Lab 7

Virtualisation

7.1 Objectives

The goal of this lab is to familiarise with virtualisation solutions and the common tasks that are involved in their operation.

7.2 Background

Virtualisation is an important technology nowadays because it finds many use-cases in real systems. The concept of virtualisation is to create the impression of a real computer, to a system that is running inside another operating system (virtual machine).

This happens by emulating all the hardware resources (cpu, memory, disk, I/O, network card, etc.) of the virtualised system, which is also known as guest. This gives the opportunity to emulate any kind of processor, even different from the processor of the physical system, also known as host.

The purpose of using a virtual machine can be:

- to provide isolation between systems for security reasons
- to have a portable OS image on a portable media (USB stick, external hard disk)
- run a legacy system on another system (eg your PowerPC-based MAC OS)
- run an OS for different architecture (eg. ARM-based Android) etc
- better use of the host resources

Since in virtualisation the CPU architecture may be different that the physical processor of the system, this process can be very slow. The reason for this is that every processor instruction of the guest system needs to be emulated, in order to be executed on the host system.

However, in the case that the CPU architecture of both the host and the guest systems is the same, there is a possibility that this execution is accelerated. Due to the importance of virtualisation, all major hardware vendors eg. Intel and AMD provide hardware support for virtualisation (VT-X, AMD-V, nested paging), so that the performance of a virtual machine is almost identical to a real system.

The are many virtualisation solutions available:

- VMPlayer from VMware
- Virtual Box from Oracle
- xen/kvm combined with qemu or bochs from Linux

Each of the products has its own characteristics, acceleration layer and its own interface. In order to provide a common interface to all the different virtualisation products, `libvirt` has been developed, which can be used together with almost any known virtualisation environment. It facilitates the installation and management of virtual machines, through two command line programs `virt-install` and `virsh`. Basically `libvirt` provides a wrapper around the low level virtualisation techniques, allowing very simple yet powerful interaction for system administrators.

In our case we are going to use `libvirt` with the `kvm` solution, which is the open-source alternative to commercial products providing a full virtualisation environment using hardware acceleration. This is not only a cost-effective solution, but as we will see it has a powerful interface which can be used to create a large scale virtualisation enviroment for production use.

KVM stands for Kernel-based Virtual Machine and is the acceleration layer of the linux kernel. As a virtual machine we are going to use `qemu`, which is a powerful simulator for many different architectures (x86, ARM, PowerPC, MIPS etc), which can take advantage of `kvm` acceleration when simulates x86 environments on x86 host systems.

Although, `qemu` can be used as an standalone product from the command line, in our examples we will see how to use it through `libvirt`.

7.3 Installation

First we need to install the required packages:

```
# apt-get install kvm libvirt0 python-libvirt virtinst virt-viewer
```

We also need to install some graphics library to be used during the installation. To this end we run the following command:

```
# apt-get install libsdl2-2.0-0
```

Afterward we need to update our kernel, which has much better support for virtualisation. To do so, download from the ASO FTP server the packages contained in the `virtualization` directory and install them using `dpkg` and reboot the system with the new kernel.

Our virtual machine requires a disk. Since the disk is a virtualised resource, the disk is usually just a file in the host system[1].

There are two main types of disks a) fixed sized and dynamically-allocated disks. As their name denotes, the fixed-sized format disk files occupy exactly their nominal size, eg. 10GB. Alternatively we can have a dynamically-allocated disk, which can start from a small size, and it will gradually grow up to its nominal size, once data are written to it. This is beneficial if we run a VM in our personal computer, eg. to run Windows inside a linux host, but we don't want to waste all the free space of our hard disk. However, it is important to notice that using this solution has an impact in the system performance when resizing the file during VM operations. Another drawback, in the case that we create many dynamically-allocated disks that their total nominal size is bigger than our free space, it might be the case that at some point, a VM-disk may require free space that will not be available, making the VM to fail.

Consequently, in our example we are going to select a fixed-size format disk. We also decide to store all our virtual disks in the /vms directory.

```
# fallocate -l 8192M /vms/vm_1.img[a]
```

[a] **Warning:** This will only work with `ext4` file system, in other filesystems we must use `dd if=/dev/zero of=/vms/vm_1.img bs=1024 count=10000000`

What is the `fallocate` command doing?

Now it's time to create our virtual machine and install an operating system on its disk. Because of a security restrictions in the network configuration we are going to use, the account we are going to use to manipulate the VMs needs to have superuser privileges.

How do we enable these privileges?

[1] It is also possible to assign a physical device file (eg. /dev/sdb) to be used as a disk for a virtual machine. This could be useful for recovering your linux system from your old laptop, but we have to make sure that the device is not mounted in the host system, otherwise the filesystem may be inconsistent.

We are going to install Debian unstable in our virtual machine. First download the image from `asoserver`:

```
# curl -k -u aso sftp://asoserver.pc.ac.upc.edu/iso/debian-7.5.0-i386-netinst.iso
```

[2]

Operating system installation is tedious task that requires effort. We have already seen so far in the course how to perform a manual installation and configuration of a linux-based operating system. An alternative way is to use the default installer of each distribution. However, even this automated way of installation requires time and human assistance in order to select different options. In a production environment where a system administrator needs to perform multiple installations of the same operating system on many machines, either physical or virtual, there is a need to fully automate this process, which is called *preseeding*. In the followin sections we are going to see what are the steps to do so.

First we need to create a text file (`aso_virt.cfg`) which will contain the answers to the options that the installer may ask:

```
d-i debian-installer/locale string en_US
d-i netcfg/get_hostname string unassigned-hostname
d-i netcfg/get_domain string unassigned-domain
d-i netcfg/hostname string debian_vm
d-i passwd/root-password password root
d-i passwd/root-password-again password root
d-i passwd/user-fullname string Virt User
d-i passwd/username string virt
d-i passwd/user-password password virt
d-i passwd/user-password-again password virt
d-i clock-setup/utc boolean true
d-i time/zone string Europe/Madrid
d-i partman-auto/disk string /dev/sda
d-i partman-auto/method string regular
d-i partman-auto/choose_recipe select atomic
d-i partman-partitioning/confirm_write_new_label boolean true
d-i partman/choose_partition select finish
d-i partman/confirm boolean true
d-i partman/confirm_nooverwrite boolean true
d-i apt-setup/use_mirror boolean false
d-i pkgsel/include string openssh-server build-essential
apt-cdrom-setup apt-setup/cdrom/set-first       boolean false
apt-cdrom-setup apt-setup/cdrom/set-next        boolean false
popularity-contest popularity-contest/participate boolean false
tasksel        tasksel/first        multiselect        standard
grub-pc        grub-pc/install_devices        multiselect        /dev/sda1
grub-installer  grub-installer/only_debian      boolean true
d-i finish-install/reboot_in_progress note
d-i debian-installer/exit/halt boolean true
```

What each of these commands do?

For your convenience you may access this file downloading:

```
# wget http://asoserver.pc.ac.upc.edu/aso_virt.cfg
```

[2]The password may be found in Lab 1.

This file needs to be read by the installer. In order to do so, we need to put it in a web-server or on a physical medium like a usb stick or a floppy disk.

In our case, we opt for the old floppy disk. First we need to create an empty virtual floppy disk named floppy.img, with 1.44MB size, and format it with fat file system. Which commands are you going to use?

Now we have to mount the virtual image in order to copy our configuration file on it.

```
# mount -t vfat floppy.img /mnt/ -o loop
# cp aso_virt.cfg /mnt/
# umount floppy.img
```

What is the purpose of **-o loop** in the mount command?

With our floppy disk created we are now ready to start the installation:

```
# virt-install -r 128 -n vm_1 - -disk /vms/vm_1.img - -disk \
> device=floppy,path=floppy.img - -cdrom debian-7.5.0-i386-netinst.iso \
> - -graphics sdl - -prompt
```

We answer yes to the questions that are asked for confirmation (due to –prompt, in order to be sure that we don't overwrite any previous vm image).

In the window that appears with the debian installer we press tab in order to modify the kernel command line:

Replace - -quiet with:

```
...preseed/file=/floppy/aso_virt.cfg
```

Next the installer asks for language, country and keyboard layout, which we skip leaving the default ones. After that the installation will proceed automatically, receiving the configuration options from the our preseeding file. The first 3 options could also be automated, by adding them in the kernel's command line, but this would require more typing :-).

Installation will take some time, so in the meanwhile we can proceed with the network configuration.

7.4 Network configuration

The VM can be connected to the network either with NAT or with a bridge. NAT is the default option and `libvirt/qemu` sets this automatically, but in a real deployment, where we want our VM to be directly part of our LAN with proper addressing the bridge option is both more performant and more useful. For this reason, we are going to see how to use this networking possibility.

In order to create a permanent bridge in our system we have to edit the file which contains the configuration of our network interfaces. Which one is it?

First let's bring our default network interface down[3]:

```
# ifdown eth0
```

Then we proceed with modifications in the configuration file, by changing the `eth0` configuration and adding a new bridge interface `br0`. The goal is to configure DHCP on the bridged interface while leaving eth0 unconfigured:

[3] be careful the interface might be different in your case

```
iface eth0 inet manual

auto br0
iface br0 inet dhcp
bridge_ports eth0
```

Now we can enable networking by bringing up the `br0` interface:

```
# ifup eth0
```

7.5 Virtual machine control

Once we finish with the installation, we can start our virtual machine with the following command:

```
# virsh start vm_1
```

How do we shutdown a vm? Reboot?

Hint: Check `virsh` man page or run the help command.

The start command will launch the `vm_1` and create a window which can be used for input and output, like sitting in front of a screen that is connected the computer that is emulated by the virtual machine. Why is this a problem in a real production environment?

In order to solve this problem we have to find another way of interacting with the VM. This is the console interface, which gives access to the guest serial console. This is an alternative interface which is used in headless servers, that instead of using a screen and keyboard for input and output, it uses a serial cable connection that gives access to that computer using another machine.

However, the operating system by default is printing the output in the screen. We can change this in the VM by as follows:

```
# sudo vi /etc/inittab
```

Uncomment the follwing line:

```
T0-23:respawn:/sbin/getty -L ttyS0 9600 vt100
```

Let's now reboot the VM in order to verify that we can access the serial console by executing:

```
# virsh console vm_1
```

At this moment we are able to connect to the VM using the serial interface, but we still require the window option. Why?

In order to make all boot messages to be printed in the console interface instead of the screen we have to instruct the bootloader to do so. For this we have to edit the file /etc/default/grub and change the corresponding lines as follows:

```
GRUB_CMDLINE_LINUX_DEFAULT="console=ttyS0"
GRUB_TERMINAL=console
```

How can you select another kernel if everything is printed in the serial console instead of the screen?

How can you change the time to boot?

Now we can inform GRUB2 about our changes to the bootloader configuration:

```
# sudo update-grub
```

Now we can reboot again the VM to verify that all boot messages are printed in the serial console. In this case we can get rid of the virtual "screen" window. For this we have to edit the VM configuration file:

```
# sudo virsh edit vm_1
```

Search for the line with the screen configuration and delete it. Now save your changes and start the VM again to verify that launching the VM doesn't spawn the new window.

How do you know that the vm is running? **Hint:** Check virsh man page.

Finally, we need to modify the VM configuration to use the bridge interface: Change entry:

```
<interface type='user'>
```

to

```
<interface type='bridge'>
<source bridge='br0'/>
```

Now our network configuration must be complete and our VM must be accessible from our network. Once we know the ip address of our VM, we can connect to it using the ssh command instead of the serial console. How can we obtain the ip address assigned to the VM?

What's the difference between ssh and serial console access?

What is the syntax of the ssh command?

7.6 Cloning a VM

For the purpose of our lab, we want to create an identical copy of our VM, so that we have to VMs that can communicate with each other, without having to install and configure everything again.

First we have to create an exact copy of the virtual machine hard disk. We can simply perform this task by copying its file to a disk named `vm_2.img`.

Then we have to copy the configuration of `vm_1`. We can get an xml file with its configuration as follows:

```
# virsh dumpxml vm_1 > vm_2.xml
```

Now let's change the xml file, so that the virtual machine name becomes `vm_2` and its virtual disk `vm_2`.

Finally, we create the VM with the command:

```
# virsh define vm_2.xml
```

After that you will be able to start and interact with `vm_2` in the same way we did before for `vm_1`.

Bibliography