest is supposed to write '0' to 'token' when it is done handling 'page ready' event and to write 1' to MSR_KVM_ASYNC_PF_ACK after clearing the location; writing to the MSR forces KVM to re-scan its queue and deliver the next pending notification.

Note, MSR_KVM_ASYNC_PF_INT MSR specifying the interrupt vector for 'page ready' APF delivery needs to be written to before enabling APF mechanism in MSR_KVM_ASYNC_PF_EN or interrupt #0 can get injected. The MSR is available if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

Note, previously, 'page ready' events were delivered via the same #PF exception as 'page not present' events but this is now deprecated. If bit 3 (interrupt based delivery) is not set APF events are not delivered.

If APF is disabled while there are outstanding APFs, they will not be delivered.

Currently 'page ready' APF events will be always delivered on the same vcpu as 'page not present' event was, but guest should not rely on that.

MSR_KVM_STEAL_TIME:
0x4b564d03

data:
64-byte alignment physical address of a memory area which must be in guest RAM, plus an enable bit in bit 0. This memory is expected to hold a copy of the following structure:

```
struct kvm_steal_time {
      __u64 steal;
      __u32 version;
      __u32 flags;
      __u8  preempted;
      __u8  u8_pad[3];
      __u32 pad[11];
}
```

whose data will be filled in by the hypervisor periodically. Only one write, or registration, is needed for each VCPU. The interval between updates of this structure is arbitrary and implementation-dependent. The hypervisor may update this structure at any time it sees fit until anything with bit0 == 0 is written to it. Guest is required to make sure this structure is initialized to zero.

Fields have the following meanings:

version:
a sequence counter. In other words, guest has to check this field before and after grabbing time information and make sure they are both equal and even. An odd version

indicates an in-progress update.

flags:
At this point, always zero. May be used to indicate changes in this structure in the future.

steal:
the amount of time in which this vCPU did not run, in nanoseconds. Time during which the vcpu is idle, will not be reported as steal time.

preempted:
indicate the vCPU who owns this struct is running or not. Non-zero values mean the vCPU has been preempted. Zero means the vCPU is not preempted. NOTE, it is always zero if the the hypervisor doesn't support this field.

MSR_KVM_EOI_EN:
0x4b564d04

data:
Bit 0 is 1 when PV end of interrupt is enabled on the vcpu; 0 when disabled. Bit 1 is reserved and must be zero. When PV end of interrupt is enabled (bit 0 set), bits 63-2 hold a 4-byte aligned physical address of a 4 byte memory area which must be in guest RAM and must be zeroed.

The first, least significant bit of 4 byte memory location will be written to by the hypervisor, typically at the time of interrupt injection. Value of 1 means that guest can skip writing EOI to the apic (using MSR or MMIO write); instead, it is sufficient to signal EOI by clearing the bit in guest memory - this location will later be polled by the hypervisor. Value of 0 means that the EOI write is required.

It is always safe for the guest to ignore the optimization and perform the APIC EOI write anyway.

Hypervisor is guaranteed to only modify this least significant bit while in the current VCPU context, this means that guest does not need to use either lock prefix or memory ordering primitives to synchronise with the hypervisor.

However, hypervisor can set and clear this memory bit at any time: therefore to make sure hypervisor does not interrupt the guest and clear the least significant bit in the memory area in the window between guest testing it to detect whether it can skip EOI apic write and between guest clearing it to signal EOI to the hypervisor, guest must both read the least significant bit in the memory area and clear it using a single CPU instruction, such as test and clear, or compare and exchange.

MSR_KVM_POLL_CONTROL:
0x4b564d05

Control host-side polling.

data:
Bit 0 enables (1) or disables (0) host-side HLT polling logic.

KVM guests can request the host not to poll on HLT, for example if they are performing polling themselves.

MSR_KVM_ASYNC_PF_INT:
0x4b564d06

data:
Second asynchronous page fault (APF) control MSR.

Bits 0-7: APIC vector for delivery of 'page ready' APF events. Bits 8-63: Reserved

Interrupt vector for asynchnonous 'page ready' notifications delivery. The vector has to be set up before asynchronous page fault mechanism is enabled in MSR_KVM_ASYNC_PF_EN. The MSR is only available if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

MSR_KVM_ASYNC_PF_ACK:
0x4b564d07

data:
Asynchronous page fault (APF) acknowledgment.

When the guest is done processing 'page ready' APF event and 'token' field in 'struct kvm_vcpu_pv_apf_data' is cleared it is supposed to write '1' to bit 0 of the MSR, this causes the host to re-scan its queue and check if there are more notifications pending. The MSR is available if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

MSR_KVM_MIGRATION_CONTROL:
0x4b564d08

data:
This MSR is available if KVM_FEATURE_MIGRATION_CONTROL is present in CPUID. Bit 0 represents whether live migration of the guest is allowed.

When a guest is started, bit 0 will be 0 if the guest has encrypted memory and 1 if the guest does not have encrypted memory. If the guest is communicating page encryption status to the host using the `KVM_HC_MAP_GPA_RANGE` hypercall, it can set bit 0 in this MSR to allow live migration of the guest.

1.6.7 Nested VMX

Overview

On Intel processors, KVM uses Intel's VMX (Virtual-Machine eXtensions) to easily and efficiently run guest operating systems. Normally, these guests *cannot* themselves be hypervisors running their own guests, because in VMX, guests cannot use VMX instructions.

The ”Nested VMX” feature adds this missing capability - of running guest hypervisors (which use VMX) with their own nested guests. It does so by allowing a guest to use VMX instructions, and correctly and efficiently emulating them using the single level of VMX available in the hardware.

We describe in much greater detail the theory behind the nested VMX feature, its implementation and its performance characteristics, in the OSDI 2010 paper ”The Turtles Project: Design and Implementation of Nested Virtualization”, available at:

https://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf

Terminology

Single-level virtualization has two levels - the host (KVM) and the guests. In nested virtualization, we have three levels: The host (KVM), which we call L0, the guest hypervisor, which we call L1, and its nested guest, which we call L2.

Running nested VMX

The nested VMX feature is enabled by default since Linux kernel v4.20. For older Linux kernel, it can be enabled by giving the ”nested=1” option to the kvm-intel module.

No modifications are required to user space (qemu). However, qemu's default emulated CPU type (qemu64) does not list the ”VMX” CPU feature, so it must be explicitly enabled, by giving qemu one of the following options:

- cpu host (emulated CPU has all features of the real CPU)
- cpu qemu64,+vmx (add just the vmx feature to a named CPU type)

ABIs

Nested VMX aims to present a standard and (eventually) fully-functional VMX implementation for the a guest hypervisor to use. As such, the official specification of the ABI that it provides is Intel's VMX specification, namely volume 3B of their ”Intel 64 and IA-32 Architectures Software Developer's Manual”. Not all of VMX's features are currently fully supported, but the goal is to eventually support them all, starting with the VMX features which are used in practice by popular hypervisors (KVM and others).

As a VMX implementation, nested VMX presents a VMCS structure to L1. As mandated by the spec, other than the two fields revision_id and abort, this structure is *opaque* to its user, who is not supposed to know or care about its internal structure. Rather, the structure is accessed through the VMREAD and VMWRITE instructions. Still, for debugging purposes, KVM developers might be interested to know the internals of this structure; This is struct vmcs12 from arch/x86/kvm/vmx.c.

The name ”vmcs12” refers to the VMCS that L1 builds for L2. In the code we also have ”vmcs01”, the VMCS that L0 built for L1, and ”vmcs02” is the VMCS which L0 builds to actually run L2 - how this is done is explained in the aforementioned paper.

For convenience, we repeat the content of struct vmcs12 here. If the internals of this structure changes, this can break live migration across KVM versions. VMCS12_REVISION (from vmx.c) should be changed if struct vmcs12 or its inner struct shadow_vmcs is ever changed.

```
typedef u64 natural_width;
struct __packed vmcs12 {
        /* According to the Intel spec, a VMCS region must start with
         * these two user-visible fields */
        u32 revision_id;
        u32 abort;

        u32 launch_state; /* set to 0 by VMCLEAR, to 1 by VMLAUNCH */
        u32 padding[7]; /* room for future expansion */
```

```
        u64 io_bitmap_a;
        u64 io_bitmap_b;
        u64 msr_bitmap;
        u64 vm_exit_msr_store_addr;
        u64 vm_exit_msr_load_addr;
        u64 vm_entry_msr_load_addr;
        u64 tsc_offset;
        u64 virtual_apic_page_addr;
        u64 apic_access_addr;
        u64 ept_pointer;
        u64 guest_physical_address;
        u64 vmcs_link_pointer;
        u64 guest_ia32_debugctl;
        u64 guest_ia32_pat;
        u64 guest_ia32_efer;
        u64 guest_pdptr0;
        u64 guest_pdptr1;
        u64 guest_pdptr2;
        u64 guest_pdptr3;
        u64 host_ia32_pat;
        u64 host_ia32_efer;
        u64 padding64[8]; /* room for future expansion */
        natural_width cr0_guest_host_mask;
        natural_width cr4_guest_host_mask;
        natural_width cr0_read_shadow;
        natural_width cr4_read_shadow;
        natural_width dead_space[4]; /* Last remnants of cr3_target_value[0-3].
↪ */
        natural_width exit_qualification;
        natural_width guest_linear_address;
        natural_width guest_cr0;
        natural_width guest_cr3;
        natural_width guest_cr4;
        natural_width guest_es_base;
        natural_width guest_cs_base;
        natural_width guest_ss_base;
        natural_width guest_ds_base;
        natural_width guest_fs_base;
        natural_width guest_gs_base;
        natural_width guest_ldtr_base;
        natural_width guest_tr_base;
        natural_width guest_gdtr_base;
        natural_width guest_idtr_base;
        natural_width guest_dr7;
        natural_width guest_rsp;
        natural_width guest_rip;
        natural_width guest_rflags;
        natural_width guest_pending_dbg_exceptions;
        natural_width guest_sysenter_esp;
        natural_width guest_sysenter_eip;
```

```
        natural_width host_cr0;
        natural_width host_cr3;
        natural_width host_cr4;
        natural_width host_fs_base;
        natural_width host_gs_base;
        natural_width host_tr_base;
        natural_width host_gdtr_base;
        natural_width host_idtr_base;
        natural_width host_ia32_sysenter_esp;
        natural_width host_ia32_sysenter_eip;
        natural_width host_rsp;
        natural_width host_rip;
        natural_width paddingl[8]; /* room for future expansion */
        u32 pin_based_vm_exec_control;
        u32 cpu_based_vm_exec_control;
        u32 exception_bitmap;
        u32 page_fault_error_code_mask;
        u32 page_fault_error_code_match;
        u32 cr3_target_count;
        u32 vm_exit_controls;
        u32 vm_exit_msr_store_count;
        u32 vm_exit_msr_load_count;
        u32 vm_entry_controls;
        u32 vm_entry_msr_load_count;
        u32 vm_entry_intr_info_field;
        u32 vm_entry_exception_error_code;
        u32 vm_entry_instruction_len;
        u32 tpr_threshold;
        u32 secondary_vm_exec_control;
        u32 vm_instruction_error;
        u32 vm_exit_reason;
        u32 vm_exit_intr_info;
        u32 vm_exit_intr_error_code;
        u32 idt_vectoring_info_field;
        u32 idt_vectoring_error_code;
        u32 vm_exit_instruction_len;
        u32 vmx_instruction_info;
        u32 guest_es_limit;
        u32 guest_cs_limit;
        u32 guest_ss_limit;
        u32 guest_ds_limit;
        u32 guest_fs_limit;
        u32 guest_gs_limit;
        u32 guest_ldtr_limit;
        u32 guest_tr_limit;
        u32 guest_gdtr_limit;
        u32 guest_idtr_limit;
        u32 guest_es_ar_bytes;
        u32 guest_cs_ar_bytes;
        u32 guest_ss_ar_bytes;
```

```
        u32 guest_ds_ar_bytes;
        u32 guest_fs_ar_bytes;
        u32 guest_gs_ar_bytes;
        u32 guest_ldtr_ar_bytes;
        u32 guest_tr_ar_bytes;
        u32 guest_interruptibility_info;
        u32 guest_activity_state;
        u32 guest_sysenter_cs;
        u32 host_ia32_sysenter_cs;
        u32 padding32[8]; /* room for future expansion */
        u16 virtual_processor_id;
        u16 guest_es_selector;
        u16 guest_cs_selector;
        u16 guest_ss_selector;
        u16 guest_ds_selector;
        u16 guest_fs_selector;
        u16 guest_gs_selector;
        u16 guest_ldtr_selector;
        u16 guest_tr_selector;
        u16 host_es_selector;
        u16 host_cs_selector;
        u16 host_ss_selector;
        u16 host_ds_selector;
        u16 host_fs_selector;
        u16 host_gs_selector;
        u16 host_tr_selector;
};
```

Authors

These patches were written by:

- Abel Gordon, abelg <at> il.ibm.com
- Nadav Har'El, nyh <at> il.ibm.com
- Orit Wasserman, oritw <at> il.ibm.com
- Ben-Ami Yassor, benami <at> il.ibm.com
- Muli Ben-Yehuda, muli <at> il.ibm.com

With contributions by:

- Anthony Liguori, aliguori <at> us.ibm.com
- Mike Day, mdday <at> us.ibm.com
- Michael Factor, factor <at> il.ibm.com
- Zvi Dubitzky, dubi <at> il.ibm.com

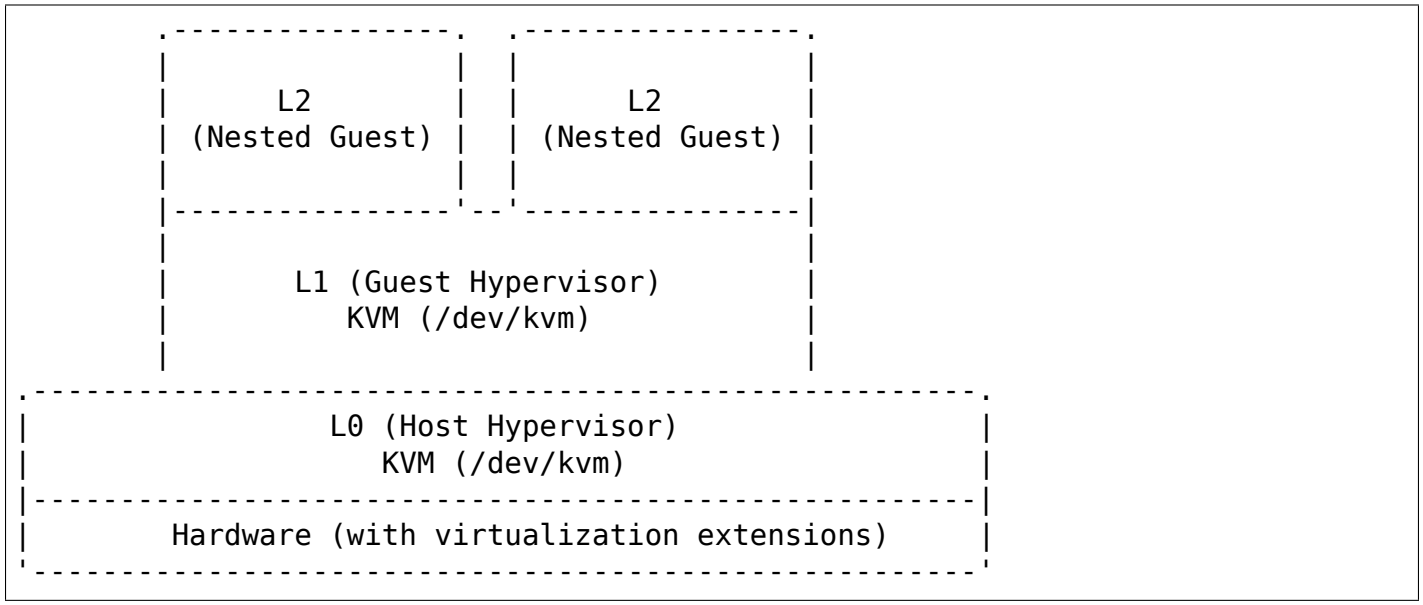
And valuable reviews by:

- Avi Kivity, avi <at> redhat.com

- Gleb Natapov, gleb <at> redhat.com
- Marcelo Tosatti, mtosatti <at> redhat.com
- Kevin Tian, kevin.tian <at> intel.com
- and others.

1.6.8 Running nested guests with KVM

A nested guest is the ability to run a guest inside another guest (it can be KVM-based or a different hypervisor). The straightforward example is a KVM guest that in turn runs on a KVM guest (the rest of this document is built on this example):



Terminology:

- L0 – level-0; the bare metal host, running KVM
- L1 – level-1 guest; a VM running on L0; also called the ”guest hypervisor”, as it itself is capable of running KVM.
- L2 – level-2 guest; a VM running on L1, this is the ”nested guest”

Note: The above diagram is modelled after the x86 architecture; s390x, ppc64 and other architectures are likely to have a different design for nesting.

For example, s390x always has an LPAR (LogicalPARtition) hypervisor running on bare metal, adding another layer and resulting in at least four levels in a nested setup — L0 (bare metal, running the LPAR hypervisor), L1 (host hypervisor), L2 (guest hypervisor), L3 (nested guest).

This document will stick with the three-level terminology (L0, L1, and L2) for all architectures; and will largely focus on x86.

Use Cases

There are several scenarios where nested KVM can be useful, to name a few:

- As a developer, you want to test your software on different operating systems (OSes). Instead of renting multiple VMs from a Cloud Provider, using nested KVM lets you rent a large enough "guest hypervisor" (level-1 guest). This in turn allows you to create multiple nested guests (level-2 guests), running different OSes, on which you can develop and test your software.
- Live migration of "guest hypervisors" and their nested guests, for load balancing, disaster recovery, etc.
- VM image creation tools (e.g. `virt-install`, etc) often run their own VM, and users expect these to work inside a VM.
- Some OSes use virtualization internally for security (e.g. to let applications run safely in isolation).

Enabling "nested" (x86)

From Linux kernel v4.20 onwards, the `nested` KVM parameter is enabled by default for Intel and AMD. (Though your Linux distribution might override this default.)

In case you are running a Linux kernel older than v4.19, to enable nesting, set the `nested` KVM module parameter to `Y` or `1`. To persist this setting across reboots, you can add it in a config file, as shown below:

1. On the bare metal host (L0), list the kernel modules and ensure that the KVM modules:

```
$ lsmod | grep -i kvm
kvm_intel             133627  0
kvm                   435079  1 kvm_intel
```

2. Show information for `kvm_intel` module:

```
$ modinfo kvm_intel | grep -i nested
parm:           nested:bool
```

3. For the nested KVM configuration to persist across reboots, place the below in `/etc/modprobed/kvm_intel.conf` (create the file if it doesn't exist):

```
$ cat /etc/modprobe.d/kvm_intel.conf
options kvm-intel nested=y
```

4. Unload and re-load the KVM Intel module:

```
$ sudo rmmod kvm-intel
$ sudo modprobe kvm-intel
```

5. Verify if the `nested` parameter for KVM is enabled:

```
$ cat /sys/module/kvm_intel/parameters/nested
Y
```

For AMD hosts, the process is the same as above, except that the module name is `kvm-amd`.

Additional nested-related kernel parameters (x86)

If your hardware is sufficiently advanced (Intel Haswell processor or higher, which has newer hardware virt extensions), the following additional features will also be enabled by default: "Shadow VMCS (Virtual Machine Control Structure)", APIC Virtualization on your bare metal host (L0). Parameters for Intel hosts:

```
$ cat /sys/module/kvm_intel/parameters/enable_shadow_vmcs
Y

$ cat /sys/module/kvm_intel/parameters/enable_apicv
Y

$ cat /sys/module/kvm_intel/parameters/ept
Y
```

Note: If you suspect your L2 (i.e. nested guest) is running slower, ensure the above are enabled (particularly `enable_shadow_vmcs` and `ept`).

Starting a nested guest (x86)

Once your bare metal host (L0) is configured for nesting, you should be able to start an L1 guest with:

```
$ qemu-kvm -cpu host [...]
```

The above will pass through the host CPU's capabilities as-is to the guest, or for better live migration compatibility, use a named CPU model supported by QEMU. e.g.:

```
$ qemu-kvm -cpu Haswell-noTSX-IBRS,vmx=on
```

then the guest hypervisor will subsequently be capable of running a nested guest with accelerated KVM.

Enabling "nested" (s390x)

1. On the host hypervisor (L0), enable the `nested` parameter on s390x:

   ```
   $ rmmod kvm
   $ modprobe kvm nested=1
   ```

Note: On s390x, the kernel parameter `hpage` is mutually exclusive with the `nested` parameter — i.e. to be able to enable `nested`, the `hpage` parameter *must* be disabled.

2. The guest hypervisor (L1) must be provided with the `sie` CPU feature — with QEMU, this can be done by using ”host passthrough” (via the command-line `-cpu host`).
3. Now the KVM module can be loaded in the L1 (guest hypervisor):

```
$ modprobe kvm
```

Live migration with nested KVM

Migrating an L1 guest, with a *live* nested guest in it, to another bare metal host, works as of Linux kernel 5.3 and QEMU 4.2.0 for Intel x86 systems, and even on older versions for s390x.

On AMD systems, once an L1 guest has started an L2 guest, the L1 guest should no longer be migrated or saved (refer to QEMU documentation on ”savevm”/”loadvm”) until the L2 guest shuts down. Attempting to migrate or save-and-load an L1 guest while an L2 guest is running will result in undefined behavior. You might see a `kernel BUG!` entry in `dmesg`, a kernel 'oops', or an outright kernel panic. Such a migrated or loaded L1 guest can no longer be considered stable or secure, and must be restarted. Migrating an L1 guest merely configured to support nesting, while not actually running L2 guests, is expected to function normally even on AMD systems but may fail once guests are started.

Migrating an L2 guest is always expected to succeed, so all the following scenarios should work even on AMD systems:

- Migrating a nested guest (L2) to another L1 guest on the *same* bare metal host.
- Migrating a nested guest (L2) to another L1 guest on a *different* bare metal host.
- Migrating a nested guest (L2) to a bare metal host.

Reporting bugs from nested setups

Debugging ”nested” problems can involve sifting through log files across L0, L1 and L2; this can result in tedious back-n-forth between the bug reporter and the bug fixer.

- Mention that you are in a ”nested” setup. If you are running any kind of ”nesting” at all, say so. Unfortunately, this needs to be called out because when reporting bugs, people tend to forget to even *mention* that they're using nested virtualization.
- Ensure you are actually running KVM on KVM. Sometimes people do not have KVM enabled for their guest hypervisor (L1), which results in them running with pure emulation or what QEMU calls it as ”TCG”, but they think they're running nested KVM. Thus confusing ”nested Virt” (which could also mean, QEMU on KVM) with ”nested KVM” (KVM on KVM).

Information to collect (generic)

The following is not an exhaustive list, but a very good starting point:

- Kernel, libvirt, and QEMU version from L0
- Kernel, libvirt and QEMU version from L1
- QEMU command-line of L1 -- when using libvirt, you'll find it here: `/var/log/libvirt/qemu/instance.log`
- QEMU command-line of L2 -- as above, when using libvirt, get the complete libvirt-generated QEMU command-line
- `cat /sys/cpuinfo` from L0
- `cat /sys/cpuinfo` from L1
- `lscpu` from L0
- `lscpu` from L1
- Full `dmesg` output from L0
- Full `dmesg` output from L1

x86-specific info to collect

Both the below commands, `x86info` and `dmidecode`, should be available on most Linux distributions with the same name:

- Output of: `x86info -a` from L0
- Output of: `x86info -a` from L1
- Output of: `dmidecode` from L0
- Output of: `dmidecode` from L1

s390x-specific info to collect

Along with the earlier mentioned generic details, the below is also recommended:

- `/proc/sysinfo` from L1; this will also include the info from L0

1.6.9 Timekeeping Virtualization for X86-Based Architectures

Author
Zachary Amsden <zamsden@redhat.com>

1. Overview

One of the most complicated parts of the X86 platform, and specifically, the virtualization of this platform is the plethora of timing devices available and the complexity of emulating those devices. In addition, virtualization of time introduces a new set of challenges because it introduces a multiplexed division of time beyond the control of the guest CPU.

First, we will describe the various timekeeping hardware available, then present some of the problems which arise and solutions available, giving specific recommendations for certain classes of KVM guests.

The purpose of this document is to collect data and information relevant to timekeeping which may be difficult to find elsewhere, specifically, information relevant to KVM and hardware-based virtualization.

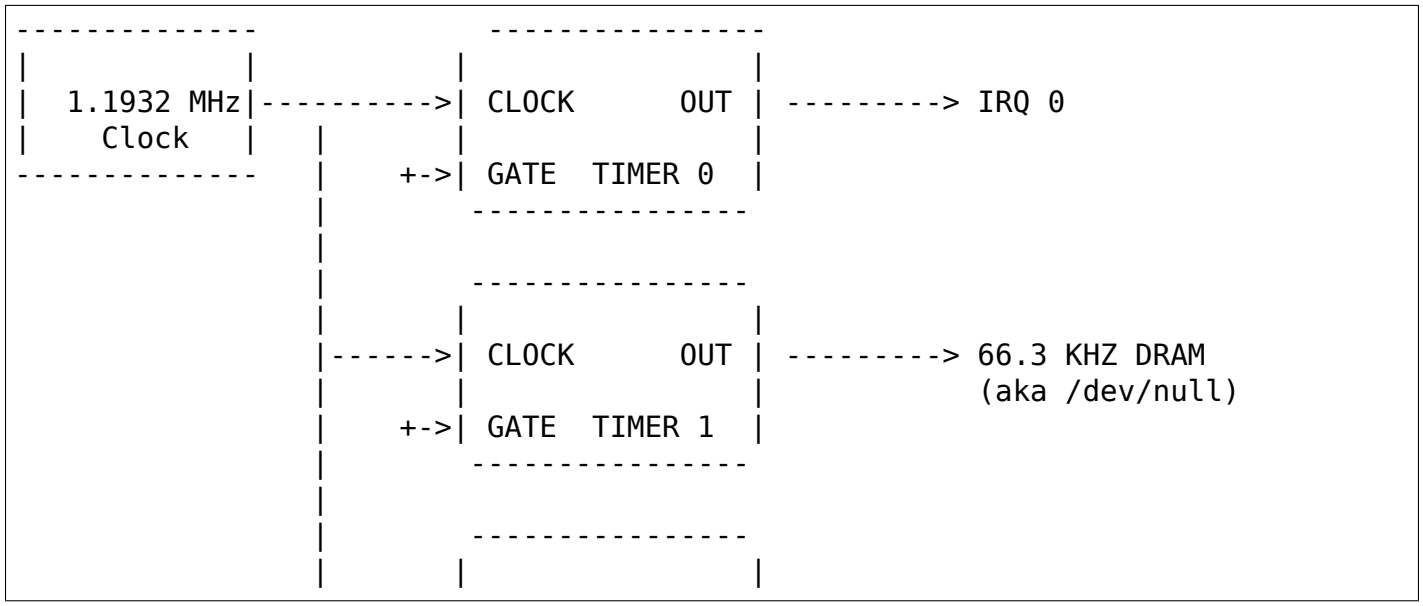
2. Timing Devices

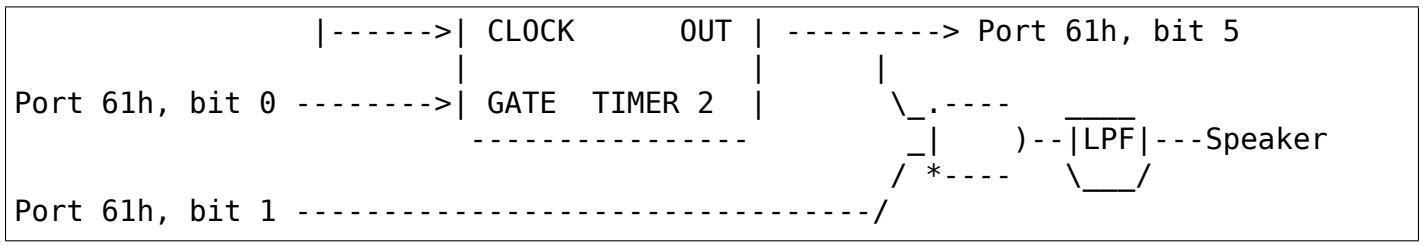
First we discuss the basic hardware devices available. TSC and the related KVM clock are special enough to warrant a full exposition and are described in the following section.

2.1. i8254 - PIT

One of the first timer devices available is the programmable interrupt timer, or PIT. The PIT has a fixed frequency 1.193182 MHz base clock and three channels which can be programmed to deliver periodic or one-shot interrupts. These three channels can be configured in different modes and have individual counters. Channel 1 and 2 were not available for general use in the original IBM PC, and historically were connected to control RAM refresh and the PC speaker. Now the PIT is typically integrated as part of an emulated chipset and a separate physical PIT is not used.

The PIT uses I/O ports 0x40 - 0x43. Access to the 16-bit counters is done using single or multiple byte access to the I/O ports. There are 6 modes available, but not all modes are available to all timers, as only timer 2 has a connected gate input, required for modes 1 and 5. The gate line is controlled by port 61h, bit 0, as illustrated in the following diagram:





The timer modes are now described.

Mode 0: Single Timeout.
This is a one-shot software timeout that counts down when the gate is high (always true for timers 0 and 1). When the count reaches zero, the output goes high.

Mode 1: Triggered One-shot.
The output is initially set high. When the gate line is set high, a countdown is initiated (which does not stop if the gate is lowered), during which the output is set low. When the count reaches zero, the output goes high.

Mode 2: Rate Generator.
The output is initially set high. When the countdown reaches 1, the output goes low for one count and then returns high. The value is reloaded and the countdown automatically resumes. If the gate line goes low, the count is halted. If the output is low when the gate is lowered, the output automatically goes high (this only affects timer 2).

Mode 3: Square Wave.
This generates a high / low square wave. The count determines the length of the pulse, which alternates between high and low when zero is reached. The count only proceeds when gate is high and is automatically reloaded on reaching zero. The count is decremented twice at each clock to generate a full high / low cycle at the full periodic rate. If the count is even, the clock remains high for N/2 counts and low for N/2 counts; if the clock is odd, the clock is high for (N+1)/2 counts and low for (N-1)/2 counts. Only even values are latched by the counter, so odd values are not observed when reading. This is the intended mode for timer 2, which generates sine-like tones by low-pass filtering the square wave output.

Mode 4: Software Strobe.
After programming this mode and loading the counter, the output remains high until the counter reaches zero. Then the output goes low for 1 clock cycle and returns high. The counter is not reloaded. Counting only occurs when gate is high.

Mode 5: Hardware Strobe.
After programming and loading the counter, the output remains high. When the gate is raised, a countdown is initiated (which does not stop if the gate is lowered). When the counter reaches zero, the output goes low for 1 clock cycle and then returns high. The counter is not reloaded.

In addition to normal binary counting, the PIT supports BCD counting. The command port, 0x43 is used to set the counter and mode for each of the three timers.

PIT commands, issued to port 0x43, using the following bit encoding:

```
Bit 7-4: Command (See table below)
Bit 3-1: Mode (000 = Mode 0, 101 = Mode 5, 11X = undefined)
Bit 0  : Binary (0) / BCD (1)
```

Command table:

```
0000 - Latch Timer 0 count for port 0x40
      sample and hold the count to be read in port 0x40;
      additional commands ignored until counter is read;
      mode bits ignored.

0001 - Set Timer 0 LSB mode for port 0x40
      set timer to read LSB only and force MSB to zero;
      mode bits set timer mode

0010 - Set Timer 0 MSB mode for port 0x40
      set timer to read MSB only and force LSB to zero;
      mode bits set timer mode

0011 - Set Timer 0 16-bit mode for port 0x40
      set timer to read / write LSB first, then MSB;
      mode bits set timer mode

0100 - Latch Timer 1 count for port 0x41 - as described above
0101 - Set Timer 1 LSB mode for port 0x41 - as described above
0110 - Set Timer 1 MSB mode for port 0x41 - as described above
0111 - Set Timer 1 16-bit mode for port 0x41 - as described above

1000 - Latch Timer 2 count for port 0x42 - as described above
1001 - Set Timer 2 LSB mode for port 0x42 - as described above
1010 - Set Timer 2 MSB mode for port 0x42 - as described above
1011 - Set Timer 2 16-bit mode for port 0x42 as described above

1101 - General counter latch
      Latch combination of counters into corresponding ports
      Bit 3 = Counter 2
      Bit 2 = Counter 1
      Bit 1 = Counter 0
      Bit 0 = Unused

1110 - Latch timer status
      Latch combination of counter mode into corresponding ports
      Bit 3 = Counter 2
      Bit 2 = Counter 1
      Bit 1 = Counter 0

      The output of ports 0x40-0x42 following this command will be:

      Bit 7 = Output pin
      Bit 6 = Count loaded (0 if timer has expired)
      Bit 5-4 = Read / Write mode
          01 = MSB only
          10 = LSB only
          11 = LSB / MSB (16-bit)
      Bit 3-1 = Mode
```

```
      Bit 0 = Binary (0) / BCD mode (1)
```

2.2. RTC

The second device which was available in the original PC was the MC146818 real time clock. The original device is now obsolete, and usually emulated by the system chipset, sometimes by an HPET and some frankenstein IRQ routing.

The RTC is accessed through CMOS variables, which uses an index register to control which bytes are read. Since there is only one index register, read of the CMOS and read of the RTC require lock protection (in addition, it is dangerous to allow userspace utilities such as hwclock to have direct RTC access, as they could corrupt kernel reads and writes of CMOS memory).

The RTC generates an interrupt which is usually routed to IRQ 8. The interrupt can function as a periodic timer, an additional once a day alarm, and can issue interrupts after an update of the CMOS registers by the MC146818 is complete. The type of interrupt is signalled in the RTC status registers.

The RTC will update the current time fields by battery power even while the system is off. The current time fields should not be read while an update is in progress, as indicated in the status register.

The clock uses a 32.768kHz crystal, so bits 6-4 of register A should be programmed to a 32kHz divider if the RTC is to count seconds.

This is the RAM map originally used for the RTC/CMOS:

```
Location    Size    Description
------------------------------------------
00h         byte    Current second (BCD)
01h         byte    Seconds alarm (BCD)
02h         byte    Current minute (BCD)
03h         byte    Minutes alarm (BCD)
04h         byte    Current hour (BCD)
05h         byte    Hours alarm (BCD)
06h         byte    Current day of week (BCD)
07h         byte    Current day of month (BCD)
08h         byte    Current month (BCD)
09h         byte    Current year (BCD)
0Ah         byte    Register A
                     bit 7   = Update in progress
                     bit 6-4 = Divider for clock
                               000 = 4.194 MHz
                               001 = 1.049 MHz
                               010 = 32 kHz
                               10X = test modes
                               110 = reset / disable
                               111 = reset / disable
                     bit 3-0 = Rate selection for periodic interrupt
                               000 = periodic timer disabled
                               001 = 3.90625 uS
                               010 = 7.8125 uS
```

```
                                011 = .122070 mS
                                100 = .244141 mS
                                   ...
                               1101 = 125 mS
                               1110 = 250 mS
                               1111 = 500 mS
0Bh         byte    Register B
                     bit 7   = Run (0) / Halt (1)
                     bit 6   = Periodic interrupt enable
                     bit 5   = Alarm interrupt enable
                     bit 4   = Update-ended interrupt enable
                     bit 3   = Square wave interrupt enable
                     bit 2   = BCD calendar (0) / Binary (1)
                     bit 1   = 12-hour mode (0) / 24-hour mode (1)
                     bit 0   = 0 (DST off) / 1 (DST enabled)
0Ch         byte    Register C (read only)
                     bit 7   = interrupt request flag (IRQF)
                     bit 6   = periodic interrupt flag (PF)
                     bit 5   = alarm interrupt flag (AF)
                     bit 4   = update interrupt flag (UF)
                     bit 3-0 = reserved
0Dh         byte    Register D (read only)
                     bit 7   = RTC has power
                     bit 6-0 = reserved
32h         byte    Current century BCD (*)
(*) location vendor specific and now determined from ACPI global tables
```

2.3. APIC

On Pentium and later processors, an on-board timer is available to each CPU as part of the Advanced Programmable Interrupt Controller. The APIC is accessed through memory-mapped registers and provides interrupt service to each CPU, used for IPIs and local timer interrupts.

Although in theory the APIC is a safe and stable source for local interrupts, in practice, many bugs and glitches have occurred due to the special nature of the APIC CPU-local memory-mapped hardware. Beware that CPU errata may affect the use of the APIC and that workarounds may be required. In addition, some of these workarounds pose unique constraints for virtualization - requiring either extra overhead incurred from extra reads of memory-mapped I/O or additional functionality that may be more computationally expensive to implement.

Since the APIC is documented quite well in the Intel and AMD manuals, we will avoid repetition of the detail here. It should be pointed out that the APIC timer is programmed through the LVT (local vector timer) register, is capable of one-shot or periodic operation, and is based on the bus clock divided down by the programmable divider register.

2.4. HPET

HPET is quite complex, and was originally intended to replace the PIT / RTC support of the X86 PC. It remains to be seen whether that will be the case, as the de facto standard of PC hardware is to emulate these older devices. Some systems designated as legacy free may support only the HPET as a hardware timer device.

The HPET spec is rather loose and vague, requiring at least 3 hardware timers, but allowing implementation freedom to support many more. It also imposes no fixed rate on the timer frequency, but does impose some extremal values on frequency, error and slew.

In general, the HPET is recommended as a high precision (compared to PIT /RTC) time source which is independent of local variation (as there is only one HPET in any given system). The HPET is also memory-mapped, and its presence is indicated through ACPI tables by the BIOS.

Detailed specification of the HPET is beyond the current scope of this document, as it is also very well documented elsewhere.

2.5. Offboard Timers

Several cards, both proprietary (watchdog boards) and commonplace (e1000) have timing chips built into the cards which may have registers which are accessible to kernel or user drivers. To the author's knowledge, using these to generate a clocksource for a Linux or other kernel has not yet been attempted and is in general frowned upon as not playing by the agreed rules of the game. Such a timer device would require additional support to be virtualized properly and is not considered important at this time as no known operating system does this.

3. TSC Hardware

The TSC or time stamp counter is relatively simple in theory; it counts instruction cycles issued by the processor, which can be used as a measure of time. In practice, due to a number of problems, it is the most complicated timekeeping device to use.

The TSC is represented internally as a 64-bit MSR which can be read with the RDMSR, RDTSC, or RDTSCP (when available) instructions. In the past, hardware limitations made it possible to write the TSC, but generally on old hardware it was only possible to write the low 32-bits of the 64-bit counter, and the upper 32-bits of the counter were cleared. Now, however, on Intel processors family 0Fh, for models 3, 4 and 6, and family 06h, models e and f, this restriction has been lifted and all 64-bits are writable. On AMD systems, the ability to write the TSC MSR is not an architectural guarantee.

The TSC is accessible from CPL-0 and conditionally, for CPL > 0 software by means of the CR4.TSD bit, which when enabled, disables CPL > 0 TSC access.

Some vendors have implemented an additional instruction, RDTSCP, which returns atomically not just the TSC, but an indicator which corresponds to the processor number. This can be used to index into an array of TSC variables to determine offset information in SMP systems where TSCs are not synchronized. The presence of this instruction must be determined by consulting CPUID feature bits.

Both VMX and SVM provide extension fields in the virtualization hardware which allows the guest visible TSC to be offset by a constant. Newer implementations promise to allow the TSC to additionally be scaled, but this hardware is not yet widely available.

3.1. TSC synchronization

The TSC is a CPU-local clock in most implementations. This means, on SMP platforms, the TSCs of different CPUs may start at different times depending on when the CPUs are powered on. Generally, CPUs on the same die will share the same clock, however, this is not always the case.

The BIOS may attempt to resynchronize the TSCs during the poweron process and the operating system or other system software may attempt to do this as well. Several hardware limitations make the problem worse - if it is not possible to write the full 64-bits of the TSC, it may be impossible to match the TSC in newly arriving CPUs to that of the rest of the system, resulting in unsynchronized TSCs. This may be done by BIOS or system software, but in practice, getting a perfectly synchronized TSC will not be possible unless all values are read from the same clock, which generally only is possible on single socket systems or those with special hardware support.

3.2. TSC and CPU hotplug

As touched on already, CPUs which arrive later than the boot time of the system may not have a TSC value that is synchronized with the rest of the system. Either system software, BIOS, or SMM code may actually try to establish the TSC to a value matching the rest of the system, but a perfect match is usually not a guarantee. This can have the effect of bringing a system from a state where TSC is synchronized back to a state where TSC synchronization flaws, however small, may be exposed to the OS and any virtualization environment.

3.3. TSC and multi-socket / NUMA

Multi-socket systems, especially large multi-socket systems are likely to have individual clocksources rather than a single, universally distributed clock. Since these clocks are driven by different crystals, they will not have perfectly matched frequency, and temperature and electrical variations will cause the CPU clocks, and thus the TSCs to drift over time. Depending on the exact clock and bus design, the drift may or may not be fixed in absolute error, and may accumulate over time.

In addition, very large systems may deliberately slew the clocks of individual cores. This technique, known as spread-spectrum clocking, reduces EMI at the clock frequency and harmonics of it, which may be required to pass FCC standards for telecommunications and computer equipment.

It is recommended not to trust the TSCs to remain synchronized on NUMA or multiple socket systems for these reasons.

3.4. TSC and C-states

C-states, or idling states of the processor, especially C1E and deeper sleep states may be problematic for TSC as well. The TSC may stop advancing in such a state, resulting in a TSC which is behind that of other CPUs when execution is resumed. Such CPUs must be detected and flagged by the operating system based on CPU and chipset identifications.

The TSC in such a case may be corrected by catching it up to a known external clocksource.

3.5. TSC frequency change / P-states

To make things slightly more interesting, some CPUs may change frequency. They may or may not run the TSC at the same rate, and because the frequency change may be staggered or slewed, at some points in time, the TSC rate may not be known other than falling within a range of values. In this case, the TSC will not be a stable time source, and must be calibrated against a known, stable, external clock to be a usable source of time.

Whether the TSC runs at a constant rate or scales with the P-state is model dependent and must be determined by inspecting CPUID, chipset or vendor specific MSR fields.

In addition, some vendors have known bugs where the P-state is actually compensated for properly during normal operation, but when the processor is inactive, the P-state may be raised temporarily to service cache misses from other processors. In such cases, the TSC on halted CPUs could advance faster than that of non-halted processors. AMD Turion processors are known to have this problem.

3.6. TSC and STPCLK / T-states

External signals given to the processor may also have the effect of stopping the TSC. This is typically done for thermal emergency power control to prevent an overheating condition, and typically, there is no way to detect that this condition has happened.

3.7. TSC virtualization - VMX

VMX provides conditional trapping of RDTSC, RDMSR, WRMSR and RDTSCP instructions, which is enough for full virtualization of TSC in any manner. In addition, VMX allows passing through the host TSC plus an additional TSC_OFFSET field specified in the VMCS. Special instructions must be used to read and write the VMCS field.

3.8. TSC virtualization - SVM

SVM provides conditional trapping of RDTSC, RDMSR, WRMSR and RDTSCP instructions, which is enough for full virtualization of TSC in any manner. In addition, SVM allows passing through the host TSC plus an additional offset field specified in the SVM control block.

3.9. TSC feature bits in Linux

In summary, there is no way to guarantee the TSC remains in perfect synchronization unless it is explicitly guaranteed by the architecture. Even if so, the TSCs in multi-sockets or NUMA systems may still run independently despite being locally consistent.

The following feature bits are used by Linux to signal various TSC attributes, but they can only be taken to be meaningful for UP or single node systems.

X86_FEATURE_TSC	The TSC is available in hardware
X86_FEATURE_RDTSCP	The RDTSCP instruction is available
X86_FEATURE_CONSTANT_TSC	The TSC rate is unchanged with P-states
X86_FEATURE_NONSTOP_TSC	The TSC does not stop in C-states
X86_FEATURE_TSC_RELIABLE	TSC sync checks are skipped (VMware)

4. Virtualization Problems

Timekeeping is especially problematic for virtualization because a number of challenges arise. The most obvious problem is that time is now shared between the host and, potentially, a number of virtual machines. Thus the virtual operating system does not run with 100% usage of the CPU, despite the fact that it may very well make that assumption. It may expect it to remain true to very exacting bounds when interrupt sources are disabled, but in reality only its virtual interrupt sources are disabled, and the machine may still be preempted at any time. This causes problems as the passage of real time, the injection of machine interrupts and the associated clock sources are no longer completely synchronized with real time.

This same problem can occur on native hardware to a degree, as SMM mode may steal cycles from the naturally on X86 systems when SMM mode is used by the BIOS, but not in such an extreme fashion. However, the fact that SMM mode may cause similar problems to virtualization makes it a good justification for solving many of these problems on bare metal.

4.1. Interrupt clocking

One of the most immediate problems that occurs with legacy operating systems is that the system timekeeping routines are often designed to keep track of time by counting periodic interrupts. These interrupts may come from the PIT or the RTC, but the problem is the same: the host virtualization engine may not be able to deliver the proper number of interrupts per second, and so guest time may fall behind. This is especially problematic if a high interrupt rate is selected, such as 1000 HZ, which is unfortunately the default for many Linux guests.

There are three approaches to solving this problem; first, it may be possible to simply ignore it. Guests which have a separate time source for tracking 'wall clock' or 'real time' may not need any adjustment of their interrupts to maintain proper time. If this is not sufficient, it may be necessary to inject additional interrupts into the guest in order to increase the effective interrupt rate. This approach leads to complications in extreme conditions, where host load or guest lag is too much to compensate for, and thus another solution to the problem has risen: the guest may need to become aware of lost ticks and compensate for them internally. Although promising in theory, the implementation of this policy in Linux has been extremely error prone, and a number of buggy variants of lost tick compensation are distributed across commonly used Linux systems.

Windows uses periodic RTC clocking as a means of keeping time internally, and thus requires interrupt slewing to keep proper time. It does use a low enough rate (ed: is it 18.2 Hz?) however that it has not yet been a problem in practice.

4.2. TSC sampling and serialization

As the highest precision time source available, the cycle counter of the CPU has aroused much interest from developers. As explained above, this timer has many problems unique to its nature as a local, potentially unstable and potentially unsynchronized source. One issue which is not unique to the TSC, but is highlighted because of its very precise nature is sampling delay. By definition, the counter, once read is already old. However, it is also possible for the counter to be read ahead of the actual use of the result. This is a consequence of the superscalar execution of the instruction stream, which may execute instructions out of order. Such execution is called non-serialized. Forcing serialized execution is necessary for precise measurement with the TSC, and requires a serializing instruction, such as CPUID or an MSR read.

Since CPUID may actually be virtualized by a trap and emulate mechanism, this serialization can pose a performance issue for hardware virtualization. An accurate time stamp counter reading may therefore not always be available, and it may be necessary for an implementation to guard against ”backwards” reads of the TSC as seen from other CPUs, even in an otherwise perfectly synchronized system.

4.3. Timespec aliasing

Additionally, this lack of serialization from the TSC poses another challenge when using results of the TSC when measured against another time source. As the TSC is much higher precision, many possible values of the TSC may be read while another clock is still expressing the same value.

That is, you may read (T,T+10) while external clock C maintains the same value. Due to non-serialized reads, you may actually end up with a range which fluctuates - from (T-1.. T+10). Thus, any time calculated from a TSC, but calibrated against an external value may have a range of valid values. Re-calibrating this computation may actually cause time, as computed after the calibration, to go backwards, compared with time computed before the calibration.

This problem is particularly pronounced with an internal time source in Linux, the kernel time, which is expressed in the theoretically high resolution timespec - but which advances in much larger granularity intervals, sometimes at the rate of jiffies, and possibly in catchup modes, at a much larger step.

This aliasing requires care in the computation and recalibration of kvmclock and any other values derived from TSC computation (such as TSC virtualization itself).

4.4. Migration

Migration of a virtual machine raises problems for timekeeping in two ways. First, the migration itself may take time, during which interrupts cannot be delivered, and after which, the guest time may need to be caught up. NTP may be able to help to some degree here, as the clock correction required is typically small enough to fall in the NTP-correctable window.

An additional concern is that timers based off the TSC (or HPET, if the raw bus clock is exposed) may now be running at different rates, requiring compensation in some way in the hypervisor by virtualizing these timers. In addition, migrating to a faster machine may preclude the use of a passthrough TSC, as a faster clock cannot be made visible to a guest without the potential of time advancing faster than usual. A slower clock is less of a problem, as it can always be caught up to the original rate. KVM clock avoids these problems by simply storing multipliers and offsets against the TSC for the guest to convert back into nanosecond resolution values.

4.5. Scheduling

Since scheduling may be based on precise timing and firing of interrupts, the scheduling algorithms of an operating system may be adversely affected by virtualization. In theory, the effect is random and should be universally distributed, but in contrived as well as real scenarios (guest device access, causes of virtualization exits, possible context switch), this may not always be the case. The effect of this has not been well studied.

In an attempt to work around this, several implementations have provided a paravirtualized scheduler clock, which reveals the true amount of CPU time for which a virtual machine has been running.

4.6. Watchdogs

Watchdog timers, such as the lock detector in Linux may fire accidentally when running under hardware virtualization due to timer interrupts being delayed or misinterpretation of the passage of real time. Usually, these warnings are spurious and can be ignored, but in some circumstances it may be necessary to disable such detection.

4.7. Delays and precision timing

Precise timing and delays may not be possible in a virtualized system. This can happen if the system is controlling physical hardware, or issues delays to compensate for slower I/O to and from devices. The first issue is not solvable in general for a virtualized system; hardware control software can't be adequately virtualized without a full real-time operating system, which would require an RT aware virtualization platform.

The second issue may cause performance problems, but this is unlikely to be a significant issue. In many cases these delays may be eliminated through configuration or paravirtualization.

4.8. Covert channels and leaks

In addition to the above problems, time information will inevitably leak to the guest about the host in anything but a perfect implementation of virtualized time. This may allow the guest to infer the presence of a hypervisor (as in a red-pill type detection), and it may allow information to leak between guests by using CPU utilization itself as a signalling channel. Preventing such problems would require completely isolated virtual time which may not track real time any longer. This may be useful in certain security or QA contexts, but in general isn't recommended for real-world deployment scenarios.

1.7 KVM Lock Overview

1.7.1 1. Acquisition Orders

The acquisition orders for mutexes are as follows:

- cpus_read_lock() is taken outside kvm_lock
- kvm->lock is taken outside vcpu->mutex
- kvm->lock is taken outside kvm->slots_lock and kvm->irq_lock
- kvm->slots_lock is taken outside kvm->irq_lock, though acquiring them together is quite rare.
- kvm->mn_active_invalidate_count ensures that pairs of invalidate_range_start() and invalidate_range_end() callbacks use the same memslots array. kvm->slots_lock and kvm->slots_arch_lock are taken on the waiting side when modifying memslots, so MMU notifiers must not take either kvm->slots_lock or kvm->slots_arch_lock.

For SRCU:

- `synchronize_srcu(&kvm->srcu)` is called inside critical sections for kvm->lock, vcpu->mutex and kvm->slots_lock. These locks _cannot_ be taken inside a kvm->srcu read-side critical section; that is, the following is broken:

```
srcu_read_lock(&kvm->srcu);
mutex_lock(&kvm->slots_lock);
```

- kvm->slots_arch_lock instead is released before the call to `synchronize_srcu()`. It _can_ therefore be taken inside a kvm->srcu read-side critical section, for example while processing a vmexit.

On x86:

- vcpu->mutex is taken outside kvm->arch.hyperv.hv_lock and kvm->arch.xen.xen_lock
- kvm->arch.mmu_lock is an rwlock; critical sections for kvm->arch.tdp_mmu_pages_lock and kvm->arch.mmu_unsync_pages_lock must also take kvm->arch.mmu_lock

Everything else is a leaf: no other lock is taken inside the critical sections.

1.7.2 2. Exception

Fast page fault:

Fast page fault is the fast path which fixes the guest page fault out of the mmu-lock on x86. Currently, the page fault can be fast in one of the following two cases:

1. Access Tracking: The SPTE is not present, but it is marked for access tracking. That means we need to restore the saved R/X bits. This is described in more detail later below.
2. Write-Protection: The SPTE is present and the fault is caused by write-protect. That means we just need to change the W bit of the spte.

What we use to avoid all the races is the Host-writable bit and MMU-writable bit on the spte:

- Host-writable means the gfn is writable in the host kernel page tables and in its KVM memslot.
- MMU-writable means the gfn is writable in the guest's mmu and it is not write-protected by shadow page write-protection.

On fast page fault path, we will use cmpxchg to atomically set the spte W bit if spte.HOST_WRITEABLE = 1 and spte.WRITE_PROTECT = 1, to restore the saved R/X bits if for an access-traced spte, or both. This is safe because whenever changing these bits can be detected by cmpxchg.

But we need carefully check these cases:

1) The mapping from gfn to pfn

The mapping from gfn to pfn may be changed since we can only ensure the pfn is not changed during cmpxchg. This is a ABA problem, for example, below case will happen:

<table>
<tr><td colspan="2">At the beginning:
<code>gpte = gfn1</code>
<code>gfn1 is mapped to pfn1 on host</code>
<code>spte is the shadow page table entry corresponding with gpte and</code>
<code>spte = pfn1</code></td></tr>
<tr><td colspan="2">On fast page fault path:</td></tr>
<tr><td>CPU 0:</td><td>CPU 1:</td></tr>
<tr><td><code>old_spte = *spte;</code></td><td></td></tr>
<tr><td></td><td>pfn1 is swapped out:
<code>spte = 0;</code>
pfn1 is re-alloced for gfn2.
gpte is changed to point to gfn2 by the guest:
<code>spte = pfn1;</code></td></tr>
<tr><td colspan="2"><code>if (cmpxchg(spte, old_spte, old_spte+W)</code>
<code>mark_page_dirty(vcpu->kvm, gfn1)</code>
<code>OOPS!!!</code></td></tr>
</table>

We dirty-log for gfn1, that means gfn2 is lost in dirty-bitmap.

For direct sp, we can easily avoid it since the spte of direct sp is fixed to gfn. For indirect sp, we disabled fast page fault for simplicity.

A solution for indirect sp could be to pin the gfn, for example via kvm_vcpu_gfn_to_pfn_atomic, before the cmpxchg. After the pinning:

- We have held the refcount of pfn; that means the pfn can not be freed and be reused for another gfn.
- The pfn is writable and therefore it cannot be shared between different gfns by KSM.

Then, we can ensure the dirty bitmaps is correctly set for a gfn.

2) Dirty bit tracking

In the origin code, the spte can be fast updated (non-atomically) if the spte is read-only and the Accessed bit has already been set since the Accessed bit and Dirty bit can not be lost.

But it is not true after fast page fault since the spte can be marked writable between reading spte and updating spte. Like below case:

<table>
<tr><td colspan="2">At the beginning:
<code>spte.W = 0</code>
<code>spte.Accessed = 1</code></td></tr>
<tr><td>CPU 0:</td><td>CPU 1:</td></tr>
<tr><td>In mmu_spte_clear_track_bits():
<code>old_spte = *spte;</code>

<code>/* 'if' condition is satisfied. */</code>
<code>if (old_spte.Accessed == 1 &&</code>
<code>old_spte.W == 0)</code>
<code>spte = 0ull;</code></td><td></td></tr>
<tr><td></td><td>on fast page fault path:
<code>spte.W = 1</code>
memory write on the spte:
<code>spte.Dirty = 1</code></td></tr>
<tr><td><code>else</code>
<code>old_spte = xchg(spte, 0ull)</code>
<code>if (old_spte.Accessed == 1)</code>
<code>kvm_set_pfn_accessed(spte.pfn);</code>
<code>if (old_spte.Dirty == 1)</code>
<code>kvm_set_pfn_dirty(spte.pfn);</code>
<code>OOPS!!!</code></td><td></td></tr>
</table>

The Dirty bit is lost in this case.

In order to avoid this kind of issue, we always treat the spte as ”volatile” if it can be updated out of mmu-lock [see spte_has_volatile_bits()]; it means the spte is always atomically updated in this case.

3) flush tlbs due to spte updated

If the spte is updated from writable to read-only, we should flush all TLBs, otherwise rmap_write_protect will find a read-only spte, even though the writable spte might be cached on a CPU's TLB.

As mentioned before, the spte can be updated to writable out of mmu-lock on fast page fault path. In order to easily audit the path, we see if TLBs needing to be flushed caused this reason in mmu_spte_update() since this is a common function to update spte (present -> present).

Since the spte is ”volatile” if it can be updated out of mmu-lock, we always atomically update the spte and the race caused by fast page fault can be avoided. See the comments in spte_has_volatile_bits() and mmu_spte_update().

Lockless Access Tracking:

This is used for Intel CPUs that are using EPT but do not support the EPT A/D bits. In this case, PTEs are tagged as A/D disabled (using ignored bits), and when the KVM MMU notifier is called to track accesses to a page (via kvm_mmu_notifier_clear_flush_young), it marks the PTE not-present in hardware by clearing the RWX bits in the PTE and storing the original R & X bits in more unused/ignored bits. When the VM tries to access the page later on, a fault is generated and the fast page fault mechanism described above is used to atomically restore the PTE to a Present state. The W bit is not saved when the PTE is marked for access tracking and during restoration to the Present state, the W bit is set depending on whether or not it was a write access. If it wasn't, then the W bit will remain clear until a write access happens, at which time it will be set using the Dirty tracking mechanism described above.

1.7.3 3. Reference

`kvm_lock`

Type
mutex

Arch
any

Protects

- vm_list
- kvm_usage_count
- hardware virtualization enable/disable

Comment
KVM also disables CPU hotplug via cpus_read_lock() during enable/disable.

kvm->mn_invalidate_lock

Type
spinlock_t

Arch
any

Protects
mn_active_invalidate_count, mn_memslots_update_rcuwait

kvm_arch::tsc_write_lock

Type
raw_spinlock_t

Arch
x86

Protects

- kvm_arch::{last_tsc_write,last_tsc_nsec,last_tsc_offset}
- tsc offset in vmcb

Comment
'raw' because updating the tsc offsets must not be preempted.

kvm->mmu_lock

Type
spinlock_t or rwlock_t

Arch
any

Protects
-shadow page/shadow tlb entry

Comment
it is a spinlock since it is used in mmu notifier.

kvm->srcu

Type
srcu lock

Arch
any

Protects

- kvm->memslots
- kvm->buses

Comment
The srcu read lock must be held while accessing memslots (e.g. when using gfn_to_* functions) and while accessing in-kernel MMIO/PIO address->device structure mapping (kvm->buses). The srcu index can be stored in kvm_vcpu->srcu_idx per vcpu if it is needed by multiple functions.

kvm->slots_arch_lock

Type
mutex

Arch
any (only needed on x86 though)

Protects
any arch-specific fields of memslots that have to be modified in a `kvm->srcu` read-side critical section.

Comment
must be held before reading the pointer to the current memslots, until after all changes to the memslots are complete

wakeup_vcpus_on_cpu_lock

Type
spinlock_t

Arch
x86

Protects
wakeup_vcpus_on_cpu

Comment
This is a per-CPU lock and it is used for VT-d posted-interrupts. When VT-d posted-interrupts are supported and the VM has assigned devices, we put the blocked vCPU on the list blocked_vcpu_on_cpu protected by blocked_vcpu_on_cpu_lock. When VT-d hardware issues wakeup notification event since external interrupts from the assigned devices happens, we will find the vCPU on the list to wakeup.

vendor_module_lock

Type
mutex

Arch
x86

Protects
loading a vendor module (kvm_amd or kvm_intel)

Comment

Exists because using kvm_lock leads to deadlock. cpu_hotplug_lock is taken outside of kvm_lock, e.g. in KVM's CPU online/offline callbacks, and many operations need to take cpu_hotplug_lock when loading a vendor module, e.g. updating static calls.

1.8 KVM VCPU Requests

1.8.1 Overview

KVM supports an internal API enabling threads to request a VCPU thread to perform some activity. For example, a thread may request a VCPU to flush its TLB with a VCPU request. The API consists of the following functions:

```
/* Check if any requests are pending for VCPU @vcpu. */
bool kvm_request_pending(struct kvm_vcpu *vcpu);

/* Check if VCPU @vcpu has request @req pending. */
bool kvm_test_request(int req, struct kvm_vcpu *vcpu);

/* Clear request @req for VCPU @vcpu. */
void kvm_clear_request(int req, struct kvm_vcpu *vcpu);

/*
 * Check if VCPU @vcpu has request @req pending. When the request is
 * pending it will be cleared and a memory barrier, which pairs with
 * another in kvm_make_request(), will be issued.
 */
bool kvm_check_request(int req, struct kvm_vcpu *vcpu);

/*
 * Make request @req of VCPU @vcpu. Issues a memory barrier, which pairs
 * with another in kvm_check_request(), prior to setting the request.
 */
void kvm_make_request(int req, struct kvm_vcpu *vcpu);

/* Make request @req of all VCPUs of the VM with struct kvm @kvm. */
bool kvm_make_all_cpus_request(struct kvm *kvm, unsigned int req);
```

Typically a requester wants the VCPU to perform the activity as soon as possible after making the request. This means most requests (kvm_make_request() calls) are followed by a call to kvm_vcpu_kick(), and kvm_make_all_cpus_request() has the kicking of all VCPUs built into it.

VCPU Kicks

The goal of a VCPU kick is to bring a VCPU thread out of guest mode in order to perform some KVM maintenance. To do so, an IPI is sent, forcing a guest mode exit. However, a VCPU thread may not be in guest mode at the time of the kick. Therefore, depending on the mode and state of the VCPU thread, there are two other actions a kick may take. All three actions are listed below:

1) Send an IPI. This forces a guest mode exit.
2) Waking a sleeping VCPU. Sleeping VCPUs are VCPU threads outside guest mode that wait on waitqueues. Waking them removes the threads from the waitqueues, allowing the threads to run again. This behavior may be suppressed, see KVM_REQUEST_NO_WAKEUP below.
3) Nothing. When the VCPU is not in guest mode and the VCPU thread is not sleeping, then there is nothing to do.

VCPU Mode

VCPUs have a mode state, `vcpu->mode`, that is used to track whether the guest is running in guest mode or not, as well as some specific outside guest mode states. The architecture may use `vcpu->mode` to ensure VCPU requests are seen by VCPUs (see "Ensuring Requests Are Seen"), as well as to avoid sending unnecessary IPIs (see "IPI Reduction"), and even to ensure IPI acknowledgements are waited upon (see "Waiting for Acknowledgements"). The following modes are defined:

OUTSIDE_GUEST_MODE

The VCPU thread is outside guest mode.

IN_GUEST_MODE

The VCPU thread is in guest mode.

EXITING_GUEST_MODE

The VCPU thread is transitioning from IN_GUEST_MODE to OUTSIDE_GUEST_MODE.

READING_SHADOW_PAGE_TABLES

The VCPU thread is outside guest mode, but it wants the sender of certain VCPU requests, namely KVM_REQ_TLB_FLUSH, to wait until the VCPU thread is done reading the page tables.

1.8.2 VCPU Request Internals

VCPU requests are simply bit indices of the `vcpu->requests` bitmap. This means general bitops, like those documented in [atomic-ops] could also be used, e.g.

```
clear_bit(KVM_REQ_UNBLOCK & KVM_REQUEST_MASK, &vcpu->requests);
```

However, VCPU request users should refrain from doing so, as it would break the abstraction. The first 8 bits are reserved for architecture independent requests; all additional bits are available for architecture dependent requests.

Architecture Independent Requests

KVM_REQ_TLB_FLUSH

KVM's common MMU notifier may need to flush all of a guest's TLB entries, calling kvm_flush_remote_tlbs() to do so. Architectures that choose to use the common kvm_flush_remote_tlbs() implementation will need to handle this VCPU request.

KVM_REQ_VM_DEAD

This request informs all VCPUs that the VM is dead and unusable, e.g. due to fatal error or because the VM's state has been intentionally destroyed.

KVM_REQ_UNBLOCK

This request informs the vCPU to exit kvm_vcpu_block. It is used for example from timer handlers that run on the host on behalf of a vCPU, or in order to update the interrupt routing and ensure that assigned devices will wake up the vCPU.

KVM_REQ_OUTSIDE_GUEST_MODE

This ”request” ensures the target vCPU has exited guest mode prior to the sender of the request continuing on. No action needs be taken by the target, and so no request is actually logged for the target. This request is similar to a ”kick”, but unlike a kick it guarantees the vCPU has actually exited guest mode. A kick only guarantees the vCPU will exit at some point in the future, e.g. a previous kick may have started the process, but there's no guarantee the to-be-kicked vCPU has fully exited guest mode.

KVM_REQUEST_MASK

VCPU requests should be masked by KVM_REQUEST_MASK before using them with bitops. This is because only the lower 8 bits are used to represent the request's number. The upper bits are used as flags. Currently only two flags are defined.

VCPU Request Flags

KVM_REQUEST_NO_WAKEUP

This flag is applied to requests that only need immediate attention from VCPUs running in guest mode. That is, sleeping VCPUs do not need to be awakened for these requests. Sleeping VCPUs will handle the requests when they are awakened later for some other reason.

KVM_REQUEST_WAIT

When requests with this flag are made with kvm_make_all_cpus_request(), then the caller will wait for each VCPU to acknowledge its IPI before proceeding. This flag only applies to VCPUs that would receive IPIs. If, for example, the VCPU is sleeping, so no IPI is necessary, then the requesting thread does not wait. This means that this flag may be safely combined with KVM_REQUEST_NO_WAKEUP. See ”Waiting for Acknowledgements” for more information about requests with KVM_REQUEST_WAIT.

1.8.3 VCPU Requests with Associated State

Requesters that want the receiving VCPU to handle new state need to ensure the newly written state is observable to the receiving VCPU thread's CPU by the time it observes the request. This means a write memory barrier must be inserted after writing the new state and before setting the VCPU request bit. Additionally, on the receiving VCPU thread's side, a corresponding read barrier must be inserted after reading the request bit and before proceeding to read the new state associated with it. See scenario 3, Message and Flag, of [lwn-mb] and the kernel documentation [memory-barriers].

The pair of functions, kvm_check_request() and kvm_make_request(), provide the memory barriers, allowing this requirement to be handled internally by the API.

1.8.4 Ensuring Requests Are Seen

When making requests to VCPUs, we want to avoid the receiving VCPU executing in guest mode for an arbitrary long time without handling the request. We can be sure this won't happen as long as we ensure the VCPU thread checks kvm_request_pending() before entering guest mode and that a kick will send an IPI to force an exit from guest mode when necessary. Extra care must be taken to cover the period after the VCPU thread's last kvm_request_pending() check and before it has entered guest mode, as kick IPIs will only trigger guest mode exits for VCPU threads that are in guest mode or at least have already disabled interrupts in order to prepare to enter guest mode. This means that an optimized implementation (see "IPI Reduction") must be certain when it's safe to not send the IPI. One solution, which all architectures except s390 apply, is to:

- set `vcpu->mode` to IN_GUEST_MODE between disabling the interrupts and the last kvm_request_pending() check;
- enable interrupts atomically when entering the guest.

This solution also requires memory barriers to be placed carefully in both the requesting thread and the receiving VCPU. With the memory barriers we can exclude the possibility of a VCPU thread observing !kvm_request_pending() on its last check and then not receiving an IPI for the next request made of it, even if the request is made immediately after the check. This is done by way of the Dekker memory barrier pattern (scenario 10 of [lwn-mb]). As the Dekker pattern requires two variables, this solution pairs `vcpu->mode` with `vcpu->requests`. Substituting them into the pattern gives:

```
CPU1                                    CPU2
=================                       =================
local_irq_disable();
WRITE_ONCE(vcpu->mode, IN_GUEST_MODE);  kvm_make_request(REQ, vcpu);
smp_mb();                               smp_mb();
if (kvm_request_pending(vcpu)) {        if (READ_ONCE(vcpu->mode) ==
                                            IN_GUEST_MODE) {
    ...abort guest entry...                 ...send IPI...
}                                       }
```

As stated above, the IPI is only useful for VCPU threads in guest mode or that have already disabled interrupts. This is why this specific case of the Dekker pattern has been extended to disable interrupts before setting `vcpu->mode` to IN_GUEST_MODE. WRITE_ONCE() and

READ_ONCE() are used to pedantically implement the memory barrier pattern, guaranteeing the compiler doesn't interfere with `vcpu->mode`'s carefully planned accesses.

IPI Reduction

As only one IPI is needed to get a VCPU to check for any/all requests, then they may be coalesced. This is easily done by having the first IPI sending kick also change the VCPU mode to something !IN_GUEST_MODE. The transitional state, EXITING_GUEST_MODE, is used for this purpose.

Waiting for Acknowledgements

Some requests, those with the KVM_REQUEST_WAIT flag set, require IPIs to be sent, and the acknowledgements to be waited upon, even when the target VCPU threads are in modes other than IN_GUEST_MODE. For example, one case is when a target VCPU thread is in READING_SHADOW_PAGE_TABLES mode, which is set after disabling interrupts. To support these cases, the KVM_REQUEST_WAIT flag changes the condition for sending an IPI from checking that the VCPU is IN_GUEST_MODE to checking that it is not OUTSIDE_GUEST_MODE.

Request-less VCPU Kicks

As the determination of whether or not to send an IPI depends on the two-variable Dekker memory barrier pattern, then it's clear that request-less VCPU kicks are almost never correct. Without the assurance that a non-IPI generating kick will still result in an action by the receiving VCPU, as the final kvm_request_pending() check does for request-accompanying kicks, then the kick may not do anything useful at all. If, for instance, a request-less kick was made to a VCPU that was just about to set its mode to IN_GUEST_MODE, meaning no IPI is sent, then the VCPU thread may continue its entry without actually having done whatever it was the kick was meant to initiate.

One exception is x86's posted interrupt mechanism. In this case, however, even the request-less VCPU kick is coupled with the same local_irq_disable() + smp_mb() pattern described above; the ON bit (Outstanding Notification) in the posted interrupt descriptor takes the role of `vcpu->requests`. When sending a posted interrupt, PIR.ON is set before reading `vcpu->mode`; dually, in the VCPU thread, vmx_sync_pir_to_irr() reads PIR after setting `vcpu->mode` to IN_GUEST_MODE.

1.8.5 Additional Considerations

Sleeping VCPUs

VCPU threads may need to consider requests before and/or after calling functions that may put them to sleep, e.g. kvm_vcpu_block(). Whether they do or not, and, if they do, which requests need consideration, is architecture dependent. kvm_vcpu_block() calls kvm_arch_vcpu_runnable() to check if it should awaken. One reason to do so is to provide architectures a function where requests may be checked if necessary.

1.8.6 References

1.9 The KVM halt polling system

The KVM halt polling system provides a feature within KVM whereby the latency of a guest can, under some circumstances, be reduced by polling in the host for some time period after the guest has elected to no longer run by cedeing. That is, when a guest vcpu has ceded, or in the case of powerpc when all of the vcpus of a single vcore have ceded, the host kernel polls for wakeup conditions before giving up the cpu to the scheduler in order to let something else run.

Polling provides a latency advantage in cases where the guest can be run again very quickly by at least saving us a trip through the scheduler, normally on the order of a few micro-seconds, although performance benefits are workload dependent. In the event that no wakeup source arrives during the polling interval or some other task on the runqueue is runnable the scheduler is invoked. Thus halt polling is especially useful on workloads with very short wakeup periods where the time spent halt polling is minimised and the time savings of not invoking the scheduler are distinguishable.

The generic halt polling code is implemented in:

> virt/kvm/kvm_main.c: kvm_vcpu_block()

The powerpc kvm-hv specific case is implemented in:

> arch/powerpc/kvm/book3s_hv.c: kvmppc_vcore_blocked()

1.9.1 Halt Polling Interval

The maximum time for which to poll before invoking the scheduler, referred to as the halt polling interval, is increased and decreased based on the perceived effectiveness of the polling in an attempt to limit pointless polling. This value is stored in either the vcpu struct:

> kvm_vcpu->halt_poll_ns

or in the case of powerpc kvm-hv, in the vcore struct:

> kvmppc_vcore->halt_poll_ns

Thus this is a per vcpu (or vcore) value.

During polling if a wakeup source is received within the halt polling interval, the interval is left unchanged. In the event that a wakeup source isn't received during the polling interval (and thus schedule is invoked) there are two options, either the polling interval and total block time[0] were less than the global max polling interval (see module params below), or the total block time was greater than the global max polling interval.

In the event that both the polling interval and total block time were less than the global max polling interval then the polling interval can be increased in the hope that next time during the longer polling interval the wake up source will be received while the host is polling and the latency benefits will be received. The polling interval is grown in the function grow_halt_poll_ns() and is multiplied by the module parameters halt_poll_ns_grow and halt_poll_ns_grow_start.

In the event that the total block time was greater than the global max polling interval then the host will never poll for long enough (limited by the global max) to wakeup during the polling interval so it may as well be shrunk in order to avoid pointless polling. The polling

interval is shrunk in the function shrink_halt_poll_ns() and is divided by the module parameter halt_poll_ns_shrink, or set to 0 iff halt_poll_ns_shrink == 0.

It is worth noting that this adjustment process attempts to hone in on some steady state polling interval but will only really do a good job for wakeups which come at an approximately constant rate, otherwise there will be constant adjustment of the polling interval.

[0] total block time:
the time between when the halt polling function is invoked and a wakeup source received (irrespective of whether the scheduler is invoked within that function).

1.9.2 Module Parameters

The kvm module has 3 tuneable module parameters to adjust the global max polling interval as well as the rate at which the polling interval is grown and shrunk. These variables are defined in include/linux/kvm_host.h and as module parameters in virt/kvm/kvm_main.c, or arch/powerpc/kvm/book3s_hv.c in the powerpc kvm-hv case.

Module Parameter	Description	Default Value
halt_poll_ns	The global max polling interval which defines the ceiling value of the polling interval for each vcpu.	KVM_HALT_POLL_NS_DEFAULT (per arch value)
halt_poll_ns_grow	The value by which the halt polling interval is multiplied in the grow_halt_poll_ns() function.	2
halt_poll_ns_grow_start	The initial value to grow to from zero in the grow_halt_poll_ns() function.	10000
halt_poll_ns_shrink	The value by which the halt polling interval is divided in the shrink_halt_poll_ns() function.	0

These module parameters can be set from the sysfs files in:

/sys/module/kvm/parameters/

Note: these module parameters are system-wide values and are not able to
be tuned on a per vm basis.

Any changes to these parameters will be picked up by new and existing vCPUs the next time they halt, with the notable exception of VMs using KVM_CAP_HALT_POLL (see next section).

1.9.3 KVM_CAP_HALT_POLL

KVM_CAP_HALT_POLL is a VM capability that allows userspace to override halt_poll_ns on a per-VM basis. VMs using KVM_CAP_HALT_POLL ignore halt_poll_ns completely (but still obey halt_poll_ns_grow, halt_poll_ns_grow_start, and halt_poll_ns_shrink).

See *The Definitive KVM (Kernel-based Virtual Machine) API Documentation* for more information on this capability.

1.9.4 Further Notes

- Care should be taken when setting the halt_poll_ns module parameter as a large value has the potential to drive the cpu usage to 100% on a machine which would be almost entirely idle otherwise. This is because even if a guest has wakeups during which very little work is done and which are quite far apart, if the period is shorter than the global max polling interval (halt_poll_ns) then the host will always poll for the entire block time and thus cpu utilisation will go to 100%.
- Halt polling essentially presents a trade-off between power usage and latency and the module parameters should be used to tune the affinity for this. Idle cpu time is essentially converted to host kernel time with the aim of decreasing latency when entering the guest.
- Halt polling will only be conducted by the host when no other tasks are runnable on that cpu, otherwise the polling will cease immediately and schedule will be invoked to allow that other task to run. Thus this doesn't allow a guest to cause denial of service of the cpu.

1.10 Review checklist for kvm patches

1. The patch must follow Documentation/process/coding-style.rst and Documentation/process/submitting-patches.rst.
2. Patches should be against kvm.git master branch.
3. If the patch introduces or modifies a new userspace API: - the API must be documented in *The Definitive KVM (Kernel-based Virtual Machine) API Documentation* - the API must be discoverable using KVM_CHECK_EXTENSION
4. New state must include support for save/restore.
5. New features must default to off (userspace should explicitly request them). Performance improvements can and should default to on.
6. New cpu features should be exposed via KVM_GET_SUPPORTED_CPUID2
7. Emulator changes should be accompanied by unit tests for qemu-kvm.git kvm/test directory.
8. Changes should be vendor neutral when possible. Changes to common code are better than duplicating changes to vendor code.
9. Similarly, prefer changes to arch independent code than to arch dependent code.
10. User/kernel interfaces and guest/host interfaces must be 64-bit clean (all variables and sizes naturally aligned on 64-bit; use specific types only - u64 rather than ulong).
11. New guest visible features must either be documented in a hardware manual or be accompanied by documentation.
12. Features must be robust against reset and kexec - for example, shared host/guest memory must be unshared to prevent the host from writing to guest memory that the guest has not reserved for this purpose.

CHAPTER

TWO

UML HOWTO

2.1 Introduction

Welcome to User Mode Linux

User Mode Linux is the first Open Source virtualization platform (first release date 1991) and second virtualization platform for an x86 PC.

2.1.1 How is UML Different from a VM using Virtualization package X?

We have come to assume that virtualization also means some level of hardware emulation. In fact, it does not. As long as a virtualization package provides the OS with devices which the OS can recognize and has a driver for, the devices do not need to emulate real hardware. Most OSes today have built-in support for a number of ”fake” devices used only under virtualization. User Mode Linux takes this concept to the ultimate extreme - there is not a single real device in sight. It is 100% artificial or if we use the correct term 100% paravirtual. All UML devices are abstract concepts which map onto something provided by the host - files, sockets, pipes, etc.

The other major difference between UML and various virtualization packages is that there is a distinct difference between the way the UML kernel and the UML programs operate. The UML kernel is just a process running on Linux - same as any other program. It can be run by an unprivileged user and it does not require anything in terms of special CPU features. The UML userspace, however, is a bit different. The Linux kernel on the host machine assists UML in intercepting everything the program running on a UML instance is trying to do and making the UML kernel handle all of its requests. This is different from other virtualization packages which do not make any difference between the guest kernel and guest programs. This difference results in a number of advantages and disadvantages of UML over let's say QEMU which we will cover later in this document.

2.1.2 Why Would I Want User Mode Linux?

- If User Mode Linux kernel crashes, your host kernel is still fine. It is not accelerated in any way (vhost, kvm, etc) and it is not trying to access any devices directly. It is, in fact, a process like any other.
- You can run a usermode kernel as a non-root user (you may need to arrange appropriate permissions for some devices).
- You can run a very small VM with a minimal footprint for a specific task (for example 32M or less).
- You can get extremely high performance for anything which is a ”kernel specific task” such as forwarding, firewalling, etc while still being isolated from the host kernel.
- You can play with kernel concepts without breaking things.
- You are not bound by ”emulating” hardware, so you can try weird and wonderful concepts which are very difficult to support when emulating real hardware such as time travel and making your system clock dependent on what UML does (very useful for things like tests).
- It's fun.

2.1.3 Why not to run UML

- The syscall interception technique used by UML makes it inherently slower for any userspace applications. While it can do kernel tasks on par with most other virtualization packages, its userspace is **slow**. The root cause is that UML has a very high cost of creating new processes and threads (something most Unix/Linux applications take for granted).
- UML is strictly uniprocessor at present. If you want to run an application which needs many CPUs to function, it is clearly the wrong choice.

2.2 Building a UML instance

There is no UML installer in any distribution. While you can use off the shelf install media to install into a blank VM using a virtualization package, there is no UML equivalent. You have to use appropriate tools on your host to build a viable filesystem image.

This is extremely easy on Debian - you can do it using debootstrap. It is also easy on OpenWRT - the build process can build UML images. All other distros - YMMV.

2.2.1 Creating an image

Create a sparse raw disk image:

```
# dd if=/dev/zero of=disk_image_name bs=1 count=1 seek=16G
```

This will create a 16G disk image. The OS will initially allocate only one block and will allocate more as they are written by UML. As of kernel version 4.19 UML fully supports TRIM (as usually used by flash drives). Using TRIM inside the UML image by specifying discard as a mount option or by running `tune2fs -o discard /dev/ubdXX` will request UML to return any unused blocks to the OS.

Create a filesystem on the disk image and mount it:

```
# mkfs.ext4 ./disk_image_name && mount ./disk_image_name /mnt
```

This example uses ext4, any other filesystem such as ext3, btrfs, xfs, jfs, etc will work too.

Create a minimal OS installation on the mounted filesystem:

```
# debootstrap buster /mnt http://deb.debian.org/debian
```

debootstrap does not set up the root password, fstab, hostname or anything related to networking. It is up to the user to do that.

Set the root password - the easiest way to do that is to chroot into the mounted image:

```
# chroot /mnt
# passwd
# exit
```

2.2.2 Edit key system files

UML block devices are called ubds. The fstab created by debootstrap will be empty and it needs an entry for the root file system:

```
/dev/ubd0   ext4    discard,errors=remount-ro  0       1
```

The image hostname will be set to the same as the host on which you are creating its image. It is a good idea to change that to avoid ”Oh, bummer, I rebooted the wrong machine”.

UML supports two classes of network devices - the older uml_net ones which are scheduled for obsoletion. These are called ethX. It also supports the newer vector IO devices which are significantly faster and have support for some standard virtual network encapsulations like Ethernet over GRE and Ethernet over L2TPv3. These are called vec0.

Depending on which one is in use, `/etc/network/interfaces` will need entries like:

```
# legacy UML network devices
auto eth0
iface eth0 inet dhcp

# vector UML network devices
auto vec0
iface vec0 inet dhcp
```

We now have a UML image which is nearly ready to run, all we need is a UML kernel and modules for it.

Most distributions have a UML package. Even if you intend to use your own kernel, testing the image with a stock one is always a good start. These packages come with a set of modules which should be copied to the target filesystem. The location is distribution dependent. For Debian these reside under /usr/lib/uml/modules. Copy recursively the content of this directory to the mounted UML filesystem:

```
# cp -rax /usr/lib/uml/modules /mnt/lib/modules
```

If you have compiled your own kernel, you need to use the usual ”install modules to a location” procedure by running:

```
# make INSTALL_MOD_PATH=/mnt/lib/modules modules_install
```

This will install modules into /mnt/lib/modules/$(KERNELRELEASE). To specify the full module installation path, use:

```
# make MODLIB=/mnt/lib/modules modules_install
```

At this point the image is ready to be brought up.

2.3 Setting Up UML Networking

UML networking is designed to emulate an Ethernet connection. This connection may be either point-to-point (similar to a connection between machines using a back-to-back cable) or a connection to a switch. UML supports a wide variety of means to build these connections to all of: local machine, remote machine(s), local and remote UML and other VM instances.

Transport	Type	Capabilities	Throughput
tap	vector	checksum, tso	> 8Gbit
hybrid	vector	checksum, tso, multipacket rx	> 6GBit
raw	vector	checksum, tso, multipacket rx, tx"	> 6GBit
EoGRE	vector	multipacket rx, tx	> 3Gbit
Eol2tpv3	vector	multipacket rx, tx	> 3Gbit
bess	vector	multipacket rx, tx	> 3Gbit
fd	vector	dependent on fd type	varies
tuntap	legacy	none	~ 500Mbit
daemon	legacy	none	~ 450Mbit
socket	legacy	none	~ 450Mbit
pcap	legacy	rx only	~ 450Mbit
ethertap	legacy	obsolete	~ 500Mbit
vde	legacy	obsolete	~ 500Mbit

- All transports which have tso and checksum offloads can deliver speeds approaching 10G on TCP streams.
- All transports which have multi-packet rx and/or tx can deliver pps rates of up to 1Mps or more.
- All legacy transports are generally limited to ~600-700MBit and 0.05Mps.
- GRE and L2TPv3 allow connections to all of: local machine, remote machines, remote network devices and remote UML instances.
- Socket allows connections only between UML instances.
- Daemon and bess require running a local switch. This switch may be connected to the host as well.

2.3.1 Network configuration privileges

The majority of the supported networking modes need `root` privileges. For example, in the legacy tuntap networking mode, users were required to be part of the group associated with the tunnel device.

For newer network drivers like the vector transports, `root` privilege is required to fire an ioctl to setup the tun interface and/or use raw sockets where needed.

This can be achieved by granting the user a particular capability instead of running UML as root. In case of vector transport, a user can add the capability `CAP_NET_ADMIN` or `CAP_NET_RAW` to the uml binary. Thenceforth, UML can be run with normal user privilges, along with full networking.

For example:

```
# sudo setcap cap_net_raw,cap_net_admin+ep linux
```

2.3.2 Configuring vector transports

All vector transports support a similar syntax:

If X is the interface number as in vec0, vec1, vec2, etc, the general syntax for options is:

```
vecX:transport="Transport Name",option=value,option=value,...,option=value
```

Common options

These options are common for all transports:

- `depth=int` - sets the queue depth for vector IO. This is the amount of packets UML will attempt to read or write in a single system call. The default number is 64 and is generally sufficient for most applications that need throughput in the 2-4 Gbit range. Higher speeds may require larger values.
- `mac=XX:XX:XX:XX:XX` - sets the interface MAC address value.
- `gro=[0,1]` - sets GRO off or on. Enables receive/transmit offloads. The effect of this option depends on the host side support in the transport which is being configured. In most cases it will enable TCP segmentation and RX/TX checksumming offloads. The setting must be identical on the host side and the UML side. The UML kernel will produce warnings if it is not. For example, GRO is enabled by default on local machine interfaces (e.g. veth pairs, bridge, etc), so it should be enabled in UML in the corresponding UML transports (raw, tap, hybrid) in order for networking to operate correctly.
- `mtu=int` - sets the interface MTU
- `headroom=int` - adjusts the default headroom (32 bytes) reserved if a packet will need to be re-encapsulated into for instance VXLAN.
- `vec=0` - disable multipacket IO and fall back to packet at a time mode

Shared Options

- `ifname=str` Transports which bind to a local network interface have a shared option - the name of the interface to bind to.
- `src, dst, src_port, dst_port` - all transports which use sockets which have the notion of source and destination and/or source port and destination port use these to specify them.
- `v6=[0,1]` to specify if a v6 connection is desired for all transports which operate over IP. Additionally, for transports that have some differences in the way they operate over v4 and v6 (for example EoL2TPv3), sets the correct mode of operation. In the absence of this option, the socket type is determined based on what do the src and dst arguments resolve/parse to.

tap transport

Example:

```
vecX:transport=tap,ifname=tap0,depth=128,gro=1
```

This will connect vec0 to tap0 on the host. Tap0 must already exist (for example created using tunctl) and UP.

tap0 can be configured as a point-to-point interface and given an IP address so that UML can talk to the host. Alternatively, it is possible to connect UML to a tap interface which is connected to a bridge.

While tap relies on the vector infrastructure, it is not a true vector transport at this point, because Linux does not support multi-packet IO on tap file descriptors for normal userspace apps like UML. This is a privilege which is offered only to something which can hook up to it at kernel level via specialized interfaces like vhost-net. A vhost-net like helper for UML is planned at some point in the future.

Privileges required: tap transport requires either:

- tap interface to exist and be created persistent and owned by the UML user using tunctl. Example `tunctl -u uml-user -t tap0`
- binary to have `CAP_NET_ADMIN` privilege

hybrid transport

Example:

```
vecX:transport=hybrid,ifname=tap0,depth=128,gro=1
```

This is an experimental/demo transport which couples tap for transmit and a raw socket for receive. The raw socket allows multi-packet receive resulting in significantly higher packet rates than normal tap.

Privileges required: hybrid requires `CAP_NET_RAW` capability by the UML user as well as the requirements for the tap transport.

raw socket transport

Example:

```
vecX:transport=raw,ifname=p-veth0,depth=128,gro=1
```

This transport uses vector IO on raw sockets. While you can bind to any interface including a physical one, the most common use it to bind to the ”peer” side of a veth pair with the other side configured on the host.

Example host configuration for Debian:

/etc/network/interfaces:

```
auto veth0
iface veth0 inet static
     address 192.168.4.1
     netmask 255.255.255.252
     broadcast 192.168.4.3
     pre-up ip link add veth0 type veth peer name p-veth0 && \
       ifconfig p-veth0 up
```

UML can now bind to p-veth0 like this:

```
vec0:transport=raw,ifname=p-veth0,depth=128,gro=1
```

If the UML guest is configured with 192.168.4.2 and netmask 255.255.255.0 it can talk to the host on 192.168.4.1

The raw transport also provides some support for offloading some of the filtering to the host. The two options to control it are:

- `bpffile=str` filename of raw bpf code to be loaded as a socket filter
- `bpfflash=int` 0/1 allow loading of bpf from inside User Mode Linux. This option allows the use of the ethtool load firmware command to load bpf code.

In either case the bpf code is loaded into the host kernel. While this is presently limited to legacy bpf syntax (not ebpf), it is still a security risk. It is not recommended to allow this unless the User Mode Linux instance is considered trusted.

Privileges required: raw socket transport requires *CAP_NET_RAW* capability.

GRE socket transport

Example:

```
vecX:transport=gre,src=$src_host,dst=$dst_host
```

This will configure an Ethernet over `GRE` (aka `GRETAP` or `GREIRB`) tunnel which will connect the UML instance to a `GRE` endpoint at host dst_host. `GRE` supports the following additional options:

- `rx_key=int` - GRE 32-bit integer key for rx packets, if set, `txkey` must be set too
- `tx_key=int` - GRE 32-bit integer key for tx packets, if set `rx_key` must be set too
- `sequence=[0,1]` - enable GRE sequence
- `pin_sequence=[0,1]` - pretend that the sequence is always reset on each packet (needed to interoperate with some really broken implementations)
- `v6=[0,1]` - force IPv4 or IPv6 sockets respectively
- GRE checksum is not presently supported

GRE has a number of caveats:

- You can use only one GRE connection per IP address. There is no way to multiplex connections as each GRE tunnel is terminated directly on the UML instance.
- The key is not really a security feature. While it was intended as such its ”security” is laughable. It is, however, a useful feature to ensure that the tunnel is not misconfigured.

An example configuration for a Linux host with a local address of 192.168.128.1 to connect to a UML instance at 192.168.129.1

/etc/network/interfaces:

```
auto gt0
iface gt0 inet static
 address 10.0.0.1
 netmask 255.255.255.0
 broadcast 10.0.0.255
 mtu 1500
 pre-up ip link add gt0 type gretap local 192.168.128.1 \
        remote 192.168.129.1 || true
 down ip link del gt0 || true
```

Additionally, GRE has been tested versus a variety of network equipment.

Privileges required: GRE requires `CAP_NET_RAW`

l2tpv3 socket transport

Warning. L2TPv3 has a ”bug”. It is the ”bug” known as ”has more options than GNU ls”. While it has some advantages, there are usually easier (and less verbose) ways to connect a UML instance to something. For example, most devices which support L2TPv3 also support GRE.

Example:

```
vec0:transport=l2tpv3,udp=1,src=$src_host,dst=$dst_host,srcport=$src_port,
 ↪dstport=$dst_port,depth=128,rx_session=0xffffffff,tx_session=0xffff
```

This will configure an Ethernet over L2TPv3 fixed tunnel which will connect the UML instance to a L2TPv3 endpoint at host $dst_host using the L2TPv3 UDP flavour and UDP destination port $dst_port.

L2TPv3 always requires the following additional options:

- `rx_session=int` - l2tpv3 32-bit integer session for rx packets
- `tx_session=int` - l2tpv3 32-bit integer session for tx packets

As the tunnel is fixed these are not negotiated and they are preconfigured on both ends.

Additionally, L2TPv3 supports the following optional parameters.

- `rx_cookie=int` - l2tpv3 32-bit integer cookie for rx packets - same functionality as GRE key, more to prevent misconfiguration than provide actual security
- `tx_cookie=int` - l2tpv3 32-bit integer cookie for tx packets
- `cookie64=[0,1]` - use 64-bit cookies instead of 32-bit.
- `counter=[0,1]` - enable l2tpv3 counter
- `pin_counter=[0,1]` - pretend that the counter is always reset on each packet (needed to interoperate with some really broken implementations)
- `v6=[0,1]` - force v6 sockets

- `udp=[0,1]` - use raw sockets (0) or UDP (1) version of the protocol

L2TPv3 has a number of caveats:

- you can use only one connection per IP address in raw mode. There is no way to multiplex connections as each L2TPv3 tunnel is terminated directly on the UML instance. UDP mode can use different ports for this purpose.

Here is an example of how to configure a Linux host to connect to UML via L2TPv3:

/etc/network/interfaces:

```
auto l2tp1
iface l2tp1 inet static
 address 192.168.126.1
 netmask 255.255.255.0
 broadcast 192.168.126.255
 mtu 1500
 pre-up ip l2tp add tunnel remote 127.0.0.1 \
        local 127.0.0.1 encap udp tunnel_id 2 \
        peer_tunnel_id 2 udp_sport 1706 udp_dport 1707 && \
        ip l2tp add session name l2tp1 tunnel_id 2 \
        session_id 0xffffffff peer_session_id 0xffffffff
 down ip l2tp del session tunnel_id 2 session_id 0xffffffff && \
        ip l2tp del tunnel tunnel_id 2
```

Privileges required: L2TPv3 requires `CAP_NET_RAW` for raw IP mode and no special privileges for the UDP mode.

BESS socket transport

BESS is a high performance modular network switch.

https://github.com/NetSys/bess

It has support for a simple sequential packet socket mode which in the more recent versions is using vector IO for high performance.

Example:

```
vecX:transport=bess,src=$unix_src,dst=$unix_dst
```

This will configure a BESS transport using the unix_src Unix domain socket address as source and unix_dst socket address as destination.

For BESS configuration and how to allocate a BESS Unix domain socket port please see the BESS documentation.

https://github.com/NetSys/bess/wiki/Built-In-Modules-and-Ports

BESS transport does not require any special privileges.

2.3.3 Configuring Legacy transports

Legacy transports are now considered obsolete. Please use the vector versions.

2.4 Running UML

This section assumes that either the user-mode-linux package from the distribution or a custom built kernel has been installed on the host.

These add an executable called linux to the system. This is the UML kernel. It can be run just like any other executable. It will take most normal linux kernel arguments as command line arguments. Additionally, it will need some UML-specific arguments in order to do something useful.

2.4.1 Arguments

Mandatory Arguments:

- `mem=int[K,M,G]` - amount of memory. By default in bytes. It will also accept K, M or G qualifiers.
- `ubdX[s,d,c,t]=` virtual disk specification. This is not really mandatory, but it is likely to be needed in nearly all cases so we can specify a root file system. The simplest possible image specification is the name of the image file for the filesystem (created using one of the methods described in *Creating an image*).
  - UBD devices support copy on write (COW). The changes are kept in a separate file which can be discarded allowing a rollback to the original pristine image. If COW is desired, the UBD image is specified as: `cow_file,master_image`. Example:`ubd0=Filesystem.cow,Filesystem.img`
  - UBD devices can be set to use synchronous IO. Any writes are immediately flushed to disk. This is done by adding `s` after the `ubdX` specification.
  - UBD performs some heuristics on devices specified as a single filename to make sure that a COW file has not been specified as the image. To turn them off, use the `d` flag after `ubdX`.
  - UBD supports TRIM - asking the Host OS to reclaim any unused blocks in the image. To turn it off, specify the `t` flag after `ubdX`.
- `root=` root device - most likely `/dev/ubd0` (this is a Linux filesystem image)

Important Optional Arguments

If UML is run as ”linux” with no extra arguments, it will try to start an xterm for every console configured inside the image (up to 6 in most Linux distributions). Each console is started inside an xterm. This makes it nice and easy to use UML on a host with a GUI. It is, however, the wrong approach if UML is to be used as a testing harness or run in a text-only environment.

In order to change this behaviour we need to specify an alternative console and wire it to one of the supported ”line” channels. For this we need to map a console to use something different from the default xterm.

Example which will divert console number 1 to stdin/stdout:

```
con1=fd:0,fd:1
```

UML supports a wide variety of serial line channels which are specified using the following syntax

conX=channel_type:options[,channel_type:options]

If the channel specification contains two parts separated by comma, the first one is input, the second one output.

- The null channel - Discard all input or output. Example `con=null` will set all consoles to null by default.
- The fd channel - use file descriptor numbers for input/output. Example: `con1=fd:0,fd:1.`
- The port channel - start a telnet server on TCP port number. Example: `con1=port:4321`. The host must have /usr/sbin/in.telnetd (usually part of a telnetd package) and the port-helper from the UML utilities (see the information for the xterm channel below). UML will not boot until a client connects.
- The pty and pts channels - use system pty/pts.
- The tty channel - bind to an existing system tty. Example: `con1=/dev/tty8` will make UML use the host 8th console (usually unused).
- The xterm channel - this is the default - bring up an xterm on this channel and direct IO to it. Note that in order for xterm to work, the host must have the UML distribution package installed. This usually contains the port-helper and other utilities needed for UML to communicate with the xterm. Alternatively, these need to be complied and installed from source. All options applicable to consoles also apply to UML serial lines which are presented as ttyS inside UML.

2.4.2 Starting UML

We can now run UML.

```
# linux mem=2048M umid=TEST \
 ubd0=Filesystem.img \
 vec0:transport=tap,ifname=tap0,depth=128,gro=1 \
 root=/dev/ubda con=null con0=null,fd:2 con1=fd:0,fd:1
```

This will run an instance with `2048M RAM` and try to use the image file called `Filesystem.img` as root. It will connect to the host using tap0. All consoles except `con1` will be disabled and console 1 will use standard input/output making it appear in the same terminal it was started.

2.4.3 Logging in

If you have not set up a password when generating the image, you will have to shut down the UML instance, mount the image, chroot into it and set it - as described in the Generating an Image section. If the password is already set, you can just log in.

2.4.4 The UML Management Console

In addition to managing the image from ”the inside” using normal sysadmin tools, it is possible to perform a number of low-level operations using the UML management console. The UML management console is a low-level interface to the kernel on a running UML instance, somewhat like the i386 SysRq interface. Since there is a full-blown operating system under UML, there is much greater flexibility possible than with the SysRq mechanism.

There are a number of things you can do with the mconsole interface:

- get the kernel version
- add and remove devices
- halt or reboot the machine
- Send SysRq commands
- Pause and resume the UML
- Inspect processes running inside UML
- Inspect UML internal /proc state

You need the mconsole client (uml_mconsole) which is a part of the UML tools package available in most Linux distritions.

You also need `CONFIG_MCONSOLE` (under 'General Setup') enabled in the UML kernel. When you boot UML, you'll see a line like:

```
mconsole initialized on /home/jdike/.uml/umlNJ32yL/mconsole
```

If you specify a unique machine id on the UML command line, i.e. `umid=debian`, you'll see this:

```
mconsole initialized on /home/jdike/.uml/debian/mconsole
```

That file is the socket that uml_mconsole will use to communicate with UML. Run it with either the umid or the full path as its argument:

```
# uml_mconsole debian
```

or

```
# uml_mconsole /home/jdike/.uml/debian/mconsole
```

You'll get a prompt, at which you can run one of these commands:

- version
- help
- halt
- reboot
- config
- remove
- sysrq
- help
- cad
- stop
- go
- proc
- stack

version

This command takes no arguments. It prints the UML version:

```
(mconsole)  version
OK Linux OpenWrt 4.14.106 #0 Tue Mar 19 08:19:41 2019 x86_64
```

There are a couple actual uses for this. It's a simple no-op which can be used to check that a UML is running. It's also a way of sending a device interrupt to the UML. UML mconsole is treated internally as a UML device.

help

This command takes no arguments. It prints a short help screen with the supported mconsole commands.

halt and reboot

These commands take no arguments. They shut the machine down immediately, with no syncing of disks and no clean shutdown of userspace. So, they are pretty close to crashing the machine:

```
(mconsole)  halt
OK
```

config

”config” adds a new device to the virtual machine. This is supported by most UML device drivers. It takes one argument, which is the device to add, with the same syntax as the kernel command line:

```
(mconsole) config ubd3=/home/jdike/incoming/roots/root_fs_debian22
```

remove

”remove” deletes a device from the system. Its argument is just the name of the device to be removed. The device must be idle in whatever sense the driver considers necessary. In the case of the ubd driver, the removed block device must not be mounted, swapped on, or otherwise open, and in the case of the network driver, the device must be down:

```
(mconsole)  remove ubd3
```

sysrq

This command takes one argument, which is a single letter. It calls the generic kernel's SysRq driver, which does whatever is called for by that argument. See the SysRq documentation in Documentation/admin-guide/sysrq.rst in your favorite kernel tree to see what letters are valid and what they do.

cad

This invokes the `Ctl-Alt-Del` action in the running image. What exactly this ends up doing is up to init, systemd, etc. Normally, it reboots the machine.

stop

This puts the UML in a loop reading mconsole requests until a 'go' mconsole command is received. This is very useful as a debugging/snapshotting tool.

go

This resumes a UML after being paused by a 'stop' command. Note that when the UML has resumed, TCP connections may have timed out and if the UML is paused for a long period of time, crond might go a little crazy, running all the jobs it didn't do earlier.

proc

This takes one argument - the name of a file in /proc which is printed to the mconsole standard output

stack

This takes one argument - the pid number of a process. Its stack is printed to a standard output.

2.5 Advanced UML Topics

2.5.1 Sharing Filesystems between Virtual Machines

Don't attempt to share filesystems simply by booting two UMLs from the same file. That's the same thing as booting two physical machines from a shared disk. It will result in filesystem corruption.

Using layered block devices

The way to share a filesystem between two virtual machines is to use the copy-on-write (COW) layering capability of the ubd block driver. Any changed blocks are stored in the private COW file, while reads come from either device - the private one if the requested block is valid in it, the shared one if not. Using this scheme, the majority of data which is unchanged is shared between an arbitrary number of virtual machines, each of which has a much smaller file containing the changes that it has made. With a large number of UMLs booting from a large root filesystem, this leads to a huge disk space saving.

Sharing file system data will also help performance, since the host will be able to cache the shared data using a much smaller amount of memory, so UML disk requests will be served from the host's memory rather than its disks. There is a major caveat in doing this on multisocket NUMA machines. On such hardware, running many UML instances with a shared master image and COW changes may cause issues like NMIs from excess of inter-socket traffic.

If you are running UML on high-end hardware like this, make sure to bind UML to a set of logical CPUs residing on the same socket using the `taskset` command or have a look at the ”tuning” section.

To add a copy-on-write layer to an existing block device file, simply add the name of the COW file to the appropriate ubd switch:

```
ubd0=root_fs_cow,root_fs_debian_22
```

where `root_fs_cow` is the private COW file and `root_fs_debian_22` is the existing shared filesystem. The COW file need not exist. If it doesn't, the driver will create and initialize it.

Disk Usage

UML has TRIM support which will release any unused space in its disk image files to the underlying OS. It is important to use either ls -ls or du to verify the actual file size.

COW validity.

Any changes to the master image will invalidate all COW files. If this happens, UML will *NOT* automatically delete any of the COW files and will refuse to boot. In this case the only solution is to either restore the old image (including its last modified timestamp) or remove all COW files which will result in their recreation. Any changes in the COW files will be lost.

Cows can moo - uml_moo : Merging a COW file with its backing file

Depending on how you use UML and COW devices, it may be advisable to merge the changes in the COW file into the backing file every once in a while.

The utility that does this is uml_moo. Its usage is:

```
uml_moo COW_file new_backing_file
```

There's no need to specify the backing file since that information is already in the COW file header. If you're paranoid, boot the new merged file, and if you're happy with it, move it over the old backing file.

`uml_moo` creates a new backing file by default as a safety measure. It also has a destructive merge option which will merge the COW file directly into its current backing file. This is really only usable when the backing file only has one COW file associated with it. If there are multiple COWs associated with a backing file, a -d merge of one of them will invalidate all of the others. However, it is convenient if you're short of disk space, and it should also be noticeably faster than a non-destructive merge.

`uml_moo` is installed with the UML distribution packages and is available as a part of UML utilities.

2.5.2 Host file access

If you want to access files on the host machine from inside UML, you can treat it as a separate machine and either nfs mount directories from the host or copy files into the virtual machine with scp. However, since UML is running on the host, it can access those files just like any other process and make them available inside the virtual machine without the need to use the network. This is possible with the hostfs virtual filesystem. With it, you can mount a host directory into the UML filesystem and access the files contained in it just as you would on the host.

SECURITY WARNING

Hostfs without any parameters to the UML Image will allow the image to mount any part of the host filesystem and write to it. Always confine hostfs to a specific ”harmless” directory (for example `/var/tmp`) if running UML. This is especially important if UML is being run as root.

Using hostfs

To begin with, make sure that hostfs is available inside the virtual machine with:

```
# cat /proc/filesystems
```

`hostfs` should be listed. If it's not, either rebuild the kernel with hostfs configured into it or make sure that hostfs is built as a module and available inside the virtual machine, and insmod it.

Now all you need to do is run mount:

```
# mount none /mnt/host -t hostfs
```

will mount the host's `/` on the virtual machine's `/mnt/host`. If you don't want to mount the host root directory, then you can specify a subdirectory to mount with the -o switch to mount:

```
# mount none /mnt/home -t hostfs -o /home
```

will mount the host's /home on the virtual machine's /mnt/home.

hostfs as the root filesystem

It's possible to boot from a directory hierarchy on the host using hostfs rather than using the standard filesystem in a file. To start, you need that hierarchy. The easiest way is to loop mount an existing root_fs file:

```
#  mount root_fs uml_root_dir -o loop
```

You need to change the filesystem type of `/` in `etc/fstab` to be 'hostfs', so that line looks like this:

```
/dev/ubd/0       /        hostfs      defaults          1   1
```

Then you need to chown to yourself all the files in that directory that are owned by root. This worked for me:

```
#  find . -uid 0 -exec chown jdike {} \;
```

Next, make sure that your UML kernel has hostfs compiled in, not as a module. Then run UML with the boot device pointing at that directory:

```
ubd0=/path/to/uml/root/directory
```

UML should then boot as it does normally.

Hostfs Caveats

Hostfs does not support keeping track of host filesystem changes on the host (outside UML). As a result, if a file is changed without UML's knowledge, UML will not know about it and its own in-memory cache of the file may be corrupt. While it is possible to fix this, it is not something which is being worked on at present.

2.5.3 Tuning UML

UML at present is strictly uniprocessor. It will, however spin up a number of threads to handle various functions.

The UBD driver, SIGIO and the MMU emulation do that. If the system is idle, these threads will be migrated to other processors on a SMP host. This, unfortunately, will usually result in LOWER performance because of all of the cache/memory synchronization traffic between cores. As a result, UML will usually benefit from being pinned on a single CPU, especially on a large system. This can result in performance differences of 5 times or higher on some benchmarks.

Similarly, on large multi-node NUMA systems UML will benefit if all of its memory is allocated from the same NUMA node it will run on. The OS will *NOT* do that by default. In order to do that, the sysadmin needs to create a suitable tmpfs ramdisk bound to a particular node and use that as the source for UML RAM allocation by specifying it in the TMP or TEMP environment variables. UML will look at the values of `TMPDIR`, `TMP` or `TEMP` for that. If that fails, it will look for shmfs mounted under `/dev/shm`. If everything else fails use `/tmp/` regardless of the filesystem type used for it:

```
mount -t tmpfs -ompol=bind:X none /mnt/tmpfs-nodeX
TEMP=/mnt/tmpfs-nodeX taskset -cX linux options options options..
```

2.6 Contributing to UML and Developing with UML

UML is an excellent platform to develop new Linux kernel concepts - filesystems, devices, virtualization, etc. It provides unrivalled opportunities to create and test them without being constrained to emulating specific hardware.

Example - want to try how Linux will work with 4096 ”proper” network devices?

Not an issue with UML. At the same time, this is something which is difficult with other virtualization packages - they are constrained by the number of devices allowed on the hardware bus they are trying to emulate (for example 16 on a PCI bus in qemu).

If you have something to contribute such as a patch, a bugfix, a new feature, please send it to `linux-um@lists.infradead.org`.

Please follow all standard Linux patch guidelines such as cc-ing relevant maintainers and run `./scripts/checkpatch.pl` on your patch. For more details see `Documentation/process/submitting-patches.rst`

Note - the list does not accept HTML or attachments, all emails must be formatted as plain text.

Developing always goes hand in hand with debugging. First of all, you can always run UML under gdb and there will be a whole section later on on how to do that. That, however, is not

the only way to debug a Linux kernel. Quite often adding tracing statements and/or using UML specific approaches such as ptracing the UML kernel process are significantly more informative.

2.6.1 Tracing UML

When running, UML consists of a main kernel thread and a number of helper threads. The ones of interest for tracing are NOT the ones that are already ptraced by UML as a part of its MMU emulation.

These are usually the first three threads visible in a ps display. The one with the lowest PID number and using most CPU is usually the kernel thread. The other threads are the disk (ubd) device helper thread and the SIGIO helper thread. Running ptrace on this thread usually results in the following picture:

```
host$ strace -p 16566
--- SIGIO {si_signo=SIGIO, si_code=POLL_IN, si_band=65} ---
epoll_wait(4, [{EPOLLIN, {u32=3721159424, u64=3721159424}}], 64, 0) = 1
epoll_wait(4, [], 64, 0)                = 0
rt_sigreturn({mask=[PIPE]})             = 16967
ptrace(PTRACE_GETREGS, 16967, NULL, 0xd5f34f38) = 0
ptrace(PTRACE_GETREGSET, 16967, NT_X86_XSTATE, [{iov_base=0xd5f35010, iov_
↪len=832}]) = 0
ptrace(PTRACE_GETSIGINFO, 16967, NULL, {si_signo=SIGTRAP, si_code=0x85, si_
↪pid=16967, si_uid=0}) = 0
ptrace(PTRACE_SETREGS, 16967, NULL, 0xd5f34f38) = 0
ptrace(PTRACE_SETREGSET, 16967, NT_X86_XSTATE, [{iov_base=0xd5f35010, iov_
↪len=2696}]) = 0
ptrace(PTRACE_SYSEMU, 16967, NULL, 0)   = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_TRAPPED, si_pid=16967, si_uid=0, si_
↪status=SIGTRAP, si_utime=65, si_stime=89} ---
wait4(16967, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP | 0x80}], WSTOPPED|__
↪WALL, NULL) = 16967
ptrace(PTRACE_GETREGS, 16967, NULL, 0xd5f34f38) = 0
ptrace(PTRACE_GETREGSET, 16967, NT_X86_XSTATE, [{iov_base=0xd5f35010, iov_
↪len=832}]) = 0
ptrace(PTRACE_GETSIGINFO, 16967, NULL, {si_signo=SIGTRAP, si_code=0x85, si_
↪pid=16967, si_uid=0}) = 0
timer_settime(0, 0, {it_interval={tv_sec=0, tv_nsec=0}, it_value={tv_sec=0, tv_
↪nsec=2830912}}, NULL) = 0
getpid()                                = 16566
clock_nanosleep(CLOCK_MONOTONIC, 0, {tv_sec=1, tv_nsec=0}, NULL) = ? ERESTART_
↪RESTARTBLOCK (Interrupted by signal)
--- SIGALRM {si_signo=SIGALRM, si_code=SI_TIMER, si_timerid=0, si_overrun=0,␣
↪si_value={int=1631716592, ptr=0x614204f0}} ---
rt_sigreturn({mask=[PIPE]})             = -1 EINTR (Interrupted system call)
```

This is a typical picture from a mostly idle UML instance.

- UML interrupt controller uses epoll - this is UML waiting for IO interrupts:

  epoll_wait(4, [{EPOLLIN, {u32=3721159424, u64=3721159424}}], 64, 0) = 1

- The sequence of ptrace calls is part of MMU emulation and running the UML userspace.

- `timer_settime` is part of the UML high res timer subsystem mapping timer requests from inside UML onto the host high resolution timers.
- `clock_nanosleep` is UML going into idle (similar to the way a PC will execute an ACPI idle).

As you can see UML will generate quite a bit of output even in idle. The output can be very informative when observing IO. It shows the actual IO calls, their arguments and returns values.

2.6.2 Kernel debugging

You can run UML under gdb now, though it will not necessarily agree to be started under it. If you are trying to track a runtime bug, it is much better to attach gdb to a running UML instance and let UML run.

Assuming the same PID number as in the previous example, this would be:

```
# gdb -p 16566
```

This will STOP the UML instance, so you must enter *cont* at the GDB command line to request it to continue. It may be a good idea to make this into a gdb script and pass it to gdb as an argument.

2.6.3 Developing Device Drivers

Nearly all UML drivers are monolithic. While it is possible to build a UML driver as a kernel module, that limits the possible functionality to in-kernel only and non-UML specific. The reason for this is that in order to really leverage UML, one needs to write a piece of userspace code which maps driver concepts onto actual userspace host calls.

This forms the so-called ”user” portion of the driver. While it can reuse a lot of kernel concepts, it is generally just another piece of userspace code. This portion needs some matching ”kernel” code which resides inside the UML image and which implements the Linux kernel part.

Note: There are very few limitations in the way ”kernel” and ”user” interact.

UML does not have a strictly defined kernel-to-host API. It does not try to emulate a specific architecture or bus. UML's ”kernel” and ”user” can share memory, code and interact as needed to implement whatever design the software developer has in mind. The only limitations are purely technical. Due to a lot of functions and variables having the same names, the developer should be careful which includes and libraries they are trying to refer to.

As a result a lot of userspace code consists of simple wrappers. E.g. `os_close_file()` is just a wrapper around `close()` which ensures that the userspace function close does not clash with similarly named function(s) in the kernel part.

2.6.4 Using UML as a Test Platform

UML is an excellent test platform for device driver development. As with most things UML, ”some user assembly may be required”. It is up to the user to build their emulation environment. UML at present provides only the kernel infrastructure.

Part of this infrastructure is the ability to load and parse fdt device tree blobs as used in Arm or Open Firmware platforms. These are supplied as an optional extra argument to the kernel command line:

```
dtb=filename
```

The device tree is loaded and parsed at boottime and is accessible by drivers which query it. At this moment in time this facility is intended solely for development purposes. UML's own devices do not query the device tree.

Security Considerations

Drivers or any new functionality should default to not accepting arbitrary filename, bpf code or other parameters which can affect the host from inside the UML instance. For example, specifying the socket used for IPC communication between a driver and the host at the UML command line is OK security-wise. Allowing it as a loadable module parameter isn't.

If such functionality is desirable for a particular application (e.g. loading BPF ”firmware” for raw socket network transports), it should be off by default and should be explicitly turned on as a command line parameter at startup.

Even with this in mind, the level of isolation between UML and the host is relatively weak. If the UML userspace is allowed to load arbitrary kernel drivers, an attacker can use this to break out of UML. Thus, if UML is used in a production application, it is recommended that all modules are loaded at boot and kernel module loading is disabled afterwards.

CHAPTER

THREE

PARAVIRT_OPS

Linux provides support for different hypervisor virtualization technologies. Historically, different binary kernels would be required in order to support different hypervisors; this restriction was removed with pv_ops. Linux pv_ops is a virtualization API which enables support for different hypervisors. It allows each hypervisor to override critical operations and allows a single kernel binary to run on all supported execution environments including native machine -- without any hypervisors.

pv_ops provides a set of function pointers which represent operations corresponding to low-level critical instructions and high-level functionalities in various areas. pv_ops allows for optimizations at run time by enabling binary patching of the low-level critical operations at boot time.

pv_ops operations are classified into three categories:

- **simple indirect call**
  These operations correspond to high-level functionality where it is known that the overhead of indirect call isn't very important.
- **indirect call which allows optimization with binary patch**
  Usually these operations correspond to low-level critical instructions. They are called frequently and are performance critical. The overhead is very important.
- **a set of macros for hand written assembly code**
  Hand written assembly codes (.S files) also need paravirtualization because they include sensitive instructions or some code paths in them are very performance critical.

CHAPTER

FOUR

GUEST HALT POLLING

The cpuidle_haltpoll driver, with the haltpoll governor, allows the guest vcpus to poll for a specified amount of time before halting.

This provides the following benefits to host side polling:

1) The POLL flag is set while polling is performed, which allows a remote vCPU to avoid sending an IPI (and the associated cost of handling the IPI) when performing a wakeup.

2) The VM-exit cost can be avoided.

The downside of guest side polling is that polling is performed even with other runnable tasks in the host.

The basic logic as follows: A global value, guest_halt_poll_ns, is configured by the user, indicating the maximum amount of time polling is allowed. This value is fixed.

Each vcpu has an adjustable guest_halt_poll_ns ("per-cpu guest_halt_poll_ns"), which is adjusted by the algorithm in response to events (explained below).

4.1 Module Parameters

The haltpoll governor has 5 tunable module parameters:

1) guest_halt_poll_ns:

Maximum amount of time, in nanoseconds, that polling is performed before halting.

Default: 200000

2) guest_halt_poll_shrink:

Division factor used to shrink per-cpu guest_halt_poll_ns when wakeup event occurs after the global guest_halt_poll_ns.

Default: 2

3) guest_halt_poll_grow:

Multiplication factor used to grow per-cpu guest_halt_poll_ns when event occurs after per-cpu guest_halt_poll_ns but before global guest_halt_poll_ns.

Default: 2

4) guest_halt_poll_grow_start:

The per-cpu guest_halt_poll_ns eventually reaches zero in case of an idle system. This value sets the initial per-cpu guest_halt_poll_ns when growing. This can be increased from 10000, to avoid misses during the initial growth stage:

10k, 20k, 40k, ... (example assumes guest_halt_poll_grow=2).

Default: 50000

5) guest_halt_poll_allow_shrink:

Bool parameter which allows shrinking. Set to N to avoid it (per-cpu guest_halt_poll_ns will remain high once achieves global guest_halt_poll_ns value).

Default: Y

The module parameters can be set from the sysfs files in:

```
/sys/module/haltpoll/parameters/
```

4.2 Further Notes

- Care should be taken when setting the guest_halt_poll_ns parameter as a large value has the potential to drive the cpu usage to 100% on a machine which would be almost entirely idle otherwise.

CHAPTER

FIVE

NITRO ENCLAVES

5.1 Overview

Nitro Enclaves (NE) is a new Amazon Elastic Compute Cloud (EC2) capability that allows customers to carve out isolated compute environments within EC2 instances [1].

For example, an application that processes sensitive data and runs in a VM, can be separated from other applications running in the same VM. This application then runs in a separate VM than the primary VM, namely an enclave. It runs alongside the VM that spawned it. This setup matches low latency applications needs.

The current supported architectures for the NE kernel driver, available in the upstream Linux kernel, are x86 and ARM64.

The resources that are allocated for the enclave, such as memory and CPUs, are carved out of the primary VM. Each enclave is mapped to a process running in the primary VM, that communicates with the NE kernel driver via an ioctl interface.

In this sense, there are two components:

1. An enclave abstraction process - a user space process running in the primary VM guest that uses the provided ioctl interface of the NE driver to spawn an enclave VM (that's 2 below).

There is a NE emulated PCI device exposed to the primary VM. The driver for this new PCI device is included in the NE driver.

The ioctl logic is mapped to PCI device commands e.g. the NE_START_ENCLAVE ioctl maps to an enclave start PCI command. The PCI device commands are then translated into actions taken on the hypervisor side; that's the Nitro hypervisor running on the host where the primary VM is running. The Nitro hypervisor is based on core KVM technology.

2. The enclave itself - a VM running on the same host as the primary VM that spawned it. Memory and CPUs are carved out of the primary VM and are dedicated for the enclave VM. An enclave does not have persistent storage attached.

The memory regions carved out of the primary VM and given to an enclave need to be aligned 2 MiB / 1 GiB physically contiguous memory regions (or multiple of this size e.g. 8 MiB). The memory can be allocated e.g. by using hugetlbfs from user space [2][3][7]. The memory size for an enclave needs to be at least 64 MiB. The enclave memory and CPUs need to be from the same NUMA node.

An enclave runs on dedicated cores. CPU 0 and its CPU siblings need to remain available for the primary VM. A CPU pool has to be set for NE purposes by an user with admin capability. See the cpu list section from the kernel documentation [4] for how a CPU pool format looks.

An enclave communicates with the primary VM via a local communication channel, using virtio-vsock [5]. The primary VM has virtio-pci vsock emulated device, while the enclave VM has a virtio-mmio vsock emulated device. The vsock device uses eventfd for signaling. The enclave VM sees the usual interfaces - local APIC and IOAPIC - to get interrupts from virtio-vsock device. The virtio-mmio device is placed in memory below the typical 4 GiB.

The application that runs in the enclave needs to be packaged in an enclave image together with the OS (e.g. kernel, ramdisk, init) that will run in the enclave VM. The enclave VM has its own kernel and follows the standard Linux boot protocol [6][8].

The kernel bzImage, the kernel command line, the ramdisk(s) are part of the Enclave Image Format (EIF); plus an EIF header including metadata such as magic number, eif version, image size and CRC.

Hash values are computed for the entire enclave image (EIF), the kernel and ramdisk(s). That's used, for example, to check that the enclave image that is loaded in the enclave VM is the one that was intended to be run.

These crypto measurements are included in a signed attestation document generated by the Nitro Hypervisor and further used to prove the identity of the enclave; KMS is an example of service that NE is integrated with and that checks the attestation doc.

The enclave image (EIF) is loaded in the enclave memory at offset 8 MiB. The init process in the enclave connects to the vsock CID of the primary VM and a predefined port - 9000 - to send a heartbeat value - 0xb7. This mechanism is used to check in the primary VM that the enclave has booted. The CID of the primary VM is 3.

If the enclave VM crashes or gracefully exits, an interrupt event is received by the NE driver. This event is sent further to the user space enclave process running in the primary VM via a poll notification mechanism. Then the user space enclave process can exit.

[1] https://aws.amazon.com/ec2/nitro/nitro-enclaves/ [2] https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html [3] https://lwn.net/Articles/807108/ [4] https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html [5] https://man7.org/linux/man-pages/man7/vsock.7.html [6] https://www.kernel.org/doc/html/latest/x86/boot.html [7] https://www.kernel.org/doc/html/latest/arm64/hugetlbpage.html [8] https://www.kernel.org/doc/html/latest/arm64/booting.html

CHAPTER

SIX

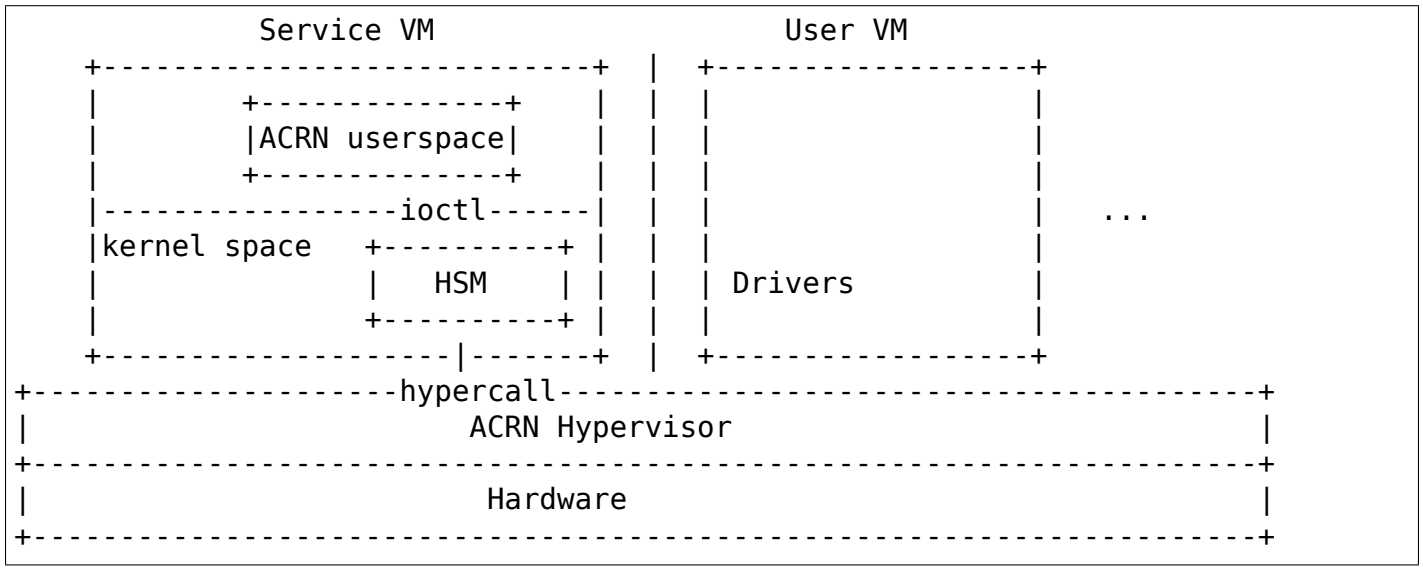
ACRN HYPERVISOR

6.1 ACRN Hypervisor Introduction

The ACRN Hypervisor is a Type 1 hypervisor, running directly on bare-metal hardware. It has a privileged management VM, called Service VM, to manage User VMs and do I/O emulation.

ACRN userspace is an application running in the Service VM that emulates devices for a User VM based on command line configurations. ACRN Hypervisor Service Module (HSM) is a kernel module in the Service VM which provides hypervisor services to the ACRN userspace.

Below figure shows the architecture.



ACRN userspace allocates memory for the User VM, configures and initializes the devices used by the User VM, loads the virtual bootloader, initializes the virtual CPU state and handles I/O request accesses from the User VM. It uses ioctls to communicate with the HSM. HSM implements hypervisor services by interacting with the ACRN Hypervisor via hypercalls. HSM exports a char device interface (/dev/acrn_hsm) to userspace.

The ACRN hypervisor is open for contribution from anyone. The source repo is available at https://github.com/projectacrn/acrn-hypervisor.

6.2 I/O request handling

An I/O request of a User VM, which is constructed by the hypervisor, is distributed by the ACRN Hypervisor Service Module to an I/O client corresponding to the address range of the I/O request. Details of I/O request handling are described in the following sections.

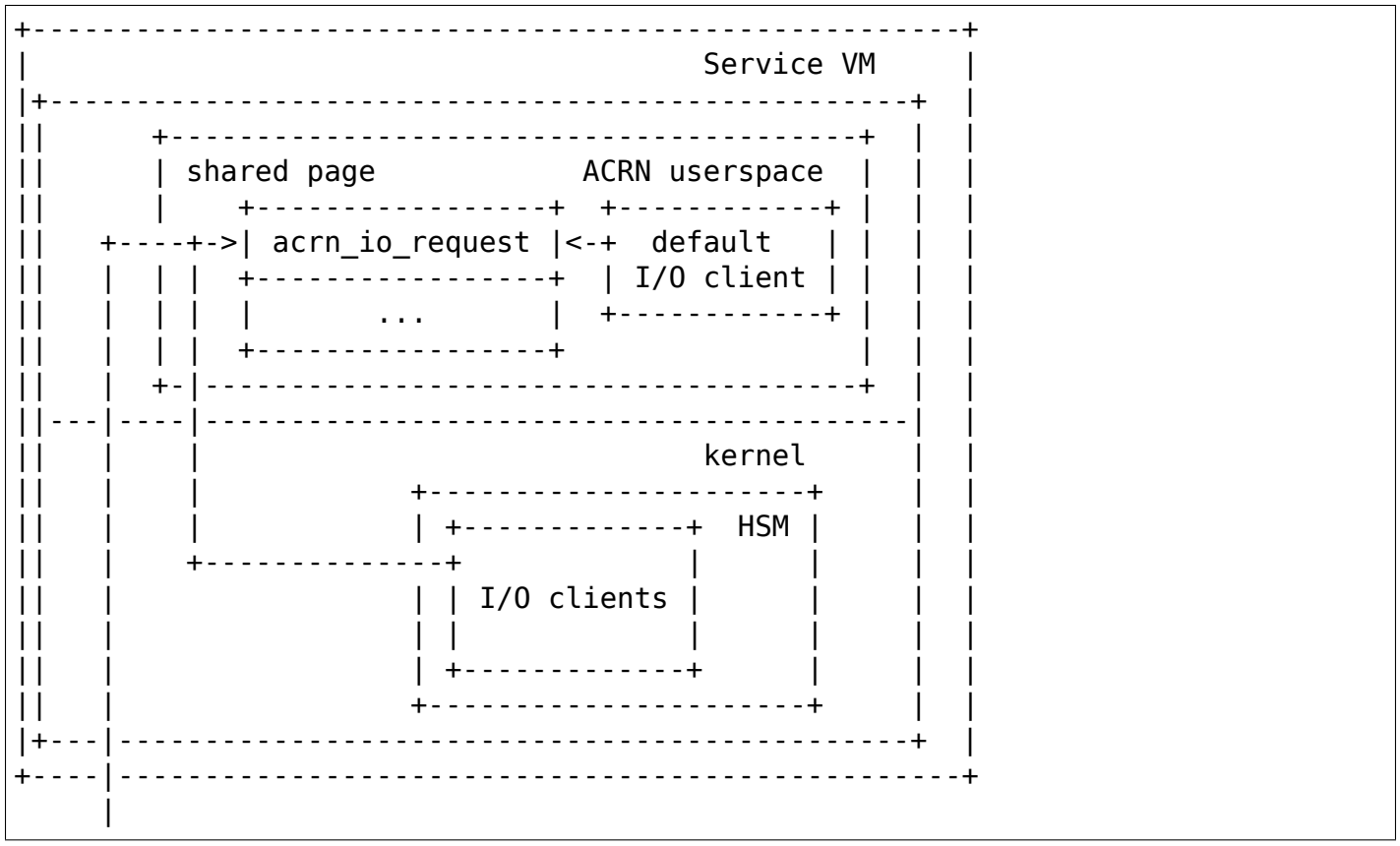
6.2.1 1. I/O request

For each User VM, there is a shared 4-KByte memory region used for I/O requests communication between the hypervisor and Service VM. An I/O request is a 256-byte structure buffer, which is 'struct acrn_io_request', that is filled by an I/O handler of the hypervisor when a trapped I/O access happens in a User VM. ACRN userspace in the Service VM first allocates a 4-KByte page and passes the GPA (Guest Physical Address) of the buffer to the hypervisor. The buffer is used as an array of 16 I/O request slots with each I/O request slot being 256 bytes. This array is indexed by vCPU ID.

6.2.2 2. I/O clients

An I/O client is responsible for handling User VM I/O requests whose accessed GPA falls in a certain range. Multiple I/O clients can be associated with each User VM. There is a special client associated with each User VM, called the default client, that handles all I/O requests that do not fit into the range of any other clients. The ACRN userspace acts as the default client for each User VM.

Below illustration shows the relationship between I/O requests shared buffer, I/O requests and I/O clients.



```
+----|-------------------------------------------------+
|  +-+-----------+                                     |
|  | I/O handler |              ACRN Hypervisor        |
|  +-------------+                                     |
+------------------------------------------------------+
```

6.2.3 3. I/O request state transition

The state transitions of an ACRN I/O request are as follows.

```
FREE -> PENDING -> PROCESSING -> COMPLETE -> FREE -> ...
```

- FREE: this I/O request slot is empty
- PENDING: a valid I/O request is pending in this slot
- PROCESSING: the I/O request is being processed
- COMPLETE: the I/O request has been processed

An I/O request in COMPLETE or FREE state is owned by the hypervisor. HSM and ACRN userspace are in charge of processing the others.

6.2.4 4. Processing flow of I/O requests

a. The I/O handler of the hypervisor will fill an I/O request with PENDING state when a trapped I/O access happens in a User VM.

b. The hypervisor makes an upcall, which is a notification interrupt, to the Service VM.

c. The upcall handler schedules a worker to dispatch I/O requests.

d. The worker looks for the PENDING I/O requests, assigns them to different registered clients based on the address of the I/O accesses, updates their state to PROCESSING, and notifies the corresponding client to handle.

e. The notified client handles the assigned I/O requests.

f. The HSM updates I/O requests states to COMPLETE and notifies the hypervisor of the completion via hypercalls.

6.3 ACRN CPUID bits

A guest VM running on an ACRN hypervisor can check some of its features using CPUID.

ACRN cpuid functions are:

function: 0x40000000

returns:

```
eax = 0x40000010
ebx = 0x4e524341
ecx = 0x4e524341
edx = 0x4e524341
```

Note that this value in ebx, ecx and edx corresponds to the string ”ACRNACRNACRN”. The value in eax corresponds to the maximum cpuid function present in this leaf, and will be updated if more functions are added in the future.

function: define ACRN_CPUID_FEATURES (0x40000001)

returns:

```
ebx, ecx, edx
eax = an OR'ed group of (1 << flag)
```

where `flag` is defined as below:

flag	value	meaning
ACRN_FEATURE_PRIVILEGED_VM	0	guest VM is a privileged VM

function: 0x40000010

returns:

```
ebx, ecx, edx
eax = (Virtual) TSC frequency in kHz.
```

CHAPTER

SEVEN

THE DEFINITIVE SEV GUEST API DOCUMENTATION

7.1 1. General description

The SEV API is a set of ioctls that are used by the guest or hypervisor to get or set a certain aspect of the SEV virtual machine. The ioctls belong to the following classes:

- Hypervisor ioctls: These query and set global attributes which affect the whole SEV firmware. These ioctl are used by platform provisioning tools.
- Guest ioctls: These query and set attributes of the SEV virtual machine.

7.2 2. API description

This section describes ioctls that is used for querying the SEV guest report from the SEV firmware. For each ioctl, the following information is provided along with a description:

Technology:
which SEV technology provides this ioctl. SEV, SEV-ES, SEV-SNP or all.

Type:
hypervisor or guest. The ioctl can be used inside the guest or the hypervisor.

Parameters:
what parameters are accepted by the ioctl.

Returns:
the return value. General error numbers (-ENOMEM, -EINVAL) are not detailed, but errors with specific meanings are.

The guest ioctl should be issued on a file descriptor of the /dev/sev-guest device. The ioctl accepts struct snp_user_guest_request. The input and output structure is specified through the req_data and resp_data field respectively. If the ioctl fails to execute due to a firmware error, then the fw_error code will be set, otherwise fw_error will be set to -1.

The firmware checks that the message sequence counter is one greater than the guests message sequence counter. If guest driver fails to increment message counter (e.g. counter overflow), then -EIO will be returned.

```
struct snp_guest_request_ioctl {
        /* Message version number */
        __u32 msg_version;
```

```
        /* Request and response structure address */
        __u64 req_data;
        __u64 resp_data;

        /* bits[63:32]: VMM error code, bits[31:0] firmware error code (see␣
↪psp-sev.h) */
        union {
                __u64 exitinfo2;
                struct {
                        __u32 fw_error;
                        __u32 vmm_error;
                };
        };
};
```

7.2.1 2.1 SNP_GET_REPORT

Technology
sev-snp

Type
guest ioctl

Parameters (in)
struct snp_report_req

Returns (out)
struct snp_report_resp on success, -negative on error

The SNP_GET_REPORT ioctl can be used to query the attestation report from the SEV-SNP firmware. The ioctl uses the SNP_GUEST_REQUEST (MSG_REPORT_REQ) command provided by the SEV-SNP firmware to query the attestation report.

On success, the snp_report_resp.data will contains the report. The report contain the format described in the SEV-SNP specification. See the SEV-SNP specification for further details.

7.2.2 2.2 SNP_GET_DERIVED_KEY

Technology
sev-snp

Type
guest ioctl

Parameters (in)
struct snp_derived_key_req

Returns (out)
struct snp_derived_key_resp on success, -negative on error

The SNP_GET_DERIVED_KEY ioctl can be used to get a key derive from a root key. The derived key can be used by the guest for any purpose, such as sealing keys or communicating with external entities.

The ioctl uses the SNP_GUEST_REQUEST (MSG_KEY_REQ) command provided by the SEV-SNP firmware to derive the key. See SEV-SNP specification for further details on the various fields passed in the key derivation request.

On success, the snp_derived_key_resp.data contains the derived key value. See the SEV-SNP specification for further details.

7.2.3 2.3 SNP_GET_EXT_REPORT

Technology
sev-snp

Type
guest ioctl

Parameters (in/out)
struct snp_ext_report_req

Returns (out)
struct snp_report_resp on success, -negative on error

The SNP_GET_EXT_REPORT ioctl is similar to the SNP_GET_REPORT. The difference is related to the additional certificate data that is returned with the report. The certificate data returned is being provided by the hypervisor through the SNP_SET_EXT_CONFIG.

The ioctl uses the SNP_GUEST_REQUEST (MSG_REPORT_REQ) command provided by the SEV-SNP firmware to get the attestation report.

On success, the snp_ext_report_resp.data will contain the attestation report and snp_ext_report_req.certs_address will contain the certificate blob. If the length of the blob is smaller than expected then snp_ext_report_req.certs_len will be updated with the expected value.

See GHCB specification for further detail on how to parse the certificate blob.

7.3 3. SEV-SNP CPUID Enforcement

SEV-SNP guests can access a special page that contains a table of CPUID values that have been validated by the PSP as part of the SNP_LAUNCH_UPDATE firmware command. It provides the following assurances regarding the validity of CPUID values:

- Its address is obtained via bootloader/firmware (via CC blob), and those binaries will be measured as part of the SEV-SNP attestation report.
- Its initial state will be encrypted/pvalidated, so attempts to modify it during run-time will result in garbage being written, or #VC exceptions being generated due to changes in validation state if the hypervisor tries to swap the backing page.
- Attempts to bypass PSP checks by the hypervisor by using a normal page, or a non-CPUID encrypted page will change the measurement provided by the SEV-SNP attestation report.
- The CPUID page contents are *not* measured, but attempts to modify the expected contents of a CPUID page as part of guest initialization will be gated by the PSP CPUID enforcement policy checks performed on the page during SNP_LAUNCH_UPDATE, and noticeable later if the guest owner implements their own checks of the CPUID values.

It is important to note that this last assurance is only useful if the kernel has taken care to make use of the SEV-SNP CPUID throughout all stages of boot. Otherwise, guest owner attestation provides no assurance that the kernel wasn't fed incorrect values at some point during boot.

7.3.1 Reference

SEV-SNP and GHCB specification: developer.amd.com/sev

The driver is based on SEV-SNP firmware spec 0.9 and GHCB spec version 2.0.

CHAPTER

EIGHT

TDX GUEST API DOCUMENTATION

8.1 1. General description

The TDX guest driver exposes IOCTL interfaces via the /dev/tdx-guest misc device to allow userspace to get certain TDX guest-specific details.

8.2 2. API description

In this section, for each supported IOCTL, the following information is provided along with a generic description.

Input parameters
: Parameters passed to the IOCTL and related details.

Output
: Details about output data and return value (with details about the non common error values).

8.2.1 2.1 TDX_CMD_GET_REPORT0

Input parameters
: struct tdx_report_req

Output
: Upon successful execution, TDREPORT data is copied to tdx_report_req.tdreport and return 0. Return -EINVAL for invalid operands, -EIO on TDCALL failure or standard error number on other common failures.

The TDX_CMD_GET_REPORT0 IOCTL can be used by the attestation software to get the TDREPORT0 (a.k.a. TDREPORT subtype 0) from the TDX module using TDCALL[TDG.MR.REPORT].

A subtype index is added at the end of this IOCTL CMD to uniquely identify the subtype-specific TDREPORT request. Although the subtype option is mentioned in the TDX Module v1.0 specification, section titled "TDG.MR.REPORT", it is not currently used, and it expects this value to be 0. So to keep the IOCTL implementation simple, the subtype option was not included as part of the input ABI. However, in the future, if the TDX Module supports more than one subtype, a new IOCTL CMD will be created to handle it. To keep the IOCTL naming consistent, a subtype index is added as part of the IOCTL CMD.

8.2.2 Reference

TDX reference material is collected here:

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

The driver is based on TDX module specification v1.0 and TDX GHCI specification v1.0.

CHAPTER

NINE

HYPER-V ENLIGHTENMENTS

9.1 Overview

The Linux kernel contains a variety of code for running as a fully enlightened guest on Microsoft's Hyper-V hypervisor. Hyper-V consists primarily of a bare-metal hypervisor plus a virtual machine management service running in the parent partition (roughly equivalent to KVM and QEMU, for example). Guest VMs run in child partitions. In this documentation, references to Hyper-V usually encompass both the hypervisor and the VMM service without making a distinction about which functionality is provided by which component.

Hyper-V runs on x86/x64 and arm64 architectures, and Linux guests are supported on both. The functionality and behavior of Hyper-V is generally the same on both architectures unless noted otherwise.

9.1.1 Linux Guest Communication with Hyper-V

Linux guests communicate with Hyper-V in four different ways:

- Implicit traps: As defined by the x86/x64 or arm64 architecture, some guest actions trap to Hyper-V. Hyper-V emulates the action and returns control to the guest. This behavior is generally invisible to the Linux kernel.
- Explicit hypercalls: Linux makes an explicit function call to Hyper-V, passing parameters. Hyper-V performs the requested action and returns control to the caller. Parameters are passed in processor registers or in memory shared between the Linux guest and Hyper-V. On x86/x64, hypercalls use a Hyper-V specific calling sequence. On arm64, hypercalls use the ARM standard SMCCC calling sequence.
- Synthetic register access: Hyper-V implements a variety of synthetic registers. On x86/x64 these registers appear as MSRs in the guest, and the Linux kernel can read or write these MSRs using the normal mechanisms defined by the x86/x64 architecture. On arm64, these synthetic registers must be accessed using explicit hypercalls.
- VMbus: VMbus is a higher-level software construct that is built on the other 3 mechanisms. It is a message passing interface between the Hyper-V host and the Linux guest. It uses memory that is shared between Hyper-V and the guest, along with various signaling mechanisms.

The first three communication mechanisms are documented in the Hyper-V Top Level Functional Spec (TLFS). The TLFS describes general Hyper-V functionality and provides details on the hypercalls and synthetic registers. The TLFS is currently written for the x86/x64 architecture only.

VMbus is not documented. This documentation provides a high-level overview of VMbus and how it works, but the details can be discerned only from the code.

9.1.2 Sharing Memory

Many aspects are communication between Hyper-V and Linux are based on sharing memory. Such sharing is generally accomplished as follows:

- Linux allocates memory from its physical address space using standard Linux mechanisms.
- Linux tells Hyper-V the guest physical address (GPA) of the allocated memory. Many shared areas are kept to 1 page so that a single GPA is sufficient. Larger shared areas require a list of GPAs, which usually do not need to be contiguous in the guest physical address space. How Hyper-V is told about the GPA or list of GPAs varies. In some cases, a single GPA is written to a synthetic register. In other cases, a GPA or list of GPAs is sent in a VMbus message.
- Hyper-V translates the GPAs into ”real” physical memory addresses, and creates a virtual mapping that it can use to access the memory.
- Linux can later revoke sharing it has previously established by telling Hyper-V to set the shared GPA to zero.

Hyper-V operates with a page size of 4 Kbytes. GPAs communicated to Hyper-V may be in the form of page numbers, and always describe a range of 4 Kbytes. Since the Linux guest page size on x86/x64 is also 4 Kbytes, the mapping from guest page to Hyper-V page is 1-to-1. On arm64, Hyper-V supports guests with 4/16/64 Kbyte pages as defined by the arm64 architecture. If Linux is using 16 or 64 Kbyte pages, Linux code must be careful to communicate with Hyper-V only in terms of 4 Kbyte pages. HV_HYP_PAGE_SIZE and related macros are used in code that communicates with Hyper-V so that it works correctly in all configurations.

As described in the TLFS, a few memory pages shared between Hyper-V and the Linux guest are ”overlay” pages. With overlay pages, Linux uses the usual approach of allocating guest memory and telling Hyper-V the GPA of the allocated memory. But Hyper-V then replaces that physical memory page with a page it has allocated, and the original physical memory page is no longer accessible in the guest VM. Linux may access the memory normally as if it were the memory that it originally allocated. The ”overlay” behavior is visible only because the contents of the page (as seen by Linux) change at the time that Linux originally establishes the sharing and the overlay page is inserted. Similarly, the contents change if Linux revokes the sharing, in which case Hyper-V removes the overlay page, and the guest page originally allocated by Linux becomes visible again.

Before Linux does a kexec to a kdump kernel or any other kernel, memory shared with Hyper-V should be revoked. Hyper-V could modify a shared page or remove an overlay page after the new kernel is using the page for a different purpose, corrupting the new kernel. Hyper-V does not provide a single ”set everything” operation to guest VMs, so Linux code must individually revoke all sharing before doing kexec. See hv_kexec_handler() and hv_crash_handler(). But the crash/panic path still has holes in cleanup because some shared pages are set using per-CPU synthetic registers and there's no mechanism to revoke the shared pages for CPUs other than the CPU running the panic path.

9.1.3 CPU Management

Hyper-V does not have a ability to hot-add or hot-remove a CPU from a running VM. However, Windows Server 2019 Hyper-V and earlier versions may provide guests with ACPI tables that indicate more CPUs than are actually present in the VM. As is normal, Linux treats these additional CPUs as potential hot-add CPUs, and reports them as such even though Hyper-V will never actually hot-add them. Starting in Windows Server 2022 Hyper-V, the ACPI tables reflect only the CPUs actually present in the VM, so Linux does not report any hot-add CPUs.

A Linux guest CPU may be taken offline using the normal Linux mechanisms, provided no VMbus channel interrupts are assigned to the CPU. See the section on VMbus Interrupts for more details on how VMbus channel interrupts can be re-assigned to permit taking a CPU offline.

9.1.4 32-bit and 64-bit

On x86/x64, Hyper-V supports 32-bit and 64-bit guests, and Linux will build and run in either version. While the 32-bit version is expected to work, it is used rarely and may suffer from undetected regressions.

On arm64, Hyper-V supports only 64-bit guests.

9.1.5 Endian-ness

All communication between Hyper-V and guest VMs uses Little-Endian format on both x86/x64 and arm64. Big-endian format on arm64 is not supported by Hyper-V, and Linux code does not use endian-ness macros when accessing data shared with Hyper-V.

9.1.6 Versioning

Current Linux kernels operate correctly with older versions of Hyper-V back to Windows Server 2012 Hyper-V. Support for running on the original Hyper-V release in Windows Server 2008/2008 R2 has been removed.

A Linux guest on Hyper-V outputs in dmesg the version of Hyper-V it is running on. This version is in the form of a Windows build number and is for display purposes only. Linux code does not test this version number at runtime to determine available features and functionality. Hyper-V indicates feature/function availability via flags in synthetic MSRs that Hyper-V provides to the guest, and the guest code tests these flags.

VMbus has its own protocol version that is negotiated during the initial VMbus connection from the guest to Hyper-V. This version number is also output to dmesg during boot. This version number is checked in a few places in the code to determine if specific functionality is present.

Furthermore, each synthetic device on VMbus also has a protocol version that is separate from the VMbus protocol version. Device drivers for these synthetic devices typically negotiate the device protocol version, and may test that protocol version to determine if specific device functionality is present.

9.1.7 Code Packaging

Hyper-V related code appears in the Linux kernel code tree in three main areas:

1. drivers/hv
2. arch/x86/hyperv and arch/arm64/hyperv
3. individual device driver areas such as drivers/scsi, drivers/net, drivers/clocksource, etc.

A few miscellaneous files appear elsewhere. See the full list under "Hyper-V/Azure CORE AND DRIVERS" and "DRM DRIVER FOR HYPERV SYNTHETIC VIDEO DEVICE" in the MAINTAINERS file.

The code in #1 and #2 is built only when CONFIG_HYPERV is set. Similarly, the code for most Hyper-V related drivers is built only when CONFIG_HYPERV is set.

Most Hyper-V related code in #1 and #3 can be built as a module. The architecture specific code in #2 must be built-in. Also, drivers/hv/hv_common.c is low-level code that is common across architectures and must be built-in.

9.2 VMbus

VMbus is a software construct provided by Hyper-V to guest VMs. It consists of a control path and common facilities used by synthetic devices that Hyper-V presents to guest VMs. The control path is used to offer synthetic devices to the guest VM and, in some cases, to rescind those devices. The common facilities include software channels for communicating between the device driver in the guest VM and the synthetic device implementation that is part of Hyper-V, and signaling primitives to allow Hyper-V and the guest to interrupt each other.

VMbus is modeled in Linux as a bus, with the expected /sys/bus/vmbus entry in a running Linux guest. The VMbus driver (drivers/hv/vmbus_drv.c) establishes the VMbus control path with the Hyper-V host, then registers itself as a Linux bus driver. It implements the standard bus functions for adding and removing devices to/from the bus.

Most synthetic devices offered by Hyper-V have a corresponding Linux device driver. These devices include:

- SCSI controller
- NIC
- Graphics frame buffer
- Keyboard
- Mouse
- PCI device pass-thru
- Heartbeat
- Time Sync
- Shutdown
- Memory balloon
- Key/Value Pair (KVP) exchange with Hyper-V

- Hyper-V online backup (a.k.a. VSS)

Guest VMs may have multiple instances of the synthetic SCSI controller, synthetic NIC, and PCI pass-thru devices. Other synthetic devices are limited to a single instance per VM. Not listed above are a small number of synthetic devices offered by Hyper-V that are used only by Windows guests and for which Linux does not have a driver.

Hyper-V uses the terms ”VSP” and ”VSC” in describing synthetic devices. ”VSP” refers to the Hyper-V code that implements a particular synthetic device, while ”VSC” refers to the driver for the device in the guest VM. For example, the Linux driver for the synthetic NIC is referred to as ”netvsc” and the Linux driver for the synthetic SCSI controller is ”storvsc”. These drivers contain functions with names like ”storvsc_connect_to_vsp”.

9.2.1 VMbus channels

An instance of a synthetic device uses VMbus channels to communicate between the VSP and the VSC. Channels are bi-directional and used for passing messages. Most synthetic devices use a single channel, but the synthetic SCSI controller and synthetic NIC may use multiple channels to achieve higher performance and greater parallelism.

Each channel consists of two ring buffers. These are classic ring buffers from a university data structures textbook. If the read and writes pointers are equal, the ring buffer is considered to be empty, so a full ring buffer always has at least one byte unused. The ”in” ring buffer is for messages from the Hyper-V host to the guest, and the ”out” ring buffer is for messages from the guest to the Hyper-V host. In Linux, the ”in” and ”out” designations are as viewed by the guest side. The ring buffers are memory that is shared between the guest and the host, and they follow the standard paradigm where the memory is allocated by the guest, with the list of GPAs that make up the ring buffer communicated to the host. Each ring buffer consists of a header page (4 Kbytes) with the read and write indices and some control flags, followed by the memory for the actual ring. The size of the ring is determined by the VSC in the guest and is specific to each synthetic device. The list of GPAs making up the ring is communicated to the Hyper-V host over the VMbus control path as a GPA Descriptor List (GPADL). See function vmbus_establish_gpadl().

Each ring buffer is mapped into contiguous Linux kernel virtual space in three parts: 1) the 4 Kbyte header page, 2) the memory that makes up the ring itself, and 3) a second mapping of the memory that makes up the ring itself. Because (2) and (3) are contiguous in kernel virtual space, the code that copies data to and from the ring buffer need not be concerned with ring buffer wrap-around. Once a copy operation has completed, the read or write index may need to be reset to point back into the first mapping, but the actual data copy does not need to be broken into two parts. This approach also allows complex data structures to be easily accessed directly in the ring without handling wrap-around.

On arm64 with page sizes > 4 Kbytes, the header page must still be passed to Hyper-V as a 4 Kbyte area. But the memory for the actual ring must be aligned to PAGE_SIZE and have a size that is a multiple of PAGE_SIZE so that the duplicate mapping trick can be done. Hence a portion of the header page is unused and not communicated to Hyper-V. This case is handled by vmbus_establish_gpadl().

Hyper-V enforces a limit on the aggregate amount of guest memory that can be shared with the host via GPADLs. This limit ensures that a rogue guest can't force the consumption of excessive host resources. For Windows Server 2019 and later, this limit is approximately 1280 Mbytes. For versions prior to Windows Server 2019, the limit is approximately 384 Mbytes.

9.2.2 VMbus messages

All VMbus messages have a standard header that includes the message length, the offset of the message payload, some flags, and a transactionID. The portion of the message after the header is unique to each VSP/VSC pair.

Messages follow one of two patterns:

- Unidirectional: Either side sends a message and does not expect a response message
- Request/response: One side (usually the guest) sends a message and expects a response

The transactionID (a.k.a. "requestID") is for matching requests & responses. Some synthetic devices allow multiple requests to be in- flight simultaneously, so the guest specifies a transactionID when sending a request. Hyper-V sends back the same transactionID in the matching response.

Messages passed between the VSP and VSC are control messages. For example, a message sent from the storvsc driver might be "execute this SCSI command". If a message also implies some data transfer between the guest and the Hyper-V host, the actual data to be transferred may be embedded with the control message, or it may be specified as a separate data buffer that the Hyper-V host will access as a DMA operation. The former case is used when the size of the data is small and the cost of copying the data to and from the ring buffer is minimal. For example, time sync messages from the Hyper-V host to the guest contain the actual time value. When the data is larger, a separate data buffer is used. In this case, the control message contains a list of GPAs that describe the data buffer. For example, the storvsc driver uses this approach to specify the data buffers to/from which disk I/O is done.

Three functions exist to send VMbus messages:

1. vmbus_sendpacket(): Control-only messages and messages with embedded data -- no GPAs
2. vmbus_sendpacket_pagebuffer(): Message with list of GPAs identifying data to transfer. An offset and length is associated with each GPA so that multiple discontinuous areas of guest memory can be targeted.
3. vmbus_sendpacket_mpb_desc(): Message with list of GPAs identifying data to transfer. A single offset and length is associated with a list of GPAs. The GPAs must describe a single logical area of guest memory to be targeted.

Historically, Linux guests have trusted Hyper-V to send well-formed and valid messages, and Linux drivers for synthetic devices did not fully validate messages. With the introduction of processor technologies that fully encrypt guest memory and that allow the guest to not trust the hypervisor (AMD SNP-SEV, Intel TDX), trusting the Hyper-V host is no longer a valid assumption. The drivers for VMbus synthetic devices are being updated to fully validate any values read from memory that is shared with Hyper-V, which includes messages from VMbus devices. To facilitate such validation, messages read by the guest from the "in" ring buffer are copied to a temporary buffer that is not shared with Hyper-V. Validation is performed in this temporary buffer without the risk of Hyper-V maliciously modifying the message after it is validated but before it is used.

9.2.3 VMbus interrupts

VMbus provides a mechanism for the guest to interrupt the host when the guest has queued new messages in a ring buffer. The host expects that the guest will send an interrupt only when an ”out” ring buffer transitions from empty to non-empty. If the guest sends interrupts at other times, the host deems such interrupts to be unnecessary. If a guest sends an excessive number of unnecessary interrupts, the host may throttle that guest by suspending its execution for a few seconds to prevent a denial-of-service attack.

Similarly, the host will interrupt the guest when it sends a new message on the VMbus control path, or when a VMbus channel ”in” ring buffer transitions from empty to non-empty. Each CPU in the guest may receive VMbus interrupts, so they are best modeled as per-CPU interrupts in Linux. This model works well on arm64 where a single per-CPU IRQ is allocated for VMbus. Since x86/x64 lacks support for per-CPU IRQs, an x86 interrupt vector is statically allocated (see HYPERVISOR_CALLBACK_VECTOR) across all CPUs and explicitly coded to call the VMbus interrupt service routine. These interrupts are visible in /proc/interrupts on the ”HYP” line.

The guest CPU that a VMbus channel will interrupt is selected by the guest when the channel is created, and the host is informed of that selection. VMbus devices are broadly grouped into two categories:

1. ”Slow” devices that need only one VMbus channel. The devices (such as keyboard, mouse, heartbeat, and timesync) generate relatively few interrupts. Their VMbus channels are all assigned to interrupt the VMBUS_CONNECT_CPU, which is always CPU 0.

2. ”High speed” devices that may use multiple VMbus channels for higher parallelism and performance. These devices include the synthetic SCSI controller and synthetic NIC. Their VMbus channels interrupts are assigned to CPUs that are spread out among the available CPUs in the VM so that interrupts on multiple channels can be processed in parallel.

The assignment of VMbus channel interrupts to CPUs is done in the function init_vp_index(). This assignment is done outside of the normal Linux interrupt affinity mechanism, so the interrupts are neither ”unmanaged” nor ”managed” interrupts.

The CPU that a VMbus channel will interrupt can be seen in /sys/bus/vmbus/devices/<deviceGUID>/ channels/<channelRelID>/cpu. When running on later versions of Hyper-V, the CPU can be changed by writing a new value to this sysfs entry. Because the interrupt assignment is done outside of the normal Linux affinity mechanism, there are no entries in /proc/irq corresponding to individual VMbus channel interrupts.

An online CPU in a Linux guest may not be taken offline if it has VMbus channel interrupts assigned to it. Any such channel interrupts must first be manually reassigned to another CPU as described above. When no channel interrupts are assigned to the CPU, it can be taken offline.

When a guest CPU receives a VMbus interrupt from the host, the function vmbus_isr() handles the interrupt. It first checks for channel interrupts by calling vmbus_chan_sched(), which looks at a bitmap setup by the host to determine which channels have pending interrupts on this CPU. If multiple channels have pending interrupts for this CPU, they are processed sequentially. When all channel interrupts have been processed, vmbus_isr() checks for and processes any message received on the VMbus control path.

The VMbus channel interrupt handling code is designed to work correctly even if an interrupt is received on a CPU other than the CPU assigned to the channel. Specifically, the code does not use CPU-based exclusion for correctness. In normal operation, Hyper-V will interrupt the assigned CPU. But when the CPU assigned to a channel is being changed via sysfs, the guest doesn't know exactly when Hyper-V will make the transition. The code must work correctly

even if there is a time lag before Hyper-V starts interrupting the new CPU. See comments in target_cpu_store().

9.2.4 VMbus device creation/deletion

Hyper-V and the Linux guest have a separate message-passing path that is used for synthetic device creation and deletion. This path does not use a VMbus channel. See vmbus_post_msg() and vmbus_on_msg_dpc().

The first step is for the guest to connect to the generic Hyper-V VMbus mechanism. As part of establishing this connection, the guest and Hyper-V agree on a VMbus protocol version they will use. This negotiation allows newer Linux kernels to run on older Hyper-V versions, and vice versa.

The guest then tells Hyper-V to ”send offers”. Hyper-V sends an offer message to the guest for each synthetic device that the VM is configured to have. Each VMbus device type has a fixed GUID known as the ”class ID”, and each VMbus device instance is also identified by a GUID. The offer message from Hyper-V contains both GUIDs to uniquely (within the VM) identify the device. There is one offer message for each device instance, so a VM with two synthetic NICs will get two offers messages with the NIC class ID. The ordering of offer messages can vary from boot-to-boot and must not be assumed to be consistent in Linux code. Offer messages may also arrive long after Linux has initially booted because Hyper-V supports adding devices, such as synthetic NICs, to running VMs. A new offer message is processed by vmbus_process_offer(), which indirectly invokes vmbus_add_channel_work().

Upon receipt of an offer message, the guest identifies the device type based on the class ID, and invokes the correct driver to set up the device. Driver/device matching is performed using the standard Linux mechanism.

The device driver probe function opens the primary VMbus channel to the corresponding VSP. It allocates guest memory for the channel ring buffers and shares the ring buffer with the Hyper-V host by giving the host a list of GPAs for the ring buffer memory. See vmbus_establish_gpadl().

Once the ring buffer is set up, the device driver and VSP exchange setup messages via the primary channel. These messages may include negotiating the device protocol version to be used between the Linux VSC and the VSP on the Hyper-V host. The setup messages may also include creating additional VMbus channels, which are somewhat mis-named as ”sub-channels” since they are functionally equivalent to the primary channel once they are created.

Finally, the device driver may create entries in /dev as with any device driver.

The Hyper-V host can send a ”rescind” message to the guest to remove a device that was previously offered. Linux drivers must handle such a rescind message at any time. Rescinding a device invokes the device driver ”remove” function to cleanly shut down the device and remove it. Once a synthetic device is rescinded, neither Hyper-V nor Linux retains any state about its previous existence. Such a device might be re-added later, in which case it is treated as an entirely new device. See vmbus_onoffer_rescind().

9.3 Clocks and Timers

9.3.1 arm64

On arm64, Hyper-V virtualizes the ARMv8 architectural system counter and timer. Guest VMs use this virtualized hardware as the Linux clocksource and clockevents via the standard arm_arch_timer.c driver, just as they would on bare metal. Linux vDSO support for the architectural system counter is functional in guest VMs on Hyper-V. While Hyper-V also provides a synthetic system clock and four synthetic per-CPU timers as described in the TLFS, they are not used by the Linux kernel in a Hyper-V guest on arm64. However, older versions of Hyper-V for arm64 only partially virtualize the ARMv8 architectural timer, such that the timer does not generate interrupts in the VM. Because of this limitation, running current Linux kernel versions on these older Hyper-V versions requires an out-of-tree patch to use the Hyper-V synthetic clocks/timers instead.

9.3.2 x86/x64

On x86/x64, Hyper-V provides guest VMs with a synthetic system clock and four synthetic per-CPU timers as described in the TLFS. Hyper-V also provides access to the virtualized TSC via the RDTSC and related instructions. These TSC instructions do not trap to the hypervisor and so provide excellent performance in a VM. Hyper-V performs TSC calibration, and provides the TSC frequency to the guest VM via a synthetic MSR. Hyper-V initialization code in Linux reads this MSR to get the frequency, so it skips TSC calibration and sets tsc_reliable. Hyper-V provides virtualized versions of the PIT (in Hyper-V Generation 1 VMs only), local APIC timer, and RTC. Hyper-V does not provide a virtualized HPET in guest VMs.

The Hyper-V synthetic system clock can be read via a synthetic MSR, but this access traps to the hypervisor. As a faster alternative, the guest can configure a memory page to be shared between the guest and the hypervisor. Hyper-V populates this memory page with a 64-bit scale value and offset value. To read the synthetic clock value, the guest reads the TSC and then applies the scale and offset as described in the Hyper-V TLFS. The resulting value advances at a constant 10 MHz frequency. In the case of a live migration to a host with a different TSC frequency, Hyper-V adjusts the scale and offset values in the shared page so that the 10 MHz frequency is maintained.

Starting with Windows Server 2022 Hyper-V, Hyper-V uses hardware support for TSC frequency scaling to enable live migration of VMs across Hyper-V hosts where the TSC frequency may be different. When a Linux guest detects that this Hyper-V functionality is available, it prefers to use Linux's standard TSC-based clocksource. Otherwise, it uses the clocksource for the Hyper-V synthetic system clock implemented via the shared page (identified as ”hyperv_clocksource_tsc_page”).

The Hyper-V synthetic system clock is available to user space via vDSO, and gettimeofday() and related system calls can execute entirely in user space. The vDSO is implemented by mapping the shared page with scale and offset values into user space. User space code performs the same algorithm of reading the TSC and applying the scale and offset to get the constant 10 MHz clock.

Linux clockevents are based on Hyper-V synthetic timer 0. While Hyper-V offers 4 synthetic timers for each CPU, Linux only uses timer 0. Interrupts from stimer0 are recorded on the ”HVS” line in /proc/interrupts. Clockevents based on the virtualized PIT and local APIC timer also work, but the Hyper-V synthetic timer is preferred.

The driver for the Hyper-V synthetic system clock and timers is drivers/clocksource/hyperv_timer.c.

9.4 PCI pass-thru devices

In a Hyper-V guest VM, PCI pass-thru devices (also called virtual PCI devices, or vPCI devices) are physical PCI devices that are mapped directly into the VM's physical address space. Guest device drivers can interact directly with the hardware without intermediation by the host hypervisor. This approach provides higher bandwidth access to the device with lower latency, compared with devices that are virtualized by the hypervisor. The device should appear to the guest just as it would when running on bare metal, so no changes are required to the Linux device drivers for the device.

Hyper-V terminology for vPCI devices is "Discrete Device Assignment" (DDA). Public documentation for Hyper-V DDA is available here: DDA

DDA is typically used for storage controllers, such as NVMe, and for GPUs. A similar mechanism for NICs is called SR-IOV and produces the same benefits by allowing a guest device driver to interact directly with the hardware. See Hyper-V public documentation here: SR-IOV

This discussion of vPCI devices includes DDA and SR-IOV devices.

9.4.1 Device Presentation

Hyper-V provides full PCI functionality for a vPCI device when it is operating, so the Linux device driver for the device can be used unchanged, provided it uses the correct Linux kernel APIs for accessing PCI config space and for other integration with Linux. But the initial detection of the PCI device and its integration with the Linux PCI subsystem must use Hyper-V specific mechanisms. Consequently, vPCI devices on Hyper-V have a dual identity. They are initially presented to Linux guests as VMBus devices via the standard VMBus "offer" mechanism, so they have a VMBus identity and appear under /sys/bus/vmbus/devices. The VMBus vPCI driver in Linux at drivers/pci/controller/pci-hyperv.c handles a newly introduced vPCI device by fabricating a PCI bus topology and creating all the normal PCI device data structures in Linux that would exist if the PCI device were discovered via ACPI on a bare- metal system. Once those data structures are set up, the device also has a normal PCI identity in Linux, and the normal Linux device driver for the vPCI device can function as if it were running in Linux on bare-metal. Because vPCI devices are presented dynamically through the VMBus offer mechanism, they do not appear in the Linux guest's ACPI tables. vPCI devices may be added to a VM or removed from a VM at any time during the life of the VM, and not just during initial boot.

With this approach, the vPCI device is a VMBus device and a PCI device at the same time. In response to the VMBus offer message, the hv_pci_probe() function runs and establishes a VMBus connection to the vPCI VSP on the Hyper-V host. That connection has a single VMBus channel. The channel is used to exchange messages with the vPCI VSP for the purpose of setting up and configuring the vPCI device in Linux. Once the device is fully configured in Linux as a PCI device, the VMBus channel is used only if Linux changes the vCPU to be interrupted in the guest, or if the vPCI device is removed from the VM while the VM is running. The ongoing operation of the device happens directly between the Linux device driver for the device and the hardware, with VMBus and the VMBus channel playing no role.

9.4.2 PCI Device Setup

PCI device setup follows a sequence that Hyper-V originally created for Windows guests, and that can be ill-suited for Linux guests due to differences in the overall structure of the Linux PCI subsystem compared with Windows. Nonetheless, with a bit of hackery in the Hyper-V virtual PCI driver for Linux, the virtual PCI device is setup in Linux so that generic Linux PCI subsystem code and the Linux driver for the device "just work".

Each vPCI device is set up in Linux to be in its own PCI domain with a host bridge. The PCI domainID is derived from bytes 4 and 5 of the instance GUID assigned to the VMBus vPCI device. The Hyper-V host does not guarantee that these bytes are unique, so hv_pci_probe() has an algorithm to resolve collisions. The collision resolution is intended to be stable across reboots of the same VM so that the PCI domainIDs don't change, as the domainID appears in the user space configuration of some devices.

hv_pci_probe() allocates a guest MMIO range to be used as PCI config space for the device. This MMIO range is communicated to the Hyper-V host over the VMBus channel as part of telling the host that the device is ready to enter d0. See hv_pci_enter_d0(). When the guest subsequently accesses this MMIO range, the Hyper-V host intercepts the accesses and maps them to the physical device PCI config space.

hv_pci_probe() also gets BAR information for the device from the Hyper-V host, and uses this information to allocate MMIO space for the BARs. That MMIO space is then setup to be associated with the host bridge so that it works when generic PCI subsystem code in Linux processes the BARs.

Finally, hv_pci_probe() creates the root PCI bus. At this point the Hyper-V virtual PCI driver hackery is done, and the normal Linux PCI machinery for scanning the root bus works to detect the device, to perform driver matching, and to initialize the driver and device.

9.4.3 PCI Device Removal

A Hyper-V host may initiate removal of a vPCI device from a guest VM at any time during the life of the VM. The removal is instigated by an admin action taken on the Hyper-V host and is not under the control of the guest OS.

A guest VM is notified of the removal by an unsolicited "Eject" message sent from the host to the guest over the VMBus channel associated with the vPCI device. Upon receipt of such a message, the Hyper-V virtual PCI driver in Linux asynchronously invokes Linux kernel PCI subsystem calls to shutdown and remove the device. When those calls are complete, an "Ejection Complete" message is sent back to Hyper-V over the VMBus channel indicating that the device has been removed. At this point, Hyper-V sends a VMBus rescind message to the Linux guest, which the VMBus driver in Linux processes by removing the VMBus identity for the device. Once that processing is complete, all vestiges of the device having been present are gone from the Linux kernel. The rescind message also indicates to the guest that Hyper-V has stopped providing support for the vPCI device in the guest. If the guest were to attempt to access that device's MMIO space, it would be an invalid reference. Hypercalls affecting the device return errors, and any further messages sent in the VMBus channel are ignored.

After sending the Eject message, Hyper-V allows the guest VM 60 seconds to cleanly shutdown the device and respond with Ejection Complete before sending the VMBus rescind message. If for any reason the Eject steps don't complete within the allowed 60 seconds, the Hyper-V host forcibly performs the rescind steps, which will likely result in cascading errors in the

guest because the device is now no longer present from the guest standpoint and accessing the device MMIO space will fail.

Because ejection is asynchronous and can happen at any point during the guest VM lifecycle, proper synchronization in the Hyper-V virtual PCI driver is very tricky. Ejection has been observed even before a newly offered vPCI device has been fully setup. The Hyper-V virtual PCI driver has been updated several times over the years to fix race conditions when ejections happen at inopportune times. Care must be taken when modifying this code to prevent re-introducing such problems. See comments in the code.

9.4.4 Interrupt Assignment

The Hyper-V virtual PCI driver supports vPCI devices using MSI, multi-MSI, or MSI-X. Assigning the guest vCPU that will receive the interrupt for a particular MSI or MSI-X message is complex because of the way the Linux setup of IRQs maps onto the Hyper-V interfaces. For the single-MSI and MSI-X cases, Linux calls hv_compse_msi_msg() twice, with the first call containing a dummy vCPU and the second call containing the real vCPU. Furthermore, hv_irq_unmask() is finally called (on x86) or the GICD registers are set (on arm64) to specify the real vCPU again. Each of these three calls interact with Hyper-V, which must decide which physical CPU should receive the interrupt before it is forwarded to the guest VM. Unfortunately, the Hyper-V decision-making process is a bit limited, and can result in concentrating the physical interrupts on a single CPU, causing a performance bottleneck. See details about how this is resolved in the extensive comment above the function hv_compose_msi_req_get_cpu().

The Hyper-V virtual PCI driver implements the irq_chip.irq_compose_msi_msg function as hv_compose_msi_msg(). Unfortunately, on Hyper-V the implementation requires sending a VMBus message to the Hyper-V host and awaiting an interrupt indicating receipt of a reply message. Since irq_chip.irq_compose_msi_msg can be called with IRQ locks held, it doesn't work to do the normal sleep until awakened by the interrupt. Instead hv_compose_msi_msg() must send the VMBus message, and then poll for the completion message. As further complexity, the vPCI device could be ejected/rescinded while the polling is in progress, so this scenario must be detected as well. See comments in the code regarding this very tricky area.

Most of the code in the Hyper-V virtual PCI driver (pci- hyperv.c) applies to Hyper-V and Linux guests running on x86 and on arm64 architectures. But there are differences in how interrupt assignments are managed. On x86, the Hyper-V virtual PCI driver in the guest must make a hypercall to tell Hyper-V which guest vCPU should be interrupted by each MSI/MSI-X interrupt, and the x86 interrupt vector number that the x86_vector IRQ domain has picked for the interrupt. This hypercall is made by hv_arch_irq_unmask(). On arm64, the Hyper-V virtual PCI driver manages the allocation of an SPI for each MSI/MSI-X interrupt. The Hyper-V virtual PCI driver stores the allocated SPI in the architectural GICD registers, which Hyper-V emulates, so no hypercall is necessary as with x86. Hyper-V does not support using LPIs for vPCI devices in arm64 guest VMs because it does not emulate a GICv3 ITS.

The Hyper-V virtual PCI driver in Linux supports vPCI devices whose drivers create managed or unmanaged Linux IRQs. If the smp_affinity for an unmanaged IRQ is updated via the /proc/irq interface, the Hyper-V virtual PCI driver is called to tell the Hyper-V host to change the interrupt targeting and everything works properly. However, on x86 if the x86_vector IRQ domain needs to reassign an interrupt vector due to running out of vectors on a CPU, there's no path to inform the Hyper-V host of the change, and things break. Fortunately, guest VMs operate in a constrained device environment where using all the vectors on a CPU doesn't happen. Since such a problem is only a theoretical concern rather than a practical concern, it has been left unaddressed.

9.4.5 DMA

By default, Hyper-V pins all guest VM memory in the host when the VM is created, and programs the physical IOMMU to allow the VM to have DMA access to all its memory. Hence it is safe to assign PCI devices to the VM, and allow the guest operating system to program the DMA transfers. The physical IOMMU prevents a malicious guest from initiating DMA to memory belonging to the host or to other VMs on the host. From the Linux guest standpoint, such DMA transfers are in "direct" mode since Hyper-V does not provide a virtual IOMMU in the guest.

Hyper-V assumes that physical PCI devices always perform cache-coherent DMA. When running on x86, this behavior is required by the architecture. When running on arm64, the architecture allows for both cache-coherent and non-cache-coherent devices, with the behavior of each device specified in the ACPI DSDT. But when a PCI device is assigned to a guest VM, that device does not appear in the DSDT, so the Hyper-V VMBus driver propagates cache-coherency information from the VMBus node in the ACPI DSDT to all VMBus devices, including vPCI devices (since they have a dual identity as a VMBus device and as a PCI device). See vmbus_dma_configure(). Current Hyper-V versions always indicate that the VMBus is cache coherent, so vPCI devices on arm64 always get marked as cache coherent and the CPU does not perform any sync operations as part of dma_map/unmap_*() calls.

9.4.6 vPCI protocol versions

As previously described, during vPCI device setup and teardown messages are passed over a VMBus channel between the Hyper-V host and the Hyper-v vPCI driver in the Linux guest. Some messages have been revised in newer versions of Hyper-V, so the guest and host must agree on the vPCI protocol version to be used. The version is negotiated when communication over the VMBus channel is first established. See hv_pci_protocol_negotiation(). Newer versions of the protocol extend support to VMs with more than 64 vCPUs, and provide additional information about the vPCI device, such as the guest virtual NUMA node to which it is most closely affined in the underlying hardware.

9.4.7 Guest NUMA node affinity

When the vPCI protocol version provides it, the guest NUMA node affinity of the vPCI device is stored as part of the Linux device information for subsequent use by the Linux driver. See hv_pci_assign_numa_node(). If the negotiated protocol version does not support the host providing NUMA affinity information, the Linux guest defaults the device NUMA node to 0. But even when the negotiated protocol version includes NUMA affinity information, the ability of the host to provide such information depends on certain host configuration options. If the guest receives NUMA node value "0", it could mean NUMA node 0, or it could mean "no information is available". Unfortunately it is not possible to distinguish the two cases from the guest side.

9.4.8 PCI config space access in a CoCo VM

Linux PCI device drivers access PCI config space using a standard set of functions provided by the Linux PCI subsystem. In Hyper-V guests these standard functions map to functions hv_pcifront_read_config() and hv_pcifront_write_config() in the Hyper-V virtual PCI driver. In normal VMs, these hv_pcifront_*() functions directly access the PCI config space, and the accesses trap to Hyper-V to be handled. But in CoCo VMs, memory encryption prevents Hyper-V from reading the guest instruction stream to emulate the access, so the hv_pcifront_*() functions must invoke hypercalls with explicit arguments describing the access to be made.

9.4.9 Config Block back-channel

The Hyper-V host and Hyper-V virtual PCI driver in Linux together implement a non-standard back-channel communication path between the host and guest. The back-channel path uses messages sent over the VMBus channel associated with the vPCI device. The functions hyperv_read_cfg_blk() and hyperv_write_cfg_blk() are the primary interfaces provided to other parts of the Linux kernel. As of this writing, these interfaces are used only by the Mellanox mlx5 driver to pass diagnostic data to a Hyper-V host running in the Azure public cloud. The functions hyperv_read_cfg_blk() and hyperv_write_cfg_blk() are implemented in a separate module (pci-hyperv-intf.c, under CONFIG_PCI_HYPERV_INTERFACE) that effectively stubs them out when running in non-Hyper-V environments.

BIBLIOGRAPHY

[white-paper] https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[api-spec] https://support.amd.com/TechDocs/55766_SEV-KM_API_Specification.pdf

[amd-apm] https://support.amd.com/TechDocs/24593.pdf (section 15.34)

[kvm-forum] https://www.linux-kvm.org/images/7/74/02x08A-Thomas_Lendacky-AMDs_Virtualizatoin_Memory_Encryption_Technology.pdf

[atomic-ops] Documentation/atomic_bitops.txt and Documentation/atomic_t.txt

[memory-barriers] Documentation/memory-barriers.txt

[lwn-mb] https://lwn.net/Articles/573436/