Palacios Internal Developer Manual

January 12, 2010

This manual is written for internal Palacios developers. It contains information on how to obtain the Palacios code base, how to go about the development process, and how to commit those changes to the mainline source tree. This assumes that the reader has read *An Introduction to the Palacios Virtual Machine Monitor – Release 1.0* and also has a slight working knowledge of *git*. You will also want to read the document *Building a bootable guest image for Palacios and Kitten* in order to understand how to build an extremely lightweight guest environment, suitable for testing.

Please note that Palacios and Kitten are under rapid development, hence this manual may very well be out of date!

Contents

List of Figures

1 Overview

Both Palacios and Kitten follow a hybrid development process that uses both the centralized repository and distributed development models. A central repository exists that holds the master version of the code base. This central repository is cloned by multiple people and in multiple places to support various development efforts. A feature of `git` is that every developer actually has a fully copy of the entire repository, and so can function independently until such time as they need to re-sync with the master version.

There are typically multiple levels of access to the central repository, that are granted based on the type of developer being granted access. The three basic developer types and their access privileges are:

- Core developers: These are the lead developers and are in charge of managing the master repository. They have full read/write access permissions to the central repository.
- Internal developers: Formal members of the development team. These people are capable of pulling directly from the central repository, but lack the ability to write directly to it.
- External developers: People who are not actual members of the development team. These people can only access the public repository which is only updated to contain the release versions.

Students doing independent study or REUs related to Palacios are set up as internal developers. EECS 441 (Resource Virtualization) students are generally either set up as internal or external developers, depending on their projects.

Because internal and external developers cannot write directly to the master repository, they need to first submit their changes to a core developer before they can be added to the mainline. We will discuss that process in Section 9.

2 Checking out Palacios

The central Palacios repository is located on *newskysaw.cs.northwestern.edu* in */home/palacios/palacios*. All internal developers have read access to the directory. Each developer must create their own local version of the repository, this is done with *git clone*.

```
git clone /home/palacios/palacios
```

On the machine *newbehemoth.cs.northwestern.edu* you will want to use the following command instead. The *newskysaw* home directories are NFS-mounted on /home-remote.

```
git clone /home-remote/palacios/palacios
```

On any other machine, you can clone the repository via ssh, provided you have a newskysaw account:

```
git clone ssh://you@newskysaw.cs.northwestern.edu//home/palaicos/palacios
```

This creates a local copy of the repository at *./palacios/*.

Note that both *newskysaw* and *newbehemoth* have all the tools installed that are needed to build and test Palacios and Kitten. If you develop on another machine, you will need to set those tools up for yourself. This isn't hard and the tools are all free. See the technical report for what tools you will need.

When you first clone the repository, you will get the *master* branch, which is used to generate releases. All development work is done in the *devel* branch of the repository. The developer can access this branch via:

```
git checkout --track -b devel origin/devel
```

or

```
/opt/vmm-tools/bin/checkout_branch devel
```

Important: Note that Palacios is very actively developed so the contents of the *devel* branch are frequently changing. In order to keep up to date with the latest version, it is necessary to periodically pull the latest changes from the master repository by running `git pull`.

3 Checking out Kitten

Kitten is available from Sandia National Labs, and is the main host OS we are targeting with Palacios. Loosely speaking, core Palacios developers are internal Kitten developers, and internal Palacios developers are external Kitten developers. The public repository for Kitten is at *http://code.google.com/kitten*. To simplify things, we are maintaining a local mirror copy in */home/palacios/kitten* that tracks the public repository.

Kitten uses Mercurial for their source management, so you will have to make sure the local mercurial version is configured correctly. Specifically you should add the following Python path to your shell environment.

```
export PYTHONPATH=/usr/local/lib64/python2.4/site-packages/
```

You can then clone Kitten from the local mirror. On *newskysaw*, run:

```
hg clone /home/palacios/kitten
```

On *newbehemoth*, run

```
hg clone /home-remote/palacios/kitten
```

On other machines, run

```
hg clone ssh://you@newskysaw.cs.northwestern.edu//home/palacios/kitten
```

Both the Kitten and Palacios clone commands should be run from the same directory. This means that both repositories should be located at the same directory level. The Kitten build process depends on this.

Important: Like Palacios, Kitten is under active development, and its source tree is frequently changing. In order to keep up to date with the latest version, it is necessary to periodically pull the latest changes from the mirror repository by running `hg` pull. followed by `hg update`.

4 Compiling Palacios

The Palacios build process has been changed recently from a homegrown environment to the widely used KBuild environment. KBuild is also used for building Kitten. Because KBuild is the build environment used for Linux, much of what you learn about configuring and building Linux kernels is readily applicable to Palacios and Kitten.

The output of the Palacios build process is a static library that includes the Palacios VMM and relevant guest support code blocks. This static library is then linked into a host operating system. Palacios internally supports GeekOS and can generate a complete OS image via a unified build process. By complete OS image, we mean an ISO image containing GeekOS, Palacios, and a guest image (another ISO image) for testing.

These days, however, Palacios is typically embedded into Kitten, not GeekOS. To combine Palacios with Kitten, it is necessary to first configure and build Palacios, then to configure and build Kitten, linking in Palacios. Kitten can also be configured to link in a guest image for testing.

4.1 Configuration

To configure Palacios, enter the top level Palacios directory and execute:

```
make clean
make xconfig
```

At this point, you will see be presented with a KBuild configuration screen, similar to what you would see in configuring a Linux kernel. Palacios has far fewer options, however. If you don't have X or don't want the graphical configuration system, you can also use the `menuconfig` or `config` targets. The available options change over time, so we do not cover all of them here, but here are a few that are usually important, with their recommended values noted:

- Target Configuration:
  - Red Storm (Cray XT3/XT4) — turn on to target Cray XT4 supercomputers. (off)

  - AMD SVM Support — targets AMD processors with the SVM hardware virtualization features (on)
  - Intel VMX support — targets Intel processors with the VMX hardware virtualization features (on)
  - Compile for a multi-threaded OS (on)
  - Enable VMM telemetry support — this is lightweight logging and data collection (on)
  - Enable VMM instrumentation — this is heavyweight logging and data collection (off)
  - Enable passthrough video — this lets a guest write directly to the video card (on)
  - Enable experimental options — this makes it possible to select features that are under current development (on). You probably want to leave VNET turned off. VNET is an experimental VMM-embedded overlay network under development by Lei Xia and Yuan Tang.
  - Enable built-in versions of stdlib functions — this adds needed stdlib functions that the host OS may not supply. For use with Kitten turn on and enable strncasecmp() and atoi().
  - Enable built-in versions of stdio functions (off)

- Symbiotic Functions (these are experimental options for Jack Lange's thesis).
  - Enable Symbiotic Functionality — This adds symbiotic features to Palacios, specifically support for discovery and configuration by symbiotic guests, the SymSpy passive information interface for asynchronous symtiotic guest $\leftrightarrow$ symbiotic VMM information flow, and the SymCall functional interface for synchronous symbiotic VMM $\rightarrow$ symbiotic guest upcalls. (on)
  - Symbiotic Swap — Enables the SwapBypass symbiotic service for symbiotic Linux guests. (off)

- Debug Configuration
  - Compile with Debug info — adds debug symbols (-g) (off)
  - Enable Debugging — makes it possible to show PrintDebug output (on). You can selectively turn on debugging output for each major VMM component, including shadow paging, nested paging, control registers, interrupts, I/O, instruction emulation and XED, halt, and the device manager. Note that the more debugging output you turn on, the slower the VMM will go since it will have to wait for the prints to finish.

- BIOS Selection — Lets you select which code blobs will be used for bootstrapping the guest. There are currently three: a BIOS, a Video BIOS, and the VMXAssist V8086 service (the latter is used only on Intel VMX). Generally, you should not need to change these.

- Virtual Devices — virtual devices can be instantiated and added to a guest. The following is a list of the currently implemented virtual devices.
  - BOCHS Debug Console Device — used for debugging output from the guest BIOS (on)
  - OS Debug Console Device — used for debugging output from the guest kernel (on)
  - 8259A PIC - legacy Programmable Interrupt Controller chip — used for bootstrap of most guests (on)

- APIC - in-processor Advanced Programmable Interrupt Controller — used for interrupt delivery on almost all guests (on)
- IOAPIC - Off-chip APIC — used for interrupt deliver for almost all guests (on)
- i440fx Northbridge — emulation of a typical PC North Bridge chip, used on almost all guests (on)
- PIIX3 Southbridge — emulation of a typical PC South Bridge chip, used on almost all guests (on)
- PCI — emulation of a PCI bus - needed for attaching most devices (on)
  - Passthrough PCI — allows us to make a hardware PCI device visible and directly accessible by the guest (on)
- NVRAM - motherboard configuration memory — needed by BIOS bootstrap (on)
- Keyboard - Generic PS/2 keyboard, including mostly broken mouse implementation (on)
- IDE — Support for virtual IDE controllers that support disks and CD ROMs (on)
- NE2K - NE2000 and RTL8139 network devices (off)
- CGA - CGA video card (paritial implementation) (off)
  - Telnet Virtual Console (off) When CGA and Telnet Console are on, it is possible to telnet to the console of the guest. Eventually the rest of this will provide simple bitmapped video console for VNC access.
- RAMDISK storage backend — used to create RAM disk implementations of block devices (on)
- NETDISK storage backend — used to create network-attached disk implementations of block devices, e.g., network block devices (on)
- TMPDISK storage backend — used to create temporary storage implementations of block devices (on)
- Linux Virtio Balloon Device — used for memory ballooning by Linux virtio-compatible guests (on)
- Linux Virtio Block Device — used for fast block device support by Linux virtio-compatible guests (on)
- Linux Virtio Network Device — used for fast network device support by Linux virtio-compatible guests (on)
- Symbiotic Swap Disk (multiple versions) — used for the SwapBypass service (off)
- Disk Performance Model — used for the SwapBypass service (off)

4.2 Compilation

After configuring Palacios—remember to save your changes—you can compile it by executing

```
make
```

This will produce the file *libv3vee.a* in the current directory. This static library contains the Palacios VMM and is ready for embedding into an OS, such as Kitten. The library provides the ability to instantiate and run virtual machines. By default, on a 64 bit machine, the library is compiled for 64 bit machines (x86_64), while on a 32 bit machine, it is compiled for 32 bit machines. You can override this using the ARCH=i386 or ARCH=x86_64 arguments to the make, provided you have the relevant tools available. The 64 bit version is what you need for use with Kitten. A 64 bit Palacios can run both 64 and 32 bit guests. Both *newskysaw* and *newbehemoth* are 64 bit machines.

5 Compiling Kitten

Kitten requires a 64-bit version of Palacios, so make sure that Palacios has been correctly compiled before compiling Kitten. The current default for Palacios is 64 bit.

5.1 Configuration

Kitten borrows a lot of concepts from Linux, including the Linux build process. As such it must be configured before it is actually compiled. The Kitten configuration process is the same as Linux, and can be accessed via any of these make targets.

- `make xconfig`
- `make config`
- `make menuconfig`

Of course, there are a range of configuration options. In the following, we note only the most important:

- Target Configuration
  - System Architecture — you probably want to set this to PC-Compatible, unless you are working on Red Storm.
  - Processor Family — you want to set this to either AMD-Opteron/Athlon64 or Intel-64/Core2, depending on whether you have a 64 bit AMD or 64 bit Intel processor.
- Virtualization
  - Include Palacios VMM — this will link against the Palacios library (on)
  - Path to pre-built Palacios tree — directory where libv3vee.a can be found.
  - Path to guest image — location of the test guest OS image that will be embedded. We will say more about this later. Essentially, however, a guest image consists of a blob that begins with an XML description of the desired guest environment and the contents of the remainder of the blob. The remainder of the blob usually contains disk or cd images.
- Networking

  - Enable LWIP TCP/IP stack. This activates a simple TCP/IP stack that things like NETDISK can use. (on)

- Device Drivers
  - VGA Console — driver for basic video. If you turn on passthrough video in Palacios, you should turn this off.
  - Serial Console — driver for serial port console. (on)
  - VM Console — driver for Kitten console on top of Palacios. If Kitten is run *as a guest*, and it has VM Console on, then it can output cleanly via the Palacios OS Console device (off).
  - NE2K Device Driver — driver for NE2K and RTL8139 network cards (on)
  - VM Network Driver — driver for Kitten network output using Palacios. If Kitten is run *as a guest*, and it has VM Network Driver, then it can send and receive packets using the Palacios Linux virtio network device. (off)

- ISOIMAGE configuration: Kitten kernel arguments. Note that this is NOT for the guest image, but rather for the Kitten image. You can leave this alone. For Palacios operation, it's important that the option `console=serial` appears. If the NE2K/RTL8139 driver should be used `net=rtl8139` should appear.

- Kernel Hacking
  - Kernel Debugging — here you can turn on various Kitten Linux-like debugging features. Only a few are noted below:
    * Compile the kernel with debug info — if this is on, you will have debugging information compiled in (-g)
    * KGDB — if you have this enabled, you will be able to attach to the running kernel from the GDB debugger. This means you can also attach to Palacios, which is embedded. If you want to debug Palacios using KGDB, be sure to turn on debugging in Palacios as well.

- Include Linux compatability layer — if this is on, you can selectively add Linux system calls and other functionality to Kitten. Kitten is able to run Linux ELF executables as user processes with this layer.

The guest OS that is to be booted as a VM is included as a blob pointed to by "Path to guest image". The blob starts with an XML description of the guest, followed by other chunks of data used, for example, as the content of virtual hard drives or CD ROMs. Please see Section 6 for basic information on how to use the guest builder to assemble a guest OS blob.

By default, the init task that is executed after Kitten boots (located in user/hello_world) does a number of Kitten tests. One of these is a test of the VMM API, which is implemented using Palacios. When this test is done, a VM is created, configured according to the XML, and the guest OS blob is launched in it.

5.2 Compilation

After Kitten has been configured it can be compiled. This is done simply by executing

```
make isoimage
```

This command will compile Kitten (with Palacios embedded in it) and the init task (which will contain the guest OS blob), and then assemble an ISO image file which can used to boot a machine. The ISO image is located at *./arch/x86_64/boot/image.iso*.

This image file can be used for booting a QEMU emulation environment, for booting a remote machine using PXE, or can be burned to CD/DVD for booting a machine physically.

6 Basic Guest Configuration

To configure a guest, you write an XML configuration file, which contains references to other files that contain data needed to instantiate stateful devices such as virtual hard drives and CD ROMs. You supply this information to a guest builder utility that assembles a guest image suitable for reference in the Kitten configuration, as described above.

The guest builder utility is located in *palacios/utils/guest_creator*. You will need to run `make` in that directory to compile it, resulting in the executable named *build_vm*. Also located in that directory is an example configuration file, named *default.xml*. We typically use this file as a template. It is carefully commented. In summary, a configuration consists of

- Physical memory size of the gust
- Basic VMM settings, such as what form of virtual paging is to be used, the scheduler rate, whether services like telemetry are on, etc.
- A memory map that maps regions of the host physical address space to the guest physical address space. This can, for example, make a framebuffer visible in the guest.
- A list of the files that will be used in assembling the image. For example, the contents of a boot CD.
- A list of the devices that the guest will have, including configuration data for each device.

There are a few subtlies involved with devices. One is that some devices form attachement points for other devices. The PCI device is an example of this. Another is that each device needs to define how it is attached (e.g. direct (implicit), via a PCI bus, etc.) Finally, there may be multiple instances of devices. For example, a PCI passthrough device is instantiated for every underlying PCI device we want to make visible in the guest.

The XML configuration format is carefully designed to be extensible. For example, new devices could use additional or new configuration options. The configuration parser in Palacios essentially ignores XML blocks it doesn't understand.

To build a guest, one runs

```
palacios/utils/guest_creator/build_vm myconfig.xml -o myimage.dat
```

Here, *myimage.dat* is the guest image that can be given to Kitten.

A common kind of guest used for testing is one that boots some form of bootable Linux distribution, or other a live OS distribution. These distributions are CD ROM images (ISOs). A range of them are available on *newskysaw* under */opt/vmm-tools/isos*. We often use Puppy Linux (*puppy.iso*) or Finnix (*finnix.iso*), for example, but isos are also available for Windows of different flavors, DOS, GeekOS, and others. If you just want to use some guest ISO image like this, you can generally just copy the default XML file, and modify the `filename=` attribute here:

```
<files>
   <!-- The file 'id' is used as a reference for
        other configuration components -->
   <file id="boot-cd" filename="/home/jarusl/image.iso" />
   <!--<file id="harddisk" filename="firefox.img" />-->
</files>
```

For careful, repeatable experimentation, it is often convenient to build your own simplified Linux guest image. It will boot *much* faster than a full blown distribution and you can readily set up an environment in which you can exert very tight control, being able to modify the Linux kernel, the included files (e.g., benchmarks), and other components very rapidly. To learn more about how to do this, please consult the separate document named *Building a bootable guest image for Palacios and Kitten*.

7 Running Palacios/Kitten

Kitten and Palacios are capable of running under QEMU, which makes debugging much simpler. QEMU is a user-level Linux or Windows program that emulates a PC machine.

The basic form of the command to start the QEMU emulator is:

```
/usr/local/qemu/bin/qemu-system-x86_64 -smp 1 -m 1024 \
        -serial file:./serial.out \
        -cdrom ./arch/x86_64/boot/image.iso  \
        < /dev/null
```

The command starts up a single processor emulated machine, with 1GB of RAM and a CD-ROM drive loaded with the Kitten ISO image. All output to the serial port is written directly to a file called *serial.out*. This command can be copied into a shell script for easy access.

We can also run Palacios/Kitten on physical hardware. The slow way is to burn the Kitten ISO image onto a CD ROM and then boot the test machine with it. The much faster way is to set the test machine up to use the PXE network boot system (most modern BIOSes support this), and boot your Kitten image over the network. The debugging output will then appear on the actual serial port of the physical machine. For the Northwestern environment, please talk to Jack Lange or Peter Dinda if you need to be able to do this. Northwestern has a range of AMD and Intel boxes for testing, as do UNM and Sandia. A different form of network boot is used for Red Storm.

8 Development Guidelines

There are standard requirements we have for code entering the mainline.

First and foremost, Palacios is designed to be OS independent and support 32-bit and 64-bit architectures. This means that developers should not include any external OS specific dependencies in any Palacios component. Also all changes need to be tested on both 32-bit and 64-bit architectures to make sure that they compile as well as run correctly.

Coding Style "The use of equal negative space, as a balance to positive space, in a composition is considered by many as good design. This basic and often overlooked principle of design gives the eye a 'place to rest,' increasing the appeal of a composition through subtle means."

Translation: Use the space bar, newlines, and parentheses.

Curly-brackets are not optional, even for single line conditionals.

Tabs should be 4 characters in width.

Special: If you are using XEmacs add the following to your `init.el` file:

```
(setq c-basic-offset 4)
(c-set-offset 'case-label 4)
```

Bad

```
if(a&&b==5||c!=0) return;
```

Good

```
if (((a) && (b == 5)) ||
    (c != 0)) {
        return;
}
```

Fail Stop Because booting a basic Linux kernel results in over 1 million VM exits catching silent errors is next to impossible. For this reason ANY time your code has an error it should return -1, and expect the execution to halt.

This includes unimplemented features and unhandled cases. These cases should ALWAYS return -1.

Function names Externally visible function names should be used rarely and have unique names. Currently we have several techniques for achieving this:

1. `#ifdefs` in the header file
   When the V3 Hypervisor is compiled it defines the symbol `__V3VEE__` Any function that is not needed outside the Hypervisor context should be inside an `#ifdef __V3VEE__` block, this will make it invisible to the host environment.

2. Static Functions
   Any utility functions that are only needed in the .c file where they are defined should be declared as static and not included in the header file. You should make an effort to use static functions whenever possible.

3. `v3_` prefix
   Major interface functions should be named with the prefix `v3_` This allows easy understanding of how to interact with the subsystems. In the case that they need to be externally visible to the host OS, make them unlikely to collide with other functions.

Debugging Output Debugging output is sent through the host OS via functions in the `os_hooks` structure. These functions have various wrappers of the form `Print*`, with `printf`-style semantics.

Two functions of note are `PrintDebug` and `PrintError`.

- PrintDebug:
  Should be used for debugging output that will often be turned off selectively by the VMM configuration.

- PrintError
  Should be used when an error occurs, this will never be optimized out and will always print.

9 Code Submission

To commit changes to the central repository they need to be exported as a patch set that can be applied directly to a mainline. Both Git and Mercurial contain functionality to allow developers to maintain changes as a patch set. There are also a few options that make dealing with patches easier.

9.1 Palacios

Git includes support for directly exporting local repository commits as a patch set. The basic operation is for a developer to commit a change to a local repository, and then export that change as a patch that can be applied to another git repository. While this is functionally possible, there are a number of issues. The main problem is that it is difficult to fully encapsulate a new feature in a single commit, and dealing with multiple patches that often overwrite each other is not a viable option either. Furthermore, once a patch is applied to the mainline, it will generate a conflicting commit that will become present when the developer next pulls from the central repository. This can result in both repositories getting out of sync. It is possible to deal with this by manually re-basing the local repository, but it is difficult and error-prone.

This approach also does not map well when patches are being revised. A normal patch will go through multiple revisions as it is reviewed and modified by others. This often leads to synchronization issues as well as errors with patch revisions. Ultimately it is the responsibility of the developer to generate a patch that will apply cleanly to the mainline.

For this reason most internal developers should seriously consider *stacked git*. Stacked git is designed to make patch development easier and less of a headache. The basic mode of operation is for a developer to initialize a patch for a new feature and then continuously apply changes to the patch. Stacked Git allows a developer to layer a series of patches on top of a local git repository, without causing the repository to unsync due to local commits. Basically, the developer never commits changes to the repository itself but instead commits the changes to a specific patch. The local patches are managed using stack operations (push/pop) which allows a developer to apply and unapply patches as needed. Stacked git also manages new changes to the underlying git repository as a result of a pull operation and prevents collisions as changes are propagated upstream. For instance if you have a local patch that is applied to the mainline as a commit, when the commit is pulled down the patch becomes empty because it is effectively identical to the mainline. It also makes incorporating external revisions to a patch easier. Stacked git is installed on *newskysaw* in `/opt/vmm-tools/bin/`

Brief command overview:

- `stg init` – Initialize stacked git in a given branch
- `stg new` – create a new patch set, an editor will open asking for a commit message that will be used when the patch is ultimately committed.
- `stg pop` – pops a patch off of the source tree.
- `stg push` – pushes a patch back on to a source tree.
- `stg export` – exports a patch to a directory as a file that can then be emailed.
- `stg refresh` – commits local changes to the patch set at the top of the applied stack.
- `stg fold` – Apply a patch file to the current pat1ch. (This is how you can manage revisions that are made by other developers).

You should definitely look at the online documentation to better understand how stacked git works. It is not required of course, but if you want your changes to be applied its up to you to generate a patch that is acceptable to a core developer. Ultimately using Stacked git should be easier than going it alone.

All patches should be emailed to Jack for inclusion in the mainline. An overview of the organization is given in Figure 1. You should assume that the first revision of a patch will not be accepted, and that you will have to make changes. Furthermore, the final form of the patch most likely will not be exactly what you submitted.

9.2 Kitten

Writing code for Kitten follows essentially the same process as Palacios. The difference is that the patches need to be emailed to the Kitten developers. To send in a patch, you can just email it to the V3Vee develop-

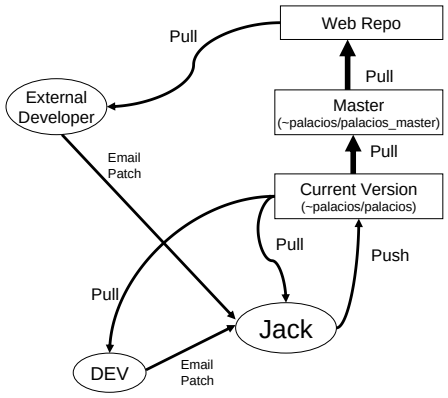

Figure 1: Development organization

ment list.

Also, instead of Stacked git you should use Mercurial patch queues. This feature is enabled in your .hgrc file.

```
[extensions]
hgext.mq=
```

Mercurial queues use the same stack operations as stacked git, however it does not automatically handle the synchronization with pull operations. Before you update from the central version of Kitten you need to pop all of the patches, and then push them once the update is complete.

Basically:

```
hg qpop -a
hg pull
hg update
hg qpush -a
```

10 Networking

Both the Kitten and GeekOS substrates on which Palacios can run currently include drivers for two simple network cards, the NE2000, and the RTL8139. The Kitten substrate is acquiring an ever increasing set of drivers for specialized network systems. A lightweight networking stack is included so that TCP/IP networking is possible from within the host OS kernel and in Palacios.

When debugging Palacios on QEMU, it is very convenient to add an RTL8139 card to your QEMU configuration, and then drive it from within Palacios. QEMU can be configured to provide local connectivity to the QEMU emulated machine, including bridging the emulated machine with a physical network. Local connectivity can be done with redirection, or with a TAP interface. For global connectivity, a TAP interface must be used; it is bridged to a physical interface.

11 Configuring the development host's QEMU network

To get local connectivity with redirection, no networking changes on the host are needed. However, people usually want to use TAP-based networking, which does require changes. For one thing, TAP interfaces can be inspected with tools like wireshark, which makes for much easier debugging of network code.

In order to get QEMU networking to function, it is necessary to create TAP interfaces, and, optionally, to bridge them to real networks. A development machine typically will have several TAP interfaces, and more can be created. Generally, each developer should have a TAP interface of his or her own. In the following, we will use our development machine, newskysaw, as an example.

To set up a TAP interface on newskysaw, the following command is used:

```
/root/util/tap_create tapX
```

When QEMU runs with a tap interface, it will use /etc/qemu-ifup to bring up the interface. On newskysaw, /etc/qemu-ifup looks like this:

```
#!/bin/bash
echo "Executing /etc/qemu-ifup - no external bridging"
echo "Bringing up $1 for bridged mode..."
NET=`echo $1 | cut -dp -f2`
sudo /sbin/ifconfig $1 172.2${NET}.0.1 up
sleep 2
```

The interface tap$N$ is brought up with the IP address 172.2N.0.1. ifconfig will also create a routing rule that sends 172.2N.0.1/16 traffic to tap$N$. The upshot is that if the code running in QEMU uses an IP address in this network (for example: 172.2N.0.2), you will be able to talk to it from newskysaw. For example, from newskysaw, if you ping 172.21.0.2, the packet (and ARP) will go out via tap1. The source address will appear to be 172.21.0.1. The QEMU machine will see these packets on its interface, and the software controlling its interface can respond to 172.21.0.1.

This form of networking is local to the machine. You can also bridge a TAP interface with a physical interface. The result of this is that a packet sent on it will be sent on the physical interface. To do this requires more effort (and is not set up by default on newskysaw). As an example, consider that on newskysaw, the physical interface eth1 is connected to a private network switch to which the lab test computers (v-test-amd, v-test-amd2, etc.) are connected. To bridge, for example, tap10, to this interface, you would do the following (with root's help):

1. You need to bring up eth1 (ifconfig eth1 up *address* netmask *mask*). It is important that the address and mask you choose are appropriate for the network eth1 is connected to.
2. You would bring up tap10 without an address: /sbin/ifconfig tap10 up
3. You would bridge tap10 and eth1: /usr/sbin/brctl addif br0 tap10; /usr/sbin/brctl addif eth1. This assumes that br0 was previously created.

Bridging tap$N$ with eth1 will only work (where "work" means sending a packet on the network and making the packet visible on localhost) if the IP address in the code running in QEMU is set correctly. This means that it needs to be set to correspond to the network of eth1). For the newskysaw configuration, this is a 10-net address.

11.1 Configuring Kitten

Kitten needs to be explicitly configured to use networking. Currently only a subset of the networking configurations are supported. To enable an Ethernet network you should enable the following options:

- Enable TCP Support
- Enable UDP Support
- Enable socket API
- Enable ARP support

The other options are not supported, and enabling them will probably break the kernel compilation.

To allow Kitten to communicate with the QEMU network card you also need to enable the appropriate device driver:
`NE2K Device Driver (rtl8139)`

The driver then needs to be listed as a Kernel Command Line argument in the *ISOIMAGE configuration*. To do this add `net=rtl819` to the end of the argument string.

Kitten currently does not support the dynamic assignment or IP addresses at runtime. Because of this it is necessary to hardcode the IP address into the device driver. For the rtl8139 network driver look in the file *drivers/net/ne2k/rtl8139.c* for the function `rtl8139_init`.

There should be a block of code that looks like the following:

```
struct ip_addr ipaddr = { htonl(0 | 10 << 24 | 0 << 16 | 2 << 8 | 16 << 0) };
struct ip_addr netmask = { htonl(0xffffff00) };
struct ip_addr gw = { htonl(0 | 10 << 24 | 0 << 16 | 2 << 8 | 2 << 0) };
```

This sets the IP address as 10.0.2.16, netmask 255.255.255.0 and gateway address 10.0.2.2. Change these assignments to match your configuration.

Kitten as the Guest OS When running Kitten as a VM, the above applies except that you will want to enable the *VMNET* device driver instead of the *rtl8139*.

11.2 Running with networking

TAP Interface Running with a TAP interface provides either local or global connectivity (depending on how the TAP interface is configured and/or bridged). From the perspective of the QEMU command line, both look the same, however. You simply add something like this to the command line:

```
-net tap,ifname=tap2 -net nic,model=rtl8139
```

The first `-net` option indicates that you want to use a tap interface, specifically `tap2`. The second `-net` option specifies that this interface will appear to code in the QEMU machine to be a network interface card of the specific model RTL8139. Note that this is a model for which we have a driver. If tap2 were bridged, we'd get global connectivity. If not, we would just get local connectivity.

Redirection It is also possible to achieve limited local connectivity even if you have no TAP support on your development machine. In redirection, QEMU essentially acts as a proxy, translating TCP or other connections and low-level packet operations on the network interface in the QEMU machine. For example, the following options will redirect the host's 9555 port to the QEMU machine's 80 port:

```
-net user -net nic,model=rtl8139  -redir tcp:9555:10.10.10.33:80
```

The first `-net` option indicates that we are using user-level networking (proxying). The second `-net` option indicates that this user-level network will appear in the QEMU machine as an RTL8139 network card. The `-redir` option indicates that connections on localhost:9555 will be translated into equivalent packet exchanges on the RTL8139 card in the QEMU machine. However, we have to tell QEMU which IP address and port to use on the QEMU machine's side. This is what the 10.10.10.33 address, and port 80 are. In the example, if you access port 9555 on localhost, say with:

```
telnet localhost 9555
```

The packets that appear in the QEMU machine will be bound for 10.10.10.33, port 80. Within the QEMU machine, your RTL8139 interface had better then be up on that address.

QEMU has many options to build up virtual or real networking. See http://www.h7.dion.ne.jp/∼qemu-win/HowToNetwork-en.html for more information.

For more questions, talk to Jack, Lei, or Peter.