Interface manual

Xen v3.0 for x86

DISCLAIMER: This documentation is always under active development and as such there may be mistakes and omissions — watch out for these and please report any you find to the developer's mailing list. The latest version is always available on-line. Contributions of material, suggestions and corrections are welcome.

Contents

Chapter 1

Introduction

Xen allows the hardware resources of a machine to be virtualized and dynamically partitioned, allowing multiple different *guest* operating system images to be run simultaneously. Virtualizing the machine in this manner provides considerable flexibility, for example allowing different users to choose their preferred operating system (e.g., Linux, NetBSD, or a custom operating system). Furthermore, Xen provides secure partitioning between virtual machines (known as *domains* in Xen terminology), and enables better resource accounting and QoS isolation than can be achieved with a conventional operating system.

Xen essentially takes a 'whole machine' virtualization approach as pioneered by IBM VM/370. However, unlike VM/370 or more recent efforts such as VMware and Virtual PC, Xen does not attempt to completely virtualize the underlying hardware. Instead parts of the hosted guest operating systems are modified to work with the VMM; the operating system is effectively ported to a new target architecture, typically requiring changes in just the machine-dependent code. The user-level API is unchanged, and so existing binaries and operating system distributions work without modification.

In addition to exporting virtualized instances of CPU, memory, network and block devices, Xen exposes a control interface to manage how these resources are shared between the running domains. Access to the control interface is restricted: it may only be used by one specially-privileged VM, known as *domain 0*. This domain is a required part of any Xen-based server and runs the application software that manages the control-plane aspects of the platform. Running the control software in *domain 0*, distinct from the hypervisor itself, allows the Xen framework to separate the notions of mechanism and policy within the system.

Chapter 2

Virtual Architecture

In a Xen/x86 system, only the hypervisor runs with full processor privileges (*ring 0* in the x86 four-ring model). It has full access to the physical memory available in the system and is responsible for allocating portions of it to running domains.

On a 32-bit x86 system, guest operating systems may use *rings 1*, *2* and *3* as they see fit. Segmentation is used to prevent the guest OS from accessing the portion of the address space that is reserved for Xen. We expect most guest operating systems will use ring 1 for their own operation and place applications in ring 3.

On 64-bit systems it is not possible to protect the hypervisor from untrusted guest code running in rings 1 and 2. Guests are therefore restricted to run in ring 3 only. The guest kernel is protected from its applications by context switching between the kernel and currently running application.

In this chapter we consider the basic virtual architecture provided by Xen: CPU state, exception and interrupt handling, and time. Other aspects such as memory and device access are discussed in later chapters.

2.1 CPU state

All privileged state must be handled by Xen. The guest OS has no direct access to CR3 and is not permitted to update privileged bits in EFLAGS. Guest OSes use *hypercalls* to invoke operations in Xen; these are analogous to system calls but occur from ring 1 to ring 0.

A list of all hypercalls is given in Appendix A.

2.2 Exceptions

A virtual IDT is provided — a domain can submit a table of trap handlers to Xen via the **set_trap_table** hypercall. The exception stack frame presented to a virtual trap handler is identical to its native equivalent.

2.3 Interrupts and events

Interrupts are virtualized by mapping them to *event channels*, which are delivered asynchronously to the target domain using a callback supplied via the **set_callbacks** hypercall. A guest OS can map these events onto its standard interrupt dispatch mechanisms. Xen is responsible for determining the target domain that will handle each physical interrupt source. For more details on the binding of event sources to event channels, see Chapter 8.

2.4 Time

Guest operating systems need to be aware of the passage of both real (or wallclock) time and their own 'virtual time' (the time for which they have been executing). Furthermore, Xen has a notion of time which is used for scheduling. The following notions of time are provided:

Cycle counter time. This provides a fine-grained time reference. The cycle counter time is used to accurately extrapolate the other time references. On SMP machines it is currently assumed that the cycle counter time is synchronized between CPUs. The current x86-based implementation achieves this within inter-CPU communication latencies.

System time. This is a 64-bit counter which holds the number of nanoseconds that have elapsed since system boot.

Wall clock time. This is the time of day in a Unix-style **struct timeval** (seconds and microseconds since 1 January 1970, adjusted by leap seconds). An NTP client hosted by *domain 0* can keep this value accurate.

Domain virtual time. This progresses at the same pace as system time, but only while a domain is executing — it stops while a domain is de-scheduled. Therefore the share of the CPU that a domain receives is indicated by the rate at which its virtual time increases.

Xen exports timestamps for system time and wall-clock time to guest operating systems through a shared page of memory. Xen also provides the cycle counter time at the instant the timestamps were calculated, and the CPU frequency in Hertz. This allows the guest to extrapolate system and wall-clock times accurately based on the current cycle counter time.

Since all time stamps need to be updated and read *atomically* a version number is also stored in the shared info page, which is incremented before and after updating the timestamps. Thus a guest can be sure that it read a consistent state by checking the two version numbers are equal and even.

Xen includes a periodic ticker which sends a timer event to the currently executing domain every 10ms. The Xen scheduler also sends a timer event whenever a domain is scheduled; this allows the guest OS to adjust for the time that has passed while it has been inactive. In addition, Xen allows each domain to request that they receive a timer event sent at a specified system time by using the **set_timer_op** hypercall. Guest OSes may use this timer to implement timeout values when they block.

2.5 Xen CPU Scheduling

Xen offers a uniform API for CPU schedulers. It is possible to choose from a number of schedulers at boot and it should be easy to add more. The SEDF and Credit schedulers are part of the normal Xen distribution. SEDF will be going away and its use should be avoided once the credit scheduler has stabilized and become the default. The Credit scheduler provides proportional fair shares of the host's CPUs to the running domains. It does this while transparently load balancing runnable VCPUs across the whole system.

Note: SMP host support Xen has always supported SMP host systems. When using the credit scheduler, a domain's VCPUs will be dynamically moved across physical CPUs to maximise domain and system throughput. VCPUs can also be manually restricted to be mapped only on a subset of the host's physical CPUs, using the pinning mechanism.

2.6 Privileged operations

Xen exports an extended interface to privileged domains (viz. *Domain 0*). This allows such domains to build and boot other domains on the server, and provides control interfaces for managing scheduling, memory, networking, and block devices.

Chapter 3

Memory

Xen is responsible for managing the allocation of physical memory to domains, and for ensuring safe use of the paging and segmentation hardware.

3.1 Memory Allocation

As well as allocating a portion of physical memory for its own private use, Xen also reserves s small fixed portion of every virtual address space. This is located in the top 64MB on 32-bit systems, the top 168MB on PAE systems, and a larger portion in the middle of the address space on 64-bit systems. Unreserved physical memory is available for allocation to domains at a page granularity. Xen tracks the ownership and use of each page, which allows it to enforce secure partitioning between domains.

Each domain has a maximum and current physical memory allocation. A guest OS may run a 'balloon driver' to dynamically adjust its current memory allocation up to its limit.

3.2 Pseudo-Physical Memory

Since physical memory is allocated and freed on a page granularity, there is no guarantee that a domain will receive a contiguous stretch of physical memory. However most operating systems do not have good support for operating in a fragmented physical address space. To aid porting such operating systems to run on top of Xen, we make a distinction between *machine memory* and *pseudo-physical memory*.

Put simply, machine memory refers to the entire amount of memory installed in the machine, including that reserved by Xen, in use by various domains, or currently unallocated. We consider machine memory to comprise a set of 4kB *machine page frames* numbered consecutively starting from 0. Machine frame numbers mean the same within Xen or any domain.

Pseudo-physical memory, on the other hand, is a per-domain abstraction. It allows a guest operating system to consider its memory allocation to consist of a contiguous range of physical page frames starting at physical frame 0, despite the fact that the underlying machine page frames may be sparsely allocated and in any order.

To achieve this, Xen maintains a globally readable *machine-to-physical* table which records the mapping from machine page frames to pseudo-physical ones. In addition, each domain is supplied with a *physical-to-machine* table which performs the inverse mapping. Clearly the machine-to-physical table has size proportional to the amount of RAM installed in the machine, while each physical-to-machine table has size proportional to the memory allocation of the given domain.

Architecture dependent code in guest operating systems can then use the two tables to provide the abstraction of pseudo-physical memory. In general, only certain specialized parts of the operating system (such as page table management) needs to understand the difference between machine and pseudo-physical addresses.

3.3 Page Table Updates

In the default mode of operation, Xen enforces read-only access to page tables and requires guest operating systems to explicitly request any modifications. Xen validates all such requests and only applies updates that it deems safe. This is necessary to prevent domains from adding arbitrary mappings to their page tables.

To aid validation, Xen associates a type and reference count with each memory page. A page has one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings of its own memory regardless of its current type.

This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the page concerned to simultaneously be of types PT and RW.

mmu_update(mmu_update_t *req, int count, int *success_count, domid_t domid)

This hypercall is used to make updates to either the domain's pagetables or to the machine to physical mapping table. It supports submitting a queue of updates, allowing batching for maximal performance. Explicitly queuing updates using this interface will cause any outstanding writable pagetable state to be flushed from the system.

3.4 Writable Page Tables

Xen also provides an alternative mode of operation in which guests have the illusion that their page tables are directly writable. Of course this is not really the case, since Xen must still validate modifications to ensure secure partitioning. To this end, Xen traps any write attempt to a memory page of type PT (i.e., that is currently part of a page table). If such an access occurs, Xen temporarily allows write access to that page while at the same time *disconnecting* it from the page table that is currently in use. This allows the guest to safely make updates to the page because the newly-updated entries cannot be used by the MMU until Xen revalidates and reconnects the page. Reconnection occurs automatically in a number of situations: for example, when the guest modifies a different page-table page, when the domain is preempted, or whenever the guest uses Xen's explicit page-table update interfaces.

Writable pagetable functionality is enabled when the guest requests it, using a **vm_assist** hypercall. Writable pagetables do *not* provide full virtualisation of the MMU, so the memory management code of the guest still needs to be aware that it is running on Xen. Since the guest's page tables are used directly, it must translate pseudo-physical addresses to real machine addresses when building page table entries. The guest may not attempt to map its own pagetables writably, since this would violate the memory type invariants; page tables will automatically be made writable by the hypervisor, as necessary.

3.5 Shadow Page Tables

Finally, Xen also supports a form of *shadow page tables* in which the guest OS uses a independent copy of page tables which are unknown to the hardware (i.e. which are never pointed to by `cr3`). Instead Xen propagates changes made to the guest's tables to the real ones, and vice versa. This is useful for logging page writes (e.g. for live migration or checkpoint). A full version of the shadow page tables also allows guest OS porting with less effort.

3.6 Segment Descriptor Tables

At start of day a guest is supplied with a default GDT, which does not reside within its own memory allocation. If the guest wishes to use other than the default 'flat' ring-1 and ring-3 segments that this GDT provides, it must register a custom GDT and/or LDT with Xen, allocated from its own memory.

The following hypercall is used to specify a new GDT:

> int **set_gdt**(unsigned long **frame_list*, int *entries*)
>
> *frame_list*: An array of up to 14 machine page frames within which the GDT resides. Any frame registered as a GDT frame may only be mapped read-only within the guest's address space (e.g., no writable mappings, no use as a page-table page, and so on). Only 14 pages may be specified because pages 15 and 16 are reserved for the hypervisor's GDT entries.
>
> *entries*: The number of descriptor-entry slots in the GDT.

The LDT is updated via the generic MMU update mechanism (i.e., via the **mmu_update** hypercall.

3.7 Start of Day

The start-of-day environment for guest operating systems is rather different to that provided by the underlying hardware. In particular, the processor is already executing in protected mode with paging enabled.

Domain 0 is created and booted by Xen itself. For all subsequent domains, the analogue of the boot-loader is the *domain builder*, user-space software running in *domain 0*. The domain builder is responsible for building the initial page tables for a domain and loading its kernel image at the appropriate virtual address.

3.8 VM assists

Xen provides a number of "assists" for guest memory management. These are available on an "opt-in" basis to provide commonly-used extra functionality to a guest.

vm_assist(unsigned int cmd, unsigned int type)

The **cmd** parameter describes the action to be taken, whilst the **type** parameter describes the kind of assist that is being referred to. Available commands are as follows:

VMASST_CMD_enable Enable a particular assist type

VMASST_CMD_disable Disable a particular assist type

And the available types are:

VMASST_TYPE_4gb_segments Provide emulated support for instructions that rely on 4GB segments (such as the techniques used by some TLS solutions).

VMASST_TYPE_4gb_segments_notify Provide a callback to the guest if the above segment fixups are used: allows the guest to display a warning message during boot.

VMASST_TYPE_writable_pagetables Enable writable pagetable mode - described above.

Chapter 4

Xen Info Pages

The **Shared info page** is used to share various CPU-related state between the guest OS and the hypervisor. This information includes VCPU status, time information and event channel (virtual interrupt) state. The **Start info page** is used to pass build-time information to the guest when it boots and when it is resumed from a suspended state. This chapter documents the fields included in the **shared_info_t** and **start_info_t** structures for use by the guest OS.

4.1 Shared info page

The **shared_info_t** is accessed at run time by both Xen and the guest OS. It is used to pass information relating to the virtual CPU and virtual machine state between the OS and the hypervisor.

The structure is declared in **xen/include/public/xen.h**:

```
typedef struct shared_info {
    vcpu_info_t vcpu_info[MAX_VIRT_CPUS];

    /*
     * A domain can create "event channels" on which it can send and receive
     * asynchronous event notifications. There are three classes of event that
     * are delivered by this mechanism:
     *  1. Bi-directional inter- and intra-domain connections. Domains must
     *     arrange out-of-band to set up a connection (usually by allocating
     *     an unbound 'listener' port and advertising that via a storage service
     *     such as xenstore).
     *  2. Physical interrupts. A domain with suitable hardware-access
     *     privileges can bind an event-channel port to a physical interrupt
     *     source.
     *  3. Virtual interrupts ('events'). A domain can bind an event-channel
     *     port to a virtual interrupt source, such as the virtual-timer
     *     device or the emergency console.
```

```
     *
     * Event channels are addressed by a "port index". Each channel is
     * associated with two bits of information:
     *  1. PENDING -- notifies the domain that there is a pending notification
     *     to be processed. This bit is cleared by the guest.
     *  2. MASK -- if this bit is clear then a 0->1 transition of PENDING
     *     will cause an asynchronous upcall to be scheduled. This bit is only
     *     updated by the guest. It is read-only within Xen. If a channel
     *     becomes pending while the channel is masked then the 'edge' is lost
     *     (i.e., when the channel is unmasked, the guest must manually handle
     *     pending notifications as no upcall will be scheduled by Xen).
     *
     * To expedite scanning of pending notifications, any 0->1 pending
     * transition on an unmasked channel causes a corresponding bit in a
     * per-vcpu selector word to be set. Each bit in the selector covers a
     * 'C long' in the PENDING bitfield array.
     */
    unsigned long evtchn_pending[sizeof(unsigned long) * 8];
    unsigned long evtchn_mask[sizeof(unsigned long) * 8];

    /*
     * Wallclock time: updated only by control software. Guests should base
     * their gettimeofday() syscall on this wallclock-base value.
     */
    uint32_t wc_version;      /* Version counter: see vcpu_time_info_t. */
    uint32_t wc_sec;          /* Secs  00:00:00 UTC, Jan 1, 1970.  */
    uint32_t wc_nsec;         /* Nsecs 00:00:00 UTC, Jan 1, 1970.  */

    arch_shared_info_t arch;

} shared_info_t;
```

vcpu_info An array of **vcpu_info_t** structures, each of which holds either runtime information about a virtual CPU, or is "empty" if the corresponding VCPU does not exist.

evtchn_pending Guest-global array, with one bit per event channel. Bits are set if an event is currently pending on that channel.

evtchn_mask Guest-global array for masking notifications on event channels.

wc_version Version counter for current wallclock time.

wc_sec Whole seconds component of current wallclock time.

wc_nsec Nanoseconds component of current wallclock time.

arch Host architecture-dependent portion of the shared info structure.

4.1.1 vcpu_info_t

```
typedef struct vcpu_info {
    /*
     * 'evtchn_upcall_pending' is written non-zero by Xen to indicate
```

```
     * a pending notification for a particular VCPU. It is then cleared
     * by the guest OS /before/ checking for pending work, thus avoiding
     * a set-and-check race. Note that the mask is only accessed by Xen
     * on the CPU that is currently hosting the VCPU. This means that the
     * pending and mask flags can be updated by the guest without special
     * synchronisation (i.e., no need for the x86 LOCK prefix).
     * This may seem suboptimal because if the pending flag is set by
     * a different CPU then an IPI may be scheduled even when the mask
     * is set. However, note:
     *  1. The task of 'interrupt holdoff' is covered by the per-event-
     *     channel mask bits. A 'noisy' event that is continually being
     *     triggered can be masked at source at this very precise
     *     granularity.
     *  2. The main purpose of the per-VCPU mask is therefore to restrict
     *     reentrant execution: whether for concurrency control, or to
     *     prevent unbounded stack usage. Whatever the purpose, we expect
     *     that the mask will be asserted only for short periods at a time,
     *     and so the likelihood of a 'spurious' IPI is suitably small.
     * The mask is read before making an event upcall to the guest: a
     * non-zero mask therefore guarantees that the VCPU will not receive
     * an upcall activation. The mask is cleared when the VCPU requests
     * to block: this avoids wakeup-waiting races.
     */
    uint8_t evtchn_upcall_pending;
    uint8_t evtchn_upcall_mask;
    unsigned long evtchn_pending_sel;
    arch_vcpu_info_t arch;
    vcpu_time_info_t time;
} vcpu_info_t; /* 64 bytes (x86) */
```

evtchn_upcall_pending This is set non-zero by Xen to indicate that there are pending events to be received.

evtchn_upcall_mask This is set non-zero to disable all interrupts for this CPU for short periods of time. If individual event channels need to be masked, the **evtchn_mask** in the **shared_info_t** is used instead.

evtchn_pending_sel When an event is delivered to this VCPU, a bit is set in this selector to indicate which word of the **evtchn_pending** array in the **shared_info_t** contains the event in question.

arch Architecture-specific VCPU info. On x86 this contains the virtualized CR2 register (page fault linear address) for this VCPU.

time Time values for this VCPU.

4.1.2 vcpu_time_info

```
typedef struct vcpu_time_info {
    /*
     * Updates to the following values are preceded and followed by an
     * increment of 'version'. The guest can therefore detect updates by
     * looking for changes to 'version'. If the least-significant bit of
```

```
     * the version number is set then an update is in progress and the guest
     * must wait to read a consistent set of values.
     * The correct way to interact with the version number is similar to
     * Linux's seqlock: see the implementations of read_seqbegin/read_seqretry.
     */
    uint32_t version;
    uint32_t pad0;
    uint64_t tsc_timestamp;   /* TSC at last update of time vals.  */
    uint64_t system_time;     /* Time, in nanosecs, since boot.    */
    /*
     * Current system time:
     *   system_time + ((tsc - tsc_timestamp) << tsc_shift) * tsc_to_system_mul
     * CPU frequency (Hz):
     *   ((10^9 << 32) / tsc_to_system_mul) >> tsc_shift
     */
    uint32_t tsc_to_system_mul;
    int8_t   tsc_shift;
    int8_t   pad1[3];
} vcpu_time_info_t; /* 32 bytes */
```

version Used to ensure the guest gets consistent time updates.

tsc_timestamp Cycle counter timestamp of last time value; could be used to expolate in between updates, for instance.

system_time Time since boot (nanoseconds).

tsc_to_system_mul Cycle counter to nanoseconds multiplier (used in extrapolating current time).

tsc_shift Cycle counter to nanoseconds shift (used in extrapolating current time).

4.1.3 arch_shared_info_t

On x86, the **arch_shared_info_t** is defined as follows (from xen/public/arch-x86_32.h):

```
typedef struct arch_shared_info {
    unsigned long max_pfn;                  /* max pfn that appears in table */
    /* Frame containing list of mfns containing list of mfns containing p2m. */
    unsigned long pfn_to_mfn_frame_list_list;
} arch_shared_info_t;
```

max_pfn The maximum PFN listed in the physical-to-machine mapping table (P2M table).

pfn_to_mfn_frame_list_list Machine address of the frame that contains the machine addresses of the P2M table frames.

4.2 Start info page

The start info structure is declared as the following (in **xen/include/public/xen.h**):

```
#define MAX_GUEST_CMDLINE 1024
typedef struct start_info {
    /* THE FOLLOWING ARE FILLED IN BOTH ON INITIAL BOOT AND ON RESUME.    */
    char magic[32];             /* "Xen-<version>.<subversion>". */
    unsigned long nr_pages;     /* Total pages allocated to this domain.  */
    unsigned long shared_info;  /* MACHINE address of shared info struct. */
    uint32_t flags;             /* SIF_xxx flags.                         */
    unsigned long store_mfn;    /* MACHINE page number of shared page.    */
    uint32_t store_evtchn;      /* Event channel for store communication. */
    unsigned long console_mfn;  /* MACHINE address of console page.       */
    uint32_t console_evtchn;    /* Event channel for console messages.    */
    /* THE FOLLOWING ARE ONLY FILLED IN ON INITIAL BOOT (NOT RESUME).     */
    unsigned long pt_base;      /* VIRTUAL address of page directory.     */
    unsigned long nr_pt_frames; /* Number of bootstrap p.t. frames.       */
    unsigned long mfn_list;     /* VIRTUAL address of page-frame list.    */
    unsigned long mod_start;    /* VIRTUAL address of pre-loaded module.  */
    unsigned long mod_len;      /* Size (bytes) of pre-loaded module.     */
    int8_t cmd_line[MAX_GUEST_CMDLINE];
} start_info_t;
```

The fields are in two groups: the first group are always filled in when a domain is booted or resumed, the second set are only used at boot time.

The always-available group is as follows:

magic A text string identifying the Xen version to the guest.

nr_pages The number of real machine pages available to the guest.

shared_info Machine address of the shared info structure, allowing the guest to map it during initialisation.

flags Flags for describing optional extra settings to the guest.

store_mfn Machine address of the Xenstore communications page.

store_evtchn Event channel to communicate with the store.

console_mfn Machine address of the console data page.

console_evtchn Event channel to notify the console backend.

The boot-only group may only be safely referred to during system boot:

pt_base Virtual address of the page directory created for us by the domain builder.

nr_pt_frames Number of frames used by the builders' bootstrap pagetables.

mfn_list Virtual address of the list of machine frames this domain owns.

mod_start Virtual address of any pre-loaded modules (e.g. ramdisk)

mod_len Size of pre-loaded module (if any).

cmd_line Kernel command line passed by the domain builder.

Chapter 5

Event Channels

Event channels are the basic primitive provided by Xen for event notifications. An event is the Xen equivalent of a hardware interrupt. They essentially store one bit of information, the event of interest is signalled by transitioning this bit from 0 to 1.

Notifications are received by a guest via an upcall from Xen, indicating when an event arrives (setting the bit). Further notifications are masked until the bit is cleared again (therefore, guests must check the value of the bit after re-enabling event delivery to ensure no missed notifications).

Event notifications can be masked by setting a flag; this is equivalent to disabling interrupts and can be used to ensure atomicity of certain operations in the guest kernel.

5.1 Hypercall interface

event_channel_op(evtchn_op_t *op)

The event channel operation hypercall is used for all operations on event channels / ports. Operations are distinguished by the value of the **cmd** field of the **op** structure. The possible commands are described below:

EVTCHNOP_alloc_unbound Allocate a new event channel port, ready to be connected to by a remote domain.

- Specified domain must exist.
- A free port must exist in that domain.

Unprivileged domains may only allocate their own ports, privileged domains may also allocate ports in other domains.

EVTCHNOP_bind_interdomain Bind an event channel for interdomain communications.

- Caller domain must have a free port to bind.
- Remote domain must exist.
- Remote port must be allocated and currently unbound.
- Remote port must be expecting the caller domain as the "remote".

EVTCHNOP_bind_virq Allocate a port and bind a VIRQ to it.

- Caller domain must have a free port to bind.
- VIRQ must be valid.
- VCPU must exist.
- VIRQ must not currently be bound to an event channel.

EVTCHNOP_bind_ipi Allocate and bind a port for notifying other virtual CPUs.

- Caller domain must have a free port to bind.
- VCPU must exist.

EVTCHNOP_bind_pirq Allocate and bind a port to a real IRQ.

- Caller domain must have a free port to bind.
- PIRQ must be within the valid range.
- Another binding for this PIRQ must not exist for this domain.
- Caller must have an available port.

EVTCHNOP_close Close an event channel (no more events will be received).

- Port must be valid (currently allocated).

EVTCHNOP_send Send a notification on an event channel attached to a port.

- Port must be valid.
- Only valid for Interdomain, IPI or Allocated Unbound ports.

EVTCHNOP_status Query the status of a port; what kind of port, whether it is bound, what remote domain is expected, what PIRQ or VIRQ it is bound to, what VCPU will be notified, etc. Unprivileged domains may only query the state of their own ports. Privileged domains may query any port.

EVTCHNOP_bind_vcpu Bind event channel to a particular VCPU - receive notification upcalls only on that VCPU.

- VCPU must exist.
- Port must be valid.
- Event channel must be either: allocated but unbound, bound to an interdomain event channel, bound to a PIRQ.

Chapter 6

Grant tables

Xen's grant tables provide a generic mechanism to memory sharing between domains. This shared memory interface underpins the split device drivers for block and network IO.

Each domain has its own **grant table**. This is a data structure that is shared with Xen; it allows the domain to tell Xen what kind of permissions other domains have on its pages. Entries in the grant table are identified by **grant references**. A grant reference is an integer, which indexes into the grant table. It acts as a capability which the grantee can use to perform operations on the granter's memory.

This capability-based system allows shared-memory communications between unprivileged domains. A grant reference also encapsulates the details of a shared page, removing the need for a domain to know the real machine address of a page it is sharing. This makes it possible to share memory correctly with domains running in fully virtualised memory.

6.1 Interface

6.1.1 Grant table manipulation

Creating and destroying grant references is done by direct access to the grant table. This removes the need to involve Xen when creating grant references, modifying access permissions, etc. The grantee domain will invoke hypercalls to use the grant references. Four main operations can be accomplished by directly manipulating the table:

Grant foreign access allocate a new entry in the grant table and fill out the access permissions accordingly. The access permissions will be looked up by Xen when the grantee attempts to use the reference to map the granted frame.

End foreign access check that the grant reference is not currently in use, then remove the mapping permissions for the frame. This prevents further mappings from taking place but does not allow forced revocations of existing mappings.

Grant foreign transfer allocate a new entry in the table specifying transfer permissions for the grantee. Xen will look up this entry when the grantee attempts to transfer a frame to the granter.

End foreign transfer remove permissions to prevent a transfer occurring in future. If the transfer is already committed, modifying the grant table cannot prevent it from completing.

6.1.2 Hypercalls

Use of grant references is accomplished via a hypercall. The grant table op hypercall takes three arguments:

grant_table_op(unsigned int cmd, void *uop, unsigned int count)

cmd indicates the grant table operation of interest. **uop** is a pointer to a structure (or an array of structures) describing the operation to be performed. The **count** field describes how many grant table operations are being batched together.

The core logic is situated in **xen/common/grant_table.c**. The grant table operation hypercall can be used to perform the following actions:

GNTTABOP_map_grant_ref Given a grant reference from another domain, map the referred page into the caller's address space.

GNTTABOP_unmap_grant_ref Remove a mapping to a granted frame from the caller's address space. This is used to voluntarily relinquish a mapping to a granted page.

GNTTABOP_setup_table Setup grant table for caller domain.

GNTTABOP_dump_table Debugging operation.

GNTTABOP_transfer Given a transfer reference from another domain, transfer ownership of a page frame to that domain.

Chapter 7

Xenstore

Xenstore is the mechanism by which control-plane activities occur. These activities include:

- Setting up shared memory regions and event channels for use with the split device drivers.
- Notifying the guest of control events (e.g. balloon driver requests)
- Reporting back status information from the guest (e.g. performance-related statistics, etc).

The store is arranged as a hierarchical collection of key-value pairs. Each domain has a directory hierarchy containing data related to its configuration. Domains are permitted to register for notifications about changes in subtrees of the store, and to apply changes to the store transactionally.

7.1 Guidelines

A few principles govern the operation of the store:

- Domains should only modify the contents of their own directories.
- The setup protocol for a device channel should simply consist of entering the configuration data into the store.
- The store should allow device discovery without requiring the relevant device drivers to be loaded: a Xen "bus" should be visible to probing code in the guest.
- The store should be usable for inter-tool communications, allowing the tools themselves to be decomposed into a number of smaller utilities, rather than

a single monolithic entity. This also facilitates the development of alternate user interfaces to the same functionality.

7.2 Store layout

There are three main paths in XenStore:

/vm stores configuration information about domain

/local/domain stores information about the domain on the local node (domid, etc.)

/tool stores information for the various tools

The **/vm** path stores configuration information for a domain. This information doesn't change and is indexed by the domain's UUID. A **/vm** entry contains the following information:

uuid uuid of the domain (somewhat redundant)

on_reboot the action to take on a domain reboot request (destroy or restart)

on_poweroff the action to take on a domain halt request (destroy or restart)

on_crash the action to take on a domain crash (destroy or restart)

vcpus the number of allocated vcpus for the domain

memory the amount of memory (in megabytes) for the domain Note: appears to sometimes be empty for domain-0

vcpu_avail the number of active vcpus for the domain (vcpus - number of disabled vcpus)

name the name of the domain

/vm/<uuid>/image/

The image path is only available for Domain-Us and contains:

ostype identifies the builder type (linux or vmx)

kernel path to kernel on domain-0

cmdline command line to pass to domain-U kernel

ramdisk path to ramdisk on domain-0

/local

The `/local` path currently only contains one directory, `/local/domain` that is indexed by domain id. It contains the running domain information. The reason to have two storage areas is that during migration, the uuid doesn't change but the domain id does. The `/local/domain` directory can be created and populated before finalizing the migration enabling localhost to localhost migration.

/local/domain/<domid>

This path contains:

cpu_time xend start time (this is only around for domain-0)

handle private handle for xend

name see /vm

on_reboot see /vm

on_poweroff see /vm

on_crash see /vm

vm the path to the VM directory for the domain

domid the domain id (somewhat redundant)

running indicates that the domain is currently running

memory the current memory in megabytes for the domain (empty for domain-0?)

maxmem_KiB the maximum memory for the domain (in kilobytes)

memory_KiB the memory allocated to the domain (in kilobytes)

cpu the current CPU the domain is pinned to (empty for domain-0?)

cpu_weight the weight assigned to the domain

vcpu_avail a bitmap telling the domain whether it may use a given VCPU

online_vcpus how many vcpus are currently online

vcpus the total number of vcpus allocated to the domain

console/ a directory for console information

- **ring-ref** the grant table reference of the console ring queue
- **port** the event channel being used for the console ring queue (local port)
- **tty** the current tty the console data is being exposed of
- **limit** the limit (in bytes) of console data to buffer

backend/ a directory containing all backends the domain hosts

- **vbd/** a directory containing vbd backends
  - **<domid>/** a directory containing vbd's for domid

<**virtual-device**>**/** a directory for a particular virtual-device on domid

frontend-id domain id of frontend

frontend the path to the frontend domain

physical-device backend device number

sector-size backend sector size

info 0 read/write, 1 read-only (is this right?)

domain name of frontend domain

params parameters for device

type the type of the device

dev the virtual device (as given by the user)

node output from block creation script

vif/ a directory containing vif backends

<**domid**>**/** a directory containing vif's for domid

<**vif number**>**/** a directory for each vif

frontend-id the domain id of the frontend

frontend the path to the frontend

mac the mac address of the vif

bridge the bridge the vif is connected to

handle the handle of the vif

script the script used to create/stop the vif

domain the name of the frontend

vtpm/ a directory containing vtpm backends

<**domid**>**/** a directory containing vtpm's for domid

<**vtpm number**>**/** a directory for each vtpm

frontend-id the domain id of the frontend

frontend the path to the frontend

instance the instance of the virtual TPM that is used

pref_instance the instance number as given in the VM configuration file; may be different from **instance**

domain the name of the domain of the frontend

device/ a directory containing the frontend devices for the domain

vbd/ a directory containing vbd frontend devices for the domain

<**virtual-device**>**/** a directory containing the vbd frontend for virtual-device

virtual-device the device number of the frontend device

backend-id the domain id of the backend

backend the path of the backend in the store (/local/domain path)

ring-ref the grant table reference for the block request ring queue

event-channel the event channel used for the block request ring queue

vif/ a directory containing vif frontend devices for the domain

<**id**>**/** a directory for vif id frontend device for the domain

backend-id the backend domain id

mac the mac address of the vif

handle the internal vif handle

backend a path to the backend's store entry

tx-ring-ref the grant table reference for the transmission ring queue

rx-ring-ref the grant table reference for the receiving ring queue

event-channel the event channel used for the two ring queues

vtpm/ a directory containing the vtpm frontend device for the domain

<**id**> a directory for vtpm id frontend device for the domain

backend-id the backend domain id

backend a path to the backend's store entry

ring-ref the grant table reference for the tx/rx ring

event-channel the event channel used for the ring

device-misc/ miscellaneous information for devices

vif/ miscellaneous information for vif devices

nextDeviceID the next device id to use

security/ access control information for the domain

- **ssidref** security reference identifier used inside the hypervisor
- **access_control/** security label used by management tools
  - **label** security label name
  - **policy** security policy name

store/ per-domain information for the store

- **port** the event channel used for the store ring queue
- **ring-ref** - the grant table reference used for the store's communication channel

image - private xend information

Chapter 8

Devices

Virtual devices under Xen are provided by a **split device driver** architecture. The illusion of the virtual device is provided by two co-operating drivers: the **frontend**, which runs an the unprivileged domain and the **backend**, which runs in a domain with access to the real device hardware (often called a **driver domain**; in practice domain 0 usually fulfills this function).

The frontend driver appears to the unprivileged guest as if it were a real device, for instance a block or network device. It receives IO requests from its kernel as usual, however since it does not have access to the physical hardware of the system it must then issue requests to the backend. The backend driver is responsible for receiving these IO requests, verifying that they are safe and then issuing them to the real device hardware. The backend driver appears to its kernel as a normal user of in-kernel IO functionality. When the IO completes the backend notifies the frontend that the data is ready for use; the frontend is then able to report IO completion to its own kernel.

Frontend drivers are designed to be simple; most of the complexity is in the backend, which has responsibility for translating device addresses, verifying that requests are well-formed and do not violate isolation guarantees, etc.

Split drivers exchange requests and responses in shared memory, with an event channel for asynchronous notifications of activity. When the frontend driver comes up, it uses Xenstore to set up a shared memory frame and an interdomain event channel for communications with the backend. Once this connection is established, the two can communicate directly by placing requests / responses into shared memory and then sending notifications on the event channel. This separation of notification from data transfer allows message batching, and results in very efficient device access.

This chapter focuses on some individual split device interfaces available to Xen guests.

8.1 Network I/O

Virtual network device services are provided by shared memory communication with a backend domain. From the point of view of other domains, the backend may be viewed as a virtual ethernet switch element with each domain having one or more virtual network interfaces connected to it.

From the point of view of the backend domain itself, the network backend driver consists of a number of ethernet devices. Each of these has a logical direct connection to a virtual network device in another domain. This allows the backend domain to route, bridge, firewall, etc the traffic to / from the other domains using normal operating system mechanisms.

8.1.1 Backend Packet Handling

The backend driver is responsible for a variety of actions relating to the transmission and reception of packets from the physical device. With regard to transmission, the backend performs these key actions:

- **Validation:** To ensure that domains do not attempt to generate invalid (e.g. spoofed) traffic, the backend driver may validate headers ensuring that source MAC and IP addresses match the interface that they have been sent from.

  Validation functions can be configured using standard firewall rules (`iptables` in the case of Linux).

- **Scheduling:** Since a number of domains can share a single physical network interface, the backend must mediate access when several domains each have packets queued for transmission. This general scheduling function subsumes basic shaping or rate-limiting schemes.

- **Logging and Accounting:** The backend domain can be configured with classifier rules that control how packets are accounted or logged. For example, log messages might be generated whenever a domain attempts to send a TCP packet containing a SYN.

On receipt of incoming packets, the backend acts as a simple demultiplexer: Packets are passed to the appropriate virtual interface after any necessary logging and accounting have been carried out.

8.1.2 Data Transfer

Each virtual interface uses two "descriptor rings", one for transmit, the other for receive. Each descriptor identifies a block of contiguous machine memory allocated to the domain.

The transmit ring carries packets to transmit from the guest to the backend domain. The return path of the transmit ring carries messages indicating that the contents have been physically transmitted and the backend no longer requires the associated pages of memory.

To receive packets, the guest places descriptors of unused pages on the receive ring. The backend will return received packets by exchanging these pages in the domain's memory with new pages containing the received data, and passing back descriptors regarding the new packets on the ring. This zero-copy approach allows the backend to maintain a pool of free pages to receive packets into, and then deliver them to appropriate domains after examining their headers.

If a domain does not keep its receive ring stocked with empty buffers then packets destined to it may be dropped. This provides some defence against receive livelock problems because an overloaded domain will cease to receive further data. Similarly, on the transmit path, it provides the application with feedback on the rate at which packets are able to leave the system.

Flow control on rings is achieved by including a pair of producer indexes on the shared ring page. Each side will maintain a private consumer index indicating the next outstanding message. In this manner, the domains cooperate to divide the ring into two message lists, one in each direction. Notification is decoupled from the immediate placement of new messages on the ring; the event channel will be used to generate notification when *either* a certain number of outstanding messages are queued, *or* a specified number of nanoseconds have elapsed since the oldest message was placed on the ring.

8.1.3 Network ring interface

The network device uses two shared memory rings for communication: one for transmit, one for receive.

Transmit requests are described by the following structure:

```
typedef struct netif_tx_request {
    grant_ref_t gref;      /* Reference to buffer page */
    uint16_t offset;       /* Offset within buffer page */
    uint16_t flags;        /* NETTXF_* */
    uint16_t id;           /* Echoed in response message. */
```

```
    uint16_t size;         /* Packet size in bytes.       */
} netif_tx_request_t;
```

gref Grant reference for the network buffer

offset Offset to data

flags Transmit flags (currently only NETTXF_csum_blank is supported, to indicate that the protocol checksum field is incomplete).

id Echoed to guest by the backend in the ring-level response so that the guest can match it to this request

size Buffer size

Each transmit request is followed by a transmit response at some later date. This is part of the shared-memory communication protocol and allows the guest to (potentially) retire internal structures related to the request. It does not imply a network-level response. This structure is as follows:

```
typedef struct netif_tx_response {
    uint16_t id;
    int16_t  status;
} netif_tx_response_t;
```

id Echo of the ID field in the corresponding transmit request.

status Success / failure status of the transmit request.

Receive requests must be queued by the frontend, accompanied by a donation of page-frames to the backend. The backend transfers page frames full of data back to the guest

```
typedef struct {
    uint16_t    id;        /* Echoed in response message.        */
    grant_ref_t gref;      /* Reference to incoming granted frame */
} netif_rx_request_t;
```

id Echoed by the frontend to identify this request when responding.

gref Transfer reference - the backend will use this reference to transfer a frame of network data to us.

Receive response descriptors are queued for each received frame. Note that these may only be queued in reply to an existing receive request, providing an in-built form of traffic throttling.

```
typedef struct {
    uint16_t id;
    uint16_t offset;       /* Offset in page of start of received packet  */
    uint16_t flags;        /* NETRXF_* */
    int16_t  status;       /* -ve: BLKIF_RSP_* ; +ve: Rx'ed pkt size. */
} netif_rx_response_t;
```

id ID echoed from the original request, used by the guest to match this response to the original request.

offset Offset to data within the transferred frame.

flags Transmit flags (currently only NETRXF_csum_valid is supported, to indicate that the protocol checksum field has already been validated).

status Success / error status for this operation.

Note that the receive protocol includes a mechanism for guests to receive incoming memory frames but there is no explicit transfer of frames in the other direction. Guests are expected to return memory to the hypervisor in order to use the network interface. They *must* do this or they will exceed their maximum memory reservation and will not be able to receive incoming frame transfers. When necessary, the backend is able to replenish its pool of free network buffers by claiming some of this free memory from the hypervisor.

8.2 Block I/O

All guest OS disk access goes through the virtual block device VBD interface. This interface allows domains access to portions of block storage devices visible to the the block backend device. The VBD interface is a split driver, similar to the network interface described above. A single shared memory ring is used between the frontend and backend drivers for each virtual device, across which IO requests and responses are sent.

Any block device accessible to the backend domain, including network-based block (iSCSI, *NBD, etc), loopback and LVM/MD devices, can be exported as a VBD. Each VBD is mapped to a device node in the guest, specified in the guest's startup configuration.

8.2.1 Data Transfer

The per-(virtual)-device ring between the guest and the block backend supports two messages:

`READ`**:** Read data from the specified block device. The front end identifies the device and location to read from and attaches pages for the data to be copied to (typically via DMA from the device). The backend acknowledges completed read requests as they finish.

`WRITE`**:** Write data to the specified block device. This functions essentially as `READ`, except that the data moves to the device instead of from it.

8.2.2 Block ring interface

The block interface is defined by the structures passed over the shared memory interface. These structures are either requests (from the frontend to the backend) or responses (from the backend to the frontend).

The request structure is defined as follows:

```
typedef struct blkif_request {
    uint8_t        operation;    /* BLKIF_OP_???                         */
    uint8_t        nr_segments;  /* number of segments                   */
    blkif_vdev_t   handle;       /* only for read/write requests         */
    uint64_t       id;           /* private guest value, echoed in resp  */
    blkif_sector_t sector_number;/* start sector idx on disk (r/w only)  */
    struct blkif_request_segment {
        grant_ref_t gref;        /* reference to I/O buffer frame        */
        /* @first_sect: first sector in frame to transfer (inclusive).   */
        /* @last_sect: last sector in frame to transfer (inclusive).     */
        uint8_t     first_sect, last_sect;
    } seg[BLKIF_MAX_SEGMENTS_PER_REQUEST];
} blkif_request_t;
```

The fields are as follows:

operation operation ID: one of the operations described above

nr_segments number of segments for scatter / gather IO described by this request

handle identifier for a particular virtual device on this interface

id this value is echoed in the response message for this IO; the guest may use it to identify the original request

sector_number start sector on the virtual device for this request

frame_and_sects This array contains structures encoding scatter-gather IO to be performed:

gref The grant reference for the foreign I/O buffer page.

first_sect First sector to access within the buffer page (0 to 7).

last_sect Last sector to access within the buffer page (0 to 7).

Data will be transferred into frames at an offset determined by the value of `first_sect`.

8.3 Virtual TPM

Virtual TPM (VTPM) support provides TPM functionality to each virtual machine that requests this functionality in its configuration file. The interface enables domains to access their own private TPM like it was a hardware TPM built into the machine.

The virtual TPM interface is implemented as a split driver, similar to the network and block interfaces described above. The user domain hosting the frontend exports a character device /dev/tpm0 to user-level applications for communicating with the virtual TPM. This is the same device interface that is also offered if a hardware TPM is available in the system. The backend provides a single interface /dev/vtpm where the virtual TPM is waiting for commands from all domains that have located their backend in a given domain.

8.3.1 Data Transfer

A single shared memory ring is used between the frontend and backend drivers. TPM requests and responses are sent in pages where a pointer to those pages and other information is placed into the ring such that the backend can map the pages into its memory space using the grant table mechanism.

The backend driver has been implemented to only accept well-formed TPM requests. To meet this requirement, the length indicator in the TPM request must correctly indicate the length of the request. Otherwise an error message is automatically sent back by the device driver.

The virtual TPM implementation listens for TPM request on /dev/vtpm. Since it must be able to apply the TPM request packet to the virtual TPM instance associated with the virtual machine, a 4-byte virtual TPM instance identifier is pretended to each packet by the backend driver (in network byte order) for internal routing of the request.

8.3.2 Virtual TPM ring interface

The TPM protocol is a strict request/response protocol and therefore only one ring is used to send requests from the frontend to the backend and responses on the reverse path.

The request/response structure is defined as follows:

```
typedef struct {
    unsigned long addr;     /* Machine address of packet.     */
```

```
    grant_ref_t ref;         /* grant table access reference.  */
    uint16_t unused;         /* unused                         */
    uint16_t size;           /* Packet size in bytes.          */
} tpmif_tx_request_t;
```

The fields are as follows:

addr The machine address of the page associated with the TPM request/response; a request/response may span multiple pages

ref The grant table reference associated with the address.

size The size of the remaining packet; up to PAGE_SIZE bytes can be found in the page referenced by 'addr'

The frontend initially allocates several pages whose addresses are stored in the ring. Only these pages are used for exchange of requests and responses.

Chapter 9

Further Information

If you have questions that are not answered by this manual, the sources of information listed below may be of interest to you. Note that bug reports, suggestions and contributions related to the software (or the documentation) should be sent to the Xen developers' mailing list (address below).

9.1 Other documentation

If you are mainly interested in using (rather than developing for) Xen, the *Xen Users' Manual* is distributed in the `docs/` directory of the Xen source distribution.

9.2 Online references

The official Xen web site can be found at:

    http://www.xensource.com

This contains links to the latest versions of all online documentation, including the latest version of the FAQ.

Information regarding Xen is also available at the Xen Wiki at

    http://wiki.xensource.com/xenwiki/

The Xen project uses Bugzilla as its bug tracking system. You'll find the Xen Bugzilla at http://bugzilla.xensource.com/bugzilla/.

9.3 Mailing lists

There are several mailing lists that are used to discuss Xen related topics. The most widely relevant are listed below. An official page of mailing lists and subscription information can be found at

`http://lists.xensource.com/`

xen-devel@lists.xensource.com Used for development discussions and bug reports. Subscribe at:
`http://lists.xensource.com/xen-devel`

xen-users@lists.xensource.com Used for installation and usage discussions and requests for help. Subscribe at:
`http://lists.xensource.com/xen-users`

xen-announce@lists.xensource.com Used for announcements only. Subscribe at:
`http://lists.xensource.com/xen-announce`

xen-changelog@lists.xensource.com Changelog feed from the unstable and 2.0 trees - developer oriented. Subscribe at:
`http://lists.xensource.com/xen-changelog`

Appendix A

Xen Hypercalls

Hypercalls represent the procedural interface to Xen; this appendix categorizes and describes the current set of hypercalls.

A.1 Invoking Hypercalls

Hypercalls are invoked in a manner analogous to system calls in a conventional operating system; a software interrupt is issued which vectors to an entry point within Xen. On x86/32 machines the instruction required is `int $82`; the (real) IDT is setup so that this may only be issued from within ring 1. The particular hypercall to be invoked is contained in `EAX` — a list mapping these values to symbolic hypercall names can be found in `xen/include/public/xen.h`.

On some occasions a set of hypercalls will be required to carry out a higher-level function; a good example is when a guest operating wishes to context switch to a new process which requires updating various privileged CPU state. As an optimization for these cases, there is a generic mechanism to issue a set of hypercalls as a batch:

> multicall(void *call_list, int nr_calls)
>
> Execute a series of hypervisor calls; `nr_calls` is the length of the array of `multicall_entry_t` structures pointed to be `call_list`. Each entry contains the hypercall operation code followed by up to 7 word-sized arguments.

Note that multicalls are provided purely as an optimization; there is no requirement to use them when first porting a guest operating system.

A.2 Virtual CPU Setup

At start of day, a guest operating system needs to setup the virtual CPU it is executing on. This includes installing vectors for the virtual IDT so that the guest OS can handle interrupts, page faults, etc. However the very first thing a guest OS must setup is a pair of hypervisor callbacks: these are the entry points which Xen will use when it wishes to notify the guest OS of an occurrence.

set_callbacks(unsigned long event_selector, unsigned long event_address, unsigned long failsafe_selector, unsigned long failsafe_address)

Register the normal ("event") and failsafe callbacks for event processing. In each case the code segment selector and address within that segment are provided. The selectors must have RPL 1; in XenLinux we simply use the kernel's CS for both **event_selector** and **failsafe_selector**.

The value **event_address** specifies the address of the guest OSes event handling and dispatch routine; the **failsafe_address** specifies a separate entry point which is used only if a fault occurs when Xen attempts to use the normal callback.

On x86/64 systems the hypercall takes slightly different arguments. This is because callback CS does not need to be specified (since teh callbacks are entered via SYSRET), and also because an entry address needs to be specified for SYSCALLs from guest user space:

set_callbacks(unsigned long event_address, unsigned long failsafe_address, unsigned long syscall_address)

After installing the hypervisor callbacks, the guest OS can install a 'virtual IDT' by using the following hypercall:

set_trap_table(trap_info_t *table)

Install one or more entries into the per-domain trap handler table (essentially a software version of the IDT). Each entry in the array pointed to by **table** includes the exception vector number with the corresponding segment selector and entry point. Most guest OSes can use the same handlers on Xen as when running on the real hardware.

A further hypercall is provided for the management of virtual CPUs:

vcpu_op(int cmd, int vcpuid, void *extra_args)

This hypercall can be used to bootstrap VCPUs, to bring them up and down and to test their current status.

A.3 Scheduling and Timer

Domains are preemptively scheduled by Xen according to the parameters installed by domain 0 (see Section A.10). In addition, however, a domain may choose to explicitly control certain behavior with the following hypercall:

sched_op_new(int cmd, void *extra_args)

Request scheduling operation from hypervisor. The following sub-commands are available:

SCHEDOP_yield voluntarily yields the CPU, but leaves the caller marked as runnable. No extra arguments are passed to this command.

SCHEDOP_block removes the calling domain from the run queue and causes it to sleep until an event is delivered to it. No extra arguments are passed to this command.

SCHEDOP_shutdown is used to end the calling domain's execution. The extra argument is a **sched_shutdown** structure which indicates the reason why the domain suspended (e.g., for reboot, halt, power-off).

SCHEDOP_poll allows a VCPU to wait on a set of event channels with an optional timeout (all of which are specified in the **sched_poll** extra argument). The semantics are similar to the UNIX **poll** system call. The caller must have event-channel upcalls masked when executing this command.

sched_op_new was not available prior to Xen 3.0.2. Older versions provide only the following hypercall:

sched_op(int cmd, unsigned long extra_arg)

This hypercall supports the following subset of **sched_op_new** commands:

SCHEDOP_yield (extra argument is 0).

SCHEDOP_block (extra argument is 0).

SCHEDOP_shutdown (extra argument is numeric reason code).

To aid the implementation of a process scheduler within a guest OS, Xen provides a virtual programmable timer:

set_timer_op(uint64_t timeout)

Request a timer event to be sent at the specified system time (time in nanoseconds since system boot).

Note that calling **set_timer_op** prior to **sched_op** allows block-with-timeout semantics.

A.4 Page Table Management

Since guest operating systems have read-only access to their page tables, Xen must be involved when making any changes. The following multi-purpose hypercall can be used to modify page-table entries, update the machine-to-physical mapping table, flush the TLB, install a new page-table base pointer, and more.

mmu_update(mmu_update_t *req, int count, int *success_count)

Update the page table for the domain; a set of **count** updates are submitted for processing in a batch, with **success_count** being updated to report the number of successful updates.

Each element of **req[]** contains a pointer (address) and value; the least significant 2-bits of the pointer are used to distinguish the type of update requested as follows:

MMU_NORMAL_PT_UPDATE: update a page directory entry or page table entry to the associated value; Xen will check that the update is safe, as described in Chapter 3.

MMU_MACHPHYS_UPDATE: update an entry in the machine-to-physical table. The calling domain must own the machine page in question (or be privileged).

Explicitly updating batches of page table entries is extremely efficient, but can require a number of alterations to the guest OS. Using the writable page table mode (Chapter 3) is recommended for new OS ports.

Regardless of which page table update mode is being used, however, there are some occasions (notably handling a demand page fault) where a guest OS will wish to modify exactly one PTE rather than a batch, and where that PTE is mapped into the current address space. This is catered for by the following:

update_va_mapping(unsigned long va, uint64_t val, unsigned long flags)

Update the currently installed PTE that maps virtual address **va** to new value **val**. As with **mmu_update**, Xen checks the modification is safe before applying it. The **flags** determine which kind of TLB flush, if any, should follow the update.

Finally, sufficiently privileged domains may occasionally wish to manipulate the pages of others:

update_va_mapping_otherdomain(unsigned long va, uint64_t val, unsigned long flags, domid_t domid)

Identical to **update_va_mapping** save that the pages being mapped must belong to the domain **domid**.

An additional MMU hypercall provides an "extended command" interface. This provides additional functionality beyond the basic table updating commands:

mmuext_op(struct mmuext_op *op, int count, int *success_count, domid_t domid)

This hypercall is used to perform additional MMU operations. These include updating `cr3` (or just re-installing it for a TLB flush), requesting various kinds of TLB flush, flushing the cache, installing a new LDT, or pinning & unpinning page-table pages (to ensure their reference count doesn't drop to zero which would require a revalidation of all entries). Some of the operations available are restricted to domains with sufficient system privileges.

It is also possible for privileged domains to reassign page ownership via an extended MMU operation, although grant tables are used instead of this where possible; see Section A.8.

Finally, a hypercall interface is exposed to activate and deactivate various optional facilities provided by Xen for memory management.

vm_assist(unsigned int cmd, unsigned int type)

Toggle various memory management modes (in particular writable page tables).

A.5 Segmentation Support

Xen allows guest OSes to install a custom GDT if they require it; this is context switched transparently whenever a domain is [de]scheduled. The following hypercall is effectively a 'safe' version of `lgdt`:

> set_gdt(unsigned long *frame_list, int entries)
>
> Install a global descriptor table for a domain; **frame_list** is an array of up to 16 machine page frames within which the GDT resides, with **entries** being the actual number of descriptor-entry slots. All page frames must be mapped read-only within the guest's address space, and the table must be large enough to contain Xen's reserved entries (see **xen/include/public/arch-x86_32.h**).

Many guest OSes will also wish to install LDTs; this is achieved by using **mmu_update** with an extended command, passing the linear address of the LDT base along with the number of entries. No special safety checks are required; Xen needs to perform this task simply since `lldt` requires CPL 0.

Xen also allows guest operating systems to update just an individual segment descriptor in the GDT or LDT:

> update_descriptor(uint64_t ma, uint64_t desc)
>
> Update the GDT/LDT entry at machine address **ma**; the new 8-byte descriptor is stored in **desc**. Xen performs a number of checks to ensure the descriptor is valid.

Guest OSes can use the above in place of context switching entire LDTs (or the GDT) when the number of changing descriptors is small.

A.6 Context Switching

When a guest OS wishes to context switch between two processes, it can use the page table and segmentation hypercalls described above to perform the the bulk of the privileged work. In addition, however, it will need to invoke Xen to switch the kernel (ring 1) stack pointer:

> stack_switch(unsigned long ss, unsigned long esp)
>
> Request kernel stack switch from hypervisor; **ss** is the new stack segment, which **esp** is the new stack pointer.

A useful hypercall for context switching allows "lazy" save and restore of floating point state:

fpu_taskswitch(int set)

This call instructs Xen to set the `TS` bit in the `cr0` control register; this means that the next attempt to use floating point will cause a trap which the guest OS can trap. Typically it will then save/restore the FP state, and clear the `TS` bit, using the same call.

This is provided as an optimization only; guest OSes can also choose to save and restore FP state on all context switches for simplicity.

Finally, a hypercall is provided for entering vm86 mode:

switch_vm86

This allows the guest to run code in vm86 mode, which is needed for some legacy software.

A.7 Physical Memory Management

As mentioned previously, each domain has a maximum and current memory allocation. The maximum allocation, set at domain creation time, cannot be modified. However a domain can choose to reduce and subsequently grow its current allocation by using the following call:

memory_op(unsigned int op, void *arg)

Increase or decrease current memory allocation (as determined by the value of **op**). The available operations are:

XENMEM_increase_reservation Request an increase in machine memory allocation; **arg** must point to a **xen_memory_reservation** structure.

XENMEM_decrease_reservation Request a decrease in machine memory allocation; **arg** must point to a **xen_memory_reservation** structure.

XENMEM_maximum_ram_page Request the frame number of the highest-addressed frame of machine memory in the system. **arg** must point to an **unsigned long** where this value will be stored.

XENMEM_current_reservation Returns current memory reservation of the specified domain.

XENMEM_maximum_reservation Returns maximum memory reservation of the specified domain.

In addition to simply reducing or increasing the current memory allocation via a 'balloon driver', this call is also useful for obtaining contiguous regions of machine memory when required (e.g. for certain PCI devices, or if using superpages).

A.8 Inter-Domain Communication

Xen provides a simple asynchronous notification mechanism via *event channels*. Each domain has a set of end-points (or *ports*) which may be bound to an event source (e.g. a physical IRQ, a virtual IRQ, or an port in another domain). When a pair of end-points in two different domains are bound together, then a 'send' operation on one will cause an event to be received by the destination domain.

The control and use of event channels involves the following hypercall:

event_channel_op(evtchn_op_t *op)

Inter-domain event-channel management; **op** is a discriminated union which allows the following 7 operations:

alloc_unbound: allocate a free (unbound) local port and prepare for connection from a specified domain.

bind_virq: bind a local port to a virtual IRQ; any particular VIRQ can be bound to at most one port per domain.

bind_pirq: bind a local port to a physical IRQ; once more, a given pIRQ can be bound to at most one port per domain. Furthermore the calling domain must be sufficiently privileged.

bind_interdomain: construct an interdomain event channel; in general, the target domain must have previously allocated an unbound port for this channel, although this can be bypassed by privileged domains during domain setup.

close: close an interdomain event channel.

send: send an event to the remote end of a interdomain event channel.

status: determine the current status of a local port.

For more details see **xen/include/public/event_channel.h**.

Event channels are the fundamental communication primitive between Xen domains and seamlessly support SMP. However they provide little bandwidth for communication *per se*, and hence are typically married with a piece of shared memory to produce effective and high-performance inter-domain communication.

Safe sharing of memory pages between guest OSes is carried out by granting access on a per page basis to individual domains. This is achieved by using the `grant_table_op` hypercall.

> grant_table_op(unsigned int cmd, void *uop, unsigned int count)
>
> Used to invoke operations on a grant reference, to setup the grant table and to dump the tables' contents for debugging.

A.9 IO Configuration

Domains with physical device access (i.e. driver domains) receive limited access to certain PCI devices (bus address space and interrupts). However many guest operating systems attempt to determine the PCI configuration by directly access the PCI BIOS, which cannot be allowed for safety.

Instead, Xen provides the following hypercall:

> physdev_op(void *physdev_op)
>
> Set and query IRQ configuration details, set the system IOPL, set the TSS IO bitmap.

For examples of using `physdev_op`, see the Xen-specific PCI code in the linux sparse tree.

A.10 Administrative Operations

A large number of control operations are available to a sufficiently privileged domain (typically domain 0). These allow the creation and management of new domains, for example. A complete list is given below: for more details on any or all of these, please see `xen/include/public/dom0_ops.h`

> dom0_op(dom0_op_t *op)
>
> Administrative domain operations for domain management. The options are:
>
> **DOM0_GETMEMLIST:** get list of pages used by the domain
>
> **DOM0_SCHEDCTL:**

DOM0_ADJUSTDOM: adjust scheduling priorities for domain

DOM0_CREATEDOMAIN: create a new domain

DOM0_DESTROYDOMAIN: deallocate all resources associated with a domain

DOM0_PAUSEDOMAIN: remove a domain from the scheduler run queue.

DOM0_UNPAUSEDOMAIN: mark a paused domain as schedulable once again.

DOM0_GETDOMAININFO: get statistics about the domain

DOM0_SETDOMAININFO: set VCPU-related attributes

DOM0_MSR: read or write model specific registers

DOM0_DEBUG: interactively invoke the debugger

DOM0_SETTIME: set system time

DOM0_GETPAGEFRAMEINFO:

DOM0_READCONSOLE: read console content from hypervisor buffer ring

DOM0_PINCPUDOMAIN: pin domain to a particular CPU

DOM0_TBUFCONTROL: get and set trace buffer attributes

DOM0_PHYSINFO: get information about the host machine

DOM0_SCHED_ID: get the ID of the current Xen scheduler

DOM0_SHADOW_CONTROL: switch between shadow page-table modes

DOM0_SETDOMAINMAXMEM: set maximum memory allocation of a domain

DOM0_GETPAGEFRAMEINFO2: batched interface for getting page frame info

DOM0_ADD_MEMTYPE: set MTRRs

DOM0_DEL_MEMTYPE: remove a memory type range

DOM0_READ_MEMTYPE: read MTRR

DOM0_PERFCCONTROL: control Xen's software performance counters

DOM0_MICROCODE: update CPU microcode

DOM0_IOPORT_PERMISSION: modify domain permissions for an IO port range (enable / disable a range for a particular domain)

DOM0_GETVCPUCONTEXT: get context from a VCPU

DOM0_GETVCPUINFO: get current state for a VCPU

DOM0_GETDOMAININFOLIST: batched interface to get domain info

DOM0_PLATFORM_QUIRK: inform Xen of a platform quirk it needs to handle (e.g. noirqbalance)

DOM0_PHYSICAL_MEMORY_MAP: get info about dom0's memory map

DOM0_MAX_VCPUS: change max number of VCPUs for a domain

DOM0_SETDOMAINHANDLE: set the handle for a domain

Most of the above are best understood by looking at the code implementing them (in `xen/common/dom0_ops.c`) and in the user-space tools that use them (mostly in `tools/libxc`).

A.11 Access Control Module Hypercalls

Hypercalls relating to the management of the Access Control Module are also restricted to domain 0 access for now. For more details on any or all of these, please see `xen/include/public/acm_ops.h`. A complete list is given below:

acm_op(int cmd, void *args)

This hypercall can be used to configure the state of the ACM, query that state, request access control decisions and dump additional information.

ACMOP_SETPOLICY: set the access control policy

ACMOP_GETPOLICY: get the current access control policy and status

ACMOP_DUMPSTATS: get current access control hook invocation statistics

ACMOP_GETSSID: get security access control information for a domain

ACMOP_GETDECISION: get access decision based on the currently enforced access control policy

Most of the above are best understood by looking at the code implementing them (in `xen/common/acm_ops.c`) and in the user-space tools that use them (mostly in `tools/security` and `tools/python/xen/lowlevel/acm`).

A.12 Debugging Hypercalls

A few additional hypercalls are mainly useful for debugging:

console_io(int cmd, int count, char *str)

Use Xen to interact with the console; operations are:

CONSOLEIO_write: Output count characters from buffer str.

CONSOLEIO_read: Input at most count characters into buffer str.

A pair of hypercalls allows access to the underlying debug registers:

set_debugreg(int reg, unsigned long value)

Set debug register **reg** to **value**

get_debugreg(int reg)

Return the contents of the debug register **reg**

And finally:

xen_version(int cmd)

Request Xen version number.

This is useful to ensure that user-space tools are in sync with the underlying hypervisor.