FreeBSD/Xen Tutorial

Roger Pau Monné royger@FreeBSD.org
Feedback by Andrew Cooper andrew.cooper3@citrix.com
Citrix Systems R&D

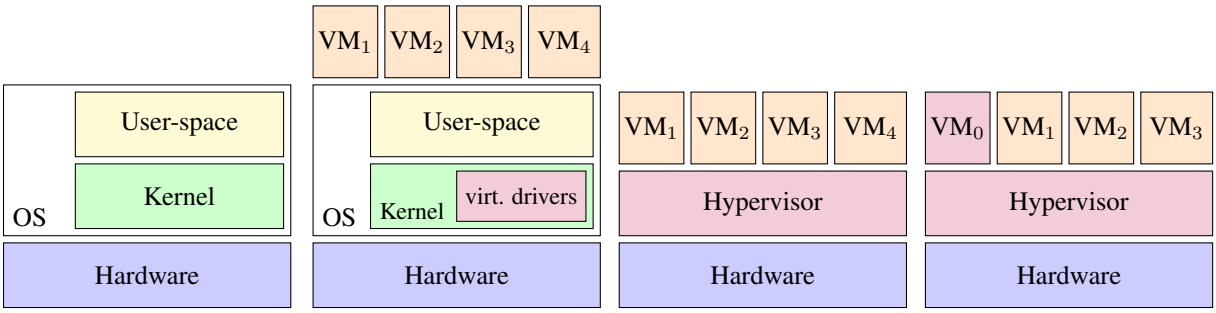
21-24th of March, 2019

Abstract

This article provides detailed instructions about how to setup a FreeBSD/Xen dom0 (host). It also describes how to create and manage guests, including detailed instructions about how to install and configure some of them.

1 Introduction

Let's start by taking a look at the different hypervisor architectures:



(a) Basic OS (b) Type II Hypervisor (c) Type I Hypervisor (d) Xen Hypervisor

Figure 1: Hypervisor architectures

Xen is a Type I hypervisor with a twist, and that's because part of the hypervisor functionality is implemented in VM_0, also know as dom0, which is the first guest that's created when Xen is booted. This functionality offloading allows Xen the hypervisor code base to be very minimal, but still enough to fully manage the hardware virtualization extensions and part of the platforms devices, like the IO-APICs, the local APICs or the IOMMUs present on the system. One of the benefits of such architecture is that the number of hardware drivers inside the hypervisor is very limited, since most of the hardware is passed-through to dom0 and not used by the hypervisor at all.

For this tutorial we are going to exclusively use a FreeBSD dom0, but note that dom0 supported OSes include Linux (in PV and PVH modes) and NetBSD (in PV mode). Bromium, a company selling a security product based on Xen, has modified it to be able to use Windows as dom0.

Let's take a closer look at the Xen architecture, and the role of dom0.

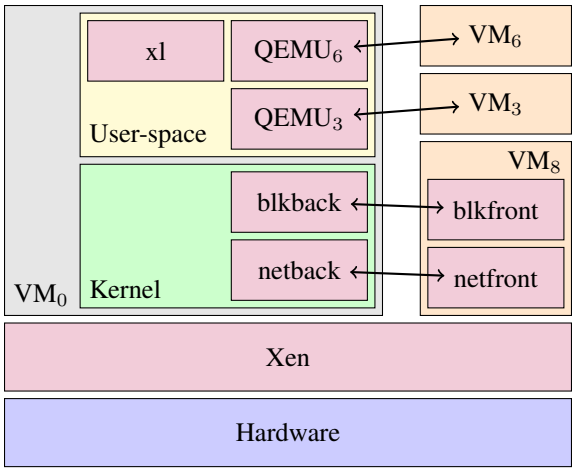


Figure 2: Xen architecture

2 Required materials

In order to setup a FreeBSD/Xen dom0 a box with a functional IOMMU is required. Almost all recent Intel Core chips have this feature and it has been present on the Xeon series for longer. Regarding AMD hardware, PVH dom0 is known to work on the Zen architecture, and likely older architectures that have a functional IOMMU.

It is recommended that the box used has serial output in order to debug possible problems with Xen and FreeBSD. Without a serial cable it is almost impossible to diagnose early boot problems.

Note that in order to boot Xen from FreeBSD legacy BIOS booting must be used. This is because multiboot2 has not yet been implemented for the FreeBSD loader.

The hardware used for the demo will be an Intel NUC5i3MYHE:

- Intel Core i3-5010U.
- 8GB of RAM.
- 500GB Hard drive.
- Intel 10/100/1000Mbps Ethernet card.
- Serial output.

The following materials are also needed in order to perform the installation:

- mfsBSD image: used to boot the system and install FreeBSD.
- FreeBSD installation sets.

One way to install the host is to use the materials that have been produced by the Xen test system (osstest) as part of the Xen continuous integration framework. The FreeBSD test run on a periodic basis, and produce new installation

images and sets in order to test FreeBSD HEAD and Xen unstable (the equivalent of FreeBSD HEAD in Xen speak). The result of those tests are send to xen-devel periodically, and the status email contains a link that can be used to fetch the images.

Another option is to use the mfsBSD RELEASE images, that can be fetched from `https://mfsbsd.vx.sk`.

Regardless of the version choose, before installing FreeBSD the mfsBSD image has to be copied to an USB drive using dd:

```
# dd if=/path/to/mfsbsd.img of=/path/of/usb/dev bs=1m
```

An Internet connection will also be needed in order to perform the tutorial.

3 Installing FreeBSD

The bare metal install of FreeBSD will be done using mfsBSD and the zfsinstall script. In the opinion of the author this is one of the fastest and simplest ways to setup a FreeBSD systems with ZFS-on-Root.

The first step is to boot the box using the USB drive that has been flashed with the mfsBSD image. The mfsBSD image provided has been configured to use both the video adapter and the serial console, so installation can be done from either input methods. The first step is to install FreeBSD on the hard drive using zfsinstall:

```
# gpart destroy -F ada0
# zfsinstall -d ada0 -u ftp://ftp.freebsd.org/pub/FreeBSD/releases/amd64/12.0-RELEASE/\
             -s 4g
```

Once the process finishes we would like to perform some modifications to the installed system before rebooting. The first step is enabling serial output, so we need to add the following to `/boot/loader.conf`:

```
boot_serial="YES"
comconsole_speed="115200"
console="comconsole"
boot_verbose="YES"
beastie_disable="YES"
```

And the following to `/boot.config`:

```
-h -S115200
```

Now we are going to configure the network interface in order to have network connectivity and automatically start sshd on boot. We need to add the following to `/etc/rc.conf`:

```
cloned_interfaces="bridge0"
ifconfig_bridge0="addm em0 SYNCDHCP"
ifconfig_em0="up"
```

Note that we are creating a bridge and adding the em0 interface to it. This is required so that later on we can give connectivity to the guests we create.

Since this is a test system, and we haven't created any users, we are also going to enable ssh root logins by adding the following line to `/etc/ssh/sshd_config`:

```
PermitRootLogin yes
```

I will also add my public key in order to ease the login procedure.

All the above actions can be placed in a install script, so they are automatically executed when the install finishes:

```
set -a
BSDINSTALL_DISTDIR="/root/sets/"
ZFSBOOT_DISKS="ada0"
DISTRIBUTIONS="base.txz kernel.txz"
nonInteractive=1

#!/bin/sh
set -ex

# Setup nic and sshd
sysrc cloned_interfaces="bridge0"
sysrc ifconfig_bridge0="addm em0 SYNCDHCP"
sysrc ifconfig_em0="up"
sysrc sshd_enable="YES"

# Allow root user login and setup ssh keys
chsh -s /bin/sh root
echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
mkdir -p /root/.ssh
cat << ENDKEYS > /root/.ssh/authorized_keys
YOUR-PUBLIC-KEY
ENDKEYS

# Setup serial console
printf "%s" "-h -S115200" >> /boot.config
cat << ENDBOOT >> /boot/loader.conf
boot_serial="YES"
comconsole_speed="115200"
console="comconsole"
boot_verbose="YES"
beastie_disable="YES"
# Assign first interface MAC address to the bridge
net.link.bridge.inherit_mac=1
ENDBOOT
```

The install script can be executed using:

```
# bsdinstall script installscript
```

Now we can reboot into the newly installed system in order to install Xen.

4 Installing Xen

For this tutorial we are going to use the Xen packages available in the ports system, currently using Xen version 4.11.1. Later on Xen unstable will also be installed from source in order to see how Xen is built for FreeBSD.

Installing is as simple as:

```
# pkg bootstrap
# pkg update
# pkg install xen-kernel xen-tools
```

Alternatively, it can be built and installed from the ports tree:

```
# portsnap fetch extract
# cd /usr/ports/emulators/xen-kernel
# pkg install -A `make build-depends-list |sed 's,^/usr/ports/,,'`
# make install
# cd /usr/ports/sysutils/xen-tools
# pkg install -A `make build-depends-list |sed 's,^/usr/ports/,,'`
# make install
```

Then we will proceed with some basic configuration steps in order to boot into our FreeBSD Xen dom0. This first step is adding the following lines to `/boot/loader.conf` so the FreeBSD loader will detect the Xen kernel and pass an appropriate command line to it.

```
xen_cmdline="dom0_mem=4g dom0=pvh,verbose com1=115200,8n1 console=com1 iommu=debug \
             guest_loglvl=all loglvl=all"
xen_kernel="/boot/xen"

vfs.zfs.arc_max="1G"
if_tap_load="YES"
```

In this example we are giving 4GB to dom0, and we set a couple of debug options in order to get more verbose output from Xen.

It is also recommended to remove the limit on the number of wired pages, since the Xen toolstack makes use of them, this is done by modifying `/etc/sysctl.conf`:

```
vm.max_wired=-1
```

And finally we need to enable the xencommons init script in `/etc/rc.conf` and enable the Xen console in `/etc/ttys`:

```
# sysrc xencommons_enable="YES"
# echo 'xc0 "/usr/libexec/getty Pc" xterm onifconsole secure' >> /etc/ttys
```

The installation is now finished and we can reboot into the FreeBSD Xen dom0.

5 The Xen toolstack

The default toolstack that comes with Xen is called xl. This is a CLI utility that can be used to manage guests running on the system. One of the first commands that we can try is:

```
# xl list
Name                                ID   Mem VCPUs      State       Time(s)
Domain-0                            0   4095     4      r-----      14.5
```

This outputs a list of the guest that are currently running. As we can see we only have one guest that's the FreeBSD dom0. The list of all supported xl commands can be fetched using `xl help`.

5.1 Guest creation

This section contains examples about how to setup some common guest types.

Xen supports different guests types, each with a different set of features and interfaces. It's important to notice HVM is the only guest type that can run completely unmodified guest OSes, other modes will requires modifications to the guest OS, and awareness it's running as a Xen guest.

	Disk and network	Interrupts and timers	Platform devices	Privileged instructions	Page table managing
HVM	Software emulated	Software emulated	Software emulated	Hardware extensions	Hardware extensions
HVM with PV drivers	Paravirtualized	Software emulated	Software emulated	Hardware extensions	Hardware extensions
PVHVM	Paravirtualized	Paravirtualized Hardware extensions	Software emulated	Hardware extesions	Hardware extensions
PVH	Paravirtualized	Paravirtualized Hardware extensions	Not available	Hardware extensions	Hardware extensions
PV	Paravirtualized	Paravirtualized	Not available	Paravirtualized	Paravirtualized

Figure 3: Xen guest spectrum

When Xen was conceived there where no hardware virtualization extensions on x86, so the cost of virtualization was very high. In order avoid this performance penalty Xen required guest OSes to be modified in order to be aware they are running virtualized so different interfaces are used in order to interact with the CPU, MMU or other devices. This kind of cooperation is what is called paravirtualization or PV. This technique is also known as "ring deprivileging": the operating system that runs originally on ring 0 is moved to another less privileged ring like ring 1. This allows the hypervisor to control the guest OS access to resources.

When x86 CPUs gained support for hardware assisted virtualization (Intel VT-x or AMD Secure Virtual Machine)

Xen implemented a new virtulization mode, called HVM, which doesn't require the OS to be aware it's running virtualized. Such mode also requires a device model (QEMU) in order to emulate the devices found on PC-compatible hardware. Note that some performance critical devices (IO-APIC, local APIC, HPET, PIT...) are emulated inside of the hypervisor instead of QEMU.

Finally Xen recently gained support for a new virtualization mode called PVH, which is very similar to HVM but doesn't require QEMU, and thus doesn't provide many of the emulated devices available to HVM guests. Due to such lack of devices the interface exposed to PVH guests is not a PC-compatible environment, but could be seen as a legacy-free x86 environment.

Let's start by creating a PV, HVM and PVH guests on our newly installed system.

5.2 Debian PV guest

In order to setup a pure Linux PV guest we are going to use Debian. Debian already provides a kernel and initramfs that can be used to setup a PV guest, and a config file that can be used with Xen. First we need to fetch all those parts:

```
# fetch http://ftp.nl.debian.org/debian/dists/jessie/main/installer-amd64/\
        current/images/netboot/xen/initrd.gz
# fetch http://ftp.nl.debian.org/debian/dists/jessie/main/installer-amd64/\
        current/images/netboot/xen/vmlinuz
# fetch http://ftp.nl.debian.org/debian/dists/jessie/main/installer-amd64/\
        current/images/netboot/xen/debian.cfg
```

We are also going to create a ZVOL in order to provide a hard drive to the guest:

```
# zfs create -V 20g zroot/debian
```

And finally we need to edit the config file `debian.cfg` in order to set the correct paths, the snipped below is a trimmed version of the config file provided by Debian:

```
kernel = "vmlinuz"
ramdisk = "initrd.gz"

memory = 1024
vcpus = 2

name = "debian"

vif = ['bridge=bridge0']
disk = ['phy:/dev/zvol/zroot/debian,xvda,w']
```

This guest has been configured to use 2 vCPUs and 1GB of RAM. The virtual network card will be added to the `bridge0` automatically by the Xen toolstack. Now we can create the guest and proceed with the installation:

```
# xl create -c debian.cfg
```

Once the install process has finished we will need to tweak the guest configuration file so it boots from the hard drive. This will require removing the `kernel` and `ramdisk` options and adding a `bootloader` option:

```
bootloader="pygrub"
```

And commenting out the `kernel` and `ramdisk` options.

Now we can boot into the installed system:

```
# xl create -c debian.cfg
```

5.3 FreeBSD HVM and PVH guests

We can setup a FreeBSD guest using two different methods, we can either use the pre-build VM images, or we can perform a normal install using the ISOs. In this example we are going to use the VM images officially provided by the project.

First we start by downloading the image:

```
# fetch http://ftp.freebsd.org/pub/FreeBSD/releases/VM-IMAGES/12.0-RELEASE/amd64/ \
       Latest/FreeBSD-12.0-RELEASE-amd64.raw.xz
# xz -d -T 0 FreeBSD-12.0-RELEASE-amd64.raw.xz
```

Then we need to create the guest configuration file:

```
# This configures an HVM rather than PV guest
type = "hvm"

# Guest name
name = "freebsd"

# Initial memory allocation (MB)
memory = 1024

# Number of vCPUs
vcpus = 2

# Network devices
vif = ['bridge=bridge0']

# Disk Devices
disk = ['/root/freebsd/FreeBSD-12.0-RELEASE-amd64.raw,raw,hda,rw']

vnc = 1
vnclisten = "0.0.0.0"
vncpasswd = "1234"
serial = "pty"
```

Note we are using the file directly as the guest hard drive. We could also dd the disk image to a ZVOL and use ZFS instead.

Before starting the guest we are going to perform some very minor tweaks to the disk image in order to setup the serial console and external connectivity:

```
#!/bin/sh

set -ex
mddev=$(mdconfig -a /root/freebsd/FreeBSD-12.0-RELEASE-amd64.raw)
mkdir -p /mnt/guest
mount /dev/${mddev}p3 /mnt/guest

chroot /mnt/guest /bin/sh <<ENDCHROOT
set -ex

# Setup PV console for PVH mode
echo 'xc0     "/usr/libexec/getty Pc"         xterm   onifconsole  secure' >> /etc/ttys

# Setup nic and sshd
sysrc ifconfig_xc0="DHCP"
sysrc sshd_enable="YES"
sysrc sendmail_enable="NONE"
sysrc sendmail_msp_queue_enable="NO"
sysrc sendmail_outbound_enable="NO"
sysrc sendmail_submit_enable="NO"

# Allow root user login and setup ssh keys
chsh -s /bin/sh root
echo 'PermitRootLogin yes' >> /etc/ssh/sshdconfig
mkdir -p /root/.ssh
cat << ENDKEYS > /root/.ssh/authorized_keys
YOUR-PUBLIC-KEY
ENDKEYS

# Setup serial console
printf "%s" "-h -S115200" >> /boot.config
cat << ENDBOOT >> /boot/loader.conf
boot_serial="YES"
comconsole_speed="115200"
console="comconsole"
boot_verbose="YES"
beastie_disable="YES"
ENDBOOT
ENDCHROOT

umount /mnt/guest
mdconfig -d -u ${mddev}
```

Now we can create the guest:

```
# xl create -c freebsd.cfg
```

The `-c` option will automatically attach to the serial console of the guest after creation. We could also attach to the VNC server in order to the see output of the VGA console:

```
# vncviewer <host>
```

Note that `<host>` is the address of the host (dom0), not the guest. That's because the VNC server is implemented by QEMU that runs as a dom0 process.

We can also boot the same FreeBSD guest in PVH mode, just by extracting the kernel and the ramdisk from the disk image.

```
# mount /dev/$(mdconfig -a /root/freebsd/FreeBSD-12.0-RELEASE-amd64.raw)p3 /mnt/guest
# cp /mnt/guest/boot/kernel/kernel .
# umount /mnt/guest
# mdconfig -d -u 0
```

Note that currently booting in PVH mode requires extracting the kernel from the image in order to make it available to the toolstack. There's work in progress in order to be able to boot a PVH guests with UEFI firmware (OVMF).

Once we have the kernel we need to create a new guest config file for PVH:

```
# This configures a PVH rather than PV guest
type = "pvh"

# Guest name
name = "freebsd"

kernel  = "/root/freebsd/kernel"
cmdline = "vfs.root.mountfrom=ufs:/dev/ada0p3"

# Initial memory allocation (MB)
memory = 1024

# Number of VCPUS
vcpus = 2

# Network devices
vif = ['bridge=bridge0']

# Disk Devices
disk =  ['/root/freebsd/FreeBSD-12.0-RELEASE-amd64.raw,raw,hda,rw']
```

PVH guests don't have almost any emulated devices, so there's no emulated VGA card, and thus no VNC server. For this guest type the Xen console is used by default.

```
# xl create -c freebsd.cfg
```

Startup time of PVH guests is usually very fast because there's no firmware and the startup of PV devices is also faster than their emulated counterparts.

6 Basic guest management

6.1 VM life cycle

xl has a set of options to manage how to deal with certain guest actions, like reboot or shutdown. The following set of guest event are detected by xl and can trigger admin-defined behaviour.

- `on_poweroff`: specifies what should be done with the domain if it shuts itself down.
- `on_reboot`: action to take if the domain shuts down with a reason code requesting a reboot.
- `on_watchdog`: action to take if the domain shuts down due to a Xen watchdog timeout.
- `on_crash`: action to take if the domain crashes.
- `on_soft_reset`: action to take if the domain performs a 'soft reset'.

The actions that can be triggered by those events are:

- `destroy`: destroy the domain.
- `restart`: destroy the domain and immediately create a new domain with the same configuration
- `rename-restart`: rename the domain which terminated, and then immediately create a new domain with the same configuration as the original
- `preserve`: keep the domain. It can be examined, and later destroyed with xl destroy.
- `coredump-destroy`: write a "coredump" of the domain to /usr/local/var/lib/xen/dump/NAME and then destroy the domain.
- `coredump-restart`: write a "coredump" of the domain to /usr/local/var/lib/xen/dump/NAME and then restart the domain.
- `soft-reset`: Reset all Xen specific interfaces for the Xen-aware HVM domain allowing it to reestablish these interfaces and continue executing the domain. PV and non-Xen-aware HVM guests are not supported.

For example a combination I often use when working on Xen guest support for different OSes is:

```
on_crash="preserve"
```

This will keep all the domain data (memory and machine state) when the domain crashes, thus allowing the admin to attempt to debug it.

Another interesting combination specially in production if your guest supports the Xen watchdog is:

```
on_watchdog="restart"
```

Or:

```
on_watchdog="coredump-restart"
```

The following options will restart a domain if a watchdog timeout is detected, thus ensuring minimal downtime and no need to use in-guest monitoring agents.

6.2 Memory and vCPU hotplug

Xen has support for both memory and vCPU hotplug to Xen guests. This feature however requires cooperation from the guest, so it's availability depends on the guest OS type and version. Note that FreeBSD doesn't support vCPU hotplug.

Memory ballooning was originally designed for PV guests which don't use hardware memory address translation and are limited to use 4K pages only. The interface was designed to easily allow PV guests to request more memory, or to free up unused memory when requested by the hypervisor.

Later on this interface was also made available to translated guests (HVM and PVH), thus allowing such guests to create holes in the physical memory map. Albeit memory ballooning is functional on HVM and PVH guests, it's usage can lead to performance degradation due to the fact that the guest physical memory map is populated using super pages when possible (2M or 1G) and making holes into the physical memory map will lead to super page shattering, bringing down performance.

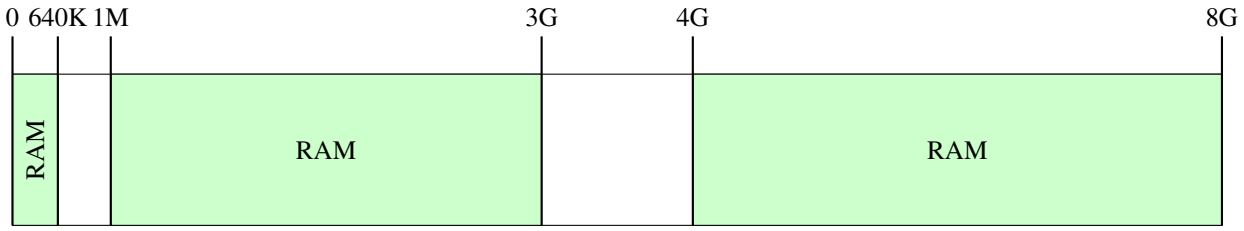


Figure 4: HVM guest memory layout

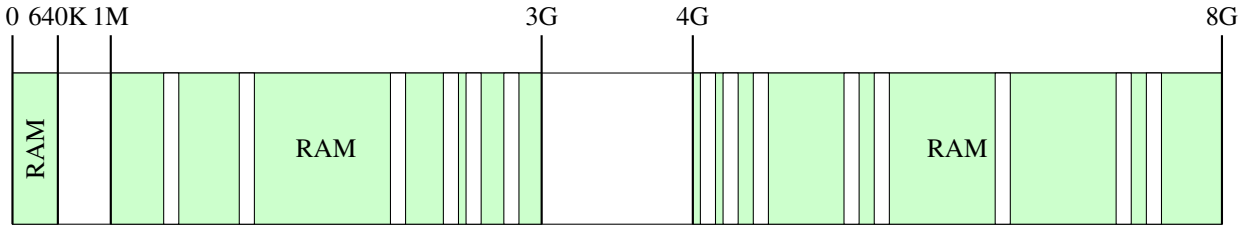


Figure 5: HVM guest memory layout with ballooned out memory

Let's start by looking at memory hotplug by launching the FreeBSD HVM guest and ballooning down it's amount of memory:

```
# xl create -c freebsd.cfg
# xl mem-set freebsd 512m
```

Note that the above command only requests the guest to balloon memory until it reaches the set target, but there's no way for Xen to enforce memory ballooning.

CPU hotplug works in a similar way, and also requires guest cooperation since Xen cannot enforce a guest to start or stop a vCPU. For this example we will use the Debian guest since there's no CPU hotplug in FreeBSD:

```
# xl create -c debian.cfg
# xl vcpu-set debian 1
```

In order to bring back the offlined vCPU we will have to set it to online in the guest itself:

```
# xl vcpu-set debian 2
[ inside the guest ]
root@debian:~# echo 1 >/sys/devices/system/cpu/cpu1/online
```

We could also setup an udev script in order to bring up vCPUs automatically as they are added to the guest.

6.3 Save and restore

Xen allows to dump the whole VM state to disk using the xl toolstack. The data dumped includes the guest memory and the state of emulated devices if there are any. The save and restore procedure is specially interesting in order to create checkpoints of guest states, or to keep the guest state across host reboots.

Lets start by performing a simple save and restore of a running guest. We are going to use the Debian PV guest we have setup earlier:

```
# xl create -c debian.cfg
[... wait guest to boot ...]
# xl save debian debian.save
# xl list
Name                                ID   Mem VCPUs      State       Time(s)
Domain-0                            0  4095     4      r-----      301.5
[ reboot? ]
# xl restore debian.save
```

The above example is fairly simple and shows the basic usage of save and restore. However there's more that can be achieved by combining save and restore and ZFS snapshots, allowing us to create checkpoints that can be used later to restore the guest to a known state.

The following example will create a snapshot of the debian guest each minute by saving the domain and creating a snapshot of the disk ZVOL:

```
#!/bin/sh

set -ex

while true; do
    time=$(date +%s)
    xl save -p debian debian-${time}.save
    zfs snapshot zroot/debian@${time}
    xl unpause debian
    sleep 60
done
```

The above script will create tuples of domain state and disk state that can be used after wards to restore a guest to a

given saved state. Note that the size of the guest saved state is related to the memory assigned to a guest, so guests with large amounts of memory will create very large save files.

After running the script you can check the ZVOL snapshots using:

```
# zfs list -t snapshot
NAME                        USED  AVAIL  REFER  MOUNTPOINT
zroot/debian@1552572976     400K      -  2.87G  -
```

Note that in case of restoring from a snapshot, the ZVOL snapshot has to be converted into a clone:

```
# zfs clone zroot/debian@1552572976 zroot/debian-1552572976
```

The guest configuration file has to be adjusted to use the clone, the disk line should be modified so it points to the cloned ZVOL:

```
disk = ['phy:/dev/zvol/zroot/debian-1552572976,xvda,w']
```

And in order to restore the guest from the checkpoint the xl command must specify the new guest configuration file that has the path to the clone:

```
# xl restore debian-1552572976.cfg debian-1552572976.save
```

6.4 Live migration

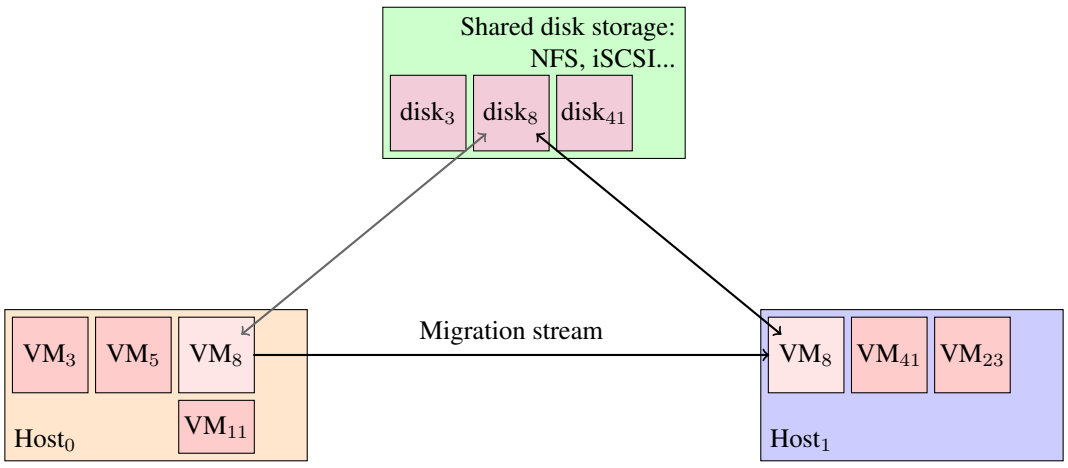


Figure 6: Example pool setup

Live migration uses the same functionality as save and restore, but instead of dumping the guest data to a file on disk the data is sent to another host that restores the domain. This functionality allows to move guests between hosts transparently from a guest point of view.

This feature is specially interesting for load balancing, or when performing host updates.

For this tutorial I only have a single host, so live migration is kind of pointless. However for the purpose of showing how it works, we can perform a local live migration.

```
# xl create -c pvh.cfg
[... wait guest to boot ...]
# xl migrate freebsd localhost
```

By default live migration sends the guest state over ssh to the destination host, other transports can be used with the `-s` option.

7 Advanced guest management

7.1 Schedulers

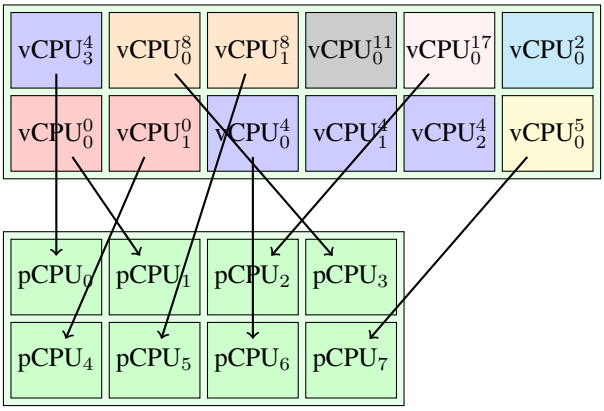


Figure 7: High level scheduler overview

The Xen scheduler(s) are the entity that decide when and where Xen guests are run. It's in charge of assigning vCPUs (guest virtual CPUs) to pCPUs (physical CPUs on the system). Note that while the number of vCPUs will likely change over the host runtime as guests are created and destroyed, pCPUs are likely to stay the same.

The scheduling of vCPUs across pCPUs is done by default based on fairness only, so that all vCPUs not idle get the same amount of CPU time. There are however ways to boost or decrease the priority of guests.

There are different scheduler implementations inside Xen, with different characteristics:

- `credit`: the credit scheduler is a proportional fair share CPU scheduler built from the ground up to be work conserving on SMP hosts. This is the default scheduler in Xen 4.11 and older versions.
- `credit2`: rebuild of the credit scheduler. This is the default scheduler in Xen 4.12 and newer versions.
- `rtds`: this rt scheduler applies Preemptive Global Earliest Deadline First real-time scheduling algorithm to schedule vCPUs in the system. Each vCPU has a dedicated period, budget and extratime. While scheduled, a vCPU burns its budget. A vCPU has its budget replenished at the beginning of each period; Unused budget

is discarded at the end of each period. A vCPU with extratime set gets extra time from the unreserved system resource.

- `arinc653`: scheduler for space and time partitioning in safety-critical avionics real-time operating systems (RTOS).
- `null`: dummy static scheduler that relies on statically assigning each vCPU to a pCPU. Doesn't support vCPUs $>$ pCPUs. Attempts to reduce the scheduling overhead on static systems (ie: embedded deployments with a known number of guests).

The default scheduler, either `credit` or `credit2` attempts to provide a fair scheduling of guests. Note that none of the Xen schedulers currently perform gang-scheduling of guests vCPUs.

The system wide scheduler can be selected at boot time using the `sched` command line parameter.

Lets start by looking at the default scheduler parameters:

```
# xl create freebsd.cfg
# xl sched-credit
Cpupool Pool-0: tslice=30ms ratelimit=1000us migration-delay=0us
Name                                ID Weight  Cap
Domain-0                             0    256    0
freebsd                              1    256    0
```

In the above example dom0 has the same priority as the guest we have just created. We might want to boost dom0 priority over guests, since it runs the backends and the device models:

```
# xl sched-credit -d 0 -w 512
```

By doubling the weight of dom0 we are giving twice as much CPU time than non-boosted guests. This might be an effective way to make sure dom0 gets enough CPU time on contended systems. Note that the weight of a domain can also be set from the guest config file using the `weight` option.

7.2 System partitioning with CPU pinning

Apart from modifying the weight of a domain Xen has other ways to partition the system and isolate the guests to certain CPUs. Let's start by looking at CPU pinning. Xen allows to select on which pCPUs a vCPU can be scheduled. Such method allows to isolate guests to certain pCPUs. Let's look at a couple of CPU pinning examples:

```
# Allow guest to run on all pCPUs except pCPU 0
cpus="all^0"
# Allow guests to run on pCPUs 0, 1, 2 and 3
cpus="0-3"
# Allow guest to run on pCPUs 0 and 3 only
cpus="0,3"
```

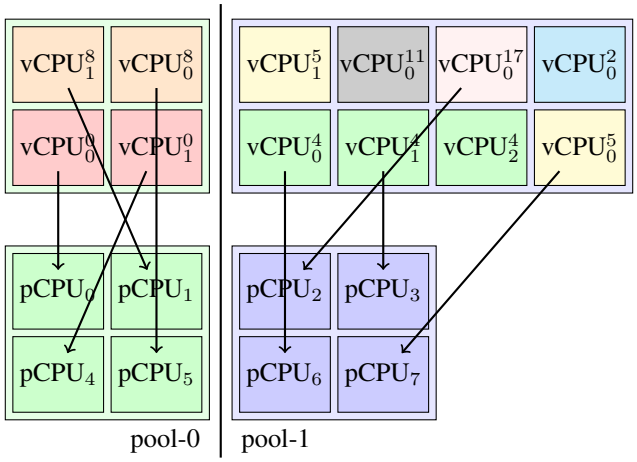


Figure 8: System partitioning with CPU pools

7.3 System partitioning with CPU pools

Another interesting topic for isolation are CPU pools. Xen can group the pCPUs of a server in cpu-pools. Each pCPU is assigned at most to one cpu-pool. Domains are each restricted to a single cpu-pool. Scheduling does not cross cpu-pool boundaries, so each cpu-pool has its own scheduler. pCPUs and domains can be moved from one cpu-pool to another only by an explicit command.

Lets start by listing the current CPU pools:

```
# xl cpupool-list
Name                CPUs   Sched     Active   Domain count
Pool-0               12    credit       y          1
```

This is the default CPU pool, containing all the pCPUs on the system using the credit scheduler. Let's create a different CPU pool and use a different scheduler on it. This is achieved using a config file and a xl command. The config file for our new CPU pool will be:

```
name="pool2"
sched="credit2"
cpus="2-3"
```

Now we need to remove the pCPUs we want to use from the default pool and finally create the new CPU pool:

```
# xl cpupool-cpu-remove Pool-0 2-3
# xl cpupool-create pool2.cfg
Using config file "pool2.cfg"
cpupool name:   pool2
scheduler:      credit2
number of cpus: 2
```

We can also create a domain and assign it to this CPU pool by adding the following to the domain config file:

```
pool="pool2"
```

And then the guest can be created as usual:

```
# xl create freebsd.cfg
# xl list -c
Name                ID   Mem VCPUs      State   Time(s)          Cpupool
Domain-0             0  2048    12     r-----     206.0           Pool-0
freebsd              1   512     2     r-----      17.5            pool2
```

It is also possible to move and existing domain between cpu pools, by using:

```
# xl cpupool-migrate freebsd pool2
```

With the above configuration we can assure that the pCPUs used by `Domain-0` will never be used by the `freebsd` guest.

7.4 PV guests in shim mode

Since the discovery of the Meltdown and Spectre vulnerabilities the Xen community has been working on improving the isolation between guests. Part of this work has focused on providing a way to run PV guests inside of a hardware assisted paging environment, like the one used for HVM or PVH guests (ie: a HVM container). In order to achieve this a shim is run inside of a PVH guest, and in turn this shim creates the PV guest. Running a PV guest inside of a PVH container with the shim can be seen as booting Xen inside of a PVH container and then launching a single PV guest.

This provides an extra layer of isolation for PV guests, since you would first need to break out into the Xen shim, and afterwards break outside of the PVH container in order to compromise the system. Breaking into the shim is not a security issue, since the shim only runs a single PV guest.

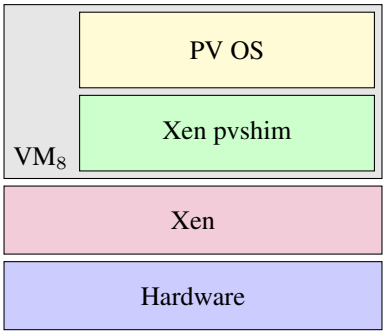


Figure 9: Diagram of a PV guest running in shim mode

In order to convert a PV guest to run in shim mode you need to add the following to it's config file:

```
type = 'pvh'
pvshim=1
```

Afterwards the guest can be started as normal, note that boot time might be slightly increased due the fact that now a Xen shim is started before the guest PV kernel.

Note that Xen allows removing the PV interfaces from the hypervisor at compile time, so a version of the hypervisor that only supports HVM and PVH can be built and classic PV guests can still be used in conjunction with the shim. Removing the PV interfaces from the hypervisor is interesting in order to reduce the surface of attack exposed to guests.

7.5 Building from source

Upstream Xen should build on FreeBSD without requiring any external patches. Lets start by installing the build dependencies and cloning the upstream repository and switching to the `staging` branch:

```
# pkg install git glib pkgconf yajl gmake pixman markdown gettext \
              python argp-standalone lzo2 git seabios
# git clone git://xenbits.xen.org/xen.git
# cd xen && git checkout staging
```

Now we can start the build, note that clang and lld will be used:

```
# ./configure --with-system-seabios=/usr/local/share/seabios/bios.bin
# gmake -j6 build-xen clang=y
# gmake -j6 install-tools clang=y
# gmake -j6 install-tests clang=y DESTDIR=/usr/local/ DEBUG_DIR=lib/debug
# cp xen/xen /boot/
# reboot
```

After rebooting we will be in our newly installed Xen staging system.

7.6 Live patching

As part of the above build process we have generated some live patching examples that we can now apply. First lets list the currently applied patches:

```
# xen-livepatch list
 ID                                      | status
-----------------------------------------+------------
```

As expected there are no live patches currently applied. Lets upload and apply a livepatch to the hypervisor that's going to change Xen's minor version to match Xen's major version:

```
# xl info
[...]
xen_major              : 4
xen_minor              : 13
[...]
# xen-livepatch upload test /usr/local/lib/debug/xen-livepatch/xen_nop.livepatch
```

```
Uploading /usr/local/lib/debug/xen-livepatch/xen_nop.livepatch... completed
# xen-livepatch apply test
Applying test... completed
# xen-livepatch list
 ID                                      | status
----------------------------------------+------------
test                                     | APPLIED
# xl info
[...]
xen_major              : 4
xen_minor              : 4
[...]
```

This live patch will change the minor version reported by the hypervisor, and it's effects can be seen when using `xl info`. Live patches can also be reverted and unloaded using:

```
# xen-livepatch revert test
# xen-livepatch unload test
# xen-livepatch list
 ID                                      | status
----------------------------------------+------------
```

8 Troubleshooting

It's inevitable that at some point things will go wrong, or you will be faced with errors that need to be understood and fixed. Thankfully the Xen toolstack and it's different components are setup by default to write logs to `/var/log/xen`.

Inside this directory we will find two different kind of logs:

- `qemu-dm-*.log`: logs created by QEMU, the device model. Those logs are created for HVM guests or guests that require backends implemented in QEMU.
- `xl-*.log`: logs created by the xl toolstack, contain information relevant to domain creation and destruction.

Finally there's also the hypervisor console, which can contain relevant information if the error happened inside the hypervisor. The output can be fetched from the serial cable, or from the xl toolstack:

```
# xl dmesg
 Xen 4.13-unstable
(XEN) Xen version 4.13-unstable (root@) (FreeBSD clang version 6.0.1 (tags/RELEASE_601/
(XEN) Latest ChangeSet: Tue Feb 12 18:33:30 2019 +0000 git:1e780ef5a5
[...]
```

If none of this sources provide any insight in order to help resolve the issue, the Xen community also provides two mailing lists:

- freebsd-xen@freebsd.org: FreeBSD/Xen specific mailing list, useful when dealing with Xen issues specific to FreeBSD.
- xen-users@lists.xenproject.org: user oriented mailing list, helpful when dealing with configuration issues.
- xen-devel@lists.xenproject.org: developer mailing list, helpful when dealing with guest or host crashes.

Both communities are very friendly, don't be afraid to send messages in case of issues. It's also fine to send questions to two lists at the same time if required, for example the FreeBSD/Xen specific mailing list and xen-users or xen-devel. Cross-posting to xen-users and xen-devel is not recommended, it's best to first post to xen-users and escalate to xen-devel if needed.