kernels-in-kernels

kernels within kernels by Lee Elston

Version 1.0

The C-ALE (Cloud & Container Apprentice Linux Engineer) is a series of seminars held at existing conferences covering topics which are fundamental to a Linux professional in the Linux Cloud and Container field of computing.

This seminar will spend equal time on lecture and hands on labs at the end of each seminar which allow you to practice the material you've learned.

> This material makes the assumption that you have minimal experience with using Linux in general, and a basic understanding of general industry terms. The assumption is also made that you have access to your own computers upon which to practice this material.

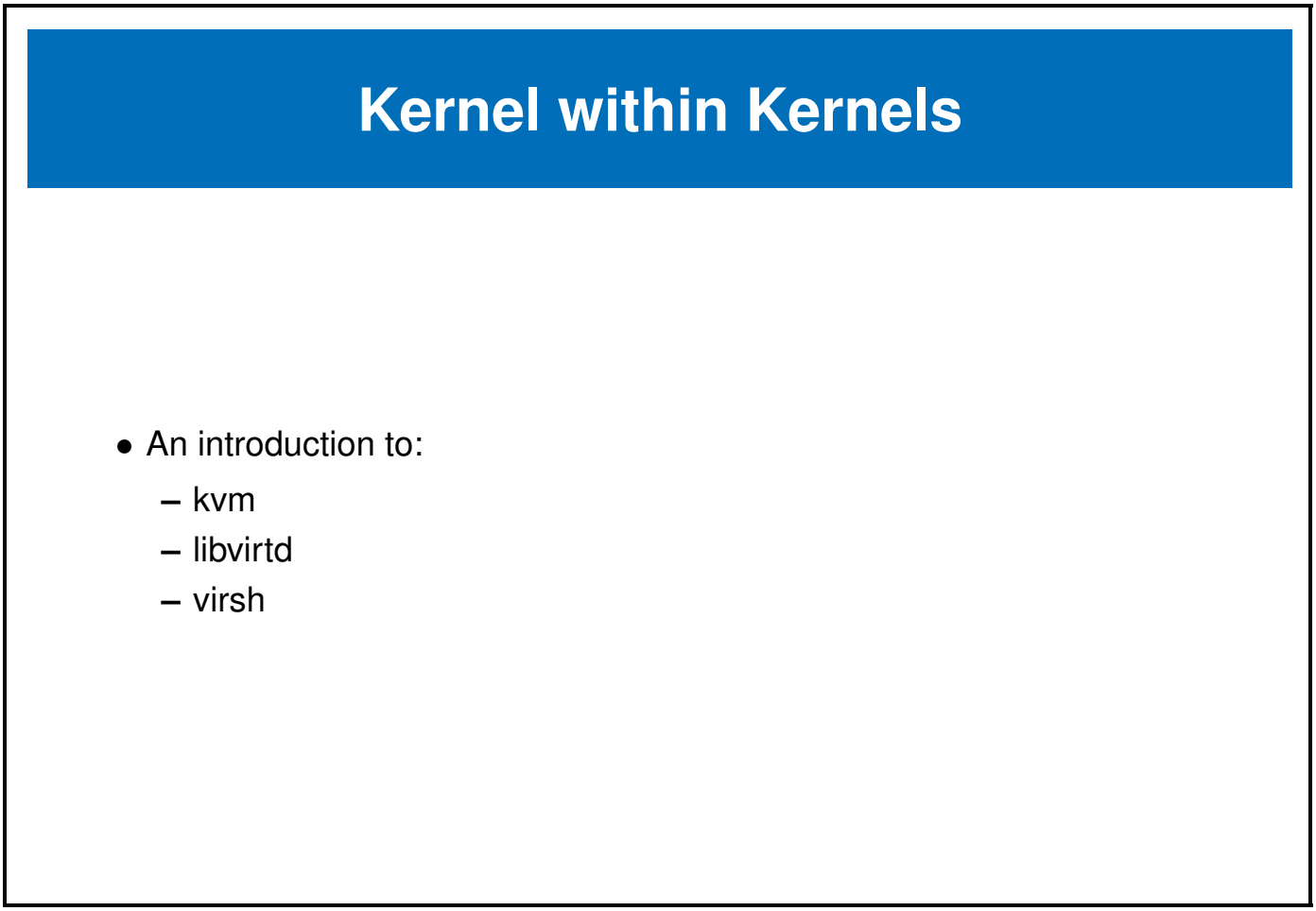
More information can be found at **https://c-ale.org/**

Contents

Chapter 1

Kernels within Kernels

1.1 Kernels within Kernels



When we talk about kernel within kernels, we are looking to run an operating system as an application on an existing running computer. Perhaps we would like to run a copy of an old operating system on a new piece of hardware, one the old operating system knows nothing about or we may require separate operating system environments for a specific application that doesn't get along with other applications. Whatever the reason we would like to have multiple operating systems running on a single platform. The tools we are going to look at to run our kernel within a kernel are: **kvm**, **libvirtd** and **virsh**.

1.2 KVM

KVM: Kernel Virtual Machine

- An open source virtualization technology built into Linux
  - turns Linux into a hypervisor that allows virtual guests to be created
  - uses Linux for process management such as memory management, io, scheduling
  - included in kernel version 2.6.20 and later
- KVM uses Virtualization extensions in the processor chips
  - Intel: VT-x (also known as vmx)
  - AMD: AMD-V (also known as svm)

The virtualization extensions may be disabled inn the BIOS of the computer, enable them as necessary. If the options are enabled they will be listed in the file `/proc/cpuinfo` in the **flags** list. Use a command like **grep -e svm -e vmx /proc/cpuinfo** to verify the kernel can see the extension flags.

These cpu extensions, **svm** and **vmx** are necessary for **KVM** but are not the only extensions to enhance virtualization. Some of the additional **Intel** are:

- **EPT** Extended Page Tables
- **SR-IOV** Single Root I/O Virtualization
- **DDIO** Data Direct I/O Technology

Virtualization enhancements exist in most processor chips, some examples can be found at:

- AMD: https://www.amd.com/en-us/solutions/servers/virtualization
- Intel: https://www.intel.ca/content/www/ca/en/virtualization/virtualization-technology/intel-virtualization-technology.html
- ARM: https://genode.org/documentation/articles/arm_virtualization/

KVM: the modules

KVM is comprised of two kernel modules, a common module and a processor specific module:

- **kvm.ko**
  - common to all architectures
  - provides the virtualization infrastructure
  - accessible through the **kvm api**
- **kvm-intel.ko** or **kvm-amd.ko**
  - specific processor type module
  - kernel driver module for specific hardware and related extensions.

There is an API interface available to use the **KVM** modules. The kvm api is a set of **ioctls** to control components of a virtual machine. The ioctls fall into three classes:

- System ioctls: used to query and set global kvm subsystem attribute's
  - use this ioctl to create a virtual machine
- VM ioctls: used to query and set attributes that affect a single virtual machine environment
  - manipulate memory layout
  - create virtual CPU's (vcpus)
- vcpu ioctls: used to control the operation of a single vcpu.
  - run a guest virtual cpu

See http://www.kernel.org/doc/Documentation/Virtual/kvm/api.txt for more information on the API or perhaps http://lwn.net/Articles/658511 for an example of coding the API directly.

QEMU: Quick Emulator, the glue

- Is a Virtual Machine Monitor (VMM) or Hypervisor
  - emulates various CPU's
  - provides device models
  - executes as an emulator using dynamic binary translation
  - uses hardware virtualization extensions when coupled with KVM
- The combination of QEMU and KVM with hardware virtualization extensions can achieve near native performance within the VM's

QEMU brings configuration interface emulation of disk, nic, etc. Documentation on the various devices that are emulated and their options can be found in the **QEMU Documentation** at:

https://www.qemu.org/documentation/

A discussion on **Understanding QEMU devices** is at:

https://www.qemu.org/2018/02/09/understanding-qemu-devices/.

kvm overview

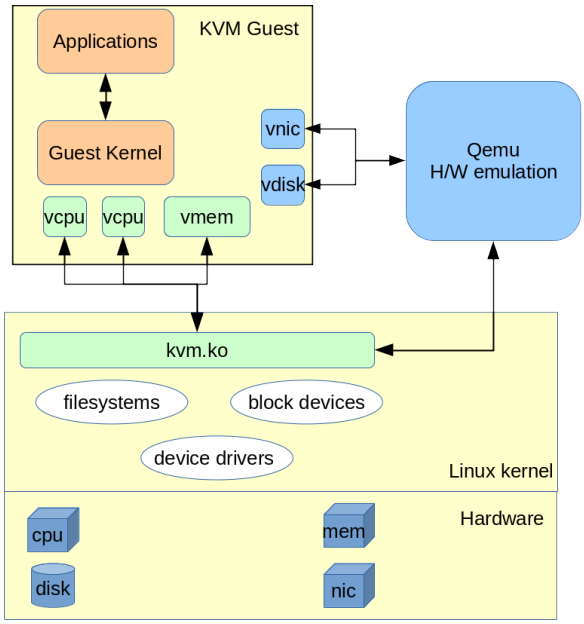


Figure 1.1: **Overview of kvm**

The key to **KVM** actually running a virtual machine is a combination of components. When these components are combined a fully functional virtual machine capable of running an operating systems is created. The components are:

- Kernel support and the two kernel modules
- Configuration method for the modules, the kvmapi
- Something that looks like disk type storage, qemu
- Bootable loader, or bios provided by qemu
- Additional virtual devices (NIC or Monitor) that link to the real devices, qemu
- A debug method, the qemu-monitor

The **KVM** kernel modules do the heavy work of securely managing the memory and cpu requests from the VM's where as **qemu** provides the **VM's** with the peripheral devices like virtual network cards, virtual storage devices and other virtual devices.

KVM/QEMU configuration

Configuring and starting a **VM** with **KVM/QEMU** can be done at the command line. There are a few steps:

- Confirm or load the appropriate **KVM** module
- Create an image file to be used as a disk
- Start **qemu-kvm** with a list of options:
  - disk image for storage
  - disk (or image) of a CD-ROM to install from
  - memory size directive
  - a boot command
- Install the operating system on the **VM**
- Stop the VM after the install is complete
- Restart the VM like before but select the disk image to boot from

The commands to install and run a **VM** might look like this:

Create a 8G disk image

```
# qemu-img create -f qcow2 vdisk.img 8G
```

Boot from a CDrom image file, and install to the disk image

```
# qemu-system-x86_64 -hda vdisk.img -cdrom /path/to/cdimage.iso -boot d -m 512
```

When the install is complete restart the virtual machine without the CDrom

```
# qemu-system-x86_64 vdisk.img -m 512
```

This will start a **VM**, a very basic one, with no network and limited resources but it will work. In order to meet the challenge of managing all these components, **libvirt** was created.

1.3 libvirt

libvirt

libvirt is an `open-source` management toolkit for managing virtualization platforms.

The project home page is `https://libvirt.org/`

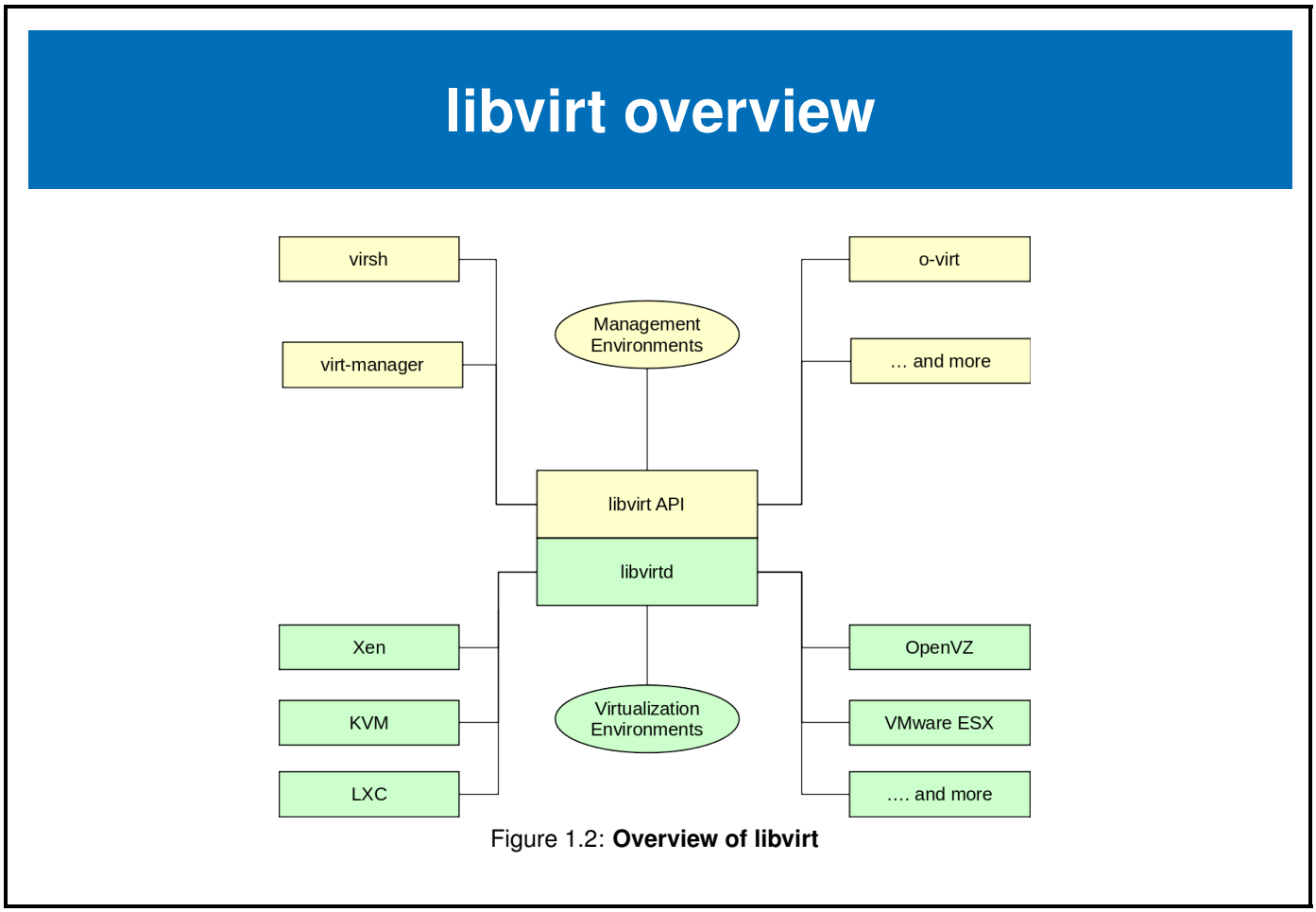


Figure 1.2: **Overview of libvirt**

The package referred to as **libvirt** is comprised of two components:

- **libvirtd**, the server side daemon
  - runs on host servers
  - performs management tasks for virtual guests (start,stop,migrate,etc)
- **libvirt API** (client libraries and utilities)
  - connect to **libvirtd** via Unix socket or TCP/IP
  - request tasks be performed on the virtual guests
  - request configuration and resource information

libvirt-daemon overview

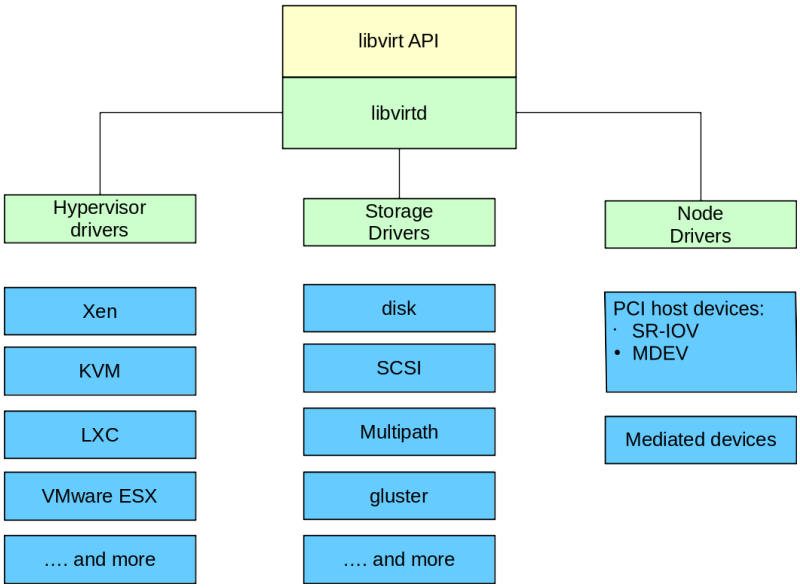


Figure 1.3: **Overview of libvirt-daemon**

The **libvirtd** daemon is the component that is sensitive to various virtualization technologies. Generic input from the client library is translated to hypervisor specific commands, so a **vm start** command for a **xen** virtual machine is the same as a **vm start** command for a **kvm** virtual machine. Depending on the distribution there may be separate packages for the various **libvirt-daemon** functions,

- libvirt is a toolkit to manage virtualization technologies
  - **KVM**/**QEMU**, **LXC**, **Xen**, . . . and many more
- libvirt provides an API interface usable by many different languages
  - **C, Python, Pearl**, . . . and more
- libvirt provides management APIs for:
  - network filters, host devices, host interfaces, virtual networks, secrets, storage pools
  - for a complete list of the API support matrix see: `https://libvirt.org/hvsupport.html`

libvirtiAPI overview

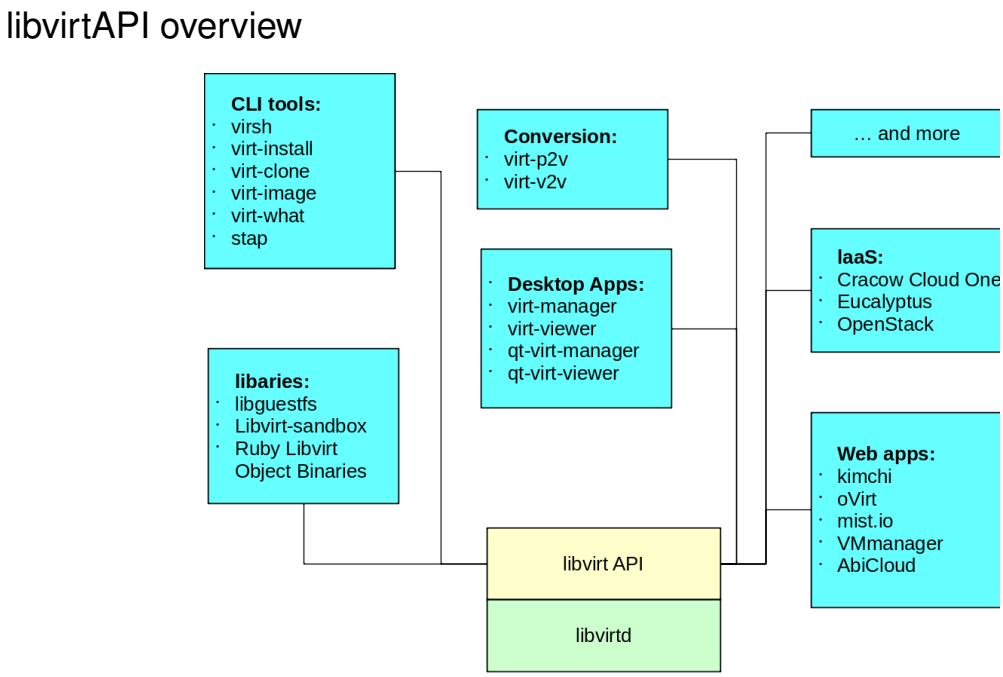


Figure 1.4: **Overview of libvirtAPI**

The **libvirt API** provides a common user interface to various virtualization environments. The connection to the virtualization environment is made by the **libvirtd** daemon running on the VM host. Once the connection is established, applications can configure and control the virtual environments with tools such as **virt-manager** or **virsh**.

The **libvirt** project utilities are available on most distributions. There are many application programs that interface with **libvirt**, some of the most common are **virt-manager**, **virt-viewer**,**virt-install**,**virsh**.

The complete list of **libvirt** applications is available at `http://www.libvirt.org`.

Tools that use the **libvirtAPI** are in these genral categories:

- Command Line Tools
- Conversion Tools
- Desktop Multi-function Applications
- Web Applications
- Infrastructure as a service (IaaS)

1.4 virsh

virsh: introduction

virsh is the main interface to manage services provided by **libvirt**. Some of the services that virsh can manage are:

- Guest Domains
- Device
- Virtual Network
- Virtual interface
- Storage Pool

The **virsh** documentation is located at https://libvirt.org/virshcmdref.html , the **man 1 virsh** page and the internal help text with the **virsh help** command.

The **virsh** command can be used to create, start, pause and destroy **virtual machines** or **guest domains**. It uses **libvirt** and supports **Xen**, **QEMU**, **KVM**,**LXC**,**OpenVZ**, **VirtualBox** and **VMware ESX** environments.

The service **libvirtd** should be running as it provides the connection to the various hypervisors for **virsh**.

virsh: Getting started

Before using **virsh** to manage our virtual machines a virtual machine configuration needs to be defined, of course this can be done through the command line, however, a short **XML** file can ease the process.

For our first VM we will use a preexisting disk, taking advantage of several defaults, we can create the following sections in our **html** file:

- main system: mem, name
- OS: memory, architecture,hypervisor, boot device
- devices: emulator,disk, console
- graphics:vnc
- and finally test the application.

```
<domain type='kvm'>
  <name>blue</name>
  <memory unit='KiB'>2097152</memory>
  <os>
    <type arch='x86_64' machine='pc'>hvm</type>
    <boot dev='hd'/>
  </os>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-kvm</emulator>

    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2'/>
      <source file='/var/lib/libvirt/images/blue.img'/>
      <target dev='vda' bus='virtio'/>
    </disk>

        <serial type='pty'>
      <target type='isa-serial' port='0'>
        <model name='isa-serial'/>
      </target>
    </serial>

    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <graphics type='vnc' port='-1' autoport='yes'/>
  </devices>
</domain>
```

virsh: define

In order for **libvirt** to recognize the virtual machine the xml configuration file needs to be registered with **libvirt**.

- validate the xml file to be used for input
- **define** the virtual machine based on the xml file
- verify the virtual machine is available

Before registering the configuration it can optionally be validated:

```
# virt-xml-validate blue.xml
blue.xml validates
```

To register a xml configuration file with **libvirt**:

```
# virsh define domain.xml
```

For our example file:

```
# virsh define blue.xml
```

The VM should now be registered with **libvirt** but not started, this can be verified with the **virsh list –all** command:

```
# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     alice                          shut off
 -     blue                           shut off
 -     bob                            shut off
```

virsh define configuration file

The **virsh define** command creates an xml file that represents the virtual machine. During the define or creation process the input xml file is parsed and its contents override the default configuration parameters.
The resultant configuration file is located: `/etc/libvirt/qemu/name/xml`.
When libvirt initializes it looks in this directory for virtual machine configurations. If the xml file is not here libvirt cannot see or manage the virtual machine.
DO NOT edit the files in the /file/etc/libvirt/qemu/ directory.

An example creating a new virtual machine from an xml file and then compare the input to the resultant configuration file.

virsh define command example:

```
# virsh define /var/lib/libvirt/blue.xml
Domain blue defined from /var/lib/libvirt/blue.xml

[root@flex /]# ls -l /var/lib/libvirt/blue.xml
-rw-r--r--. 1 root root 695 Jul  6 13:52 /var/lib/libvirt/blue.xml

               [root@flex /]# ls -l /etc/libvirt/qemu/blue.xml
-rw-------. 1 root root 1907 Jul 23 09:11 /etc/libvirt/qemu/blue.xml
```

The differences in the files is some warning text that the file is auto created and application of some additional defaults for values not specified in the input xml file. **libvirt** uses the directory `/etc/libvirt/qemu/` to store its working copy of the configuration. If the xml file does not exist in this directory **libvirt** will have no knowledge of the VM. Do not edit these files directly, use the command **virsh edit domain** to make changes.

virsh: starting and stopping

The commands to start and stop a registered domain are:

- **virsh start domain**
- **virsh shutdown domain**
- **virsh destroy domain**

The **virsh start domain** command functions as expected, perform the equivalent of a power on for the virtual machine. An option of **–console** may be appended to the end of the start command (after the domain) to start a text console on the VM. The guest OS must support a text based console for this ti be useful.

Stopping a virtual machine may be more interesting as it depends on which features the guest os supports for shutdown, the current shutdown mode choices are: acpi,agent,initctl,signal and paravirt. The mode may be specified with the **–mode** parameter added to the **virsh shutdown domain**.
Note that often the shutdown command is ignored by the guest os. It may be a better option to execute a shutdown command on the virtual machine itself rather than using the virsh command in this instance.

The **virsh destroy domain** is equivalent to pulling the power plug on the virtual machine.

virsh connecting to the VM

When connecting to a VM there are two common methods:

- text console
- graphical console

- **text console**, can be started two ways:
  - as a start option to **virsh start test1 –console**
  - as a virsh command, **virsh console test1**
- **graphic console**, supports two protocols:
  - **VNC**
  - **SPICE**

  The tool **virt-viewer** is a lightweight libvirt application that provides a GUI based terminal session. **virt-viewer** can use either **SPICE** or **VNC** for connection to the VM.

virsh undefine

The virsh command **undefine** removes the registered domain configuration.

Some examples of the **virsh undefine** command:

- delete the libvirt configuration for vm blue.

```
# virsh undefine blue
```

- delete the libvirt configuration for vm blue and remove all associated storage devices.

```
# virsh undefine blue --remove-all-storage
```

1.5 virt-manager

virt-manager

virt-manager project is the home to several common virtual machine management tools based on **libvirt**. Some of the tools are:

- **virt-manager** graphic tool for creating and managing virtual machines, KVM,Xen and LXC.
- **virt-viewer** graphic tool for connecting to the client VM's, uses VNC or SPICE protocols
- **virt-install** command driven installer for creating VM's

For additional information and documentation see
https://virt-manager.org

The screenshot below is an example of **virt-manager** displaying the virtual hardware details of a virtual machine.

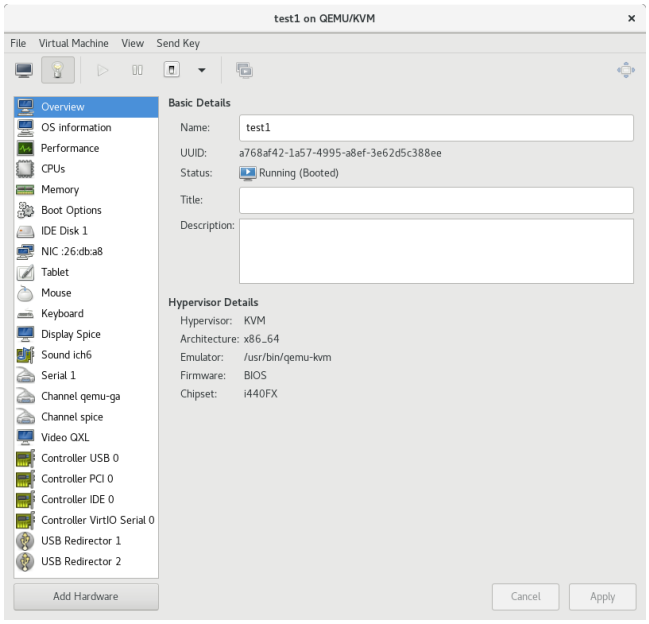


Figure 1.5: **virt-manager virtual hardware details**

virt-install

The **virt-install** is an application that uses the **libvirt** interface, some of the features are:

- command line interface
- text or graphic installer interface
- local or network install
- an import only option

More information may be obtained from the associated man page or the project web site, **virt-install** is part of **virt-manager** project. Additional information can be found at https://virt-manager.org/.

The **virt-install** tool adds a layer of abstraction to the install process of a virtual machine. It uses **libvirt** and presents the user with a command line interface. Options are passed to **virt-install** describing the installation. The minimum items required to start an install are:

- `--name`
- `--memory`
- `storage (--disk or --filesystem)`
- `install option (--cdrom,--location,--livecd)`

The example here installs from a DVD image using a text or serial console.

```
#!/bin/bash
# Sample using virt-install to install Debian from
# DVD1 of the Debian official distribution
# Minimum disk seems to be 1.5G
# Install contacts outside resources so a network is required.

 virt-install \
--name test-loc \
--ram 512 \
--disk size=1.5 --vcpus 1 \
--os-type linux \
--os-variant debian9 \
--graphics none \
--boot useserial=yes  \
--cdrom /var/ftp/pub/iso/debian-9.4.0-amd64-DVD-1.iso
```

virt-install –import

The option **import** to the **virt-install** command:

- skips the install process
- creates a default configuration for the new VM
- incorporates the command line options
- starts the new VM in a **virt-viewer** session

The resulting configuration may be viewed with the **virsh dumpxml name** command.

```
#!/bin/bash
# demonstrate the import function of virt-install

virt-install \
              --name import \
              --memory 512 \
              --disk /var/lib/libvirt/images/eric-x86_64.img \
              --os-type  linux --os-variant centos6.2  \
              --import
```

Another example of the import function, this time we were given a **vmdk** disk image that used a **SCSI** virtual adapter on the originating system. Unfortunately the initial ram disk does not have a **SCSI** driver built in. This is not a problem, the virtual adapter type can be changed easily using the **virt-install** import with disk options:

```
# virt-install  --name notscsi --memory 1024  \
        --disk path=/home/lee/Virt-KVM/CentOS7.vmdk,bus=ide \
        --ostype linux --os-variant centos7.0 \
        --import
```

The os-variant is available with the **osinfo-query os** command.

1.6 Labs

Exercise 1.1: Verify system is ready for KVM

Verify the CPU chip extensions are available and enabled to support virtualization. Confirm the required packages are available.

This exercise requires a 64 bit architecture system with a minimum of 4GiB if memory and 5GiB of free disk space. Internet connectivity is recommended but not critical.

It is common practice to run virtual machines as a non-root user, but in this exercise we will execute the commands as root.

Solution 1.1

1. Verify the cpu has virtualization support enabled.

   ```
   # grep -e svm -e vmx /proc/cpuinfo
   ```

   The output should have one of the flags highlighted, either **svm** or **vmx**. If no output was produced, verify in your **BIOS** that virtualization is supported.

2. Get the **knk-resources.tar** file. The presenter will have the information as to how to obtain the file.

3. Install required packages:

   - On **CentOS7**:

     ```
     # yum install qemu-kvm libvirt virt-manager virt-install virt-viewer
     ```

   - On **OpenSUSE**:

     ```
     # zypper install qem-kvm libvirt virt-manager virt-viewer bridge-utils
     ```

   - On **Ubuntu**:

     ```
     # apt-get install qemu-kvm libvirt-bin virtinst \
                       virt-manager libosinfo-bin virt-viewer
     ```

4. Verify and optionally start and enable the **libvirtd** service.

   ```
   # systemctl status libvirtd
   # systemctl start libvirtd
   # systemctl enable libvirtd
   # systemctl status libvirtd
   ```

5. If using **Ubuntu** or **Centos** please log in with **GNOME Xorg** not **GNOME Wayland**.

6. If using **openSUSE** you may have to add **export LIBVIRT_DEFAULT_URI=qemu:///system** to `/etc/bash.bashrc.local`.

   This will allow a regular user to administer all the virtual machines.

   ```
   # echo "export LIBVIRT_DEFAULT_URI=qemu:///system"  >> /etc/bash.bashrc.local
   ```

7. If challenged for the administrator password using **virsh** commands similar to the following screenshot.

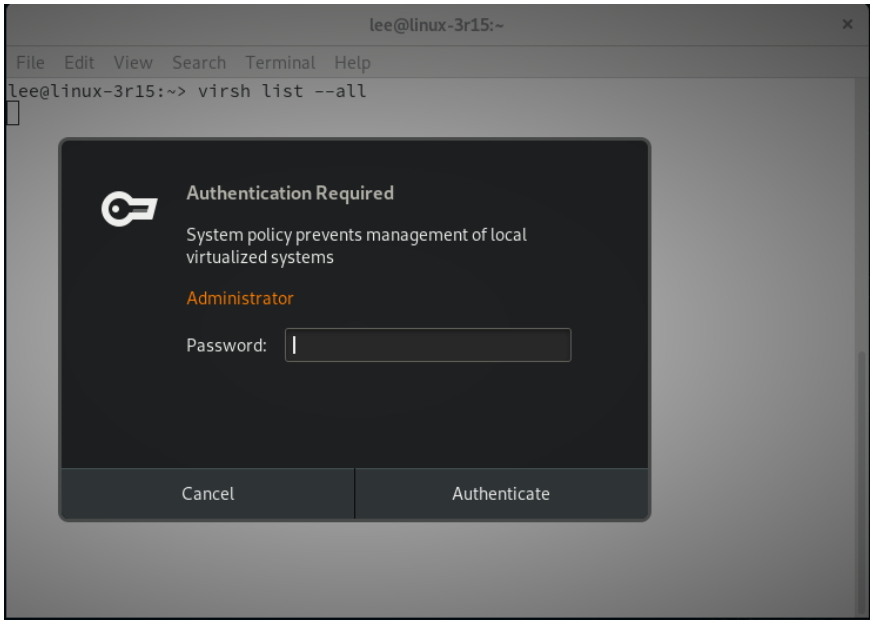


Figure 1.6: **libvirt authentication prompt**

Set the **unix_sock_group** variable in `/etc/libvirt/libvirtd.conf` to **libvirt**, it may be added to the bottom of the file like:

```
# echo 'unix_sock_group = "libvirt" ' >> /etc/libvirt/libvirtd.conf
```

Then add the additional supplemental group **libvirt** group to your user:

```
# usermod -a -G libvirt <username>
```

Exercise 1.2: Create and run a Virtual Machine based on an image

Often a virtual disk is used for distributing a pre-configured environment, for this exercise a pre-installed operating system image file is available in the **knk-resources.tar** file. The image file is called **tiny2.vmdk**.

This exercise will create a virtual machine using an **xml** file as input to **virsh**.

The parameters for the virtual machine are:

- VM name is **tiny2**
- 1 virtual CPU
- 512M memory
- the virtual disk is `/var/lib/libvirt/images/tiny2.vmdk`
- use **virt-viewer** to access the virtual machine

In this exercise a pre-installed disk `tiny2.vmdk` contains a operating system and an application to be made available. A **virsh define** command is to be used with a **xml** configuration file.

Solution 1.2

1. Create a directory to store the exercise resources in and extract the tarball.

```
# mkdir  /var/tmp/knk
# cd /var/tmp/knk
# tar -xvf <download directory>/knk-resources.tar
```


2. Create a copy of the disk image and store it at /var/lib/libvirt/images/tiny2.vmdk

```
# cp /var/tmp/knk/tiny2.vmdk  /var/lib/libvirt/images/
```

3. **Note:** As an added extra credit adventure, some distro's do not support writing to a **vmdk** image file. If **CentOS7** is being used for the host, the tiny2.vmdk must be converted to qcow2. The conversion is done with the **qemu-img** command.

   This will convert the file:

```
# cd /var/lib/libvirt/images/
# qemu-img convert -O qcow2 tiny2.vmdk tiny2.qcow2
```

   You will have to remember to adjust the file name in the rest of the lab exercise.

4. Create an **xml** configuration file. (sample files are included in the /var/tmp/knk/ directory)

   Note: There are three xml sample files in the resources directory, the names relate to the distro the host computer is running:

   - tiny2.xml-ubuntu
   - tiny2.xml-centos
   - tiny2.xml-suse

   Copy or rename the appopiate **xml** file to be **tiny2.xml**

```
# cp tiny2.xml-ubuntu tiny2.xml
```

   The name and extension is not manditory but consistant naming helps with the exercise.

   Please feel free to compare the xml files as they illustrate minor changes in the distributions.

   An example of the configuratioin file is:

```
<domain type='kvm'>
<name>tiny2</name>
  <memory unit='KiB'>524288</memory>
  <os>
    <type arch='x86_64' machine='pc-i440fx-2.11'>hvm</type>
  </os>
  <devices>
    <emulator>/usr/bin/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2'/>
      <source file='/var/lib/libvirt/images/tiny2.qcow2'/>
      <target dev='hda' bus='ide'/>
      <address type='drive' controller='0' bus='0' target='0' unit='0'/>
    </disk>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <channel type='spicevmc'>
      <target type='virtio' name='com.redhat.spice.0'/>
      <address type='virtio-serial' controller='0' bus='0' port='1'/>
    </channel>
     <graphics type='spice' autoport='yes'>
      <listen type='address'/>
      <image compression='off'/>
    </graphics>
  </devices>
</domain>
```

5. define the virtual machine

```
# virsh define tiny2.xml
```

6. verify the configuration file

   ```
   # virsh dumpxml tiny2
   ```

7. start the virtual machine

   ```
   # virsh start tiny2
   ```

8. connect to the virtual machine

   ```
   # virt-viewer tiny2
   ```

9. exit the virt-viewer and verify the state of the VM

   ```
   # virsh list --all
   ```

10. try a graceful shutdown of the VM, success or failure will depend on the OS

    ```
    # virsh shutdown tiny2
    ```

11. issue a forced shutdown of the VM if the graceful shutdown did not work

    ```
    # virsh destroy tiny2
    ```

Exercise 1.3: Create a Virtual Machine using virt-install

In the previous exercise **virsh** was used to create the configuration using an xml file as input.

The object of this exercise is to simplify the install procedure by using the **virt-install** tool.

One feature of **virt-install** is the automatic configuration of a default network connection. This feature can be overridden if desired. In this exercise the default network connection will be used.

The parameters for the virtual machine are:

- name is **tiny3**
- 1 virtual CPU
- 512M memory
- the virtual disk is `/var/lib/libvirt/images/tiny3.vmdk`, a copy of the previously used **tiny2.vmdk**
- use a network connection called **default**
- use **virt-viewer** to access the virtual machine

Once the **VM** has been created and tested that it runs, compare the active configuration that **libvirt** has stored. The configuration created by **virt-install** will have many more default options enabled, including an active network connection NAT'd to the default adapter on the host computer.

Solution 1.3

1. Create a copy of **tiny2.vmdk** and place it in `/var/lib/libvirt/images/tiny3.vmdk`

   ```
   # cp /var/lib/libvirt/images/tiny2.vmdk /var/lib/libvirt/images/tiny3.vmdk
   ```

2. Confirm or create a user called **dnsmasq**, it is used by libvirt.

   ```
   # grep dnsmasq /etc/passwd
   ```

   if the user **dnsmasq** does not exist, create it.

   ```
   # useradd dnsmasq
   ```


3. Verify a virtual network called **default** exists and is set to autostart.

```
# virsh net-list --all
 Name                 State      Autostart     Persistent
----------------------------------------------------------
```

If the **default** network does not exist, use **virsh** to define the network from the xml file `default-net.xml`.

The contents of the `default-net.xml` are:

```
<network>
  <name>default</name>
  <uuid>96ea6db6-cfb9-48fa-ad6a-bd21ab81c0d3</uuid>
  <forward mode='nat'>
    <nat>
      <port start='1024' end='65535'/>
    </nat>
  </forward>
  <bridge name='virbr0' stp='on' delay='0'/>
  <mac address='52:54:00:95:cb:bd'/>
  <domain name='default'/>
  <ip address='192.168.122.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.122.128' end='192.168.122.254'/>
    </dhcp>
  </ip>
</network>
```

Based on the default-net.xml file, the command to define the network is:

```
# virsh net-define default-net.xml
Network default defined from default-net.xml
```

Then set the network to autostart:

```
# virsh net-autostart default
Network default marked as autostarted
```

Verify the network looks good.

```
# virsh net-list --all
 Name                 State      Autostart     Persistent
----------------------------------------------------------
 default              inactive   yes           yes
```

Start the new network.

```
# virsh net-start default
```

4. Use the virt-install command to **import** the existing virtual machine's disk into a new configuration.

```
# virt-install \
              --name tiny3 \
              --memory 512 \
              --disk /var/lib/libvirt/images/tiny3.vmdk \
              --os-type linux --os-variant generic  \
              --import
```

5. Examine the active configurations stored by **libvirt**.

```
# virsh dumpxml tiny3 > /tmp/tiny3.xml.dump
# virsh dumpxml tiny2 > /tmp/tiny2.xml.dump
```

Use commands like diff and ls to see the difference in the xml.dump files.

Exercise 1.4: Create a VM and install from a CD-rom image

This exercise will demonstrate installation of an operating system from a CD-rom image using **virt-install**. The CD-rom image is available in resources tar file installed earlier. Since installation of operating systems can take a long time and require access to download updates or repository data a small self contained OS was chosen for this exercise.

Note: During the initial start up, a default of a text interface has been added. This avoids some interesting issues with the mouse during and after installation of Tiny Core Linux. This change has bee added specifically for the confrence environment. If a graphics interface is desire, the **TinyCore-Plus** installation media is recomended.

The parameters for the virtual machine are:

- name is **tinycore**
- 1 virtual CPU
- 512M memory
- 100M new virtual disk image
- the virtual disk is `/var/lib/libvirt/images/tinycore.qcow2`
- the virtual cd is `/var/tmp/knk/k-n-k-3.iso`
- use **virt-viewer** to access the virtual machine

1. Copy the CD-rom image to `/var/lib/libvirt/images` directory.

   ```
   # cp /var/tmp/knk/k-n-k-3.iso /var/lib/libvirt/images/k-n-k-3.iso
   ```

2. Boot from the CD-rom using **virt-install** using the options specified below:

   ```
   $ virt-install  --name tinycore --memory 512 --vcpus 1 \
                   --cdrom /var/lib/libvirt/images/k-n-k-3.iso  \
                   --disk /var/lib/libvirt/images/tinycore.qcow2,size=0.1 \
                   --os-type linux --os-variant generic --boot useserial=on
   ```

   A **virt-viewer** window should pop open.

3. When the initial Core Plus screen appears, press the **enter** key.

   After the boot messages finish, the user friendly Tiny Core command prompt will be visible.

4. To launch the installer at the prompt type:

   ```
   $ sudo tc-install.sh
   ```

5. Please fill in the installation choices as listed below:

   - Core Installation

     ```
     c for cdrom, enter to continue
     ```

   - Install type

     ```
     f for frugal, enter to continue
     ```

   - Select Target for Installation

     ```
     1 for whole disk
     ```

   - Bootloader

     ```
     y for yes for bootloader
     ```

   - Install Extensions

     ```
     TCE/CDE Directory is left blank, press enter
     ```

   - Disk Formatting Options

     ```
     Select 3 for ext4
     ```

- Boot Options

  ```
  No additional boot options, press enter
  ```

- Last chance before destroying a data on sda

  ```
  yes, start the install
  ```

The installation should complete in a few seconds.

6. When the installation is complete type in:

   ```
   sudo reboot
   ```

Exercise 1.5: Create a Virtual Machine and install with a text console

Note: This is an optional lab not intended for completion at the conference but rather a take home exercise. You will need to download an install DVD. This example uses a **Debian 9.5.0** DVD as its source and is approximately 3.4G. The images can be found at:

`https://cdimage.debian.org/debian-cd/current/amd64/iso-dvd/`

Note: This installation **will** access the Internet to communicate with the **Debian** repository servers. To avoid the outbound connection and speed up the install, shutdown the network access on the host system before starting the installation.

The parameters for the virtual machine are:

- name is **test-loc**
- 1 virtual CPU
- 512 memory
- 1.5G disk
- the virtual disk is going to be manually created, the suggested location is `/var/lib/libvirt/images/deb9.qcow2`
- the image used as the DVD can reside anyplace there is space
- use **virt-viewer** to access the virtual machine
- the install and finished installation will be text only

Solution 1.5

1. Create a disk image with a size of **1.5GiB** and a type of **qcow2**.

   ```
   # qemu-img create -f qcow2 /var/lib/libvirt/images/deb9.qcow2 1.5g
   ```

2. Use **virt-install** to create the **VM** and start the install sequence. Answer the **OS** installation questions for a minimal configuration.

   ```
   virt-install \
          --name deb9 \
          --memory 2048 \
          --disk /var/lib/libvirt/images/deb9.qcow2 --vcpus 2 \
          --os-type linux \
          --os-variant debian9 \
          --boot useserial=on  \
          --cdrom /var/ftp/pub/iso/debian-9.5.0-amd64-DVD-1.iso
   ```

3. When the installation is completed, a **virt-viewer** window should open up with the login prompt for the new **VM**.

Exercise 1.6: Challenge exercise using virt-manager, a KVM management GUI

In the progression of configuration tools for KVM from **xml** files to **virt-install** utility and now to feature packed GUI, **virt-manager**. Everything done in our exercises can be done with **virt-manager**. If the configuration was done using libvirt (all of ours were) then you should be able to manage all of the previously created VM's.

The challenge in this exercise is to repeat the exercise steps using **virt-manager**.