Interface manual

Xen v2.0 for x86

Xen is Copyright (c) 2002-2004, The Xen Team

University of Cambridge, UK

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

On a Xen-based system, the hypervisor itself runs in *ring 0*. It has full access to the physical memory available in the system and is responsible for allocating portions of it to the domains. Guest operating systems run in and 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.

In this chapter we consider the basic virtual architecture provided by Xen: the basic 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. Most trap handlers are identical to native x86 handlers, although the page-fault handler is somewhat different.

2.3 Interrupts and events

Interrupts are virtualized by mapping them to *events*, 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 events, see Chapter 4.

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* two version numbers are also stored in the shared info page. The first is incremented prior to an update, while the second is only incremented afterwards. Thus a guest can be sure that it read a consistent state by checking the two version numbers are equal.

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 BVT, Atropos and Round Robin schedulers are part of the normal Xen distribution. BVT provides proportional fair shares of the CPU to the running domains. Atropos can be used to reserve absolute shares of the CPU for each domain. Round-robin is provided as an example of Xen's internal scheduler API.

Note: SMP host support   Xen has always supported SMP host systems. Domains are statically assigned to CPUs, either at creation time or when manually pinning to a particular CPU. The current schedulers then run locally on each CPU to decide which of the assigned domains should be run there. The user-level control software can be used to perform coarse-grain load-balancing between CPUs.

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

Xen resides within a small fixed portion of physical memory; it also reserves the top 64MB of every virtual address space. The remaining 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 4K *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.

Xen also provides an alternative mode of operation in which guests be 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.

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.

3.4   Segment Descriptor Tables

On boot 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. Note that a number of GDT entries are reserved by Xen – any custom GDT must also include sufficient space for these entries.

For example, 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 16 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).
>
> *entries*: The number of descriptor-entry slots in the GDT. Note that the table must be large enough to contain Xen's reserved entries; thus we must have '*entries* > *LAST_RESERVED_GDT_ENTRY* '. Note also that, after registering the GDT, slots *FIRST_* through *LAST_RESERVED_GDT_ENTRY* are no longer usable by the guest and may be overwritten by Xen.

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

3.5   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.

Chapter 4

Devices

Devices such as network and disk are exported to guests using a split device driver. The device driver domain, which accesses the physical device directly also runs a *backend* driver, serving requests to that device from guests. Each guest will use a simple *frontend* driver, to access the backend. Communication between these domains is composed of two parts: First, data is placed onto a shared memory page between the domains. Second, an event channel between the two domains is used to pass notification that data is outstanding. This separation of notification from data transfer allows message batching, and results in very efficient device access.

Event channels are used extensively in device virtualization; each domain has a number of end-points or *ports* each of which may be bound to one of the following *event sources*:

- a physical interrupt from a real device,
- a virtual interrupt (callback) from Xen, or
- a signal from another domain

Events are lightweight and do not carry much information beyond the source of the notification. Hence when performing bulk data transfer, events are typically used as synchronization primitives over a shared memory transport. Event channels are managed via the `event_channel_op()` hypercall; for more details see Section A.8.

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

4.1   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.

4.1.1 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.

4.1.2 Data Transfer

Each virtual interface uses two "descriptor rings", one for transmit, the other for receive. Each descriptor identifies a block of contiguous physical 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 overload 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.

4.2 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, across which read and write messages 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.

Old (Xen 1.2) virtual disks are not supported under Xen 2.0, since similar functionality can be achieved using the more complete LVM system, which is already in widespread use.

4.2.1 Data Transfer

The single ring between the guest and the block backend supports three messages:

PROBE: Return a list of the VBDs available to this guest from the backend. The request includes a descriptor of a free page into which the reply will be written by the backend.

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.

Chapter 5

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).

5.1 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.

5.2 Online references

The official Xen web site is found at:

```
http://www.cl.cam.ac.uk/Research/SRG/netos/xen/
```

This contains links to the latest versions of all on-line documentation.

5.3 Mailing lists

There are currently four official Xen mailing lists:

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

Of these, xen-devel is the most active.

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.

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; an exception is the page fault handler (exception vector 14) where a modified stack-frame layout is used.

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(unsigned long op)

> Request scheduling operation from hypervisor. The options are: *yield*, *block*, and *shutdown*. *yield* keeps the calling domain runnable but may cause a reschedule if other domains are runnable. *block* removes the calling domain from the run queue and cause is to sleeps until an event is

delivered to it. *shutdown* is used to end the domain's execution; the caller can additionally specify whether the domain should reboot, halt or suspend.

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). The hypercall actually passes the 64-bit timeout value as a pair of 32-bit values.

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).

MMU_EXTENDED_COMMAND: perform additional MMU operations. The set of additional MMU operations is considerable, and includes updating cr3 (or just re-installing it for a 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).

19

> Further extended commands are used to deal with granting and ac-
> quiring page ownership; see Section A.8.

> More details on the precise format of all commands can be found in
> `xen/include/public/xen.h`.

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 mod-
ify exactly one PTE rather than a batch. This is catered for by the following:

> update_va_mapping(unsigned long page_nr, unsigned long val,
> unsigned long flags)

> Update the currently installed PTE for the page `page_nr` to `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 page_nr, unsigned
> long val, unsigned long flags, uint16_t domid)

> Identical to `update_va_mapping()` save that the pages being mapped
> must belong to the domain `domid`.

This privileged operation is currently used by backend virtual device drivers to safely
map pages containing I/O data.

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(unsigned long ma, unsigned long word1, unsigned long word2)
>
> Update the GDT/LDT entry at machine address `ma`; the new 8-byte descriptor is stored in `word1` and `word2`. 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 final useful hypercall for context switching allows "lazy" save and restore of floating point state:

> fpu_taskswitch(void)
>
> 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.

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

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:

dom_mem_op(unsigned int op, unsigned long *extent_list, unsigned long nr_extents, unsigned int extent_order)

Increase or decrease current memory allocation (as determined by the value of op). Each invocation provides a list of extents each of which is 2^s pages in size, where s is the value of extent_order.

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)

Grant or remove access to a particular page to a particular domain.

This is not currently widely in use by guest operating systems, but we intend to integrate support more fully in the near future.

A.9   PCI 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)

Perform a PCI configuration option; depending on the value of `phys-dev_op` this can be a PCI config read, a PCI config write, or a small number of other queries.

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_CREATEDOMAIN: create a new domain

DOM0_PAUSEDOMAIN: remove a domain from the scheduler run queue.

DOM0_UNPAUSEDOMAIN: mark a paused domain as schedulable once again.

DOM0_DESTROYDOMAIN: deallocate all resources associated with a domain

DOM0_GETMEMLIST: get list of pages used by the domain

DOM0_SCHEDCTL:

DOM0_ADJUSTDOM: adjust scheduling priorities for domain

DOM0_BUILDDOMAIN: do final guest OS setup for domain

DOM0_GETDOMAINFO: get statistics about the domain

DOM0_GETPAGEFRAMEINFO:

DOM0_GETPAGEFRAMEINFO2:

DOM0_IOPL: set I/O privilege level

DOM0_MSR: read or write model specific registers

DOM0_DEBUG: interactively invoke the debugger

DOM0_SETTIME: set system time

DOM0_READCONSOLE: read console content from hypervisor buffer ring

DOM0_PINCPUDOMAIN: pin domain to a particular CPU

DOM0_GETTBUFS: get information about the size and location of the trace buffers (only on trace-buffer enabled builds)

DOM0_PHYSINFO: get information about the host machine

DOM0_PCIDEV_ACCESS: modify PCI device access permissions

DOM0_SCHED_ID: get the ID of the current Xen scheduler

DOM0_SHADOW_CONTROL: switch between shadow page-table modes

DOM0_SETDOMAININITIALMEM: set initial memory allocation of a domain

DOM0_SETDOMAINMAXMEM: set maximum memory allocation of a domain

DOM0_SETDOMAINVMASSIST: set domain VM assist options

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 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.

A.12 Deprecated Hypercalls

Xen is under constant development and refinement; as such there are plans to improve the way in which various pieces of functionality are exposed to guest OSes.

> vm_assist(unsigned int cmd, unsigned int type)
>
> Toggle various memory management modes (in particular wrritable page tables and superpage support).

This is likely to be replaced with mode values in the shared information page since this is more resilient for resumption after migration or checkpoint.