Linux Virt Documentation

The kernel development community

Jul 14, 2020

CONTENTS

CHAPTER

ONE

KVM

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

1.1.1 1. General description

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

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

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

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

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

1.1.2 2. File descriptors

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

In general file descriptors can be migrated among processes by means of fork() and the SCM_RIGHTS facility of unix domain socket. These kinds of tricks are explicitly not supported by kvm. While they will not cause harm to the host, their actual behavior is not guaranteed by the API. See “General description” for details on the ioctl usage model that is supported by KVM.

It is important to note that althought 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 via fork(), dup(), etc ⋯without careful consideration is strongly discouraged and may have unwanted side effects, e.g. memory allocated by and on behalf of the VM’ s process may not be freed/unaccounted when the VM is shut down.

1.1.3 3. Extensions

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

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

1.1.4 4. API description

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

Capability: which KVM extension provides this ioctl. Can be ‘basic’ , which means that is will be provided by any kernel that supports API version 12 (see section 4.1), a KVM_CAP_xyz constant, which means availability needs to be checked with KVM_CHECK_EXTENSION (see section 4.4), or ‘none’ which means that while not all kernels

support this ioctl, there's no capability bit to check its availability: for kernels that don't support the ioctl, the ioctl returns -ENOTTY.

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

Type: system, vm, or vcpu.

Parameters: what parameters are accepted by the ioctl.

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

4.1 KVM_GET_API_VERSION

Capability basic

Architectures all

Type system ioctl

Parameters none

Returns the constant KVM_API_VERSION (=12)

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

4.2 KVM_CREATE_VM

Capability basic

Architectures all

Type system ioctl

Parameters machine type identifier (KVM_VM_*)

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

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

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).

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

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.

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 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];
};
```

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.

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 specifies the address space for which you want to return the dirty bitmap. They must be less than the value that KVM_CHECK_EXTENSION returns for the KVM_CAP_MULTI_ADDRESS_SPACE capability.

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.

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

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 ARM, 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 ARM, 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

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 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.

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

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];
};
```

4.21 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, ARM, 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 ARM/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, arm, 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).

ARM/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 arm/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 */
```

(continues on next page)

(continued from previous page)

```
                __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];
};
```

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 only flag defined now is 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.

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

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.

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

4.31 KVM_GET_VCPU_EVENTS

Capability KVM_CAP_VCPU_EVENTS

Extended by KVM_CAP_INTR_SHADOW

Architectures x86, arm, 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 {
```

(continues on next page)

(continued from previous page)

```
                __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.

ARM/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, arm, 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.

ARM/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];
```

(continues on next page)

(continued from previous page)

```
      __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.

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

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 vcpu 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, arm, arm64

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,arm/arm64]
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,arm/arm64]
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 arm/arm64:

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, arm, arm64

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 arm/arm64:

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.

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];
};
```

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

4.43 KVM_SET_XSAVE

Capability KVM_CAP_XSAVE

Architectures x86

Type vcpu ioctl

Parameters struct kvm_xsave (in)

Returns 0 on success, -1 on error

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

This ioctl would copy userspace’s xsave struct to the kernel.

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).

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 arm 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 arm/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;
              __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
```

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;
```

(continues on next page)

(continued from previous page)

```
};

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;
};
```

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_SIAR	64
PPC	KVM_REG_PPC_SDAR	64
PPC	KVM_REG_PPC_SIER	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

Continued on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
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
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

Continued on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
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
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

Continued on next page

Table 1 – continued from previous page

Arch	Register	Width (bits)
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
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

Continued on next page

Table 1 – continued from previous page

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

arm64 CCSIDR registers are demultiplexed by CSSELR value:

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

arm64 system registers have the following id bit patterns:

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

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

arm64 firmware pseudo-registers have the following bit pattern:

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

arm64 SVE registers have the following bit patterns:

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

[1] These encodings are not accepted for SVE-enabled vcpus. See KVM_ARM_VCPU_INIT.
The equivalent register content can be accessed via bits [127:0] of the corresponding SVE Zn registers instead for vcpus that have SVE enabled (see below).

[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.

```
__u64 vector_lengths[KVM_ARM64_SVE_VLS_WORDS];

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

(See Documentation/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>
```

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 arm 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 arm 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 arm/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

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

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 arm, 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.

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 arm, arm64

Type vm ioctl

Parameters struct 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 in VCPU matching underlying host.

4.84 KVM_GET_REG_LIST

Capability basic

Architectures arm, 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 arm, 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.

ARM/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  |
```

ARM/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;
```

(continues on next page)

(continued from previous page)

```
    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, 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]

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.

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

Architectures s390

Type 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 logical (virtual) memory of a VCPU.

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 */
      __u8 ar;                /* the access register number */
      __u8 reserved[31];      /* should be set to 0 */
};
```

The type of operation is specified in the "op" field. It is either KVM_S390_MEMOP_LOGICAL_READ for reading from logical memory space or KVM_S390_MEMOP_LOGICAL_WRITE for writing to logical memory space. The KVM_S390_MEMOP_F_CHECK_ONLY flag can be set in the "flags" field to check whether the corresponding memory access would create an access exception (without touching the data in the memory at the destination). In case an access exception occurred while walking the MMU tables of the guest, the ioctl returns a positive error number to indicate 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 in the "flags" field.

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 KVM_S390_MEMOP_LOGICAL_READ, or where the data that should be written is stored for a KVM_S390_MEMOP_LOGICAL_WRITE. When KVM_S390_MEMOP_F_CHECK_ONLY is specified, "buf" is unused and can be NULL. "ar" designates the access register number to be used; the valid range is 0..15.

The "reserved" field is meant for future extensions. It is not used by KVM with the currently defined set of flags.

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_KEYS_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_ALLOC_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_ALLOC_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;
```

(continues on next page)

(continued from previous page)

```
                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_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.

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, -ENOSPC if there was a hash collision

```
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,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 starts, stops or monitors the preparation of a new potential HPT for the guest, essentially implementing the H_RESIZE_HPT_PREPARE hypercall.

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.

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

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.

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.

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

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 system

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;
```

(continues on next page)

(continued from previous page)

```
};

#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
__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, arm, arm64, mips

Type vm ioctl

Parameters struct kvm_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 specifies the address space for which you want to return the dirty bitmap. They must be less than the value that KVM_CHECK_EXTENSION returns for the KVM_CAP_MULTI_ADDRESS_SPACE capability.

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

Architectures x86

Type 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_NESTED_FEATURES leaf is only exposed when Enlightened VMCS was enabled on the corresponding vCPU (KVM_CAP_HYPERV_ENLIGHTENED_VMCS).

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 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.

4.119 KVM_ARM_VCPU_FINALIZE

Architectures arm, 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.

1.1.5 5. The kvm_run structure

Application code obtains a pointer to the kvm_run structure by mmap()ing a vcpu fd. From that point, application code can control execution by changing fields in kvm_run prior to calling the KVM_RUN ioctl, and obtain information about the reason KVM_RUN returned by looking up structure members.

```
struct kvm_run {
      /* in */
      __u8 request_interrupt_window;
```

Request that KVM_RUN return when it becomes possible to inject external interrupts into the guest. Useful in conjunction with KVM_INTERRUPT.

```
__u8 immediate_exit;
```

This field is polled once when KVM_RUN starts; if non-zero, KVM_RUN exits immediately, returning -EINTR. In the common scenario where a signal is used to "kick" a VCPU out of KVM_RUN, this field can be used to avoid usage of KVM_SET_SIGNAL_MASK, which has worse scalability. Rather than blocking the signal outside KVM_RUN, userspace can set up a signal handler that sets run->immediate_exit to a non-zero value.

This field is ignored if KVM_CAP_IMMEDIATE_EXIT is not available.

```
__u8 padding1[6];

/* out */
__u32 exit_reason;
```

When KVM_RUN has returned successfully (return value 0), this informs application code why KVM_RUN has returned. Allowable values for this field are detailed below.

```
__u8 ready_for_interrupt_injection;
```

If request_interrupt_window has been specified, this field indicates an interrupt can be injected now with KVM_INTERRUPT.

```
__u8 if_flag;
```

The value of the current interrupt flag. Only valid if in-kernel local APIC is not used.

```
__u16 flags;
```

More architecture-specific flags detailing state of the VCPU that may affect the device' s behavior. The only currently defined flag is KVM_RUN_X86_SMM, which is valid on x86 machines and is set if the VCPU is in system management mode.

```
/* 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.

```
/* 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 and KVM_EXIT_EPR 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;
```

(continues on next page)

(continued from previous page)

```
        __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 ARM/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.

```
/* 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;
```

(continues on next page)

(continued from previous 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 arm and 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 HSR (arm) and ESR_EL2 (arm64) 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.

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

/*
 * shared registers between kvm and userspace.
 * kvm_valid_regs specifies the register classes set by the host
 * kvm_dirty_regs specified the register classes dirtied by userspace
 * struct kvm_sync_regs is architecture specific, as well as the
 * bits for kvm_valid_regs and kvm_dirty_regs
 */
__u64 kvm_valid_regs;
__u64 kvm_dirty_regs;
union {
        struct kvm_sync_regs regs;
        char padding[SYNC_REGS_SIZE_BYTES];
} s;
```

If KVM_CAP_SYNC_REGS is defined, these fields allow userspace to access certain guest registers without having to call SET/GET_*REGS. Thus we can avoid some system call overhead if userspace has to handle the exit. Userspace can query the validity of the structure by checking kvm_valid_regs for specific bits. These bits are architecture specific and usually define the validity of a groups of registers. (e.g. one bit for general purpose registers)

Please note that the kernel is allowed to use the kvm_run structure as the primary storage for certain register types. Therefore, the kernel may use the values in kvm_run even if the corresponding bit in kvm_dirty_regs is not set.

```
};
```

1.1.6 6. Capabilities that can be enabled on vCPUs

There are certain capabilities that change the behavior of the virtual CPU or the virtual machine when enabled. To enable them, please see section 4.37. Below you can find a list of capabilities and what their effect on the vCPU or the virtual machine is when enabling them.

The following information is provided along with the description:

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

Target: whether this is a per-vcpu or per-vm capability.

Parameters: what parameters are accepted by the capability.

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

6.1 KVM_CAP_PPC_OSI

Architectures ppc

Target vcpu

Parameters none

Returns 0 on success; -1 on error

This capability enables interception of OSI hypercalls that otherwise would be treated as normal system calls to be injected into the guest. OSI hypercalls were invented by Mac-on-Linux to have a standardized communication mechanism between the guest and the host.

When this capability is enabled, KVM_EXIT_OSI can occur.

6.2 KVM_CAP_PPC_PAPR

Architectures ppc

Target vcpu

Parameters none

Returns 0 on success; -1 on error

This capability enables interception of PAPR hypercalls. PAPR hypercalls are done using the hypercall instruction "sc 1".

It also sets the guest privilege level to "supervisor" mode. Usually the guest runs in "hypervisor" privilege mode with a few missing features.

In addition to the above, it changes the semantics of SDR1. In this mode, the HTAB address part of SDR1 contains an HVA instead of a GPA, as PAPR keeps the HTAB invisible to the guest.

When this capability is enabled, KVM_EXIT_PAPR_HCALL can occur.

6.3 KVM_CAP_SW_TLB

Architectures ppc

Target vcpu

Parameters args[0] is the address of a struct kvm_config_tlb

Returns 0 on success; -1 on error

```
struct kvm_config_tlb {
      __u64 params;
      __u64 array;
      __u32 mmu_type;
      __u32 array_len;
};
```

Configures the virtual CPU's TLB array, establishing a shared memory area between userspace and KVM. The "params" and "array" fields are userspace addresses of mmu-type-specific data structures. The "array_len" field is an safety mechanism, and should be set to the size in bytes of the memory that userspace has reserved for the array. It must be at least the size dictated by "mmu_type" and "params".

While KVM_RUN is active, the shared region is under control of KVM. Its contents are undefined, and any modification by userspace results in boundedly undefined behavior.

On return from KVM_RUN, the shared region will reflect the current state of the guest's TLB. If userspace makes any changes, it must call KVM_DIRTY_TLB to tell KVM which entries have been changed, prior to calling KVM_RUN again on this vcpu.

For mmu types KVM_MMU_FSL_BOOKE_NOHV and KVM_MMU_FSL_BOOKE_HV:

- The "params" field is of type "struct kvm_book3e_206_tlb_params".
- The "array" field points to an array of type "struct kvm_book3e_206_tlb_entry".
- The array consists of all entries in the first TLB, followed by all entries in the second TLB.
- Within a TLB, entries are ordered first by increasing set number. Within a set, entries are ordered by way (increasing ESEL).
- The hash for determining set number in TLB0 is: (MAS2 >> 12) & (num_sets - 1) where "num_sets" is the tlb_sizes[] value divided by the tlb_ways[] value.
- The tsize field of mas1 shall be set to 4K on TLB0, even though the hardware ignores this value for TLB0.

6.4 KVM_CAP_S390_CSS_SUPPORT

Architectures s390

Target vcpu

Parameters none

Returns 0 on success; -1 on error

This capability enables support for handling of channel I/O instructions.

TEST PENDING INTERRUPTION and the interrupt portion of TEST SUBCHANNEL are handled in-kernel, while the other I/O instructions are passed to userspace.

When this capability is enabled, KVM_EXIT_S390_TSCH will occur on TEST SUBCHANNEL intercepts.

Note that even though this capability is enabled per-vcpu, the complete virtual machine is affected.

6.5 KVM_CAP_PPC_EPR

Architectures ppc

Target vcpu

Parameters args[0] defines whether the proxy facility is active

Returns 0 on success; -1 on error

This capability enables or disables the delivery of interrupts through the external proxy facility.

When enabled (args[0] != 0), every time the guest gets an external interrupt delivered, it automatically exits into user space with a KVM_EXIT_EPR exit to receive the topmost interrupt vector.

When disabled (args[0] == 0), behavior is as if this facility is unsupported.

When this capability is enabled, KVM_EXIT_EPR can occur.

6.6 KVM_CAP_IRQ_MPIC

Architectures ppc

Parameters args[0] is the MPIC device fd; args[1] is the MPIC CPU number for this vcpu

This capability connects the vcpu to an in-kernel MPIC device.

6.7 KVM_CAP_IRQ_XICS

Architectures ppc

Target vcpu

Parameters args[0] is the XICS device fd; args[1] is the XICS CPU number (server ID) for this vcpu

This capability connects the vcpu to an in-kernel XICS device.

6.8 KVM_CAP_S390_IRQCHIP

Architectures s390

Target vm

Parameters none

This capability enables the in-kernel irqchip for s390. Please refer to "4.24 KVM_CREATE_IRQCHIP" for details.

6.9 KVM_CAP_MIPS_FPU

Architectures mips

Target vcpu

Parameters args[0] is reserved for future use (should be 0).

This capability allows the use of the host Floating Point Unit by the guest. It allows the Config1.FP bit to be set to enable the FPU in the guest. Once this is done the `KVM_REG_MIPS_FPR_*` and `KVM_REG_MIPS_FCR_*` registers can be accessed (depending on the current guest FPU register mode), and the Status.FR, Config5.FRE bits are accessible via the KVM API and also from the guest, depending on them being supported by the FPU.

6.10 KVM_CAP_MIPS_MSA

Architectures mips

Target vcpu

Parameters args[0] is reserved for future use (should be 0).

This capability allows the use of the MIPS SIMD Architecture (MSA) by the guest. It allows the Config3.MSAP bit to be set to enable the use of MSA by the guest. Once this is done the `KVM_REG_MIPS_VEC_*` and `KVM_REG_MIPS_MSA_*` registers can be accessed, and the Config5.MSAEn bit is accessible via the KVM API and also from the guest.

6.74 KVM_CAP_SYNC_REGS

Architectures s390, x86

Target s390: always enabled, x86: vcpu

Parameters none

Returns x86: KVM_CHECK_EXTENSION returns a bit-array indicating which register sets are supported (bitfields defined in arch/x86/include/uapi/asm/kvm.h).

As described above in the kvm_sync_regs struct info in section 5 (kvm_run): KVM_CAP_SYNC_REGS "allow[s] userspace to access certain guest registers without having to call SET/GET_*REGS" . This reduces overhead by eliminating repeated ioctl calls for setting and/or getting register values. This is particularly

important when userspace is making synchronous guest state modifications, e.g. when emulating and/or intercepting instructions in userspace.

For s390 specifics, please refer to the source code.

For x86:

- the register sets to be copied out to kvm_run are selectable by userspace (rather that all sets being copied out for every exit).
- vcpu_events are available in addition to regs and sregs.

For x86, the 'kvm_valid_regs' field of struct kvm_run is overloaded to function as an input bit-array field set by userspace to indicate the specific register sets to be copied out on the next exit.

To indicate when userspace has modified values that should be copied into the vCPU, the all architecture bitarray field, 'kvm_dirty_regs' must be set. This is done using the same bitflags as for the 'kvm_valid_regs' field. If the dirty bit is not set, then the register set values will not be copied into the vCPU even if they've been modified.

Unused bitfields in the bitarrays must be set to zero.

```
struct kvm_sync_regs {
      struct kvm_regs regs;
      struct kvm_sregs sregs;
      struct kvm_vcpu_events events;
};
```

6.75 KVM_CAP_PPC_IRQ_XIVE

Architectures ppc

Target vcpu

Parameters args[0] is the XIVE device fd; args[1] is the XIVE CPU number (server ID) for this vcpu

This capability connects the vcpu to an in-kernel XIVE device.

1.1.7 7. Capabilities that can be enabled on VMs

There are certain capabilities that change the behavior of the virtual machine when enabled. To enable them, please see section 4.37. Below you can find a list of capabilities and what their effect on the VM is when enabling them.

The following information is provided along with the description:

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

Parameters: what parameters are accepted by the capability.

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

7.1 KVM_CAP_PPC_ENABLE_HCALL

Architectures ppc

Parameters args[0] is the sPAPR hcall number; args[1] is 0 to disable, 1 to enable in-kernel handling

This capability controls whether individual sPAPR hypercalls (hcalls) get handled by the kernel or not. Enabling or disabling in-kernel handling of an hcall is effective across the VM. On creation, an initial set of hcalls are enabled for in-kernel handling, which consists of those hcalls for which in-kernel handlers were implemented before this capability was implemented. If disabled, the kernel will not to attempt to handle the hcall, but will always exit to userspace to handle it. Note that it may not make sense to enable some and disable others of a group of related hcalls, but KVM does not prevent userspace from doing that.

If the hcall number specified is not one that has an in-kernel implementation, the KVM_ENABLE_CAP ioctl will fail with an EINVAL error.

7.2 KVM_CAP_S390_USER_SIGP

Architectures s390

Parameters none

This capability controls which SIGP orders will be handled completely in user space. With this capability enabled, all fast orders will be handled completely in the kernel:

- SENSE
- SENSE RUNNING
- EXTERNAL CALL
- EMERGENCY SIGNAL
- CONDITIONAL EMERGENCY SIGNAL

All other orders will be handled completely in user space.

Only privileged operation exceptions will be checked for in the kernel (or even in the hardware prior to interception). If this capability is not enabled, the old way of handling SIGP orders is used (partially in kernel and user space).

7.3 KVM_CAP_S390_VECTOR_REGISTERS

Architectures s390

Parameters none

Returns 0 on success, negative value on error

Allows use of the vector registers introduced with z13 processor, and provides for the synchronization between host and user space. Will return -EINVAL if the machine does not support vectors.

7.4 KVM_CAP_S390_USER_STSI

Architectures s390

Parameters none

This capability allows post-handlers for the STSI instruction. After initial handling in the kernel, KVM exits to user space with KVM_EXIT_S390_STSI to allow user space to insert further data.

Before exiting to userspace, kvm handlers should fill in s390_stsi field of vcpu->run:

```
struct {
      __u64 addr;
      __u8 ar;
      __u8 reserved;
      __u8 fc;
      __u8 sel1;
      __u16 sel2;
} s390_stsi;

@addr - guest address of STSI SYSIB
@fc   - function code
@sel1 - selector 1
@sel2 - selector 2
@ar   - access register number
```

KVM handlers should exit to userspace with rc = -EREMOTE.

7.5 KVM_CAP_SPLIT_IRQCHIP

Architectures x86

Parameters args[0] - number of routes reserved for userspace IOAPICs

Returns 0 on success, -1 on error

Create a local apic for each processor in the kernel. This can be used instead of KVM_CREATE_IRQCHIP if the userspace VMM wishes to emulate the IOAPIC and PIC (and also the PIT, even though this has to be enabled separately).

This capability also enables in kernel routing of interrupt requests; when KVM_CAP_SPLIT_IRQCHIP only routes of KVM_IRQ_ROUTING_MSI type are used in the IRQ routing table. The first args[0] MSI routes are reserved for the IOAPIC pins. Whenever the LAPIC receives an EOI for these routes, a KVM_EXIT_IOAPIC_EOI vmexit will be reported to userspace.

Fails if VCPU has already been created, or if the irqchip is already in the kernel (i.e. KVM_CREATE_IRQCHIP has already been called).

7.6 KVM_CAP_S390_RI

Architectures s390

Parameters none

Allows use of runtime-instrumentation introduced with zEC12 processor. Will return -EINVAL if the machine does not support runtime-instrumentation. Will return -EBUSY if a VCPU has already been created.

7.7 KVM_CAP_X2APIC_API

Architectures x86

Parameters args[0] - features that should be enabled

Returns 0 on success, -EINVAL when args[0] contains invalid features

Valid feature flags in args[0] are:

```
#define KVM_X2APIC_API_USE_32BIT_IDS            (1ULL << 0)
#define KVM_X2APIC_API_DISABLE_BROADCAST_QUIRK  (1ULL << 1)
```

Enabling KVM_X2APIC_API_USE_32BIT_IDS changes the behavior of KVM_SET_GSI_ROUTING, KVM_SIGNAL_MSI, KVM_SET_LAPIC, and KVM_GET_LAPIC, allowing the use of 32-bit APIC IDs. See KVM_CAP_X2APIC_API in their respective sections.

KVM_X2APIC_API_DISABLE_BROADCAST_QUIRK must be enabled for x2APIC to work in logical mode or with more than 255 VCPUs. Otherwise, KVM treats 0xff as a broadcast even in x2APIC mode in order to support physical x2APIC without interrupt remapping. This is undesirable in logical mode, where 0xff represents CPUs 0-7 in cluster 0.

7.8 KVM_CAP_S390_USER_INSTR0

Architectures s390

Parameters none

With this capability enabled, all illegal instructions 0x0000 (2 bytes) will be intercepted and forwarded to user space. User space can use this mechanism e.g. to realize 2-byte software breakpoints. The kernel will not inject an operating exception for these instructions, user space has to take care of that.

This capability can be enabled dynamically even if VCPUs were already created and are running.

7.9 KVM_CAP_S390_GS

Architectures s390

Parameters none

Returns 0 on success; -EINVAL if the machine does not support guarded storage; -EBUSY if a VCPU has already been created.

Allows use of guarded storage for the KVM guest.

7.10 KVM_CAP_S390_AIS

Architectures s390

Parameters none

Allow use of adapter-interruption suppression. :Returns: 0 on success; -EBUSY if a VCPU has already been created.

7.11 KVM_CAP_PPC_SMT

Architectures ppc

Parameters vsmt_mode, flags

Enabling this capability on a VM provides userspace with a way to set the desired virtual SMT mode (i.e. the number of virtual CPUs per virtual core). The virtual SMT mode, vsmt_mode, must be a power of 2 between 1 and 8. On POWER8, vsmt_mode must also be no greater than the number of threads per subcore for the host. Currently flags must be 0. A successful call to enable this capability will result in vsmt_mode being returned when the KVM_CAP_PPC_SMT capability is subsequently queried for the VM. This capability is only supported by HV KVM, and can only be set before any VCPUs have been created. The KVM_CAP_PPC_SMT_POSSIBLE capability indicates which virtual SMT modes are available.

7.12 KVM_CAP_PPC_FWNMI

Architectures ppc

Parameters none

With this capability a machine check exception in the guest address space will cause KVM to exit the guest with NMI exit reason. This enables QEMU to build error log and branch to guest kernel registered machine check handling routine. Without this capability KVM will branch to guests' 0x200 interrupt vector.

7.13 KVM_CAP_X86_DISABLE_EXITS

Architectures x86

Parameters args[0] defines which exits are disabled

Returns 0 on success, -EINVAL when args[0] contains invalid exits

Valid bits in args[0] are:

```
#define KVM_X86_DISABLE_EXITS_MWAIT            (1 << 0)
#define KVM_X86_DISABLE_EXITS_HLT              (1 << 1)
#define KVM_X86_DISABLE_EXITS_PAUSE            (1 << 2)
#define KVM_X86_DISABLE_EXITS_CSTATE           (1 << 3)
```

Enabling this capability on a VM provides userspace with a way to no longer intercept some instructions for improved latency in some workloads, and is suggested when vCPUs are associated to dedicated physical CPUs. More bits can be added in the future; userspace can just pass the KVM_CHECK_EXTENSION result to KVM_ENABLE_CAP to disable all such vmexits.

Do not enable KVM_FEATURE_PV_UNHALT if you disable HLT exits.

7.14 KVM_CAP_S390_HPAGE_1M

Architectures s390

Parameters none

Returns 0 on success, -EINVAL if hpage module parameter was not set or cmma is enabled, or the VM has the KVM_VM_S390_UCONTROL flag set

With this capability the KVM support for memory backing with 1m pages through hugetlbfs can be enabled for a VM. After the capability is enabled, cmma can't be enabled anymore and pfmfi and the storage key interpretation are disabled. If cmma has already been enabled or the hpage module parameter is not set to 1, -EINVAL is returned.

While it is generally possible to create a huge page backed VM without this capability, the VM will not be able to run.

7.15 KVM_CAP_MSR_PLATFORM_INFO

Architectures x86

Parameters args[0] whether feature should be enabled or not

With this capability, a guest may read the MSR_PLATFORM_INFO MSR. Otherwise, a #GP would be raised when the guest tries to access. Currently, this capability does not enable write permissions of this MSR for the guest.

7.16 KVM_CAP_PPC_NESTED_HV

Architectures ppc

Parameters none

Returns 0 on success, -EINVAL when the implementation doesn't support nested-HV virtualization.

HV-KVM on POWER9 and later systems allows for "nested-HV" virtualization, which provides a way for a guest VM to run guests that can run using the CPU's supervisor mode (privileged non-hypervisor state). Enabling this capability on a VM depends on the CPU having the necessary functionality and on the facility being enabled with a kvm-hv module parameter.

7.17 KVM_CAP_EXCEPTION_PAYLOAD

Architectures x86

Parameters args[0] whether feature should be enabled or not

With this capability enabled, CR2 will not be modified prior to the emulated VM-exit when L1 intercepts a #PF exception that occurs in L2. Similarly, for kvm-intel only, DR6 will not be modified prior to the emulated VM-exit when L1 intercepts a #DB exception that occurs in L2. As a result, when KVM_GET_VCPU_EVENTS reports a pending #PF (or #DB) exception for L2, exception.has_payload will be set and the faulting address (or the new DR6 bits*) will be reported in the exception_payload field. Similarly, when userspace injects a #PF (or #DB) into L2 using KVM_SET_VCPU_EVENTS, it is expected to set exception.has_payload and to put the faulting address - or the new DR6 bits[3] - in the exception_payload field.

This capability also enables exception.pending in struct kvm_vcpu_events, which allows userspace to distinguish between pending and injected exceptions.

7.18 KVM_CAP_MANUAL_DIRTY_LOG_PROTECT2

Architectures x86, arm, 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

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

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.

1.1.8 8. Other capabilities.

This section lists capabilities that give information about other features of the KVM implementation.

8.1 KVM_CAP_PPC_HWRNG

Architectures ppc

This capability, if KVM_CHECK_EXTENSION indicates that it is available, means that 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 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 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 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 arm, 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 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 arm, 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

Architecture 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

Architecture: 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.

1.2 Secure Encrypted Virtualization (SEV)

1.2.1 Overview

Secure Encrypted Virtualization (SEV) is a feature found on AMD processors.

SEV is an extension to the AMD-V architecture which supports running virtual machines (VMs) under the control of a hypervisor. When enabled, the memory contents of a VM will be transparently encrypted with a key unique to that VM.

The hypervisor can determine the SEV support through the CPUID instruction. The CPUID function 0x8000001f reports information related to SEV:

```
0x8000001f[eax]:
                Bit[1]  indicates support for SEV
    ...
```

(continues on next page)

(continued from previous page)

```
          [ecx]:
                Bits[31:0]  Number of encrypted guests supported␣
↪simultaneously
```

If support for SEV is present, MSR 0xc001_0010 (MSR_K8_SYSCFG) and MSR 0xc001_0015 (MSR_K7_HWCR) can be used to determine if it can be enabled:

```
0xc001_0010:
        Bit[23]    1 = memory encryption can be enabled
                   0 = memory encryption can not be enabled

0xc001_0015:
        Bit[0]     1 = memory encryption can be enabled
                   0 = memory encryption can not be enabled
```

When SEV support is available, it can be enabled in a specific VM by setting the SEV bit before executing VMRUN.:

```
VMCB[0x90]:
        Bit[1]      1 = SEV is enabled
                    0 = SEV is disabled
```

SEV hardware uses ASIDs to associate a memory encryption key with a VM. Hence, the ASID for the SEV-enabled guests must be from 1 to a maximum value defined in the CPUID 0x8000001f[ecx] field.

1.2.2 SEV Key Management

The SEV guest key management is handled by a separate processor called the AMD Secure Processor (AMD-SP). Firmware running inside the AMD-SP provides a secure key management interface to perform common hypervisor activities such as encrypting bootstrap code, snapshot, migrating and debugging the guest. For more information, see the SEV Key Management spec [**?**]

The main ioctl to access SEV is KVM_MEM_ENCRYPT_OP. If the argument to KVM_MEM_ENCRYPT_OP is NULL, the ioctl returns 0 if SEV is enabled and ENOTTY` if it is disabled (on some older versions of Linux, the ioctl runs normally even with a NULL argument, and therefore will likely return ``EFAULT). If non-NULL, the argument to KVM_MEM_ENCRYPT_OP must be a struct kvm_sev_cmd:

```
struct kvm_sev_cmd {
        __u32 id;
        __u64 data;
        __u32 error;
        __u32 sev_fd;
};
```

The `id` field contains the subcommand, and the `data` field points to another struct containing arguments specific to command. The `sev_fd` should point to a file descriptor that is opened on the `/dev/sev` device, if needed (see individual commands).

On output, `error` is zero on success, or an error code. Error codes are defined in `<linux/psp-dev.h>`.

KVM implements the following commands to support common lifecycle events of SEV guests, such as launching, running, snapshotting, migrating and decommissioning.

1. KVM_SEV_INIT

The KVM_SEV_INIT command is used by the hypervisor to initialize the SEV platform context. In a typical workflow, this command should be the first command issued.

Returns: 0 on success, -negative on error

2. KVM_SEV_LAUNCH_START

The KVM_SEV_LAUNCH_START command is used for creating the memory encryption context. To create the encryption context, user must provide a guest policy, the owner's public Diffie-Hellman (PDH) key and session information.

Parameters: struct kvm_sev_launch_start (in/out)

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_start {
        __u32 handle;           /* if zero then firmware creates a new␣
↪handle */
        __u32 policy;           /* guest's policy */

        __u64 dh_uaddr;         /* userspace address pointing to the guest␣
↪owner's PDH key */
        __u32 dh_len;

        __u64 session_addr;     /* userspace address which points to the␣
↪guest session information */
        __u32 session_len;
};
```

On success, the 'handle' field contains a new handle and on error, a negative value.

KVM_SEV_LAUNCH_START requires the `sev_fd` field to be valid.

For more details, see SEV spec Section 6.2.

3. KVM_SEV_LAUNCH_UPDATE_DATA

The KVM_SEV_LAUNCH_UPDATE_DATA is used for encrypting a memory region. It also calculates a measurement of the memory contents. The measurement is a signature of the memory contents that can be sent to the guest owner as an attestation that the memory was encrypted correctly by the firmware.

Parameters (in): struct kvm_sev_launch_update_data

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_update {
        __u64 uaddr;    /* userspace address to be encrypted (must be 16-
↪byte aligned) */
        __u32 len;      /* length of the data to be encrypted (must be 16-
↪byte aligned) */
};
```

For more details, see SEV spec Section 6.3.

4. KVM_SEV_LAUNCH_MEASURE

The KVM_SEV_LAUNCH_MEASURE command is used to retrieve the measurement of the data encrypted by the KVM_SEV_LAUNCH_UPDATE_DATA command. The guest owner may wait to provide the guest with confidential information until it can verify the measurement. Since the guest owner knows the initial contents of the guest at boot, the measurement can be verified by comparing it to what the guest owner expects.

Parameters (in): struct kvm_sev_launch_measure

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_measure {
        __u64 uaddr;    /* where to copy the measurement */
        __u32 len;      /* length of measurement blob */
};
```

For more details on the measurement verification flow, see SEV spec Section 6.4.

5. KVM_SEV_LAUNCH_FINISH

After completion of the launch flow, the KVM_SEV_LAUNCH_FINISH command can be issued to make the guest ready for the execution.

Returns: 0 on success, -negative on error

6. KVM_SEV_GUEST_STATUS

The KVM_SEV_GUEST_STATUS command is used to retrieve status information about a SEV-enabled guest.

Parameters (out): struct kvm_sev_guest_status

Returns: 0 on success, -negative on error

```
struct kvm_sev_guest_status {
        __u32 handle;   /* guest handle */
        __u32 policy;   /* guest policy */
        __u8 state;     /* guest state (see enum below) */
};
```

SEV guest state:

```
enum {
SEV_STATE_INVALID = 0;
SEV_STATE_LAUNCHING,    /* guest is currently being launched */
SEV_STATE_SECRET,       /* guest is being launched and ready to accept the␣
↪ciphertext data */
SEV_STATE_RUNNING,      /* guest is fully launched and running */
SEV_STATE_RECEIVING,    /* guest is being migrated in from another SEV␣
↪machine */
SEV_STATE_SENDING       /* guest is getting migrated out to another SEV␣
↪machine */
};
```

7. KVM_SEV_DBG_DECRYPT

The KVM_SEV_DEBUG_DECRYPT command can be used by the hypervisor to request the firmware to decrypt the data at the given memory region.

Parameters (in): struct kvm_sev_dbg

Returns: 0 on success, -negative on error

```
struct kvm_sev_dbg {
        __u64 src_uaddr;        /* userspace address of data to decrypt */
        __u64 dst_uaddr;        /* userspace address of destination */
        __u32 len;              /* length of memory region to decrypt */
};
```

The command returns an error if the guest policy does not allow debugging.

8. KVM_SEV_DBG_ENCRYPT

The KVM_SEV_DEBUG_ENCRYPT command can be used by the hypervisor to request the firmware to encrypt the data at the given memory region.

Parameters (in): struct kvm_sev_dbg

Returns: 0 on success, -negative on error

```
struct kvm_sev_dbg {
        __u64 src_uaddr;        /* userspace address of data to encrypt */
        __u64 dst_uaddr;        /* userspace address of destination */
        __u32 len;              /* length of memory region to encrypt */
};
```

The command returns an error if the guest policy does not allow debugging.

9. KVM_SEV_LAUNCH_SECRET

The KVM_SEV_LAUNCH_SECRET command can be used by the hypervisor to inject secret data after the measurement has been validated by the guest owner.

Parameters (in): struct kvm_sev_launch_secret

Returns: 0 on success, -negative on error

```
struct kvm_sev_launch_secret {
        __u64 hdr_uaddr;        /* userspace address containing the packet␣
↪header */
        __u32 hdr_len;

        __u64 guest_uaddr;      /* the guest memory region where the␣
↪secret should be injected */
        __u32 guest_len;

        __u64 trans_uaddr;      /* the hypervisor memory region which␣
↪contains the secret */
        __u32 trans_len;
};
```

1.2.3 References

See [**?**], [**?**], [**?**] and [**?**] for more info.

1.3 KVM CPUID bits

Author Glauber Costa <glommer@gmail.com>

A guest running on a kvm host, can check some of its features using cpuid. This is not always guaranteed to work, since userspace can mask-out some, or even all KVM-related cpuid features before launching a guest.

KVM cpuid functions are:

function: KVM_CPUID_SIGNATURE (0x40000000)

returns:

```
eax = 0x40000001
ebx = 0x4b4d564b
ecx = 0x564b4d56
edx = 0x4d
```

Note that this value in ebx, ecx and edx corresponds to the string "KVMKVMKVM". The value in eax corresponds to the maximum cpuid function present in this leaf, and will be updated if more functions are added in the future. Note also that old hosts set eax value to 0x0. This should be interpreted as if the value was 0x40000001. This function queries the presence of KVM cpuid leafs.

function: define KVM_CPUID_FEATURES (0x40000001)

returns:

```
ebx, ecx
eax = an OR'ed group of (1 << flag)
```

where `flag` is defined as below:

flag	value	meaning
KVM_FEATURE_CLOCKSOURCE	0	kvmclock available at msrs 0x11 and 0x12
KVM_FEATURE_NOP_IO_DELAY	1	not necessary to perform delays on PIO operations
KVM_FEATURE_MMU_OP	2	deprecated
KVM_FEATURE_CLOCKSOURCE2	3	kvmclock available at msrs 0x4b564d00 and 0x4b564d01
KVM_FEATURE_ASYNC_PF	4	async pf can be enabled by writing to msr 0x4b564d02
KVM_FEATURE_STEAL_TIME	5	steal time can be enabled by writing to msr 0x4b564d03
KVM_FEATURE_PV_EOI	6	paravirtualized end of interrupt handler can be enabled by writing to msr 0x4b564d04
KVM_FEATURE_PV_UNHALT	7	guest checks this feature bit before enabling paravirtualized spinlock support
KVM_FEATURE_PV_TLB_FLUSH	9	guest checks this feature bit before enabling paravirtualized tlb flush
KVM_FEATURE_ASYNC_PF_VMEXIT	10	paravirtualized async PF VM EXIT can be enabled by setting bit 2 when writing to msr 0x4b564d02
KVM_FEATURE_PV_SEND_IPI	11	guest checks this feature bit before enabling paravirtualized sebd IPIs
KVM_FEATURE_PV_POLL_CONTROL	12	host-side polling on HLT can be disabled by writing to msr 0x4b564d05.
KVM_FEATURE_PV_SCHED_YIELD	13	guest checks this feature bit before using paravirtualized sched yield.
KVM_FEATURE_ASYNC_PF_INT	14	guest checks this feature bit before using the second async pf control msr 0x4b564d06 and async pf acknowledgment msr 0x4b564d07.
KVM_FEATURE_CLOCKSOURCE_STABLE_BIT	24	host will warn if no guest-side per-cpu warps are expeced in kvmclock

```
edx = an OR'ed group of (1 << flag)
```

Where `flag` here is defined as below:

flag	value	meaning
KVM_HINTS_REALTIME	0	guest checks this feature bit to determine that vCPUs are never preempted for an unlimited time allowing optimizations

1.4 The KVM halt polling system

The KVM halt polling system provides a feature within KVM whereby the latency of a guest can, under some circumstances, be reduced by polling in the host for some time period after the guest has elected to no longer run by cedeing. That is, when a guest vcpu has ceded, or in the case of powerpc when all of the vcpus of a single vcore have ceded, the host kernel polls for wakeup conditions before giving up the cpu to the scheduler in order to let something else run.

Polling provides a latency advantage in cases where the guest can be run again very quickly by at least saving us a trip through the scheduler, normally on the order of a few micro-seconds, although performance benefits are workload dependant. In the event that no wakeup source arrives during the polling interval or some other task on the runqueue is runnable the scheduler is invoked. Thus halt polling is especially useful on workloads with very short wakeup periods where the time spent halt polling is minimised and the time savings of not invoking the scheduler are distinguishable.

The generic halt polling code is implemented in:

virt/kvm/kvm_main.c: kvm_vcpu_block()

The powerpc kvm-hv specific case is implemented in:

arch/powerpc/kvm/book3s_hv.c: kvmppc_vcore_blocked()

1.4.1 Halt Polling Interval

The maximum time for which to poll before invoking the scheduler, referred to as the halt polling interval, is increased and decreased based on the perceived effectiveness of the polling in an attempt to limit pointless polling. This value is stored in either the vcpu struct:

kvm_vcpu->halt_poll_ns

or in the case of powerpc kvm-hv, in the vcore struct:

kvmppc_vcore->halt_poll_ns

Thus this is a per vcpu (or vcore) value.

During polling if a wakeup source is received within the halt polling interval, the interval is left unchanged. In the event that a wakeup source isn't received during the polling interval (and thus schedule is invoked) there are two options, either

the polling interval and total block time[0] were less than the global max polling interval (see module params below), or the total block time was greater than the global max polling interval.

In the event that both the polling interval and total block time were less than the global max polling interval then the polling interval can be increased in the hope that next time during the longer polling interval the wake up source will be received while the host is polling and the latency benefits will be received. The polling interval is grown in the function grow_halt_poll_ns() and is multiplied by the module parameters halt_poll_ns_grow and halt_poll_ns_grow_start.

In the event that the total block time was greater than the global max polling interval then the host will never poll for long enough (limited by the global max) to wakeup during the polling interval so it may as well be shrunk in order to avoid pointless polling. The polling interval is shrunk in the function shrink_halt_poll_ns() and is divided by the module parameter halt_poll_ns_shrink, or set to 0 iff halt_poll_ns_shrink == 0.

It is worth noting that this adjustment process attempts to hone in on some steady state polling interval but will only really do a good job for wakeups which come at an approximately constant rate, otherwise there will be constant adjustment of the polling interval.

[0] total block time: the time between when the halt polling function is invoked and a wakeup source received (irrespective of whether the scheduler is invoked within that function).

1.4.2 Module Parameters

The kvm module has 3 tuneable module parameters to adjust the global max polling interval as well as the rate at which the polling interval is grown and shrunk. These variables are defined in include/linux/kvm_host.h and as module parameters in virt/kvm/kvm_main.c, or arch/powerpc/kvm/book3s_hv.c in the powerpc kvm-hv case.

Module Parameter	Description	Default Value
halt_poll_ns	The global max polling interval which defines the ceiling value of the polling interval for each vcpu.	KVM_HALT_POLL_NS_DEFAULT (per arch value)
halt_poll_ns_grow	The value by which the halt polling interval is multiplied in the grow_halt_poll_ns() function.	2
halt_poll_ns_grow_start	The initial value to grow to from zero in the grow_halt_poll_ns() function.	10000
halt_poll_ns_shrink	The value by which the halt polling interval is divided in the shrink_halt_poll_ns() function.	0

These module parameters can be set from the debugfs files in:

/sys/module/kvm/parameters/

Note: that these module parameters are system wide values and are not able to be tuned on a per vm basis.

1.4.3 Further Notes

- Care should be taken when setting the halt_poll_ns module parameter as a large value has the potential to drive the cpu usage to 100% on a machine which would be almost entirely idle otherwise. This is because even if a guest has wakeups during which very little work is done and which are quite far apart, if the period is shorter than the global max polling interval (halt_poll_ns) then the host will always poll for the entire block time and thus cpu utilisation will go to 100%.
- Halt polling essentially presents a trade off between power usage and latency and the module parameters should be used to tune the affinity for this. Idle cpu time is essentially converted to host kernel time with the aim of decreasing latency when entering the guest.
- Halt polling will only be conducted by the host when no other tasks are runnable on that cpu, otherwise the polling will cease immediately and schedule will be invoked to allow that other task to run. Thus this doesn't allow a guest to denial of service the cpu.

1.5 Linux KVM Hypercall

X86: KVM Hypercalls have a three-byte sequence of either the vmcall or the vmmcall instruction. The hypervisor can replace it with instructions that are guaranteed to be supported.

Up to four arguments may be passed in rbx, rcx, rdx, and rsi respectively. The hypercall number should be placed in rax and the return value will be placed in rax. No other registers will be clobbered unless explicitly stated by the particular hypercall.

S390: R2-R7 are used for parameters 1-6. In addition, R1 is used for hypercall number. The return value is written to R2.

S390 uses diagnose instruction as hypercall (0x500) along with hypercall number in R1.

For further information on the S390 diagnose call as supported by KVM, refer to Documentation/virt/kvm/s390-diag.rst.

PowerPC: It uses R3-R10 and hypercall number in R11. R4-R11 are used as output registers. Return value is placed in R3.

KVM hypercalls uses 4 byte opcode, that are patched with 'hypercall-instructions' property inside the device tree's /hypervisor node. For more information refer to Documentation/virt/kvm/ppc-pv.rst

MIPS: KVM hypercalls use the HYPCALL instruction with code 0 and the hypercall number in $2 (v0). Up to four arguments may be placed in $4-$7 (a0-a3) and the return value is placed in $2 (v0).

1.5.1 KVM Hypercalls Documentation

The template for each hypercall is: 1. Hypercall name. 2. Architecture(s) 3. Status (deprecated, obsolete, active) 4. Purpose

1. KVM_HC_VAPIC_POLL_IRQ

Architecture x86

Status active

Purpose Trigger guest exit so that the host can check for pending interrupts on reentry.

2. KVM_HC_MMU_OP

Architecture x86

Status deprecated.

Purpose Support MMU operations such as writing to PTE, flushing TLB, release PT.

3. KVM_HC_FEATURES

Architecture PPC

Status active

Purpose Expose hypercall availability to the guest. On x86 platforms, cpuid used to enumerate which hypercalls are available. On PPC, either device tree based lookup (which is also what EPAPR dictates) OR KVM specific enumeration mechanism (which is this hypercall) can be used.

4. KVM_HC_PPC_MAP_MAGIC_PAGE

Architecture PPC

Status active

Purpose To enable communication between the hypervisor and guest there is a shared page that contains parts of supervisor visible register state. The guest can map this shared page to access its supervisor register through memory using this hypercall.

5. KVM_HC_KICK_CPU

Architecture x86

Status active

Purpose Hypercall used to wakeup a vcpu from HLT state

Usage example A vcpu of a paravirtualized guest that is busywaiting in guest kernel mode for an event to occur (ex: a spinlock to become available) can execute HLT instruction once it has busy-waited for more than a threshold time-interval. Execution of HLT instruction would cause the hypervisor to put the vcpu to sleep until occurrence of an appropriate event. Another vcpu of the same guest can wakeup the sleeping vcpu by issuing KVM_HC_KICK_CPU hypercall, specifying APIC ID (a1) of the vcpu to be woken up. An additional argument (a0) is used in the hypercall for future use.

6. KVM_HC_CLOCK_PAIRING

Architecture x86

Status active

Purpose Hypercall used to synchronize host and guest clocks.

Usage:

a0: guest physical address where host copies "struct kvm_clock_offset" structure.

a1: clock_type, ATM only KVM_CLOCK_PAIRING_WALLCLOCK (0) is supported (corresponding to the host' s CLOCK_REALTIME clock).

```
struct kvm_clock_pairing {
        __s64 sec;
        __s64 nsec;
        __u64 tsc;
        __u32 flags;
        __u32 pad[9];
};
```

Where:

- sec: seconds from clock_type clock.
- nsec: nanoseconds from clock_type clock.
- tsc: guest TSC value used to calculate sec/nsec pair
- flags: flags, unused (0) at the moment.

The hypercall lets a guest compute a precise timestamp across host and guest. The guest can use the returned TSC value to compute the CLOCK_REALTIME for its clock, at the same instant.

Returns KVM_EOPNOTSUPP if the host does not use TSC clocksource, or if clock type is different than KVM_CLOCK_PAIRING_WALLCLOCK.

6. KVM_HC_SEND_IPI

Architecture x86

Status active

Purpose Send IPIs to multiple vCPUs.

- a0: lower part of the bitmap of destination APIC IDs
- a1: higher part of the bitmap of destination APIC IDs
- a2: the lowest APIC ID in bitmap
- a3: APIC ICR

The hypercall lets a guest send multicast IPIs, with at most 128 128 destinations per hypercall in 64-bit mode and 64 vCPUs per hypercall in 32-bit mode. The destinations are represented by a bitmap contained in the first two arguments (a0 and a1). Bit 0 of a0 corresponds to the APIC ID in the third argument (a2), bit 1 corresponds to the APIC ID a2+1, and so on.

Returns the number of CPUs to which the IPIs were delivered successfully.

7. KVM_HC_SCHED_YIELD

Architecture x86

Status active

Purpose Hypercall used to yield if the IPI target vCPU is preempted

a0: destination APIC ID

Usage example When sending a call-function IPI-many to vCPUs, yield if any of the IPI target vCPUs was preempted.

1.6 KVM Lock Overview

1.6.1 1. Acquisition Orders

The acquisition orders for mutexes are as follows:

- kvm->lock is taken outside vcpu->mutex
- kvm->lock is taken outside kvm->slots_lock and kvm->irq_lock
- kvm->slots_lock is taken outside kvm->irq_lock, though acquiring them together is quite rare.

On x86, vcpu->mutex is taken outside kvm->arch.hyperv.hv_lock.

Everything else is a leaf: no other lock is taken inside the critical sections.

1.6.2 2. Exception

Fast page fault:

Fast page fault is the fast path which fixes the guest page fault out of the mmu-lock on x86. Currently, the page fault can be fast in one of the following two cases:

1. Access Tracking: The SPTE is not present, but it is marked for access tracking i.e. the SPTE_SPECIAL_MASK is set. That means we need to restore the saved R/X bits. This is described in more detail later below.

2. Write-Protection: The SPTE is present and the fault is caused by write-protect. That means we just need to change the W bit of the spte.

What we use to avoid all the race is the SPTE_HOST_WRITEABLE bit and SPTE_MMU_WRITEABLE bit on the spte:

- SPTE_HOST_WRITEABLE means the gfn is writable on host.
- SPTE_MMU_WRITEABLE means the gfn is writable on mmu. The bit is set when the gfn is writable on guest mmu and it is not write-protected by shadow page write-protection.

On fast page fault path, we will use cmpxchg to atomically set the spte W bit if spte.SPTE_HOST_WRITEABLE = 1 and spte.SPTE_WRITE_PROTECT = 1, or restore the saved R/X bits if VMX_EPT_TRACK_ACCESS mask is set, or both. This is safe because whenever changing these bits can be detected by cmpxchg.

But we need carefully check these cases:

1) The mapping from gfn to pfn

The mapping from gfn to pfn may be changed since we can only ensure the pfn is not changed during cmpxchg. This is a ABA problem, for example, below case will happen:

<table>
<tr><td colspan="2">At the beginning:
gpte = gfn1
gfn1 is mapped to pfn1 on host
spte is the shadow page table entry corresponding with gpte and
spte = pfn1</td></tr>
<tr><td colspan="2">On fast page fault path:</td></tr>
<tr><td>CPU 0:</td><td>CPU 1:</td></tr>
<tr><td>old_spte = *spte;</td><td></td></tr>
<tr><td></td><td>pfn1 is swapped out:
spte = 0;
pfn1 is re-alloced for gfn2.
gpte is changed to point to gfn2 by the guest:
spte = pfn1;</td></tr>
<tr><td colspan="2">if (cmpxchg(spte, old_spte, old_spte+W)
mark_page_dirty(vcpu->kvm, gfn1)
OOPS!!!</td></tr>
</table>

We dirty-log for gfn1, that means gfn2 is lost in dirty-bitmap.

For direct sp, we can easily avoid it since the spte of direct sp is fixed to gfn. For indirect sp, we disabled fast page fault for simplicity.

A solution for indirect sp could be to pin the gfn, for example via kvm_vcpu_gfn_to_pfn_atomic, before the cmpxchg. After the pinning:

- We have held the refcount of pfn that means the pfn can not be freed and be reused for another gfn.
- The pfn is writable and therefore it cannot be shared between different gfns by KSM.

Then, we can ensure the dirty bitmaps is correctly set for a gfn.

2) Dirty bit tracking

In the origin code, the spte can be fast updated (non-atomically) if the spte is read-only and the Accessed bit has already been set since the Accessed bit and Dirty bit can not be lost.

But it is not true after fast page fault since the spte can be marked writable between reading spte and updating spte. Like below case:

<table>
<tr><td colspan="2">At the beginning:
spte.W = 0
spte.Accessed = 1</td></tr>
<tr><td>CPU 0:</td><td>CPU 1:</td></tr>
<tr><td>In mmu_spte_clear_track_bits():
old_spte = *spte;

/* 'if' condition is satisfied. */
if (old_spte.Accessed == 1 &&
old_spte.W == 0)
spte = 0ull;</td><td></td></tr>
<tr><td></td><td>on fast page fault path:
spte.W = 1
memory write on the spte:
spte.Dirty = 1</td></tr>
<tr><td>else
old_spte = xchg(spte, 0ull)
if (old_spte.Accessed == 1)
kvm_set_pfn_accessed(spte.pfn);
if (old_spte.Dirty == 1)
kvm_set_pfn_dirty(spte.pfn);
OOPS!!!</td><td></td></tr>
</table>

The Dirty bit is lost in this case.

In order to avoid this kind of issue, we always treat the spte as "volatile" if it can be updated out of mmu-lock, see spte_has_volatile_bits(), it means, the spte is always atomically updated in this case.

3) flush tlbs due to spte updated

If the spte is updated from writable to readonly, we should flush all TLBs, otherwise rmap_write_protect will find a read-only spte, even though the writable spte might be cached on a CPU's TLB.

As mentioned before, the spte can be updated to writable out of mmu-lock on fast page fault path, in order to easily audit the path, we see if TLBs need be flushed caused by this reason in mmu_spte_update() since this is a common function to update spte (present -> present).

Since the spte is "volatile" if it can be updated out of mmu-lock, we always atomically update the spte, the race caused by fast page fault can be avoided, See the comments in spte_has_volatile_bits() and mmu_spte_update().

Lockless Access Tracking:

This is used for Intel CPUs that are using EPT but do not support the EPT A/D bits. In this case, when the KVM MMU notifier is called to track accesses to a page (via kvm_mmu_notifier_clear_flush_young), it marks the PTE as not-present

by clearing the RWX bits in the PTE and storing the original R & X bits in some unused/ignored bits. In addition, the SPTE_SPECIAL_MASK is also set on the PTE (using the ignored bit 62). When the VM tries to access the page later on, a fault is generated and the fast page fault mechanism described above is used to atomically restore the PTE to a Present state. The W bit is not saved when the PTE is marked for access tracking and during restoration to the Present state, the W bit is set depending on whether or not it was a write access. If it wasn't, then the W bit will remain clear until a write access happens, at which time it will be set using the Dirty tracking mechanism described above.

1.6.3 3. Reference

Name kvm_lock

Type mutex

Arch any

Protects

- vm_list

Name kvm_count_lock

Type raw_spinlock_t

Arch any

Protects

- hardware virtualization enable/disable

Comment 'raw' because hardware enabling/disabling must be atomic /wrt migration.

Name kvm_arch::tsc_write_lock

Type raw_spinlock

Arch x86

Protects

- kvm_arch::{last_tsc_write,last_tsc_nsec,last_tsc_offset}
- tsc offset in vmcb

Comment 'raw'because updating the tsc offsets must not be preempted.

Name kvm->mmu_lock

Type spinlock_t

Arch any

Protects -shadow page/shadow tlb entry

Comment it is a spinlock since it is used in mmu notifier.

Name kvm->srcu

Type srcu lock

Arch any

Protects

- kvm->memslots
- kvm->buses

Comment The srcu read lock must be held while accessing memslots (e.g. when using gfn_to_* functions) and while accessing in-kernel MMIO/PIO address->device structure mapping (kvm->buses). The srcu index can be stored in kvm_vcpu->srcu_idx per vcpu if it is needed by multiple functions.

Name blocked_vcpu_on_cpu_lock

Type spinlock_t

Arch x86

Protects blocked_vcpu_on_cpu

Comment This is a per-CPU lock and it is used for VT-d posted-interrupts. When VT-d posted-interrupts is supported and the VM has assigned devices, we put the blocked vCPU on the list blocked_vcpu_on_cpu protected by blocked_vcpu_on_cpu_lock, when VT-d hardware issues wakeup notification event since external interrupts from the assigned devices happens, we will find the vCPU on the list to wakeup.

1.7 The x86 kvm shadow mmu

The mmu (in arch/x86/kvm, files mmu.[ch] and paging_tmpl.h) is responsible for presenting a standard x86 mmu to the guest, while translating guest physical addresses to host physical addresses.

The mmu code attempts to satisfy the following requirements:

- **correctness:** the guest should not be able to determine that it is running on an emulated mmu except for timing (we attempt to comply with the specification, not emulate the characteristics of a particular implementation such as tlb size)
- **security:** the guest must not be able to touch host memory not assigned to it
- **performance:** minimize the performance penalty imposed by the mmu
- **scaling:** need to scale to large memory and large vcpu guests
- **hardware:** support the full range of x86 virtualization hardware
- **integration:** Linux memory management code must be in control of guest memory so that swapping, page migration, page merging, transparent hugepages, and similar features work without change
- **dirty tracking:** report writes to guest memory to enable live migration and framebuffer-based displays

- **footprint:** keep the amount of pinned kernel memory low (most memory should be shrinkable)
- **reliability:** avoid multipage or GFP_ATOMIC allocations

1.7.1 Acronyms

pfn	host page frame number
hpa	host physical address
hva	host virtual address
gfn	guest frame number
gpa	guest physical address
gva	guest virtual address
ngpa	nested guest physical address
ngva	nested guest virtual address
pte	page table entry (used also to refer generically to paging structure entries)
gpte	guest pte (referring to gfns)
spte	shadow pte (referring to pfns)
tdp	two dimensional paging (vendor neutral term for NPT and EPT)

1.7.2 Virtual and real hardware supported

The mmu supports first-generation mmu hardware, which allows an atomic switch of the current paging mode and cr3 during guest entry, as well as two-dimensional paging (AMD's NPT and Intel's EPT). The emulated hardware it exposes is the traditional 2/3/4 level x86 mmu, with support for global pages, pae, pse, pse36, cr0.wp, and 1GB pages. Emulated hardware also able to expose NPT capable hardware on NPT capable hosts.

1.7.3 Translation

The primary job of the mmu is to program the processor's mmu to translate addresses for the guest. Different translations are required at different times:

- when guest paging is disabled, we translate guest physical addresses to host physical addresses (gpa->hpa)
- when guest paging is enabled, we translate guest virtual addresses, to guest physical addresses, to host physical addresses (gva->gpa->hpa)
- when the guest launches a guest of its own, we translate nested guest virtual addresses, to nested guest physical addresses, to guest physical addresses, to host physical addresses (ngva->ngpa->gpa->hpa)

The primary challenge is to encode between 1 and 3 translations into hardware that support only 1 (traditional) and 2 (tdp) translations. When the number of required translations matches the hardware, the mmu operates in direct mode; otherwise it operates in shadow mode (see below).

1.7.4 Memory

Guest memory (gpa) is part of the user address space of the process that is using kvm. Userspace defines the translation between guest addresses and user addresses (gpa->hva); note that two gpas may alias to the same hva, but not vice versa.

These hvas may be backed using any method available to the host: anonymous memory, file backed memory, and device memory. Memory might be paged by the host at any time.

1.7.5 Events

The mmu is driven by events, some from the guest, some from the host.

Guest generated events:

- writes to control registers (especially cr3)
- invlpg/invlpga instruction execution
- access to missing or protected translations

Host generated events:

- changes in the gpa->hpa translation (either through gpa->hva changes or through hva->hpa changes)
- memory pressure (the shrinker)

1.7.6 Shadow pages

The principal data structure is the shadow page, 'struct kvm_mmu_page'. A shadow page contains 512 sptes, which can be either leaf or nonleaf sptes. A shadow page may contain a mix of leaf and nonleaf sptes.

A nonleaf spte allows the hardware mmu to reach the leaf pages and is not related to a translation directly. It points to other shadow pages.

A leaf spte corresponds to either one or two translations encoded into one paging structure entry. These are always the lowest level of the translation stack, with optional higher level translations left to NPT/EPT. Leaf ptes point at guest pages.

The following table shows translations encoded by leaf ptes, with higher-level translations in parentheses:

Non-nested guests:

```
nonpaging:      gpa->hpa
paging:         gva->gpa->hpa
paging, tdp:    (gva->)gpa->hpa
```

Nested guests:

```
non-tdp:       ngva->gpa->hpa  (*)
tdp:           (ngva->)ngpa->gpa->hpa

(*) the guest hypervisor will encode the ngva->gpa translation␣
↪into its page
    tables if npt is not present
```

Shadow pages contain the following information:

role.level: The level in the shadow paging hierarchy that this shadow page belongs to. 1=4k sptes, 2=2M sptes, 3=1G sptes, etc.

role.direct: If set, leaf sptes reachable from this page are for a linear range. Examples include real mode translation, large guest pages backed by small host pages, and gpa->hpa translations when NPT or EPT is active. The linear range starts at (gfn << PAGE_SHIFT) and its size is determined by role.level (2MB for first level, 1GB for second level, 0.5TB for third level, 256TB for fourth level) If clear, this page corresponds to a guest page table denoted by the gfn field.

role.quadrant: When role.gpte_is_8_bytes=0, the guest uses 32-bit gptes while the host uses 64-bit sptes. That means a guest page table contains more ptes than the host, so multiple shadow pages are needed to shadow one guest page. For first-level shadow pages, role.quadrant can be 0 or 1 and denotes the first or second 512-gpte block in the guest page table. For second-level page tables, each 32-bit gpte is converted to two 64-bit sptes (since each first-level guest page is shadowed by two first-level shadow pages) so role.quadrant takes values in the range 0..3. Each quadrant maps 1GB virtual address space.

role.access: Inherited guest access permissions in the form uwx. Note execute permission is positive, not negative.

role.invalid: The page is invalid and should not be used. It is a root page that is currently pinned (by a cpu hardware register pointing to it); once it is unpinned it will be destroyed.

role.gpte_is_8_bytes: Reflects the size of the guest PTE for which the page is valid, i.e. '1' if 64-bit gptes are in use, '0' if 32-bit gptes are in use.

role.nxe: Contains the value of efer.nxe for which the page is valid.

role.cr0_wp: Contains the value of cr0.wp for which the page is valid.

role.smep_andnot_wp: Contains the value of cr4.smep && !cr0.wp for which the page is valid (pages for which this is true are different from other pages; see the treatment of cr0.wp=0 below).

role.smap_andnot_wp: Contains the value of cr4.smap && !cr0.wp for which the page is valid (pages for which this is true are different from other pages; see the treatment of cr0.wp=0 below).

role.ept_sp: This is a virtual flag to denote a shadowed nested EPT page. ept_sp is true if "cr0_wp && smap_andnot_wp", an otherwise invalid combination.

role.smm: Is 1 if the page is valid in system management mode. This field

determines which of the kvm_memslots array was used to build this shadow page; it is also used to go back from a struct kvm_mmu_page to a memslot, through the kvm_memslots_for_spte_role macro and __gfn_to_memslot.

role.ad_disabled: Is 1 if the MMU instance cannot use A/D bits. EPT did not have A/D bits before Haswell; shadow EPT page tables also cannot use A/D bits if the L1 hypervisor does not enable them.

gfn: Either the guest page table containing the translations shadowed by this page, or the base page frame for linear translations. See role.direct.

spt: A pageful of 64-bit sptes containing the translations for this page. Accessed by both kvm and hardware. The page pointed to by spt will have its page->private pointing back at the shadow page structure. sptes in spt point either at guest pages, or at lower-level shadow pages. Specifically, if sp1 and sp2 are shadow pages, then sp1->spt[n] may point at __pa(sp2->spt). sp2 will point back at sp1 through parent_pte. The spt array forms a DAG structure with the shadow page as a node, and guest pages as leaves.

gfns: An array of 512 guest frame numbers, one for each present pte. Used to perform a reverse map from a pte to a gfn. When role.direct is set, any element of this array can be calculated from the gfn field when used, in this case, the array of gfns is not allocated. See role.direct and gfn.

root_count: A counter keeping track of how many hardware registers (guest cr3 or pdptrs) are now pointing at the page. While this counter is nonzero, the page cannot be destroyed. See role.invalid.

parent_ptes: The reverse mapping for the pte/ptes pointing at this page's spt. If parent_ptes bit 0 is zero, only one spte points at this page and parent_ptes points at this single spte, otherwise, there exists multiple sptes pointing at this page and (parent_ptes & ~0x1) points at a data structure with a list of parent sptes.

unsync: If true, then the translations in this page may not match the guest's translation. This is equivalent to the state of the tlb when a pte is changed but before the tlb entry is flushed. Accordingly, unsync ptes are synchronized when the guest executes invlpg or flushes its tlb by other means. Valid for leaf pages.

unsync_children: How many sptes in the page point at pages that are unsync (or have unsynchronized children).

unsync_child_bitmap: A bitmap indicating which sptes in spt point (directly or indirectly) at pages that may be unsynchronized. Used to quickly locate all unsychronized pages reachable from a given page.

clear_spte_count: Only present on 32-bit hosts, where a 64-bit spte cannot be written atomically. The reader uses this while running out of the MMU lock to detect in-progress updates and retry them until the writer has finished the write.

write_flooding_count: A guest may write to a page table many times, causing a lot of emulations if the page needs to be write-protected (see "Synchronized and unsynchronized pages" below). Leaf pages can be unsyn-

chronized so that they do not trigger frequent emulation, but this is not possible for non-leafs. This field counts the number of emulations since the last time the page table was actually used; if emulation is triggered too frequently on this page, KVM will unmap the page to avoid emulation in the future.

1.7.7 Reverse map

The mmu maintains a reverse mapping whereby all ptes mapping a page can be reached given its gfn. This is used, for example, when swapping out a page.

1.7.8 Synchronized and unsynchronized pages

The guest uses two events to synchronize its tlb and page tables: tlb flushes and page invalidations (invlpg).

A tlb flush means that we need to synchronize all sptes reachable from the guest' s cr3. This is expensive, so we keep all guest page tables write protected, and synchronize sptes to gptes when a gpte is written.

A special case is when a guest page table is reachable from the current guest cr3. In this case, the guest is obliged to issue an invlpg instruction before using the translation. We take advantage of that by removing write protection from the guest page, and allowing the guest to modify it freely. We synchronize modified gptes when the guest invokes invlpg. This reduces the amount of emulation we have to do when the guest modifies multiple gptes, or when the a guest page is no longer used as a page table and is used for random guest data.

As a side effect we have to resynchronize all reachable unsynchronized shadow pages on a tlb flush.

1.7.9 Reaction to events

- guest page fault (or npt page fault, or ept violation)

This is the most complicated event. The cause of a page fault can be:

- a true guest fault (the guest translation won' t allow the access) (*)
- access to a missing translation
- access to a protected translation - when logging dirty pages, memory is write protected - synchronized shadow pages are write protected (*)
- access to untranslatable memory (mmio)

(*) not applicable in direct mode

Handling a page fault is performed as follows:

- if the RSV bit of the error code is set, the page fault is caused by guest accessing MMIO and cached MMIO information is available.
  - walk shadow page table

  - check for valid generation number in the spte (see "Fast invalidation of MMIO sptes" below)
  - cache the information to vcpu->arch.mmio_gva, vcpu->arch.mmio_access and vcpu->arch.mmio_gfn, and call the emulator
- If both P bit and R/W bit of error code are set, this could possibly be handled as a "fast page fault" (fixed without taking the MMU lock). See the description in Documentation/virt/kvm/locking.rst.
- if needed, walk the guest page tables to determine the guest translation (gva->gpa or ngpa->gpa)
  - if permissions are insufficient, reflect the fault back to the guest
- determine the host page
  - if this is an mmio request, there is no host page; cache the info to vcpu->arch.mmio_gva, vcpu->arch.mmio_access and vcpu->arch.mmio_gfn
- walk the shadow page table to find the spte for the translation, instantiating missing intermediate page tables as necessary
  - If this is an mmio request, cache the mmio info to the spte and set some reserved bit on the spte (see callers of kvm_mmu_set_mmio_spte_mask)
- try to unsynchronize the page
  - if successful, we can let the guest continue and modify the gpte
- emulate the instruction
  - if failed, unshadow the page and let the guest continue
- update any translations that were modified by the instruction

invlpg handling:

- walk the shadow page hierarchy and drop affected translations
- try to reinstantiate the indicated translation in the hope that the guest will use it in the near future

Guest control register updates:

- mov to cr3
  - look up new shadow roots
  - synchronize newly reachable shadow pages
- mov to cr0/cr4/efer
  - set up mmu context for new paging mode
  - look up new shadow roots
  - synchronize newly reachable shadow pages

Host translation updates:

- mmu notifier called with updated hva
- look up affected sptes through reverse map

- drop (or update) translations

1.7.10 Emulating cr0.wp

If tdp is not enabled, the host must keep cr0.wp=1 so page write protection works for the guest kernel, not guest guest userspace. When the guest cr0.wp=1, this does not present a problem. However when the guest cr0.wp=0, we cannot map the permissions for gpte.u=1, gpte.w=0 to any spte (the semantics require allowing any guest kernel access plus user read access).

We handle this by mapping the permissions to two possible sptes, depending on fault type:

- kernel write fault: spte.u=0, spte.w=1 (allows full kernel access, disallows user access)
- read fault: spte.u=1, spte.w=0 (allows full read access, disallows kernel write access)

(user write faults generate a #PF)

In the first case there are two additional complications:

- if CR4.SMEP is enabled: since we've turned the page into a kernel page, the kernel may now execute it. We handle this by also setting spte.nx. If we get a user fetch or read fault, we'll change spte.u=1 and spte.nx=gpte.nx back. For this to work, KVM forces EFER.NX to 1 when shadow paging is in use.
- if CR4.SMAP is disabled: since the page has been changed to a kernel page, it can not be reused when CR4.SMAP is enabled. We set CR4.SMAP && !CR0.WP into shadow page's role to avoid this case. Note, here we do not care the case that CR4.SMAP is enabled since KVM will directly inject #PF to guest due to failed permission check.

To prevent an spte that was converted into a kernel page with cr0.wp=0 from being written by the kernel after cr0.wp has changed to 1, we make the value of cr0.wp part of the page role. This means that an spte created with one value of cr0.wp cannot be used when cr0.wp has a different value - it will simply be missed by the shadow page lookup code. A similar issue exists when an spte created with cr0.wp=0 and cr4.smep=0 is used after changing cr4.smep to 1. To avoid this, the value of !cr0.wp && cr4.smep is also made a part of the page role.

1.7.11 Large pages

The mmu supports all combinations of large and small guest and host pages. Supported page sizes include 4k, 2M, 4M, and 1G. 4M pages are treated as two separate 2M pages, on both guest and host, since the mmu always uses PAE paging.

To instantiate a large spte, four constraints must be satisfied:

- the spte must point to a large host page
- the guest pte must be a large pte of at least equivalent size (if tdp is enabled, there is no guest pte and this condition is satisfied)

- if the spte will be writeable, the large page frame may not overlap any write-protected pages
- the guest page must be wholly contained by a single memory slot

To check the last two conditions, the mmu maintains a ->disallow_lpage set of arrays for each memory slot and large page size. Every write protected page causes its disallow_lpage to be incremented, thus preventing instantiation of a large spte. The frames at the end of an unaligned memory slot have artificially inflated ->disallow_lpages so they can never be instantiated.

1.7.12 Fast invalidation of MMIO sptes

As mentioned in "Reaction to events" above, kvm will cache MMIO information in leaf sptes. When a new memslot is added or an existing memslot is changed, this information may become stale and needs to be invalidated. This also needs to hold the MMU lock while walking all shadow pages, and is made more scalable with a similar technique.

MMIO sptes have a few spare bits, which are used to store a generation number. The global generation number is stored in kvm_memslots(kvm)->generation, and increased whenever guest memory info changes.

When KVM finds an MMIO spte, it checks the generation number of the spte. If the generation number of the spte does not equal the global generation number, it will ignore the cached MMIO information and handle the page fault through the slow path.

Since only 19 bits are used to store generation-number on mmio spte, all pages are zapped when there is an overflow.

Unfortunately, a single memory access might access kvm_memslots(kvm) multiple times, the last one happening when the generation number is retrieved and stored into the MMIO spte. Thus, the MMIO spte might be created based on out-of-date information, but with an up-to-date generation number.

To avoid this, the generation number is incremented again after synchronize_srcu returns; thus, bit 63 of kvm_memslots(kvm)->generation set to 1 only during a memslot update, while some SRCU readers might be using the old copy. We do not want to use an MMIO sptes created with an odd generation number, and we can do this without losing a bit in the MMIO spte. The "update in-progress" bit of the generation is not stored in MMIO spte, and is so is implicitly zero when the generation is extracted out of the spte. If KVM is unlucky and creates an MMIO spte while an update is in-progress, the next access to the spte will always be a cache miss. For example, a subsequent access during the update window will miss due to the in-progress flag diverging, while an access after the update window closes will have a higher generation number (as compared to the spte).

1.7.13 Further reading

- NPT presentation from KVM Forum 2008 http://www.linux-kvm.org/images/c/c8/KvmForum2008%24kdf2008_21.pdf

1.8 KVM-specific MSRs

Author Glauber Costa <glommer@redhat.com>, Red Hat Inc, 2010

KVM makes use of some custom MSRs to service some requests.

Custom MSRs have a range reserved for them, that goes from 0x4b564d00 to 0x4b564dff. There are MSRs outside this area, but they are deprecated and their use is discouraged.

1.8.1 Custom MSR list

The current supported Custom MSR list is:

MSR_KVM_WALL_CLOCK_NEW: 0x4b564d00

data: 4-byte alignment physical address of a memory area which must be in guest RAM. This memory is expected to hold a copy of the following structure:

```
struct pvclock_wall_clock {
       u32   version;
       u32   sec;
       u32   nsec;
 } __attribute__((__packed__));
```

whose data will be filled in by the hypervisor. The hypervisor is only guaranteed to update this data at the moment of MSR write. Users that want to reliably query this information more than once have to write more than once to this MSR. Fields have the following meanings:

version: guest has to check version before and after grabbing time information and check that they are both equal and even. An odd version indicates an in-progress update.

sec: number of seconds for wallclock at time of boot.

nsec: number of nanoseconds for wallclock at time of boot.

In order to get the current wallclock time, the system_time from MSR_KVM_SYSTEM_TIME_NEW needs to be added.

Note that although MSRs are per-CPU entities, the effect of this particular MSR is global.

Availability of this MSR must be checked via bit 3 in 0x4000001 cpuid leaf prior to usage.

MSR_KVM_SYSTEM_TIME_NEW: 0x4b564d01

data: 4-byte aligned physical address of a memory area which must be in guest RAM, plus an enable bit in bit 0. This memory is expected to hold a copy of the following structure:

```
struct pvclock_vcpu_time_info {
      u32   version;
      u32   pad0;
      u64   tsc_timestamp;
      u64   system_time;
      u32   tsc_to_system_mul;
      s8    tsc_shift;
      u8    flags;
      u8    pad[2];
} __attribute__((__packed__)); /* 32 bytes */
```

whose data will be filled in by the hypervisor periodically. Only one write, or registration, is needed for each VCPU. The interval between updates of this structure is arbitrary and implementation-dependent. The hypervisor may update this structure at any time it sees fit until anything with bit0 == 0 is written to it.

Fields have the following meanings:

version: guest has to check version before and after grabbing time information and check that they are both equal and even. An odd version indicates an in-progress update.

tsc_timestamp: the tsc value at the current VCPU at the time of the update of this structure. Guests can subtract this value from current tsc to derive a notion of elapsed time since the structure update.

system_time: a host notion of monotonic time, including sleep time at the time this structure was last updated. Unit is nanoseconds.

tsc_to_system_mul: multiplier to be used when converting tsc-related quantity to nanoseconds

tsc_shift: shift to be used when converting tsc-related quantity to nanoseconds. This shift will ensure that multiplication with tsc_to_system_mul does not overflow. A positive value denotes a left shift, a negative value a right shift.

The conversion from tsc to nanoseconds involves an additional right shift by 32 bits. With this information, guests can derive per-CPU time by doing:

```
time = (current_tsc - tsc_timestamp)
if (tsc_shift >= 0)
        time <<= tsc_shift;
else
        time >>= -tsc_shift;
time = (time * tsc_to_system_mul) >> 32
time = time + system_time
```

flags: bits in this field indicate extended capabilities coordinated between the guest and the hypervisor. Availability of specific flags has to be checked in 0x40000001 cpuid leaf. Current flags are:

flag bit	cpuid bit	meaning
0	24	time measures taken across multiple cpus are guaranteed to be monotonic
1	N/A	guest vcpu has been paused by the host See 4.70 in api.txt

Availability of this MSR must be checked via bit 3 in 0x4000001 cpuid leaf prior to usage.

MSR_KVM_WALL_CLOCK: 0x11

data and functioning: same as MSR_KVM_WALL_CLOCK_NEW. Use that instead.

This MSR falls outside the reserved KVM range and may be removed in the future. Its usage is deprecated.

Availability of this MSR must be checked via bit 0 in 0x4000001 cpuid leaf prior to usage.

MSR_KVM_SYSTEM_TIME: 0x12

data and functioning: same as MSR_KVM_SYSTEM_TIME_NEW. Use that instead.

This MSR falls outside the reserved KVM range and may be removed in the future. Its usage is deprecated.

Availability of this MSR must be checked via bit 0 in 0x4000001 cpuid leaf prior to usage.

The suggested algorithm for detecting kvmclock presence is then:

```
if (!kvm_para_available())    /* refer to cpuid.txt */
        return NON_PRESENT;

flags = cpuid_eax(0x40000001);
if (flags & 3) {
        msr_kvm_system_time = MSR_KVM_SYSTEM_TIME_NEW;
        msr_kvm_wall_clock = MSR_KVM_WALL_CLOCK_NEW;
        return PRESENT;
} else if (flags & 0) {
        msr_kvm_system_time = MSR_KVM_SYSTEM_TIME;
        msr_kvm_wall_clock = MSR_KVM_WALL_CLOCK;
        return PRESENT;
} else
        return NON_PRESENT;
```

MSR_KVM_ASYNC_PF_EN: 0x4b564d02

data: Asynchronous page fault (APF) control MSR.

Bits 63-6 hold 64-byte aligned physical address of a 64 byte memory area which must be in guest RAM and must be zeroed. This memory is expected to hold a copy of the following structure:

```
struct kvm_vcpu_pv_apf_data {
      /* Used for 'page not present' events delivered via #PF */
      __u32 flags;

      /* Used for 'page ready' events delivered via interrupt␣
↪notification */
      __u32 token;

      __u8 pad[56];
      __u32 enabled;
};
```

Bits 5-4 of the MSR are reserved and should be zero. Bit 0 is set to 1 when asynchronous page faults are enabled on the vcpu, 0 when disabled. Bit 1 is 1 if asynchronous page faults can be injected when vcpu is in cpl == 0. Bit 2 is 1 if asynchronous page faults are delivered to L1 as #PF vmexits. Bit 2 can be set only if KVM_FEATURE_ASYNC_PF_VMEXIT is present in CPUID. Bit 3 enables interrupt based delivery of 'page ready' events. Bit 3 can only be set if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

'Page not present' events are currently always delivered as synthetic #PF exception. During delivery of these events APF CR2 register contains a token that will be used to notify the guest when missing page becomes available. Also, to make it possible to distinguish between real #PF and APF, first 4 bytes of 64 byte memory location ('flags') will be written to by the hypervisor at the time of injection. Only first bit of 'flags' is currently supported, when set, it indicates that the guest is dealing with asynchronous 'page not present' event. If during a page fault APF 'flags' is '0' it means that this is regular page fault. Guest is supposed to clear 'flags' when it is done handling #PF exception so the next event can be delivered.

Note, since APF 'page not present' events use the same exception vector as regular page fault, guest must reset 'flags' to '0' before it does something that can generate normal page fault.

Bytes 5-7 of 64 byte memory location ('token') will be written to by the hypervisor at the time of APF 'page ready' event injection. The content of these bytes is a token which was previously delivered as 'page not present' event. The event indicates the page in now available. Guest is supposed to write '0' to 'token' when it is done handling 'page ready' event and to write 1' to MSR_KVM_ASYNC_PF_ACK after clearing the location; writing to the MSR forces KVM to re-scan its queue and deliver the next pending notification.

Note, MSR_KVM_ASYNC_PF_INT MSR specifying the interrupt vector for 'page ready' APF delivery needs to be written to before enabling APF mechanism in MSR_KVM_ASYNC_PF_EN or interrupt #0 can get injected. The MSR is available if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

Note, previously, 'page ready' events were delivered via the same #PF exception as 'page not present' events but this is now deprecated. If bit 3 (interrupt based delivery) is not set APF events are not delivered.

If APF is disabled while there are outstanding APFs, they will not be delivered.

Currently 'page ready' APF events will be always delivered on the same vcpu

as 'page not present' event was, but guest should not rely on that.

MSR_KVM_STEAL_TIME: 0x4b564d03

data: 64-byte alignment physical address of a memory area which must be in guest RAM, plus an enable bit in bit 0. This memory is expected to hold a copy of the following structure:

```
struct kvm_steal_time {
      __u64 steal;
      __u32 version;
      __u32 flags;
      __u8  preempted;
      __u8  u8_pad[3];
      __u32 pad[11];
}
```

whose data will be filled in by the hypervisor periodically. Only one write, or registration, is needed for each VCPU. The interval between updates of this structure is arbitrary and implementation-dependent. The hypervisor may update this structure at any time it sees fit until anything with bit0 == 0 is written to it. Guest is required to make sure this structure is initialized to zero.

Fields have the following meanings:

version: a sequence counter. In other words, guest has to check this field before and after grabbing time information and make sure they are both equal and even. An odd version indicates an in-progress update.

flags: At this point, always zero. May be used to indicate changes in this structure in the future.

steal: the amount of time in which this vCPU did not run, in nanoseconds. Time during which the vcpu is idle, will not be reported as steal time.

preempted: indicate the vCPU who owns this struct is running or not. Non-zero values mean the vCPU has been preempted. Zero means the vCPU is not preempted. NOTE, it is always zero if the the hypervisor doesn't support this field.

MSR_KVM_EOI_EN: 0x4b564d04

data: Bit 0 is 1 when PV end of interrupt is enabled on the vcpu; 0 when disabled. Bit 1 is reserved and must be zero. When PV end of interrupt is enabled (bit 0 set), bits 63-2 hold a 4-byte aligned physical address of a 4 byte memory area which must be in guest RAM and must be zeroed.

The first, least significant bit of 4 byte memory location will be written to by the hypervisor, typically at the time of interrupt injection. Value of 1 means that guest can skip writing EOI to the apic (using MSR or MMIO write); instead, it is sufficient to signal EOI by clearing the bit in guest memory - this location will later be polled by the hypervisor. Value of 0 means that the EOI write is required.

It is always safe for the guest to ignore the optimization and perform the APIC EOI write anyway.

Hypervisor is guaranteed to only modify this least significant bit while in the current VCPU context, this means that guest does not need to use either lock prefix or memory ordering primitives to synchronise with the hypervisor.

However, hypervisor can set and clear this memory bit at any time: therefore to make sure hypervisor does not interrupt the guest and clear the least significant bit in the memory area in the window between guest testing it to detect whether it can skip EOI apic write and between guest clearing it to signal EOI to the hypervisor, guest must both read the least significant bit in the memory area and clear it using a single CPU instruction, such as test and clear, or compare and exchange.

MSR_KVM_POLL_CONTROL: 0x4b564d05

Control host-side polling.

data: Bit 0 enables (1) or disables (0) host-side HLT polling logic.

KVM guests can request the host not to poll on HLT, for example if they are performing polling themselves.

MSR_KVM_ASYNC_PF_INT: 0x4b564d06

data: Second asynchronous page fault (APF) control MSR.

Bits 0-7: APIC vector for delivery of 'page ready' APF events. Bits 8-63: Reserved

Interrupt vector for asynchnonous 'page ready' notifications delivery. The vector has to be set up before asynchronous page fault mechanism is enabled in MSR_KVM_ASYNC_PF_EN. The MSR is only available if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

MSR_KVM_ASYNC_PF_ACK: 0x4b564d07

data: Asynchronous page fault (APF) acknowledgment.

When the guest is done processing 'page ready' APF event and 'token' field in 'struct kvm_vcpu_pv_apf_data' is cleared it is supposed to write '1' to bit 0 of the MSR, this causes the host to re-scan its queue and check if there are more notifications pending. The MSR is available if KVM_FEATURE_ASYNC_PF_INT is present in CPUID.

1.9 Nested VMX

1.9.1 Overview

On Intel processors, KVM uses Intel's VMX (Virtual-Machine eXtensions) to easily and efficiently run guest operating systems. Normally, these guests cannot themselves be hypervisors running their own guests, because in VMX, guests cannot use VMX instructions.

The "Nested VMX" feature adds this missing capability - of running guest hypervisors (which use VMX) with their own nested guests. It does so by allowing a guest to use VMX instructions, and correctly and efficiently emulating them using the single level of VMX available in the hardware.

We describe in much greater detail the theory behind the nested VMX feature, its implementation and its performance characteristics, in the OSDI 2010 paper "The Turtles Project: Design and Implementation of Nested Virtualization", available at:

http://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf

1.9.2 Terminology

Single-level virtualization has two levels - the host (KVM) and the guests. In nested virtualization, we have three levels: The host (KVM), which we call L0, the guest hypervisor, which we call L1, and its nested guest, which we call L2.

1.9.3 Running nested VMX

The nested VMX feature is disabled by default. It can be enabled by giving the "nested=1" option to the kvm-intel module.

No modifications are required to user space (qemu). However, qemu's default emulated CPU type (qemu64) does not list the "VMX" CPU feature, so it must be explicitly enabled, by giving qemu one of the following options:

- cpu host (emulated CPU has all features of the real CPU)
- cpu qemu64,+vmx (add just the vmx feature to a named CPU type)

1.9.4 ABIs

Nested VMX aims to present a standard and (eventually) fully-functional VMX implementation for the a guest hypervisor to use. As such, the official specification of the ABI that it provides is Intel's VMX specification, namely volume 3B of their "Intel 64 and IA-32 Architectures Software Developer's Manual". Not all of VMX's features are currently fully supported, but the goal is to eventually support them all, starting with the VMX features which are used in practice by popular hypervisors (KVM and others).

As a VMX implementation, nested VMX presents a VMCS structure to L1. As mandated by the spec, other than the two fields revision_id and abort, this structure is opaque to its user, who is not supposed to know or care about its internal structure. Rather, the structure is accessed through the VMREAD and VMWRITE instructions. Still, for debugging purposes, KVM developers might be interested to know the internals of this structure; This is struct vmcs12 from arch/x86/kvm/vmx.c.

The name "vmcs12" refers to the VMCS that L1 builds for L2. In the code we also have "vmcs01", the VMCS that L0 built for L1, and "vmcs02" is the VMCS which L0 builds to actually run L2 - how this is done is explained in the aforementioned paper.

For convenience, we repeat the content of struct vmcs12 here. If the internals of this structure changes, this can break live migration across KVM versions. VMCS12_REVISION (from vmx.c) should be changed if struct vmcs12 or its inner struct shadow_vmcs is ever changed.

```
typedef u64 natural_width;
struct __packed vmcs12 {
        /* According to the Intel spec, a VMCS region must start with
         * these two user-visible fields */
        u32 revision_id;
        u32 abort;

        u32 launch_state; /* set to 0 by VMCLEAR, to 1 by VMLAUNCH */
        u32 padding[7]; /* room for future expansion */

        u64 io_bitmap_a;
        u64 io_bitmap_b;
        u64 msr_bitmap;
        u64 vm_exit_msr_store_addr;
        u64 vm_exit_msr_load_addr;
        u64 vm_entry_msr_load_addr;
        u64 tsc_offset;
        u64 virtual_apic_page_addr;
        u64 apic_access_addr;
        u64 ept_pointer;
        u64 guest_physical_address;
        u64 vmcs_link_pointer;
        u64 guest_ia32_debugctl;
        u64 guest_ia32_pat;
        u64 guest_ia32_efer;
        u64 guest_pdptr0;
        u64 guest_pdptr1;
        u64 guest_pdptr2;
        u64 guest_pdptr3;
        u64 host_ia32_pat;
        u64 host_ia32_efer;
        u64 padding64[8]; /* room for future expansion */
        natural_width cr0_guest_host_mask;
        natural_width cr4_guest_host_mask;
        natural_width cr0_read_shadow;
        natural_width cr4_read_shadow;
        natural_width dead_space[4]; /* Last remnants of cr3_target_
↪value[0-3]. */
        natural_width exit_qualification;
        natural_width guest_linear_address;
        natural_width guest_cr0;
        natural_width guest_cr3;
        natural_width guest_cr4;
        natural_width guest_es_base;
        natural_width guest_cs_base;
        natural_width guest_ss_base;
        natural_width guest_ds_base;
        natural_width guest_fs_base;
        natural_width guest_gs_base;
        natural_width guest_ldtr_base;
        natural_width guest_tr_base;
        natural_width guest_gdtr_base;
        natural_width guest_idtr_base;
        natural_width guest_dr7;
        natural_width guest_rsp;
        natural_width guest_rip;
```

(continues on next page)

(continued from previous page)

```
        natural_width guest_rflags;
        natural_width guest_pending_dbg_exceptions;
        natural_width guest_sysenter_esp;
        natural_width guest_sysenter_eip;
        natural_width host_cr0;
        natural_width host_cr3;
        natural_width host_cr4;
        natural_width host_fs_base;
        natural_width host_gs_base;
        natural_width host_tr_base;
        natural_width host_gdtr_base;
        natural_width host_idtr_base;
        natural_width host_ia32_sysenter_esp;
        natural_width host_ia32_sysenter_eip;
        natural_width host_rsp;
        natural_width host_rip;
        natural_width paddingl[8]; /* room for future expansion */
        u32 pin_based_vm_exec_control;
        u32 cpu_based_vm_exec_control;
        u32 exception_bitmap;
        u32 page_fault_error_code_mask;
        u32 page_fault_error_code_match;
        u32 cr3_target_count;
        u32 vm_exit_controls;
        u32 vm_exit_msr_store_count;
        u32 vm_exit_msr_load_count;
        u32 vm_entry_controls;
        u32 vm_entry_msr_load_count;
        u32 vm_entry_intr_info_field;
        u32 vm_entry_exception_error_code;
        u32 vm_entry_instruction_len;
        u32 tpr_threshold;
        u32 secondary_vm_exec_control;
        u32 vm_instruction_error;
        u32 vm_exit_reason;
        u32 vm_exit_intr_info;
        u32 vm_exit_intr_error_code;
        u32 idt_vectoring_info_field;
        u32 idt_vectoring_error_code;
        u32 vm_exit_instruction_len;
        u32 vmx_instruction_info;
        u32 guest_es_limit;
        u32 guest_cs_limit;
        u32 guest_ss_limit;
        u32 guest_ds_limit;
        u32 guest_fs_limit;
        u32 guest_gs_limit;
        u32 guest_ldtr_limit;
        u32 guest_tr_limit;
        u32 guest_gdtr_limit;
        u32 guest_idtr_limit;
        u32 guest_es_ar_bytes;
        u32 guest_cs_ar_bytes;
        u32 guest_ss_ar_bytes;
        u32 guest_ds_ar_bytes;
        u32 guest_fs_ar_bytes;
```

(continues on next page)

(continued from previous page)

```
        u32 guest_gs_ar_bytes;
        u32 guest_ldtr_ar_bytes;
        u32 guest_tr_ar_bytes;
        u32 guest_interruptibility_info;
        u32 guest_activity_state;
        u32 guest_sysenter_cs;
        u32 host_ia32_sysenter_cs;
        u32 padding32[8]; /* room for future expansion */
        u16 virtual_processor_id;
        u16 guest_es_selector;
        u16 guest_cs_selector;
        u16 guest_ss_selector;
        u16 guest_ds_selector;
        u16 guest_fs_selector;
        u16 guest_gs_selector;
        u16 guest_ldtr_selector;
        u16 guest_tr_selector;
        u16 host_es_selector;
        u16 host_cs_selector;
        u16 host_ss_selector;
        u16 host_ds_selector;
        u16 host_fs_selector;
        u16 host_gs_selector;
        u16 host_tr_selector;
};
```

1.9.5 Authors

These patches were written by:

- Abel Gordon, abelg <at> il.ibm.com
- Nadav Har’ El, nyh <at> il.ibm.com
- Orit Wasserman, oritw <at> il.ibm.com
- Ben-Ami Yassor, benami <at> il.ibm.com
- Muli Ben-Yehuda, muli <at> il.ibm.com

With contributions by:

- Anthony Liguori, aliguori <at> us.ibm.com
- Mike Day, mdday <at> us.ibm.com
- Michael Factor, factor <at> il.ibm.com
- Zvi Dubitzky, dubi <at> il.ibm.com

And valuable reviews by:

- Avi Kivity, avi <at> redhat.com
- Gleb Natapov, gleb <at> redhat.com
- Marcelo Tosatti, mtosatti <at> redhat.com
- Kevin Tian, kevin.tian <at> intel.com

- and others.

1.10 The PPC KVM paravirtual interface

The basic execution principle by which KVM on PowerPC works is to run all kernel space code in PR=1 which is user space. This way we trap all privileged instructions and can emulate them accordingly.

Unfortunately that is also the downfall. There are quite some privileged instructions that needlessly return us to the hypervisor even though they could be handled differently.

This is what the PPC PV interface helps with. It takes privileged instructions and transforms them into unprivileged ones with some help from the hypervisor. This cuts down virtualization costs by about 50% on some of my benchmarks.

The code for that interface can be found in arch/powerpc/kernel/kvm*

1.10.1 Querying for existence

To find out if we’re running on KVM or not, we leverage the device tree. When Linux is running on KVM, a node /hypervisor exists. That node contains a compatible property with the value “linux,kvm”.

Once you determined you’re running under a PV capable KVM, you can now use hypercalls as described below.

1.10.2 KVM hypercalls

Inside the device tree’s /hypervisor node there’s a property called ‘hypercall-instructions’. This property contains at most 4 opcodes that make up the hypercall. To call a hypercall, just call these instructions.

The parameters are as follows:

Register	IN	OUT
r0	•	volatile
r3	1st parameter	Return code
r4	2nd parameter	1st output value
r5	3rd parameter	2nd output value
r6	4th parameter	3rd output value
r7	5th parameter	4th output value
r8	6th parameter	5th output value
r9	7th parameter	6th output value
r10	8th parameter	7th output value
r11	hypercall number	8th output value
r12	•	volatile

Hypercall definitions are shared in generic code, so the same hypercall numbers apply for x86 and powerpc alike with the exception that each KVM hypercall also needs to be ORed with the KVM vendor code which is (42 << 16).

Return codes can be as follows:

Code	Meaning
0	Success
12	Hypercall not implemented
<0	Error

1.10.3 The magic page

To enable communication between the hypervisor and guest there is a new shared page that contains parts of supervisor visible register state. The guest can map this shared page using the KVM hypercall KVM_HC_PPC_MAP_MAGIC_PAGE.

With this hypercall issued the guest always gets the magic page mapped at the desired location. The first parameter indicates the effective address when the MMU is enabled. The second parameter indicates the address in real mode, if applicable to the target. For now, we always map the page to -4096. This way we can access it using absolute load and store functions. The following instruction reads the first field of the magic page:

```
ld      rX, -4096(0)
```

The interface is designed to be extensible should there be need later to add additional registers to the magic page. If you add fields to the magic page, also define a new hypercall feature to indicate that the host can give you more registers. Only if the host supports the additional features, make use of them.

The magic page layout is described by struct kvm_vcpu_arch_shared in arch/powerpc/include/asm/kvm_para.h.

1.10.4 Magic page features

When mapping the magic page using the KVM hypercall KVM_HC_PPC_MAP_MAGIC_PAGE, a second return value is passed to the guest. This second return value contains a bitmap of available features inside the magic page.

The following enhancements to the magic page are currently available:

KVM_MAGIC_FEAT_SR	Maps SR registers r/w in the magic page
KVM_MAGIC_FEAT_MAS0_TO_SPRG7	Maps MASn, ESR, PIR and high SPRGs

For enhanced features in the magic page, please check for the existence of the feature before using them!

1.10.5 Magic page flags

In addition to features that indicate whether a host is capable of a particular feature we also have a channel for a guest to tell the guest whether it's capable of something. This is what we call "flags".

Flags are passed to the host in the low 12 bits of the Effective Address.

The following flags are currently available for a guest to expose:

MAGIC_PAGE_FLAG_NOT_MAPPED_NX Guest handles NX bits correctly wrt magic page

1.10.6 MSR bits

The MSR contains bits that require hypervisor intervention and bits that do not require direct hypervisor intervention because they only get interpreted when entering the guest or don't have any impact on the hypervisor's behavior.

The following bits are safe to be set inside the guest:

- MSR_EE
- MSR_RI

If any other bit changes in the MSR, please still use mtmsr(d).

1.10.7 Patched instructions

The "ld" and "std" instructions are transformed to "lwz" and "stw" instructions respectively on 32 bit systems with an added offset of 4 to accommodate for big endianness.

The following is a list of mapping the Linux kernel performs when running as guest. Implementing any of those mappings is optional, as the instruction traps also act on the shared page. So calling privileged instructions still works as before.

From	To
mfmsr rX	ld rX, magic_page->msr
mfsprg rX, 0	ld rX, magic_page->sprg0
mfsprg rX, 1	ld rX, magic_page->sprg1
mfsprg rX, 2	ld rX, magic_page->sprg2
mfsprg rX, 3	ld rX, magic_page->sprg3
mfsrr0 rX	ld rX, magic_page->srr0
mfsrr1 rX	ld rX, magic_page->srr1
mfdar rX	ld rX, magic_page->dar
mfdsisr rX	lwz rX, magic_page->dsisr
mtmsr rX	std rX, magic_page->msr
mtsprg 0, rX	std rX, magic_page->sprg0
mtsprg 1, rX	std rX, magic_page->sprg1
mtsprg 2, rX	std rX, magic_page->sprg2
mtsprg 3, rX	std rX, magic_page->sprg3
mtsrr0 rX	std rX, magic_page->srr0
mtsrr1 rX	std rX, magic_page->srr1
mtdar rX	std rX, magic_page->dar
mtdsisr rX	stw rX, magic_page->dsisr
tlbsync	nop
mtmsrd rX, 0	b <special mtmsr section>
mtmsr rX	b <special mtmsr section>
mtmsrd rX, 1	b <special mtmsrd section>
[Book3S only]	
mtsrin rX, rY	b <special mtsrin section>
[BookE only]	
wrteei [0|1]	b <special wrteei section>

Some instructions require more logic to determine what’s going on than a load or store instruction can deliver. To enable patching of those, we keep some RAM around where we can live translate instructions to. What happens is the following:

1) copy emulation code to memory
2) patch that code to fit the emulated instruction
3) patch that code to return to the original pc + 4
4) patch the original instruction to branch to the new code

That way we can inject an arbitrary amount of code as replacement for a single instruction. This allows us to check for pending interrupts when setting EE=1 for example.

1.10.8 Hypercall ABIs in KVM on PowerPC

1) KVM hypercalls (ePAPR)

These are ePAPR compliant hypercall implementation (mentioned above). Even generic hypercalls are implemented here, like the ePAPR idle hcall. These are available on all targets.

2) PAPR hypercalls

PAPR hypercalls are needed to run server PowerPC PAPR guests (-M pseries in QEMU). These are the same hypercalls that pHyp, the POWER hypervisor implements. Some of them are handled in the kernel, some are handled in user space. This is only available on book3s_64.

3) OSI hypercalls

Mac-on-Linux is another user of KVM on PowerPC, which has its own hypercall (long before KVM). This is supported to maintain compatibility. All these hypercalls get forwarded to user space. This is only useful on book3s_32, but can be used with book3s_64 as well.

1.11 The s390 DIAGNOSE call on KVM

KVM on s390 supports the DIAGNOSE call for making hypercalls, both for native hypercalls and for selected hypercalls found on other s390 hypervisors.

Note that bits are numbered as by the usual s390 convention (most significant bit on the left).

1.11.1 General remarks

DIAGNOSE calls by the guest cause a mandatory intercept. This implies all supported DIAGNOSE calls need to be handled by either KVM or its userspace.

All DIAGNOSE calls supported by KVM use the RS-a format:

```
-------------------------------------
|  '83'  | R1 | R3 | B2 |     D2     |
-------------------------------------
0        8    12   16   20           31
```

The second-operand address (obtained by the base/displacement calculation) is not used to address data. Instead, bits 48-63 of this address specify the function code, and bits 0-47 are ignored.

The supported DIAGNOSE function codes vary by the userspace used. For DIAGNOSE function codes not specific to KVM, please refer to the documentation for the s390 hypervisors defining them.

1.11.2 DIAGNOSE function code ‘X’ 500’ - KVM virtio functions

If the function code specifies 0x500, various virtio-related functions are performed.

General register 1 contains the virtio subfunction code. Supported virtio subfunctions depend on KVM’ s userspace. Generally, userspace provides either s390-virtio (subcodes 0-2) or virtio-ccw (subcode 3).

Upon completion of the DIAGNOSE instruction, general register 2 contains the function’ s return code, which is either a return code or a subcode specific value.

Subcode 0 - s390-virtio notification and early console printk Handled by userspace.

Subcode 1 - s390-virtio reset Handled by userspace.

Subcode 2 - s390-virtio set status Handled by userspace.

Subcode 3 - virtio-ccw notification Handled by either userspace or KVM (ioeventfd case).

General register 2 contains a subchannel-identification word denoting the subchannel of the virtio-ccw proxy device to be notified.

General register 3 contains the number of the virtqueue to be notified.

General register 4 contains a 64bit identifier for KVM usage (the kvm_io_bus cookie). If general register 4 does not contain a valid identifier, it is ignored.

After completion of the DIAGNOSE call, general register 2 may contain a 64bit identifier (in the kvm_io_bus cookie case), or a negative error value, if an internal error occurred.

See also the virtio standard for a discussion of this hypercall.

1.11.3 DIAGNOSE function code ‘X’ 501 - KVM breakpoint

If the function code specifies 0x501, breakpoint functions may be performed. This function code is handled by userspace.

This diagnose function code has no subfunctions and uses no parameters.

1.12 s390 (IBM Z) Ultravisor and Protected VMs

1.12.1 Summary

Protected virtual machines (PVM) are KVM VMs that do not allow KVM to access VM state like guest memory or guest registers. Instead, the PVMs are mostly managed by a new entity called Ultravisor (UV). The UV provides an API that can be used by PVMs and KVM to request management actions.

Each guest starts in non-protected mode and then may make a request to transition into protected mode. On transition, KVM registers the guest and its VCPUs with the Ultravisor and prepares everything for running it.

The Ultravisor will secure and decrypt the guest's boot memory (i.e. kernel/initrd). It will safeguard state changes like VCPU starts/stops and injected interrupts while the guest is running.

As access to the guest' s state, such as the SIE state description, is normally needed to be able to run a VM, some changes have been made in the behavior of the SIE instruction. A new format 4 state description has been introduced, where some fields have different meanings for a PVM. SIE exits are minimized as much as possible to improve speed and reduce exposed guest state.

1.12.2 Interrupt injection

Interrupt injection is safeguarded by the Ultravisor. As KVM doesn' t have access to the VCPUs' lowcores, injection is handled via the format 4 state description.

Machine check, external, IO and restart interruptions each can be injected on SIE entry via a bit in the interrupt injection control field (offset 0x54). If the guest cpu is not enabled for the interrupt at the time of injection, a validity interception is recognized. The format 4 state description contains fields in the interception data block where data associated with the interrupt can be transported.

Program and Service Call exceptions have another layer of safeguarding; they can only be injected for instructions that have been intercepted into KVM. The exceptions need to be a valid outcome of an instruction emulation by KVM, e.g. we can never inject a addressing exception as they are reported by SIE since KVM has no access to the guest memory.

1.12.3 Mask notification interceptions

KVM cannot intercept lctl(g) and lpsw(e) anymore in order to be notified when a PVM enables a certain class of interrupt. As a replacement, two new interception codes have been introduced: One indicating that the contents of CRs 0, 6, or 14 have been changed, indicating different interruption subclasses; and one indicating that PSW bit 13 has been changed, indicating that a machine check intervention was requested and those are now enabled.

1.12.4 Instruction emulation

With the format 4 state description for PVMs, the SIE instruction already interprets more instructions than it does with format 2. It is not able to interpret every instruction, but needs to hand some tasks to KVM; therefore, the SIE and the ultravisor safeguard emulation inputs and outputs.

The control structures associated with SIE provide the Secure Instruction Data Area (SIDA), the Interception Parameters (IP) and the Secure Interception General Register Save Area. Guest GRs and most of the instruction data, such as I/O data structures, are filtered. Instruction data is copied to and from the SIDA when needed. Guest GRs are put into / retrieved from the Secure Interception General Register Save Area.

Only GR values needed to emulate an instruction will be copied into this save area and the real register numbers will be hidden.

The Interception Parameters state description field still contains the the bytes of the instruction text, but with pre-set register values instead of the actual ones. I.e. each instruction always uses the same instruction text, in order not to leak guest instruction text. This also implies that the register content that a guest had in r<n> may be in r<m> from the hypervisor's point of view.

The Secure Instruction Data Area contains instruction storage data. Instruction data, i.e. data being referenced by an instruction like the SCCB for sclp, is moved via the SIDA. When an instruction is intercepted, the SIE will only allow data and program interrupts for this instruction to be moved to the guest via the two data areas discussed before. Other data is either ignored or results in validity interceptions.

1.12.5 Instruction emulation interceptions

There are two types of SIE secure instruction intercepts: the normal and the notification type. Normal secure instruction intercepts will make the guest pending for instruction completion of the intercepted instruction type, i.e. on SIE entry it is attempted to complete emulation of the instruction with the data provided by KVM. That might be a program exception or instruction completion.

The notification type intercepts inform KVM about guest environment changes due to guest instruction interpretation. Such an interception is recognized, for example, for the store prefix instruction to provide the new lowcore location. On SIE reentry, any KVM data in the data areas is ignored and execution continues as if the guest instruction had completed. For that reason KVM is not allowed to inject a program interrupt.

1.12.6 Links

KVM Forum 2019 presentation

1.13 s390 (IBM Z) Boot/IPL of Protected VMs

1.13.1 Summary

The memory of Protected Virtual Machines (PVMs) is not accessible to I/O or the hypervisor. In those cases where the hypervisor needs to access the memory of a PVM, that memory must be made accessible. Memory made accessible to the hypervisor will be encrypted. See s390 (IBM Z) Ultravisor and Protected VMs for details."

On IPL (boot) a small plaintext bootloader is started, which provides information about the encrypted components and necessary metadata to KVM to decrypt the protected virtual machine.

Based on this data, KVM will make the protected virtual machine known to the Ultravisor (UV) and instruct it to secure the memory of the PVM, decrypt the components and verify the data and address list hashes, to ensure integrity. Afterwards KVM can run the PVM via the SIE instruction which the UV will intercept and execute on KVM's behalf.

As the guest image is just like an opaque kernel image that does the switch into PV mode itself, the user can load encrypted guest executables and data via every available method (network, dasd, scsi, direct kernel, ⋯) without the need to change the boot process.

1.13.2 Diag308

This diagnose instruction is the basic mechanism to handle IPL and related operations for virtual machines. The VM can set and retrieve IPL information blocks, that specify the IPL method/devices and request VM memory and subsystem resets, as well as IPLs.

For PVMs this concept has been extended with new subcodes:

Subcode 8: Set an IPL Information Block of type 5 (information block for PVMs) Subcode 9: Store the saved block in guest memory Subcode 10: Move into Protected Virtualization mode

The new PV load-device-specific-parameters field specifies all data that is necessary to move into PV mode.

- PV Header origin
- PV Header length
- **List of Components composed of**
  - AES-XTS Tweak prefix
  - Origin
  - Size

The PV header contains the keys and hashes, which the UV will use to decrypt and verify the PV, as well as control flags and a start PSW.

The components are for instance an encrypted kernel, kernel parameters and initrd. The components are decrypted by the UV.

After the initial import of the encrypted data, all defined pages will contain the guest content. All non-specified pages will start out as zero pages on first access.

When running in protected virtualization mode, some subcodes will result in exceptions or return error codes.

Subcodes 4 and 7, which specify operations that do not clear the guest memory, will result in specification exceptions. This is because the UV will clear all memory when a secure VM is removed, and therefore non-clearing IPL subcodes are not allowed.

Subcodes 8, 9, 10 will result in specification exceptions. Re-IPL into a protected mode is only possible via a detour into non protected mode.

1.13.3 Keys

Every CEC will have a unique public key to enable tooling to build encrypted images. See s390-tools for the tooling.

1.14 Timekeeping Virtualization for X86-Based Architectures

Author Zachary Amsden <zamsden@redhat.com>

1.14.1 1. Overview

One of the most complicated parts of the X86 platform, and specifically, the virtualization of this platform is the plethora of timing devices available and the complexity of emulating those devices. In addition, virtualization of time introduces a new set of challenges because it introduces a multiplexed division of time beyond the control of the guest CPU.

First, we will describe the various timekeeping hardware available, then present some of the problems which arise and solutions available, giving specific recommendations for certain classes of KVM guests.

The purpose of this document is to collect data and information relevant to timekeeping which may be difficult to find elsewhere, specifically, information relevant to KVM and hardware-based virtualization.

1.14.2 2. Timing Devices

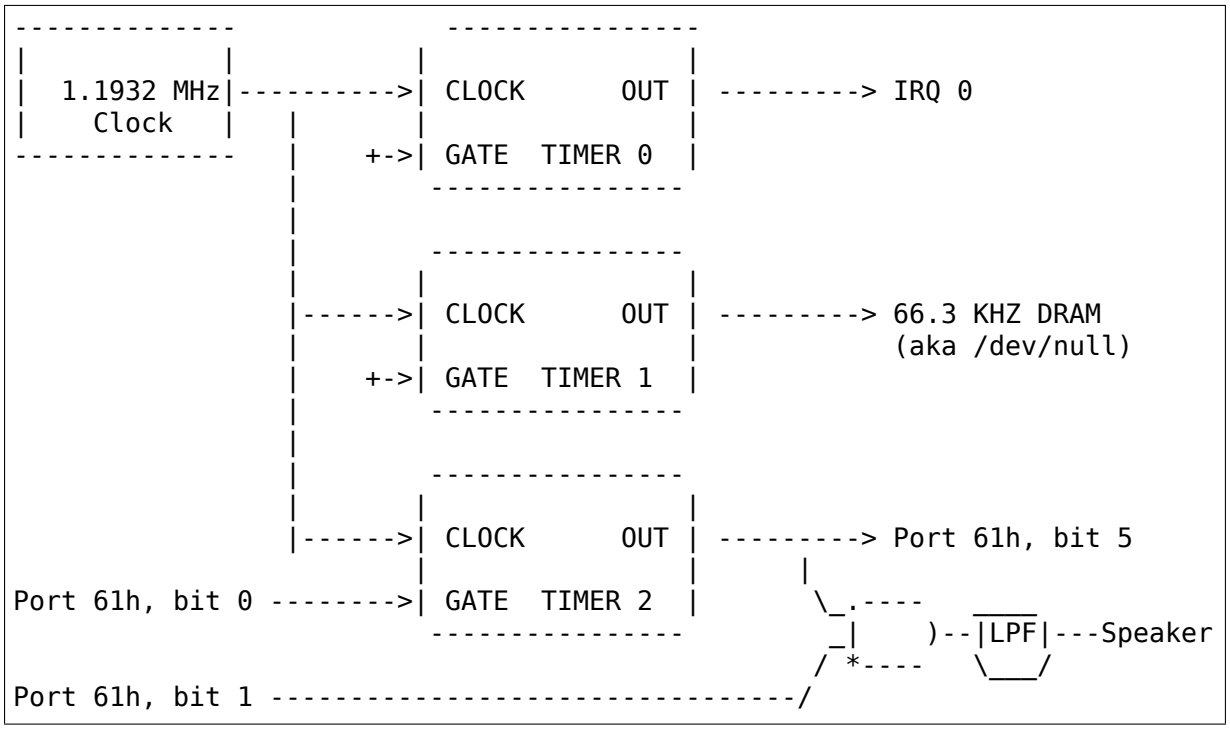
First we discuss the basic hardware devices available. TSC and the related KVM clock are special enough to warrant a full exposition and are described in the following section.

2.1. i8254 - PIT

One of the first timer devices available is the programmable interrupt timer, or PIT. The PIT has a fixed frequency 1.193182 MHz base clock and three channels which can be programmed to deliver periodic or one-shot interrupts. These three channels can be configured in different modes and have individual counters. Channel 1 and 2 were not available for general use in the original IBM PC, and historically were connected to control RAM refresh and the PC speaker. Now the PIT is typically integrated as part of an emulated chipset and a separate physical PIT is not used.

The PIT uses I/O ports 0x40 - 0x43. Access to the 16-bit counters is done using single or multiple byte access to the I/O ports. There are 6 modes available, but not all modes are available to all timers, as only timer 2 has a connected gate

input, required for modes 1 and 5. The gate line is controlled by port 61h, bit 0, as illustrated in the following diagram:



The timer modes are now described.

Mode 0: Single Timeout. This is a one-shot software timeout that counts down when the gate is high (always true for timers 0 and 1). When the count reaches zero, the output goes high.

Mode 1: Triggered One-shot. The output is initially set high. When the gate line is set high, a countdown is initiated (which does not stop if the gate is lowered), during which the output is set low. When the count reaches zero, the output goes high.

Mode 2: Rate Generator. The output is initially set high. When the countdown reaches 1, the output goes low for one count and then returns high. The value is reloaded and the countdown automatically resumes. If the gate line goes low, the count is halted. If the output is low when the gate is lowered, the output automatically goes high (this only affects timer 2).

Mode 3: Square Wave. This generates a high / low square wave. The count determines the length of the pulse, which alternates between high and low when zero is reached. The count only proceeds when gate is high and is automatically reloaded on reaching zero. The count is decremented twice at each clock to generate a full high / low cycle at the full periodic rate. If the count is even, the clock remains high for N/2 counts and low for N/2 counts; if the clock is odd, the clock is high for (N+1)/2 counts and low for (N-1)/2 counts. Only even values are latched by the counter, so odd values are not observed when reading. This is the intended mode for timer 2, which generates sine-like tones by low-pass filtering the square wave output.

Mode 4: Software Strobe. After programming this mode and loading the counter, the output remains high until the counter reaches zero. Then the output goes low for 1 clock cycle and returns high. The counter is not reloaded.

Counting only occurs when gate is high.

Mode 5: Hardware Strobe. After programming and loading the counter, the output remains high. When the gate is raised, a countdown is initiated (which does not stop if the gate is lowered). When the counter reaches zero, the output goes low for 1 clock cycle and then returns high. The counter is not reloaded.

In addition to normal binary counting, the PIT supports BCD counting. The command port, 0x43 is used to set the counter and mode for each of the three timers.

PIT commands, issued to port 0x43, using the following bit encoding:

```
Bit 7-4: Command (See table below)
Bit 3-1: Mode (000 = Mode 0, 101 = Mode 5, 11X = undefined)
Bit 0  : Binary (0) / BCD (1)
```

Command table:

```
0000 - Latch Timer 0 count for port 0x40
      sample and hold the count to be read in port 0x40;
      additional commands ignored until counter is read;
      mode bits ignored.

0001 - Set Timer 0 LSB mode for port 0x40
      set timer to read LSB only and force MSB to zero;
      mode bits set timer mode

0010 - Set Timer 0 MSB mode for port 0x40
      set timer to read MSB only and force LSB to zero;
      mode bits set timer mode

0011 - Set Timer 0 16-bit mode for port 0x40
      set timer to read / write LSB first, then MSB;
      mode bits set timer mode

0100 - Latch Timer 1 count for port 0x41 - as described above
0101 - Set Timer 1 LSB mode for port 0x41 - as described above
0110 - Set Timer 1 MSB mode for port 0x41 - as described above
0111 - Set Timer 1 16-bit mode for port 0x41 - as described above

1000 - Latch Timer 2 count for port 0x42 - as described above
1001 - Set Timer 2 LSB mode for port 0x42 - as described above
1010 - Set Timer 2 MSB mode for port 0x42 - as described above
1011 - Set Timer 2 16-bit mode for port 0x42 as described above

1101 - General counter latch
      Latch combination of counters into corresponding ports
      Bit 3 = Counter 2
      Bit 2 = Counter 1
      Bit 1 = Counter 0
      Bit 0 = Unused

1110 - Latch timer status
      Latch combination of counter mode into corresponding ports
      Bit 3 = Counter 2
      Bit 2 = Counter 1
```

(continues on next page)

(continued from previous page)

```
      Bit 1 = Counter 0

      The output of ports 0x40-0x42 following this command will be:

      Bit 7 = Output pin
      Bit 6 = Count loaded (0 if timer has expired)
      Bit 5-4 = Read / Write mode
          01 = MSB only
          10 = LSB only
          11 = LSB / MSB (16-bit)
      Bit 3-1 = Mode
      Bit 0 = Binary (0) / BCD mode (1)
```

2.2. RTC

The second device which was available in the original PC was the MC146818 real time clock. The original device is now obsolete, and usually emulated by the system chipset, sometimes by an HPET and some frankenstein IRQ routing.

The RTC is accessed through CMOS variables, which uses an index register to control which bytes are read. Since there is only one index register, read of the CMOS and read of the RTC require lock protection (in addition, it is dangerous to allow userspace utilities such as hwclock to have direct RTC access, as they could corrupt kernel reads and writes of CMOS memory).

The RTC generates an interrupt which is usually routed to IRQ 8. The interrupt can function as a periodic timer, an additional once a day alarm, and can issue interrupts after an update of the CMOS registers by the MC146818 is complete. The type of interrupt is signalled in the RTC status registers.

The RTC will update the current time fields by battery power even while the system is off. The current time fields should not be read while an update is in progress, as indicated in the status register.

The clock uses a 32.768kHz crystal, so bits 6-4 of register A should be programmed to a 32kHz divider if the RTC is to count seconds.

This is the RAM map originally used for the RTC/CMOS:

```
Location    Size    Description
------------------------------------------
00h         byte    Current second (BCD)
01h         byte    Seconds alarm (BCD)
02h         byte    Current minute (BCD)
03h         byte    Minutes alarm (BCD)
04h         byte    Current hour (BCD)
05h         byte    Hours alarm (BCD)
06h         byte    Current day of week (BCD)
07h         byte    Current day of month (BCD)
08h         byte    Current month (BCD)
09h         byte    Current year (BCD)
0Ah         byte    Register A
                     bit 7   = Update in progress
                     bit 6-4 = Divider for clock
```

(continues on next page)

(continued from previous page)

```
                                  000 = 4.194 MHz
                                  001 = 1.049 MHz
                                  010 = 32 kHz
                                  10X = test modes
                                  110 = reset / disable
                                  111 = reset / disable
                       bit 3-0 = Rate selection for periodic interrupt
                                  000 = periodic timer disabled
                                  001 = 3.90625 uS
                                  010 = 7.8125 uS
                                  011 = .122070 mS
                                  100 = .244141 mS
                                     ...
                                 1101 = 125 mS
                                 1110 = 250 mS
                                 1111 = 500 mS
0Bh          byte    Register B
                      bit 7   = Run (0) / Halt (1)
                      bit 6   = Periodic interrupt enable
                      bit 5   = Alarm interrupt enable
                      bit 4   = Update-ended interrupt enable
                      bit 3   = Square wave interrupt enable
                      bit 2   = BCD calendar (0) / Binary (1)
                      bit 1   = 12-hour mode (0) / 24-hour mode (1)
                      bit 0   = 0 (DST off) / 1 (DST enabled)
0Ch          byte    Register C (read only)
                      bit 7   = interrupt request flag (IRQF)
                      bit 6   = periodic interrupt flag (PF)
                      bit 5   = alarm interrupt flag (AF)
                      bit 4   = update interrupt flag (UF)
                      bit 3-0 = reserved
0Dh          byte    Register D (read only)
                      bit 7   = RTC has power
                      bit 6-0 = reserved
32h          byte    Current century BCD (*)
(*) location vendor specific and now determined from ACPI global tables
```

2.3. APIC

On Pentium and later processors, an on-board timer is available to each CPU as part of the Advanced Programmable Interrupt Controller. The APIC is accessed through memory-mapped registers and provides interrupt service to each CPU, used for IPIs and local timer interrupts.

Although in theory the APIC is a safe and stable source for local interrupts, in practice, many bugs and glitches have occurred due to the special nature of the APIC CPU-local memory-mapped hardware. Beware that CPU errata may affect the use of the APIC and that workarounds may be required. In addition, some of these workarounds pose unique constraints for virtualization - requiring either extra overhead incurred from extra reads of memory-mapped I/O or additional functionality that may be more computationally expensive to implement.

Since the APIC is documented quite well in the Intel and AMD manuals, we will avoid repetition of the detail here. It should be pointed out that the APIC timer

is programmed through the LVT (local vector timer) register, is capable of one-shot or periodic operation, and is based on the bus clock divided down by the programmable divider register.

2.4. HPET

HPET is quite complex, and was originally intended to replace the PIT / RTC support of the X86 PC. It remains to be seen whether that will be the case, as the de facto standard of PC hardware is to emulate these older devices. Some systems designated as legacy free may support only the HPET as a hardware timer device.

The HPET spec is rather loose and vague, requiring at least 3 hardware timers, but allowing implementation freedom to support many more. It also imposes no fixed rate on the timer frequency, but does impose some extremal values on frequency, error and slew.

In general, the HPET is recommended as a high precision (compared to PIT /RTC) time source which is independent of local variation (as there is only one HPET in any given system). The HPET is also memory-mapped, and its presence is indicated through ACPI tables by the BIOS.

Detailed specification of the HPET is beyond the current scope of this document, as it is also very well documented elsewhere.

2.5. Offboard Timers

Several cards, both proprietary (watchdog boards) and commonplace (e1000) have timing chips built into the cards which may have registers which are accessible to kernel or user drivers. To the author' s knowledge, using these to generate a clocksource for a Linux or other kernel has not yet been attempted and is in general frowned upon as not playing by the agreed rules of the game. Such a timer device would require additional support to be virtualized properly and is not considered important at this time as no known operating system does this.

1.14.3 3. TSC Hardware

The TSC or time stamp counter is relatively simple in theory; it counts instruction cycles issued by the processor, which can be used as a measure of time. In practice, due to a number of problems, it is the most complicated timekeeping device to use.

The TSC is represented internally as a 64-bit MSR which can be read with the RDMSR, RDTSC, or RDTSCP (when available) instructions. In the past, hardware limitations made it possible to write the TSC, but generally on old hardware it was only possible to write the low 32-bits of the 64-bit counter, and the upper 32-bits of the counter were cleared. Now, however, on Intel processors family 0Fh, for models 3, 4 and 6, and family 06h, models e and f, this restriction has been lifted and all 64-bits are writable. On AMD systems, the ability to write the TSC MSR is not an architectural guarantee.

The TSC is accessible from CPL-0 and conditionally, for CPL > 0 software by means of the CR4.TSD bit, which when enabled, disables CPL > 0 TSC access.

Some vendors have implemented an additional instruction, RDTSCP, which returns atomically not just the TSC, but an indicator which corresponds to the processor number. This can be used to index into an array of TSC variables to determine offset information in SMP systems where TSCs are not synchronized. The presence of this instruction must be determined by consulting CPUID feature bits.

Both VMX and SVM provide extension fields in the virtualization hardware which allows the guest visible TSC to be offset by a constant. Newer implementations promise to allow the TSC to additionally be scaled, but this hardware is not yet widely available.

3.1. TSC synchronization

The TSC is a CPU-local clock in most implementations. This means, on SMP platforms, the TSCs of different CPUs may start at different times depending on when the CPUs are powered on. Generally, CPUs on the same die will share the same clock, however, this is not always the case.

The BIOS may attempt to resynchronize the TSCs during the poweron process and the operating system or other system software may attempt to do this as well. Several hardware limitations make the problem worse - if it is not possible to write the full 64-bits of the TSC, it may be impossible to match the TSC in newly arriving CPUs to that of the rest of the system, resulting in unsynchronized TSCs. This may be done by BIOS or system software, but in practice, getting a perfectly synchronized TSC will not be possible unless all values are read from the same clock, which generally only is possible on single socket systems or those with special hardware support.

3.2. TSC and CPU hotplug

As touched on already, CPUs which arrive later than the boot time of the system may not have a TSC value that is synchronized with the rest of the system. Either system software, BIOS, or SMM code may actually try to establish the TSC to a value matching the rest of the system, but a perfect match is usually not a guarantee. This can have the effect of bringing a system from a state where TSC is synchronized back to a state where TSC synchronization flaws, however small, may be exposed to the OS and any virtualization environment.

3.3. TSC and multi-socket / NUMA

Multi-socket systems, especially large multi-socket systems are likely to have individual clocksources rather than a single, universally distributed clock. Since these clocks are driven by different crystals, they will not have perfectly matched frequency, and temperature and electrical variations will cause the CPU clocks, and thus the TSCs to drift over time. Depending on the exact clock and bus design, the drift may or may not be fixed in absolute error, and may accumulate over time.

In addition, very large systems may deliberately slew the clocks of individual cores. This technique, known as spread-spectrum clocking, reduces EMI at the clock frequency and harmonics of it, which may be required to pass FCC standards for telecommunications and computer equipment.

It is recommended not to trust the TSCs to remain synchronized on NUMA or multiple socket systems for these reasons.

3.4. TSC and C-states

C-states, or idling states of the processor, especially C1E and deeper sleep states may be problematic for TSC as well. The TSC may stop advancing in such a state, resulting in a TSC which is behind that of other CPUs when execution is resumed. Such CPUs must be detected and flagged by the operating system based on CPU and chipset identifications.

The TSC in such a case may be corrected by catching it up to a known external clocksource.

3.5. TSC frequency change / P-states

To make things slightly more interesting, some CPUs may change frequency. They may or may not run the TSC at the same rate, and because the frequency change may be staggered or slewed, at some points in time, the TSC rate may not be known other than falling within a range of values. In this case, the TSC will not be a stable time source, and must be calibrated against a known, stable, external clock to be a usable source of time.

Whether the TSC runs at a constant rate or scales with the P-state is model dependent and must be determined by inspecting CPUID, chipset or vendor specific MSR fields.

In addition, some vendors have known bugs where the P-state is actually compensated for properly during normal operation, but when the processor is inactive, the P-state may be raised temporarily to service cache misses from other processors. In such cases, the TSC on halted CPUs could advance faster than that of non-halted processors. AMD Turion processors are known to have this problem.

3.6. TSC and STPCLK / T-states

External signals given to the processor may also have the effect of stopping the TSC. This is typically done for thermal emergency power control to prevent an overheating condition, and typically, there is no way to detect that this condition has happened.

3.7. TSC virtualization - VMX

VMX provides conditional trapping of RDTSC, RDMSR, WRMSR and RDTSCP instructions, which is enough for full virtualization of TSC in any manner. In addition, VMX allows passing through the host TSC plus an additional TSC_OFFSET field specified in the VMCS. Special instructions must be used to read and write the VMCS field.

3.8. TSC virtualization - SVM

SVM provides conditional trapping of RDTSC, RDMSR, WRMSR and RDTSCP instructions, which is enough for full virtualization of TSC in any manner. In addition, SVM allows passing through the host TSC plus an additional offset field specified in the SVM control block.

3.9. TSC feature bits in Linux

In summary, there is no way to guarantee the TSC remains in perfect synchronization unless it is explicitly guaranteed by the architecture. Even if so, the TSCs in multi-sockets or NUMA systems may still run independently despite being locally consistent.

The following feature bits are used by Linux to signal various TSC attributes, but they can only be taken to be meaningful for UP or single node systems.

X86_FEATURE_TSC	The TSC is available in hardware
X86_FEATURE_RDTSCP	The RDTSCP instruction is available
X86_FEATURE_CONSTANT_TSC	The TSC rate is unchanged with P-states
X86_FEATURE_NONSTOP_TSC	The TSC does not stop in C-states
X86_FEATURE_TSC_RELIABLE	TSC sync checks are skipped (VMware)

1.14.4 4. Virtualization Problems

Timekeeping is especially problematic for virtualization because a number of challenges arise. The most obvious problem is that time is now shared between the host and, potentially, a number of virtual machines. Thus the virtual operating system does not run with 100% usage of the CPU, despite the fact that it may very well make that assumption. It may expect it to remain true to very exacting bounds when interrupt sources are disabled, but in reality only its virtual interrupt sources are disabled, and the machine may still be preempted at any time. This causes problems as the passage of real time, the injection of machine interrupts and the associated clock sources are no longer completely synchronized with real time.

This same problem can occur on native hardware to a degree, as SMM mode may steal cycles from the naturally on X86 systems when SMM mode is used by the BIOS, but not in such an extreme fashion. However, the fact that SMM mode may cause similar problems to virtualization makes it a good justification for solving many of these problems on bare metal.

4.1. Interrupt clocking

One of the most immediate problems that occurs with legacy operating systems is that the system timekeeping routines are often designed to keep track of time by counting periodic interrupts. These interrupts may come from the PIT or the RTC, but the problem is the same: the host virtualization engine may not be able to deliver the proper number of interrupts per second, and so guest time may fall behind. This is especially problematic if a high interrupt rate is selected, such as 1000 HZ, which is unfortunately the default for many Linux guests.

There are three approaches to solving this problem; first, it may be possible to simply ignore it. Guests which have a separate time source for tracking 'wall clock' or 'real time' may not need any adjustment of their interrupts to maintain proper time. If this is not sufficient, it may be necessary to inject additional interrupts into the guest in order to increase the effective interrupt rate. This approach leads to complications in extreme conditions, where host load or guest lag is too much to compensate for, and thus another solution to the problem has risen: the guest may need to become aware of lost ticks and compensate for them internally. Although promising in theory, the implementation of this policy in Linux has been extremely error prone, and a number of buggy variants of lost tick compensation are distributed across commonly used Linux systems.

Windows uses periodic RTC clocking as a means of keeping time internally, and thus requires interrupt slewing to keep proper time. It does use a low enough rate (ed: is it 18.2 Hz?) however that it has not yet been a problem in practice.

4.2. TSC sampling and serialization

As the highest precision time source available, the cycle counter of the CPU has aroused much interest from developers. As explained above, this timer has many problems unique to its nature as a local, potentially unstable and potentially unsynchronized source. One issue which is not unique to the TSC, but is highlighted because of its very precise nature is sampling delay. By definition, the counter, once read is already old. However, it is also possible for the counter to be read ahead of the actual use of the result. This is a consequence of the superscalar execution of the instruction stream, which may execute instructions out of order. Such execution is called non-serialized. Forcing serialized execution is necessary for precise measurement with the TSC, and requires a serializing instruction, such as CPUID or an MSR read.

Since CPUID may actually be virtualized by a trap and emulate mechanism, this serialization can pose a performance issue for hardware virtualization. An accurate time stamp counter reading may therefore not always be available, and it may be necessary for an implementation to guard against "backwards" reads of the TSC as seen from other CPUs, even in an otherwise perfectly synchronized system.

4.3. Timespec aliasing

Additionally, this lack of serialization from the TSC poses another challenge when using results of the TSC when measured against another time source. As the TSC is much higher precision, many possible values of the TSC may be read while another clock is still expressing the same value.

That is, you may read (T,T+10) while external clock C maintains the same value. Due to non-serialized reads, you may actually end up with a range which fluctuates - from (T-1.. T+10). Thus, any time calculated from a TSC, but calibrated against an external value may have a range of valid values. Re-calibrating this computation may actually cause time, as computed after the calibration, to go backwards, compared with time computed before the calibration.

This problem is particularly pronounced with an internal time source in Linux, the kernel time, which is expressed in the theoretically high resolution timespec - but which advances in much larger granularity intervals, sometimes at the rate of jiffies, and possibly in catchup modes, at a much larger step.

This aliasing requires care in the computation and recalibration of kvmclock and any other values derived from TSC computation (such as TSC virtualization itself).

4.4. Migration

Migration of a virtual machine raises problems for timekeeping in two ways. First, the migration itself may take time, during which interrupts cannot be delivered, and after which, the guest time may need to be caught up. NTP may be able to help to some degree here, as the clock correction required is typically small enough to fall in the NTP-correctable window.

An additional concern is that timers based off the TSC (or HPET, if the raw bus clock is exposed) may now be running at different rates, requiring compensation in some way in the hypervisor by virtualizing these timers. In addition, migrating to a faster machine may preclude the use of a passthrough TSC, as a faster clock cannot be made visible to a guest without the potential of time advancing faster than usual. A slower clock is less of a problem, as it can always be caught up to the original rate. KVM clock avoids these problems by simply storing multipliers and offsets against the TSC for the guest to convert back into nanosecond resolution values.

4.5. Scheduling

Since scheduling may be based on precise timing and firing of interrupts, the scheduling algorithms of an operating system may be adversely affected by virtualization. In theory, the effect is random and should be universally distributed, but in contrived as well as real scenarios (guest device access, causes of virtualization exits, possible context switch), this may not always be the case. The effect of this has not been well studied.

In an attempt to work around this, several implementations have provided a paravirtualized scheduler clock, which reveals the true amount of CPU time for which a virtual machine has been running.

4.6. Watchdogs

Watchdog timers, such as the lock detector in Linux may fire accidentally when running under hardware virtualization due to timer interrupts being delayed or misinterpretation of the passage of real time. Usually, these warnings are spurious and can be ignored, but in some circumstances it may be necessary to disable such detection.

4.7. Delays and precision timing

Precise timing and delays may not be possible in a virtualized system. This can happen if the system is controlling physical hardware, or issues delays to compensate for slower I/O to and from devices. The first issue is not solvable in general for a virtualized system; hardware control software can't be adequately virtualized without a full real-time operating system, which would require an RT aware virtualization platform.

The second issue may cause performance problems, but this is unlikely to be a significant issue. In many cases these delays may be eliminated through configuration or paravirtualization.

4.8. Covert channels and leaks

In addition to the above problems, time information will inevitably leak to the guest about the host in anything but a perfect implementation of virtualized time. This may allow the guest to infer the presence of a hypervisor (as in a red-pill type detection), and it may allow information to leak between guests by using CPU utilization itself as a signalling channel. Preventing such problems would require completely isolated virtual time which may not track real time any longer. This may be useful in certain security or QA contexts, but in general isn't recommended for real-world deployment scenarios.

1.15 KVM VCPU Requests

1.15.1 Overview

KVM supports an internal API enabling threads to request a VCPU thread to perform some activity. For example, a thread may request a VCPU to flush its TLB with a VCPU request. The API consists of the following functions:

```
/* Check if any requests are pending for VCPU @vcpu. */
bool kvm_request_pending(struct kvm_vcpu *vcpu);

/* Check if VCPU @vcpu has request @req pending. */
bool kvm_test_request(int req, struct kvm_vcpu *vcpu);

/* Clear request @req for VCPU @vcpu. */
void kvm_clear_request(int req, struct kvm_vcpu *vcpu);
```

(continues on next page)

(continued from previous page)

```
/*
 * Check if VCPU @vcpu has request @req pending. When the request is
 * pending it will be cleared and a memory barrier, which pairs with
 * another in kvm_make_request(), will be issued.
 */
bool kvm_check_request(int req, struct kvm_vcpu *vcpu);

/*
 * Make request @req of VCPU @vcpu. Issues a memory barrier, which pairs
 * with another in kvm_check_request(), prior to setting the request.
 */
void kvm_make_request(int req, struct kvm_vcpu *vcpu);

/* Make request @req of all VCPUs of the VM with struct kvm @kvm. */
bool kvm_make_all_cpus_request(struct kvm *kvm, unsigned int req);
```

Typically a requester wants the VCPU to perform the activity as soon as possible after making the request. This means most requests (kvm_make_request() calls) are followed by a call to kvm_vcpu_kick(), and kvm_make_all_cpus_request() has the kicking of all VCPUs built into it.

VCPU Kicks

The goal of a VCPU kick is to bring a VCPU thread out of guest mode in order to perform some KVM maintenance. To do so, an IPI is sent, forcing a guest mode exit. However, a VCPU thread may not be in guest mode at the time of the kick. Therefore, depending on the mode and state of the VCPU thread, there are two other actions a kick may take. All three actions are listed below:

1) Send an IPI. This forces a guest mode exit.

2) Waking a sleeping VCPU. Sleeping VCPUs are VCPU threads outside guest mode that wait on waitqueues. Waking them removes the threads from the waitqueues, allowing the threads to run again. This behavior may be suppressed, see KVM_REQUEST_NO_WAKEUP below.

3) Nothing. When the VCPU is not in guest mode and the VCPU thread is not sleeping, then there is nothing to do.

VCPU Mode

VCPUs have a mode state, `vcpu->mode`, that is used to track whether the guest is running in guest mode or not, as well as some specific outside guest mode states. The architecture may use `vcpu->mode` to ensure VCPU requests are seen by VCPUs (see “Ensuring Requests Are Seen”), as well as to avoid sending unnecessary IPIs (see “IPI Reduction”), and even to ensure IPI acknowledgements are waited upon (see “Waiting for Acknowledgements”). The following modes are defined:

OUTSIDE_GUEST_MODE

The VCPU thread is outside guest mode.

IN_GUEST_MODE

The VCPU thread is in guest mode.

EXITING_GUEST_MODE

The VCPU thread is transitioning from IN_GUEST_MODE to OUTSIDE_GUEST_MODE.

READING_SHADOW_PAGE_TABLES

The VCPU thread is outside guest mode, but it wants the sender of certain VCPU requests, namely KVM_REQ_TLB_FLUSH, to wait until the VCPU thread is done reading the page tables.

1.15.2 VCPU Request Internals

VCPU requests are simply bit indices of the `vcpu->requests` bitmap. This means general bitops, like those documented in [**?**] could also be used, e.g.

```
clear_bit(KVM_REQ_UNHALT & KVM_REQUEST_MASK, &vcpu->requests);
```

However, VCPU request users should refrain from doing so, as it would break the abstraction. The first 8 bits are reserved for architecture independent requests, all additional bits are available for architecture dependent requests.

Architecture Independent Requests

KVM_REQ_TLB_FLUSH

KVM's common MMU notifier may need to flush all of a guest's TLB entries, calling kvm_flush_remote_tlbs() to do so. Architectures that choose to use the common kvm_flush_remote_tlbs() implementation will need to handle this VCPU request.

KVM_REQ_MMU_RELOAD

When shadow page tables are used and memory slots are removed it's necessary to inform each VCPU to completely refresh the tables. This request is used for that.

KVM_REQ_PENDING_TIMER

This request may be made from a timer handler run on the host on behalf of a VCPU. It informs the VCPU thread to inject a timer interrupt.

KVM_REQ_UNHALT

This request may be made from the KVM common function kvm_vcpu_block(), which is used to emulate an instruction that causes a CPU to halt until one of an architectural specific set of events and/or interrupts is received (determined by checking kvm_arch_vcpu_runnable()). When that event or interrupt arrives kvm_vcpu_block() makes the request. This is in contrast to when kvm_vcpu_block() returns due to any other reason, such as a pending signal, which does not indicate the VCPU's halt emulation should stop, and therefore does not make the request.

KVM_REQUEST_MASK

VCPU requests should be masked by KVM_REQUEST_MASK before using them with bitops. This is because only the lower 8 bits are used to represent the request's number. The upper bits are used as flags. Currently only two flags are defined.

VCPU Request Flags

KVM_REQUEST_NO_WAKEUP

> This flag is applied to requests that only need immediate attention from VCPUs running in guest mode. That is, sleeping VCPUs do not need to be awaken for these requests. Sleeping VCPUs will handle the requests when they are awaken later for some other reason.

KVM_REQUEST_WAIT

> When requests with this flag are made with kvm_make_all_cpus_request(), then the caller will wait for each VCPU to acknowledge its IPI before proceeding. This flag only applies to VCPUs that would receive IPIs. If, for example, the VCPU is sleeping, so no IPI is necessary, then the requesting thread does not wait. This means that this flag may be safely combined with KVM_REQUEST_NO_WAKEUP. See "Waiting for Acknowledgements" for more information about requests with KVM_REQUEST_WAIT.

1.15.3 VCPU Requests with Associated State

Requesters that want the receiving VCPU to handle new state need to ensure the newly written state is observable to the receiving VCPU thread's CPU by the time it observes the request. This means a write memory barrier must be inserted after writing the new state and before setting the VCPU request bit. Additionally, on the receiving VCPU thread's side, a corresponding read barrier must be inserted after reading the request bit and before proceeding to read the new state associated with it. See scenario 3, Message and Flag, of [**?**] and the kernel documentation [**?**].

The pair of functions, kvm_check_request() and kvm_make_request(), provide the memory barriers, allowing this requirement to be handled internally by the API.

1.15.4 Ensuring Requests Are Seen

When making requests to VCPUs, we want to avoid the receiving VCPU executing in guest mode for an arbitrary long time without handling the request. We can be sure this won't happen as long as we ensure the VCPU thread checks kvm_request_pending() before entering guest mode and that a kick will send an IPI to force an exit from guest mode when necessary. Extra care must be taken to cover the period after the VCPU thread's last kvm_request_pending() check and before it has entered guest mode, as kick IPIs will only trigger guest mode exits for VCPU threads that are in guest mode or at least have already disabled interrupts in order to prepare to enter guest mode. This means that an optimized

implementation (see “IPI Reduction”) must be certain when it’s safe to not send the IPI. One solution, which all architectures except s390 apply, is to:

- set `vcpu->mode` to IN_GUEST_MODE between disabling the interrupts and the last kvm_request_pending() check;
- enable interrupts atomically when entering the guest.

This solution also requires memory barriers to be placed carefully in both the requesting thread and the receiving VCPU. With the memory barriers we can exclude the possibility of a VCPU thread observing !kvm_request_pending() on its last check and then not receiving an IPI for the next request made of it, even if the request is made immediately after the check. This is done by way of the Dekker memory barrier pattern (scenario 10 of [**?**]). As the Dekker pattern requires two variables, this solution pairs `vcpu->mode` with `vcpu->requests`. Substituting them into the pattern gives:

```
CPU1                                    CPU2
================                        ================
local_irq_disable();
WRITE_ONCE(vcpu->mode, IN_GUEST_MODE);  kvm_make_request(REQ, vcpu);
smp_mb();                               smp_mb();
if (kvm_request_pending(vcpu)) {        if (READ_ONCE(vcpu->mode) ==
                                            IN_GUEST_MODE) {
    ...abort guest entry...                 ...send IPI...
}                                       }
```

As stated above, the IPI is only useful for VCPU threads in guest mode or that have already disabled interrupts. This is why this specific case of the Dekker pattern has been extended to disable interrupts before setting `vcpu->mode` to IN_GUEST_MODE. WRITE_ONCE() and READ_ONCE() are used to pedantically implement the memory barrier pattern, guaranteeing the compiler doesn’t interfere with `vcpu->mode`’s carefully planned accesses.

IPI Reduction

As only one IPI is needed to get a VCPU to check for any/all requests, then they may be coalesced. This is easily done by having the first IPI sending kick also change the VCPU mode to something !IN_GUEST_MODE. The transitional state, EXITING_GUEST_MODE, is used for this purpose.

Waiting for Acknowledgements

Some requests, those with the KVM_REQUEST_WAIT flag set, require IPIs to be sent, and the acknowledgements to be waited upon, even when the target VCPU threads are in modes other than IN_GUEST_MODE. For example, one case is when a target VCPU thread is in READING_SHADOW_PAGE_TABLES mode, which is set after disabling interrupts. To support these cases, the KVM_REQUEST_WAIT flag changes the condition for sending an IPI from checking that the VCPU is IN_GUEST_MODE to checking that it is not OUTSIDE_GUEST_MODE.

Request-less VCPU Kicks

As the determination of whether or not to send an IPI depends on the two-variable Dekker memory barrier pattern, then it's clear that request-less VCPU kicks are almost never correct. Without the assurance that a non-IPI generating kick will still result in an action by the receiving VCPU, as the final kvm_request_pending() check does for request-accompanying kicks, then the kick may not do anything useful at all. If, for instance, a request-less kick was made to a VCPU that was just about to set its mode to IN_GUEST_MODE, meaning no IPI is sent, then the VCPU thread may continue its entry without actually having done whatever it was the kick was meant to initiate.

One exception is x86's posted interrupt mechanism. In this case, however, even the request-less VCPU kick is coupled with the same local_irq_disable() + smp_mb() pattern described above; the ON bit (Outstanding Notification) in the posted interrupt descriptor takes the role of `vcpu->requests`. When sending a posted interrupt, PIR.ON is set before reading `vcpu->mode`; dually, in the VCPU thread, vmx_sync_pir_to_irr() reads PIR after setting `vcpu->mode` to IN_GUEST_MODE.

1.15.5 Additional Considerations

Sleeping VCPUs

VCPU threads may need to consider requests before and/or after calling functions that may put them to sleep, e.g. kvm_vcpu_block(). Whether they do or not, and, if they do, which requests need consideration, is architecture dependent. kvm_vcpu_block() calls kvm_arch_vcpu_runnable() to check if it should awaken. One reason to do so is to provide architectures a function where requests may be checked if necessary.

Clearing Requests

Generally it only makes sense for the receiving VCPU thread to clear a request. However, in some circumstances, such as when the requesting thread and the receiving VCPU thread are executed serially, such as when they are the same thread, or when they are using some form of concurrency control to temporarily execute synchronously, then it's possible to know that the request may be cleared immediately, rather than waiting for the receiving VCPU thread to handle the request in VCPU RUN. The only current examples of this are kvm_vcpu_block() calls made by VCPUs to block themselves. A possible side-effect of that call is to make the KVM_REQ_UNHALT request, which may then be cleared immediately when the VCPU returns from the call.

1.15.6 References

1.16 Review checklist for kvm patches

1. The patch must follow Documentation/process/coding-style.rst and Documentation/process/submitting-patches.rst.
2. Patches should be against kvm.git master branch.
3. If the patch introduces or modifies a new userspace API: - the API must be documented in Documentation/virt/kvm/api.rst - the API must be discoverable using KVM_CHECK_EXTENSION
4. New state must include support for save/restore.
5. New features must default to off (userspace should explicitly request them). Performance improvements can and should default to on.
6. New cpu features should be exposed via KVM_GET_SUPPORTED_CPUID2
7. Emulator changes should be accompanied by unit tests for qemu-kvm.git kvm/test directory.
8. Changes should be vendor neutral when possible. Changes to common code are better than duplicating changes to vendor code.
9. Similarly, prefer changes to arch independent code than to arch dependent code.
10. User/kernel interfaces and guest/host interfaces must be 64-bit clean (all variables and sizes naturally aligned on 64-bit; use specific types only - u64 rather than ulong).
11. New guest visible features must either be documented in a hardware manual or be accompanied by documentation.
12. Features must be robust against reset and kexec - for example, shared host/guest memory must be unshared to prevent the host from writing to guest memory that the guest has not reserved for this purpose.

1.17 ARM

1.17.1 Internal ABI between the kernel and HYP

This file documents the interaction between the Linux kernel and the hypervisor layer when running Linux as a hypervisor (for example KVM). It doesn't cover the interaction of the kernel with the hypervisor when running as a guest (under Xen, KVM or any other hypervisor), or any hypervisor-specific interaction when the kernel is used as a host.

Note: KVM/arm has been removed from the kernel. The API described here is still valid though, as it allows the kernel to kexec when booted at HYP. It can also be used by a hypervisor other than KVM if necessary.

On arm and arm64 (without VHE), the kernel doesn't run in hypervisor mode, but still needs to interact with it, allowing a built-in hypervisor to be either installed or torn down.

In order to achieve this, the kernel must be booted at HYP (arm) or EL2 (arm64), allowing it to install a set of stubs before dropping to SVC/EL1. These stubs are accessible by using a 'hvc #0' instruction, and only act on individual CPUs.

Unless specified otherwise, any built-in hypervisor must implement these functions (see arch/arm{,64}/include/asm/virt.h):

- ```
  r0/x0 = HVC_SET_VECTORS
  r1/x1 = vectors
  ```

  Set HVBAR/VBAR_EL2 to 'vectors' to enable a hypervisor. 'vectors' must be a physical address, and respect the alignment requirements of the architecture. Only implemented by the initial stubs, not by Linux hypervisors.

- ```
  r0/x0 = HVC_RESET_VECTORS
  ```

  Turn HYP/EL2 MMU off, and reset HVBAR/VBAR_EL2 to the initials stubs' exception vector value. This effectively disables an existing hypervisor.

- ```
  r0/x0 = HVC_SOFT_RESTART
  r1/x1 = restart address
  x2 = x0's value when entering the next payload (arm64)
  x3 = x1's value when entering the next payload (arm64)
  x4 = x2's value when entering the next payload (arm64)
  ```

  Mask all exceptions, disable the MMU, move the arguments into place (arm64 only), and jump to the restart address while at HYP/EL2. This hypercall is not expected to return to its caller.

Any other value of r0/x0 triggers a hypervisor-specific handling, which is not documented here.

The return value of a stub hypercall is held by r0/x0, and is 0 on success, and HVC_STUB_ERR on error. A stub hypercall is allowed to clobber any of the caller-saved registers (x0-x18 on arm64, r0-r3 and ip on arm). It is thus recommended to use a function call to perform the hypercall.

1.17.2 Power State Coordination Interface (PSCI)

KVM implements the PSCI (Power State Coordination Interface) specification in order to provide services such as CPU on/off, reset and power-off to the guest.

The PSCI specification is regularly updated to provide new features, and KVM implements these updates if they make sense from a virtualization point of view.

This means that a guest booted on two different versions of KVM can observe two different "firmware" revisions. This could cause issues if a given guest is tied to a particular PSCI revision (unlikely), or if a migration causes a different PSCI version to be exposed out of the blue to an unsuspecting guest.

In order to remedy this situation, KVM exposes a set of "firmware pseudo-registers" that can be manipulated using the GET/SET_ONE_REG interface. These registers
  ```

can be saved/restored by userspace, and set to a convenient value if required.

The following register is defined:

- KVM_REG_ARM_PSCI_VERSION:
  - Only valid if the vcpu has the KVM_ARM_VCPU_PSCI_0_2 feature set (and thus has already been initialized)
  - Returns the current PSCI version on GET_ONE_REG (defaulting to the highest PSCI version implemented by KVM and compatible with v0.2)
  - Allows any PSCI version implemented by KVM and compatible with v0.2 to be set with SET_ONE_REG
  - Affects the whole VM (even if the register view is per-vcpu)
- **KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1:** Holds the state of the firmware support to mitigate CVE-2017-5715, as offered by KVM to the guest via a HVC call. The workaround is described under SMCCC_ARCH_WORKAROUND_1 in [1].

  Accepted values are:

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1_NOT_AVAIL: KVM does not offer firmware support for the workaround. The mitigation status for the guest is unknown.

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1_AVAIL: The workaround HVC call is available to the guest and required for the mitigation.

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_1_NOT_REQUIRED: The workaround HVC call is available to the guest, but it is not needed on this VCPU.
- **KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2:** Holds the state of the firmware support to mitigate CVE-2018-3639, as offered by KVM to the guest via a HVC call. The workaround is described under SMCCC_ARCH_WORKAROUND_2 in[1].

  Accepted values are:

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_NOT_AVAIL: A workaround is not available. KVM does not offer firmware support for the workaround.

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_UNKNOWN: The workaround state is unknown. KVM does not offer firmware support for the workaround.

  KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_AVAIL: The workaround is available, and can be disabled by a vCPU. If KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_ENABLED is set, it is active for this vCPU.

[1] https://developer.arm.com/-/media/developer/pdf/ARM_DEN_0070A_Firmware_interfaces_for_mitigating_CVE-2017-5715.pdf

KVM_REG_ARM_SMCCC_ARCH_WORKAROUND_2_NOT_REQUIRED:
The workaround is always active on this vCPU or it is not needed.

1.17.3 Paravirtualized time support for arm64

Arm specification DEN0057/A defines a standard for paravirtualised time support for AArch64 guests:

https://developer.arm.com/docs/den0057/a

KVM/arm64 implements the stolen time part of this specification by providing some hypervisor service calls to support a paravirtualized guest obtaining a view of the amount of time stolen from its execution.

Two new SMCCC compatible hypercalls are defined:

- PV_TIME_FEATURES: 0xC5000020
- PV_TIME_ST: 0xC5000021

These are only available in the SMC64/HVC64 calling convention as paravirtualized time is not available to 32 bit Arm guests. The existence of the PV_FEATURES hypercall should be probed using the SMCCC 1.1 ARCH_FEATURES mechanism before calling it.

PV_TIME_FEATURES

Function ID:	(uint32)	0xC5000020
PV_call_id:	(uint32)	The function to query for support. Currently only PV_TIME_ST is supported.
Return value:	(int64)	NOT_SUPPORTED (-1) or SUCCESS (0) if the relevant PV-time feature is supported by the hypervisor.

PV_TIME_ST

Function ID:	(uint32)	0xC5000021
Return value:	(int64)	IPA of the stolen time data structure for this VCPU. On failure: NOT_SUPPORTED (-1)

The IPA returned by PV_TIME_ST should be mapped by the guest as normal memory with inner and outer write back caching attributes, in the inner shareable domain. A total of 16 bytes from the IPA returned are guaranteed to be meaningfully filled by the hypervisor (see structure below).

PV_TIME_ST returns the structure for the calling VCPU.

Stolen Time

The structure pointed to by the PV_TIME_ST hypercall is as follows:

Field	Byte Length	Byte Offset	Description
Revision	4	0	Must be 0 for version 1.0
Attributes	4	4	Must be 0
Stolen time	8	8	Stolen time in unsigned nanoseconds indicating how much time this VCPU thread was involuntarily not running on a physical CPU.

All values in the structure are stored little-endian.

The structure will be updated by the hypervisor prior to scheduling a VCPU. It will be present within a reserved region of the normal memory given to the guest. The guest should not attempt to write into this memory. There is a structure per VCPU of the guest.

It is advisable that one or more 64k pages are set aside for the purpose of these structures and not used for other purposes, this enables the guest to map the region using 64k pages and avoids conflicting attributes with other memory.

For the user space interface see Documentation/virt/kvm/devices/vcpu.rst section "3. GROUP: KVM_ARM_VCPU_PVTIME_CTRL" .

1.18 Devices

1.18.1 ARM Virtual Interrupt Translation Service (ITS)

Device types supported: KVM_DEV_TYPE_ARM_VGIC_ITS ARM Interrupt Translation Service Controller

The ITS allows MSI(-X) interrupts to be injected into guests. This extension is optional. Creating a virtual ITS controller also requires a host GICv3 (see arm-vgic-v3.txt), but does not depend on having physical ITS controllers.

There can be multiple ITS controllers per guest, each of them has to have a separate, non-overlapping MMIO region.

Groups

KVM_DEV_ARM_VGIC_GRP_ADDR

Attributes:

KVM_VGIC_ITS_ADDR_TYPE (rw, 64-bit) Base address in the guest physical address space of the GICv3 ITS control register frame. This address needs to be 64K aligned and the region covers 128K.

Errors:

-E2BIG	Address outside of addressable IPA range
-EINVAL	Incorrectly aligned address
-EEXIST	Address already configured
-EFAULT	Invalid user pointer for attr->addr.
-ENODEV	Incorrect attribute or the ITS is not supported.

KVM_DEV_ARM_VGIC_GRP_CTRL

Attributes:

KVM_DEV_ARM_VGIC_CTRL_INIT request the initialization of the ITS, no additional parameter in kvm_device_attr.addr.

KVM_DEV_ARM_ITS_CTRL_RESET reset the ITS, no additional parameter in kvm_device_attr.addr. See “ITS Reset State” section.

KVM_DEV_ARM_ITS_SAVE_TABLES save the ITS table data into guest RAM, at the location provisioned by the guest in corresponding registers/table entries.

The layout of the tables in guest memory defines an ABI. The entries are laid out in little endian format as described in the last paragraph.

KVM_DEV_ARM_ITS_RESTORE_TABLES restore the ITS tables from guest RAM to ITS internal structures.

The GICV3 must be restored before the ITS and all ITS registers but the GITS_CTLR must be restored before restoring the ITS tables.

The GITS_IIDR read-only register must also be restored before calling KVM_DEV_ARM_ITS_RESTORE_TABLES as the IIDR revision field encodes the ABI revision.

The expected ordering when restoring the GICv3/ITS is described in section “ITS Restore Sequence”.

Errors:

-ENXIO	ITS not properly configured as required prior to setting this attribute
-ENOMEM	Memory shortage when allocating ITS internal data
-EINVAL	Inconsistent restored data
-EFAULT	Invalid guest ram access
-EBUSY	One or more VCPUS are running
-EACCES	The virtual ITS is backed by a physical GICv4 ITS, and the state is not available

KVM_DEV_ARM_VGIC_GRP_ITS_REGS

Attributes: The attr field of kvm_device_attr encodes the offset of the ITS register, relative to the ITS control frame base address (ITS_base).

kvm_device_attr.addr points to a __u64 value whatever the width of the addressed register (32/64 bits). 64 bit registers can only be accessed with full length.

Writes to read-only registers are ignored by the kernel except for:

- GITS_CREADR. It must be restored otherwise commands in the queue will be re-executed after restoring CWRITER. GITS_CREADR must be restored before restoring the GITS_CTLR which is likely to enable the ITS. Also it must be restored after GITS_CBASER since a write to GITS_CBASER resets GITS_CREADR.
- GITS_IIDR. The Revision field encodes the table layout ABI revision. In the future we might implement direct injection of virtual LPIs. This will require an upgrade of the table layout and an evolution of the ABI. GITS_IIDR must be restored before calling KVM_DEV_ARM_ITS_RESTORE_TABLES.

For other registers, getting or setting a register has the same effect as reading/writing the register on real hardware.

Errors:

-ENXIO	Offset does not correspond to any supported register
-EFAULT	Invalid user pointer for attr->addr
-EINVAL	Offset is not 64-bit aligned
-EBUSY	one or more VCPUS are running

ITS Restore Sequence:

The following ordering must be followed when restoring the GIC and the ITS:

a) restore all guest memory and create vcpus

b) restore all redistributors

c) provide the ITS base address (KVM_DEV_ARM_VGIC_GRP_ADDR)

d) restore the ITS in the following order:

1. Restore GITS_CBASER
2. Restore all other `GITS_` registers, except GITS_CTLR!
3. Load the ITS table data (KVM_DEV_ARM_ITS_RESTORE_TABLES)
4. Restore GITS_CTLR

Then vcpus can be started.

ITS Table ABI REV0:

Revision 0 of the ABI only supports the features of a virtual GICv3, and does not support a virtual GICv4 with support for direct injection of virtual interrupts for nested hypervisors.

The device table and ITT are indexed by the DeviceID and EventID, respectively. The collection table is not indexed by CollectionID, and the entries in the collection are listed in no particular order. All entries are 8 bytes.

Device Table Entry (DTE):

```
bits:     | 63| 62 ... 49 | 48 ... 5 | 4 ... 0 |
values:   | V |   next    | ITT_addr |  Size   |
```

where:

- V indicates whether the entry is valid. If not, other fields are not meaningful.
- next: equals to 0 if this entry is the last one; otherwise it corresponds to the DeviceID offset to the next DTE, capped by 2^14 -1.
- ITT_addr matches bits [51:8] of the ITT address (256 Byte aligned).
- Size specifies the supported number of bits for the EventID, minus one

Collection Table Entry (CTE):

```
bits:     | 63| 62 ..  52  | 51 ... 16 | 15  ...   0 |
values:   | V |    RES0    |  RDBase   |    ICID     |
```

where:

- V indicates whether the entry is valid. If not, other fields are not meaningful.

- RES0: reserved field with Should-Be-Zero-or-Preserved behavior.
- RDBase is the PE number (GICR_TYPER.Processor_Number semantic),
- ICID is the collection ID

Interrupt Translation Entry (ITE):

```
bits:     | 63 ... 48 | 47 ... 16 | 15 ... 0 |
values:   |    next   |   pINTID  |  ICID    |
```

where:

- next: equals to 0 if this entry is the last one; otherwise it corresponds to the EventID offset to the next ITE capped by 2^16 -1.
- pINTID is the physical LPI ID; if zero, it means the entry is not valid and other fields are not meaningful.
- ICID is the collection ID

ITS Reset State:

RESET returns the ITS to the same state that it was when first created and initialized. When the RESET command returns, the following things are guaranteed:

- The ITS is not enabled and quiescent GITS_CTLR.Enabled = 0 .Quiescent=1
- There is no internally cached state
- No collection or device table are used GITS_BASER<n>.Valid = 0
- GITS_CBASER = 0, GITS_CREADR = 0, GITS_CWRITER = 0
- The ABI version is unchanged and remains the one set when the ITS device was first created.

1.18.2 ARM Virtual Generic Interrupt Controller v2 (VGIC)

Device types supported:

- KVM_DEV_TYPE_ARM_VGIC_V2 ARM Generic Interrupt Controller v2.0

Only one VGIC instance may be instantiated through either this API or the legacy KVM_CREATE_IRQCHIP API. The created VGIC will act as the VM interrupt controller, requiring emulated user-space devices to inject interrupts to the VGIC instead of directly to CPUs.

GICv3 implementations with hardware compatibility support allow creating a guest GICv2 through this interface. For information on creating a guest GICv3 device and guest ITS devices, see arm-vgic-v3.txt. It is not possible to create both a GICv3 and GICv2 device on the same VM.

Groups:

KVM_DEV_ARM_VGIC_GRP_ADDR Attributes:

KVM_VGIC_V2_ADDR_TYPE_DIST (rw, 64-bit) Base address in the guest physical address space of the GIC distributor register mappings. Only valid for KVM_DEV_TYPE_ARM_VGIC_V2. This address needs to be 4K aligned and the region covers 4 KByte.

KVM_VGIC_V2_ADDR_TYPE_CPU (rw, 64-bit) Base address in the guest physical address space of the GIC virtual cpu interface register mappings. Only valid for KVM_DEV_TYPE_ARM_VGIC_V2. This address needs to be 4K aligned and the region covers 4 KByte.

Errors:

-E2BIG	Address outside of addressable IPA range
-EINVAL	Incorrectly aligned address
-EEXIST	Address already configured
-ENXIO	The group or attribute is unknown/unsupported for this device or hardware support is missing.
-EFAULT	Invalid user pointer for attr->addr.

KVM_DEV_ARM_VGIC_GRP_DIST_REGS Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63   ....  40 | 39 ..  32  |  31   ....    0 |
values:   |    reserved   | vcpu_index |      offset     |
```

All distributor regs are (rw, 32-bit)

The offset is relative to the "Distributor base address" as defined in the GICv2 specs. Getting or setting such a register has the same effect as reading or writing the register on the actual hardware from the cpu whose index is specified with the vcpu_index field. Note that most distributor fields are not banked, but return the same value regardless of the vcpu_index used to access the register.

GICD_IIDR.Revision is updated when the KVM implementation of an emulated GICv2 is changed in a way directly observable by the guest or userspace. Userspace should read GICD_IIDR from KVM and write back the read value to confirm its expected behavior is aligned with the KVM implementation. Userspace should set GICD_IIDR before setting any other registers (both KVM_DEV_ARM_VGIC_GRP_DIST_REGS and KVM_DEV_ARM_VGIC_GRP_CPU_REGS) to ensure the expected behavior. Unless GICD_IIDR has been set from userspace, writes to the interrupt group registers (GICD_IGROUPR) are ignored.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	One or more VCPUs are running
-EINVAL	Invalid vcpu_index supplied

KVM_DEV_ARM_VGIC_GRP_CPU_REGS Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63   ....  40 | 39 ..  32  |  31   ....    0 |
values:   |    reserved   | vcpu_index |      offset     |
```

All CPU interface regs are (rw, 32-bit)

The offset specifies the offset from the "CPU interface base address" as defined in the GICv2 specs. Getting or setting such a register has the same effect as reading or writing the register on the actual hardware.

The Active Priorities Registers APRn are implementation defined, so we set a fixed format for our implementation that fits with the model of a "GICv2 implementation without the security extensions" which we present to the guest. This interface always exposes four register APR[0-3] describing the maximum possible 128 preemption levels. The semantics of the register indicate if any interrupts in a given preemption level are in the active state by setting the corresponding bit.

Thus, preemption level X has one or more active interrupts if and only if:

APRn[X mod 32] == 0b1, where n = X / 32

Bits for undefined preemption levels are RAZ/WI.

Note that this differs from a CPU's view of the APRs on hardware in which a GIC without the security extensions expose group 0 and group 1 active priorities in separate register groups, whereas we show a combined view similar to GICv2's GICH_APR.

For historical reasons and to provide ABI compatibility with userspace we export the GICC_PMR register in the format of the GICH_VMCR.VMPriMask field in the lower 5 bits of a word, meaning that userspace must always use the lower 5 bits to communicate with the KVM device and must shift the value left by 3 places to obtain the actual priority mask level.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	One or more VCPUs are running
-EINVAL	Invalid vcpu_index supplied

KVM_DEV_ARM_VGIC_GRP_NR_IRQS Attributes:

A value describing the number of interrupts (SGI, PPI and SPI) for this GIC instance, ranging from 64 to 1024, in increments of 32.

Errors:

-EINVAL	Value set is out of the expected range
-EBUSY	Value has already be set, or GIC has already been initialized with default values.

KVM_DEV_ARM_VGIC_GRP_CTRL Attributes:

KVM_DEV_ARM_VGIC_CTRL_INIT request the initialization of the VGIC or ITS, no additional parameter in kvm_device_attr.addr.

Errors:

-ENXIO	VGIC not properly configured as required prior to calling this attribute
-ENODEV	no online VCPU
-ENOMEM	memory shortage when allocating vgic internal data

1.18.3 ARM Virtual Generic Interrupt Controller v3 and later (VGICv3)

Device types supported:

- KVM_DEV_TYPE_ARM_VGIC_V3 ARM Generic Interrupt Controller v3.0

Only one VGIC instance may be instantiated through this API. The created VGIC will act as the VM interrupt controller, requiring emulated user-space devices to inject interrupts to the VGIC instead of directly to CPUs. It is not possible to create both a GICv3 and GICv2 on the same VM.

Creating a guest GICv3 device requires a host GICv3 as well.

Groups:

KVM_DEV_ARM_VGIC_GRP_ADDR Attributes:

KVM_VGIC_V3_ADDR_TYPE_DIST (rw, 64-bit) Base address in the guest physical address space of the GICv3 distributor register mappings. Only valid for KVM_DEV_TYPE_ARM_VGIC_V3. This address needs to be 64K aligned and the region covers 64 KByte.

KVM_VGIC_V3_ADDR_TYPE_REDIST (rw, 64-bit) Base address in the guest physical address space of the GICv3 redistributor register mappings. There are two 64K pages for

each VCPU and all of the redistributor pages are contiguous. Only valid for KVM_DEV_TYPE_ARM_VGIC_V3. This address needs to be 64K aligned.

KVM_VGIC_V3_ADDR_TYPE_REDIST_REGION (rw, 64-bit)
The attribute data pointed to by kvm_device_attr.addr is a __u64 value:

```
bits:     | 63   ....  52  |  51   ....   16 | 15 - 12 ␣
↪|11 - 0
values:   |     count      |       base      |  flags  ␣
↪| index
```

- index encodes the unique redistributor region index
- flags: reserved for future use, currently 0
- base field encodes bits [51:16] of the guest physical base address of the first redistributor in the region.
- count encodes the number of redistributors in the region. Must be greater than 0.

There are two 64K pages for each redistributor in the region and redistributors are laid out contiguously within the region. Regions are filled with redistributors in the index order. The sum of all region count fields must be greater than or equal to the number of VCPUs. Redistributor regions must be registered in the incremental index order, starting from index 0.

The characteristics of a specific redistributor region can be read by presetting the index field in the attr data. Only valid for KVM_DEV_TYPE_ARM_VGIC_V3.

It is invalid to mix calls with KVM_VGIC_V3_ADDR_TYPE_REDIST and KVM_VGIC_V3_ADDR_TYPE_REDIST_REGION attributes.

Errors:

-E2BIG	Address outside of addressable IPA range
-EINVAL	Incorrectly aligned address, bad redistributor region count/index, mixed redistributor region attribute usage
-EEXIST	Address already configured
-ENOENT	Attempt to read the characteristics of a non existing redistributor region
-ENXIO	The group or attribute is unknown/unsupported for this device or hardware support is missing.
-EFAULT	Invalid user pointer for attr->addr.

KVM_DEV_ARM_VGIC_GRP_DIST_REGS, KVM_DEV_ARM_VGIC_GRP_REDIST_REG
Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63   ....  32  |  31   ....    0 |
values:   |      mpidr     |      offset     |
```

All distributor regs are (rw, 32-bit) and kvm_device_attr.addr points to a __u32 value. 64-bit registers must be accessed by separately accessing the lower and higher word.

Writes to read-only registers are ignored by the kernel.

KVM_DEV_ARM_VGIC_GRP_DIST_REGS accesses the main distributor registers. KVM_DEV_ARM_VGIC_GRP_REDIST_REGS accesses the redistributor of the CPU specified by the mpidr.

The offset is relative to the "[Re]Distributor base address" as defined in the GICv3/4 specs. Getting or setting such a register has the same effect as reading or writing the register on real hardware, except for the following registers: GICD_STATUSR, GICR_STATUSR, GICD_ISPENDR, GICR_ISPENDR0, GICD_ICPENDR, and GICR_ICPENDR0. These registers behave differently when accessed via this interface compared to their architecturally defined behavior to allow software a full view of the VGIC's internal state.

The mpidr field is used to specify which redistributor is accessed. The mpidr is ignored for the distributor.

The mpidr encoding is based on the affinity information in the architecture defined MPIDR, and the field is encoded as follows:

```
| 63 .... 56 | 55 .... 48 | 47 .... 40 | 39 .... 32 |
|    Aff3    |    Aff2    |    Aff1    |    Aff0    |
```

Note that distributor fields are not banked, but return the same value regardless of the mpidr used to access the register.

GICD_IIDR.Revision is updated when the KVM implementation is changed in a way directly observable by the guest or userspace. Userspace should read GICD_IIDR from KVM and write back the read value to confirm its expected behavior is aligned with the KVM implementation. Userspace should set GICD_IIDR before setting any other registers to ensure the expected behavior.

The GICD_STATUSR and GICR_STATUSR registers are architecturally defined such that a write of a clear bit has no effect, whereas a write with a set bit clears that value. To allow userspace to freely set the values of these two registers, setting the attributes with the register offsets for these two registers simply sets the non-reserved bits to the value written.

Accesses (reads and writes) to the GICD_ISPENDR register region and GICR_ISPENDR0 registers get/set the value of the latched pending state for the interrupts.

This is identical to the value returned by a guest read from ISPENDR for an edge triggered interrupt, but may differ for level

triggered interrupts. For edge triggered interrupts, once an interrupt becomes pending (whether because of an edge detected on the input line or because of a guest write to ISPENDR) this state is "latched", and only cleared when either the interrupt is activated or when the guest writes to ICPENDR. A level triggered interrupt may be pending either because the level input is held high by a device, or because of a guest write to the ISPENDR register. Only ISPENDR writes are latched; if the device lowers the line level then the interrupt is no longer pending unless the guest also wrote to ISPENDR, and conversely writes to ICPENDR or activations of the interrupt do not clear the pending status if the line level is still being held high. (These rules are documented in the GICv3 specification descriptions of the ICPENDR and ISPENDR registers.) For a level triggered interrupt the value accessed here is that of the latch which is set by ISPENDR and cleared by ICPENDR or interrupt activation, whereas the value returned by a guest read from ISPENDR is the logical OR of the latch value and the input line level.

Raw access to the latch state is provided to userspace so that it can save and restore the entire GIC internal state (which is defined by the combination of the current input line level and the latch state, and cannot be deduced from purely the line level and the value of the ISPENDR registers).

Accesses to GICD_ICPENDR register region and GICR_ICPENDR0 registers have RAZ/WI semantics, meaning that reads always return 0 and writes are always ignored.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	One or more VCPUs are running

KVM_DEV_ARM_VGIC_GRP_CPU_SYSREGS Attributes:

The attr field of kvm_device_attr encodes two values:

```
bits:     | 63      ....       32 | 31  ....  16 | 15  ....
↪  0 |
values:   |         mpidr          |      RES     |   ␣
↪instr    |
```

The mpidr field encodes the CPU ID based on the affinity information in the architecture defined MPIDR, and the field is encoded as follows:

```
| 63 .... 56 | 55 .... 48 | 47 .... 40 | 39 .... 32 |
|    Aff3    |    Aff2    |    Aff1    |    Aff0    |
```

The instr field encodes the system register to access based on the fields defined in the A64 instruction set encoding for system register access (RES means the bits are reserved for future use and should be zero):

```
| 15 ... 14 | 13 ... 11 | 10 ... 7 | 6 ... 3 | 2 ... 0 |
|   Op 0    |    Op1    |    CRn   |   CRm   |   Op2   |
```

All system regs accessed through this API are (rw, 64-bit) and kvm_device_attr.addr points to a __u64 value.

KVM_DEV_ARM_VGIC_GRP_CPU_SYSREGS accesses the CPU interface registers for the CPU specified by the mpidr field.

CPU interface registers access is not implemented for AArch32 mode. Error -ENXIO is returned when accessed in AArch32 mode.

Errors:

-ENXIO	Getting or setting this register is not yet supported
-EBUSY	VCPU is running
-EINVAL	Invalid mpidr or register value supplied

KVM_DEV_ARM_VGIC_GRP_NR_IRQS Attributes:

A value describing the number of interrupts (SGI, PPI and SPI) for this GIC instance, ranging from 64 to 1024, in increments of 32.

kvm_device_attr.addr points to a __u32 value.

Errors:

-EINVAL	Value set is out of the expected range
-EBUSY	Value has already be set.

KVM_DEV_ARM_VGIC_GRP_CTRL Attributes:

KVM_DEV_ARM_VGIC_CTRL_INIT request the initialization of the VGIC, no additional parameter in kvm_device_attr.addr.

KVM_DEV_ARM_VGIC_SAVE_PENDING_TABLES save all LPI pending bits into guest RAM pending tables.

The first kB of the pending table is not altered by this operation.

Errors:

-ENXIO	VGIC not properly configured as required prior to calling this attribute
-ENODEV	no online VCPU
-ENOMEM	memory shortage when allocating vgic internal data
-EFAULT	Invalid guest ram access
-EBUSY	One or more VCPUS are running

KVM_DEV_ARM_VGIC_GRP_LEVEL_INFO Attributes:

The attr field of kvm_device_attr encodes the following values:

```
bits:     | 63      ....       32 | 31   ....    10 | 9  ..
↪..  0 |
values:   |         mpidr         |      info       |  ␣
↪vINTID   |
```

The vINTID specifies which set of IRQs is reported on.

The info field specifies which information userspace wants to get or set using this interface. Currently we support the following info values:

VGIC_LEVEL_INFO_LINE_LEVEL: Get/Set the input level of the IRQ line for a set of 32 contiguously numbered interrupts.

vINTID must be a multiple of 32.

kvm_device_attr.addr points to a __u32 value which will contain a bitmap where a set bit means the interrupt level is asserted.

Bit[n] indicates the status for interrupt vINTID + n.

SGIs and any interrupt with a higher ID than the number of interrupts supported, will be RAZ/WI. LPIs are always edge-triggered and are therefore not supported by this interface.

PPIs are reported per VCPU as specified in the mpidr field, and SPIs are reported with the same value regardless of the mpidr specified.

The mpidr field encodes the CPU ID based on the affinity information in the architecture defined MPIDR, and the field is encoded as follows:

```
| 63 .... 56 | 55 .... 48 | 47 .... 40 | 39 .... 32 |
|    Aff3    |    Aff2    |    Aff1    |    Aff0    |
```

Errors:

-EINVAL	vINTID is not multiple of 32 or info field is not VGIC_LEVEL_INFO_LINE_LEVEL

1.18.4 MPIC interrupt controller

Device types supported:

- KVM_DEV_TYPE_FSL_MPIC_20 Freescale MPIC v2.0
- KVM_DEV_TYPE_FSL_MPIC_42 Freescale MPIC v4.2

Only one MPIC instance, of any type, may be instantiated. The created MPIC will act as the system interrupt controller, connecting to each vcpu's interrupt inputs.

Groups:

KVM_DEV_MPIC_GRP_MISC Attributes:

KVM_DEV_MPIC_BASE_ADDR (rw, 64-bit) Base address of the 256 KiB MPIC register space. Must be naturally aligned. A value of zero disables the mapping. Reset value is zero.

KVM_DEV_MPIC_GRP_REGISTER (rw, 32-bit) Access an MPIC register, as if the access were made from the guest. "attr" is the byte offset into the MPIC register space. Accesses must be 4-byte aligned.

MSIs may be signaled by using this attribute group to write to the relevant MSIIR.

KVM_DEV_MPIC_GRP_IRQ_ACTIVE (rw, 32-bit) IRQ input line for each standard openpic source. 0 is inactive and 1 is active, regardless of interrupt sense.

For edge-triggered interrupts: Writing 1 is considered an activating edge, and writing 0 is ignored. Reading returns 1 if a previously signaled edge has not been acknowledged, and 0 otherwise.

"attr" is the IRQ number. IRQ numbers for standard sources are the byte offset of the relevant IVPR from EIVPR0, divided by 32.

IRQ Routing:

The MPIC emulation supports IRQ routing. Only a single MPIC device can be instantiated. Once that device has been created, it's available as irqchip id 0.

This irqchip 0 has 256 interrupt pins, which expose the interrupts in the main array of interrupt sources (a.k.a. "SRC" interrupts).

The numbering is the same as the MPIC device tree binding – based on the register offset from the beginning of the sources array, without regard to any subdivisions in chip documentation such as "internal" or "external" interrupts.

Access to non-SRC interrupts is not implemented through IRQ routing mechanisms.

1.18.5 FLIC (floating interrupt controller)

FLIC handles floating (non per-cpu) interrupts, i.e. I/O, service and some machine check interruptions. All interrupts are stored in a per-vm list of pending interrupts. FLIC performs operations on this list.

Only one FLIC instance may be instantiated.

FLIC provides support to - add interrupts (KVM_DEV_FLIC_ENQUEUE) - inspect currently pending interrupts (KVM_FLIC_GET_ALL_IRQS) - purge all pending floating interrupts (KVM_DEV_FLIC_CLEAR_IRQS) - purge one pending floating I/O interrupt (KVM_DEV_FLIC_CLEAR_IO_IRQ) - enable/disable for the guest transparent async page faults - register and modify adapter interrupt sources (KVM_DEV_FLIC_ADAPTER_*) - modify AIS (adapter-interruption-suppression) mode state (KVM_DEV_FLIC_AISM) - inject adapter interrupts on

a specified adapter (KVM_DEV_FLIC_AIRQ_INJECT) - get/set all AIS mode states (KVM_DEV_FLIC_AISM_ALL)

Groups:

KVM_DEV_FLIC_ENQUEUE Passes a buffer and length into the kernel which are then injected into the list of pending interrupts. attr->addr contains the pointer to the buffer and attr->attr contains the length of the buffer. The format of the data structure kvm_s390_irq as it is copied from userspace is defined in usr/include/linux/kvm.h.

KVM_DEV_FLIC_GET_ALL_IRQS Copies all floating interrupts into a buffer provided by userspace. When the buffer is too small it returns -ENOMEM, which is the indication for userspace to try again with a bigger buffer.

-ENOBUFS is returned when the allocation of a kernelspace buffer has failed.

-EFAULT is returned when copying data to userspace failed. All interrupts remain pending, i.e. are not deleted from the list of currently pending interrupts. attr->addr contains the userspace address of the buffer into which all interrupt data will be copied. attr->attr contains the size of the buffer in bytes.

KVM_DEV_FLIC_CLEAR_IRQS Simply deletes all elements from the list of currently pending floating interrupts. No interrupts are injected into the guest.

KVM_DEV_FLIC_CLEAR_IO_IRQ Deletes one (if any) I/O interrupt for a subchannel identified by the subsystem identification word passed via the buffer specified by attr->addr (address) and attr->attr (length).

KVM_DEV_FLIC_APF_ENABLE Enables async page faults for the guest. So in case of a major page fault the host is allowed to handle this async and continues the guest.

KVM_DEV_FLIC_APF_DISABLE_WAIT Disables async page faults for the guest and waits until already pending async page faults are done. This is necessary to trigger a completion interrupt for every init interrupt before migrating the interrupt list.

KVM_DEV_FLIC_ADAPTER_REGISTER

Register an I/O adapter interrupt source. Takes a kvm_s390_io_adapter describing the adapter to register:

```
struct kvm_s390_io_adapter {
        __u32 id;
        __u8 isc;
        __u8 maskable;
        __u8 swap;
        __u8 flags;
};
```

id contains the unique id for the adapter, isc the I/O interruption subclass to use, maskable whether this adapter may be masked (interrupts turned

off), swap whether the indicators need to be byte swapped, and flags contains further characteristics of the adapter.

Currently defined values for 'flags' are:

- KVM_S390_ADAPTER_SUPPRESSIBLE: adapter is subject to AIS (adapter-interrupt-suppression) facility. This flag only has an effect if the AIS capability is enabled.

Unknown flag values are ignored.

KVM_DEV_FLIC_ADAPTER_MODIFY Modifies attributes of an existing I/O adapter interrupt source. Takes a kvm_s390_io_adapter_req specifying the adapter and the operation:

```
struct kvm_s390_io_adapter_req {
        __u32 id;
        __u8 type;
        __u8 mask;
        __u16 pad0;
        __u64 addr;
};
```

id specifies the adapter and type the operation. The supported operations are:

KVM_S390_IO_ADAPTER_MASK mask or unmask the adapter, as specified in mask

KVM_S390_IO_ADAPTER_MAP This is now a no-op. The mapping is purely done by the irq route.

KVM_S390_IO_ADAPTER_UNMAP This is now a no-op. The mapping is purely done by the irq route.

KVM_DEV_FLIC_AISM modify the adapter-interruption-suppression mode for a given isc if the AIS capability is enabled. Takes a kvm_s390_ais_req describing:

```
struct kvm_s390_ais_req {
        __u8 isc;
        __u16 mode;
};
```

isc contains the target I/O interruption subclass, mode the target adapter-interruption-suppression mode. The following modes are currently supported:

- KVM_S390_AIS_MODE_ALL: ALL-Interruptions Mode, i.e. airq injection is always allowed;
- KVM_S390_AIS_MODE_SINGLE: SINGLE-Interruption Mode, i.e. airq injection is only allowed once and the following adapter interrupts will be suppressed until the mode is set again to ALL-Interruptions or SINGLE-Interruption mode.

KVM_DEV_FLIC_AIRQ_INJECT Inject adapter interrupts on a specified adapter. attr->attr contains the unique id for the adapter, which allows for adapter-specific checks and actions. For adapters subject to

AIS, handle the airq injection suppression for an isc according to the adapter-interruption-suppression mode on condition that the AIS capability is enabled.

KVM_DEV_FLIC_AISM_ALL Gets or sets the adapter-interruption-suppression mode for all ISCs. Takes a kvm_s390_ais_all describing:

```
struct kvm_s390_ais_all {
       __u8 simm; /* Single-Interruption-Mode mask */
       __u8 nimm; /* No-Interruption-Mode mask *
};
```

simm contains Single-Interruption-Mode mask for all ISCs, nimm contains No-Interruption-Mode mask for all ISCs. Each bit in simm and nimm corresponds to an ISC (MSB0 bit 0 to ISC 0 and so on). The combination of simm bit and nimm bit presents AIS mode for a ISC.

KVM_DEV_FLIC_AISM_ALL is indicated by KVM_CAP_S390_AIS_MIGRATION.

Note: The KVM_SET_DEVICE_ATTR/KVM_GET_DEVICE_ATTR device ioctls executed on FLIC with an unknown group or attribute gives the error code EINVAL (instead of ENXIO, as specified in the API documentation). It is not possible to conclude that a FLIC operation is unavailable based on the error code resulting from a usage attempt.

Note: The KVM_DEV_FLIC_CLEAR_IO_IRQ ioctl will return EINVAL in case a zero schid is specified.

1.18.6 Generic vcpu interface

The virtual cpu "device" also accepts the ioctls KVM_SET_DEVICE_ATTR, KVM_GET_DEVICE_ATTR, and KVM_HAS_DEVICE_ATTR. The interface uses the same struct kvm_device_attr as other devices, but targets VCPU-wide settings and controls.

The groups and attributes per virtual cpu, if any, are architecture specific.

1. GROUP: KVM_ARM_VCPU_PMU_V3_CTRL

Architectures ARM64

1.1. ATTRIBUTE: KVM_ARM_VCPU_PMU_V3_IRQ

Parameters in kvm_device_attr.addr the address for PMU overflow interrupt is a pointer to an int

Returns:

-EBUSY	The PMU overflow interrupt is already set
-ENXIO	The overflow interrupt not set when attempting to get it
-ENODEV	PMUv3 not supported
-EINVAL	Invalid PMU overflow interrupt number supplied or trying to set the IRQ number without using an in-kernel irqchip.

A value describing the PMUv3 (Performance Monitor Unit v3) overflow interrupt number for this vcpu. This interrupt could be a PPI or SPI, but the interrupt type must be same for each vcpu. As a PPI, the interrupt number is the same for all vcpus, while as an SPI it must be a separate number per vcpu.

1.2 ATTRIBUTE: KVM_ARM_VCPU_PMU_V3_INIT

Parameters no additional parameter in kvm_device_attr.addr

Returns:

-ENODEV	PMUv3 not supported or GIC not initialized
-ENXIO	PMUv3 not properly configured or in-kernel irqchip not configured as required prior to calling this attribute
-EBUSY	PMUv3 already initialized

Request the initialization of the PMUv3. If using the PMUv3 with an in-kernel virtual GIC implementation, this must be done after initializing the in-kernel irqchip.

2. GROUP: KVM_ARM_VCPU_TIMER_CTRL

Architectures ARM, ARM64

2.1. ATTRIBUTES: KVM_ARM_VCPU_TIMER_IRQ_VTIMER, KVM_ARM_VCPU_TIMER_IRQ_PTIMER

Parameters in kvm_device_attr.addr the address for the timer interrupt is a pointer to an int

Returns:

-EINVAL	Invalid timer interrupt number
-EBUSY	One or more VCPUs has already run

A value describing the architected timer interrupt number when connected to an in-kernel virtual GIC. These must be a PPI (16 <= intid < 32). Setting the attribute overrides the default values (see below).

KVM_ARM_VCPU_TIMER_IRQ_VTIMER	The EL1 virtual timer intid (default: 27)
KVM_ARM_VCPU_TIMER_IRQ_PTIMER	The EL1 physical timer intid (default: 30)

Setting the same PPI for different timers will prevent the VCPUs from running. Setting the interrupt number on a VCPU configures all VCPUs created at that time to use the number provided for a given timer, overwriting any previously configured values on other VCPUs. Userspace should configure the interrupt numbers on at least one VCPU after creating all VCPUs and before running any VCPUs.

3. GROUP: KVM_ARM_VCPU_PVTIME_CTRL

Architectures ARM64

3.1 ATTRIBUTE: KVM_ARM_VCPU_PVTIME_IPA

Parameters 64-bit base address

Returns:

-ENXIO	Stolen time not implemented
-EEXIST	Base address already set for this VCPU
-EINVAL	Base address not 64 byte aligned

Specifies the base address of the stolen time structure for this VCPU. The base address must be 64 byte aligned and exist within a valid guest memory region. See Documentation/virt/kvm/arm/pvtime.rst for more information including the layout of the stolen time structure.

1.18.7 VFIO virtual device

Device types supported:

- KVM_DEV_TYPE_VFIO

Only one VFIO instance may be created per VM. The created device tracks VFIO groups in use by the VM and features of those groups important to the correctness and acceleration of the VM. As groups are enabled and disabled for use by the VM, KVM should be updated about their presence. When registered with KVM, a reference to the VFIO-group is held by KVM.

Groups: KVM_DEV_VFIO_GROUP

KVM_DEV_VFIO_GROUP attributes:

KVM_DEV_VFIO_GROUP_ADD: Add a VFIO group to VFIO-KVM device tracking
kvm_device_attr.addr points to an int32_t file descriptor for the VFIO group.

KVM_DEV_VFIO_GROUP_DEL: Remove a VFIO group from VFIO-KVM device trac
kvm_device_attr.addr points to an int32_t file descriptor for the VFIO group.

KVM_DEV_VFIO_GROUP_SET_SPAPR_TCE: attaches a guest visible TCE table
allocated by sPAPR KVM. kvm_device_attr.addr points to a struct:

```
struct kvm_vfio_spapr_tce {
        __s32   groupfd;
        __s32   tablefd;
};
```

where:

- @groupfd is a file descriptor for a VFIO group;
- @tablefd is a file descriptor for a TCE table allocated via KVM_CREATE_SPAPR_TCE.

1.18.8 Generic vm interface

The virtual machine "device" also accepts the ioctls KVM_SET_DEVICE_ATTR, KVM_GET_DEVICE_ATTR, and KVM_HAS_DEVICE_ATTR. The interface uses the same struct kvm_device_attr as other devices, but targets VM-wide settings and controls.

The groups and attributes per virtual machine, if any, are architecture specific.

1. GROUP: KVM_S390_VM_MEM_CTRL

Architectures s390

1.1. ATTRIBUTE: KVM_S390_VM_MEM_ENABLE_CMMA

Parameters none

Returns -EBUSY if a vcpu is already defined, otherwise 0

Enables Collaborative Memory Management Assist (CMMA) for the virtual machine.

1.2. ATTRIBUTE: KVM_S390_VM_MEM_CLR_CMMA

Parameters none

Returns -EINVAL if CMMA was not enabled; 0 otherwise

Clear the CMMA status for all guest pages, so any pages the guest marked as unused are again used any may not be reclaimed by the host.

1.3. ATTRIBUTE KVM_S390_VM_MEM_LIMIT_SIZE

Parameters in attr->addr the address for the new limit of guest memory

Returns -EFAULT if the given address is not accessible; -EINVAL if the virtual machine is of type UCONTROL; -E2BIG if the given guest memory is to big for that machine; -EBUSY if a vcpu is already defined; -ENOMEM if not enough memory is available for a new shadow guest mapping; 0 otherwise.

Allows userspace to query the actual limit and set a new limit for the maximum guest memory size. The limit will be rounded up to 2048 MB, 4096 GB, 8192 TB respectively, as this limit is governed by the number of page table levels. In the case that there is no limit we will set the limit to KVM_S390_NO_MEM_LIMIT (U64_MAX).

2. GROUP: KVM_S390_VM_CPU_MODEL

Architectures s390

2.1. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE (r/o)

Allows user space to retrieve machine and kvm specific cpu related information:

```
struct kvm_s390_vm_cpu_machine {
     __u64 cpuid;           # CPUID of host
     __u32 ibc;             # IBC level range offered by host
     __u8  pad[4];
     __u64 fac_mask[256];   # set of cpu facilities enabled by KVM
     __u64 fac_list[256];   # set of cpu facilities offered by host
}
```

Parameters address of buffer to store the machine related cpu data of type struct kvm_s390_vm_cpu_machine*

Returns -EFAULT if the given address is not accessible from kernel space; -ENOMEM if not enough memory is available to process the ioctl; 0 in case of success.

2.2. ATTRIBUTE: KVM_S390_VM_CPU_PROCESSOR (r/w)

Allows user space to retrieve or request to change cpu related information for a vcpu:

```
struct kvm_s390_vm_cpu_processor {
     __u64 cpuid;           # CPUID currently (to be) used by this vcpu
     __u16 ibc;             # IBC level currently (to be) used by this vcpu
     __u8  pad[6];
     __u64 fac_list[256];   # set of cpu facilities currently (to be) used
                            # by this vcpu
}
```

KVM does not enforce or limit the cpu model data in any form. Take the information retrieved by means of KVM_S390_VM_CPU_MACHINE as hint for reasonable configuration setups. Instruction interceptions triggered by additionally set facility bits that are not handled by KVM need to by imlemented in the VM driver code.

Parameters address of buffer to store/set the processor related cpu data of type struct kvm_s390_vm_cpu_processor*.

Returns -EBUSY in case 1 or more vcpus are already activated (only in write case); -EFAULT if the given address is not accessible from kernel space; -ENOMEM if not enough memory is available to process the ioctl; 0 in case of success.

2.3. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_FEAT (r/o)

Allows user space to retrieve available cpu features. A feature is available if provided by the hardware and supported by kvm. In theory, cpu features could even be completely emulated by kvm.

```
struct kvm_s390_vm_cpu_feat {
      __u64 feat[16]; # Bitmap (1 = feature available), MSB 0 bit numbering
};
```

Parameters address of a buffer to load the feature list from.

Returns -EFAULT if the given address is not accessible from kernel space; 0 in case of success.

2.4. ATTRIBUTE: KVM_S390_VM_CPU_PROCESSOR_FEAT (r/w)

Allows user space to retrieve or change enabled cpu features for all VCPUs of a VM. Features that are not available cannot be enabled.

See 2.3. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_FEAT (r/o) for a description of the parameter struct.

Parameters address of a buffer to store/load the feature list from.

Returns -EFAULT if the given address is not accessible from kernel space; -EINVAL if a cpu feature that is not available is to be enabled; -EBUSY if at least one VCPU has already been defined; 0 in case of success.

2.5. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_SUBFUNC (r/o)

Allows user space to retrieve available cpu subfunctions without any filtering done by a set IBC. These subfunctions are indicated to the guest VCPU via query or “test bit” subfunctions and used e.g. by cpacf functions, plo and ptff.

A subfunction block is only valid if KVM_S390_VM_CPU_MACHINE contains the STFL(E) bit introducing the affected instruction. If the affected instruction indicates subfunctions via a “query subfunction”, the response block is contained in the returned struct. If the affected instruction indicates subfunctions via a “test bit”mechanism, the subfunction codes are contained in the returned struct in MSB 0 bit numbering.

```
struct kvm_s390_vm_cpu_subfunc {
     u8 plo[32];           # always valid (ESA/390 feature)
     u8 ptff[16];          # valid with TOD-clock steering
     u8 kmac[16];          # valid with Message-Security-Assist
     u8 kmc[16];           # valid with Message-Security-Assist
     u8 km[16];            # valid with Message-Security-Assist
     u8 kimd[16];          # valid with Message-Security-Assist
     u8 klmd[16];          # valid with Message-Security-Assist
     u8 pckmo[16];         # valid with Message-Security-Assist-Extension 3
     u8 kmctr[16];         # valid with Message-Security-Assist-Extension 4
```

(continues on next page)

(continued from previous page)

```
    u8 kmf[16];           # valid with Message-Security-Assist-Extension 4
    u8 kmo[16];           # valid with Message-Security-Assist-Extension 4
    u8 pcc[16];           # valid with Message-Security-Assist-Extension 4
    u8 ppno[16];          # valid with Message-Security-Assist-Extension 5
    u8 kma[16];           # valid with Message-Security-Assist-Extension 8
    u8 kdsa[16];          # valid with Message-Security-Assist-Extension 9
    u8 reserved[1792];    # reserved for future instructions
};
```

Parameters address of a buffer to load the subfunction blocks from.

Returns -EFAULT if the given address is not accessible from kernel space; 0 in case of success.

2.6. ATTRIBUTE: KVM_S390_VM_CPU_PROCESSOR_SUBFUNC (r/w)

Allows user space to retrieve or change cpu subfunctions to be indicated for all VCPUs of a VM. This attribute will only be available if kernel and hardware support are in place.

The kernel uses the configured subfunction blocks for indication to the guest. A subfunction block will only be used if the associated STFL(E) bit has not been disabled by user space (so the instruction to be queried is actually available for the guest).

As long as no data has been written, a read will fail. The IBC will be used to determine available subfunctions in this case, this will guarantee backward compatibility.

See 2.5. ATTRIBUTE: KVM_S390_VM_CPU_MACHINE_SUBFUNC (r/o) for a description of the parameter struct.

Parameters address of a buffer to store/load the subfunction blocks from.

Returns -EFAULT if the given address is not accessible from kernel space; -EINVAL when reading, if there was no write yet; -EBUSY if at least one VCPU has already been defined; 0 in case of success.

3. GROUP: KVM_S390_VM_TOD

Architectures s390

3.1. ATTRIBUTE: KVM_S390_VM_TOD_HIGH

Allows user space to set/get the TOD clock extension (u8) (superseded by KVM_S390_VM_TOD_EXT).

Parameters address of a buffer in user space to store the data (u8) to

Returns -EFAULT if the given address is not accessible from kernel space; -EINVAL if setting the TOD clock extension to != 0 is not supported

3.2. ATTRIBUTE: KVM_S390_VM_TOD_LOW

Allows user space to set/get bits 0-63 of the TOD clock register as defined in the POP (u64).

Parameters address of a buffer in user space to store the data (u64) to

Returns -EFAULT if the given address is not accessible from kernel space

3.3. ATTRIBUTE: KVM_S390_VM_TOD_EXT

Allows user space to set/get bits 0-63 of the TOD clock register as defined in the POP (u64). If the guest CPU model supports the TOD clock extension (u8), it also allows user space to get/set it. If the guest CPU model does not support it, it is stored as 0 and not allowed to be set to a value != 0.

Parameters address of a buffer in user space to store the data (kvm_s390_vm_tod_clock) to

Returns -EFAULT if the given address is not accessible from kernel space; -EINVAL if setting the TOD clock extension to != 0 is not supported

4. GROUP: KVM_S390_VM_CRYPTO

Architectures s390

4.1. ATTRIBUTE: KVM_S390_VM_CRYPTO_ENABLE_AES_KW (w/o)

Allows user space to enable aes key wrapping, including generating a new wrapping key.

Parameters none

Returns 0

4.2. ATTRIBUTE: KVM_S390_VM_CRYPTO_ENABLE_DEA_KW (w/o)

Allows user space to enable dea key wrapping, including generating a new wrapping key.

Parameters none

Returns 0

4.3. ATTRIBUTE: KVM_S390_VM_CRYPTO_DISABLE_AES_KW (w/o)

Allows user space to disable aes key wrapping, clearing the wrapping key.

Parameters none

Returns 0

4.4. ATTRIBUTE: KVM_S390_VM_CRYPTO_DISABLE_DEA_KW (w/o)

Allows user space to disable dea key wrapping, clearing the wrapping key.

Parameters none

Returns 0

5. GROUP: KVM_S390_VM_MIGRATION

Architectures s390

5.1. ATTRIBUTE: KVM_S390_VM_MIGRATION_STOP (w/o)

Allows userspace to stop migration mode, needed for PGSTE migration. Setting this attribute when migration mode is not active will have no effects.

Parameters none

Returns 0

5.2. ATTRIBUTE: KVM_S390_VM_MIGRATION_START (w/o)

Allows userspace to start migration mode, needed for PGSTE migration. Setting this attribute when migration mode is already active will have no effects.

Parameters none

Returns -ENOMEM if there is not enough free memory to start migration mode; -EINVAL if the state of the VM is invalid (e.g. no memory defined); 0 in case of success.

5.3. ATTRIBUTE: KVM_S390_VM_MIGRATION_STATUS (r/o)

Allows userspace to query the status of migration mode.

Parameters address of a buffer in user space to store the data (u64) to; the data itself is either 0 if migration mode is disabled or 1 if it is enabled

Returns -EFAULT if the given address is not accessible from kernel space; 0 in case of success.

1.18.9 XICS interrupt controller

Device type supported: KVM_DEV_TYPE_XICS

Groups:

1. **KVM_DEV_XICS_GRP_SOURCES** Attributes:

   One per interrupt source, indexed by the source number.

2. **KVM_DEV_XICS_GRP_CTRL** Attributes:

   2.1 KVM_DEV_XICS_NR_SERVERS (write only)

   The kvm_device_attr.addr points to a __u32 value which is the number of interrupt server numbers (ie, highest possible vcpu id plus one).

   Errors:

-EINVAL	Value greater than KVM_MAX_VCPU_ID.
-EFAULT	Invalid user pointer for attr->addr.
-EBUSY	A vcpu is already connected to the device.

This device emulates the XICS (eXternal Interrupt Controller Specification) defined in PAPR. The XICS has a set of interrupt sources, each identified by a 20-bit source number, and a set of Interrupt Control Presentation (ICP) entities, also called “servers”, each associated with a virtual CPU.

The ICP entities are created by enabling the KVM_CAP_IRQ_ARCH capability for each vcpu, specifying KVM_CAP_IRQ_XICS in args[0] and the interrupt server number (i.e. the vcpu number from the XICS’s point of view) in args[1] of the kvm_enable_cap struct. Each ICP has 64 bits of state which can be read and written using the KVM_GET_ONE_REG and KVM_SET_ONE_REG ioctls on the vcpu.

The 64 bit state word has the following bitfields, starting at the least-significant end of the word:

- Unused, 16 bits
- Pending interrupt priority, 8 bits Zero is the highest priority, 255 means no interrupt is pending.
- Pending IPI (inter-processor interrupt) priority, 8 bits Zero is the highest priority, 255 means no IPI is pending.
- Pending interrupt source number, 24 bits Zero means no interrupt pending, 2 means an IPI is pending
- Current processor priority, 8 bits Zero is the highest priority, meaning no interrupts can be delivered, and 255 is the lowest priority.

Each source has 64 bits of state that can be read and written using the KVM_GET_DEVICE_ATTR and KVM_SET_DEVICE_ATTR ioctls, specifying the KVM_DEV_XICS_GRP_SOURCES attribute group, with the attribute number being the interrupt source number. The 64 bit state word has the following bitfields, starting from the least-significant end of the word:

- Destination (server number), 32 bits

  This specifies where the interrupt should be sent, and is the interrupt server number specified for the destination vcpu.

- Priority, 8 bits

  This is the priority specified for this interrupt source, where 0 is the highest priority and 255 is the lowest. An interrupt with a priority of 255 will never be delivered.

- Level sensitive flag, 1 bit

  This bit is 1 for a level-sensitive interrupt source, or 0 for edge-sensitive (or MSI).

- Masked flag, 1 bit

  This bit is set to 1 if the interrupt is masked (cannot be delivered regardless of its priority), for example by the ibm,int-off RTAS call, or 0 if it is not masked.

- Pending flag, 1 bit

  This bit is 1 if the source has a pending interrupt, otherwise 0.

Only one XICS instance may be created per VM.

1.18.10 POWER9 eXternal Interrupt Virtualization Engine (XIVE Gen1)

Device types supported:

- KVM_DEV_TYPE_XIVE POWER9 XIVE Interrupt Controller generation 1

This device acts as a VM interrupt controller. It provides the KVM interface to configure the interrupt sources of a VM in the underlying POWER9 XIVE interrupt controller.

Only one XIVE instance may be instantiated. A guest XIVE device requires a POWER9 host and the guest OS should have support for the XIVE native exploitation interrupt mode. If not, it should run using the legacy interrupt mode, referred as XICS (POWER7/8).

- Device Mappings

  The KVM device exposes different MMIO ranges of the XIVE HW which are required for interrupt management. These are exposed to the guest in VMAs populated with a custom VM fault handler.

  1. Thread Interrupt Management Area (TIMA)

  Each thread has an associated Thread Interrupt Management context composed of a set of registers. These registers let the thread handle priority management and interrupt acknowledgment. The most important are :

  - Interrupt Pending Buffer (IPB)
  - Current Processor Priority (CPPR)
  - Notification Source Register (NSR)

  They are exposed to software in four different pages each proposing a view with a different privilege. The first page is for the physical thread context and the second for the hypervisor. Only the third (operating system) and the fourth (user level) are exposed the guest.

  2. Event State Buffer (ESB)

  Each source is associated with an Event State Buffer (ESB) with either a pair of even/odd pair of pages which provides commands to manage the source: to trigger, to EOI, to turn off the source for instance.

  3. Device pass-through

  When a device is passed-through into the guest, the source interrupts are from a different HW controller (PHB4) and the ESB pages exposed to the guest should accommadate this change.

  The passthru_irq helpers, kvmppc_xive_set_mapped() and kvmppc_xive_clr_mapped() are called when the device HW irqs are mapped into or unmapped from the guest IRQ number space. The KVM device extends these helpers to clear the ESB pages of the guest IRQ number being mapped and then lets the VM fault handler repopulate. The handler will insert the ESB page corresponding to the HW interrupt of the device being passed-through or the initial IPI ESB page if the device has being removed.

The ESB remapping is fully transparent to the guest and the OS device driver. All handling is done within VFIO and the above helpers in KVM-PPC.

- Groups:

1. **KVM_DEV_XIVE_GRP_CTRL** Provides global controls on the device

   Attributes: 1.1 KVM_DEV_XIVE_RESET (write only) Resets the interrupt controller configuration for sources and event queues. To be used by kexec and kdump.

   Errors: none

   1.2 KVM_DEV_XIVE_EQ_SYNC (write only) Sync all the sources and queues and mark the EQ pages dirty. This to make sure that a consistent memory state is captured when migrating the VM.

   Errors: none

   1.3 KVM_DEV_XIVE_NR_SERVERS (write only) The kvm_device_attr.addr points to a __u32 value which is the number of interrupt server numbers (ie, highest possible vcpu id plus one).

   Errors:

-EINVAL	Value greater than KVM_MAX_VCPU_ID.
-EFAULT	Invalid user pointer for attr->addr.
-EBUSY	A vCPU is already connected to the device.

2. **KVM_DEV_XIVE_GRP_SOURCE (write only)** Initializes a new source in the XIVE device and mask it.

   Attributes: Interrupt source number (64-bit)

   The kvm_device_attr.addr points to a __u64 value:

```
bits:     | 63   ....  2 |   1   |   0
values:   |    unused    | level | type
```

   - type: 0:MSI 1:LSI
   - level: assertion level in case of an LSI.

   Errors:

-E2BIG	Interrupt source number is out of range
-ENOMEM	Could not create a new source block
-EFAULT	Invalid user pointer for attr->addr.
-ENXIO	Could not allocate underlying HW interrupt

3. **KVM_DEV_XIVE_GRP_SOURCE_CONFIG (write only)** Configures source targeting

   Attributes: Interrupt source number (64-bit)

   The kvm_device_attr.addr points to a __u64 value:

```
bits:     | 63   ....  33 |  32  | 31 .. 3 |  2 .. 0
values:   |    eisn       | mask |  server | priority
```

- priority: 0-7 interrupt priority level
- server: CPU number chosen to handle the interrupt
- mask: mask flag (unused)
- eisn: Effective Interrupt Source Number

Errors:

-ENOENT	Unknown source number
-EINVAL	Not initialized source number
-EINVAL	Invalid priority
-EINVAL	Invalid CPU number.
-EFAULT	Invalid user pointer for attr->addr.
-ENXIO	CPU event queues not configured or configuration of the underlying HW interrupt failed
-EBUSY	No CPU available to serve interrupt

4. **KVM_DEV_XIVE_GRP_EQ_CONFIG (read-write)** Configures an event queue of a CPU

   Attributes: EQ descriptor identifier (64-bit)

   The EQ descriptor identifier is a tuple (server, priority):

```
bits:     | 63   ....  32 | 31 .. 3 |  2 .. 0
values:   |    unused     |  server | priority
```

   The kvm_device_attr.addr points to:

```
struct kvm_ppc_xive_eq {
    __u32 flags;
    __u32 qshift;
    __u64 qaddr;
    __u32 qtoggle;
    __u32 qindex;
    __u8  pad[40];
};
```

   - **flags: queue flags**

     KVM_XIVE_EQ_ALWAYS_NOTIFY (required) forces notification without using the coalescing mechanism provided by the XIVE END ESBs.

   - qshift: queue size (power of 2)
   - qaddr: real address of queue
   - qtoggle: current queue toggle bit

- qindex: current queue index
- pad: reserved for future use

Errors:

-ENOENT	Invalid CPU number
-EINVAL	Invalid priority
-EINVAL	Invalid flags
-EINVAL	Invalid queue size
-EINVAL	Invalid queue address
-EFAULT	Invalid user pointer for attr->addr.
-EIO	Configuration of the underlying HW failed

5. **KVM_DEV_XIVE_GRP_SOURCE_SYNC (write only)** Synchronize the source to flush event notifications

   Attributes: Interrupt source number (64-bit)

   Errors:

-ENOENT	Unknown source number
-EINVAL	Not initialized source number

- VCPU state

  The XIVE IC maintains VP interrupt state in an internal structure called the NVT. When a VP is not dispatched on a HW processor thread, this structure can be updated by HW if the VP is the target of an event notification.

  It is important for migration to capture the cached IPB from the NVT as it synthesizes the priorities of the pending interrupts. We capture a bit more to report debug information.

  KVM_REG_PPC_VP_STATE (2 * 64bits):

```
bits:     |  63  ....  32  |  31  ....  0  |
values:   |   TIMA word0   |   TIMA word1  |
bits:     | 127       ..........       64  |
values:   |            unused              |
```

- Migration:

  Saving the state of a VM using the XIVE native exploitation mode should follow a specific sequence. When the VM is stopped :

  1. Mask all sources (PQ=01) to stop the flow of events.

  2. Sync the XIVE device with the KVM control KVM_DEV_XIVE_EQ_SYNC to flush any in-flight event notification and to stabilize the EQs. At this stage, the EQ pages are marked dirty to make sure they are transferred in the migration sequence.

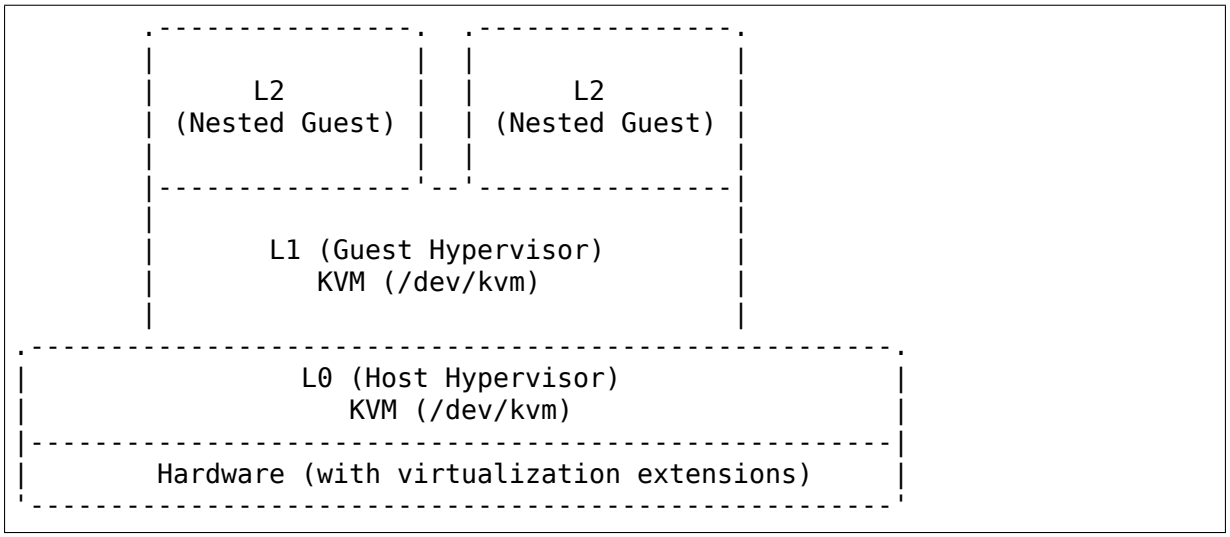
  3. Capture the state of the source targeting, the EQs configuration and the state of thread interrupt context registers.

  Restore is similar:

1. Restore the EQ configuration. As targeting depends on it.
2. Restore targeting
3. Restore the thread interrupt contexts
4. Restore the source states
5. Let the vCPU run

1.19 Running nested guests with KVM

A nested guest is the ability to run a guest inside another guest (it can be KVM-based or a different hypervisor). The straightforward example is a KVM guest that in turn runs on a KVM guest (the rest of this document is built on this example):



Terminology:

- L0 – level-0; the bare metal host, running KVM
- L1 – level-1 guest; a VM running on L0; also called the "guest hypervisor", as it itself is capable of running KVM.
- L2 – level-2 guest; a VM running on L1, this is the "nested guest"

Note: The above diagram is modelled after the x86 architecture; s390x, ppc64 and other architectures are likely to have a different design for nesting.

For example, s390x always has an LPAR (LogicalPARtition) hypervisor running on bare metal, adding another layer and resulting in at least four levels in a nested setup —L0 (bare metal, running the LPAR hypervisor), L1 (host hypervisor), L2 (guest hypervisor), L3 (nested guest).

This document will stick with the three-level terminology (L0, L1, and L2) for all architectures; and will largely focus on x86.

1.19.1 Use Cases

There are several scenarios where nested KVM can be useful, to name a few:

- As a developer, you want to test your software on different operating systems (OSes). Instead of renting multiple VMs from a Cloud Provider, using nested KVM lets you rent a large enough "guest hypervisor" (level-1 guest). This in turn allows you to create multiple nested guests (level-2 guests), running different OSes, on which you can develop and test your software.
- Live migration of "guest hypervisors" and their nested guests, for load balancing, disaster recovery, etc.
- VM image creation tools (e.g. `virt-install`, etc) often run their own VM, and users expect these to work inside a VM.
- Some OSes use virtualization internally for security (e.g. to let applications run safely in isolation).

1.19.2 Enabling "nested" (x86)

From Linux kernel v4.19 onwards, the `nested` KVM parameter is enabled by default for Intel and AMD. (Though your Linux distribution might override this default.)

In case you are running a Linux kernel older than v4.19, to enable nesting, set the `nested` KVM module parameter to `Y` or `1`. To persist this setting across reboots, you can add it in a config file, as shown below:

1. On the bare metal host (L0), list the kernel modules and ensure that the KVM modules:

```
$ lsmod | grep -i kvm
kvm_intel             133627  0
kvm                   435079  1 kvm_intel
```

2. Show information for `kvm_intel` module:

```
$ modinfo kvm_intel | grep -i nested
parm:           nested:bool
```

3. For the nested KVM configuration to persist across reboots, place the below in `/etc/modprobed/kvm_intel.conf` (create the file if it doesn't exist):

```
$ cat /etc/modprobe.d/kvm_intel.conf
options kvm-intel nested=y
```

4. Unload and re-load the KVM Intel module:

```
$ sudo rmmod kvm-intel
$ sudo modprobe kvm-intel
```

5. Verify if the `nested` parameter for KVM is enabled:

```
$ cat /sys/module/kvm_intel/parameters/nested
Y
```

For AMD hosts, the process is the same as above, except that the module name is `kvm-amd`.

1.19.3 Additional nested-related kernel parameters (x86)

If your hardware is sufficiently advanced (Intel Haswell processor or higher, which has newer hardware virt extensions), the following additional features will also be enabled by default: "Shadow VMCS (Virtual Machine Control Structure)", APIC Virtualization on your bare metal host (L0). Parameters for Intel hosts:

```
$ cat /sys/module/kvm_intel/parameters/enable_shadow_vmcs
Y

$ cat /sys/module/kvm_intel/parameters/enable_apicv
Y

$ cat /sys/module/kvm_intel/parameters/ept
Y
```

Note: If you suspect your L2 (i.e. nested guest) is running slower, ensure the above are enabled (particularly `enable_shadow_vmcs` and `ept`).

1.19.4 Starting a nested guest (x86)

Once your bare metal host (L0) is configured for nesting, you should be able to start an L1 guest with:

```
$ qemu-kvm -cpu host [...]
```

The above will pass through the host CPU's capabilities as-is to the gues); or for better live migration compatibility, use a named CPU model supported by QEMU. e.g.:

```
$ qemu-kvm -cpu Haswell-noTSX-IBRS,vmx=on
```

then the guest hypervisor will subsequently be capable of running a nested guest with accelerated KVM.

1.19.5 Enabling "nested" (s390x)

1. On the host hypervisor (L0), enable the `nested` parameter on s390x:

   ```
   $ rmmod kvm
   $ modprobe kvm nested=1
   ```

Note: On s390x, the kernel parameter `hpage` is mutually exclusive with the `nested` paramter —i.e. to be able to enable `nested`, the `hpage` parameter must be disabled.

---

2. The guest hypervisor (L1) must be provided with the `sie` CPU feature —with QEMU, this can be done by using "host passthrough" (via the command-line `-cpu host`).
3. Now the KVM module can be loaded in the L1 (guest hypervisor):

```
$ modprobe kvm
```

1.19.6 Live migration with nested KVM

Migrating an L1 guest, with a live nested guest in it, to another bare metal host, works as of Linux kernel 5.3 and QEMU 4.2.0 for Intel x86 systems, and even on older versions for s390x.

On AMD systems, once an L1 guest has started an L2 guest, the L1 guest should no longer be migrated or saved (refer to QEMU documentation on "savevm"/"loadvm") until the L2 guest shuts down. Attempting to migrate or save-and-load an L1 guest while an L2 guest is running will result in undefined behavior. You might see a `kernel BUG!` entry in `dmesg`, a kernel 'oops', or an outright kernel panic. Such a migrated or loaded L1 guest can no longer be considered stable or secure, and must be restarted. Migrating an L1 guest merely configured to support nesting, while not actually running L2 guests, is expected to function normally even on AMD systems but may fail once guests are started.

Migrating an L2 guest is always expected to succeed, so all the following scenarios should work even on AMD systems:

- Migrating a nested guest (L2) to another L1 guest on the same bare metal host.
- Migrating a nested guest (L2) to another L1 guest on a different bare metal host.
- Migrating a nested guest (L2) to a bare metal host.

1.19.7 Reporting bugs from nested setups

Debugging "nested" problems can involve sifting through log files across L0, L1 and L2; this can result in tedious back-n-forth between the bug reporter and the bug fixer.

- Mention that you are in a "nested" setup. If you are running any kind of "nesting" at all, say so. Unfortunately, this needs to be called out because when reporting bugs, people tend to forget to even mention that they're using nested virtualization.
- Ensure you are actually running KVM on KVM. Sometimes people do not have KVM enabled for their guest hypervisor (L1), which results in them running with pure emulation or what QEMU calls it as "TCG", but they think they're running nested KVM. Thus confusing "nested Virt" (which could also mean, QEMU on KVM) with "nested KVM" (KVM on KVM).

Information to collect (generic)

The following is not an exhaustive list, but a very good starting point:

- Kernel, libvirt, and QEMU version from L0
- Kernel, libvirt and QEMU version from L1
- QEMU command-line of L1 – when using libvirt, you' ll find it here: `/var/log/libvirt/qemu/instance.log`
- QEMU command-line of L2 – as above, when using libvirt, get the complete libvirt-generated QEMU command-line
- `cat /sys/cpuinfo` from L0
- `cat /sys/cpuinfo` from L1
- `lscpu` from L0
- `lscpu` from L1
- Full `dmesg` output from L0
- Full `dmesg` output from L1

x86-specific info to collect

Both the below commands, `x86info` and `dmidecode`, should be available on most Linux distributions with the same name:

- Output of: `x86info -a` from L0
- Output of: `x86info -a` from L1
- Output of: `dmidecode` from L0
- Output of: `dmidecode` from L1

s390x-specific info to collect

Along with the earlier mentioned generic details, the below is also recommended:

- `/proc/sysinfo` from L1; this will also include the info from L0

CHAPTER

TWO

USER MODE LINUX HOWTO

Author User Mode Linux Core Team

Last-updated Sat Jan 25 16:07:55 CET 2020

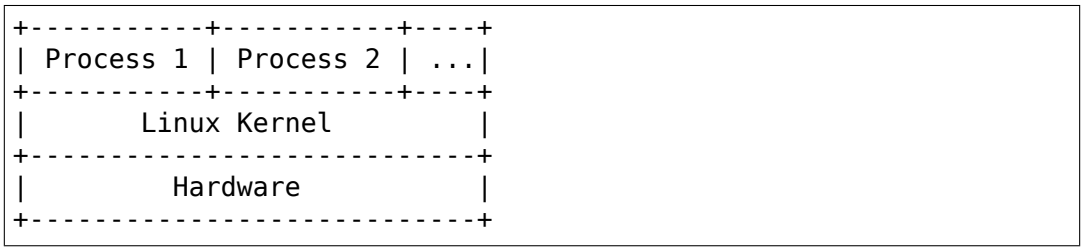
This document describes the use and abuse of Jeff Dike's User Mode Linux: a port of the Linux kernel as a normal Intel Linux process.

2.1 1. Introduction

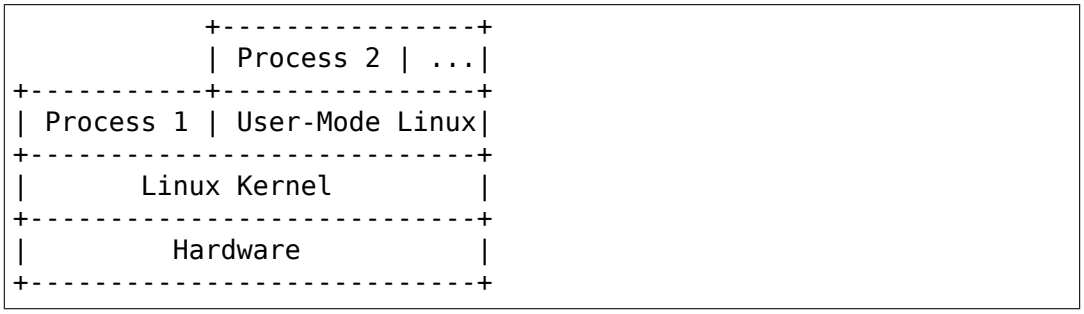
Welcome to User Mode Linux. It's going to be fun.

2.1.1 1.1. How is User Mode Linux Different?

Normally, the Linux Kernel talks straight to your hardware (video card, keyboard, hard drives, etc), and any programs which run ask the kernel to operate the hardware, like so:



The User Mode Linux Kernel is different; instead of talking to the hardware, it talks to a real Linux kernel (called the host kernel from now on), like any other program. Programs can then run inside User-Mode Linux as if they were running under a normal kernel, like so:



2.1.2 1.2. Why Would I Want User Mode Linux?

1. If User Mode Linux crashes, your host kernel is still fine.
2. You can run a usermode kernel as a non-root user.
3. You can debug the User Mode Linux like any normal process.
4. You can run gprof (profiling) and gcov (coverage testing).
5. You can play with your kernel without breaking things.
6. You can use it as a sandbox for testing new apps.
7. You can try new development kernels safely.
8. You can run different distributions simultaneously.
9. It’s extremely fun.

2.2 2. Compiling the kernel and modules

2.2.1 2.1. Compiling the kernel

Compiling the user mode kernel is just like compiling any other kernel.

1. Download the latest kernel from your favourite kernel mirror, such as:

   https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.4.14.tar.xz

2. Make a directory and unpack the kernel into it:

```
host%
mkdir ~/uml

host%
cd ~/uml

host%
tar xvf linux-5.4.14.tar.xz
```

3. Run your favorite config; `make xconfig ARCH=um` is the most convenient. `make config ARCH=um` and `make menuconfig ARCH=um` will work as well. The defaults will give you a useful kernel. If you want to change something, go ahead, it probably won’t hurt anything.

   Note: If the host is configured with a 2G/2G address space split rather than the usual 3G/1G split, then the packaged UML binaries will not run. They will immediately segfault. See 4. UML on 2G/2G hosts for the scoop on running UML on your system.

4. Finish with `make linux ARCH=um`: the result is a file called `linux` in the top directory of your source tree.

2.2.2 2.2. Compiling and installing kernel modules

UML modules are built in the same way as the native kernel (with the exception of the ‘ARCH=um’ that you always need for UML):

```
host% make modules ARCH=um
```

Any modules that you want to load into this kernel need to be built in the user-mode pool. Modules from the native kernel won’t work.

You can install them by using ftp or something to copy them into the virtual machine and dropping them into `/lib/modules/$(uname -r)`.

You can also get the kernel build process to install them as follows:

1. with the kernel not booted, mount the root filesystem in the top level of the kernel pool:

   ```
   host% mount root_fs mnt -o loop
   ```

2. run:

   ```
   host%
   make modules_install INSTALL_MOD_PATH=`pwd`/mnt ARCH=um
   ```

3. unmount the filesystem:

   ```
   host% umount mnt
   ```

4. boot the kernel on it

When the system is booted, you can use insmod as usual to get the modules into the kernel. A number of things have been loaded into UML as modules, especially filesystems and network protocols and filters, so most symbols which need to be exported probably already are. However, if you do find symbols that need exporting, let us know at http://user-mode-linux.sourceforge.net/, and they’ll be “taken care of”.

2.2.3 2.3. Compiling and installing uml_utilities

Many features of the UML kernel require a user-space helper program, so a uml_utilities package is distributed separately from the kernel patch which provides these helpers. Included within this is:

- port-helper - Used by consoles which connect to xterms or ports
- tunctl - Configuration tool to create and delete tap devices
- uml_net - Setuid binary for automatic tap device configuration
- uml_switch - User-space virtual switch required for daemon transport

  The uml_utilities tree is compiled with:

```
host#
make && make install
```

Note that UML kernel patches may require a specific version of the uml_utilities distribution. If you don't keep up with the mailing lists, ensure that you have the latest release of uml_utilities if you are experiencing problems with your UML kernel, particularly when dealing with consoles or command-line switches to the helper programs

2.3 3. Running UML and logging in

2.3.1 3.1. Running UML

It runs on 2.2.15 or later, and all kernel versions since 2.4.

Booting UML is straightforward. Simply run 'linux': it will try to mount the file `root_fs` in the current directory. You do not need to run it as root. If your root filesystem is not named `root_fs`, then you need to put a `ubd0=root_fs_whatever` switch on the linux command line.

You will need a filesystem to boot UML from. There are a number available for download from http://user-mode-linux.sourceforge.net. There are also several tools at http://user-mode-linux.sourceforge.net/ which can be used to generate UML-compatible filesystem images from media. The kernel will boot up and present you with a login prompt.

Note: If the host is configured with a 2G/2G address space split rather than the usual 3G/1G split, then the packaged UML binaries will not run. They will immediately segfault. See 4. UML on 2G/2G hosts for the scoop on running UML on your system.

2.3.2 3.2. Logging in

The prepackaged filesystems have a root account with password 'root' and a user account with password 'user'. The login banner will generally tell you how to log in. So, you log in and you will find yourself inside a little virtual machine. Our filesystems have a variety of commands and utilities installed (and it is fairly easy to add more), so you will have a lot of tools with which to poke around the system.

There are a couple of other ways to log in:

- On a virtual console

  Each virtual console that is configured (i.e. the device exists in /dev and /etc/inittab runs a getty on it) will come up in its own xterm. If you get tired of the xterms, read 5. Setting up serial lines and consoles to see how to attach the consoles to something else, like host ptys.

- Over the serial line

  In the boot output, find a line that looks like:

```
serial line 0 assigned pty /dev/ptyp1
```

Attach your favorite terminal program to the corresponding tty. I.e. for minicom, the command would be:

```
host% minicom -o -p /dev/ttyp1
```

- Over the net

  If the network is running, then you can telnet to the virtual machine and log in to it. See 6. Setting up the network to learn about setting up a virtual network.

When you' re done using it, run halt, and the kernel will bring itself down and the process will exit.

2.3.3 3.3. Examples

Here are some examples of UML in action:

- A login session http://user-mode-linux.sourceforge.net/old/login.html
- A virtual network http://user-mode-linux.sourceforge.net/old/net.html

2.4 4. UML on 2G/2G hosts

2.4.1 4.1. Introduction

Most Linux machines are configured so that the kernel occupies the upper 1G (0xc0000000 - 0xffffffff) of the 4G address space and processes use the lower 3G (0x00000000 - 0xbfffffff). However, some machine are configured with a 2G/2G split, with the kernel occupying the upper 2G (0x80000000 - 0xffffffff) and processes using the lower 2G (0x00000000 - 0x7fffffff).

2.4.2 4.2. The problem

The prebuilt UML binaries on this site will not run on 2G/2G hosts because UML occupies the upper .5G of the 3G process address space (0xa0000000 - 0xbfffffff). Obviously, on 2G/2G hosts, this is right in the middle of the kernel address space, so UML won' t even load - it will immediately segfault.

2.4.3 4.3. The solution

The fix for this is to rebuild UML from source after enabling CONFIG_HOST_2G_2G (under 'General Setup'). This will cause UML to load itself in the top .5G of that smaller process address space, where it will run fine. See 2. Compiling the kernel and modules if you need help building UML from source.

2.5 5. Setting up serial lines and consoles

It is possible to attach UML serial lines and consoles to many types of host I/O channels by specifying them on the command line.

You can attach them to host ptys, ttys, file descriptors, and ports. This allows you to do things like:

- have a UML console appear on an unused host console,
- hook two virtual machines together by having one attach to a pty and having the other attach to the corresponding tty
- make a virtual machine accessible from the net by attaching a console to a port on the host.

The general format of the command line option is `device=channel`.

2.5.1 5.1. Specifying the device

Devices are specified with "con" or "ssl" (console or serial line, respectively), optionally with a device number if you are talking about a specific device.

Using just "con" or "ssl" describes all of the consoles or serial lines. If you want to talk about console #3 or serial line #10, they would be "con3" and "ssl10", respectively.

A specific device name will override a less general "con=" or "ssl=". So, for example, you can assign a pty to each of the serial lines except for the first two like this:

```
ssl=pty ssl0=tty:/dev/tty0 ssl1=tty:/dev/tty1
```

The specificity of the device name is all that matters; order on the command line is irrelevant.

2.5.2 5.2. Specifying the channel

There are a number of different types of channels to attach a UML device to, each with a different way of specifying exactly what to attach to.

- pseudo-terminals - device=pty pts terminals - device=pts

  This will cause UML to allocate a free host pseudo-terminal for the device. The terminal that it got will be announced in the boot log. You access it by attaching a terminal program to the corresponding tty:

- screen /dev/pts/n
- screen /dev/ttyxx
- minicom -o -p /dev/ttyxx - minicom seems not able to handle pts devices
- kermit - start it up, 'open' the device, then 'connect'
- terminals - device=tty:tty device file

  This will make UML attach the device to the specified tty (i.e:

```
con1=tty:/dev/tty3
```

will attach UML's console 1 to the host's /dev/tty3). If the tty that you specify is the slave end of a tty/pty pair, something else must have already opened the corresponding pty in order for this to work.

- xterms - device=xterm

  UML will run an xterm and the device will be attached to it.

- Port - device=port:port number

  This will attach the UML devices to the specified host port. Attaching console 1 to the host's port 9000 would be done like this:

```
con1=port:9000
```

Attaching all the serial lines to that port would be done similarly:

```
ssl=port:9000
```

You access these devices by telnetting to that port. Each active telnet session gets a different device. If there are more telnets to a port than UML devices attached to it, then the extra telnet sessions will block until an existing telnet detaches, or until another device becomes active (i.e. by being activated in /etc/inittab).

This channel has the advantage that you can both attach multiple UML devices to it and know how to access them without reading the UML boot log. It is also unique in allowing access to a UML from remote machines without requiring that the UML be networked. This could be useful in allowing public access to UMLs because they would be accessible from the net, but wouldn't need any kind of network filtering or access control because they would have no network access.

If you attach the main console to a portal, then the UML boot will appear to hang. In reality, it's waiting for a telnet to connect, at which point the boot will proceed.

- already-existing file descriptors - device=file descriptor

  If you set up a file descriptor on the UML command line, you can attach a UML device to it. This is most commonly used to put the main console back on stdin and stdout after assigning all the other consoles to something else:

```
con0=fd:0,fd:1 con=pts
```

- Nothing - device=null

  This allows the device to be opened, in contrast to 'none', but reads will block, and writes will succeed and the data will be thrown out.

- None - device=none

  This causes the device to disappear.

You can also specify different input and output channels for a device by putting a comma between them:

```
ssl3=tty:/dev/tty2,xterm
```

will cause serial line 3 to accept input on the host's /dev/tty2 and display output on an xterm. That's a silly example - the most common use of this syntax is to reattach the main console to stdin and stdout as shown above.

If you decide to move the main console away from stdin/stdout, the initial boot output will appear in the terminal that you're running UML in. However, once the console driver has been officially initialized, then the boot output will start appearing wherever you specified that console 0 should be. That device will receive all subsequent output.

2.5.3 5.3. Examples

There are a number of interesting things you can do with this capability.

First, this is how you get rid of those bleeding console xterms by attaching them to host ptys:

```
con=pty con0=fd:0,fd:1
```

This will make a UML console take over an unused host virtual console, so that when you switch to it, you will see the UML login prompt rather than the host login prompt:

```
con1=tty:/dev/tty6
```

You can attach two virtual machines together with what amounts to a serial line as follows:

Run one UML with a serial line attached to a pty:

```
ssl1=pty
```

Look at the boot log to see what pty it got (this example will assume that it got /dev/ptyp1).

Boot the other UML with a serial line attached to the corresponding tty:

```
ssl1=tty:/dev/ttyp1
```

Log in, make sure that it has no getty on that serial line, attach a terminal program like minicom to it, and you should see the login prompt of the other virtual machine.

2.6 6. Setting up the network

This page describes how to set up the various transports and to provide a UML instance with network access to the host, other machines on the local net, and the rest of the net.

As of 2.4.5, UML networking has been completely redone to make it much easier to set up, fix bugs, and add new features.

There is a new helper, uml_net, which does the host setup that requires root privileges.

There are currently five transport types available for a UML virtual machine to exchange packets with other hosts:

- ethertap
- TUN/TAP
- Multicast
- a switch daemon
- slip
- slirp
- pcap

  The TUN/TAP, ethertap, slip, and slirp transports allow a UML instance to exchange packets with the host. They may be directed to the host or the host may just act as a router to provide access to other physical or virtual machines.

The pcap transport is a synthetic read-only interface, using the libpcap binary to collect packets from interfaces on the host and filter them. This is useful for building preconfigured traffic monitors or sniffers.

The daemon and multicast transports provide a completely virtual network to other virtual machines. This network is completely disconnected from the physical network unless one of the virtual machines on it is acting as a gateway.

With so many host transports, which one should you use? Here's when you should use each one:

- ethertap - if you want access to the host networking and it is running 2.2
- TUN/TAP - if you want access to the host networking and it is running 2.4. Also, the TUN/TAP transport is able to use a preconfigured device, allowing it to avoid using the setuid uml_net helper, which is a security advantage.
- Multicast - if you want a purely virtual network and you don't want to set up anything but the UML
- a switch daemon - if you want a purely virtual network and you don't mind running the daemon in order to get somewhat better performance
- slip - there is no particular reason to run the slip backend unless ethertap and TUN/TAP are just not available for some reason
- slirp - if you don't have root access on the host to setup networking, or if you don't want to allocate an IP to your UML
- pcap - not much use for actual network connectivity, but great for monitoring traffic on the host

  Ethertap is available on 2.4 and works fine. TUN/TAP is preferred to it because it has better performance and ethertap is officially considered obsolete in 2.4. Also, the root helper only needs to run occasionally for TUN/TAP, rather than handling every packet, as it does with ethertap. This is a slight security advantage since it provides fewer opportunities for a nasty UML user to somehow exploit the helper's root privileges.

2.6.1 6.1. General setup

First, you must have the virtual network enabled in your UML. If are running a prebuilt kernel from this site, everything is already enabled. If you build the kernel yourself, under the "Network device support" menu, enable "Network device support", and then the three transports.

The next step is to provide a network device to the virtual machine. This is done by describing it on the kernel command line.

The general format is:

```
eth <n> = <transport> , <transport args>
```

For example, a virtual ethernet device may be attached to a host ethertap device as follows:

```
eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254
```

This sets up eth0 inside the virtual machine to attach itself to the host /dev/tap0, assigns it an ethernet address, and assigns the host tap0 interface an IP address.

Note that the IP address you assign to the host end of the tap device must be different than the IP you assign to the eth device inside UML. If you are short on IPs and don't want to consume two per UML, then you can reuse the host's eth IP address for the host ends of the tap devices. Internally, the UMLs must still get unique IPs for their eth devices. You can also give the UMLs non-routable IPs (192.168.x.x or 10.x.x.x) and have the host masquerade them. This will let outgoing connections work, but incoming connections won't without more work, such as port forwarding from the host. Also note that when you configure the host side of an interface, it is only acting as a gateway. It will respond to pings sent to it locally, but is not useful to do that since it's a host interface. You are not talking to the UML when you ping that interface and get a response.

You can also add devices to a UML and remove them at runtime. See the 10. The Management Console page for details.

The sections below describe this in more detail.

Once you've decided how you're going to set up the devices, you boot UML, log in, configure the UML side of the devices, and set up routes to the outside world. At that point, you will be able to talk to any other machines, physical or virtual, on the net.

If ifconfig inside UML fails and the network refuses to come up, run tell you what went wrong.

2.6.2 6.2. Userspace daemons

You will likely need the setuid helper, or the switch daemon, or both. They are both installed with the RPM and deb, so if you've installed either, you can skip the rest of this section.

If not, then you need to check them out of CVS, build them, and install them. The helper is uml_net, in CVS /tools/uml_net, and the daemon is uml_switch, in CVS /tools/uml_router. They are both built with a plain 'make'. Both need to be installed in a directory that's in your path - /usr/bin is recommend. On top of that, uml_net needs to be setuid root.

2.6.3 6.3. Specifying ethernet addresses

Below, you will see that the TUN/TAP, ethertap, and daemon interfaces allow you to specify hardware addresses for the virtual ethernet devices. This is generally not necessary. If you don't have a specific reason to do it, you probably shouldn't. If one is not specified on the command line, the driver will assign one based on the device IP address. It will provide the address fe:fd:nn:nn:nn:nn where nn.nn.nn.nn is the device IP address. This is nearly always sufficient to guarantee a unique hardware address for the device. A couple of exceptions are:

- Another set of virtual ethernet devices are on the same network and they are assigned hardware addresses using a different scheme which may conflict with the UML IP address-based scheme
- You aren't going to use the device for IP networking, so you don't assign the device an IP address

  If you let the driver provide the hardware address, you should make sure that the device IP address is known before the interface is brought up. So, inside UML, this will guarantee that:

```
UML#
ifconfig eth0 192.168.0.250 up
```

If you decide to assign the hardware address yourself, make sure that the first byte of the address is even. Addresses with an odd first byte are broadcast addresses, which you don't want assigned to a device.

2.6.4 6.4. UML interface setup

Once the network devices have been described on the command line, you should boot UML and log in.

The first thing to do is bring the interface up:

```
UML# ifconfig ethn ip-address up
```

You should be able to ping the host at this point.

To reach the rest of the world, you should set a default route to the host:

```
UML# route add default gw host ip
```

Again, with host ip of 192.168.0.4:

```
UML# route add default gw 192.168.0.4
```

This page used to recommend setting a network route to your local net. This is wrong, because it will cause UML to try to figure out hardware addresses of the local machines by arping on the interface to the host. Since that interface is basically a single strand of ethernet with two nodes on it (UML and the host) and arp requests don't cross networks, they will fail to elicit any responses. So, what you want is for UML to just blindly throw all packets at the host and let it figure out what to do with them, which is what leaving out the network route and adding the default route does.

Note: If you can't communicate with other hosts on your physical ethernet, it's probably because of a network route that's automatically set up. If you run 'route -n' and see a route that looks like this:

```
Destination     Gateway         Genmask         Flags Metric Ref␣
↪  Use Iface
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0␣
↪  0   eth0
```

with a mask that's not 255.255.255.255, then replace it with a route to your host:

```
UML#
route del -net 192.168.0.0 dev eth0 netmask 255.255.255.0


UML#
route add -host 192.168.0.4 dev eth0
```

This, plus the default route to the host, will allow UML to exchange packets with any machine on your ethernet.

2.6.5 6.5. Multicast

The simplest way to set up a virtual network between multiple UMLs is to use the mcast transport. This was written by Harald Welte and is present in UML version 2.4.5-5um and later. Your system must have multicast enabled in the kernel and there must be a multicast-capable network device on the host. Normally, this is eth0, but if there is no ethernet card on the host, then you will likely get strange error messages when you bring the device up inside UML.

To use it, run two UMLs with:

```
eth0=mcast
```

on their command lines. Log in, configure the ethernet device in each machine with different IP addresses:

```
UML1# ifconfig eth0 192.168.0.254


UML2# ifconfig eth0 192.168.0.253
```

and they should be able to talk to each other.

The full set of command line options for this transport are:

```
ethn=mcast,ethernet address,multicast
address,multicast port,ttl
```

There is also a related point-to-point only "ucast" transport. This is useful when your network does not support multicast, and all network connections are simple point to point links.

The full set of command line options for this transport are:

```
ethn=ucast,ethernet address,remote address,listen port,remote port
```

2.6.6 6.6. TUN/TAP with the uml_net helper

TUN/TAP is the preferred mechanism on 2.4 to exchange packets with the host. The TUN/TAP backend has been in UML since 2.4.9-3um.

The easiest way to get up and running is to let the setuid uml_net helper do the host setup for you. This involves insmod-ing the tun.o module if necessary, configuring the device, and setting up IP forwarding, routing, and proxy arp. If you are new to UML networking, do this first. If you' re concerned about the security implications of the setuid helper, use it to get up and running, then read the next section to see how to have UML use a preconfigured tap device, which avoids the use of uml_net.

If you specify an IP address for the host side of the device, the uml_net helper will do all necessary setup on the host - the only requirement is that TUN/TAP be available, either built in to the host kernel or as the tun.o module.

The format of the command line switch to attach a device to a TUN/TAP device is:

```
eth <n> =tuntap,,, <IP address>
```

For example, this argument will attach the UML' s eth0 to the next available tap device and assign an ethernet address to it based on its IP address:

```
eth0=tuntap,,,192.168.0.254
```

Note that the IP address that must be used for the eth device inside UML is fixed by the routing and proxy arp that is set up on the TUN/TAP device on the host. You can use a different one, but it won't work because reply packets won' t reach the UML. This is a feature. It prevents a nasty UML user from doing things like setting the UML IP to the same as the network' s nameserver or mail server.

There are a couple potential problems with running the TUN/TAP transport on a 2.4 host kernel

- TUN/TAP seems not to work on 2.4.3 and earlier. Upgrade the host kernel or use the ethertap transport.
- With an upgraded kernel, TUN/TAP may fail with:

  ```
  File descriptor in bad state
  ```

This is due to a header mismatch between the upgraded kernel and the kernel that was originally installed on the machine. The fix is to make sure that /usr/src/linux points to the headers for the running kernel.

These were pointed out by Tim Robinson <timro at trkr dot net> in the past.

2.6.7 6.7. TUN/TAP with a preconfigured tap device

If you prefer not to have UML use uml_net (which is somewhat insecure), with UML 2.4.17-11, you can set up a TUN/TAP device beforehand. The setup needs to be done as root, but once that's done, there is no need for root assistance. Setting up the device is done as follows:

- Create the device with tunctl (available from the UML utilities tarball):

```
host#  tunctl -u uid
```

where uid is the user id or username that UML will be run as. This will tell you what device was created.

- Configure the device IP (change IP addresses and device name to suit):

```
host#  ifconfig tap0 192.168.0.254 up
```

- Set up routing and arping if desired - this is my recipe, there are other ways of doing the same thing:

```
host#
bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'

host#
route add -host 192.168.0.253 dev tap0

host#
bash -c 'echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp'

host#
arp -Ds 192.168.0.253 eth0 pub
```

Note that this must be done every time the host boots - this configuration is not stored across host reboots. So, it's probably a good idea to stick it in an rc file. An even better idea would be a little utility which reads the information from a config file and sets up devices at boot time.

- Rather than using up two IPs and ARPing for one of them, you can also provide direct access to your LAN by the UML by using a bridge:

```
host#
brctl addbr br0


host#
ifconfig eth0 0.0.0.0 promisc up


host#
ifconfig tap0 0.0.0.0 promisc up


host#
```

(continues on next page)

(continued from previous page)

```
ifconfig br0 192.168.0.1 netmask 255.255.255.0 up


host#
brctl stp br0 off


host#
brctl setfd br0 1


host#
brctl sethello br0 1


host#
brctl addif br0 eth0


host#
brctl addif br0 tap0
```

Note that 'br0' should be setup using ifconfig with the existing IP address of eth0, as eth0 no longer has its own IP.

- Also, the /dev/net/tun device must be writable by the user running UML in order for the UML to use the device that' s been configured for it. The simplest thing to do is:

```
host#  chmod 666 /dev/net/tun
```

Making it world-writable looks bad, but it seems not to be exploitable as a security hole. However, it does allow anyone to cre- ate useless tap devices (useless because they can' t configure them), which is a DOS attack. A somewhat more secure alternative would to be to create a group containing all the users who have preconfigured tap devices and chgrp /dev/net/tun to that group with mode 664 or 660.

- Once the device is set up, run UML with 'eth0=tuntap,device name' (i.e. 'eth0=tuntap,tap0') on the command line (or do it with the mconsole config command).
- Bring the eth device up in UML and you' re in business.

  If you don' t want that tap device any more, you can make it non-persistent with:

```
host#  tunctl -d tap device
```

Finally, tunctl has a -b (for brief mode) switch which causes it to output only the name of the tap device it created. This makes it suitable for capture by a script:

```
host#  TAP=`tunctl -u 1000 -b`
```

2.6.8 6.8. Ethertap

Ethertap is the general mechanism on 2.2 for userspace processes to exchange packets with the kernel.

To use this transport, you need to describe the virtual network device on the UML command line. The general format for this is:

```
eth <n> =ethertap, <device> , <ethernet address> , <tap IP␣
↪address>
```

So, the previous example:

```
eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254
```

attaches the UML eth0 device to the host /dev/tap0, assigns it the ethernet address fe:fd:0:0:0:1, and assigns the IP address 192.168.0.254 to the tap device.

The tap device is mandatory, but the others are optional. If the ethernet address is omitted, one will be assigned to it.

The presence of the tap IP address will cause the helper to run and do whatever host setup is needed to allow the virtual machine to communicate with the outside world. If you're not sure you know what you're doing, this is the way to go.

If it is absent, then you must configure the tap device and whatever arping and routing you will need on the host. However, even in this case, the uml_net helper still needs to be in your path and it must be setuid root if you're not running UML as root. This is because the tap device doesn't support SIGIO, which UML needs in order to use something as a source of input. So, the helper is used as a convenient asynchronous IO thread.

If you're using the uml_net helper, you can ignore the following host setup - uml_net will do it for you. You just need to make sure you have ethertap available, either built in to the host kernel or available as a module.

If you want to set things up yourself, you need to make sure that the appropriate /dev entry exists. If it doesn't, become root and create it as follows:

```
mknod /dev/tap <minor>  c 36  <minor>  + 16
```

For example, this is how to create /dev/tap0:

```
mknod /dev/tap0 c 36 0 + 16
```

You also need to make sure that the host kernel has ethertap support. If ethertap is enabled as a module, you apparently need to insmod ethertap once for each ethertap device you want to enable. So,:

```
host#
insmod ethertap
```

will give you the tap0 interface. To get the tap1 interface, you need to run:

```
host#
insmod ethertap unit=1 -o ethertap1
```

2.6.9 6.9. The switch daemon

Note: This is the daemon formerly known as uml_router, but which was renamed so the network weenies of the world would stop growling at me.

The switch daemon, uml_switch, provides a mechanism for creating a totally virtual network. By default, it provides no connection to the host network (but see -tap, below).

The first thing you need to do is run the daemon. Running it with no arguments will make it listen on a default pair of unix domain sockets.

If you want it to listen on a different pair of sockets, use:

```
-unix control socket data socket
```

If you want it to act as a hub rather than a switch, use:

```
-hub
```

If you want the switch to be connected to host networking (allowing the umls to get access to the outside world through the host), use:

```
-tap tap0
```

Note that the tap device must be preconfigured (see "TUN/TAP with a preconfigured tap device", above). If you're using a different tap device than tap0, specify that instead of tap0.

uml_switch can be backgrounded as follows:

```
host%
uml_switch [ options ] < /dev/null > /dev/null
```

The reason it doesn't background by default is that it listens to stdin for EOF. When it sees that, it exits.

The general format of the kernel command line switch is:

```
ethn=daemon,ethernet address,socket
type,control socket,data socket
```

You can leave off everything except the 'daemon'. You only need to specify the ethernet address if the one that will be assigned to it isn't acceptable for some reason. The rest of the arguments describe how to communicate with the daemon. You should only specify them if you told the daemon to use different sockets than the default. So, if you ran

the daemon with no arguments, running the UML on the same machine with:

```
eth0=daemon
```

will cause the eth0 driver to attach itself to the daemon correctly.

2.6.10 6.10. Slip

Slip is another, less general, mechanism for a process to communicate with the host networking. In contrast to the ethertap interface, which exchanges ethernet frames with the host and can be used to transport any higher-level protocol, it can only be used to transport IP.

The general format of the command line switch is:

```
ethn=slip,slip IP
```

The slip IP argument is the IP address that will be assigned to the host end of the slip device. If it is specified, the helper will run and will set up the host so that the virtual machine can reach it and the rest of the network.

There are some oddities with this interface that you should be aware of. You should only specify one slip device on a given virtual machine, and its name inside UML will be 'umn', not 'eth0' or whatever you specified on the command line. These problems will be fixed at some point.

2.6.11 6.11. Slirp

slirp uses an external program, usually /usr/bin/slirp, to provide IP only networking connectivity through the host. This is similar to IP masquerading with a firewall, although the translation is performed in user-space, rather than by the kernel. As slirp does not set up any interfaces on the host, or changes routing, slirp does not require root access or setuid binaries on the host.

The general format of the command line switch for slirp is:

```
ethn=slirp,ethernet address,slirp path
```

The ethernet address is optional, as UML will set up the interface with an ethernet address based upon the initial IP address of the interface. The slirp path is generally /usr/bin/slirp, although it will depend on distribution.

The slirp program can have a number of options passed to the command line and we can't add them to the UML command line, as they will be parsed incorrectly. Instead, a wrapper shell script can be written or the options inserted into the /.slirprc file. More information on all of the slirp options can be found in its man pages.

The eth0 interface on UML should be set up with the IP 10.2.0.15, although you can use anything as long as it is not used by a network you will be connecting to. The default route on UML should be set to use:

```
UML#
route add default dev eth0
```

slirp provides a number of useful IP addresses which can be used by UML, such as 10.0.2.3 which is an alias for the DNS server specified in /etc/resolv.conf on the host or the IP given in the 'dns' option for slirp.

Even with a baudrate setting higher than 115200, the slirp connection is limited to 115200. If you need it to go faster, the slirp binary needs to be compiled with FULL_BOLT defined in config.h.

2.6.12 6.12. pcap

The pcap transport is attached to a UML ethernet device on the command line or with uml_mconsole with the following syntax:

```
ethn=pcap,host interface,filter
expression,option1,option2
```

The expression and options are optional.

The interface is whatever network device on the host you want to sniff. The expression is a pcap filter expression, which is also what tcpdump uses, so if you know how to specify tcpdump filters, you will use the same expressions here. The options are up to two of 'promisc', control whether pcap puts the host interface into promiscuous mode. 'optimize' and 'nooptimize' control whether the pcap expression optimizer is used.

Example:

```
eth0=pcap,eth0,tcp

eth1=pcap,eth0,!tcp
```

will cause the UML eth0 to emit all tcp packets on the host eth0 and the UML eth1 to emit all non-tcp packets on the host eth0.

2.6.13 6.13. Setting up the host yourself

If you don't specify an address for the host side of the ethertap or slip device, UML won't do any setup on the host. So this is what is needed to get things working (the examples use a host-side IP of 192.168.0.251 and a UML-side IP of 192.168.0.250 - adjust to suit your own network):

- The device needs to be configured with its IP address. Tap devices are also configured with an mtu of 1484. Slip devices are configured with a point-to-point address pointing at the UML ip address:

```
host#  ifconfig tap0 arp mtu 1484 192.168.0.251 up


host#
ifconfig sl0 192.168.0.251 pointopoint 192.168.0.250 up
```

- If a tap device is being set up, a route is set to the UML IP:

```
UML# route add -host 192.168.0.250 gw 192.168.0.251
```

- To allow other hosts on your network to see the virtual machine, proxy arp is set up for it:

```
host#  arp -Ds 192.168.0.250 eth0 pub
```

- Finally, the host is set up to route packets:

```
host#  echo 1 > /proc/sys/net/ipv4/ip_forward
```

2.7 7. Sharing Filesystems between Virtual Machines

2.7.1 7.1. A warning

Don't attempt to share filesystems simply by booting two UMLs from the same file. That's the same thing as booting two physical machines from a shared disk. It will result in filesystem corruption.

2.7.2 7.2. Using layered block devices

The way to share a filesystem between two virtual machines is to use the copy-on-write (COW) layering capability of the ubd block driver. As of 2.4.6-2um, the driver supports layering a read-write private device over a read-only shared device. A machine's writes are stored in the private device, while reads come from either device - the private one if the requested block is valid in it, the shared one if not. Using this scheme, the majority of data which is unchanged is shared between an arbitrary number of virtual machines, each of which has a much smaller file containing the changes that it has made. With a large number of UMLs booting from a large root filesystem, this leads to a huge disk space saving. It will also help performance, since the host will be able to cache the shared data using a much smaller amount of memory, so UML disk requests will be served from the host's memory rather than its disks.

To add a copy-on-write layer to an existing block device file, simply add the name of the COW file to the appropriate ubd switch:

```
ubd0=root_fs_cow,root_fs_debian_22
```

where 'root_fs_cow' is the private COW file and 'root_fs_debian_22' is the existing shared filesystem. The COW file need not exist. If it doesn'

t, the driver will create and initialize it. Once the COW file has been initialized, it can be used on its own on the command line:

```
ubd0=root_fs_cow
```

The name of the backing file is stored in the COW file header, so it would be redundant to continue specifying it on the command line.

2.7.3 7.3. Note!

When checking the size of the COW file in order to see the gobs of space that you' re saving, make sure you use 'ls -ls' to see the actual disk consumption rather than the length of the file. The COW file is sparse, so the length will be very different from the disk usage. Here is a 'ls -l' of a COW file and backing file from one boot and shutdown:

```
host% ls -l cow.debian debian2.2
-rw-r--r--    1 jdike    jdike    492504064 Aug  6 21:16 cow.
↪debian
-rwxrw-rw-    1 jdike    jdike    537919488 Aug  6 20:42 debian2.2
```

Doesn' t look like much saved space, does it? Well, here' s 'ls -ls' :

```
host% ls -ls cow.debian debian2.2
   880 -rw-r--r--    1 jdike    jdike    492504064 Aug  6 21:16␣
↪cow.debian
525832 -rwxrw-rw-    1 jdike    jdike    537919488 Aug  6 20:42␣
↪debian2.2
```

Now, you can see that the COW file has less than a meg of disk, rather than 492 meg.

2.7.4 7.4. Another warning

Once a filesystem is being used as a readonly backing file for a COW file, do not boot directly from it or modify it in any way. Doing so will invalidate any COW files that are using it. The mtime and size of the backing file are stored in the COW file header at its creation, and they must continue to match. If they don' t, the driver will refuse to use the COW file.

If you attempt to evade this restriction by changing either the backing file or the COW header by hand, you will get a corrupted filesystem.

Among other things, this means that upgrading the distribution in a backing file and expecting that all of the COW files using it will see the upgrade will not work.

2.7.5 7.5. uml_moo : Merging a COW file with its backing file

Depending on how you use UML and COW devices, it may be advisable to merge the changes in the COW file into the backing file every once in a while.

The utility that does this is uml_moo. Its usage is:

```
host% uml_moo COW file new backing file
```

There's no need to specify the backing file since that information is already in the COW file header. If you're paranoid, boot the new merged file, and if you're happy with it, move it over the old backing file.

uml_moo creates a new backing file by default as a safety measure. It also has a destructive merge option which will merge the COW file directly into its current backing file. This is really only usable when the backing file only has one COW file associated with it. If there are multiple COWs associated with a backing file, a -d merge of one of them will invalidate all of the others. However, it is convenient if you're short of disk space, and it should also be noticeably faster than a non-destructive merge.

uml_moo is installed with the UML deb and RPM. If you didn't install UML from one of those packages, you can also get it from the UML utilities http://user-mode-linux.sourceforge.net/utilities tar file in tools/moo.

2.8 8. Creating filesystems

You may want to create and mount new UML filesystems, either because your root filesystem isn't large enough or because you want to use a filesystem other than ext2.

This was written on the occasion of reiserfs being included in the 2.4.1 kernel pool, and therefore the 2.4.1 UML, so the examples will talk about reiserfs. This information is generic, and the examples should be easy to translate to the filesystem of your choice.

2.9 8.1. Create the filesystem file

dd is your friend. All you need to do is tell dd to create an empty file of the appropriate size. I usually make it sparse to save time and to avoid allocating disk space until it's actually used. For example, the following command will create a sparse 100 meg file full of zeroes:

```
host%
dd if=/dev/zero of=new_filesystem seek=100 count=1 bs=1M
```

8.2. Assign the file to a UML device

Add an argument like the following to the UML command line:

```
ubd4=new_filesystem
```

making sure that you use an unassigned ubd device number.

8.3. Creating and mounting the filesystem

Make sure that the filesystem is available, either by being built into the kernel, or available as a module, then boot up UML and log in. If the root filesystem doesn't have the filesystem utilities (mkfs, fsck, etc), then get them into UML by way of the net or hostfs.

Make the new filesystem on the device assigned to the new file:

```
host#  mkreiserfs /dev/ubd/4


<----------- MKREISERFSv2 ----------->

ReiserFS version 3.6.25
Block size 4096 bytes
Block count 25856
Used blocks 8212
        Journal - 8192 blocks (18-8209), journal header is in␣
↪block 8210
        Bitmaps: 17
        Root block 8211
Hash function "r5"
ATTENTION: ALL DATA WILL BE LOST ON '/dev/ubd/4'! (y/n)y
journal size 8192 (from 18)
Initializing journal - 0%....20%....40%....60%....80%....100%
Syncing..done.
```

Now, mount it:

```
UML#
mount /dev/ubd/4 /mnt
```

and you're in business.

2.10 9. Host file access

If you want to access files on the host machine from inside UML, you can treat it as a separate machine and either nfs mount directories from the host or copy files into the virtual machine with scp or rcp. However, since UML is running on the host, it can access those files just like any other process and make them available inside the virtual machine without needing to use the network.

This is now possible with the hostfs virtual filesystem. With it, you can mount a host directory into the UML filesystem and access the files contained in it just as you would on the host.

2.10.1 9.1. Using hostfs

To begin with, make sure that hostfs is available inside the virtual machine with:

```
UML# cat /proc/filesystems
```

. hostfs should be listed. If it's not, either rebuild the kernel with hostfs configured into it or make sure that hostfs is built as a module and available inside the virtual machine, and insmod it.

Now all you need to do is run mount:

```
UML# mount none /mnt/host -t hostfs
```

will mount the host's / on the virtual machine's /mnt/host.

If you don't want to mount the host root directory, then you can specify a subdirectory to mount with the -o switch to mount:

```
UML# mount none /mnt/home -t hostfs -o /home
```

will mount the hosts's /home on the virtual machine's /mnt/home.

2.10.2 9.2. hostfs as the root filesystem

It's possible to boot from a directory hierarchy on the host using hostfs rather than using the standard filesystem in a file.

To start, you need that hierarchy. The easiest way is to loop mount an existing root_fs file:

```
host#  mount root_fs uml_root_dir -o loop
```

You need to change the filesystem type of / in etc/fstab to be 'hostfs', so that line looks like this:

```
/dev/ubd/0       /        hostfs      defaults          1   1
```

Then you need to chown to yourself all the files in that directory that are owned by root. This worked for me:

```
host#  find . -uid 0 -exec chown jdike {} \;
```

Next, make sure that your UML kernel has hostfs compiled in, not as a module. Then run UML with the boot device pointing at that directory:

```
ubd0=/path/to/uml/root/directory
```

UML should then boot as it does normally.

2.10.3 9.3. Building hostfs

If you need to build hostfs because it's not in your kernel, you have two choices:

- Compiling hostfs into the kernel:

  Reconfigure the kernel and set the 'Host filesystem' option under

- Compiling hostfs as a module:

  Reconfigure the kernel and set the 'Host filesystem' option under be in arch/um/fs/hostfs/hostfs.o. Install that in `/lib/modules/$(uname -r)/fs` in the virtual machine, boot it up, and:

```
UML# insmod hostfs
```

2.11 10. The Management Console

The UML management console is a low-level interface to the kernel, somewhat like the i386 SysRq interface. Since there is a full-blown operating system under UML, there is much greater flexibility possible than with the SysRq mechanism.

There are a number of things you can do with the mconsole interface:

- get the kernel version
- add and remove devices
- halt or reboot the machine
- Send SysRq commands
- Pause and resume the UML

You need the mconsole client (uml_mconsole) which is present in CVS (/tools/mconsole) in 2.4.5-9um and later, and will be in the RPM in 2.4.6.

You also need CONFIG_MCONSOLE (under 'General Setup') enabled in UML. When you boot UML, you'll see a line like:

```
mconsole initialized on /home/jdike/.uml/umlNJ32yL/mconsole
```

If you specify a unique machine id one the UML command line, i.e.:

```
umid=debian
```

you'll see this:

```
mconsole initialized on /home/jdike/.uml/debian/mconsole
```

That file is the socket that uml_mconsole will use to communicate with UML. Run it with either the umid or the full path as its argument:

```
host% uml_mconsole debian
```

or:

```
host% uml_mconsole /home/jdike/.uml/debian/mconsole
```

You'll get a prompt, at which you can run one of these commands:

- version
- halt
- reboot
- config
- remove
- sysrq
- help
- cad
- stop
- go

2.11.1 10.1. version

This takes no arguments. It prints the UML version:

```
(mconsole)  version
OK Linux usermode 2.4.5-9um #1 Wed Jun 20 22:47:08 EDT 2001 i686
```

There are a couple actual uses for this. It's a simple no-op which can be used to check that a UML is running. It's also a way of sending an interrupt to the UML. This is sometimes useful on SMP hosts, where there's a bug which causes signals to UML to be lost, often causing it to appear to hang. Sending such a UML the mconsole version command is a good way to 'wake it up' before networking has been enabled, as it does not do anything to the function of the UML.

2.11.2 10.2. halt and reboot

These take no arguments. They shut the machine down immediately, with no syncing of disks and no clean shutdown of userspace. So, they are pretty close to crashing the machine:

```
(mconsole)  halt
OK
```

2.11.3 10.3. config

"config" adds a new device to the virtual machine. Currently the ubd and network drivers support this. It takes one argument, which is the device to add, with the same syntax as the kernel command line:

```
(mconsole)
config ubd3=/home/jdike/incoming/roots/root_fs_debian22

OK
(mconsole)  config eth1=mcast
OK
```

2.11.4 10.4. remove

"remove" deletes a device from the system. Its argument is just the name of the device to be removed. The device must be idle in whatever sense the driver considers necessary. In the case of the ubd driver, the removed block device must not be mounted, swapped on, or otherwise open, and in the case of the network driver, the device must be down:

```
(mconsole)  remove ubd3
OK
(mconsole)  remove eth1
OK
```

2.11.5 10.5. sysrq

This takes one argument, which is a single letter. It calls the generic kernel' s SysRq driver, which does whatever is called for by that argument. See the SysRq documentation in Documentation/admin-guide/sysrq.rst in your favorite kernel tree to see what letters are valid and what they do.

2.11.6 10.6. help

"help" returns a string listing the valid commands and what each one does.

2.11.7 10.7. cad

This invokes the Ctl-Alt-Del action on init. What exactly this ends up doing is up to /etc/inittab. Normally, it reboots the machine. With UML, this is usually not desired, so if a halt would be better, then find the section of inittab that looks like this:

```
# What to do when CTRL-ALT-DEL is pressed.
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

and change the command to halt.

2.11.8 10.8. stop

This puts the UML in a loop reading mconsole requests until a 'go' mconsole command is received. This is very useful for making backups of UML filesystems, as the UML can be stopped, then synced via 'sysrq s', so that everything is written to the filesystem. You can then copy the filesystem and then send the UML 'go' via mconsole.

Note that a UML running with more than one CPU will have problems after you send the 'stop' command, as only one CPU will be held in a mconsole loop and all others will continue as normal. This is a bug, and will be fixed.

2.11.9 10.9. go

This resumes a UML after being paused by a 'stop' command. Note that when the UML has resumed, TCP connections may have timed out and if the UML is paused for a long period of time, crond might go a little crazy, running all the jobs it didn't do earlier.

2.12 11. Kernel debugging

Note: The interface that makes debugging, as described here, possible is present in 2.4.0-test6 kernels and later.

Since the user-mode kernel runs as a normal Linux process, it is possible to debug it with gdb almost like any other process. It is slightly different because the kernel's threads are already being ptraced for system call interception, so gdb can't ptrace them. However, a mechanism has been added to work around that problem.

In order to debug the kernel, you need build it from source. See 2. Compiling the kernel and modules for information on doing that. Make sure that you enable CONFIG_DEBUGSYM and CONFIG_PT_PROXY during the config. These will compile the kernel with `-g`, and enable the ptrace proxy so that gdb works with UML, respectively.

2.12.1 11.1. Starting the kernel under gdb

You can have the kernel running under the control of gdb from the beginning by putting 'debug' on the command line. You will get an xterm with gdb running inside it. The kernel will send some commands to gdb which will leave it stopped at the beginning of start_kernel. At this point, you can get things going with 'next', 'step', or 'cont'.

There is a transcript of a debugging session here <debug- session.html> , with breakpoints being set in the scheduler and in an interrupt handler.

2.12.2 11.2. Examining sleeping processes

Not every bug is evident in the currently running process. Sometimes, processes hang in the kernel when they shouldn't because they've deadlocked on a semaphore or something similar. In this case, when you ^C gdb and get a backtrace, you will see the idle thread, which isn't very relevant.

What you want is the stack of whatever process is sleeping when it shouldn't be. You need to figure out which process that is, which is generally fairly easy. Then you need to get its host process id, which you can do either by looking at ps on the host or at task.thread.extern_pid in gdb.

Now what you do is this:

- detach from the current thread:

```
(UML gdb)  det
```

- attach to the thread you are interested in:

```
(UML gdb)  att <host pid>
```

- look at its stack and anything else of interest:

```
(UML gdb)  bt
```

Note that you can't do anything at this point that requires that a process execute, e.g. calling a function

- when you're done looking at that process, reattach to the current thread and continue it:

```
(UML gdb)
att 1


(UML gdb)
c
```

Here, specifying any pid which is not the process id of a UML thread will cause gdb to reattach to the current thread. I commonly use 1, but any other invalid pid would work.

2.12.3 11.3. Running ddd on UML

ddd works on UML, but requires a special kludge. The process goes like this:

- Start ddd:

```
host% ddd linux
```

- With ps, get the pid of the gdb that ddd started. You can ask the gdb to tell you, but for some reason that confuses things and causes a hang.
- run UML with 'debug=parent gdb-pid=<pid>' added to the command line - it will just sit there after you hit return
- type 'att 1' to the ddd gdb and you will see something like:

```
0xa013dc51 in __kill ()


(gdb)
```

- At this point, type 'c', UML will boot up, and you can use ddd just as you do on any other process.

2.12.4 11.4. Debugging modules

gdb has support for debugging code which is dynamically loaded into the process. This support is what is needed to debug kernel modules under UML.

Using that support is somewhat complicated. You have to tell gdb what object file you just loaded into UML and where in memory it is. Then, it can read the symbol table, and figure out where all the symbols are from the load address that you provided. It gets more interesting when you load the module again (i.e. after an rmmod). You have to tell gdb to forget about all its symbols, including the main UML ones for some reason, then load then all back in again.

There's an easy way and a hard way to do this. The easy way is to use the umlgdb expect script written by Chandan Kudige. It basically automates the process for you.

First, you must tell it where your modules are. There is a list in the script that looks like this:

```
set MODULE_PATHS {
"fat" "/usr/src/uml/linux-2.4.18/fs/fat/fat.o"
"isofs" "/usr/src/uml/linux-2.4.18/fs/isofs/isofs.o"
"minix" "/usr/src/uml/linux-2.4.18/fs/minix/minix.o"
}
```

You change that to list the names and paths of the modules that you are going to debug. Then you run it from the toplevel directory of your UML pool and it basically tells you what to do:

```
            ******** GDB pid is 21903 ********
Start UML as: ./linux <kernel switches> debug gdb-pid=21903



GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 Free Software Foundation, Inc.
```

(continues on next page)

(continued from previous page)

```
GDB is free software, covered by the GNU General Public License,␣
↪and you are
welcome to change it and/or distribute copies of it under certain␣
↪conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"␣
↪for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) b sys_init_module
Breakpoint 1 at 0xa0011923: file module.c, line 349.
(gdb) att 1
```

After you run UML and it sits there doing nothing, you hit return at the 'att 1' and continue it:

```
Attaching to program: /home/jdike/linux/2.4/um/./linux, process 1
0xa00f4221 in __kill ()
(UML gdb)  c
Continuing.
```

At this point, you debug normally. When you insmod something, the expect magic will kick in and you' ll see something like:

```
 *** Module hostfs loaded ***
Breakpoint 1, sys_init_module (name_user=0x805abb0 "hostfs",
    mod_user=0x8070e00) at module.c:349
349             char *name, *n_name, *name_tmp = NULL;
(UML gdb)  finish
Run till exit from #0  sys_init_module (name_user=0x805abb0
↪"hostfs",
    mod_user=0x8070e00) at module.c:349
0xa00e2e23 in execute_syscall (r=0xa8140284) at syscall_kern.c:411
411             else res = EXECUTE_SYSCALL(syscall, regs);
Value returned is $1 = 0
(UML gdb)
p/x (int)module_list + module_list->size_of_struct

$2 = 0xa9021054
(UML gdb)  symbol-file ./linux
Load new symbol table from "./linux"? (y or n) y
Reading symbols from ./linux...
done.
(UML gdb)
add-symbol-file /home/jdike/linux/2.4/um/arch/um/fs/hostfs/hostfs.
↪o 0xa9021054

add symbol table from file "/home/jdike/linux/2.4/um/arch/um/fs/
↪hostfs/hostfs.o" at
        .text_addr = 0xa9021054
 (y or n) y

Reading symbols from /home/jdike/linux/2.4/um/arch/um/fs/hostfs/
↪hostfs.o...
done.
(UML gdb)  p *module_list
```

(continues on next page)

(continued from previous page)

```
$1 = {size_of_struct = 84, next = 0xa0178720, name = 0xa9022de0
↪"hostfs",
  size = 9016, uc = {usecount = {counter = 0}, pad = 0}, flags =␣
↪1,
  nsyms = 57, ndeps = 0, syms = 0xa9023170, deps = 0x0, refs =␣
↪0x0,
  init = 0xa90221f0 <init_hostfs>, cleanup = 0xa902222c <exit_
↪hostfs>,
  ex_table_start = 0x0, ex_table_end = 0x0, persist_start = 0x0,
  persist_end = 0x0, can_unload = 0, runsize = 0, kallsyms_start␣
↪= 0x0,
  kallsyms_end = 0x0,
  archdata_start = 0x1b855 <Address 0x1b855 out of bounds>,
  archdata_end = 0xe5890000 <Address 0xe5890000 out of bounds>,
  kernel_data = 0xf689c35d <Address 0xf689c35d out of bounds>}
>> Finished loading symbols for hostfs ...
```

That's the easy way. It's highly recommended. The hard way is described below in case you're interested in what's going on.

Boot the kernel under the debugger and load the module with insmod or modprobe. With gdb, do:

```
(UML gdb)  p module_list
```

This is a list of modules that have been loaded into the kernel, with the most recently loaded module first. Normally, the module you want is at module_list. If it's not, walk down the next links, looking at the name fields until find the module you want to debug. Take the address of that structure, and add module.size_of_struct (which in 2.4.10 kernels is 96 (0x60)) to it. Gdb can make this hard addition for you :-):

```
(UML gdb)
printf "%#x\n", (int)module_list module_list->size_of_struct
```

The offset from the module start occasionally changes (before 2.4.0, it was module.size_of_struct + 4), so it's a good idea to check the init and cleanup addresses once in a while, as describe below. Now do:

```
(UML gdb)
add-symbol-file /path/to/module/on/host that_address
```

Tell gdb you really want to do it, and you're in business.

If there's any doubt that you got the offset right, like breakpoints appear not to work, or they're appearing in the wrong place, you can check it by looking at the module structure. The init and cleanup fields should look like:

```
init = 0x588066b0 <init_hostfs>, cleanup = 0x588066c0 <exit_
↪hostfs>
```

with no offsets on the symbol names. If the names are right, but they are offset, then the offset tells you how much you need to add to the address you gave to add-symbol-file.

When you want to load in a new version of the module, you need to get gdb to forget about the old one. The only way I've found to do that is to tell gdb to forget about all symbols that it knows about:

```
(UML gdb)  symbol-file
```

Then reload the symbols from the kernel binary:

```
(UML gdb)  symbol-file /path/to/kernel
```

and repeat the process above. You'll also need to re-enable breakpoints. They were disabled when you dumped all the symbols because gdb couldn't figure out where they should go.

2.12.5 11.5. Attaching gdb to the kernel

If you don't have the kernel running under gdb, you can attach gdb to it later by sending the tracing thread a SIGUSR1. The first line of the console output identifies its pid:

```
tracing thread pid = 20093
```

When you send it the signal:

```
host% kill -USR1 20093
```

you will get an xterm with gdb running in it.

If you have the mconsole compiled into UML, then the mconsole client can be used to start gdb:

```
(mconsole)  (mconsole) config gdb=xterm
```

will fire up an xterm with gdb running in it.

2.12.6 11.6. Using alternate debuggers

UML has support for attaching to an already running debugger rather than starting gdb itself. This is present in CVS as of 17 Apr 2001. I sent it to Alan for inclusion in the ac tree, and it will be in my 2.4.4 release.

This is useful when gdb is a subprocess of some UI, such as emacs or ddd. It can also be used to run debuggers other than gdb on UML. Below is an example of using strace as an alternate debugger.

To do this, you need to get the pid of the debugger and pass it in with the

If you are using gdb under some UI, then tell it to 'att 1', and you'll find yourself attached to UML.

If you are using something other than gdb as your debugger, then you'll need to get it to do the equivalent of 'att 1' if it doesn't do it automatically.

An example of an alternate debugger is strace. You can strace the actual kernel as follows:

- Run the following in a shell:

```
host%
sh -c 'echo pid=$$; echo -n hit return; read x; exec strace -
↪p 1 -o strace.out'
```

- Run UML with ‘debug’ and ‘gdb-pid=<pid>’ with the pid printed out by the previous command
- Hit return in the shell, and UML will start running, and strace output will start accumulating in the output file.

  Note that this is different from running:

```
host% strace ./linux
```

That will strace only the main UML thread, the tracing thread, which doesn’ t do any of the actual kernel work. It just oversees the vir- tual machine. In contrast, using strace as described above will show you the low-level activity of the virtual machine.

2.13 12. Kernel debugging examples

2.13.1 12.1. The case of the hung fsck

When booting up the kernel, fsck failed, and dropped me into a shell to fix things up. I ran fsck -y, which hung:

```
Setting hostname uml                       [ OK ]
Checking root filesystem
/dev/fhd0 was not cleanly unmounted, check forced.
Error reading block 86894 (Attempt to read block from filesystem␣
↪resulted in short read) while reading indirect blocks of inode␣
↪19780.

/dev/fhd0: UNEXPECTED INCONSISTENCY; RUN fsck MANUALLY.
        (i.e., without -a or -p options)
[ FAILED ]

*** An error occurred during the file system check.
*** Dropping you to a shell; the system will reboot
*** when you leave the shell.
Give root password for maintenance
(or type Control-D for normal startup):

[root@uml /root]# fsck -y /dev/fhd0
fsck -y /dev/fhd0
Parallelizing fsck version 1.14 (9-Jan-1999)
e2fsck 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
/dev/fhd0 contains a file system with errors, check forced.
Pass 1: Checking inodes, blocks, and sizes
```

(continues on next page)

(continued from previous page)

```
Error reading block 86894 (Attempt to read block from filesystem␣
↪resulted in short read) while reading indirect blocks of inode␣
↪19780.  Ignore error? yes

Inode 19780, i_blocks is 1548, should be 540.  Fix? yes

Pass 2: Checking directory structure
Error reading block 49405 (Attempt to read block from filesystem␣
↪resulted in short read).  Ignore error? yes

Directory inode 11858, block 0, offset 0: directory corrupted
Salvage? yes

Missing '.' in directory inode 11858.
Fix? yes

Missing '..' in directory inode 11858.
Fix? yes
```

The standard drill in this sort of situation is to fire up gdb on the signal thread, which, in this case, was pid 1935. In another window, I run gdb and attach pid 1935:

```
~/linux/2.3.26/um 1016: gdb linux
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,␣
↪and you are
welcome to change it and/or distribute copies of it under certain␣
↪conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"␣
↪for details.
This GDB was configured as "i386-redhat-linux"...

(gdb) att 1935
Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 1935
0x100756d9 in __wait4 ()
```

Let's see what's currently running:

```
(gdb) p current_task.pid
$1 = 0
```

It's the idle thread, which means that fsck went to sleep for some reason and never woke up.

Let's guess that the last process in the process list is fsck:

```
(gdb) p current_task.prev_task.comm
$13 = "fsck.ext2\000\000\000\000\000\000"
```

It is, so let's see what it thinks it's up to:

```
(gdb) p current_task.prev_task.thread
$14 = {extern_pid = 1980, tracing = 0, want_tracing = 0, forking␣
↪= 0,
  kernel_stack_page = 0, signal_stack = 1342627840, syscall = {id␣
↪= 4, args = {
      3, 134973440, 1024, 0, 1024}, have_result = 0, result =␣
↪50590720},
  request = {op = 2, u = {exec = {ip = 1350467584, sp =␣
↪2952789424}, fork = {
        regs = {1350467584, 2952789424, 0 <repeats 15 times>},␣
↪sigstack = 0,
        pid = 0}, switch_to = 0x507e8000, thread = {proc =␣
↪0x507e8000,
        arg = 0xaffffdb0, flags = 0, new_pid = 0}, input_request␣
↪= {
        op = 1350467584, fd = -1342177872, proc = 0, pid = 0}}}}
```

The interesting things here are the fact that its .thread.syscall.id is __NR_write (see the big switch in arch/um/kernel/syscall_kern.c or the defines in include/asm-um/arch/unistd.h), and that it never returned. Also, its .request.op is OP_SWITCH (see arch/um/include/user_util.h). These mean that it went into a write, and, for some reason, called schedule().

The fact that it never returned from write means that its stack should be fairly interesting. Its pid is 1980 (.thread.extern_pid). That process is being ptraced by the signal thread, so it must be detached before gdb can attach it:

```
(gdb) call detach(1980)

Program received signal SIGSEGV, Segmentation fault.
<function called from gdb>
The program being debugged stopped while in a function called␣
↪from GDB.
When the function (detach) is done executing, GDB will silently
stop (instead of continuing to evaluate the expression containing
the function call).
(gdb) call detach(1980)
$15 = 0
```

The first detach segfaults for some reason, and the second one succeeds.

Now I detach from the signal thread, attach to the fsck thread, and look at its stack:

```
(gdb) det
Detaching from program: /home/dike/linux/2.3.26/um/linux Pid 1935
(gdb) att 1980
Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 1980
0x10070451 in __kill ()
(gdb) bt
#0  0x10070451 in __kill ()
#1  0x10068ccd in usr1_pid (pid=1980) at process.c:30
#2  0x1006a03f in _switch_to (prev=0x50072000, next=0x507e8000)
    at process_kern.c:156
```

(continues on next page)

(continued from previous page)

```
#3  0x1006a052 in switch_to (prev=0x50072000, next=0x507e8000,␣
↪last=0x50072000)
    at process_kern.c:161
#4  0x10001d12 in schedule () at core.c:777
#5  0x1006a744 in __down (sem=0x507d241c) at semaphore.c:71
#6  0x1006aa10 in __down_failed () at semaphore.c:157
#7  0x1006c5d8 in segv_handler (sc=0x5006e940) at trap_user.c:174
#8  0x1006c5ec in kern_segv_handler (sig=11) at trap_user.c:182
#9  <signal handler called>
#10 0x10155404 in errno ()
#11 0x1006c0aa in segv (address=1342179328, is_write=2) at trap_
↪kern.c:50
#12 0x1006c5d8 in segv_handler (sc=0x5006eaf8) at trap_user.c:174
#13 0x1006c5ec in kern_segv_handler (sig=11) at trap_user.c:182
#14 <signal handler called>
#15 0xc0fd in ?? ()
#16 0x10016647 in sys_write (fd=3,
    buf=0x80b8800 <Address 0x80b8800 out of bounds>, count=1024)
    at read_write.c:159
#17 0x1006d5b3 in execute_syscall (syscall=4, args=0x5006ef08)
    at syscall_kern.c:254
#18 0x1006af87 in really_do_syscall (sig=12) at syscall_user.c:35
#19 <signal handler called>
#20 0x400dc8b0 in ?? ()
```

The interesting things here are:

- There are two segfaults on this stack (frames 9 and 14)
- The first faulting address (frame 11) is 0x50000800:

```
(gdb) p (void *)1342179328
$16 = (void *) 0x50000800
```

The initial faulting address is interesting because it is on the idle thread's stack. I had been seeing the idle thread segfault for no apparent reason, and the cause looked like stack corruption. In hopes of catching the culprit in the act, I had turned off all protections to that stack while the idle thread wasn't running. This apparently tripped that trap.

However, the more immediate problem is that second segfault and I'm going to concentrate on that. First, I want to see where the fault happened, so I have to go look at the sigcontent struct in frame 8:

```
(gdb) up
#1  0x10068ccd in usr1_pid (pid=1980) at process.c:30
30          kill(pid, SIGUSR1);
(gdb)
#2  0x1006a03f in _switch_to (prev=0x50072000, next=0x507e8000)
    at process_kern.c:156
156         usr1_pid(getpid());
(gdb)
#3  0x1006a052 in switch_to (prev=0x50072000, next=0x507e8000,␣
↪last=0x50072000)
    at process_kern.c:161
161         _switch_to(prev, next);
```

(continues on next page)

(continued from previous page)

```
(gdb)
#4  0x10001d12 in schedule () at core.c:777
777             switch_to(prev, next, prev);
(gdb)
#5  0x1006a744 in __down (sem=0x507d241c) at semaphore.c:71
71                      schedule();
(gdb)
#6  0x1006aa10 in __down_failed () at semaphore.c:157
157     }
(gdb)
#7  0x1006c5d8 in segv_handler (sc=0x5006e940) at trap_user.c:174
174       segv(sc->cr2, sc->err & 2);
(gdb)
#8  0x1006c5ec in kern_segv_handler (sig=11) at trap_user.c:182
182       segv_handler(sc);
(gdb) p *sc
Cannot access memory at address 0x0.
```

That's not very useful, so I'll try a more manual method:

```
(gdb) p *((struct sigcontext *) (&sig + 1))
$19 = {gs = 0, __gsh = 0, fs = 0, __fsh = 0, es = 43, __esh = 0,␣
↪ds = 43,
  __dsh = 0, edi = 1342179328, esi = 1350378548, ebp = 1342630440,
  esp = 1342630420, ebx = 1348150624, edx = 1280, ecx = 0, eax =␣
↪0,
  trapno = 14, err = 4, eip = 268480945, cs = 35, __csh = 0,␣
↪eflags = 66118,
  esp_at_signal = 1342630420, ss = 43, __ssh = 0, fpstate = 0x0,␣
↪oldmask = 0,
  cr2 = 1280}
```

The ip is in handle_mm_fault:

```
(gdb) p (void *)268480945
$20 = (void *) 0x1000b1b1
(gdb) i sym $20
handle_mm_fault + 57 in section .text
```

Specifically, it's in pte_alloc:

```
(gdb) i line *$20
Line 124 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
   starts at address 0x1000b1b1 <handle_mm_fault+57>
   and ends at 0x1000b1b7 <handle_mm_fault+63>.
```

To find where in handle_mm_fault this is, I'll jump forward in the code until I see an address in that procedure:

```
(gdb) i line *0x1000b1c0
Line 126 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
   starts at address 0x1000b1b7 <handle_mm_fault+63>
   and ends at 0x1000b1c3 <handle_mm_fault+75>.
(gdb) i line *0x1000b1d0
Line 131 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
```

(continues on next page)

(continued from previous page)

```
   starts at address 0x1000b1d0 <handle_mm_fault+88>
   and ends at 0x1000b1da <handle_mm_fault+98>.
(gdb) i line *0x1000b1e0
Line 61 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
   starts at address 0x1000b1da <handle_mm_fault+98>
   and ends at 0x1000b1e1 <handle_mm_fault+105>.
(gdb) i line *0x1000b1f0
Line 134 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
   starts at address 0x1000b1f0 <handle_mm_fault+120>
   and ends at 0x1000b200 <handle_mm_fault+136>.
(gdb) i line *0x1000b200
Line 135 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
   starts at address 0x1000b200 <handle_mm_fault+136>
   and ends at 0x1000b208 <handle_mm_fault+144>.
(gdb) i line *0x1000b210
Line 139 of "/home/dike/linux/2.3.26/um/include/asm/pgalloc.h"
   starts at address 0x1000b210 <handle_mm_fault+152>
   and ends at 0x1000b219 <handle_mm_fault+161>.
(gdb) i line *0x1000b220
Line 1168 of "memory.c" starts at address 0x1000b21e <handle_mm_
↪fault+166>
   and ends at 0x1000b222 <handle_mm_fault+170>.
```

Something is apparently wrong with the page tables or vma_structs, so lets go back to frame 11 and have a look at them:

```
#11 0x1006c0aa in segv (address=1342179328, is_write=2) at trap_
↪kern.c:50
50        handle_mm_fault(current, vma, address, is_write);
(gdb) call pgd_offset_proc(vma->vm_mm, address)
$22 = (pgd_t *) 0x80a548c
```

That's pretty bogus. Page tables aren't supposed to be in process text or data areas. Let's see what's in the vma:

```
(gdb) p *vma
$23 = {vm_mm = 0x507d2434, vm_start = 0, vm_end = 134512640,
  vm_next = 0x80a4f8c, vm_page_prot = {pgprot = 0}, vm_flags =␣
↪31200,
  vm_avl_height = 2058, vm_avl_left = 0x80a8c94, vm_avl_right =␣
↪0x80d1000,
  vm_next_share = 0xaffffdb0, vm_pprev_share = 0xaffffe63,
  vm_ops = 0xaffffe7a, vm_pgoff = 2952789626, vm_file =␣
↪0xaffffffec,
  vm_private_data = 0x62}
(gdb) p *vma.vm_mm
$24 = {mmap = 0x507d2434, mmap_avl = 0x0, mmap_cache = 0x8048000,
  pgd = 0x80a4f8c, mm_users = {counter = 0}, mm_count = {counter␣
↪= 134904288},
  map_count = 134909076, mmap_sem = {count = {counter = 135073792}
↪,
    sleepers = -1342177872, wait = {lock = <optimized out or zero␣
↪length>,
      task_list = {next = 0xaffffe63, prev = 0xaffffe7a},
      __magic = -1342177670, __creator = -1342177300}, __magic =␣
↪98},
```

(continues on next page)

(continued from previous page)

```
  page_table_lock = {}, context = 138, start_code = 0, end_code =␣
↪0,
  start_data = 0, end_data = 0, start_brk = 0, brk = 0, start_
↪stack = 0,
  arg_start = 0, arg_end = 0, env_start = 0, env_end = 0, rss =␣
↪1350381536,
  total_vm = 0, locked_vm = 0, def_flags = 0, cpu_vm_mask = 0,␣
↪swap_cnt = 0,
  swap_address = 0, segments = 0x0}
```

This also pretty bogus. With all of the 0x80xxxxx and 0xaffffxxx addresses, this is looking like a stack was plonked down on top of these structures. Maybe it's a stack overflow from the next page:

```
(gdb) p vma
$25 = (struct vm_area_struct *) 0x507d2434
```

That's towards the lower quarter of the page, so that would have to have been pretty heavy stack overflow:

```
(gdb) x/100x $25
0x507d2434:     0x507d2434      0x00000000      0x08048000      ␣
↪0x080a4f8c
0x507d2444:     0x00000000      0x080a79e0      0x080a8c94      ␣
↪0x080d1000
0x507d2454:     0xaffffdb0      0xaffffe63      0xaffffe7a      ␣
↪0xaffffe7a
0x507d2464:     0xaffffffec     0x00000062      0x0000008a      ␣
↪0x00000000
0x507d2474:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2484:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2494:     0x00000000      0x00000000      0x507d2fe0      ␣
↪0x00000000
0x507d24a4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d24b4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d24c4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d24d4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d24e4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d24f4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2504:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2514:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2524:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2534:     0x00000000      0x00000000      0x507d25dc      ␣
↪0x00000000
```

(continues on next page)

(continued from previous page)

```
0x507d2544:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2554:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2564:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2574:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2584:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d2594:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d25a4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
0x507d25b4:     0x00000000      0x00000000      0x00000000      ␣
↪0x00000000
```

It's not stack overflow. The only "stack-like" piece of this data is the vma_struct itself.

At this point, I don't see any avenues to pursue, so I just have to admit that I have no idea what's going on. What I will do, though, is stick a trap on the segfault handler which will stop if it sees any writes to the idle thread's stack. That was the thing that happened first, and it may be that if I can catch it immediately, what's going on will be somewhat clearer.

2.13.2 12.2. Episode 2: The case of the hung fsck

After setting a trap in the SEGV handler for accesses to the signal thread's stack, I reran the kernel.

fsck hung again, this time by hitting the trap:

```
Setting hostname uml                                    [ OK ]
Checking root filesystem
/dev/fhd0 contains a file system with errors, check forced.
Error reading block 86894 (Attempt to read block from filesystem␣
↪resulted in short read) while reading indirect blocks of inode␣
↪19780.

/dev/fhd0: UNEXPECTED INCONSISTENCY; RUN fsck MANUALLY.
        (i.e., without -a or -p options)
[ FAILED ]

*** An error occurred during the file system check.
*** Dropping you to a shell; the system will reboot
*** when you leave the shell.
Give root password for maintenance
(or type Control-D for normal startup):

[root@uml /root]# fsck -y /dev/fhd0
fsck -y /dev/fhd0
Parallelizing fsck version 1.14 (9-Jan-1999)
```

(continues on next page)

(continued from previous page)

```
e2fsck 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
/dev/fhd0 contains a file system with errors, check forced.
Pass 1: Checking inodes, blocks, and sizes
Error reading block 86894 (Attempt to read block from filesystem␣
↪resulted in short read) while reading indirect blocks of inode␣
↪19780.  Ignore error? yes

Pass 2: Checking directory structure
Error reading block 49405 (Attempt to read block from filesystem␣
↪resulted in short read).  Ignore error? yes

Directory inode 11858, block 0, offset 0: directory corrupted
Salvage? yes

Missing '.' in directory inode 11858.
Fix? yes

Missing '..' in directory inode 11858.
Fix? yes

Untested (4127) [100fe44c]: trap_kern.c line 31
```

I need to get the signal thread to detach from pid 4127 so that I can attach to it with gdb. This is done by sending it a SIGUSR1, which is caught by the signal thread, which detaches the process:

```
kill -USR1 4127
```

Now I can run gdb on it:

```
~/linux/2.3.26/um 1034: gdb linux
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,␣
↪and you are
welcome to change it and/or distribute copies of it under certain␣
↪conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"␣
↪for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) att 4127
Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 4127
0x10075891 in __libc_nanosleep ()
```

The backtrace shows that it was in a write and that the fault address (address in frame 3) is 0x50000800, which is right in the middle of the signal thread's stack page:

```
(gdb) bt
#0  0x10075891 in __libc_nanosleep ()
#1  0x1007584d in __sleep (seconds=1000000)
    at ../sysdeps/unix/sysv/linux/sleep.c:78
#2  0x1006ce9a in stop () at user_util.c:191
#3  0x1006bf88 in segv (address=1342179328, is_write=2) at trap_
↪kern.c:31
```

(continues on next page)

(continued from previous page)

```
#4  0x1006c628 in segv_handler (sc=0x5006eaf8) at trap_user.c:174
#5  0x1006c63c in kern_segv_handler (sig=11) at trap_user.c:182
#6  <signal handler called>
#7  0xc0fd in ?? ()
#8  0x10016647 in sys_write (fd=3, buf=0x80b8800 "R.", count=1024)
    at read_write.c:159
#9  0x1006d603 in execute_syscall (syscall=4, args=0x5006ef08)
    at syscall_kern.c:254
#10 0x1006af87 in really_do_syscall (sig=12) at syscall_user.c:35
#11 <signal handler called>
#12 0x400dc8b0 in ?? ()
#13 <signal handler called>
#14 0x400dc8b0 in ?? ()
#15 0x80545fd in ?? ()
#16 0x804daae in ?? ()
#17 0x8054334 in ?? ()
#18 0x804d23e in ?? ()
#19 0x8049632 in ?? ()
#20 0x80491d2 in ?? ()
#21 0x80596b5 in ?? ()
(gdb) p (void *)1342179328
$3 = (void *) 0x50000800
```

Going up the stack to the segv_handler frame and looking at where in the code the access happened shows that it happened near line 110 of block_dev.c:

```
(gdb) up
#1  0x1007584d in __sleep (seconds=1000000)
    at ../sysdeps/unix/sysv/linux/sleep.c:78
../sysdeps/unix/sysv/linux/sleep.c:78: No such file or directory.
(gdb)
#2  0x1006ce9a in stop () at user_util.c:191
191       while(1) sleep(1000000);
(gdb)
#3  0x1006bf88 in segv (address=1342179328, is_write=2) at trap_
↪kern.c:31
31          KERN_UNTESTED();
(gdb)
#4  0x1006c628 in segv_handler (sc=0x5006eaf8) at trap_user.c:174
174       segv(sc->cr2, sc->err & 2);
(gdb) p *sc
$1 = {gs = 0, __gsh = 0, fs = 0, __fsh = 0, es = 43, __esh = 0,␣
↪ds = 43,
    __dsh = 0, edi = 1342179328, esi = 134973440, ebp =␣
↪1342631484,
    esp = 1342630864, ebx = 256, edx = 0, ecx = 256, eax = 1024,␣
↪trapno = 14,
    err = 6, eip = 268550834, cs = 35, __csh = 0, eflags = 66070,
    esp_at_signal = 1342630864, ss = 43, __ssh = 0, fpstate = 0x0,
↪ oldmask = 0,
    cr2 = 1342179328}
(gdb) p (void *)268550834
$2 = (void *) 0x1001c2b2
(gdb) i sym $2
```

(continues on next page)

(continued from previous page)

```
block_write + 1090 in section .text
(gdb) i line *$2
Line 209 of "/home/dike/linux/2.3.26/um/include/asm/arch/string.h"
    starts at address 0x1001c2a1 <block_write+1073>
    and ends at 0x1001c2bf <block_write+1103>.
(gdb) i line *0x1001c2c0
Line 110 of "block_dev.c" starts at address 0x1001c2bf <block_
↪write+1103>
    and ends at 0x1001c2e3 <block_write+1139>.
```

Looking at the source shows that the fault happened during a call to copy_from_user to copy the data into the kernel:

```
107             count -= chars;
108             copy_from_user(p,buf,chars);
109             p += chars;
110             buf += chars;
```

p is the pointer which must contain 0x50000800, since buf contains 0x80b8800 (frame 8 above). It is defined as:

```
p = offset + bh->b_data;
```

I need to figure out what bh is, and it just so happens that bh is passed as an argument to mark_buffer_uptodate and mark_buffer_dirty a few lines later, so I do a little disassembly:

```
(gdb) disas 0x1001c2bf 0x1001c2e0
Dump of assembler code from 0x1001c2bf to 0x1001c2d0:
0x1001c2bf <block_write+1103>:  addl   %eax,0xc(%ebp)
0x1001c2c2 <block_write+1106>:  movl   0xffffffd4(%ebp),%edx
0x1001c2c8 <block_write+1112>:  btsl   $0x0,0x18(%edx)
0x1001c2cd <block_write+1117>:  btsl   $0x1,0x18(%edx)
0x1001c2d2 <block_write+1122>:  sbbl   %ecx,%ecx
0x1001c2d4 <block_write+1124>:  testl  %ecx,%ecx
0x1001c2d6 <block_write+1126>:  jne    0x1001c2e3 <block_
↪write+1139>
0x1001c2d8 <block_write+1128>:  pushl  $0x0
0x1001c2da <block_write+1130>:  pushl  %edx
0x1001c2db <block_write+1131>:  call   0x1001819c <__mark_buffer_
↪dirty>
End of assembler dump.
```

At that point, bh is in %edx (address 0x1001c2da), which is calculated at 0x1001c2c2 as %ebp + 0xffffffd4, so I figure exactly what that is, taking %ebp from the sigcontext_struct above:

```
(gdb) p (void *)1342631484
$5 = (void *) 0x5006ee3c
(gdb) p 0x5006ee3c+0xffffffd4
$6 = 1342630928
(gdb) p (void *)$6
$7 = (void *) 0x5006ec10
(gdb) p *((void **)$7)
$8 = (void *) 0x50100200
```

Now, I look at the structure to see what's in it, and particularly, what its b_data field contains:

```
(gdb) p *((struct buffer_head *)0x50100200)
$13 = {b_next = 0x50289380, b_blocknr = 49405, b_size = 1024, b_
↪list = 0,
  b_dev = 15872, b_count = {counter = 1}, b_rdev = 15872, b_state␣
↪= 24,
  b_flushtime = 0, b_next_free = 0x501001a0, b_prev_free =␣
↪0x50100260,
  b_this_page = 0x501001a0, b_reqnext = 0x0, b_pprev = 0x507fcf58,
  b_data = 0x50000800 "", b_page = 0x50004000,
  b_end_io = 0x10017f60 <end_buffer_io_sync>, b_dev_id = 0x0,
  b_rsector = 98810, b_wait = {lock = <optimized out or zero␣
↪length>,
    task_list = {next = 0x50100248, prev = 0x50100248}, __magic =␣
↪1343226448,
    __creator = 0}, b_kiobuf = 0x0}
```

The b_data field is indeed 0x50000800, so the question becomes how that happened. The rest of the structure looks fine, so this probably is not a case of data corruption. It happened on purpose somehow.

The b_page field is a pointer to the page_struct representing the 0x50000000 page. Looking at it shows the kernel's idea of the state of that page:

```
(gdb) p *$13.b_page
$17 = {list = {next = 0x50004a5c, prev = 0x100c5174}, mapping =␣
↪0x0,
    index = 0, next_hash = 0x0, count = {counter = 1}, flags =␣
↪132, lru = {
    next = 0x50008460, prev = 0x50019350}, wait = {
    lock = <optimized out or zero length>, task_list = {next =␣
↪0x50004024,
        prev = 0x50004024}, __magic = 1342193708, __creator = 0},
    pprev_hash = 0x0, buffers = 0x501002c0, virtual = 1342177280,
    zone = 0x100c5160}
```

Some sanity-checking: the virtual field shows the "virtual" address of this page, which in this kernel is the same as its "physical" address, and the page_struct itself should be mem_map[0], since it represents the first page of memory:

```
(gdb) p (void *)1342177280
$18 = (void *) 0x50000000
(gdb) p mem_map
$19 = (mem_map_t *) 0x50004000
```

These check out fine.

Now to check out the page_struct itself. In particular, the flags field shows whether the page is considered free or not:

```
(gdb) p (void *)132
$21 = (void *) 0x84
```

The “reserved” bit is the high bit, which is definitely not set, so the kernel considers the signal stack page to be free and available to be used.

At this point, I jump to conclusions and start looking at my early boot code, because that’s where that page is supposed to be reserved.

In my setup_arch procedure, I have the following code which looks just fine:

```
bootmap_size = init_bootmem(start_pfn, end_pfn - start_pfn);
free_bootmem(__pa(low_physmem) + bootmap_size, high_physmem - low_
↪physmem);
```

Two stack pages have already been allocated, and low_physmem points to the third page, which is the beginning of free memory. The init_bootmem call declares the entire memory to the boot memory manager, which marks it all reserved. The free_bootmem call frees up all of it, except for the first two pages. This looks correct to me.

So, I decide to see init_bootmem run and make sure that it is marking those first two pages as reserved. I never get that far.

Stepping into init_bootmem, and looking at bootmem_map before looking at what it contains shows the following:

```
(gdb) p bootmem_map
$3 = (void *) 0x50000000
```

Aha! The light dawns. That first page is doing double duty as a stack and as the boot memory map. The last thing that the boot memory manager does is to free the pages used by its memory map, so this page is getting freed even its marked as reserved.

The fix was to initialize the boot memory manager before allocating those two stack pages, and then allocate them through the boot memory manager. After doing this, and fixing a couple of subsequent buglets, the stack corruption problem disappeared.

2.14 13. What to do when UML doesn’t work

2.14.1 13.1. Strange compilation errors when you build from source

As of test11, it is necessary to have “ARCH=um” in the environment or on the make command line for all steps in building UML, including clean, distclean, or mrproper, config, menuconfig, or xconfig, dep, and linux. If you forget for any of them, the i386 build seems to contaminate the UML build. If this happens, start from scratch with:

```
host%
make mrproper ARCH=um
```

and repeat the build process with ARCH=um on all the steps.

See 2. Compiling the kernel and modules for more details.

Another cause of strange compilation errors is building UML in /usr/src/linux. If you do this, the first thing you need to do is clean up the mess you made. The /usr/src/linux/asm link will now point to /usr/src/linux/asm-um. Make it point back to /usr/src/linux/asm-i386. Then, move your UML pool someplace else and build it there. Also see below, where a more specific set of symptoms is described.

2.14.2 13.3. A variety of panics and hangs with /tmp on a reiserfs filesystem

I saw this on reiserfs 3.5.21 and it seems to be fixed in 3.5.27. Panics preceded by:

```
Detaching pid nnnn
```

are diagnostic of this problem. This is a reiserfs bug which causes a thread to occasionally read stale data from a mmapped page shared with another thread. The fix is to upgrade the filesystem or to have /tmp be an ext2 filesystem.

13.4. The compile fails with errors about conflicting types for 'open', 'dup', and 'waitpid'

This happens when you build in /usr/src/linux. The UML build makes the include/asm link point to include/asm-um. /usr/include/asm points to /usr/src/linux/include/asm, so when that link gets moved, files which need to include the asm-i386 versions of headers get the incompatible asm-um versions. The fix is to move the include/asm link back to include/asm-i386 and to do UML builds someplace else.

2.14.3 13.5. UML doesn't work when /tmp is an NFS filesystem

This seems to be a similar situation with the ReiserFS problem above. Some versions of NFS seems not to handle mmap correctly, which UML depends on. The workaround is have /tmp be a non-NFS directory.

2.14.4 13.6. UML hangs on boot when compiled with gprof support

If you build UML with gprof support and, early in the boot, it does this:

```
kernel BUG at page_alloc.c:100!
```

you have a buggy gcc. You can work around the problem by removing UM_FASTCALL from CFLAGS in arch/um/Makefile-i386. This will open up another bug, but that one is fairly hard to reproduce.

2.14.5 13.7. syslogd dies with a SIGTERM on startup

The exact boot error depends on the distribution that you're booting, but Debian produces this:

```
/etc/rc2.d/S10sysklogd: line 49:    93 Terminated
start-stop-daemon --start --quiet --exec /sbin/syslogd -- $SYSLOGD
```

This is a syslogd bug. There's a race between a parent process installing a signal handler and its child sending the signal.

2.14.6 13.8. TUN/TAP networking doesn't work on a 2.4 host

There are a couple of problems which were reported by Tim Robinson <timro at trkr dot net>

- It doesn't work on hosts running 2.4.7 (or thereabouts) or earlier. The fix is to upgrade to something more recent and then read the next item.
- If you see:

  ```
  File descriptor in bad state
  ```

when you bring up the device inside UML, you have a header mismatch between the original kernel and the upgraded one. Make /usr/src/linux point at the new headers. This will only be a problem if you build uml_net yourself.

2.15 13.9. You can network to the host but not to other machines on the net

If you can connect to the host, and the host can connect to UML, but you cannot connect to any other machines, then you may need to enable IP Masquerading on the host. Usually this is only experienced when using private IP addresses (192.168.x.x or 10.x.x.x) for host/UML networking, rather than the public address space that your host is connected to. UML does not enable IP Masquerading, so you will need to create a static rule to enable it:

```
host%
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Replace eth0 with the interface that you use to talk to the rest of the world.

Documentation on IP Masquerading, and SNAT, can be found at http://www.netfilter.org.

If you can reach the local net, but not the outside Internet, then that is usually a routing problem. The UML needs a default route:

```
UML#
route add default gw gateway IP
```

The gateway IP can be any machine on the local net that knows how to reach the outside world. Usually, this is the host or the local net- work's gateway.

Occasionally, we hear from someone who can reach some machines, but not others on the same net, or who can reach some ports on other machines, but not others. These are usually caused by strange firewalling somewhere between the UML and the other box. You track this down by running tcpdump on every interface the packets travel over and see where they disappear. When you find a machine that takes the packets in, but does not send them onward, that's the culprit.

2.16 13.10. I have no root and I want to scream

Thanks to Birgit Wahlich for telling me about this strange one. It turns out that there's a limit of six environment variables on the kernel command line. When that limit is reached or exceeded, argument processing stops, which means that the 'root=' argument that UML usually adds is not seen. So, the filesystem has no idea what the root device is, so it panics.

The fix is to put less stuff on the command line. Glomming all your setup variables into one is probably the best way to go.

2.17 13.11. UML build conflict between ptrace.h and ucontext.h

On some older systems, /usr/include/asm/ptrace.h and /usr/include/sys/ucontext.h define the same names. So, when they're included together, the defines from one completely mess up the parsing of the other, producing errors like:

```
/usr/include/sys/ucontext.h:47: parse error before
`10`
```

plus a pile of warnings.

This is a libc botch, which has since been fixed, and I don't see any way around it besides upgrading.

2.17.1 13.12. The UML BogoMips is exactly half the host's BogoMips

On i386 kernels, there are two ways of running the loop that is used to calculate the BogoMips rating, using the TSC if it' s there or using a one-instruction loop. The TSC produces twice the BogoMips as the loop. UML uses the loop, since it has nothing resembling a TSC, and will get almost exactly the same BogoMips as a host using the loop. However, on a host with a TSC, its BogoMips will be double the loop BogoMips, and therefore double the UML BogoMips.

2.17.2 13.13. When you run UML, it immediately segfaults

If the host is configured with the 2G/2G address space split, that' s why. See ref:UML_on_2G/2G_hosts for the details on getting UML to run on your host.

2.17.3 13.14. xterms appear, then immediately disappear

If you' re running an up to date kernel with an old release of uml_utilities, the port-helper program will not work properly, so xterms will exit straight after they appear. The solution is to upgrade to the latest release of uml_utilities. Usually this problem occurs when you have installed a packaged release of UML then compiled your own development kernel without upgrading the uml_utilities from the source distribution.

2.17.4 13.15. Any other panic, hang, or strange behavior

If you' re seeing truly strange behavior, such as hangs or panics that happen in random places, or you try running the debugger to see what' s happening and it acts strangely, then it could be a problem in the host kernel. If you' re not running a stock Linus or -ac kernel, then try that. An early version of the preemption patch and a 2.4.10 SuSE kernel have caused very strange problems in UML.

Otherwise, let me know about it. Send a message to one of the UML mailing lists - either the developer list - user-mode-linux-devel at lists dot sourceforge dot net (subscription info) or the user list - user-mode-linux-user at lists dot sourceforge do net (subscription info), whichever you prefer. Don' t assume that everyone knows about it and that a fix is imminent.

If you want to be super-helpful, read 14. Diagnosing Problems and follow the instructions contained therein.

2.18 14. Diagnosing Problems

If you get UML to crash, hang, or otherwise misbehave, you should report this on one of the project mailing lists, either the developer list - user-mode-linux-devel at lists dot sourceforge dot net (subscription info) or the user list - user-mode-linux-user at lists dot sourceforge dot net (subscription info). When you do, it is likely that I will want more information. So, it would be helpful to read the stuff below, do whatever is applicable in your case, and report the results to the list.

For any diagnosis, you're going to need to build a debugging kernel. The binaries from this site aren't debuggable. If you haven't done this before, read about 2. Compiling the kernel and modules and 11. Kernel debugging UML first.

2.18.1 14.1. Case 1 : Normal kernel panics

The most common case is for a normal thread to panic. To debug this, you will need to run it under the debugger (add 'debug' to the command line). An xterm will start up with gdb running inside it. Continue it when it stops in start_kernel and make it crash. Now `^C gdb` and

If the panic was a "Kernel mode fault", then there will be a segv frame on the stack and I'm going to want some more information. The stack might look something like this:

```
(UML gdb)  backtrace
#0  0x1009bf76 in __sigprocmask (how=1, set=0x5f347940, oset=0x0)
    at ../sysdeps/unix/sysv/linux/sigprocmask.c:49
#1  0x10091411 in change_sig (signal=10, on=1) at process.c:218
#2  0x10094785 in timer_handler (sig=26) at time_kern.c:32
#3  0x1009bf38 in __restore ()
    at ../sysdeps/unix/sysv/linux/i386/sigaction.c:125
#4  0x1009534c in segv (address=8, ip=268849158, is_write=2, is_
↪user=0)
    at trap_kern.c:66
#5  0x10095c04 in segv_handler (sig=11) at trap_user.c:285
#6  0x1009bf38 in __restore ()
```

I'm going to want to see the symbol and line information for the value of ip in the segv frame. In this case, you would do the following:

```
(UML gdb)  i sym 268849158
```

and:

```
(UML gdb)  i line *268849158
```

The reason for this is the __restore frame right above the segv_han- dler frame is hiding the frame that actually segfaulted. So, I have to get that information from the faulting ip.

2.18.2 14.2. Case 2 : Tracing thread panics

The less common and more painful case is when the tracing thread panics. In this case, the kernel debugger will be useless because it needs a healthy tracing thread in order to work. The first thing to do is get a backtrace from the tracing thread. This is done by figuring out what its pid is, firing up gdb, and attaching it to that pid. You can figure out the tracing thread pid by looking at the first line of the console output, which will look like this:

```
tracing thread pid = 15851
```

or by running ps on the host and finding the line that looks like this:

```
jdike 15851 4.5 0.4 132568 1104 pts/0 S 21:34 0:05 ./linux␣
↪[(tracing thread)]
```

If the panic was ‘segfault in signals’, then follow the instructions above for collecting information about the location of the seg fault.

If the tracing thread flaked out all by itself, then send that backtrace in and wait for our crack debugging team to fix the problem.

14.3. Case 3 : Tracing thread panics caused by other threads

However, there are cases where the misbehavior of another thread caused the problem. The most common panic of this type is:

```
wait_for_stop failed to wait for <pid> to stop with <signal␣
↪number>
```

In this case, you’ ll need to get a backtrace from the process men- tioned in the panic, which is complicated by the fact that the kernel debugger is defunct and without some fancy footwork, another gdb can’ t attach to it. So, this is how the fancy footwork goes:

In a shell:

```
host% kill -STOP pid
```

Run gdb on the tracing thread as described in case 2 and do:

```
(host gdb) call detach(pid)
```

If you get a segfault, do it again. It always works the second time.

Detach from the tracing thread and attach to that other thread:

```
(host gdb) detach




(host gdb) attach pid
```

If gdb hangs when attaching to that process, go back to a shell and do:

```
host%
kill -CONT pid
```

And then get the backtrace:

```
(host gdb)  backtrace
```

2.18.3 14.4. Case 4 : Hangs

Hangs seem to be fairly rare, but they sometimes happen. When a hang happens, we need a backtrace from the offending process. Run the kernel debugger as described in case 1 and get a backtrace. If the current process is not the idle thread, then send in the backtrace. You can tell that it's the idle thread if the stack looks like this:

```
#0  0x100b1401 in __libc_nanosleep ()
#1  0x100a2885 in idle_sleep (secs=10) at time.c:122
#2  0x100a546f in do_idle () at process_kern.c:445
#3  0x100a5508 in cpu_idle () at process_kern.c:471
#4  0x100ec18f in start_kernel () at init/main.c:592
#5  0x100a3e10 in start_kernel_proc (unused=0x0) at um_arch.c:71
#6  0x100a383f in signal_tramp (arg=0x100a3dd8) at trap_user.c:50
```

If this is the case, then some other process is at fault, and went to sleep when it shouldn't have. Run ps on the host and figure out which process should not have gone to sleep and stayed asleep. Then attach to it with gdb and get a backtrace as described in case 3.

2.19 15. Thanks

A number of people have helped this project in various ways, and this page gives recognition where recognition is due.

If you're listed here and you would prefer a real link on your name, or no link at all, instead of the despammed email address pseudo-link, let me know.

If you're not listed here and you think maybe you should be, please let me know that as well. I try to get everyone, but sometimes my bookkeeping lapses and I forget about contributions.

2.19.1 15.1. Code and Documentation

Rusty Russell <rusty at linuxcare.com.au> -

- wrote the HOWTO http://user-mode-linux.sourceforge.net/old/UserModeLinux-HOWTO.html
- prodded me into making this project official and putting it on SourceForge
- came up with the way cool UML logo http://user-mode-linux.sourceforge.net/uml-small.png
- redid the config process

Peter Moulder <reiter at netspace.net.au> - Fixed my config and build processes, and added some useful code to the block driver

Bill Stearns <wstearns at pobox.com> -

- HOWTO updates
- lots of bug reports
- lots of testing
- dedicated a box (uml.ists.dartmouth.edu) to support UML development
- wrote the mkrootfs script, which allows bootable filesystems of RPM-based distributions to be cranked out
- cranked out a large number of filesystems with said script

Jim Leu <jleu at mindspring.com> - Wrote the virtual ethernet driver and associated usermode tools

Lars Brinkhoff http://lars.nocrew.org/ - Contributed the ptrace proxy from his own project to allow easier kernel debugging

Andrea Arcangeli <andrea at suse.de> - Redid some of the early boot code so that it would work on machines with Large File Support

Chris Emerson - Did the first UML port to Linux/ppc

Harald Welte <laforge at gnumonks.org> - Wrote the multicast transport for the network driver

Jorgen Cederlof - Added special file support to hostfs

Greg Lonnon <glonnon at ridgerun dot com> - Changed the ubd driver to allow it to layer a COW file on a shared read-only filesystem and wrote the iomem emulation support

Henrik Nordstrom http://hem.passagen.se/hno/ - Provided a variety of patches, fixes, and clues

Lennert Buytenhek - Contributed various patches, a rewrite of the network driver, the first implementation of the mconsole driver, and did the bulk of the work needed to get SMP working again.

Yon Uriarte - Fixed the TUN/TAP network backend while I slept.

Adam Heath - Made a bunch of nice cleanups to the initialization code, plus various other small patches.

Matt Zimmerman - Matt volunteered to be the UML Debian maintainer and is doing a real nice job of it. He also noticed and fixed a number of actually and potentially exploitable security holes in uml_net. Plus the occasional patch. I like patches.

James McMechan - James seems to have taken over maintenance of the ubd driver and is doing a nice job of it.

Chandan Kudige - wrote the umlgdb script which automates the reloading of module symbols.

Steve Schmidtke - wrote the UML slirp transport and hostaudio drivers, enabling UML processes to access audio devices on the host. He also submitted patches for the slip transport and lots of other things.

David Coulson http://davidcoulson.net -

- Set up the http://usermodelinux.org site, which is a great way of keeping the UML user community on top of UML goings-on.
- Site documentation and updates
- Nifty little UML management daemon UMLd
- Lots of testing and bug reports

2.19.2 15.2. Flushing out bugs

- Yuri Pudgorodsky
- Gerald Britton
- Ian Wehrman
- Gord Lamb
- Eugene Koontz
- John H. Hartman
- Anders Karlsson
- Daniel Phillips
- John Fremlin
- Rainer Burgstaller
- James Stevenson
- Matt Clay
- Cliff Jefferies
- Geoff Hoff
- Lennert Buytenhek
- Al Viro

- Frank Klingenhoefer
- Livio Baldini Soares
- Jon Burgess
- Petru Paler
- Paul
- Chris Reahard
- Sverker Nilsson
- Gong Su
- johan verrept
- Bjorn Eriksson
- Lorenzo Allegrucci
- Muli Ben-Yehuda
- David Mansfield
- Howard Goff
- Mike Anderson
- John Byrne
- Sapan J. Batia
- Iris Huang
- Jan Hudec
- Voluspa

2.19.3 15.3. Buglets and clean-ups

- Dave Zarzycki
- Adam Lazur
- Boria Feigin
- Brian J. Murrell
- JS
- Roman Zippel
- Wil Cooley
- Ayelet Shemesh
- Will Dyson
- Sverker Nilsson
- dvorak
- v.naga srinivas

- Shlomi Fish
- Roger Binns
- johan verrept
- MrChuoi
- Peter Cleve
- Vincent Guffens
- Nathan Scott
- Patrick Caulfield
- jbearce
- Catalin Marinas
- Shane Spencer
- Zou Min
- Ryan Boder
- Lorenzo Colitti
- Gwendal Grignou
- Andre' Breiler
- Tsutomu Yasuda

2.19.4 15.4. Case Studies

- Jon Wright
- William McEwan
- Michael Richardson

2.19.5 15.5. Other contributions

Bill Carr <Bill.Carr at compaq.com> made the Red Hat mkrootfs script work with RH 6.2.

Michael Jennings <mikejen at hevanet.com> sent in some material which is now gracing the top of the index page http://user-mode-linux.sourceforge.net/ of this site.

SGI (and more specifically Ralf Baechle <ralf at uni-koblenz.de>) gave me an account on oss.sgi.com. The bandwidth there made it possible to produce most of the filesystems available on the project download page.

Laurent Bonnaud <Laurent.Bonnaud at inpg.fr> took the old grotty Debian filesystem that I've been distributing and updated it to 2.2. It is now available by itself here.

Rik van Riel gave me some ftp space on ftp.nl.linux.org so I can make releases even when Sourceforge is broken.

Rodrigo de Castro looked at my broken pte code and told me what was wrong with it, letting me fix a long-standing (several weeks) and serious set of bugs.

Chris Reahard built a specialized root filesystem for running a DNS server jailed inside UML. It' s available from the download http://user-mode-linux.sourceforge.net/old/dl-sf.html page in the Jail Filesystems section.

CHAPTER

THREE

PARAVIRT_OPS

Linux provides support for different hypervisor virtualization technologies. Historically different binary kernels would be required in order to support different hypervisors, this restriction was removed with pv_ops. Linux pv_ops is a virtualization API which enables support for different hypervisors. It allows each hypervisor to override critical operations and allows a single kernel binary to run on all supported execution environments including native machine – without any hypervisors.

pv_ops provides a set of function pointers which represent operations corresponding to low level critical instructions and high level functionalities in various areas. pv-ops allows for optimizations at run time by enabling binary patching of the low-ops critical operations at boot time.

pv_ops operations are classified into three categories:

- **simple indirect call** These operations correspond to high level functionality where it is known that the overhead of indirect call isn't very important.
- **indirect call which allows optimization with binary patch** Usually these operations correspond to low level critical instructions. They are called frequently and are performance critical. The overhead is very important.
- **a set of macros for hand written assembly code** Hand written assembly codes (.S files) also need paravirtualization because they include sensitive instructions or some of code paths in them are very performance critical.

CHAPTER

FOUR

GUEST HALT POLLING

The cpuidle_haltpoll driver, with the haltpoll governor, allows the guest vcpus to poll for a specified amount of time before halting.

This provides the following benefits to host side polling:

1) The POLL flag is set while polling is performed, which allows a remote vCPU to avoid sending an IPI (and the associated cost of handling the IPI) when performing a wakeup.

2) The VM-exit cost can be avoided.

The downside of guest side polling is that polling is performed even with other runnable tasks in the host.

The basic logic as follows: A global value, guest_halt_poll_ns, is configured by the user, indicating the maximum amount of time polling is allowed. This value is fixed.

Each vcpu has an adjustable guest_halt_poll_ns (“per-cpu guest_halt_poll_ns”), which is adjusted by the algorithm in response to events (explained below).

4.1 Module Parameters

The haltpoll governor has 5 tunable module parameters:

1) guest_halt_poll_ns:

Maximum amount of time, in nanoseconds, that polling is performed before halting.

Default: 200000

2) guest_halt_poll_shrink:

Division factor used to shrink per-cpu guest_halt_poll_ns when wakeup event occurs after the global guest_halt_poll_ns.

Default: 2

3) guest_halt_poll_grow:

Multiplication factor used to grow per-cpu guest_halt_poll_ns when event occurs after per-cpu guest_halt_poll_ns but before global guest_halt_poll_ns.

Default: 2

4) guest_halt_poll_grow_start:

The per-cpu guest_halt_poll_ns eventually reaches zero in case of an idle system. This value sets the initial per-cpu guest_halt_poll_ns when growing. This can be increased from 10000, to avoid misses during the initial growth stage:

10k, 20k, 40k, ···(example assumes guest_halt_poll_grow=2).

Default: 50000

5) guest_halt_poll_allow_shrink:

Bool parameter which allows shrinking. Set to N to avoid it (per-cpu guest_halt_poll_ns will remain high once achieves global guest_halt_poll_ns value).

Default: Y

The module parameters can be set from the debugfs files in:

```
/sys/module/haltpoll/parameters/
```

4.2 Further Notes

- Care should be taken when setting the guest_halt_poll_ns parameter as a large value has the potential to drive the cpu usage to 100% on a machine which would be almost entirely idle otherwise.

BIBLIOGRAPHY

[white-paper] http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[api-spec] http://support.amd.com/TechDocs/55766_SEV-KM_API_Specification.pdf

[amd-apm] http://support.amd.com/TechDocs/24593.pdf (section 15.34)

[kvm-forum] http://www.linux-kvm.org/images/7/74/02x08A-Thomas_Lendacky-AMDs_Virtualizatoin_Memory_Encryption_Technology.pdf

[atomic-ops] Documentation/core-api/atomic_ops.rst

[memory-barriers] Documentation/memory-barriers.txt

[lwn-mb] https://lwn.net/Articles/573436/