Hardware virtualization with QEMU

If you want to experience the kick of looking at a Windows boot screen in a desktop window while you're working in Slackware, read this article on the advancements in open source solutions for hardware virtualization - aka *running a computer inside a computer*.

Virtualization in Linux

One of the often cited reasons people have for not completely switching to Linux, is the fact that for some Windows applications, there is no real Linux alternative. Solutions for running DOS/Windows applications in Linux have been available for quite some time though: there is the Wine Project that allows you to run individual Windows applications by translating the Windows library calls to Linux library calls. In DOSemu and DOSBox you can run old DOS based programs. DOSemu and DOSBox provide a more or less complete virtual computer environment to the DOS Operating System. For the Windows OS there are a few virtualization solutions, too.
The most prominent is the commercial VMWare which is also the fastest performer. Open Source alternatives like Bochs and Plex86 (formerly called FreeMWare) have tried, but failed to result in a workable environment.

QEMU

The new kid on the block is QEMU which is an actively developed project with a fairly large base of enthousiast followers and contributors. QEMU is able to provide a virtualized computer hardware environment on which you can install and run Windows, Linux, and lots of other Operating Systems. QEMU calls this *System Emulation*. QEMU can also operate in *User Emulation* mode, where it is able to run single Linux processes - thereby translating calls for a specific architecture into another architecture. Most notably, User Emulation is used on non-intel platforms to run a version of the Wine Emulator, so that individual Windows applications can actually be used on foreign platforms.

QEMU virtualizes a complete hardware environment, including CPU (i386, x86_64, ppc, sparc, mips, arm platforms) and peripheral devices like VGA card, network interfaces, IDE, sound and USB components, and more. This enables you to to run an unmodified operating system in QEMU. Since QEMU is a user process, crashes of the Guest Operating System do not affect the host at all. Supported operating systems include for instance Linux and BSD distributions, and the more recent releases of Microsoft Windows such as Windows 2000 and XP (and 98 and NT4 too, if you do without the acceleration support from kqemu). On the QEMU website, you can download one or more of several disk images of fully operational virtual computers. More disk images (always virtual computers installed using freely distributable Operating Systems such as FreeDOS, Linux and BSD) are available from the Free OS Zoo website.

Because QEMU source code is licensed under the GPL, a third party has used it to develop an enhanced version they sell under the name Win4Lin Pro. Modifications to the GPL-ed source code are contributed back to qemu, so both parties profit. The qemu PC emulator can be made a lot faster by loading a special driver which is able to accelerate the Virtual Machine on x86 hosts. This driver is implemented as a separate kernel module called *kqemu*. Qemu can use this to run the virtual machine (the *guest*) at nearly the raw speed of the host machine. The kqemu software has a proprietary license, but nevertheless, you're free to use it for non-commercial as well as commercial

purposes. You may distribute kqemu only with the author's explicit permission.

The strengths of QEMU are that it's evolving quite fast, it is uncomplicated to setup and use, and there is no requirement to patch your kernel in order to get it running. The fact that this is free software does not relieve you of the requirement to own valid licenses of any software - like MS Windows - you want to install in QEMU.

Unlike Xen, QEMU allows you to run closed-source Operating Systems like MS Windows as a *guest* in a virtual machine without the requirement to modify the code. Xen offers near-native speed for the OS-es that run as *guests* in a Xen virtual machine, but requires you to modify the code of the guest OS in order to get these speeds. This leaves you with Open Source OS-es within Xen for the moment.

Emulated hardware and supported Guest OS-es

The QEMU Virtual Machine emulates a set of hardware components that is independent of the real hardware on which it is running. Since Slackware runs on x86 architecture, I will limit myself to a list of emulated hardware available to Slack (other emulated architectures may have other hardware peripherals available to the Guest OS).

- IDE-Controller supporting up to 4 drives (the drives are disk images on the host computer
- IDE CDROM device (in the form of a CD ISO image, or a real CDROM device)
- Floppy disk controller supporting up to 2 drives (floppy disk images)
- Graphics card (either a Cirrus Logic GD5446 PCI, or VGA-VESA)
- PS/2 Mouse
- Ethernet network card (Realtek RTL8139 PCI or NE2000 PCI)
- A serial port (COM 1)
- A parallel port (LPT 1)
- Soundcard (Soundblaster 16 and/or ES1370)
- A USB-UHCI host controller (the Intel SB82371)

The list of Operating Systems that run inside QEMU is quite long. Here is an unofficial list of supported Guest OS-es. I have run various Linuxes (for x68 and x86_64) and Windows 98/2000/XP inside QEMU.

Performance

Recent developments in the kqemu accelerator module bring performance of QEMU on par with the commercial VMware. However, VMware offers quite a few extra features compared to QEMU. Now that you can get certain VMware software versions for free, VMware has again become a compelling alternative (although Open Source lovers will still prefer QEMU naturally!).

You should realize that running an operating system directly on the hardware is always faster than running the same operating system inside a virtual machine, because of the processor overhead that the virtualization of all hardware dictates. Even so, the guest operating system inside the QEMU virtual machine feels suprisingly responsive if you have a decent CPU and lots of RAM.

Obtaining and building QEMU

Qemu is not part of Slackware, but it is easily built from source, and ready-made packages are available from my SlackBuild repository. If you want to build yourself you can use my SlackBuild script found at the above URL, or go to the QEMU website and compile the source code yourself.

NOTE Building qemu and kqemu has changed with qemu 0.8.1.
Until qemu 0.8.0, if you want to use kqemu you need to

1. unpack the qemu source archive
2. change into the 'kqemu' subdirectory of the extracted directory structure
3. unpack the kqemu archive in this subdirectory
4. move up one directory level to the top of the qemu source tree and run the

   ```
   ./configure
   make
   make install
   ```

   commands to install the software (both qemu and kqemu.
   With the release of 0.8.1, building kqemu is separated from building qemu. When you want kqemu to accelerate the emulation in qemu, you

5. download and unpack qemu sources
6. configure and build qemu with kqemu support (this is the default if you run `./configure`)
7. install qemu on your computer using `make install`
8. download, unpack kqemu archive
9. configure and build kqemu, and install it on your computer using `make install`.

Running QEMU and kqemu

Running QEMU is as easy as just... running `qemu <many parameters>`. QEMU all by itself results in quite sluggish emulation which makes the experience working with the Guest OS not an exhilarating one. The accelerator module kqemu improves QEMU's emulation speed a lot. If you installed my kqemu package, several files in `/etc/` will be altered, so that kqemu's acceleration will be available to you after each boot of the computer. If you do not want to reboot, but want to use QEMU with kqemu right away, then you should run these commands once (as root) before you start QEMU:

```
mount /dev/shm
modprobe kqemu
```

You should prepare your Slackware system for the kqemu module so that it can operate as efficiently as possibble. Your computer will automatically have been setup in case you installed my Slackware packages from the URL I provided in the previous section. If you compiled everything yourself, please follow these instructions:

- The acceleration that kqemu offers works best if kqemu can map the Virtual Machine's memory pages to your computer's physical RAM. If kqemu finds a mounted RAM filesystem at `/dev/shm` it will create a hidden file there which will hold the RAM of the virtual machine. If it does not find a RAM filesystem, the VM's memory will be mapped to a disk file in your `/tmp` directory, and of course this will not nearly be as fast as working entirely in RAM.

So, add this line to `/etc/fstab`

```
none        /dev/shm        tmpfs    defaults        0 0
```

By default, adding this line in `/etc/fstab` will allow a maximum of 50% of your physical RAM to be used up by the RAM filesystem. If you want your QEMU to emulate a computer with an *X MB* amount of memory, then this amount of memory "X" must be available to qemu in `/dev/shm`, or else qemu will refuse to start. This means for instance, that if your computer has 1GB of RAM, and the above line is present in `/etc/fstab`, qemu will be able to emulate a computer with a maximum of 500MB RAM (actually, a little less than 50% is effectively available to the VM, so try to give QEMU a couple of percents less than 50% of your physical RAM).
If you need more RAM for QEMU, you must change the defaults like this:

```
none        /dev/shm        tmpfs    size=750M         0 0
```

which makes 750MB of your RAM available for use by (k)qemu.

Reserving part of your RAM to be used by `/dev/shm` and a tmpfs, does not mean that you permanently loose this amount of RAM for other applications! Only the memory in `/dev/shm` that is actually *used* by an application will become unavailable to other applications. Having a line like above in your `/etc/fstab` does in no way affect your machine and it's performance if no program uses it. So would be OK to always have such a line.

- The installation of kqemu creates a device node `/dev/kqemu` that will by default be writeable by anyone. If you don't like that, you can limit access to the device node by letting it be owned by a Linux group (for instance, create a group called *qemu*) restrict access to the device node to just that group and add yourself to that group.
If you're doing all of this yourself, you should add these lines to `/etc/rc.d/rc.local` to ensure that the device `/dev/kqemu` is available and accessible:

```
## For 2.6 kernels, as long as udev does not re-create the ##
## /dev/kqemu device node automatically, you can keep      ##
## the lines below:                                        ##
##                                                         ##
# Create the KQEMU device
if ! [ -e /dev/kqemu ]; then
  mknod /dev/kqemu c 250 0
  chmod 666 /dev/kqemu
fi
```

My kqemu package also installs a udev rule file as `/etc/udev/rules.d/50-kqemu-rule` in case you run QEMU on a 2.6 kernel, with these contents:

```
# kqemu
KERNEL=="kqemu", NAME="%k", MODE="0666"
```

If you decided to restrict access to `/dev/kqemu`, then you should modify this file to for instance

```
# kqemu
KERNEL=="kqemu", NAME="%k", MODE="0660" , GROUP="qemu"
```

.

- To make the kqemu module load on boot, you should add the following line to the file `/etc/rc.d/rc.modules`:

```
/sbin/modprobe kqemu
```

Use the following line instead for kqemu if you have UDEV:

```
/sbin/modprobe kqemu major=0
```

This will automatically create the device node `/dev/kqemu` on demand.
Note that nowadays Slackware comes with `/etc/rc.d/rc.modules` as a symbolic link to the file `/etc/rc.d/rc.modules-<kernelversion>`. You have to check carefully that you modify the *modules* file for the kernel you are currently running.

This concludes the alterations needed for a performance boost of your Virtual Machines inside QEMU. As I said earlier, running qemu is really simple - it is a single binary with a lot of optional commandline parameters that customize the virtual machine it will setup for use. QEMU will use the kqemu accelerator if it finds the kernel module loaded in memory (and if it's built with support for kqemu). QEMU provides an additional layer of acceleration called *kernel-kqemu*. In this acceleration mode, the Guest kernel processes will also be accelerated as opposed to the “regular” functionality of kqemu to only accelerate the Guest's user processes. You do need to supply an explicit parameter to the qemu commandline:

```
qemu -kernel-kqemu <other parameters>
```

If you don't want kqemu functionality at all, for instance because some programs and Guest OS-es will not work reliably or not at all with acceleration enabled, you can explicitly tell qemu on the commandline to do without:

```
qemu -no-kqemu <other parameters>
```

QEMU uses SDL for it's output (video as well as sound), and you need to run QEMU in an X Window session. There is an option to the qemu binary called `-nographic` that you can use to circumvent the X session requirement, but you still need the X and SDL libraries installed on your system. Typically, you would use `-nographic` for a virtual machine you've already installed previously, and that allows remote access through SSH, VNC or XDMP, and where you no longer need the virtual machine's local console.

Installing an Operating System in QEMU

In this section, we're going to install a Guest Operating System in QEMU. Since my guess is that many

people use QEMU to run a copy of Windows on a Linux host, I will use Windows XP as an example. The hardware that QEMU emulates (see the previous section for a list) is fully supported by at least Windows Xp/2000/NT and the few tweaks that are needed to make Windows 98/95 work well are documented in the qemu user-documentation. In my experience, Windows 2000 is an optimal compromise between speed and functionality, but Windows XP works well on fast hardware and with the latest kqemu accelerator.
The instructions and tips that are documented in this section will mostly apply to the installation of any other Operating System.

Remember that QEMU builds a virtual computer around the Operating System you're going to use inside of it. So, the hard drives are virtualized as well. So, before starting QEMU, we will create a big file that QEMU can then use as the virtual hard disk:

```
dd if=/dev/zero of=winxp.img bs=1k count=0 seek=4000000
```

This will create a so-called *sparse file* with a size of about 4GB. A sparse file means that it can ultimately grow to 4GB of occupied disk blocks, but the file will only allocate enogh blocks to store the actual data. So, if you run `du` on that directory, it will report 4GB used, but actually, the file will currently use no space at all! Now, a real file using up 4GB of disk space before you even wrote to it, can be created using this command (it's up to you to decide what you want to use, either command is OK):

```
dd if=/dev/zero of=winxp.img bs=1k count=4000000
```

I think that 4GB is enough to hold Windows XP and several large applications, but if you think you need more, just change the `4000000` into a bigger value.

Now, to install Windows, you need a Windows installation CD (and a valid license key of course). You have two options; if you have a “real” CDROM medium, you can let QEMU use the physical drive. If all you have is an ISO image of the installation CD, then QEMU can present this ISO image as a CDROM drive to the virtual machine. We will use an ISO file in this excercise, and assume you keep it stored as `./ISOS/winxp_pro_us.iso`.

This is the big moment... we're going to fire up QEMU and tell it to use the 4GB empty file we have just created as the virtual computer's first hard drive, and use an ISO image to represent as a CD-ROM drive in the virtual machime. QEMU should also boot from that CD. Run

```
qemu -localtime -m 256 -hda winxp.img -cdrom ./ISOS/winxp_pro_us.iso -boot d
```

Alternatively, as an example, the commandline would have looked like this if we had chosen to use a real CD in the CDROM drive `/dev/cdrom`:

```
qemu -localtime -m 256 -hda winxp.img -cdrom /dev/cdrom -boot d
```

. The `-boot d` parameter instructs QEMU to let the virtual machine boot from the (virtual) CD-ROM drive we've specified using *-cdrom*. The '-hda winxp.img' of course means that the first IDE harddisk is to be represented by our image file (QEMU can use a hdb, hdc and hdd as well, in case you need more hard disks inside QEMU).

When you run the above command, you'll see a new window appear, and you'll actually see the Windows XP installer boot (assuming you didn't mess up during one of the previous steps)! To use the

mouse in QEMU, you need to click in the window to give it focus. QEMU grabs the mouse, and to give it back to your other applications you press Ctrl+Alt (it even says so in QEMU's window bar). Another key combination worth mentioning is the full-screen toggle Ctrl+Alt+F. If QEMU is running full-screen you can fool anyone that you're suddenly running the evil OS 🙂
The installation of XP is a process that I'm not going to repeat, I am assuming you'll know how to proceed. What you need to be aware of, is that the installation of Windows 2000/XP in QEMU still has a couple of annoyances. These are:

- Excruciatingly slow progress from time to time, especially during the stage where Windows tries to detect the hardware in your system.
- Errors of “disk full” half-way the install. QEMU even knows a separate commandline parameter `-win2k-hack` that you should add exclusively for the duration of a Windows 2000 installation if you experience *disk full errors* during the installation
- Windows XP sometimes installs without a problem but at the first boot displays the following message:

  ```
  A problem is preventing Windows from accurately checking the
  license for this computer. Error code: 0x800703e6.
  ```

  If you see this, you have bad luck. You can get around this by booting XP in *Safe Mode* without networking support.

Even though the slow pace of the Windows installation can make you wonder if you did the right thing installing QEMU, these annoyances will be gone once Windows installation has finished and you reboot into the OS for the first time.

When you're done with installing from CD, do not forget to at least remove the `-boot d` commandline parameter, so that the virtual computer will start from it's hard drive instead of booting from the CDROM.

Networking your virtual machine

User mode networking

By default, QEMU uses a feature which is called *user-mode network*. QEMU will run an internal DHCP server that can assign an IP address to the virtual computer in case that is configured to use DHCP. The network range and gateway are hard-coded into QEMU but it allows your virtual machine to call out to your host machine and beyond. For user-mode networking, you don't need to configure anything on your host.
QEMU will enable user mode networking by default if you do not pass it any network parameters at all, or if you start it like this:

```
qemu -net nic -net user <other qemu parameters>
```

The network characteristics for user mode networking are as follows:

Gateway/DHCP/TFTP server:	10.0.2.2

DNS server:	10.0.2.3
Samba server:	10.0.2.4
Netmask:	255.255.255.0
Guest IP:	any address above 10.0.2.15

There are limitations to the user mode networking due to the nature of the implementation.

- QEMU will act as a firewall between guest OS and the host computer, so that no network communication is possible from any host program to the guest OS. For instance, you will not be able to setup a ssh session to the guest.
  To get around this dilemma, QEMU has a "`-redir`" optional argument which enables you to redirect certain ports on the host. Traffic destined for these ports will end up at the guest. For example, let's redirect port 22000 on the localhost to port 22 in the guest, in order to setup a ssh session into the guest:

  ```
  qemu -redir tcp:22000::22 <other qemu options>
  ```

  If the guest is running a SSH server at port 22, then it is possible to connect a ssh session on the host to the guest's SSH server like this:

  ```
  ssh -p 22000 localhost
  ```

- There is actually no proper network connection between the guest and the world outside the Virtual Machine. QEMU will intercept TCP and UDP packets from the guest, dissect them and pass their data payload on to destination computers as if QEMU itself were sending the data. In reverse, QEMU will grab the return traffic and re-assemble TCP and UDP packets for the guest OS. This private implementation of a network stack in QEMU results in TCP and UDP traffic working transparently for the guest OS, but ICMP packets (ping, traceroute for instance) will not be able to pass the boundary. This should not be a concern to you. However lots of people will run a `ping` as their first test of the network in QEMU's guest OS and are led to believe that the guest's network is not functional because they do not see a ping response.

Connecting multiple guests to a VLAN

QEMU knows the concept of VLAN's. Think of a VLAN as a virtual switch, emulated by QEMU, to which you connect your guest. You typically use a vlan to connect multiple guests into a virtual network. By default, QEMU assigns *vlan=0* to your guest if you do not specify a vlan number. So, the following two commands are basically the same:

```
qemu -net nic,vlan=0 -net user <other qemu parameters>
qemu <other qemu parameters>
```

In addition, I advise you to assign your guest OS's network card a fixed MAC address. If you do not assign a MAC address, QEMU will randomly pick a value and many OS-es will not like that (for instance, Slackware's UDEV will create new interfaces eth1, eth2, ... everytime it finds a new MAC address for your interface card at boot). You can assign a MAC address as follows (you can make up any value that is valid, i.e. 6 bytes separated by colons, in hexadecimal representation):

```
qemu -net nic,vlan=0,macaddr=51:45:4d:55:00:01
```

When several guests are connected using the same vlan number, any network packet that is sent by one of the guests, will be distributed over the vlan to all of the other guests. This allows for transparent network communication between the guests.
NOTE this has nothing to do with the ability of the guests to connect to the world outside QEMU!

Virtual Distributed Ethernet (VDE)

To overcome the limitations of *user mode networking*, I suggest you read my article on VDE where I show how you can use VDE (virtual distributed ethernet) in combination with dnsmasq to get a much enhanced network experience. Using VDE, you will be able to make your QEMU guest fully accessible to programs running on your host, and if you bridge your network connections, you can make your Virtual Machines appear on your LAN as if they were real machines.

Advanced topics

So far we have been through the bare minimum to get your Guest OS installed and working on your host computer and interacting with the network. Much more is possible with QEMU, and I will dedicate this chapter to the advanced topics that are most interesting or least wel-documented in other places. I will also assume you're running a recent version of QEMU. The emulator advances in great strides, and some of the features and command line switches I talk about are not available in all releases. Generally speaking, try to upgrade to the latest official release. At the point of writing, I am using 0.8.1.

Using copy-on-write files

QEMU uses a disk image file containing your installed Operating System. We created that image file in a previous section as a *raw* image, meaning the file has no special format. QEMU knows of several disk image formats, and one of them I want to introduce to you: the *QCOW* or ““qemy copy-on-write” format. The QCOW file can be used in combination with a *base image file*. Suppose you're happy with the installed OS in your `winxp.img` disk image file. You want to try out a new application, or you want to do some potentially disastrous tests in your virtual machine. You don't want to risk the possibility to thrash your installed OS and want a safe way to test with the option to revert to the original state if things go terribly wrong.
You can then create a QCOW file that is based on the `winxp.img` file. Create the QCOW (qemu copy-on-write) file like this:

```
qemu-img create -b winxp.img -f qcow winxp.qcow
```

You then change the qemu commandline from

```
qemu -localtime -m 256 -hda winxp.img
```

to

```
qemu -localtime -m 256 -hda winxp.qcow
```

to use your new QCOW file.
When you tell QEMU to use the QCOW file instead of the RAW image file, QEMU will do all file writes to the QCOW image, thus treating the original RAW file as a *read-only* file. The effect is, that if you stop QEMU after you did some work, the original image file `winxp.img` is *unaltered* and all changes are recorded into the QCOW file `winxp.qcow`. If you delete the QCOW subsequently and re-start QEMU with the original commandline using the original `winxp.img` file, there will be no trace left of the previous QEMU session you ran using the QCOW file.
So, this is ideal for testing stuff out - in fact, this is how I test and build my new Slackware packages. When you're happy with the result, you don't delete the QCOW file, bit instead commit the changes recorded in that file back to the base image file:

```
qemu-img commit winxp.qcow
```

If you want, you can get rid of the now emptied QCOW file.
Read the qemu-img man page if you want to know more about the supported image file formats (some support compression and encryption!).

The qemu monitor

Besides the graphical screen, QEMU has a *monitor* screen. You can toggle the QEMU output from graphics to monitor mode (after giving the window focus by clicking in it with the mouse) by pressing Ctrl+Alt+2 and back to graphics mode by pressing Ctrl+Alt+1. The QEMU monitor is used for several purposes:

- Remove or insert removable medias images (such as CD-ROM or floppies)
- Freeze/unfreeze the Virtual Machine (VM) and save or restore its state from a disk file.
- Inspect the virtual machine state without an external debugger.
- Send key presses to the guest OS running inside the virtual machine.

Changing a CD/floppy in the virtual machine

An interesting feature of the monitor from an end user perspective is the ability to change removable media images. Two scenarios:

1. Suppose you want to install Slackware in QEMU. You start QEMU with something like

   ```
   qemu -localtime -m 256 -hda slackware.img -cdrom ./ISOS/slackware-10.2-install-d1.iso -boot d
   ```

   and after Slackware finishes with CD1 it will ask for CD2. How does one “change the CD” in QEMU? This is simple; switch to the monitor by pressing Ctrl+Alt+2 and run these commands (the `help` monitor command is always at hand if you should forget what to do)

   ```
   eject -f cdrom
   change cdrom ./ISOS/slackware-10.2-install-d2.iso
   ```

   i.e. supply the full path to the ISO image on the Host filesystem.

2. You are running QEMU and decide you need a CDROM to install a piece of software on the

currently running OS. Unfortunately, you did not start qemu with the parameter `-cdrom /dev/cdrom` or currently have another CDROM medium in the drive. Neither do you want to stop your QEMU session to change the CD in the drive. In this case too, you switch to the QEMU monitor. If there already was a CDROM in the drive, you first run

```
eject -f cdrom
```

Next, you change/insert the physical medium for the CD you want to use in the running QEMU. Finally, make the new CDROM known to the guest OS in QEMU:

```
change cdrom /dev/cdrom
```

This is enough to make the new drive visible in the guest OS.

Switch back out of the monitor to the Guest using Ctrl+Alt+1.

Send key presses

In some cases, the host OS, or the host hardware, won't let you use certain keyboard combinations. Two examples:

1. You own an Apple computer. The PowerBook for instance does not have the Del key - and Windows really needs this key for to compose Ctrl+Alt+Del combination in order to login to the computer. The solution is to switch to the QEMU monitor using Ctrl+Alt+2 keypresses and then run the *literal* command

```
sendkey ctrl-alt-delete
```

1. You're running X on the host as well as on the guest. In the guest, you need to run a keyboard sequence containing Ctrl+Alt - like, you want to switch to a console using Ctrl+Alt+F6 or end your X session using Ctrl+Alt+Backspace. The host will intercept this key combination even if the QEMU window has focus. The solution here is to switch to the QEMU monitor (Ctrl+Alt+2) and then run the *literal* command - for instance

```
sendkey ctrl-alt-f6
```

By 'literal' i mean that you type the characters exactly as I showed you.

Mounting a QEMU disk image on the Host

On the host, you can mount filesystems created in partitions that are present in QEMU's virtual disk image - and access the files contained in there - under certain conditions:

1. QEMU most **not** be running when you mount the virtual disk partitions!
2. The disk image must have a *RAW* format. Other formats, such as QCOW for instance, can not be accessed from outside QEMU.

A QEMU disk image file will contain one or more partitions. We need to find out at what location in the file these partitions start, so that we can tell the `mount -o loop` command at which file offset to look. The `fdisk -ul` command will show us exactly that type of information. The '-u' parameter will output sector count. For example:

```
$ /sbin/fdisk -lu slack102.img
       You must set cylinders.
       You can do this from the extra functions menu.

       Disk slack102.img: 0 MB, 0 bytes
       255 heads, 63 sectors/track, 0 cylinders, total 0 sectors
       Units = sectors of 1 * 512 = 512 bytes

              Device Boot      Start         End      Blocks   Id  System
       slack102.img1   *          63      208844      104391   83  Linux
       slack102.img2          208845      401624       96390   82  Linux
swap
       slack102.img3          401625     4192964     1895670   83  Linux
```

You can safely ignore the warnings about the “set cylinders”, they do not matter here.
The output shows a disk image consisting (apart from a swap partition) of two *Linux* partitions, which probably contain the / and /home filesystems. The first partition starts at sector 63 while the other data partition (partition #3) starts at sector 401625. These are the offsets we have to multiply with 512 (the sector byte size) to get the offset number we will feed to the `mount` command. Proceed with mounting the two partitions (I create the mount points too for completeness sake):

```
mkdir -p /mnt/qemu/part1 /mnt/qemu/part2
mount -o loop,offset=$(( 63 * 512 )) -t auto slack102.img /mnt/qemu/part1
mount -o loop,offset=$(( 401625 * 512 )) -t auto slack102.img
/mnt/qemu/part2
```

I assume here, that setting the filesystem type to “auto” will work for you. If not, and you know what type the filesystems are (for instance *ext3*), then you can change the ”-t auto“ to ”-t ext3“ in these two mount commands.
Once mounted, you can copy files from and to the disk image.

If you *do* want to be able to access your data stored in a QEMU partition, and still use QCOW for the advantages that file format offers, I suggest you create two disk image files, and use QCOW on the disk image where you will install your Guest Operating System. Create the second disk image as a RAW file, and use this to for a filesystem where you will keep your data. That way, at least the data remains accessible in the Host.

Accessing the host parallel port

If you need to print to a printer that is attached to the host computer's parallel port, you will have to pass an additional parameter on the qemu command line, like this:

```
qemu -parallel /dev/parport0 <other qemu parameters>
```

You might not have this device node `/dev/parport0` available on your host, but QEMU requires the parallel port device name to start with the string *parport*. If your parallel port is called *lp0* you can easily create a symbolic link called “parport0” to that device by running

```
cd /dev
ln -s lp0 parport0
```

Printing to the Host's CUPS server

The Guest OS can print to a CUPS printer that you configured on the Host. No other software on the Host or the Guest is needed. The Guest's printing system must be able to use *IPP*, the “Internet Printing Protocol”. This is the CUPS native protocol.

If your Guest is Windows 2000/XP, you need to do the following:

1. Click on “add printer” from the printer list window;
2. Select “networkprinter or a printer connected to another computer”;
3. Pick “Connect to a printer on the Internet ...” and enter the correct URL;
4. Install the driver for your printer (you will need to have the driver setup program ready if it is not supported by Windows by default);

   The URL for your CUPS printer will look like

   ```
   http://10.111.111.254:631/printers/lp0
   ```

   where the address `10.111.111.254` is the IP address for the Host and `lp0` is the name of the printer that you defined in CUPS. Naturally, you will need to substitute values that are appropriate for your own computer!
   If you use the `rc.vdenetwork` init script I documented in my QEMU/VDE network page then `10.111.111.254` will actually be the correct IP address to use.

If your Guest is Slackware Linux, then you need to setup the CUPS client in the Guest. I assume that you already installed the CUPS package when you installed Linux in the Guest. This is what you should do:

1. Edit the file `/etc/cups/client.conf` and look for the line that says ”`#ServerName myhost.domain.com`“. Change that line into

   ```
   ServerName 10.111.111.254
   ```

   where you supply a correct IP address for your host. If you use the `rc.vdenetwork` init script I documented in my QEMU/VDE network page then `10.111.111.254` will actually be the correct IP address to use;

2. Make the CUPS init script executable so that CUPS will start automatically when the Guest boots:

```
chmod +x /etc/rc.d/rc.cups
```

3. Reboot the Guest or (re-) start CUPS:

```
/etc/rc.d/rc.cups restart
```

4. Your Guest should now be able to use your Host CUPS printer.

In case you configured your Host's CUPS server to make it visible to other CUPS clients on your network (read my "setting up CUPS" page), then the above CUPS client configuration won't be necessary. The Host's CUPS printer will automatically appear - ready for use - in your Guest even if CUPS is running in its default configuration.

Sound support in the Guest

QEMU has pretty good sound support. It emulates several sound hardcards. You can print a list of the emulated cards by running

```
qemu -soundhw ?
```

Support for the Adlib sound card in QEMU is not enabled by default (Soundblaster 16 and Ensoniq ES1370 are available in the VM by default) You can add support for Adlib to the emulator when you build the software, by adding

```
--enable-adlib
```

to the `configure` command. If you want information about the tunable audio parameters, run

```
qemu -audio-help
```

For sound output, QEMU uses SDL which in turn uses OSS. Slackware uses the ALSA sound driver which has OSS emulation support. Since version 0.8.0, QEMU can use ALSA directly for sound output, but you will have to explicitly enable this when building the sofware. Add

```
--enable-alsa
```

to the `configure` command.
The scripts in the final section of this Wiki page show you how to setup your environment so that you can actually hear the sound that QEMU outputs. The requirement for sound to come from your speakers is the environment variable `QEMU_AUDIO_DRV` which must have a value of either ”`sdl`“ or ”`oss`“, or in case you compiled-in ALSA support, it may have a value of ”`alsa`“ as well.

Using an USB device in QEMU

USB devices that are connected to the host machine can be accessed directly inside the Virtual Machine. QEMU needs an aditional parameter `-usb` to actually enable USB support in the VM. The

emulated Intel SB82371 UHCI-Controller has a 8-port USB hub. If you want access to one of your physical devices, you will need to find out it's *Vendor-ID* and *Product-ID*. This information is obtained by examining the output of

```
/sbin/lsusb
```

or

```
cat /proc/bus/usb/devices
```

. You can either tell QEMU to make the device available by looking up the *VendorID* and *ProductID* and passing it on the commandline

```
qemu -usb -usbdevice host:<VendorID>:<ProductID> <other_parameters>
```

or starting QEMU just with USB support enabled:

```
qemu -usb <other_parameters>
```

After booting the guest OS in the VM, switch to the monitor by pressing Ctrl+Alt+2 and enter the following command

```
usb_add host:<VendorID>:<ProductID>
```

When you return to the graphical screen of the Guest by pressing Ctrl+Alt+1 you will see a USB “device attach” event in the messagelog (Linux) or on-screen (Windows).

- An example: You have a HP Scanjet 3300C connected to the USB port of your computer. The output of `lsusb` is

  ```
  # lsusb
  Bus 003 Device 002: ID 03f0:0205 ScanJet 3300C
  ```

  The command that you use to make this scanner accessible in QEMU is

  ```
  qemu -usb -usbdevice host:03f0:0205 <other_parameters>
  ```

The important thing to consider with using real USB devices, is that the Host must not have any driver loaded for the USB device. If your hotplug loads drivers automatically, you can enter the driver's name into the file `/etc/hostplug/blacklist` to make sure that hotplug will leave it alone. In the QEMU dccumentation I noticed that this is supposedly no longer an issue with (recent) 2.6 kernels. YMMV as I have not yet tried this.

PXE booting your QEMU virtual machine

In another article on this Wiki, i talk about the advantages of network booting (PXE boot) when you are installing Slackware. Booting the Slackware installer from the network can sometimes be the only way of installing Slackware - imagine a PC without floppy and CDROM drives. QEMU on the other hand, has no need for network boot since you can just create an ISO image and pass that as the "*-cdrom*" parameter and that's it.
Yet there are times when you'd want your QEMU virtual machine were able to use PXE to boot from the network, if only to test your network setup without sacrificing real hardware. There is only one problem, the emulated QEMU network card (Realtek 8029, a NE2000 clone) does not support booting from the network (it does not emulate a boot ROM).

There are two ways to overcome this problem.

- The first is to use the ROM-o-Matic. ROM-o-matic.net dynamically generates Etherboot ROM images. We will let it create a PXE-capable ISO image that has support for our emulated Realtec network card. We can then use this small (~130kB) ISO image with the "*-cdrom*" parameter to QEMU to let it serve as a "replacement" PXE-boot ROM for the emulated network card.
  The ISO generation works as follows - on the ROM-o-Matic start page,
  - Select *latest production release*
  - Choose NIC/ROM type **ns8390:rtl8029 - [0x10ec,0x8029]**
  - Choose ROM output format **ISO bootable image without legacy floppy emulation (.iso)**
  - To generate and download a ROM image press: Get ROM
    No further customizations are needed, the default options are exactly right for our purposes.
  - Save the generated ISO file on your local hard disk (for instance as *qemu_pxeboot.iso*)

    To use this ISO to boot from your PXE server, run qemu as follows:

    ```
    qemu -cdrom qemu_pxeboot.iso -boot d <other options>
    ```

    Read the article on installing Slackware using PXE if you want to know more about how to setup a full-blown PXE server.

- The other option you have is available since QEMU 0.9.0 and that is to use the *`-option-rom`* parameter. As of the 0.9.0 release, QEMU can do a PXE boot using one of the PXE-capable boot roms available in `/usr/share/qemu`:
  - /usr/share/qemu/pxe-ne2k_pci.bin
  - /usr/share/qemu/pxe-pcnet.bin
  - /usr/share/qemu/pxe-rtl8139.bin
    In order to support booting from the network, the `-boot` parameter has been extended with the possible value *n* for "network". An example of the command to PXE-boot your QEMU Virtual Machine from the network follows:

    ```
    qemu -localtime -m 256 -hda slackware.img -boot n -option-rom
    /usr/share/qemu/pxe-ne2k_pci.bin <other_parameters>
    ```

If you read the Wiki article on VDE in order to improve the networking support for QEMU, and are running dnsmasq using the example `rc.vdenetwork` script, you can easily add support for network booting to the script. Look in the script for

```
DNSMASQ_OPTIONS=""
```

and change it into

```
DNSMASQ_OPTIONS="--dhcp-boot=/slackware-11.0/pxelinux.0,\"10.111.111.254\",10.111.111.254"
```

The two assumptions here (which you are free to change of course) are:

- QEMU sees your *host* as having the IP Address `10.111.111.254`
- Your *host* has a TFTP server setup so that `/slackware-11.0/pxelinux.0` is the location under the tftproot directory where the PXE bootloader `pxelinux.0` can be downloaded.

If your TFTP server is not installed on the *host* but somewhere on the “real” network, you're in a bit of a problem: the PXE client and the TFTP server are separated by a NAT firewall!
Actually, there is no problem if you are using VDE for the network support. The `rc.vdenetwork` script installs a few basic iptables rules that make your *host* act as the NAT (masquerading) router for the QEMU *guests*. The TFTP requests will not be able to pass the NAT (which causes Slackware's *tftp-hpa* server to log errors like `in.tftpd[8146]: tftpd: read(ack): Connection refused`).
We need a some extra work on the iptables firewall here, to let the TFTP traffic pass along. Add the following commands to the `rc.vdenetwork` script, or another place if your firewall is not setup there:

```
modprobe ipt_state
modprobe ipt_helper
modprobe ip_conntrack_tftp
modprobe ip_nat_tftp
iptables -A INPUT -m helper --helper tftp -j ACCEPT
iptables -A OUTPUT -m helper --helper tftp -j ACCEPT
```

after which the QEMU VM will find the TFTP server and start downloading the bootcode, and the Operating System.

Copy/paste text between Host and Guest

Sharing files between Host and Guest

Fix Me!

Code Text

A QEMU start script

For ease of use, we will create a script that will start QEMU with a lot of default options. The script will write the output into a *.log* and *.err* file and then change into a background process, so that you can just start the script in a terminal and get your prompt back, or add it to your Window Manager menu. Call the script for instance `start_winxp.sh` and make it executable.

```
#!/bin/sh
#
# Start Windows XP Pro in QEMU

PARAMS=$*

# Qemu can use SDL sound instead of the default OSS
export QEMU_AUDIO_DRV=sdl


# Whereas SDL can play through alsa:
export SDL_AUDIODRIVER=alsa

# Change this to the directory where _you_ keep your QEMU images:
IMAGEDIR=/home/alien/QEMU

# Change this to the directory where _you_ keep your installation CDROM's
ISO images:
ISODIR=/home/alien/ISOS

# Now, change directory to your image directory
cd $IMAGEDIR

# If you want the WinXP CD available, use a '-cdrom' parameter:
qemu -net user,vlan=0 -net nic,vlan=0 -m 256 -localtime -soundhw all -hda
winxp.img -cdrom ${ISODIR}/winxp_pro_us.iso  1>winxp.log 2>winxp.err
${PARAMS} &
```

A script that uses VDE for a better networking support, would look like this:

```
#!/bin/sh
#
# Start Windows XP Pro in QEMU using VDE for better network support

PARAMS=$*

# Qemu can use SDL sound instead of the default OSS
export QEMU_AUDIO_DRV=sdl
```

```
# Whereas SDL can play through alsa:
export SDL_AUDIODRIVER=alsa

# Change this to the directory where _you_ keep your QEMU images:
IMAGEDIR=/home/alien/QEMU

# Change this to the directory where _you_ keep your installation CDROM's
ISO images:
ISODIR=/home/alien/ISOS

# Now, change directory to your image directory
cd $IMAGEDIR

# If you want to boot from the WinXP CD add a '-boot d' parameter to the
commandline;
#   if you don't need the CDROM present in the VM, leave '-cdrom
${ISODIR}/winxp_pro_us.iso' out:
# I made the MAC address up - make sure it is unique on your (virtual)
network.

# This command returns to the command prompt immediately,
#   and QEMU's error output is redirected to files.
vdeqemu -net vde,vlan=0 -net nic,vlan=0 -m 256 -localtime -soundhw all -hda
winxp.img -cdrom ${ISODIR}/winxp_pro_us.iso  1>winxp.log 2>winxp.err
${PARAMS} &
```

Further info and pointers

- QEMU homepage
- QEMU user documentation
- QEMU daily CVS snapshot
- QEMU users forum
- IRC: the #qemu channel on Freenode.