One of the exciting new features in FreeBSD 10.0 is the bhyve hypervisor. Hypervisors and virtual machines are used in a wide variety of applications. This article focuses on using bhyve as a tool for aiding development of FreeBSD itself. Not all of the details covered are specific to FreeBSD development, however, and many may prove useful for other applications.

The hypervisor itself contains both user and kernel components. The kernel driver is contained in the vmm.ko module and can be loaded either at boot from the boot loader or at runtime. It must be loaded before any guests can be created. When a guest is created, the kernel driver creates a device file in /dev/vmm which is used by the user programs to interact with the guest.

The primary user component is the http://www.freebsd.org/cgi/man.cgi?query=bhyve&sektion=8&manpath=FreeBSD+10.0-RELEASE bhyve(8) program. It constructs the emulated device tree in the guest and provides the imple-
Bridged Configuration

One simple network setup bridges the guest network interfaces directly onto a network to which the host is connected. On the host, a single "http://www.freebsd.org/cgi/man.cgi?query=bhyveload&sektion=8&man-path=FreeBSD+10.0-RELEASE" bhyveload(8) program can be used to load the kernel and prepare the guest for execution. Support for some other operating systems is available via the "https://github.com/grehan-freebsd/grub2-bhyve" grub2-bhyve program which is available via the "http://www.freeports.org/sysutils/grub2-bhyve" sysutils/grub2-bhyve port or as a prebuilt package.

The bhyveload(8) program in FreeBSD 10.0 only supports 64-bit guests. Support for 32-bit guests will be included in FreeBSD 10.1.

Network Setup

The network connections between the guests and the host can be configured in several ways. Two different setups are described below, but they are not the only possible configurations.

The only guest network driver currently supported by bhyve is the "http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html" VirtIO network interface driver. Each network interface exposed to the guest is associated with a http://www.freebsd.org/cgi/man.cgi?query=tap%284%29 tap(4) interface in the host. The tap(4) driver allows a user program to inject packets into the network stack and accept packets from the network stack. By design, each tap(4) interface will only pass traffic if it is opened by a user process by setting the net.link.tap.up_on_open sysctl to 1.

Example 1: Manually Connecting a Guest to the Host’s LAN

The guest can then configure the VirtIO network interface bound to tap0 for the LAN on the host’s re0 interface using DHCP or a static address.

The /etc/rc.d/bridge script allows bridges to be configured during boot automatically by variables in /etc/rc.conf. The autobridge_interfaces variable lists the bridge interfaces to configure. For each bridge interface, an autobridge_<name> variable lists other network interfaces that should be added as bridge members. The list can include shell globs to match multiple interfaces. Note that /etc/rc.d/bridge will not create the named bridge interfaces. They should be created by listing them in the cloned_interfaces variable along with the desired tap(4) interfaces. Example 2 lists the /etc/rc.conf settings to create three tap(4) interfaces bridged to a local LAN on the host’s re0 interface.

Example 2: Bridged Configuration

Private Network with NAT

A more complex network setup creates a private network on the host for the guests and uses network address translation (NAT) to provide limited access from guests to other networks.
This may be a more appropriate setup when the host is mobile or connects to untrusted networks.

This setup also uses an if_bridge(4) interface, but only the tap(4) interfaces used by guests are added as members to the bridge. The bridge members are assigned addresses from a private subnet. The bridge interface is assigned an address from the private subnet as well to connect the bridge to the host's network stack. This allows the guests and host to communicate over the private subnet used by the bridge.

The host acts as a router for the guests to permit guest access to remote systems. IP forwarding is enabled on the host and guest connections are translated via "http://www.freebsd.org/cgi/man.cgi?query=natd%288%29 natd(8) The guests use the host's address in the private subnet as their default route.

Example 3 lists the /etc/rc.conf settings to create three tap(4) interfaces and a bridge interface using the 192.168.16.0/24 subnet. It also translates network connections over the host's wlan0 interface using natd(8).

Example 3: Private Network Configuration

```
autobridgeInterfaces="bridge0"
autobridgeBridge0="tap*"
clonedInterfaces="bridge0 tap0 tap1 tap2"
ifconfig_bridge0="inet 192.168.16.1/24"
gateway_enable="YES"
natd_enable="YES"
natd_interface="wlan0"
firewall_enable="YES"
firewall_type="open"
```

Example 4: Enabling dnsmasq's DNS and DHCP Servers

```
domain-needed
bogus-priv
interface=bridge0
dhcp-range=192.168.16.10,192.168.16.200,12h
```

Example 5 gives the changes to both files to use dnsmasq's DNS server and assign the subnet.

In addition to providing local DNS names for DHCP clients, dnsmasq also provides DNS names for any entries in /etc/hosts on the host. An entry to /etc/hosts that maps the IP address assigned to bridge0 to a hostname (e.g., "host") will allow guests to use that hostname to contact the host.

The last thing remaining is to configure the host machine to use dnsmasq’s DNS server. Allowing the host to use dnsmasq’s DNS server allows the host to resolve the name of each guest. The dnsmasq daemon can use "http://www.freebsd.org/cgi/man.cgi?query=resolverconf%288%29" resolverconf(8) to handle updates to the host’s DNS configuration provided by DHCP or VPN clients. This is implemented by resolverconf(8) updating two configuration files that are read by dnsmasq each time the host’s DNS configuration changes. Finally, the host should always use dnsmasq’s DNS server and rely on it to forward requests to other upstream DNS servers. Enabling all this requires changes to both dnsmasq’s configuration file and /etc/resolverconf.conf. More details about configuring resolverconf(8) can be found in "http://www.freebsd.org/cgi/man.cgi?query=resolverconf.conf%288%29" resolverconf.conf(5).

Example 5 gives the changes to both files to use dnsmasq as the host’s name resolver.
Running Guests via vmrun.sh

Executing a guest requires several steps. First, any state from a previous guest using the same name must be cleared before a new guest can begin. This is done by passing the --destroy flag to bhyvectl(8). Second, the guest must be created and the guest’s kernel must be loaded into its address space by bhyveload(8) or grub2-bhyve. Finally, the bhyve(8) program is used to create virtual devices and provide runtime support for guest execution. Doing this all by hand for each guest invocation can be a bit tedious, so FreeBSD ships with a wrapper script for FreeBSD guests: /usr/share/examples/bhyve/vmrun.sh.

The vmrun.sh script manages a simple FreeBSD guest. It performs the three steps above in a loop so that the guest restarts after a reboot similar to real hardware. It provides a fixed set of virtual devices to the guest including a network interface backed by a tap(4) interface, a local disk backed by a disk image, and an optional second disk backed by an install image. To make guest installations easier, vmrun.sh checks the provided disk image for a valid boot sector. If none is found, it instructs bhyveload(8) to boot from the install image, otherwise it boots from the disk image. In FreeBSD 10.1 and later, vmrun.sh will terminate its loop when the guest requests soft off via ACPI.

The simplest way to bootstrap a new FreeBSD guest is to use a FreeBSD guests. The simplest way to run a guest port is used as the system console device for standard file descriptors as the backend of the nously and uses its bhyve(8) synchro-

from the disk image.

to boot the guest ment can be dropped installed, the.

image passed to vmrun.sh. FreeBSD 8.4 also the normal install ISO as the optional install process can be used by using the standard install ISO as the optional install image passed to vmrun.sh. FreeBSD 8.4 also works as a bhyve guest. However, its installer does not fully support VirtIO block devices, so the initial install must be performed manually using steps similar to those from the "https://wiki.freebsd.org/RootOnZFS" RootOnZFS guide. Example 6 creates a 64-bit guest named “vm0” and boots the install CD for FreeBSD 10.0-RELEASE. Once the guest has been installed, the --I argument can be dropped to boot the guest from the disk image.

The vmrun.sh script runs bhyve(8) synchronously and uses its standard file descriptors as the backend of the first serial port assigned to the guest. This serial port is used as the system console device for FreeBSD guests. The simplest way to run a guest in the background using vmrun.sh is to use a

```
# mkdir vm0
# truncate -s 8g vm0/disk.img
# sh /usr/share/examples/bhyve/vmrun.sh -t tap0 -d vm0/disk.img -I FreeBSD-10.0-RELEASE-amd64-disc1.iso vm0
```

Example 6: Creating a FreeBSD/amd64 10.0 Guest

and 10.1 expect to use a video console and keyboard as the system console. As such, they do not enable a login prompt on the serial console.
A login prompt should be enabled on the serial console by editing `/etc/ttys` and marking the `ttyu0` terminal "on." Note that this can be done from the host after the install has completed by mounting the disk image on the host using "http://www.freebsd.org/cgi/man.cgi?query=mdconfig%288%29 mdconfig(8)."

Example 7 lists the `/etc/rc.conf` file for a simple FreeBSD guest.

Example 8: Booting a Kernel from the Host.

```bash
> cd ~/work/freebsd/head/sys/amd64/compile/GUEST
> make install DESTDIR=~/bhyve/vm0/host KMODOWN=john
...
> cd ~/bhyve
> sudo sh vmrun.sh -t tap0 -d vm0/disk.img -H vm0/host vm0
...
OK unload
OK load host0:/boot/kernel/kernel
host0:/boot/kernel/kernel text=0x523888 data=0x79df8+0x10e2e8 sym=[0x8+0x9fb58+0x8+0xbaf41]
OK boot
...
```

"Once kgdb(1) attaches to the remote target, it can be used to debug the guest kernel."

Using a bhyve Guest as a Target

One way bhyve can be used while developing FreeBSD is to allow a host to debug a guest as if the guest were a remote target. Specifically, a test kernel can be built on the host, booted inside of the guest, and debugged from the host using http://www.freebsd.org/cgi/man.cgi?query=kgdb%281%29 kgdb(1).

Once a guest is created and configured and a test kernel has been built on the host, the next step is to boot the guest with the test kernel. The traditional method is to install the kernel into the guest's filesystem either by exporting the build directory to the guest via NFS, copying the kernel into the guest over the network, or mounting the guest's filesystem on the host directly via mdconfig(8). An alternate method is available via bhyveload(8) which is similar to booting a test machine over the network.
Using bhyveload(8)’s Host Filesystem

The bhyveload(8) program allows a directory on the host’s filesystem to be exported to the loader environment. This can be used to load a kernel and modules from a host filesystem rather than the guest’s disk image. The directory on the host’s filesystem is passed to bhyveload(8) via the -h flag. The bhyveload(8) program exports a host0: device to the loader environment. The path passed to the host0: device in the loader environment is appended to the configured directory to generate a host pathname. Note that the directory passed to bhyveload(8) must be an absolute pathname. The vmrun.sh script in FreeBSD 10.1 and later allow the directory to be set via the -H argument. The script will convert a relative pathname to an absolute pathname before passing it to bhyveload(8).

Booting a test kernel from the host inside of the guest involves the above three steps:

Example 8 installs a kernel with the kernel config GUEST into a host directory for the guest “vm0”. It uses vmrun.sh’s -H argument to specify the host directory passed to bhyveload(8). It also shows the commands used at the loader prompt to boot the test kernel.

The guest can be configured to load a kernel from the host0: filesystem on the next boot using “http://www.freebsd.org/cgi/man.cgi?query=nextboot%288%29” nextboot(8). To boot the host0:/boot/kernel/kernel, run nextboot -e bootfile=host0:/boot/kernel/kernel before rebooting. Note that this will not adjust

3 STEPS for booting a test kernel

• Install the kernel into the directory on the host by setting the DESTDIR variable to the directory when invoking make install or make installkernel. A non-root user with write access to the directory can perform this step directly by setting the KMODOWN make variable to the invoking user.
• Pass the directory’s path to bhyveload(8) either via the -h flag to bhyveload(8) or the -H flag to vmrun.sh.
• Explicitly load the new kernel at the bhyveload(8) prompt via the loader path host0:/boot/kernel/kernel.

ISILON The industry leader in Scale-Out Network Attached Storage (NAS)

Isilon is deeply invested in advancing FreeBSD performance and scalability. We are looking to hire and develop FreeBSD committers for kernel product development and to improve the Open Source Community.

We’re Hiring!

With offices around the world, we likely have a job for you! Please visit our website at http://www.emc.com/careers or send direct inquiries to annie.romas@emc.com.
Example 9: Using kgdb(1) with bvmdebug: In the Guest

```bash
> sudo sh vmrun.sh -t tap0 -d vm0/disk.img -H vm0/host -g 1234 vm0
...
OK load host0:/boot/kernel/kernel
host0:/boot/kernel/kernel text=0x523888 data=0x79df8+0x10e2e8 syms=[0x8+0x9fb58+0x8+0xbaf41]
OK boot
Booting...
GDB: debug ports: bvm
GDB: current port: bvm
...
```

```bash
root@vm0:~ # sysctl debug.kdb.enter=1
default.kdb.enter: 0KDB: enter: sysctl debug.kdb.enter
[k thread pid 693 tid 100058 ]
Stopped at kdb_sysctl_enter+0x87: movq $0,kdb_why
db> gdb
(ctr-c will return control to ddb)
Switching to gdb back-end
Waiting for connection from gdb
-> 0
root@vm0:~ #
```

Example 10: Using kgdb(1) with bvmdebug: On the Host

```bash
> cd ~/work/freebsd/head/sys/amd64/compile/GUEST
> kgdb kernel.debug
...
(kgdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: Invalid remote reply:
kdb_sysctl_enter (oidp=<value optimized out>, arg1=<value optimized out>,
arg2=1021, req=<value optimized out>) at ../../../kern/subr_kdb.c:446
446                     kdb_why = KDB_WHY_UNSET;
Current language: auto; currently minimal
(kgdb) c
Continuing.
```

### Using bhyve

The bhyve(8) hypervisor provides an optional debug port that can be used by the host to debug the guest's kernel using kgdb(1). To use this feature, the guest kernel must include the bvmdebug device driver, the KDB kernel debugger, and the GDB debugger backend. The debug port must also be enabled by passing the `-g` flag to bhyve(8). The flag requires an argument to specify the local TCP port on which bhyve(8) will listen for a connection from kgdb(1). The vmrun.sh script also accepts a `-g` flag which is passed through to bhyve(8).

When the guest boots, its kernel will detect the debug port as an available GDB backend automatically. To connect kgdb(1) on the host to the guest, first enter the kernel debugger by setting the `debug.kdb.enter` system control node to a non-zero value. At the debugger prompt, invoke the `gdb` command. On the host, run kgdb(1) using the guest's kernel as the kernel image. The `target remote` command can be used to connect to the TCP port passed to bhyve(8). Once kgdb(1) attaches to the remote target, it can be used to debug...
the guest kernel. Examples 9 and 10 demonstrate these steps using a guest kernel built on the host.

Using kgdb(1) with a Virtual Serial Port

A serial port can also be used to allow the host to debug the guest’s kernel. This can be done by loading the "http://www.freebsd.org/cgi/man.cgi?query=nmdm%284%29" nmdm(4) driver on the host and using a nmdm(4) device for the serial port used for debugging.

To avoid spewing garbage on the console, connect the nmdm(4) device to the second serial port. This is enabled in the hypervisor by passing `-l com2,/dev/nmdm0B` to bhyve(8). The guest must be configured to use the second serial port for debugging by setting the kernel environment variable `hint.uart.1.flags=0x80` from bhyveload(8). The kgdb(1) debugger on the host connects to the guest by using `target remote /dev/nmdm0A`.

Conclusion

The bhyve hypervisor is a nice addition to a FreeBSD developer’s toolbox. Guests can be used both to develop new features and to test merges to stable branches. The hypervisor has a wide variety of uses beyond developing FreeBSD as well.

John Baldwin joined the FreeBSD Project as a committer in 1999. He has worked in several areas of the system, including SMP infrastructure, the network stack, virtual memory, and device driver support. John has served on the Core and Release Engineering teams and organizes an annual FreeBSD developer summit each spring.