Palacios Internal Developer Manual

March 4, 2009

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*.

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 resync 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.

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 8.

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
```

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

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 targetting with Palacios. Loosely speaking core Palacios developers are internal Kitten developers, and internal Palacios developers are external Kitten developers. Because we have limited access to the Kitten repository, we are maintaining a local mirror copy in */home/palacios/kitten*.

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:

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

Both the Kitten and Palacios clone commands should be run from the same direcotyr. 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 very actively developed so 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

Palacios is capable of targeting 32 and 64 bit operating systems, and includes a build process that supports both these architectures. Furthermore, Palacios has multiple build locations, with multiple makefiles: a top level build directory and a Palacios specific build directory. The Palacios build process first generates a static library that includes the Palacios VMM. 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. To combine Palacios with Kitten, it is necessary to first compile Palacios and then to compile Kitten externally link it with Palacios. The output of the compilation process is a bit more complex and generates multiple binaries, and the specifics can be found in the Makefiles.

The top level build directory provides a number of high level make targets, and is located in *palacios/build/*. It supports building 32 and 64 bit versions of the Palacios library independently as well as building an integrated version of GeekOS. The basic targets are:

- `make palacios-full32` – Generates a 32 bit version of the Palacios static library
- `make palacios-full64` – Generates a 64 bit version of the Palacios static library
- `make geekos` – Compiles the GeekOS kernel, and link it with the Palacios static library
- `make geekos-iso` – Generate an ISO boot disk image from the GeekOS kernel that has been compiled

The second build directory is located at *palacios/palacios/build* and handles only the Palacios compilation process. It supports a differnt set of targets and arguments:

- `make ARCH=32` – iteratively compiles a 32 bit version of Palacios
- `make ARCH=64` – iteratively compiles a 64 bit version of Palacios
- `make ARCH=32 world` – fully recompiles a 32 bit version of Palacios
- `make ARCH=64 world` – fully recompiles a 64 bit version of Palacios

Both build levels support compilation directives that control the debugging messages that are generated by Palacios. These are specified by appending a `DEBUG_<COMPONENT>=1` to the end of the `make` command. The components that are currently supported are:

- `DEBUG_ALL=1` – enables debugging for all the VMM components (*Warning:* this generates a *lot* of debug information.
- `DEBUG_SHADOW_PAGING=1`
- `DEBUG_CTRL_REGS=1`
- `DEBUG_INTERRUPTS=1`
- `DEBUG_IO=1`
- `DEBUG_KEYBOARD=1`
- `DEBUG_PIC=1`
- `DEBUG_PIT=1`
- `DEBUG_NVRAM=1`
- `DEBUG_GENERIC=1`
- `DEBUG_EMULATOR=1`
- `DEBUG_RAMDISK=1`
- `DEBUG_XED=1`
- `DEBUG_HALT=1`
- `DEBUG_DEV_MGR=1`
- `DEBUG_APIC=1`

5 Compiling Kitten

Kitten requires a 64 bit version of Palacios, so make sure that Palacios has been correctly compiled before compiling Kitten.

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`

There are some specific configuration options that should be disabled to work with Palacios. Because Palacios is configured by default to provide a guest with direct access to the VGA console, the *VGA console* device driver should be disbabled in the Kitten configuration. Similarly the *VM console* driver should be disabled as well.

Furthermore, because the VGA console is not being used the *Kernel Command Line Arguments* must be modified to remove the *VGA* device from the console list.

The guest OS that is booted as a VM is included as an ISO image in raw binary format inside Kitten's *init_task*. To change the guest ISO, you must change the makefile for the init_task. This is located in *user/hello_world/Makefile* and the syntax is well commented. On *newskysaw* a collection of guest ISO images are located in */opt/vmm-tools/isos/*.

5.2 Compilation

After Kitten has been configured the compilation can be done. The general process is to compile a reference build of Kitten, followed by compiling Palacios support as a kernel module, and then doing a new full recompilation of Kitten.

The specific compilation steps are run from the top level Kitten directory:

```
make
cd palacios
make -C .. M=`pwd`
cp built-in.o ../modules/palacios-mod.o
cd ..
make
make isoimage
```

→ **Note:** ***This should probably explain how to change the iso (helloworld,etc)***

This generates an ISO boot image containing Kitten, Palacios, and the guest that will be run as a VM. The ISO image is located at *./arch/x86_64/boot/image.iso*.

6 Running Palacios/Kitten

Kitten and Palacios are capable of running under Qemu, which makes debugging much simpler.

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 1gig of RAM and a cdrom drive loaded with the Kitten ISO image. Furthermore 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.

7 Development Guidelines

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

First and foremost, Palacios is designed to be OS indenpendent and support 32 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 and 64 bit architectures to make sure that they compile as well as run corrrectly.

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 spacebar, 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 xallows easy understanding of how to interact with the subsystems. And 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.

8 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.

8.1 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 rebasing 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 intialize 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 patch. (This is how you can manage revisions that are made by other developers).

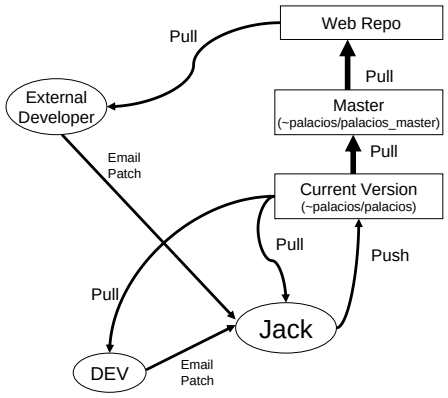

Figure 1: Development organization

You should definately 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.

8.2 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 development 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
```

9 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.

10 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 developmet 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 comand 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 controling 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 and that it.
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.

10.1 Configuring Kitten

To enable networking in Qemu, networking needs to be enabled in the configuration.

Make sure turn on the network device driver, networking, and input kernel command 'console=serial net=rtl8139'

How to set ip address in kitten:

Kitten ip address setting is in file drivers/net/ne2k/rtl8139.c, in the code below which is located in function rtl8139_init.

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 it as you need.

10.2 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 secod `-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 a 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.