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

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.

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.

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.

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.

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).

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.

4.6 KVM_SET_MEMORY_REGION

Capability:	basic
Architectures:	all
Type:	vm ioctl
Parameters:	struct kvm_memory_region (in)
Returns:	0 on success, -1 on error

This ioctl is obsolete and has been removed.

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.

4.9 KVM_SET_MEMORY_ALIAS

Capability:	basic
Architectures:	x86
Type:	vm ioctl
Parameters:	struct kvm_memory_alias (in)
Returns:	0 (success), -1 (error)

This ioctl is obsolete and has been removed.

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;
};
```

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
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 overleaded 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 virutal 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.

4.17 KVM_DEBUG_GUEST

Capability:	basic
Architectures:	none
Type:	vcpu ioctl
Parameters:	none)
Returns:	-1 on error

Support for this has been removed. Use KVM_SET_GUEST_DEBUG instead.

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

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
Type:	vcpu ioctl
Parameters:	struct kvm_fpu (out)
Returns:	0 on success, -1 on error

Reads the floating point state from the vcpu.

```
/* 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;
};
```

4.23 KVM_SET_FPU

Capability:	basic
Architectures:	x86
Type:	vcpu ioctl
Parameters:	struct kvm_fpu (in)
Returns:	0 on success, -1 on error

Writes the floating point state to the vcpu.

```
/* 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;
};
```

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 (deasserted).

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 the KVM_XEN_HVM_CONFIG_INTERCEPT_HCALL flag is returned from the KVM_CAP_XEN_HVM check, it may be set in the flags field of this ioctl. This 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.

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_event (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.

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_event (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.

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_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).

It is recommended to use this API instead of the KVM_SET_MEMORY_REGION ioctl. The KVM_SET_MEMORY_REGION does not allow fine grained control over memory allocation and is deprecated.

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
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
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]
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]

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 is paused or not.

4.39 KVM_SET_MP_STATE

Capability:	KVM_CAP_MP_STATE
Architectures:	x86, s390, arm64, riscv
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.

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
Architectures:	x86
Type:	vcpu 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.

4.56 KVM_GET_TSC_KHZ

Capability:	KVM_CAP_GET_TSC_KHZ
Architectures:	x86
Type:	vcpu 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

(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
PPC	KVM_REG_PPC_EPCR	32

Arch	Register	Width (bits)
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
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

Arch	Register	Width (bits)
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
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

Arch	Register	Width (bits)
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
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

Encoding	Register	Bits	kvm_regs member
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] [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

[1] (*1*,*2*,*3*) 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).

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]
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 */
```

[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.

(See Documentation/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.

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 excution 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

Encoding	Register	Description
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
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

Encoding	Register	Description
...		
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

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
```

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 encoutering 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
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)
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 archietctural 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
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), > 0 if an exception occurred while walking the page tables

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 */
              };
              __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`

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.

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.

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 is > 0. 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 are as for logical accesses.

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_X86_SET_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 a read should immediately fail, while a 1 indicates that a read for a particular MSR should be handled 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 a write should immediately fail, while a 1 indicates that a write for a particular MSR should be handled regardless of the default filter action.

`KVM_MSR_FILTER_READ | KVM_MSR_FILTER_WRITE`

> Filter both read and write accesses to MSRs using the given bitmap. A 0 in the bitmap indicates that both reads and writes should immediately fail, while a 1 indicates that reads and writes for a particular MSR are not filtered by this range.

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 fall back to allowing access to the MSR.

`KVM_MSR_FILTER_DEFAULT_DENY`

> If no filter range matches an MSR index that is getting accessed, KVM will fall back to rejecting access to the MSR. In this mode, all MSRs that should be processed by KVM need to explicitly be marked as allowed in the bitmaps.

This ioctl allows user space to define up to 16 bitmaps of MSR ranges to specify whether a certain MSR access should be explicitly filtered for or not.

If this ioctl has never been invoked, MSR accesses are not guarded and the default KVM in-kernel emulation behavior is fully preserved.

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.

As soon as the filtering is in place, every MSR access is processed through the filtering except for accesses to the x2APIC MSRs (from 0x800 to 0x8ff); x2APIC MSRs are always allowed, independent of the `default_allow` setting, and their behavior depends on the `X2APIC_ENABLE` bit of the APIC base register.

> **Warning**
>
> MSR accesses coming from nested vmentry/vmexit are not filtered. This includes both writes to individual VMCS fields and reads/writes through the MSR lists pointed to by the VMCS.

If a bit is within one of the defined ranges, read and write accesses are guarded by the bitmap's value for the MSR index if the kind of access is included in the `struct kvm_msr_filter_range` flags. If no range cover this particular access, the behavior is determined by the flags field in the kvm_msr_filter struct: `KVM_MSR_FILTER_DEFAULT_ALLOW` and `KVM_MSR_FILTER_DEFAULT_DENY`.

Each bitmap range specifies a range of MSRs to potentially allow access on. The range goes from MSR index [base .. base+nmsrs]. The flags field indicates whether reads, writes or both reads and writes are filtered by setting a 1 bit in the bitmap for the corresponding MSR index.

If an MSR access is not permitted through the filtering, it generates a #GP inside the guest. When combined with KVM_CAP_X86_USER_SPACE_MSR, that allows user space to deflect and potentially handle various MSR accesses into user space.

If a vCPU is in running state while this ioctl is invoked, the vCPU may experience inconsistent filtering behavior on MSR accesses.

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 succsful 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

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.

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 KVM_S390_CMMA_PEEK is not set but migration mode was not enabled, with -EFAULT if the userspace address is invalid or if no page table is present for the addresses (e.g. when using hugepages).

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 Documentation/virt/kvm/amd-memory-encryption.rst.

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_RECOMMENDED 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 performaned 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

```
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 that the guest can program. The argument holds a list of events which will be allowed or denied. The eventsel+umask of each event the guest attempts to program is compared against the events field to determine whether the guest should have access. The events field only controls general purpose counters; fixed purpose counters are controlled by the fixed_counter_bitmap.

No flags are defined yet, the field must be zero.

Valid values for 'action':

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

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];
};
```

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.

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.

4.126 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 a read should immediately fail, while a 1 indicates that a read for a particular MSR should be handled 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 a write should immediately fail, while a 1 indicates that a write for a particular MSR should be handled regardless of the default filter action.

`KVM_MSR_FILTER_READ | KVM_MSR_FILTER_WRITE`

Filter both read and write accesses to MSRs using the given bitmap. A 0 in the bitmap indicates that both reads and writes should immediately fail, while a 1 indicates that reads and writes for a particular MSR are not filtered by this range.

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 fall back to allowing access to the MSR.

`KVM_MSR_FILTER_DEFAULT_DENY`

> If no filter range matches an MSR index that is getting accessed, KVM will fall back to rejecting access to the MSR. In this mode, all MSRs that should be processed by KVM need to explicitly be marked as allowed in the bitmaps.

This ioctl allows user space to define up to 16 bitmaps of MSR ranges to specify whether a certain MSR access should be explicitly filtered for or not.

If this ioctl has never been invoked, MSR accesses are not guarded and the default KVM in-kernel emulation behavior is fully preserved.

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.

As soon as the filtering is in place, every MSR access is processed through the filtering except for accesses to the x2APIC MSRs (from 0x800 to 0x8ff); x2APIC MSRs are always allowed, independent of the `default_allow` setting, and their behavior depends on the `X2APIC_ENABLE` bit of the APIC base register.

If a bit is within one of the defined ranges, read and write accesses are guarded by the bitmap's value for the MSR index if the kind of access is included in the `struct kvm_msr_filter_range` flags. If no range cover this particular access, the behavior is determined by the flags field in the kvm_msr_filter struct: `KVM_MSR_FILTER_DEFAULT_ALLOW` and `KVM_MSR_FILTER_DEFAULT_DENY`.

Each bitmap range specifies a range of MSRs to potentially allow access on. The range goes from MSR index [base .. base+nmsrs]. The flags field indicates whether reads, writes or both reads and writes are filtered by setting a 1 bit in the bitmap for the corresponding MSR index.

If an MSR access is not permitted through the filtering, it generates a #GP inside the guest. When combined with KVM_CAP_X86_USER_SPACE_MSR, that allows user space to deflect and potentially handle various MSR accesses into user space.

Note, invoking this ioctl with 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.

4.127 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;
              struct {
                      __u64 gfn;
              } shared_info;
              __u64 pad[4];
      } 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 is not aware of the Xen CPU id which is used as the index into the vcpu_info[] array, so cannot 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 â€” 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.

KVM_XEN_ATTR_TYPE_UPCALL_VECTOR

Sets the exception vector used to deliver Xen event channel upcalls.

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.

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;
      } 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.

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.

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.

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.

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. 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 thats the struct contain 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

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_MAX              KVM_STATS_UNIT_CYCLES
```

```
#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.) The bucket value indicates how many samples fell in the bucket's range.
- `KVM_STATS_TYPE_LOG_HIST` The statistic is reported as a logarithmic histogram. The number 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)). The bucket value indicates how many samples fell in the bucket's range.

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.

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.

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.

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)

/* 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;
        __u32 longmode;
        __u32 pad;
} hypercall;
```

Unused. This was once used for 'hypercall to userspace'. To implement such functionality, use KVM_EXIT_IO (x86) or

KVM_EXIT_MMIO (all except s390).

> **Note**
>
> KVM_EXIT_IO is significantly faster than KVM_EXIT_MMIO.

```
/* 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 it's 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
                      __u32 type;
                      __u64 flags;
              } 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. The 'flags' field describes architecture specific flags for the system-level event.

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.

Valid flags are:

- KVM_SYSTEM_EVENT_RESET_FLAG_PSCI_RESET2 (arm64 only) -- the guest issued a SYSTEM_RESET2 call according to v1.1 of the PSCI specification.

```
/* 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 will 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 trap occurred. User space will only receive MSR exit traps 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 user space which MSR the guest wants to read. To respond to this request with a successful read, user space 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, user space 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 user space which MSR the guest wants to write. Once finished processing the event, user space must continue vCPU execution. If the MSR write was unsuccessful, user space also sets the "error" field to "1".

```
              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;
```

System Message: WARNING/2 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\virt\kvm\[linux-master][Documentation][virt][kvm]api.rst`, line 6193)

Literal block ends without a blank line; unexpected unindent.

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.

```
        /* 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.

```
};
```

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.

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.

[3] For the new DR6 bits, note that bit 16 is set iff the #DB exception will clear DR6.RTM.

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:

```
#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

This capability overrides the kvm module parameter halt_poll_ns for the target VM.

VCPU polling allows a VCPU to poll for wakeup events instead of immediately scheduling during guest halts. The maximum time a VCPU can spend polling is controlled by the kvm module parameter halt_poll_ns. This capability allows the maximum halt time to specified on a per-VM basis, effectively overriding the module parameter for the target VM.

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 enables trapping of #GP invoking RDMSR and WRMSR instructions into user space.

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, user space may enable this capability. With it enabled, MSR accesses that match the mask specified in args[0] and trigger a #GP event inside the guest by KVM will instead trigger

KVM_EXIT_X86_RDMSR and KVM_EXIT_X86_WRMSR exit notifications which user space can then handle to implement model specific MSR handling and/or user notifications to inform a user that an MSR was not handled.

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 priveleged 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/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 not-shareable (no MAP_SHARED), attempts to create a memslot with a MAP_SHARED 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.

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 compatibilty, 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 Documentation/virt/kvm/devices/vcpu.rst "KVM_ARM_VCPU_PVTIME_CTRL". For x86 see Documentation/virt/kvm/msr.rst "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 reject 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

Architectures: x86
Parameters: args[0] - size of the dirty log ring

KVM is capable of tracking dirty memory using ring buffers that are mmaped 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 it's 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:

```
     dirtied         harvested        reset
00 -----------> 01 -------------> 1X -------+
 ^                                          |
 |                                          |
 +------------------------------------------+
```

To harvest the dirty pages, userspace accesses the mmaped 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.

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: the capability KVM_CAP_DIRTY_LOG_RING and the corresponding ioctl KVM_RESET_DIRTY_RINGS are mutual exclusive to the existing ioctls KVM_GET_DIRTY_LOG and KVM_CLEAR_DIRTY_LOG. After enabling KVM_CAP_DIRTY_LOG_RING with an acceptable dirty ring size, the virtual machine will switch to ring-buffer dirty page tracking and further KVM_GET_DIRTY_LOG or KVM_CLEAR_DIRTY_LOG ioctls will fail.

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 << 2)
#define KVM_XEN_HVM_CONFIG_EVTCHN_2LEVEL      (1 << 3)
```

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.

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 implmented 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.

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.

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