Neutrino as a Guest in QEMU#

Introduction

QEMU is a portable open source emulator for x86, MIPS, ARM, PPC, and Sparc. For the purposes of QNX, this makes a handy tool for quick tests and debugging. Some supported features are snapshots, basic networking, and gdb integration. It is also possible to debug the kernel much like you can do with VMware, but for all architectures supported by QNX except for SuperHitachi.

This page describes how to set up Neutrino to run on QEMU on any host OS and make use of it's features. Installation of QEMU itself is not covered here. See either the qemu website or, for a Neutrino host, Neutrino as host for QEMU.

Installing a Neutrino System on QEMU (x86 PC)#

In any case, the first thing required is a virtual hard drive image. QEMU supports multiple image formats, but the focus here is on the raw format, because it is the fastest and easiest to work with. A raw image is so simple that you can treat it just like a block device and use utilities such as dd and dinit to set it up. To create a raw image for a QEMU system, type

```
qemu-img create -f raw <new-image-filename> <size>
```

where size can be specified in Gb, Mb, or kb, such as 3G, 1500M, 2000k.

Note:#

- The CD installer requires a minimum drive size of 2G in order to install. This is problematic for raw images on filesystems that can't handle files this large, like FAT32 and QNX4 filesystem. With a Neutrino host, you'll either have to use a different image format, or just make a custom target image. I strongly prefer the latter, because as a test target, the entire development environment is quite unnecessary and makes everything bigger.

Method 1 - Use the installer CD#

After creating an image of at least 2G in size, we can invoke QEMU like:

```
qemu -hda <hdd-image> -cdrom <physical device OR .iso file> -boot d -k en-us
```

The -boot option tells QEMU to boot from the cdrom, and the -k option specifies the keyboard layout, which seems to always be required for Neutrino. Once the installation is complete, we can boot regularly like

```
qemu -hda <hdd-image> -k en-us
```

You'll notice warnings on startup about no P & P BIOS and pci-servers. This can be easily fixed for future boots by making a new .boot file. From a terminal, change working directories to /usr/qnx632/target/qnx6/x86/boot/build and edit the file "qnxbasedma.build". In the script file, we have to start pci-bios before diskboot. The new script section should look like:

```
[+script] startup-script = {
   # To save memory make everyone use the libc in the boot image!
   # For speed (less symbolic lookups) we point to libc.so.2 instead of libc.so
   procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2
```

```
    pci-bios            # add this line because diskboot fails to start pci-bios
    waitfor /dev/pci    # make sure it worked before we continue

    # Default user programs to priorty 10, other scheduler (pri=10o)
    # Tell "diskboot" this is a hard disk boot (-b1)
    # Tell "diskboot" to use DMA on IDE drives (-D1)
    # Start 4 text consoles buy passing "-n4" to "devc-con" (-o)
    # By adding "-e" linux ext2 filesystem will be mounted as well.
    [pri=10o] PATH=/proc/boot diskboot -b1 -D1 -odevc-con,-n4 -odevc-con-hid,-n4
}
```

After saving this file, we can overwrite the .boot file like:

```
mkifs qnxbasedma.build /.boot
```

If you did everything right it should work, but you can use .altboot if you're the paranoid type.

If this is not done, networking will not work, so we have to manually start it every time. To do this, type:

```
pci-bios
slay io-net
io-net -dne2000 -ptcpip
dhcp.client
```

Running diskboot is still less than ideal for any system where we know the hardware configuration. For a much faster boot, we could replace diskboot altogether and manually start drivers and mount filesystems. The easiest way to go about this is to pass "-vv" to diskboot in the buildfile, watch the output on startup, and manually start whatever it does in the startup script. Mounting filesystems can be tricky if we don't do it the same as diskboot does. Example buildfile to come later.

Advantages of this method:#

- This is the easiest way to set up Neutrino with the full set of programs and drivers including Photon.

Disadvantages of this method:#

- Unnecessarily big and slow
- Requires large drives
- Requires entering license key
- installs all components for the full development system, when really you're probably running QEMU within a full development system and just need quick tests.

Method 2 - Buildfiles, mkifs, dinit#

It's very easy to set up a very minimal system under QEMU. Run mkifs on NeutrinoAsGuestInQEMU/qemu-minimal.build to get a shell prompt, networking, and not much else. We should also be able to make tftp or ftp work to transfer files from the host machine. The steps are:

```
mkifs qemu-minimal.build qemu-minimal.ifs
dinit -H -R -f qemu-minimal.ifs <raw-disk-image>
dinit -B pc2 <raw-disk-image>
```

The reason that we need to run dinit twice is that when it runs on a file, it always uses the floppy bootloader, which will not work. The second dinit rewrites only the boot sector with the proper hard disk loader.

Advantages of this method:#

- We can get away with really small disk images and fast boot time
- Neutrino is running much closer to how it would actually be in small embedded targets
- Makes an extremely quick and easy test for scaled down systems
- Forces the user to learn how to use build files - an invaluable skill for making real systems

Disadvantages of this method#

- If we want more programs and utilities to play with, we have to add them to the build file, or transfer them to the filesystem later. It's certainly possible to embed lots of tools in the image, but that's the choice of the user.

Communicating with the Host System#

Once we start io-net, run dhcp.client, and actually have an IP on the target, there are several ways to talk to the host. As far as the target is concerned, the host has an IP of 10.0.2.2. We can use that with telnet, ftp, or whatever is required.

Port Redirection and qconn#

Yes, qconn can be used and it works great. We just have to redirect the right port when starting QEMU on the command line. For qconn, we need port 8000 on the target, so we need to invoke QEMU with a -redir argument:

```
qemu -hda <raw-image> -k en-us -redir <tcp|udp>:<port on host>:<target ip>:<port on target>
```

For qconn specifically, assuming that the target gets the usual first ip of 10.0.2.15, and an arbitrary host port of 8001, this looks like:

```
qemu -hda <raw-image> -k en-us -redir tcp:8001:10.0.2.15:8000
```

Note that we can use port 8000 for both ports, but if the host is a Neutrino system, this might not be a good idea.

Now, when we set up the system profiler, the QNX connector is localhost:8001 (not the local QNX connector), and the "QNX connector is on the target machine" should be checked. The -redir argument can be used multiple times, allowing connection from host to target for muliple programs.

The QEMU Monitor and I/O ports#

Pressing <ctl><alt><#> changes the view mode of QEMU. 1 is the main video out, 2 is the QEMU monitor, 3 is the first serial port, and 4 is a parallel port. See note on the other wiki about doing this with a Neutrino host. The monitor has commands to restart the system, pause, resume, and change removeable media. It also controlls snapshots if using qcow images. Just type help, or consult the official documentation for QEMU. The serial and parallel ports can have shells or other output redirected to them for whatever the user needs.

Debugging the Kernel using QEMU#

QEMU allows redirection of serial ports to any device on the host system, so debugging the kernel is a simple process. See Debugging the kernel with QEMU

Non-x86 targets#

The startup for non-x86 targets is a little different. They can't boot directly from disk images, and require a kernel image passed as an argument. This section will be updated as soon as I know how to make them work.

Questions and Omissions#

If you think something else should be coverered here, then post in the QEMU project forums.