Xen Documentation

Release 4.13.1-pre

The Xen development community

Jan 14, 2020

Contents

Note: Xen's Sphinx/RST documentation is a work in progress. The existing documentation can be found at https://xenbits.xen.org/docs/

Xen is an open source, bare metal hypervisor. It runs as the most privileged piece of software on the system, and shares the resources of the hardware between virtual machines. See *Introduction* for an introduction to a Xen system.

CHAPTER 1

User documentation

This is documentation for an administrator of a Xen system. It is intended for someone who is not necesserily a developer, has installed Xen from their preferred distribution, and is attempting to run virtual machines and configure the system.

1.1 Admin Guide

1.1.1 Introduction

Xen is an open source, bare metal hypervisor. It runs as the most privileged piece of software, and shares the resources of the hardware between virtual machines.

In Xen terminology, there are *domains*, commonly abbreviated to dom, which are identified by their numeric *domid*.

When Xen boots, dom0 is automatically started as well. Dom0 is a virtual machine which, by default, is granted full permissions[1]. A typical setup might be:

Dom0 takes the role of *control domain*, responsible for creating and managing other virtual machines, and the role of *hardware domain*, responsible for hardware and marshalling guest I/O.

Xen is deliberately minimal, and has no device drivers[2]. Xen manages RAM, schedules virtual CPUs on the available physical CPUs, and marshals interrupts.

Xen also provides a hypercall interface to guests, including event channels (virtual interrupts), grant tables (shared memory), on which a lot of higher level functionality is built.

[1] A common misconception with Xen's architecture is that dom0 is somehow different to other guests. The choice of id 0 is not an accident, and follows in UNIX heritage.

[2] This definition might be fuzzy. Xen can talk to common serial UARTs, and knows how to drive various CPU internal devices such as IOMMUs, but has no knowledge of network cards, disks, etc. All of that is the hardware domains responsibility.

1.1.2 Microcode Loading

Like many other pieces of hardware, CPUs themselves have errata which are discovered after shipping, and need to be addressed in the field. Microcode can be considered as firmware for the processor, and updates are published as needed by the CPU vendors.

Microcode is included as part of the system firmware by an OEM, and a system firmware update is the preferred way of obtaining updated microcode. However, this is often not the most expedient way to get updates, so Xen supports loading microcode itself.

Distros typically package microcode updates for users, and may provide hooks to cause microcode to be automatically loaded at boot time. Consult your dom0 distro guidance for microcode loading.

Microcode can make almost arbitrary changes to the processor, including to software visible features. This includes removing features (e.g. the Haswell TSX errata which necessitated disabling the feature entirely), or the addition of brand new features (e.g. the Spectre v2 controls to work around speculative execution vulnerabilities).

Boot time microcode loading

Where possible, microcode should be loaded at boot time. This allows the CPU to be updated to its eventual configuration before Xen starts making setup decisions based on the visible features.

Xen will report during boot if it performed a microcode update:

```
[root@host ~]# xl dmesg | grep microcode
(XEN) microcode: CPU0 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU2 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU4 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU6 updated from revision 0x1a to 0x25, date = 2018-04-02
```

The exact details printed are system and microcode specific. After boot, the current microcode version can obtained from with dom0:

```
[root@host ~]# head /proc/cpuinfo
processor    : 0
vendor_id    : GenuineIntel
cpu family   : 6
model        : 60
model name   : Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz
stepping     : 3
microcode    : 0x25
cpu MHz      : 3392.148
cache size   : 8192 KB
physical id  : 0
```

Loading microcode from a single file

Xen handles microcode blobs in the binary form shipped by vendors, which is also the format which the processor accepts. This format contains header information which Xen and various userspace tools can use to identify the correct blob for a specific CPU.

Tools such as Dracut will identify the correct blob for the current CPU, which will be a few kilobytes, for minimal overhead during boot.

Additionally, Xen is capable of handling a number of blobs concatenated together, and will locate the appropriate blob based on the header information.

This option is less efficient during boot, but may be preferred in situations where the exact CPU details aren't known ahead of booting (e.g. install media).

The file containing the blob(s) needs to be accessible to Xen as early as possible.

- For multiboot/multiboot2 boots, this is achieved by loading the file as a multiboot module. The `ucode=$num` command line option can be used to identify which multiboot module contains the microcode, including negative indexing to count from the end.
- For EFI boots, there isn't really a concept of modules. A microcode file can be specified in the EFI configuration file with `ucode=$file`. Use of this mechanism will override any `ucode=` settings on the command line.

Loading microcode from a Linux initrd

For systems using a Linux based dom0, it usually suffices to install the appropriate distro package, and add `ucode=scan` to Xen's command line.

Xen is compatible with the Linux initrd microcode protocol. The initrd is expected to be generated with an uncompressed CPIO archive at the beginning which contains contains one of these two files:

```
kernel/x86/microcode/GenuineIntel.bin
kernel/x86/microcode/AuthenticAMD.bin
```

The `ucode=scan` command line option will cause Xen to search through all modules to find any CPIO archives, and search the archive for the applicable file. Xen will stop searching at the first match.

Run time microcode loading

Warning: If at all possible, microcode updates should be done by firmware updates, or at boot time. Not all microcode updates (or parts thereof) can be applied at runtime.

The `xen-ucode` utility can be used to initiate a runtime microcode load. It will pass the blob to Xen, which will check to see whether the blob is correct for the processor, and newer than the running microcode.

If these checks pass, the entire system will be rendezvoused and an update will be initiated on all CPUs in parallel. As with boot time loading, diagnostics will be put out onto the console:

```
[root@host ~]# xl dmesg | grep microcode
(XEN) microcode: CPU0 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU2 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU4 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU6 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) 4 cores are to update their microcode
(XEN) microcode: CPU0 updated from revision 0x25 to 0x27, date = 2019-02-26
(XEN) microcode: CPU4 updated from revision 0x25 to 0x27, date = 2019-02-26
(XEN) microcode: CPU2 updated from revision 0x25 to 0x27, date = 2019-02-26
(XEN) microcode: CPU6 updated from revision 0x25 to 0x27, date = 2019-02-26
```

CHAPTER 2

Guest documentation

This documentation concerns the APIs and ABIs available to guests. It is intended for OS developers trying to use a Xen feature, and for Xen developers to avoid breaking things.

2.1 Guest documentation

2.1.1 x86

Hypercall ABI

Hypercalls are system calls to Xen. Two modes of guest operation are supported, and up to 6 individual parameters are supported.

Hypercalls may only be issued by kernel-level software[1].

Registers

The registers used for hypercalls depends on the operating mode of the guest.

ABI	Hypercall Index	Parameters (1 - 6)	Result
64bit	RAX	RDI RSI RDX R10 R8 R9	RAX
32bit	EAX	EBX ECX EDX ESI EDI EBP	EAX

32 and 64bit PV guests have an ABI fixed by their guest type. The ABI for an HVM guest depends on whether the vCPU is operating in a 64bit segment or not[2].

[1] For HVM guests, `HVMOP_guest_request_vm_event` may be configured to be usable from userspace, but this behaviour is not default.

[2] While it is possible to use compatibility mode segments in a 64bit kernel, hypercalls issues from such a mode will be interpreted with the 32bit ABI. Such a setup is not expected in production scenarios.

Parameters

Different hypercalls take a different number of parameters. Each hypercall potentially clobbers each of its parameter registers; a guest may not rely on the parameter registers staying the same. A debug build of Xen checks this by deliberately poisoning the parameter registers before returning back to the guest.

Mode transfer

The exact sequence of instructions required to issue a hypercall differs between virtualisation mode and hardware vendor.

Guest	Transfer instruction
32bit PV	INT 0x82
64bit PV	SYSCALL
Intel HVM	VMCALL
AMD HVM	VMMCALL

To abstract away the details, Xen implements an interface known as the Hypercall Page. This allows a guest to make a hypercall without needing to perform mode-specific or vendor-specific setup.

Hypercall Page

The hypercall page is a page of guest RAM into which Xen will write suitable transfer stubs.

Creating a hypercall page is an isolated operation from Xen's point of view. It is the guests responsibility to ensure that the hypercall page, once written by Xen, is mapped with executable permissions so it may be used. Multiple hypercall pages may be created by the guest, if it wishes.

The stubs are arranged by hypercall index, and start on 32-byte boundaries. To invoke a specific hypercall, `call` the relevant stub[3]:

```
call hypercall_page + index * 32
```

There result is an ABI which is invariant of the exact operating mode or hardware vendor. This is intended to simplify guest kernel interfaces by abstracting away the details of how it is currently running.

Creating Hypercall Pages

Guests which are started using the PV boot protocol may set set `XEN_ELFNOTE_HYPERCALL_PAGE` to have the nominated page written as a hypercall page during construction. This mechanism is common for PV guests, and allows hypercalls to be issued with no additional setup.

Any guest can locate the Xen CPUID leaves and read the *hypercall transfer page* information, which specifies an MSR that can be used to create additional hypercall pages. When a guest physical address is written to the MSR, Xen writes a hypercall page into the nominated guest page. This mechanism is common for HVM guests which are typically started via legacy means.

[3] `HYPERCALL_iret` is special. It is only implemented for PV guests and takes all its parameters on the stack. This stub should be `jmp`'d to, rather than `call`'d. HVM guests have this stub implemented as `ud2a` to prevent accidental use.

CHAPTER 3

Hypervisor developer documentation

This is documentation for a hypervisor developer. It is intended for someone who is building Xen from source, and is running the new hypervisor in some kind of development environment.

3.1 Hypervisor documentation

3.1.1 Code Coverage

Xen can be compiled with coverage support. When configured, Xen will record the coverage of its own basic blocks. Being a piece of system software rather than a userspace, it can't automatically write coverage out to the filesystem, so some extra steps are required to collect and process the data.

> **Warning:** ARM doesn't currently boot when the final binary exceeds 2MB in size, and the coverage build tends to exceed this limit.

Compiling Xen

Coverage support is dependent on the compiler and toolchain used. As Xen isn't a userspace application, it can't use the compiler supplied library, and instead has to provide some parts of the implementation itself.

For x86, coverage support was introduced with GCC 3.4 or later, and Clang 3.9 or later, and Xen is compatible with these. However, the compiler internal formats do change occasionally, and this may involve adjustments to Xen. While we do our best to keep up with these changes, Xen may not be compatible with bleeding edge compilers.

To build with coverage support, enable `CONFIG_COVERAGE` in Kconfig. The build system will automatically select the appropriate format based on the compiler in use.

The resulting binary will record its own coverage while running.

Accessing the raw coverage data

The `SYSCTL_coverage_op` hypercall is used to interact with the coverage data. A dom0 userspace helper, `xenconv` is provided as well, which thinly wraps this hypercall.

The `read` subcommand can be used to obtain the raw coverage data:

```
[root@host ~]# xencov read > coverage.dat
```

This is toolchain-specific data and needs to be fed back to the appropriate programs to post-process.

Alternatively, the `reset` subcommand can be used reset all counters back to 0:

```
[root@host ~]# xencov reset
```

GCC coverage

A build using GCC's coverage will result in `*.gcno` artefact for every object file. The raw coverage data needs splitting to form the matching `*.gcda` files.

An example of how to view the data is as follows. It uses `lcov` which is a graphical frontend to `gcov`.

- Obtain the raw coverage data from the test host, and pull it back to the build working tree.
- Use `xencov_split` to extract the `*.gcda` files. Note that full build paths are used by the tools, so splitting needs to output relative to `/`.
- Use `geninfo` to post-process the raw data.
- Use `genhtml` to render the results as HTML.
- View the results in a browser.

```
xen.git/xen$ ssh root@host xencov read > coverage.dat
xen.git/xen$ ../tools/xencov_split coverage.dat --output-dir=/
xen.git/xen$ geninfo . -o cov.info
xen.git/xen$ genhtml cov.info -o cov/
xen.git/xen$ $BROWSER cov/index.html
```

Clang coverage

An example of how to view the data is as follows.

- Obtain the raw coverage data from the test host, and pull it back to the build working tree.
- Use `llvm-profdata` to post-process the raw data.
- Use `llvm-cov show` in combination with `xen-syms` from the build to render the results as HTML.
- View the results in a browser.

```
xen.git/xen$ ssh root@host xencov read > xen.profraw
xen.git/xen$ llvm-profdata merge xen.profraw -o xen.profdata
xen.git/xen$ llvm-cov show -format=html -output-dir=cov/ xen-syms -instr-profile=xen.
↪profdata
xen.git/xen$ $BROWSER cov/index.html
```

Full documentation on Clang's coverage capabilities can be found at: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

CHAPTER 4

Miscellanea

4.1 Glossary

control domain A *domain*, commonly dom0, with the permission and responsibility to create and manage other domains on the system.

domain A domain is Xen's unit of resource ownership, and generally has at the minimum some RAM and virtual CPUs.

The terms *domain* and *guest* are commonly used interchangeably, but they mean subtly different things.

A guest is a single, end user, virtual machine.

In some cases, e.g. during live migration, one guest will be comprised of two domains for a period of time, while it is in transit.

domid The numeric identifier of a running *domain*. It is unique to a single instance of Xen, used as the identifier in various APIs, and is typically allocated sequentially from 0.

guest The term 'guest' has two different meanings, depending on context, and should not be confused with *domain*.

When discussing a Xen system as a whole, a 'guest' refer to a virtual machine which is the "useful output" of running the system in the first place (e.g. an end-user VM). Virtual machines providing system services, (e.g. the control and/or hardware domains), are not considered guests in this context.

In the code, "guest context" and "guest state" is considered in terms of the CPU architecture, and contrasted against hypervisor context/state. In this case, it refers to all code running lower privilege privilege level the hypervisor. As such, it covers all domains, including ones providing system services.

hardware domain A *domain*, commonly dom0, which shares responsibility with Xen about the system as a whole.

By default it gets all devices, including all disks and network cards, so is responsible for multiplexing guest I/O.

Index