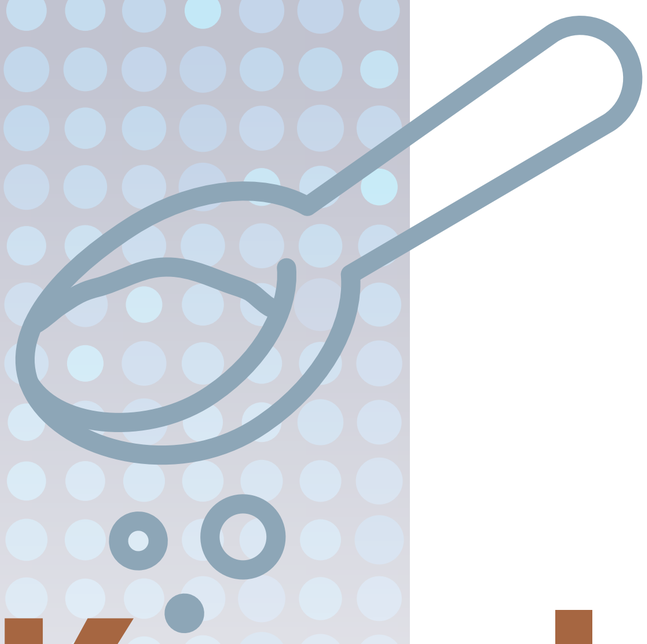


Kernel Development Recipes

BY MARK JOHNSTON

A question that I frequently see is something along the lines of, "How do I get started in operating system kernel development?" This is difficult to answer in general, but there is a simpler side to the question, which is, "How can I set myself up to (efficiently) build and test kernel changes?" In other words, while submitting one's first kernel patch is a significant milestone, a frequent contributor may be working on multiple different patches, testing patches (possibly obtained from other developers), bisecting to find the cause of a regression, or debugging a kernel panic. It is important to have a workflow that minimizes the amount of time spent waiting or fiddling with configuration knobs or shell scripts.

FreeBSD, being a large and rather long-lived project (even older than some of its developers) and targeting a variety of hardware platforms (from palm-sized SoCs to large multi-core servers with terabytes of RAM), does not have a one-size-fits-all approach to these kinds of tasks. However, the benefits of a fast edit-compile-test loop are universal; this article seeks to illustrate a few tricks which can reduce the friction of many kernel development tasks.

The steps below assume a FreeBSD host and POSIX-compatible shell, such `as /bin/sh`. While the project currently does not provide much in the way of canned scripts for building, booting and testing kernel changes, some of the suggestions here can be incorporated into one's development environment or into a CI system shared by multiple developers.

Git Worktrees

Before starting any kind of work on FreeBSD one needs a copy of the src repository:

```
$ git clone https://git.freebsd.org/src.git freebsd
```

When working on multiple tasks, it will be useful to have multiple copies of the FreeBSD source files. Having a single copy becomes awkward if a build of the currently checked out branch is running and while waiting you want to switch to a different branch to work on something else, or possibly because switching branches would update timestamps on source files and thus unnecessarily slow down future incremental builds.

It is of course possible to keep multiple clones of a repository, but a better solution is to use `git` worktrees, which let one checkout multiple branches from a single clone. This requires less

disk space and ensures that all of your work is contained in a single copy of the repository. For example, it can be useful to have worktrees for frequently accessed stable and release branches:

```
$ cd freebsd
$ git worktree add dev/stable/13 origin/stable/13
$ git worktree add dev/releng/13.0 origin/releng/13.0
$ git worktree add dev/stable/12 origin/stable/12
$ git worktree add dev/releng/12.2 origin/releng/12.2
...
```

Note that worktrees may be located outside of the clone, for example:

```
$ git worktree add ../freebsd-stable/13 origin/stable/13
```

When working on a larger project, it is useful to keep a worktree dedicated to that branch.

Building and Booting a Custom Kernel — Quickly

Suppose that you have written a small, simple kernel patch and want to do some sanity testing before submitting it for review. The standard command for building a FreeBSD kernel is well-known:

```
$ cd freebsd
$ make buildkernel
```

This will perform a clean, single-threaded kernel build. It is usually preferable to use as much CPU as is available, so consider adding `-j $(sysctl -n hw.ncpu)` to the `make(1)` flags: this will run as many build jobs in parallel as there are CPUs in the system. Once a kernel has been built from a particular source path, it is not usually necessary to rebuild every single source file each time a change needs to be tested — an incremental rebuild is sufficient. To request an incremental build, add the `-DKERNFAST` flag to the `make(1)` invocation:

```
$ make -j $(sysctl -n hw.ncpu) -DKERNFAST buildkernel
```

By default, a `buildkernel` will rebuild not only the kernel, but also every single kernel module that comes with it, which on an amd64 system running a `GENERIC` kernel comes out to 822 modules. Many of these modules are also linked into the kernel proper (depending on the kernel configuration file in use), so a `buildkernel` may spend considerable time rebuilding modules that will never be loaded. The `MODULES_OVERRIDE` variable can be used to override this behaviour; instead of building a kernel and all modules, a `buildkernel` with `MODULES_OVERRIDE` will build only a kernel and the modules specified by the variable's value. For example, to build a kernel together with only the `tmpfs.ko` and `nullfs.ko` modules, run the following:

```
$ make -j $(sysctl -n hw.ncpu) -DKERNFAST \
  MODULES_OVERRIDE="tmpfs nullfs" buildkernel
```

If you know that your testing requires only a handful of modules, your build times can benefit from setting `MODULES_OVERRIDE` since this may greatly reduce the number of filesystem accesses performed during a build. If the source tree is located on a slow disk, or remotely and accessed over NFS, for example, the `MODULES_OVERRIDE` option may save a lot of time.

Now that a test kernel is available, it can be tested. The procedure for testing a change depends heavily on the nature of the change; a developer working on a WiFi driver will have different testing workflows than a developer working on bringing up FreeBSD on a new platform or on the kernel memory allocator. For many purposes, however, a simple `bhyve` virtual machine (VM) is sufficient.

The FreeBSD project provides VM images that are handy for use in testing kernel changes. While it is of course possible to build VM images locally, using the `release(7)` scripts for example, the pre-built images are quite convenient:

```
$ IMAGE="FreeBSD-14.0-CURRENT-amd64.raw.xz"
$ URL="https://ftp.freebsd.org/pub/FreeBSD/snapshots/VM-IMAGES/14.0-CURRENT/amd64/Latest"
$ fetch -o "/tmp/${IMAGE}" "${URL}/${IMAGE}"
$ unxz "/tmp/${IMAGE}"
$ sudo pkg install -y bhyve-firmware
$ sudo sh /usr/share/examples/bhyve/vmrun.sh -E -d \
  "/tmp/${IMAGE%.xz}" myvm
```

This boots up the snapshot image using bhyve, in a VM called “myvm”. However, the VM will be running the kernel included with the snapshot. There are several ways to update the image’s kernel. For instance, a copy of the source tree could be copied or mounted into the booted VM, and a new kernel can be built within the VM. However, this will be slow unless the VM is given a large amount of CPU and RAM, and poses the logistical problem of exporting the source tree. Another approach could be to mount the disk image on the host and install a new kernel directly, but this requires some synchronization to ensure that the VM and the host do not have the image mounted at the same time.

A different approach is to provide a second disk containing only the custom kernel and modules, and configuring the VM to boot from that instead. Such disk images can be created quickly and without any special privileges on the host. First, the following commands can be used to create such an image:

```
$ cd /usr/src
$ make buildkernel
$ make installkernel -DNO_ROOT DESTDIR=/tmp/kernel
$ cd /tmp/kernel
$ makefs -B little -S 512 -Z -o version=2 /tmp/kernfs METALOG
$ rm -f /tmp/kernfs.raw
$ mkimg -s gpt -f raw -S 512 -p freebsd-ufs/kern:=/tmp/kernfs \
  -o /tmp/kernfs.raw
```

The `make` commands build and install a kernel to `/tmp/kernel` without requiring root privileges. This also creates an `mtree(8)` manifest in `/tmp/kernel/METALOG` which is used by

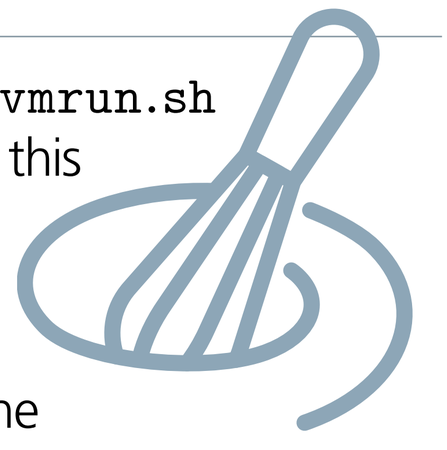
`makefs(8)` to build a small filesystem. Finally, `mkimg(1)` adds a GPT to the filesystem, making it accessible to the FreeBSD boot loader.

Now we can boot the VM again with the extra disk:

```
$ sudo sh /usr/share/examples/bhyve/vmrun.sh -E -d /tmp/kernfs.raw \
  -d "${IMAGE%.xz}" myvm
```

This still boots the kernel from the original image. However, the boot loader can be configured to load the kernel from `kernfs.raw` instead, by adding the following line to `/boot/loader.conf`:

```
kernel="disk0p1:/boot/kernel"
```



"`disk0p1`" here refers to the first partition of disk0, the first disk listed in the `vmrun.sh` command-line arguments, which in this case is `/tmp/kernfs.raw`. After making this change and rebooting, the VM should boot into the custom kernel.

The custom kernel filesystem is not mounted by default, which means that tools like `kldload(8)` will not be able to automatically load kernel modules corresponding to the custom kernel. To remedy this, add the following lines to the corresponding system configuration files:

`/boot/loader.conf`:

```
# Make sure that nullfs is available.
nullfs_load="YES"
```

`/etc/sysctl.conf`:

```
# Fix up paths used by the kernel linker.
kern.bootfile=/boot/kernel/kernel
kern.module_path=/boot/kernel
```

`/etc/fstab`:

```
# Mount the custom kernel filesystem at /boot/kernel.
/dev/gpt/kern /mnt ufs ro 0 0
/mnt/boot/kernel /boot/kernel nullfs ro 0 0
```

Finally, consider adding `autoboot_delay=1` to `/boot/loader.conf`: this reduces the loader delay from ten seconds to one, which helps considerably when reboots are frequent.

While the setup was a bit involved, it only needs to be performed once, and we now have a way to quickly boot up a freshly built kernel! Kernel builds can be done while the VM is running, and rebooting the VM will cause the latest kernel build to be loaded and run. When creating more elaborate testing environments, it may be desirable to build the base VM image locally as well, in which case the configuration updates described above may be automated.

Debugging a Custom Kernel

One benefit of using `bhyve(8)` is the GDB protocol stub available to the host. QEMU has a similar feature. Using this, the host system can run a debugger on the guest kernel. Since our custom kernel was built on the host, this functionality is trivial to use. The `vmrun.sh` currently does not support enabling the GDB stub, but it can be enabled using a raw bhvye(8) invocation:

```
# bhyve -c 1 -m 512M -H -A -P -G :1234 \
        -s 0:0,hostbridge \
        -s 1:0,lpc \
        -s 2:0,virtio-blk,kernfs.raw \
        -s 3:0,virtio-blk,FreeBSD-14.0-CUREENT-amd64.raw \
        -l com1,stdio \
        -l bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd \
        myvm
```

Here, the `-G :1234` parameter instructs `bhyve(8)` to listen on port 1234 for connections from a debugger. When booting a VM, `bhyve(8)` may optionally pause waiting for a connection before booting the kernel; this is handy for debugging problems that arise early during kernel boot-up. To enable this, specify `-G w:1234`.

While the VM is running (or waiting for a connection), the `kgdb` program (installed with the `gdb` package) can attach to the guest using port 1234:

```
$ kgdb -q /tmp/kernel/usr/lib/debug/boot/kernel/kernel.debug
Reading symbols from /tmp/kernel/usr/lib/debug/boot/kernel/kernel.debug...
(kgdb) target remote localhost:1234
Remote debugging using localhost:1234
cpu_idle_acpi (sbt=432162053) at
/usr/home/markj/src/freebsd/sys/x86/x86/cpu_machdep.c:551
551             atomic_store_int(state, STATE_RUNNING);
(kgdb) set solib-search-path /tmp/kernel/usr/lib/debug/boot/kernel
Reading symbols from
/tmp/kernel/usr/lib/debug/boot/kernel/nullfs.ko.debug...
(kgdb) bt
#0  cpu_idle_acpi (sbt=432162053) at
/usr/home/markj/src/freebsd/sys/x86/x86/cpu_machdep.c:551
#1  0xffffffff81096ccf in cpu_idle (busy=0) at
/usr/home/markj/src/freebsd/sys/x86/x86/cpu_machdep.c:668
#2  0xffffffff80c62b41 in sched_idletd (dummy=<optimized out>,
dummy@entry=0x0 <nullfs_init>)
    at /usr/home/markj/src/freebsd/sys/kern/sched_ule.c:2952
#3  0xffffffff80be838a in fork_exit (callout=0xffffffff80c62660
<sched_idletd>, arg=0x0 <nullfs_init>,
    frame=0xfffffe0001141f40) at
/usr/home/markj/src/freebsd/sys/kern/kern_fork.c:1088
#4  <signal handler called>
(kgdb)
```

At this point the VM is suspended waiting for a command from the debugger. The `continue` command can be used to resume the VM, and hitting ctrl-C in the debugger window will suspend

the VM again. This functionality is extremely useful for debugging kernel deadlocks and boot-time problems. It is also possible to attach a debugger after the guest kernel has panicked, making it easy to inspect threads and local variables without having to configure and recover kernel dumps.

Testing a Custom Kernel

We now have a fairly quick loop for incrementally testing changes to the FreeBSD kernel, as well as some tooling for debugging problems when they arise. At this point some manual testing might indicate that the patch is correct and ready for review. It may be useful though to run some automated tests to increase confidence in the patch; the FreeBSD kernel is a large, monolithic body of code, and there is always some potential for unforeseen regressions. A good place to start is the FreeBSD regression test suite.

If you followed the steps above to set up a kernel development VM, then the test suite is already installed; use the following commands to run it:

```
# cd /usr/tests
# kyua -v test_suites.FreeBSD.allow_sysctl_side_effects=1 test
```

The `allow_sysctl_side_effects` flag enables tests which depend on being able to modify global sysctl values, which is perfectly fine in a dedicated VM. Some tests will also be skipped if they depend on third-party ports, such as Python. After a run, a summary of results (including skipped tests) can be viewed with:

```
# kyua report
```

A `bhyve` VM can be set up to automatically run the test suite upon booting up. One simple way to achieve this is to add an `/etc/rc.local` script which runs the test suite, prints results to the console, and shuts down the VM. A separate disk could be used to store the output of `kyua report`, making the results easy to recover on the host.

The regression test suite covers a large number of FreeBSD features but is designed to complete quickly. So while it can help find bugs with relatively little effort, more intensive stress tests may be required. FreeBSD has several ways to further test kernel changes.

stress2

`stress2` is a large stress test suite maintained by Peter Holm. It contains many hundreds of regression tests for the core kernel's filesystem and memory management subsystems. The stress2 suite is included in the FreeBSD source tree, in `tools/test/stress2`, but is not part of an installation. To run the tests, assuming a source tree is available in the test system, run:

```
# cp -R ${SRCDIR}/tools/test/stress2 /tmp/stress2
# pw user add stress2
# cd /tmp/stress2
# echo stress2 | make test
```

The `stress2` suite requires at least several gigabytes of RAM and a large disk. It can take multiple days to complete but is an excellent way to test systemic changes to the kernel. The individual tests are located under the `misc` subdirectory and can be run directly.

syzkaller

Finally, `syzkaller` has emerged as an effective tool for exercising portions of the kernel reachable from the system call interface. Being a fuzzer, it is not particularly useful for proving the correctness of a change, but it is very good at triggering rarely executed error paths and so can help validate error handling code which may otherwise be difficult to trigger. It is also effective at provoking race conditions.

A detailed overview of `syzkaller` appeared in a previous *FreeBSD Journal* article. Setting up a `syzkaller` instance is a somewhat involved task. Documentation is available in the `syzkaller` repository for setting up a FreeBSD host to run `syzkaller` (which performs fuzzing using QEMU or bhyve VMs).

An alternative approach which automates many of the setup steps makes use of Bastille templates. Bastille is a system for deploying and managing jails on FreeBSD; Bastille templates allow one to run code and modify configuration in a running jail. A Bastille template for running `syzkaller` is available. To use it, first install Bastille and create a thin, VNET-based jail based on FreeBSD 13.0:

```
# pkg install bastille
# bastille bootstrap 13.0-RELEASE
# bastille bootstrap https://github.com/markjdb/bastille-syzkaller
# bastille create -V syzkaller 13.0-RELEASE 0.0.0.0 epair0b
```

This assumes that `epair0b` is Bastille's "uplink" interface; it can be bridged with a host interface to provide full network access.

Then, follow the setup steps documented in the bastille-syzkaller template's README to create a ZFS dataset for `syzkaller` and enable the required capabilities in the syzkaller jail. Finally, the template can be applied, and `syzkaller` started, with:

```
# bastille template syzkaller markjdb/bastille-syzkaller \
  --arg FREEBSD_HOST_SRC_PATH=${SRCDIR}
# bastille service syzkaller syz-manager onestart
```

Typically `${SRCDIR}` would be a git worktree referencing the branch to be built and tested; this worktree is `nullfs`-mounted into the jail. The kernel fuzzed by `syzkaller` can be rebuilt using the `build.sh` script installed by the template in the jail root user's home directory:

```
# bastille console syzkaller
# service syz-manager onestop
# sh /root/build.sh
# service syz-manager onestart
```

At this point, `syzkaller` starts a web server listening on port 8080, displaying crash reports and the progress of the fuzzing processes.

MARK JOHNSTON has been a FreeBSD user since 2010 and a committer since 2013. He currently works for the FreeBSD Foundation, where he spends time on improving the stability and performance of the kernel, and on providing code reviews.