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 three 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).

  Only run VM ioctls 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.

  Only run vcpu ioctls from the same thread that was used to create the vcpu.

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 ioctl on a VM fd will create a virtual cpu and return a file descriptor pointing to it. Finally, ioctls on a vcpu fd can be used to control the vcpu, including 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. The only supported use is one virtual machine per process, and one vcpu per thread.

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 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), or a KVM_CAP_xyz constant, which
means availability needs to be checked with KVM_CHECK_EXTENSION
(see section 4.4).

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: none
Returns: a VM fd that can be used to control the new virtual machine.

The new VM has no virtual cpus and no memory. An mmap() of a VM fd
will access the virtual machine's physical address space; offset zero
corresponds to guest physical address zero. Use of mmap() on a VM fd
is discouraged if userspace memory allocation (KVM_CAP_USER_MEMORY) is
available.

4.3 KVM_GET_MSR_INDEX_LIST

Capability: basic
Architectures: x86
Type: system
Parameters: struct kvm_msr_list (in/out)

```
Returns: 0 on success; -1 on error
Errors:
  E2BIG:     the msr index list is to be to fit in the array specified by
             the user.

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

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

4.4 KVM_CHECK_EXTENSION

Capability: basic
Architectures: all
Type: system 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.

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.

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

```
struct kvm_memory_region {
	__u32 slot;
	__u32 flags;
	__u64 guest_phys_addr;
	__u64 memory_size; /* bytes */
};
```

```
/* for kvm_memory_region::flags */
#define KVM_MEM_LOG_DIRTY_PAGES  1UL
```

This ioctl allows the user to create or modify a guest physical memory
slot. When changing an existing slot, it may be moved in the guest
physical memory space, or its flags may be modified. It may not be
resized. Slots may not overlap.

The flags field supports just one flag, KVM_MEM_LOG_DIRTY_PAGES, which
instructs kvm to keep track of writes to memory within the slot. See
the KVM_GET_DIRTY_LOG ioctl.

It is recommended to use the KVM_SET_USER_MEMORY_REGION ioctl instead
of this API, if available. This newer API allows placing guest memory
at specified locations in the host address space, yielding better
control and easy access.

4.6 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. The vcpu id is a small integer
in the range [0, max_vcpus).

4.7 KVM_GET_DIRTY_LOG (vm ioctl)

Capability: basic
Architectures: x86
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.

4.8 KVM_SET_MEMORY_ALIAS

Capability: basic
Architectures: x86

```
Type: vm ioctl
Parameters: struct kvm_memory_alias (in)
Returns: 0 (success), -1 (error)

struct kvm_memory_alias {
	__u32 slot;  /* this has a different namespace than memory slots */
	__u32 flags;
	__u64 guest_phys_addr;
	__u64 memory_size;
	__u64 target_phys_addr;
};

Defines a guest physical address space region as an alias to another
region.  Useful for aliased address, for example the VGA low memory
window. Should not be used with userspace memory.

4.9 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

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).

4.10 KVM_GET_REGS

Capability: basic
Architectures: all
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;
};

4.11 KVM_SET_REGS

Capability: basic
Architectures: all
```

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.

4.12 KVM_GET_SREGS

Capability: basic
Architectures: x86
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];
};
```

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.

4.13 KVM_SET_SREGS

Capability: basic
Architectures: x86
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.

4.14 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];
};
```

4.15 KVM_INTERRUPT

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

Queues a hardware interrupt vector to be injected. This is only
useful if in-kernel local APIC is not used.

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

Note 'irq' is an interrupt vector, not an interrupt pin or line.

4.16 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.

4.17 KVM_GET_MSRS

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

Reads model-specific registers from the vcpu. Supported msr indices can
be obtained using KVM_GET_MSR_INDEX_LIST.

```
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.

4.18 KVM_SET_MSRS

Capability: basic
Architectures: x86
Type: vcpu ioctl
Parameters: struct kvm_msrs (in)
Returns: 0 on success, -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.

4.19 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.

```
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];
};
```

4.20 KVM_SET_SIGNAL_MASK

Capability: basic
Architectures: x86
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];
};
```

4.21 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;
};
```

4.22 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;
};
```

4.23 KVM_CREATE_IRQCHIP

Capability: KVM_CAP_IRQCHIP
Architectures: x86, ia64
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 ia64, a IOSAPIC is created.

4.24 KVM_IRQ_LINE

Capability: KVM_CAP_IRQCHIP
Architectures: x86, ia64
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. Requires 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.

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

4.25 KVM_GET_IRQCHIP

Capability: KVM_CAP_IRQCHIP
Architectures: x86, ia64
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;
};
```

4.26 KVM_SET_IRQCHIP

Capability: KVM_CAP_IRQCHIP
Architectures: x86, ia64
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;
};
```

4.27 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];
};
```

4.27 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.

```
struct kvm_clock_data {
        __u64 clock;  /* kvmclock current value */
        __u32 flags;
        __u32 pad[9];
};
```

4.28 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.

```
struct kvm_clock_data {
        __u64 clock;  /* kvmclock current value */
        __u32 flags;
        __u32 pad[9];
};
```

4.29 KVM_GET_VCPU_EVENTS

Capability: KVM_CAP_VCPU_EVENTS
Extended by: KVM_CAP_INTR_SHADOW
Architectures: x86
Type: vm ioctl
Parameters: struct kvm_vcpu_event (out)
Returns: 0 on success, -1 on error

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 pad;
		__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;
};
```

KVM_VCPUEVENT_VALID_SHADOW may be set in the flags field to signal that interrupt.shadow contains a valid state. Otherwise, this field is undefined.

4.30 KVM_SET_VCPU_EVENTS

Capability: KVM_CAP_VCPU_EVENTS
Extended by: KVM_CAP_INTR_SHADOW
Architectures: x86
Type: vm ioctl
Parameters: struct kvm_vcpu_event (in)
Returns: 0 on success, -1 on error

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 and sipi_vector. 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

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.

4.32 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];
};
```

4.33 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.

4.34 KVM_SET_USER_MEMORY_REGION

Capability: KVM_CAP_USER_MEM
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
```

This ioctl allows the user to create or modify a guest physical memory slot. When changing an existing slot, it may be moved in the guest physical memory space, or its flags may be modified. It may not be resized. Slots may not overlap in guest physical address space.

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.

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 just one flag, KVM_MEM_LOG_DIRTY_PAGES, which instructs kvm to keep track of writes to memory within the slot. See the KVM_GET_DIRTY_LOG ioctl.

When the KVM_CAP_SYNC_MMU capability, 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.

4.35 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).

4.36 KVM_ENABLE_CAP

Capability: KVM_CAP_ENABLE_CAP
Architectures: ppc
Type: vcpu ioctl
Parameters: struct kvm_enable_cap (in)
Returns: 0 on success; -1 on error

+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];
};
```

4.37 KVM_GET_MP_STATE

Capability: KVM_CAP_MP_STATE
Architectures: x86, ia64
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
 - KVM_MP_STATE_UNINITIALIZED:   the vcpu is an application processor (AP)
                                 which has not yet received an INIT signal
 - KVM_MP_STATE_INIT_RECEIVED:   the vcpu has received an INIT signal, and is
                                 now ready for a SIPI
 - KVM_MP_STATE_HALTED:          the vcpu has executed a HLT instruction and
                                 is waiting for an interrupt
 - KVM_MP_STATE_SIPI_RECEIVED:   the vcpu has just received a SIPI (vector
                                 accesible via KVM_GET_VCPU_EVENTS)
```

This ioctl is only useful after KVM_CREATE_IRQCHIP. Without an in-kernel
irqchip, the multiprocessing state must be maintained by userspace.

4.38 KVM_SET_MP_STATE

Capability: KVM_CAP_MP_STATE
Architectures: x86, ia64
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.

This ioctl is only useful after KVM_CREATE_IRQCHIP. Without an in-kernel
irqchip, the multiprocessing state must be maintained by userspace.

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 padding1[7];

	/* 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.

```
	__u8 padding2[2];

	/* 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;
		} 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.

```
		struct {
			struct kvm_debug_exit_arch arch;
		} debug;
```

Unused.

```
		/* 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.

NOTE: For KVM_EXIT_IO, KVM_EXIT_MMIO and KVM_EXIT_OSI, 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. Userspace
can re-enter the guest with an unmasked signal pending to complete
pending operations.

```
		/* 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_DCR */
		struct {
			__u32 dcrn;
			__u32 data;
			__u8  is_write;
		} dcr;
```

powerpc specific.

```
		/* 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.

```
		/* Fix the size of the union. */
		char padding[256];
	};
};
```