Institutionalizing FreeBSD Isolated and Virtualized Hosts Using `bsdinstall(8)`, `zfs(8)` and `nfsd(8)`

Michael Dexter
editor@callfortesting.org

BSDCan 2018, Ottawa, Canada

The FreeBSD operating system includes the isolation and virtualization multiplicity facilities `chroot(8)`, `jail(8)`, bhyve and Xen, but offers limited facilities for the creation and management of isolated or virtualized targets[1]. The '`bsdinstall(8) jail`', `jail.conf(8)`, and `vmrun.sh` tools support rudimentary Jail and bhyve target creation and management and are not designed for automation using in-base or external tools. This paper will explore how the `bsdinstall(8)`, `nfsd(8)` and `zfs(8)` in-base FreeBSD resources can be leveraged to create authentic, block and file storage-backed operating system installations suitable for use with not only `chroot(8)` and `jail(8)` isolated environments, `bhyve(8)` and Xen virtual machines, but also network-booted hardware machines. Leveraging in-base tools in support of in-base multiplicity facilities enables not only traditional isolated or virtualized "guest" *hosts*[2], but also enables the efficient, more-comprehensive testing of supported and end-of-life releases of BSD operating systems for regression investigation. While the work described in this paper is based on the FreeBSD operating system, its fundamental methodologies described should apply in part or in full to other BSD operating systems.

Isolation vs. Virtualization

BSD Unix has a rich history of providing isolation and virtualization multiplicity facilities. Isolation facilities shall be narrowly defined as those providing one or more *isolated*[3] instances of one or more software Application Binary Interfaces (ABIs), typified by the FreeBSD `chroot(8)` and `jail(8)` facilities providing an isolated FreeBSD or Linux-compatible execution environment managed by a single global kernel. In practice, the "Jail" is a root sub-directory located below the root directory that behaves like an authentic root environment in which executables of a compatible ABI can execute. Virtualization facilities by contrast shall be narrowly defined as those that provide one or more instances of a hardware Instruction Set Architecture (ISA), typified by the FreeBSD bhyve and Xen virtual machine managers/hypervisors providing authentic, albeit virtualized "machines" that allow privileged binary execution, such as by an independent kernel and operating system. In practice, the "VM" is an additional hardware machine of the same architecture (i.e. amd64) within the single hardware machine. The definitive criteria for virtual machines was described by Popek and Goldberg in 1974[4] and dictate that such machines should be "efficient, isolated duplicate[s] of the real machine" whose resources are fully managed by the "virtual machine manager".

While authentic ABI and ISA compatibility are assumed from isolation and virtualization facilities, little formal regard has been given, in theory or practice, to the *fidelity* of operating system installations to isolated or virtualized targets. That is, the installation procedures for independent hardware machines are

1 Kristaps Džonsons. A Taxonomy of Computer System Multiplicity. Swedish Royal Institute of Technology. "The target [whether operating systems, instances or individual processes] defines what is being pluralized, while the strategy is the general means of implementation."
2 In the network, rather than virtualization sense.
3 Kristaps Džonsons. Logical Resource Isolation in the NetBSD Kernel. *AsiaBSDCon 2008.*
4 Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412-421, July 1974.

often repeated for isolated and virtualized destinations with little regard for the correctness of the installation or configuration of the operating system, or the automation of the act of installation. Inversely, traditional operating system installation procedures to independent hardware machines have not benefited from the automation strategies developed for guest "containers", "VMs", "cloud instances", "zones", "droplets[5]" and the like. To this day, a BSD Unix installation, regardless of the specific operating system, involves booting to a read-only CD-ROM, DVD-ROM, or equivalent flash media device and engaging and interactive installer. While this may represent an consistent operator experience that results in an consistent and authentic installation, it reveals the failure of operating systems to adapt to this era of ubiquitous multiplicity. "Cloud" and "container" environments have given rise to "cloud-native" operating systems such as a OSv, CoreOS, Rancher OS, and RedHat Project Atomic, yet these operating systems rarely represent fundamental departures from the Unix model of computing. Arguably, "cloud-native operating system" efforts more demonstrate *misunderstandings* of the Unix model of computing than actual *flaws* in the Unix model of computing that require addressing. The limited awareness of isolated and virtualized installation targets on the part of operating system installers is without question a challenge, but arguably *also* stems from *misunderstandings* of the Unix model of computing. For example, FreeBSD has supported "root on NFS" network booting from as early as its first release but its installers have done little to accommodate this installation scenario. The situation for `jail(8)` environments is better but limited, despite `jail(8)` functionality arriving in the FreeBSD 4.0 release in the year 2000.

FreeBSD bsdinstall(8)

The FreeBSD operating system has a long history of installation utilities and an equally-long history of user dissatisfaction with those utilities. Early `sh(1)` shell script-based installation assistants were considered too "user unfriendly", giving way to the C-based `sysinstall(8)`, and later `sh(1)` and C-based `bsdinstall(8)` and `bsdconfig(8)` installers/configurators, plus the `sh(1)`-based `pc-sysinstall(8)`. While each installer has demonstrated its own unique balance of flexibility, usability and site-specific configurability, the fundamental steps of a Unix system installation have not changed significantly in decades:

- One or more persistent storage devices is sliced or partitioned
- The slices or partitions are formatted with one or more file systems
- Operating system binaries and supporting data are copied to a mounted root file system, typically from a `tar(1)` archive
- Minimal or extensive system configuration is performed, often with dedicated utilities or the modification of text files
- A boot loader is installed to the appropriate location of the persistent storage device

Introduced with FreeBSD 9.0, `bsdinstall(8)` performs all of these steps with a relatively-high level of flexibility in comparison to the installers found in other BSD operating systems. It provides multiple slice and partitioning schemes, a choice of UFS and ZFS file systems, and allows for basic system configuration including networking, users and some services. `bsdinstall(8)` also offers a basic scripting facility with which an automated installation configuration script can be passed into `bsdinstall(8)`, plus a rudimentary "`jail`" mode that allows the operating system to be installed into a root sub-directory sans a kernel. Complimenting `bsdinstall(8)`'s many strengths are several weaknesses:

- `bsdinstall(8)` assumes a "fresh" installation on dedicated hardware, sans "`jail`" mode
- `bsdinstall(8)` assumes the installation of one specific version of the operating system

5 A vendor-specific term that illustrates the breadth of terminology referring to multiplicity targets.

- `bsdinstall(8)` is dependent on several functions provided by `bsdconfig(8)`
- `bsdinstall(8)` does not separate application logic from presentation, with the GPL-licensed `dialog(1)` as a dependence
- While largely `sh(1)`-based, `bsdinstall(8)` ignores '`exit`' statements, limiting debugability
- The C-based components of `bsdinstall(8)` require a different development methodology than its `sh(1)` components

Fortunately, many of these limitations can be overcome in service of reducing the distinctions between independent hardware and isolated or virtualized system installations.

`bsdinstall(8)` Proof-of-Concept Improvements

`bsdinstall(8)` uses environment variables to override its defaults, based on user input either collected by a `dialog(1)`-based user interface or its automated "`script`" mode. `bsdinstall(8)` was designed to work in conjunction with UFS-formatted bootable installation media and to install only the installation media-booted version of FreeBSD to dedicated hardware. The following changes to `bsdinstall(8)` enable its use with isolated and virtualized destinations and are currently under peer review:

- Add support for execution on a read/write ZFS-booted, rather than read-only UFS system
- Add support for "nested[6]" boot environments for use with the `/etc/rc.d/zfsbe rc(8)` script
- Add the support for the installation of boot blocks from the requested distribution sets rather than the installer
- Enable default zpool name collision avoidance
- Add the ability to keep target root directories mounted after installation completion
- Remove dependencies on the GPL-licensed `dialog(1)` library where possible

`bsdinstall(8)` ZFS-Booted Execution

The ZFS support in `bsdinstall(8)` found in `/usr/libexec/bsdinstall/zfsboot` assumes that the installer is booted from UFS and by default exports all zpools during its operation. With the FreeNAS[7] and pfSense[8] FreeBSD derivatives booting to ZFS as of their 9.3 and 2.4 releases respectively, it is logical to conclude that the FreeBSD installer will itself move to ZFS, or at a minimum allow for its execution on a ZFS-booted system. This modification only requires the disabling of the "`f_dprintf "$funcname: Exporting ZFS pools…`" section of the `/usr/libexec/bsdinstall/zfsboot` code.

`bsdinstall(8)` "Nested" Boot Environments

A ZFS "boot environment" is a dataset that serves as the root directory for the operating system that can be accompanied by additional root file systems containing similar or different versions of FreeBSD, or potentially other ZFS-enabled operating systems. By default, a `bsdinstall(8)` "ZFS" installation uses the following layout of zpool, boot environment root, and boot environment: `zroot/ROOT/default`. The default layout shares child datasets such as `/usr` and `/var` as per the following `zfsboot` code:

6 Also referred to as "deep" boot environments
7 https://freenas.org
8 https://pfsense.org

```
ZFSBOOT_DATASETS="
        # DATASET       OPTIONS (comma or space separated; or both)
        /$ZFSBOOT_BEROOT_NAME                           mountpoint=none
        /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME      mountpoint=/
        /tmp            mountpoint=/tmp,exec=on,setuid=off
        /usr            mountpoint=/usr,canmount=off
        /usr/home       # NB: /home is a symlink to /usr/home
        /usr/ports      setuid=off
        /usr/src
        /var            mountpoint=/var,canmount=off
        /var/audit      exec=off,setuid=off
        /var/crash      exec=off,setuid=off
        /var/log        exec=off,setuid=off
        /var/mail       atime=on
        /var/tmp        setuid=off
" # END-QUOTE
```

An example result of the resulting *nested* layout:

```
# zfs list
NAME                 USED  AVAIL  REFER  MOUNTPOINT
zroot               1.04M   195G    96K  /
zroot/ROOT            88K   195G    88K  /tmp
zroot/ROOT/default   352K   195G    88K  /usr
zroot/tmp             88K   195G    88K  /usr/home
zroot/usr            352K   195G    88K  /usr/ports
zroot/usr/home        88K   195G    88K  /usr/src
zroot/usr/ports       88K   195G    88K  /usr/ports
zroot/usr/src         88K   195G    88K  /usr/src
zroot/var            528K   195G    88K  /var
zroot/var/audit       88K   195G    88K  /var/audit
zroot/var/crash       88K   195G    88K  /var/crash
zroot/var/log         88K   195G    88K  /var/log
zroot/var/mail        88K   195G    88K  /var/mail
zroot/var/tmp         88K   195G    88K  /var/tmp
```

The sharing of `/usr` and `/var` is not always desirable, particularly with the goal of supporting non-FreeBSD operating systems. Prefixing the code with the variables representing the zpool and boot environment root will change this behavior:

```
ZFSBOOT_DATASETS="
    /$ZFSBOOT_BEROOT_NAME                                   mountpoint=none
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME              mountpoint=/,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/tmp          mountpoint=tmp,exec=on,setuid=off,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/usr          mountpoint=/usr,canmount=off
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/usr/home     canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/usr/ports    setuid=off,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/usr/src      canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/var          mountpoint=/var,canmount=off
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/var/audit    exec=off,setuid=off,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/var/crash    exec=off,setuid=off,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/var/log      exec=off,setuid=off,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/var/mail     atime=on,canmount=noauto
    /$ZFSBOOT_BEROOT_NAME/$ZFSBOOT_BOOTFS_NAME/var/tmp      setuid=off,canmount=noauto
    " # END-QUOTE
```

Which results in:

```
# zfs list
zroot/ROOT/default            1.04M   195G    96K  /
zroot/ROOT/default/tmp          88K   195G    88K  /tmp
zroot/ROOT/default/usr         352K   195G    88K  /usr
zroot/ROOT/default/usr/home     88K   195G    88K  /usr/home
zroot/ROOT/default/usr/ports    88K   195G    88K  /usr/ports
zroot/ROOT/default/usr/src      88K   195G    88K  /usr/src
zroot/ROOT/default/var         528K   195G    88K  /var
zroot/ROOT/default/var/audit    88K   195G    88K  /var/audit
zroot/ROOT/default/var/crash    88K   195G    88K  /var/crash
```

```
zroot/ROOT/default/var/log      88K   195G    88K  /var/log
zroot/ROOT/default/var/mail     88K   195G    88K  /var/mail
zroot/ROOT/default/var/tmp      88K   195G    88K  /var/tmp
```

As of the FreeBSD 11.1 release, the `/etc/rc.d/zfsbe` `rc(8)` script is provided to automatically mount "child" datasets such as `/usr` to provide proper system operation. This behavior depends on the added "`canmount=noauto`" property which can be seen in the example though this property *must* be performed after bsdinstall(8) has completed installation. With this revised reconfiguration, fully-separate installations of FreeBSD can exist as, for example:

```
zroot/ROOT/default            1.04M   195G    96K  /
zroot/ROOT/default/tmp          88K   195G    88K  /tmp
zroot/ROOT/default/usr         352K   195G    88K  /usr
...
zroot/ROOT/current            1.04M   195G    96K  /
zroot/ROOT/current/tmp          88K   195G    88K  /tmp
zroot/ROOT/current/usr         352K   195G    88K  /usr
...
```

The default boot environment is set with the `bootfs` property of the zpool and can be overridden by the operator at boot time using the FreeBSD interactive boot menu. In addition to these "nested" boot environments, a "flat" boot environment is also desirable that simply places the full operating system in one single dataset with no descendants. The versions of FreeBSD in each boot environment can vary significantly and the operator need only consider if the global boot loader is compatible with all installed versions of FreeBSD. In practice for example, FreeBSD 9.0 and 12-CURRENT can be "dual booted" using this approach. Greater discrepancies in versions are both possible and desired. The selection of a "nested" boot environment dataset layout in `bsdinstall(8)` is configured with the proposed parameter `ZFSBOOT_DATASET_NESTING`.

Distribution Set Boot Blocks

By default, `bsdinstall(8)` installs the boot blocks for the target operating system from the booted OS of the installer. The synchronization of these may be logical for a "fresh install" but is not suitable for say, installations of a "CURRENT" snapshot of the operating system adjacent to a "STABLE" version. To overcome this limitation, a proposed parameter `BOOT_BLOCKS_FROM_DISTSET` is added to `bsdinstall(8),` plus a path to the desired boot blocks, `BOOT_BLOCKS_PATH=/boot`. The FreeBSD boot blocks are contained in the `/boot` directory of the `base.txz` distribution set. If the `BOOT_BLOCKS_FROM_DISTSET` variable is set, the modified `bsdinstall(8)` selectively extracts the `/boot` directory by using `tar(1)` excludes to the destination `/tmp/bsdinstall_bootblocks/boot`:

```
cat $BSDINSTALL_DISTDIR/base.txz | \
       tar -xf - -C /tmp/bsdinstall_bootblocks/ \
       --exclude ./.cshrc \
       --exclude ./.profile \
       --exclude ./COPYRIGHT \
       --exclude ./bin \
```

...

The boot block installation steps in turn operate from the proposed `BOOT_BLOCKS_PATH` directory. A FreeBSD 12-CURRENT installation would thus be installed with the corresponding FreeBSD 12-CURRENT boot blocks.

zpool Name Collision Avoidance

By default, all ZFS pools configured with `bsdinstall(8)` are named "`zroot`" unless the user provides a custom name. Separate boot pools used by GELI encryption and some non-GPT partitioning layouts default to "`bootpool`". Should a boot environment be automatically configured with a pool named "`zroot`" from a primary system booted from the default of "zroot", a conflict will occur. Because an operator has an option to provide a custom zpool name, a simple mechanism is employed to avoid conflicts: the name "`zroot`" is simply incremented with a number (i.e. `zroot1`) should "`zroot`" already exists. The following code will remove potential zpool conflicts in `bsdinstall(8)`:

```
_counter=""
zpools=$(zpool list -H -o name)
        for _name in $zpools ; do
                if [ "$_name" = "${ZFSBOOT_POOL_NAME}$_counter" ]; then
                        _counter=$(($_counter+1))
                fi
        done
        ZFSBOOT_POOL_NAME="${ZFSBOOT_POOL_NAME}$_counter"
```

Mounted Destinations for Further Configuration

By default, `bsdinstall(8)` performs various "clean up" operations upon completion of the installation and prior to the assumed reboot of the hardware to which FreeBSD was installed. This behavior is not necessarily desired during installation to additional targets given that the installation source operating system may have extensive configuration resources available, rather than the limited set on a traditional "install" system from which `bsdinstall(8)` is run. The proposed `KEEP_MOUNTED` variable and an "if" statement at the end of the script "verb" `/usr/libexec/bsdinstall/` allow `bsdinstall(8)` to support this ability:

```
if [ ! $KEEP_MOUNTED ]; then
        if [ $ZFSBOOT_DISKS ]; then
                cat /tmp/bsdinstall_zpools | while read _zpool ; do
                f_dprintf "Exporting zpool $_zpool"
                        zpool export $_zpool
                done
        else
                f_dprintf "Running bsdinstall umount"
                bsdinstall umount
        fi
fi
```

Reduce `dialog(1)` Dependencies

`bsdinstall(8)`'s extensive dependence on the `dialog(1)` "dialog box" library provides its biggest challenge to its improvement for three key reasons:

- This dependency appears to be responsible for `bsdinstall(8)`'s inability to respond to "exit" statement, described as "extremely difficult to fix[9]" by `bsdinstall(8)` co-author Devin Teske
- `bsdinstall(8)` does not separate application logic from its `dialog(1)`-based user interface
- `dialog(1)` is undesired as a dependency because of its licensing. Acceptably-licensed alternatives exist but do not provide the required feature set

It is beyond the scope of this exploration to solve architectural flaws in bsdinstall(8).

9 In private conversation.

Fortunately however, the "script", "zfsboot" and "config" verbs of bsdinstall(8) automation called with 'bsdinstall script <script name>' have only one dialog(1) dependency in the form of the checksum verb called within the "script" verb. This dependency can be easily resolved by copying the core checksum logic from the "checksum" verb found in /usr/libexec/bsdinstall/checksum, into the script verb. This change would enable bsdinstall(8) "automated install" scripting to work without any dependence on dialog(1) and its dependency, ncurses(3X). The modified bsdinstall(8) can be used with the following automation script to create a disk image-backed virtual machine installation formatted for UFS or ZFS:

```
export BSDINSTALL_DISTDIR="/pub/FreeBSD/snapshots/amd64/amd64/12.0-CURRENT"
export ZFSBOOT_DISKS="md0"
export ZFSBOOT_PARTITION_SCHEME="GPT"
export ZFSBOOT_POOL_NAME="zroot"
export ZFSBOOT_BEROOT_NAME="ROOT"
export ZFSBOOT_BOOTFS_NAME="default"
export ZFSBOOT_DATASET_NESTING="1"
export BOOT_BLOCKS_FROM_DISTSET="1"
# Alternative UFS layout in place of the ZFS variable ZFSBOOT_DATASETS
#export PARTITIONS="md0 {512M freebsd-ufs /, 100M freebsd-swap, 512M freebsd-ufs, /var, auto
freebsd-ufs /usr }"
```

Note that BSDINSTALL_DISTDIR refers to a local copy of the standard FreeBSD download mirror.

Preparation of a "malloc" RAM DISK for testing purposes:

```
# mdconfig -t malloc -s 4G
md0
```

FreeBSD installation with bsdinstall(8):

```
# bsdconfig script <the script>
```

bhyve booting of the resulting image with vmrun.sh:

```
# sh /usr/share/examples/bhyve/vmrun.sh -m 2G -d /dev/md0 vm
```

At this point, several key goals have been achieved by modifying bsdinstall(8):

- A virtual machine installation of a snapshot version of FreeBSD to a block device is performed using the in-base bsdinstall(8)
- An optional "nested" boot environment ZFS dataset layout is enabled
- The installed system version receives the corresponding version boot blocks
- The installation is performed from a ZFS-booted system
- The *automated* installation is performed without the use of dialog(1)

However, the "fresh install" orientation of bsdinstall(8) that by default only installs a single version of FreeBSD from distribution sets contained in /usr/freebsd-dist, fails to do justice to the broad operating system support provided by the bhyve and Xen hypervisors. bhyve is capable of booting FreeBSD 5.0 onward and while FreeBSD 5.0 is packaged for use with the older sysinstall(8) installer, the FreeBSD 5.0 through 8.4 distribution sets are easily repackaged for use with bsdinstall(8). Recall that the BSD Unix installation strategy has not changed significantly since before the announcement of FreeBSD, nor has the CD-ROM ISO format. The primary change to the FreeBSD binary packaging between FreeBSD 5.0-RELEASE and 11.1-RELEASE was the move from gzip(1)-compressed, tar(1)-archived floppy disk-sized segmented distribution sets (i.e. base.aa, base.ab…), to xz(1)-compressed, non-segmented distribution sets (i.e. base.txz). Ultimately, it is desirable to move previous release ISOs

such as FreeBSD 5.0 entirely to the contemporary layout used by `bsdinstall(8)` to facilitate regression investigation:

1. Move the FreeBSD 5.0 ISO to the contemporary naming for consistent management:

```
# mv 5.0-RELEASE-i386-disc1.iso FreeBSD-5.0-RELEASE-i386-disc1.iso
```

2. Extract the ISO contents using `libarchive(3)` via `tar(1)`:

```
# tar -xf FreeBSD-5.0-RELEASE-i386-disc1.iso -C /tmp/freebsd5.0iso/
```

3. Extract the segmented distribution sets:

```
# cat /tmp/freebsd5.0/base/base.?? | tar --unlink -xpzf - -C /tmp/freebsd5.0/
```

4. Re-archive the base binaries with the modern "`.txz`" naming but *without* `xz(1)` compression:

```
# tar -cf base.txz -C /tmp/freebsd5.0/base
```

5. Generate a contemporary `MANIFEST` file containing archive checksums:

```
# sh /usr/src/release/scripts/make-manifest.sh base.t?? >> MANIFEST
```

Repeat for the `kernel.txz` and other distribution sets as desired. An uncompressed archive with compressed naming is used for its backward compatibility (easily accessed with '`base.t??`') and because the operation is assumed to be performed on a contemporary ZFS file system with compression enabled. The time saved by not compressing the distribution set archives alone makes this strategy desirable.

The repackaging of pre-`bsdinstall(8)` FreeBSD 5.0 through 8.4 distribution sets is easily scripted because of the stationary nature of past releases. While source distribution sets can also be repackaged, they are best checked out from a local SVN mirror in order to include SVN metadata that will be useful during regression investigation. Finally, past binary packages are of particular value for regression investigation because their dependencies and source archives are unlikely to be available.

With this groundwork in place, an environment is established in which `bsdinstall(8)` can be used to produce block device-backed virtual machines using end-of-life versions of FreeBSD. Compatibility issues arise however with the storage and network interfaces provided by bhyve. VirtIO devices only became available in FreeBSD 8.4 and while bhyve provides e1000 network and AHCI storage emulation, these are inadequate for supporting FreeBSD 5.0. The ATA emulation project[10] is not yet of use for this purpose but the ne2000 project[11] from the same author is. This fact provides a unique opportunity.

In addition to supporting `sh(1)`, `tar(1)` and ISO-9660 since its founding, FreeBSD has also supported booting from the Network File System (NFS) from as early as FreeBSD version 1.0. These timeless standards, combined with the ZFS boot environments, the high-performance kernel-level NFS daemon, plus the bhyve and Xen hypervisors, produce an environment that not only supports the booting of FreeBSD 5.0 onward virtual machines, FreeBSD derivatives such as TrueOS, FreeNAS and pfSense, but also on network-connected hardware machines and in a manner that is not limited to the FreeBSD operating system.

10 https://reviews.freebsd.org/D5473
11 https://wiki.freebsd.org/SummerOfCode2015/NE2000EmulationForBhyve

Networked Boot Environments

Recall that ZFS boot environments allow for multiple versions of a ZFS-enabled operating system to be installed on a common zpool. If "nested", a boot environment provides a fully-separated operating system root directory for each installation without any shared directories such as `/usr`. Boot environments can also:

- Be used for `chroot(8)` and `jail(8)` isolation environments
- Theoretically hardware-boot foreign ZFS-enabled OSs such as Illumos or NetBSD
- Be shared via NFS to provide local virtual machine, and networked hardware machine booting of the operating system they contain
- Provide ZFS features such as snapshotting to ZFS-unaware operating systems such as end-of-live versions of FreeBSD or foreign operating systems
- Contain nested root directories such as for custom-built operating system releases that are further shared over NFS to create a highly-flexible development, production or regression investigation environment

An end-of-life version of FreeBSD older that the 7.0 release or most non-FreeBSD operating systems will obviously not support hardware booting from a ZFS boot environment but this does not diminish the value of using the ZFS boot environment infrastructure for them. One or more authentic ZFS datasets shared over the ubiquitous NFS protocol provide a valid and consistent strategy for the management of a network boot environment for virtual and hardware machines. As a file, rather than block-backed storage strategy, this approach provides many advantages when developing, testing and experimenting, thanks to the "panopticon" transparency that the hardware machine has over its isolated or virtualized targets. Data can easily be transferred to and from the file system system of the hardware machine and isolated or virtualized targets, and a formal communication mechanism using ssh(1) or VirtIO Serial should be considered for privileged communication between the many environments. With a formal communication mechanism, a ZFS-unaware operating system could request ZFS administrative operations such as snapshots from its ZFS-enabled "host".

Arguably, ZFS boot environments provide the ultimate Unix `chroot(8)` environment functionality.

Networked Boot Environment Components

`nfsd(8)`

Beyond the ZFS boot environments themselves, the FreeBSD in-kernel `nfsd(8)` provides the other half of the "heavy lifting" in the Networked Boot Environments scenario. Because ZFS datasets are file systems in the traditional sense with each each maintaining unique inode numbers, it is important to understand that nested ZFS datasets must each be shared over NFS individually. The NFS '`-alldirs`' argument in the `exports(5)` configuration file will not provide the desired behavior. However, the `/etc/rc.d/zfsbe` `rc(8)` script provides the logic to overcome this challenge by operating on nested datasets based on various properties such as their "`canmount`" property. For example, the syntax to mount the nested datasets of a requested boot environment would be:

```
zfs list -rH -o mountpoint,name,canmount,mounted -s mountpoint -t filesystem zroot/ROOT/default | \
while read _mp _name _canmount _mounted ; do
        [ "$_canmount" = "off" ] && continue
        [ "$_mounted" = "yes" ] && continue
        case "$_mp" in
                none|legacy)
                ;;
                *)
                        #echo Making /ROOT/default${_mp} if missing
                        [ -d /ROOT/default${_mp} ] || mkdir -p /ROOT/default${_mp}
                        #echo Mounting $_name on /ROOT/default${_mp}
                        mount -t zfs $_name /ROOT/default${_mp}
                        [ $? = 0 ] || { echo mount failed ; exit 1 ; }
                ;;
        esac
done
```

This same logic can be followed by an ‘`unmount`’ operation and NFS sharing operations, observing that the nested datasets must be accessed from the virtual machine’s `fstab(5)`. While this fact, plus the components of the traditional NFS/TFTP/DHCPd infrastructure stack represent a reasonable level of complexity, this complexity is justified in service of the goal and has proven to be quite robust.

With basic NFS sharing infrastructure in place, one additional tool is required to boot root-on-NFS FreeBSD virtual machines using `bhyveload(8)`. The `github.com/stblassitude/boot_root_nfs` utility provides the ability to “taste” an NFS share for key information *outside* of a PXE environment. This utility provides these dynamic parameters to `bhyveload(8)` provided by the NFS server:

```
-e boot.nfsroot.server=192.168.1.202
-e boot.nfsroot.nfshandle=X631083b5dea37b840a00040000000000e10000000000000000000000X
-e boot.nfsroot.nfshandlelen=28
-e boot.nfsroot.path=/mnt
```

These parameters, plus `boot.netif.hwaddr`, `boot.netif.ip`, `boot.netif.name`, `boot.netif.netmask`, `boot.nfsroot.path`, `vfs.root.mountfrom` and `vfs.root.mountfrom.options` are all the boot loader parameters needed to boot FreeBSD over NFS *without* PXE. A PXE environment can be added to this storage infrastructure, enabling UEFI-GOP bhyve virtual machine booting, hardware machine booting and potentially Xen PXE virtual machine booting. Finally, the `wake(8)` command can be used to “Wake on LAN” physical machines for the purpose of network booting them. It is reasonable to propose that the `boot_root_nfs` functionality be added to `bhyveload(8)`.

Support Projects

In addition to the `boot_root_nfs` utility, the bhyve NE2000 emulated network adapter and ATA emulation projects enable the network booting of FreeBSD 5.0 onward. This emulated NE2000 device is scheduled for inclusion in `bhyve(8)` once adequate validation has been performed and the ATA emulation project remains in development.

`be(8)` Proof-of-Concept Utility

`be(8)` is a proof-of-concept utility for the unified utility to create, share, manage and destroy boot environments in the manner described above. It is intended to compliment the `zpool(8)` and `zfs(8)` utilities and follows a similar syntax:

```
# be create -l(ayout) freebsd zroot/ROOT/freebsd11.1
# be mount zroot/ROOT/freebsd11.1
# be install -a amd64 -r 11.1 zroot/ROOT/freebsd11.1
# be sharenfs zroot/ROOT/freebsd11.1
# be bootnfs zroot/ROOT/freebsd11.1
```

Because boot environments lack a default `mountpoint` on the hardware machine's file system, it is reasonable to mount them to the boot environment root path (`/ROOT/*`) to avoid confusion. Boot environment `zroot/ROOT/freebsd11.1` would logically be mounted on `/ROOT/freebsd11.1` for configuration, `chroot(8)` or `jail(8)` booting, and NFS sharing. In practice, the boot environment root "`ROOT`" could be abbreviated "`be`" for brevity. This change is easily made to a `bsdinstall(8)` automation script and could be an user-configurable option.

Ultimately, the `be(8)` proof of concept promises to enable simultaneous scenarios such as:

`zroot/ROOT/freebsd11.1` → Hardware machine boot
`zroot/ROOT/freebsd10.4` → `chroot(8)` isolated environment boot
`zroot/ROOT/freebsd9.3` → `jail(8)` isolated environment boot
`zroot/ROOT/freebsd8.4` → `bhyve(8)` virtual machine boot
`zroot/ROOT/trueos` → `bhyve(8)` virtual machine boot
`zroot/ROOT/freenas` → `bhyve(8)` virtual machine boot
`zroot/ROOT/pfsense` → `bhyve(8)` virtual machine boot
`zroot/ROOT/illumos` → `bhyve(8)` virtual machine boot
`zroot/ROOT/openbsd` → LAN PXE hardware machine boot

Block Storage-Backed VM Opportunities

Up to this point, we have proposed modifications to `bsdinstall(8)` to overcome some of its limitations with for use with block-backed virtual machines, and leveraged its logic for use with file-backed isolated environments and virtual machines. It is desirable to leverage the logic within `bsdinstall(8)` to provide authentic FreeBSD installations to isolated and virtual targets. While a modified `bsdinstall(8)` can perform reasonable scripted installations to block devices using UFS or ZFS, further exploration is in order. Where `be(8)` can create and share boot environments, a complimentary utility could validate (`smart(8)`[12]), create (`dd(1)` and `truncate(1)` disk images and `zfs(8)` zvols) and share (`ctl(8)` iSCSI, Fibre Channel and `ggatec(8)`) block devices. In conjunction with `be(8)`'s ability to provision and share file-level storage, a complimentary utility that provisions and shares block-level storage would constitute the key building blocks of a Network Attached Storage (NAS) system. The most challenging aspect of addressing the block-level challenges would be the rewriting of the UFS disk configuration steps of `bsdinstall(8)`, but the `/usr/libexec/bsdinstall/zfsboot` provides much of the infrastructure needed for disk partitioning using multiple schemes. The `pc-sysinstall(8)` utility may also provide reusable logic to achieve this goal.

12 Michael Dexter: diskctl(8): A permissively-licensed S.M.A.R.T. and raw disk command utility framework. *AsiaBSDCon 2016* | https://github.com/ctuffli/smart

Additional Challenges

While the FreeBSD operating system provides up-to-date OpenZFS and a performant NFSd, its `tftpd(8)` service managed by `inetd(8)` is not designed for multiplicity use. While OpenZFS and NFSd can provide dozens, if not hundreds of parallel boot environments and exports respectively, the in-base `tftpd(8)` is only designed for a single TFTP directory share in service of PXE network booting. Candidate alternatives to the FreeBSD in-base TFTPd include the `net/freebsd-tftp` port which was at one point intended to replace the in-base `tftpd(8)` and `tftp(8)`, `ftp/tftp-hpa` from the Linux kernel project, `ftp/atftp` and `net/utftpd`, all of which are available under a number of desirable and undesirable licenses. FreeBSD also includes a rudimentary implementation of the ZFS `sharenfs` property, which allows a dataset to be shared using the in-base NFSd. Unfortunately, the ZFS `sharenfs` functionality suffers from a lack of `mountpoint` conflict checking, the persistence of its private `/etc/zfs/exports` configuration file after the disabling of the property, and unfigurable NFS sharing options. Some of these shortcomings should not be difficult to address but raise the question of if full, in-base PXE server functionality is desired in FreeBSD and other BSD Unix operating systems.

The in-base `vmrun.sh` bhyve virtual machine boot utility is also a candidate for improvement. `vmrun.sh` currently lacks any notion of persistent virtual machine configurations akin to the `jail.conf(5)` configuration format. While the preferred configuration syntax remains to be determined, the steady advancement of the underlying tools will facilitate the development of in-base and external isolated and virtualized target management tools towards the goal of their *institutionalized* life cycle management.

Finally, the 9P[13] file sharing protocol developed for Bell Labs' Plan 9 operating system would provide a desirable compliment to NFS for local file-level virtual machine storage in the Networked Boot Environment scenario, but it is not production-ready on FreeBSD.

Practical Applications

While a series of proposed `bsdinstall(8)` modifications can extend the utility's usefulness to block-backed virtual machines, the proof-of-concept `be(8)` is intended to provide the basis for a *framework* for the creation and management of isolated environments and virtual machines. It intentionally uses `sh(1)` `OPTARG` "flags" (i.e. `-p <path>`) to facilitate further scripting with `sh(1)`. For example, a list of releases could be cycled through using a for loop, provisioning each one with `be(8)`. At present, `bsdinstall(8)` can be fully-automated by generating automation scripts and calling them with the `bsdinstall(8)` "`script`" verb. Like `bsdinstall(8)`, `be(8)` also relies on reasonable defaults that are easily overridden by the operator. This approach keeps the syntax as brief as possible for interactive command-line use and as scriptable as possible for automated use.

13 https://en.wikipedia.org/wiki/9P_(protocol)

As a human and machine friendly isolated and virtual target deployment mechanism, `be(8)` can be used to:

- Deploy isolated environments and virtual machines
- Preflight new boot environment installations[14]
- Investigate regressions by sequentially testing each major release and narrowing the scope to individual commits in predetermined and automatically-determined ranges as needed[15]
- Determine which version of an OS can build its previous and subsequent releases
- Determine the minimum buildable and executable build settings for *every* release of an operating system to facilitate rapid compilation
- Determine the optimal number of virtual CPUs for any given task on any given OS
- Determine the optimal number of parallel build "jobs"
- Offload operating system build tasks to external hardware machines
- Determine in an operating system is progressing or regressing with regard to metrics ranging from the performance of various subsystems to the ratio of manual pages to components

Conclusions

The goal of institutionalizing isolated and virtualized targets in FreeBSD and other BSD Unix operating systems is achievable primarily by removing the distinction between hardware and virtual machine installations. This paper has demonstrated how the FreeBSD installer, `bsdinstall(8)` can be modified to greatly enhance its use with block-backed virtual machines in service of this goal. This paper has also demonstrated how the proof-of-concept `be(8)` utility could facilitate the provisioning of file-backed isolated environments and virtual machines in a consistent manner for use with a hardware machine, virtual machines, networked hardware machines, and isolated environments. The fine-grained configuration and management of the resulting hosts can be performed with existing in-base and external tools, but many opportunities exist for those configuration abilities to be brought in-base, considering that many Unix administrative tasks have not changed significantly in decades. As such, "cloud" and "container" computing can rightfully be referred to as *Unix computing.*

14 bhyve-Bootable OpenZFS Boot Environments http://callfortesting.org/bhyve-boot-environments/

15 Categorizing development commits by subsystem would greatly accelerate this process.