



®

FreeBSD[®] **JOURNAL**

March/April 2024

Development Workflow and CI

FreeBSD Kernel
Development Workflow

KDE CI and FreeBSD

More Modern Kernel
Debugging Tools

ZFS Images from Scratch,
or `makefs -t zfs`



FreeBSD[®] JOURNAL

The FreeBSD Journal is Now Free!

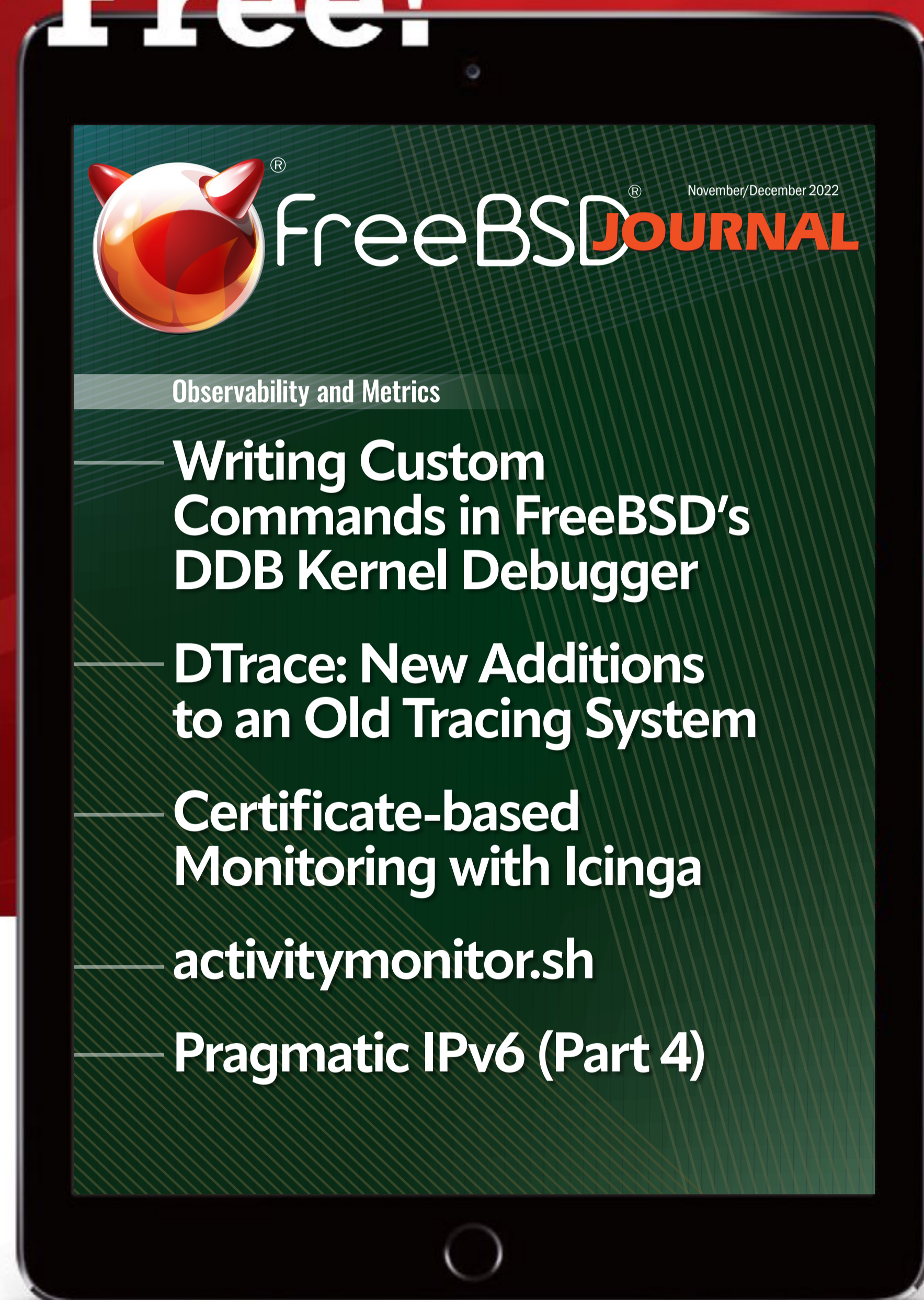
Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!

2024 Editorial Calendar

- Networking
(January-February)
- Development Workflow and CI (March-April)
- Configuration Management Showdown
(May-June)
- Storage and File Systems (July-August)
- To come (September-October)
- To come (November-December)



Find out more at: freebsd.foundation/journal

Editorial Board

John Baldwin • Member of the FreeBSD Core Team and Chair of FreeBSD Journal Editorial Board

Tom Jones • FreeBSD Developer, Internet Engineer and Researcher at the University of Aberdeen

Ed Maste • Senior Director of Technology, FreeBSD Foundation and Member of the FreeBSD Core Team

Benedict Reuschling • FreeBSD Documentation Committer and Member of the FreeBSD Core Team

Mariusz Zaborski • FreeBSD Developer

Advisory Board

Anne Dickison • Marketing Director, FreeBSD Foundation

Justin Gibbs • Founder of the FreeBSD Foundation, President and Treasurer of the FreeBSD Foundation Board

Allan Jude • CTO at Klara Inc., the global FreeBSD Professional Services and Support company

Dru Lavigne • Author of *BSD Hacks* and *The Best of FreeBSD Basics*

Michael W Lucas • Author of more than 40 books including *Absolute FreeBSD*, the *FreeBSD Mastery* series, and *git commit murder*

Kirk McKusick • Lead author of *The Design and Implementation* book series

George Neville-Neil • Past President of the FreeBSD Foundation Board, and co-author of *The Design and Implementation of the FreeBSD Operating System*

Hiroki Sato • Director of the FreeBSD Foundation Board, Chair of AsiaBSDCon, and Assistant Professor at Tokyo Institute of Technology

Robert N. M. Watson • Director of the FreeBSD Foundation Board, Founder of the TrustedBSD Project, and University Senior Lecturer at the University of Cambridge

S&W PUBLISHING LLC

PO BOX 3757 CHAPEL HILL, NC 27515-3757

Editor-at-Large • James Maurer
maurer.jim@gmail.com

Design & Production • Reuter & Associates

FreeBSD Journal (ISBN: 978-0-61 5-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-51 42 • fax: 720/222-2350
email: info@freebsd.foundation.org

Copyright © 2024 by FreeBSD Foundation. All rights reserved.
This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER from the Foundation

Welcome to the March/April issue of the *FreeBSD Journal*! This issue is all about development, whether of FreeBSD itself or using FreeBSD as a platform for developing other software. We open with a practical guide to FreeBSD kernel development from Navdeep Parhar. Navdeep describes a flexible and robust setup featuring virtual machines and the use of PCI passthrough for device driver development. Next, Ben Cooksley pulls back the curtain on KDE's CI system outlining how the KDE project tests changes on FreeBSD. Tom Jones covers kernel debugging support in LLVM's debugger (lldb). Specifically, Tom provides an example of extending lldb with new commands written in lua. Finally, Mark Johnston narrates the story of ZFS support in the makefs tool. While this is a useful feature for both users and developers, Mark used FreeBSD's unique development environment to implement it.

One of the best places to swap development stories is at conferences whether in the hallway track between talks or over evening dinners. Olivier Certner provides a detailed glimpse into EuroBSDCon 2023 held at Coimbra, Portugal. A full slate of BSD conferences are planned this year including AsiaBSDCon (just completed), BSDCan May/June in Ottawa, Canada, and EuroBSDCon in September in Dublin, Ireland.

As always, we love to hear from readers. If you have feedback on any of our articles, suggestions for topics for a future article, or are interested in writing an article, please email us at info@freebsdjournal.com.

John Baldwin

Chair of the *FreeBSD Journal* Editorial Board



Development Workflow and CI

8 FreeBSD Kernel Development Workflow

By Navdeep Parhar

21 KDE CI and FreeBSD

By Ben Cooksley

26 More Modern Kernel Debugging Tools

By Tom Jones

38 ZFS Images from Scratch, or makefs -t zfs

By Mark Johnston

3 Foundation Letter

By John Baldwin

5 We Get Letters

by Michael W. Lucas

46 Practical Ports: Enhance Your Git Experience

By Benedict Reuschling

54 Conference Report: EuroBSDCon 2023

By Olivier Certner

62 Events Calendar

By Anne Dickison

WeGetletters

by Michael W Lucas



Oh, bloviating BSD-er,

I work for a wonderful small company with smart, kind leaders that let the IT group do its job. We've built all our infrastructure according to carefully designed plans that scale to meet our needs. Our network is meticulously segmented to isolate risky services from vital data, our servers are automatically patched, and we spend most of our time smoothing our tiny kinks. I just learned that we're being bought by a multinational corporation that's constantly in the news for security breaches and often mentioned as a place where competent people used to work. Is there any way that we can save what we've built?

—Worried and Raging

Dear WAR,

While "no" is sufficient answer to your question, the Journal editors insist that I respond in more depth so that they're not left with blank pages. I don't understand why they don't simply cover that space with advertising, especially as I was not officially informed that the sales department is on a week-long gelato cruise that I was not invited to, but I suppose amateurs and hobbyists have a right to develop their meager skills without my presence highlighting their inferiority. (The trick is to eat through the dairy coma until your pancreas transcurses its fleshy limits and understanding that water breaks are not only for cleansing the palate. If your undisciplined palate can still differentiate flavors after the day's third hogshead, that is.)

Your problem distills to finances. Once you involve business, everything distills to finances. Those cozy leaders you worked for? Their kindness was either a ploy or weakness. Building a small company into something profitable enough to sell for a small fortune means attracting skilled people, and kindness is the bait—especially when businesses define "kindness" as "torture them less." If they're honestly kind, well, that's pure weakness and financiers can sniff out weakness like trash pandas honing in on yesterday's tuna salad, with similar results. Sure, the new owners might talk about "good will" and promise that nobody will lose their job but that job is already lost, transparently replaced by some churning monstrosity that constricts an inch every day. Take a deep breath now. That air has to last you the rest of this job.

The real problem is that you care about your work. Designing and deploying systems that work well proves you care, when a thick layer of impact-absorbent apathy solves most problems encountered at work. If you must care, though, the easy solution is to ignore directives coming from the new owners. If they're incompetent, they won't notice. Maybe they tell you to install a few servers running infamously insecure operating systems. Your network is well-segmented, so put them somewhere that the inevitable breaches will have no impact on the important work. Maintain and use your existing services until the new owners can provide equivalent replacements, which will arrive on Saint Never's Day.

This presumes that something in your infrastructure is worth saving. —Is it?

Perhaps you have extensive monitoring and log analysis, all meticulously tuned to inform you of every little wobble. You can identify the host spewing stray packets with a single netflow query and know how many times a second a hopeful spambot flings garbage at `xmlrpc.php`. Your mail server sneers at spam. You've even taught fail2ban manners *without* resorting to a spiked club. You have all this, right? Or do you merely have plans for all these? Plans offer the greatest gift, which is Hope, but hope and a good swift kick to the teeth will get you a minuscule stash of legal narcotics and a substantial dentist bill. Are you protecting the dream or the reality? Dreams can be moved. Whatever you're planning to do can be planned just as well elsewhere, and always remember Rule of System Administration #15: *Today's plans address yesterday's failures*. Failure is a renewable resource, granting you endless opportunity to brew new plans.

Designing and deploying systems that work well proves you care.

Or perhaps the new owners are one of those giant tech firms whose major product is buyouts. They have a team dedicated to managing vermin like you. The day the buyout is announced two tech goons arrive—special goons chosen for their innate ability to ignore your worth, wearing camouflaging Unix conference T-shirts and conversant with the language of competence but carrying a Windows server and a router with the console port pins snipped off for the corporate dark fiber being installed tomorrow. They might even buy pizza as a gesture of friendship and cooperation. Eat the pizza and show your teeth. The goons will mistake it for a smile.

Hope might be the greatest of gifts, but it is also the most treacherous. The company's new owners will let you hope things will stay the same. When everything shifts, they'll let you hope for improvement, relying on your hope to keep you in place as they extract everything worthwhile from their new asset. Buyouts, like blackmail, work on hope. Logic declares that the only possible end of a rousing game of blackmail is the death of a participant and that if you didn't start the game, your first move is choosing between victory and victimhood—again, like buyouts.

Steel your soul, and immediately contact everyone who owes you a favor. You need a new employer before those goons return with wire cutters to snip the power cables on all your existing servers.

The good news is that predatory financiers can buy your company, but they can't buy the things that make your company profitable. They have the company contracts, but they don't have the relationships and expertise that made you successful. When you overcome that hideous hope and depart for a firm that sucks less, you take that with you.

And that's what you save. The connections with your coworkers and customers. What's in your head. Configuring that perfect computing environment will go much more quickly next time, except it won't be perfectly adapted to your new employer and you'll need fresh plans.

This time, you might even implement them. Probably not. But you'll hope, and that's a gift.

Have a question for Michael?
Send it to letters@freebsdjournal.org



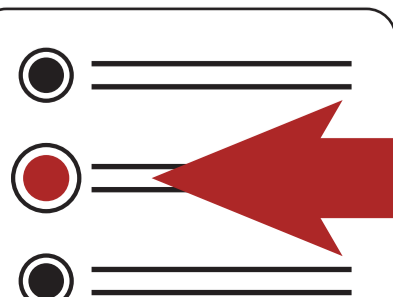
MICHAEL W LUCAS is the author of a bunch of BSD-related books and articles. After years of requests, he finally put up a complete bibliography at <https://mwl.io> so that people have a better chance of avoiding his work. He also owes the esteemed Terry Pratchett a sincere and heartfelt apology.

Books that will help you. Or not.

“While we appreciate Mr Lucas’ unique contributions to the Journal, we do feel his specific talents are not being fully utilized. Please buy his books, his hours, autographed photos, whatever, so that he is otherwise engaged.”

— John Baldwin
FreeBSD Journal Editorial Board Chair

<https://mwl.io>



FreeBSD Kernel Development Workflow

BY NAVDEEP PARHAR

The kernel is like any other software and is developed with the typical workflow of clone, edit, build, test, debug, commit. But unlike userspace software, kernel development necessarily involves repeated reboots (and lockups, and panics) and is inconvenient without a dedicated test system. This article describes a practical setup for kernel development that uses virtual machines for testing and debugging.

A VM based setup for kernel development has a number of advantages:

- Isolation. The host system is not affected by a VM reboot or crash.
- Speed. A VM reboots much faster than a bare metal system.
- Debuggability. VMs are easy to setup for live source-level kernel debugging.
- Flexibility. VMs can be used to build a “network in a box” for working on networking code without an actual physical network. eg. on a laptop in an airplane.
- Manageability. VMs are easy to create, reconfigure, clone, and move.

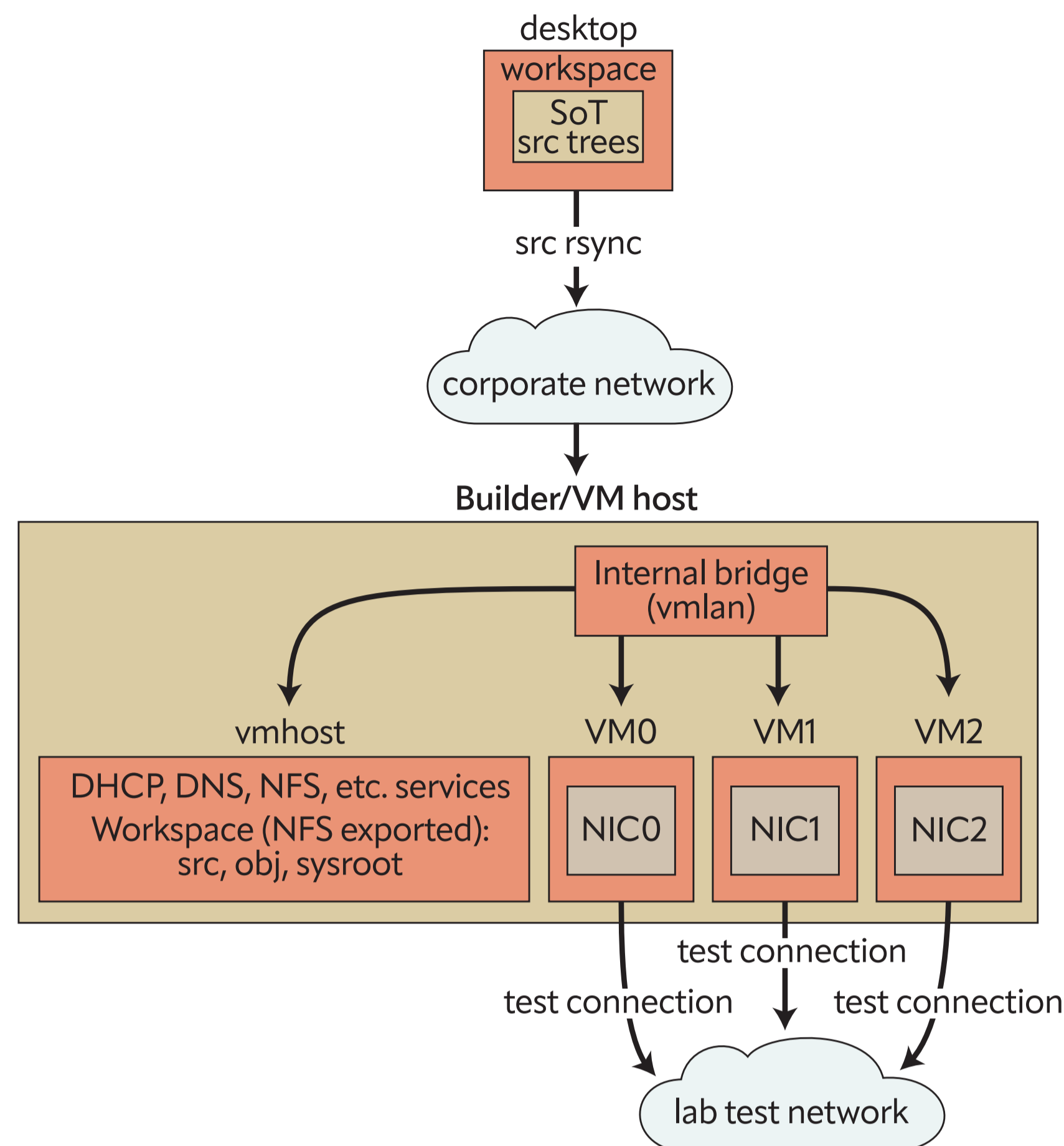
Overview

The system used to build the kernel also runs test VMs, all of which are connected to an internal bridge. The host provides DHCP, DNS, and other services to the IP network configured on the bridge. The source tree and build artifacts are all within a self-contained work area on the host which is exported to this internal network. The work area is mounted inside the test VM and a new test kernel is installed from there. The VMs have an extra serial port configured for remote kernel debugging, with the gdb stub in the VM’s kernel connected to the gdb client on the host system over a virtual null modem cable.

The simplest way to set it up is to use a single system, typically a laptop or workstation, for everything. This is the most portable development environment of all but the system must have enough resources (CPU, memory, and storage) to build a FreeBSD source tree and to run the VMs.

In work environments it is more common to have dedicated servers in the lab for development and testing, separate from the developer’s workstation. Server class systems have higher specs than desktops and are better suited for building source code and running VMs. They also offer a wider variety of PCIe expansion slots and are suited for PCIe device driver development.

Unlike userspace software, kernel development necessarily involves repeated reboots and is inconvenient without a dedicated test system.



Configuration

The rest of this article assumes a two system setup and uses the hostnames 'desktop' and 'builder' to refer to them. The primary copy of the source code is on the desktop, where it is edited by the user and then synced to the builder. The rest takes place on the builder, as root. The builder is also known as 'vmhost' on its internal network.

```

WS=dev
DWSDIR=~ /work/ws
WSDIR=/ws

```

All the checked out trees on the desktop are assumed to be in their own directory `${WS}` in a common parent directory `${DWSDIR}`. The examples use the workspace 'dev' in the '~user/work/ws' directory. The `${WSDIR}` directory on the builder is the self-contained work area with build configuration files, source trees, a shared obj directory, and a sysroot for gdb. The examples use the location '/ws' on the builder.

Desktop Setup

Source Tree

The FreeBSD source code is available in a git repository at <https://git.FreeBSD.org/src.git>. New development takes place in the main branch and changes applicable to recent stable branches are merged back from main after a soak-in period.

Create a local working copy by cloning a branch from the official repository or a mirror.

```

desktop# pkg install git

```

```

desktop$ git ls-remote https://git.freebsd.org/src.git heads/main heads/stable/*

```

```

desktop$ git clone --single-branch -b main ${REPO} ${DWSDIR}/${WS}

```

```

desktop$ git clone --single-branch -b main https://git.freebsd.org/src.git ~/work/ws/dev

```

Custom KERNCONF for Development

Every kernel is built from a plain text kernel configuration (KERNCONF) file. Traditionally a kernel's ident string (the output of 'uname -i') matches the name of its configuration file. eg. the GENERIC kernel is built from a file named GENERIC. There are a number of KERNCONF options for debugging and diagnostics and it is useful to have them enabled during early development. The GENERIC configuration in the main branch already has a reasonable subset enabled and is suitable for development work. However, modern compilers seem to optimize away variables and other debug information aggressively, even at low optimization levels, and it is sometimes useful to build a kernel with all optimizations disabled. Use the custom KERNCONF shown here, called DEBUG0, for this purpose. It is almost always simpler to include an existing configuration and use nooptions/options and nomakeoptions/makeoptions to make adjustments, instead of writing one from scratch.

The DEBUG makeoptions is added to the compiler flags for both the kernel and the modules. The stack size needs to be increased to accommodate the larger stack footprint of unoptimized code.

```
desktop$ cat ${DWSDIR}/${WS}/sys/amd64/conf/DEBUG0
desktop$ cat ~/work/ws/dev/sys/amd64/conf/DEBUG0
include GENERIC
ident DEBUG0
nomakeoptions DEBUG
makeoptions DEBUG="-g -O0"
options KSTACK_PAGES=16
```

Getting the Source Tree to the Builder

Copy the source tree to the builder, where it will be built as root. Remember to synchronize the contents after making changes on the desktop but before building on the builder.

```
desktop# pkg install rsync

desktop$ rsync -az0 --del --no-o --no-g ${DWSDIR}/${DWS} root@builder:${DWSDIR}/src/
desktop$ rsync -az0 --del --no-o --no-g ~/work/ws/dev root@builder:/ws/src/
```

Builder Setup

Build Configuration

Create make.conf and src.conf files in the workspace area on the builder instead of modifying the global configuration files in /etc. The obj directory is also in the workspace area and not /usr/obj. Use meta mode for fast incremental rebuilds. Meta mode requires filemon. All the kernels in the KERNCONF= list will be built by default and the first one will be installed by default. It is always possible to provide a KERNCONF on the command line and override the default.

```
builder# kldload -n filemon
builder# sysrc kld_list+="filemon"

builder# mkdir -p $WSDIR/src $WSDIR/obj $WSDIR/sysroot
```



```
builder# mkdir -p /ws/src /ws/obj /ws/sysroot
```

```
builder# cat $WSDIR/src/src-env.conf
builder# cat /ws/src/src-env.conf
MAKEOBJDIRPREFIX?=/ws/obj
WITH_META_MODE="YES"
```

```
builder# cat /ws/src/make.conf
KERNCONF=DEBUGO GENERIC-NODEBUG
INSTKERNNAME?=dev
```

```
builder# cat /ws/src/src.conf
WITHOUT_REPRODUCIBLE_BUILD="YES"
```

Networking Configuration

Identify an unused network/mask for use as the internal network. The examples use 192.168.200.0/24. The first host (192.168.200.1) is always the VM host (the builder). Host numbers with two digits are reserved for known VMs. Host numbers with three digits 100 are handed out by the DHCP server to unknown VMs.

1. Create a bridge interface for use as a virtual switch that connects all the VMs and the host. Assign a fixed IP address and hostname to the bridge.

```
builder# echo '192.168.200.1 vmhost' >> /etc/hosts
builder# sysrc cloned_interfaces="bridge0"
builder# sysrc ifconfig_bridge0="inet vmhost/24 up"
builder# service netif start bridge0
```

2. Configure the host to perform IP forwarding and NAT for its VMs. This is strictly optional and should be done if and only if the VMs need access to the external network. The public interface is igb1 in the example here.

```
builder# cat /etc/pf.conf
ext_if="igb1"
int_if="bridge0"
set skip on lo
scrub in
nat on $ext_if inet from !($ext_if) -> ($ext_if)
pass out
builder# sysrc pf_enable="YES"
builder# sysrc gateway_enable="YES"
```

3. Start ntpd on the host. The DHCP server will offer itself as an ntp server to the VMs.

```
builder# sysrc ntpd_enable="YES"
builder# service ntpd start
```


4. DHCP and DNS.

Install dnsmasq and configure it as a DHCP and DNS server for the internal network.

```
builder# pkg install dnsmasq
builder# cat /usr/local/etc/dnsmasq.conf
no-poll
interface=bridge0
domain=vmlan,192.168.200.0/24,local
host-record=vmhost,vmhost.vmlan,192.168.200.1
synth-domain=vmlan,192.168.200.100,192.168.200.199,anon-vm*
dhcp-range=192.168.200.100,192.168.200.199,255.255.255.0
dhcp-option=option:domain-search,vmlan
dhcp-option=option:ntp-server,192.168.200.1
dhcp-hostsfile=/ws/vm-dhcp.conf
```

Add it as the first nameserver in the local resolv.conf. The dnsmasq resolver will service queries from the internal network and the builder's loopback interface only.

```
builder# sysrc dnsmasq_enable="YES"
builder# service dnsmasq start
builder# head /etc/resolv.conf
search corp-net.example.com
nameserver 127.0.0.1
...
```

5. Export the entire work area to the internal network.

```
builder# cat /etc/exports
V4: /ws
/ws -ro -mapall=root -network 192.168.200.0/24
builder# sysrc nfs_server_enable="YES"
builder# sysrc nfsv4_server_only="YES"
builder# service nfsd start
```

vm-bhyve (bhyve Frontend)

vm-bhyve is an easy to use frontend for bhyve.

Identify a ZFS pool for use with the VMs and create a dataset for vm-bhyve on the pool. Specify the name of this pool and dataset in vm_dir in rc.conf. Initialize vm-bhyve once vm_dir is set properly.

```
builder# kldload -n vmm
builder# kldload -n nmdm
builder# sysrc kld_list+="vmm nmdm"
builder# pkg install vm-bhyve
builder# zfs create rpool/vm
builder# sysrc vm_dir="zfs:rpool/vm"
builder# vm init
```



```
builder# sysrc vm_enable="YES"
builder# service vm start
```

All the VMs will use a serial console in text mode, accessible using tmux.

```
builder# pkg install tmux
builder# vm set console=tmux
```

Add the previously created bridge interface as a vm-bhyve switch.

```
builder# vm switch create -t manual -b bridge0 vmlan
```

Establish reasonable defaults for new VMs. Edit the default template at `$vm_dir/templates/default.conf` as needed. Specify at least 2 serial ports—one for the serial console and one for remote debugging. Connect all new VMs to the vmlan switch.

```
builder# vim /rpool/vm/.templates/default.conf
loader="uefi"
cpu=2
memory=2G
comports="com1 com2"
network0_type="virtio-net"
network0_switch="vmlan"
disk0_size="20G"
disk0_type="virtio-blk"
disk0_name="disk0.img"
```

Seed Images

The easiest way to have FreeBSD up and running in a new VM is to seed it with a disk image that has it preinstalled. The VM will boot its default kernel or a dev kernel and its user-space needs to work with both so it is best to use the same version of FreeBSD in the VM as the dev tree.

Disk images for releases and for recent snapshots of the main and stable branches are available from [FreeBSD.org](https://www.freebsd.org).

```
# fetch https://download.freebsd.org/releases/VM-IMAGES/14.0-RELEASE/amd64/Latest/FreeBSD-14.0-RELEASE-amd64.raw.xz
# fetch https://download.freebsd.org/snapshots/VM-IMAGES/15.0-CURRENT/amd64/Latest/FreeBSD-15.0-CURRENT-amd64.raw.xz

# unxz -c FreeBSD-14.0-RELEASE-amd64.raw.xz > seed-14_0.img
# unxz -c FreeBSD-15.0-CURRENT-amd64.raw.xz > seed-main.img
# du -Ash seed-main.img; du -sh seed-main.img
6.0G    seed-main.img
1.6G    seed-main.img
```

Disk images can also be generated from a source tree. This example shows how to build an image with a non-debug kernel and some other space-saving options.

```
# cd /usr/src
```



```
# make -j1C KERNCONF=GENERIC-NODEBUG buildworld buildkernel
# make -j1C -C release WITH_VMIMAGES=1 clean obj
# make -j1C -C release WITHOUT_KERNEL_SYMBOLS=1 WITHOUT_DEBUG_FILES=1 \
  NOPORTS=1 NOSRC=1 WITH_VMIMAGES=1 VMFORMATS=raw VMSIZE=4g SWAPSIZE=2g \
  KERNCONF=GENERIC-NODEBUG vm-image

# cp /usr/obj/usr/src/amd64.amd64/release/vm.ufs.raw seed-main.img
# du -Ash seed-main.img; du -sh seed-main.img
6.0G    seed-main.img
626M    seed-main.img
```

Modify the vanilla image for use as a test VM on the internal network.
Create a memory disk from the image and mount the UFS partition. This will be the pre-installed OS's root partition when it boots inside the VM.

```
# mdconfig -af seed-main.img
md0
# gpart show -p md0
# mount /dev/md0p4 /mnt
```

Remove hostname from rc.conf to force the one provided by the DHCP server to be used.

```
# sysrc -R /mnt -x hostname
# sysrc -R /mnt -x ifconfig_DEFAULT
# sysrc -R /mnt ifconfig_vtnet0="SYNCDHCP"
# sysrc -R /mnt ntpd_enable="YES"
# sysrc -R /mnt ntpd_sync_on_start="YES"
# sysrc -R /mnt kld_list+="filemon"
```

Enable ssh access to the VM out of the box. Note that this is a development environment inside a lab network and there are no concerns about operating as root or reusing the same host keys. Copy the host keys and root's .ssh to the correct locations. It is convenient to use the same keys on all the VMs. Update the sshd configuration to allow root to login and enable the service.

```
# cp -a .../vm-ssh-hostkeys/ssh_host_*key* /mnt/etc/ssh/
# cp -a .../vm-root-dotssh /mnt/root/.ssh
# vim /mnt/etc/sshd_config
PermitRootLogin yes
# sysrc -R /mnt sshd_enable="YES"
```

Configure the first serial port as a potential console and the second one for remote kernel debugging.

```
# vim /mnt/boot/loader.conf
kern.msgbuf_show_timestamp="2"
hint.uart.0.flags="0x10"
hint.uart.1.flags="0x80"
```

Create the mount point for the work area and add an entry in fstab to mount it on boot. /dev/fd and /proc are useful in general.

```
# mkdir -p /mnt/ws
# vim /mnt/etc/fstab
...
fdesc  /dev/fd fdescfs rw      0      0
proc   /proc   procfs  rw      0      0
vmhost:/ /ws nfs ro,nfsv4 0 0
```

All done. Unmount and destroy the md.

```
# umount /mnt
# mdconfig -du 0
```

seed-main.img file is ready for use.

New Test VM

Create a new VM and note its auto-generated MAC address. Update the configuration so that the DHCP service provides the assigned hostname and IP address to known VMs. These statically assigned addresses must not overlap with the dhcp-range. The convention in this article is to use 2 digit host numbers for known VMs and 3 digit host numbers for dynamic dhcp-range.

Create a 'dhcp-host' entry with the hostname assigned to the VM, its MAC address, and a fixed IP that is not from the dynamic range. Then reload the resolver.

```
builder# vm create vm0
builder# vm info vm0 | grep fixed-mac-address
builder# echo 'vm0,58:9c:fc:03:40:dc,192.168.200.10' >> /ws/vm-dhcp.conf
builder# service dnsmasq reload
```

Replace the disk0.img file with a copy of the seed image and increase its size to the desired disk size for the VM. A VM's disk image can be resized this way any time the VM is not running. Run "service growfs onestart" in the VM the first time it boots with a resized disk.

```
builder# cp seed-main.img /rpool/vm/vm0/disk0.img
builder# truncate -s 30G /rpool/vm/vm0/disk0.img
```

First Boot

Review the VM's configuration before first boot.

```
builder# vm configure vm0
```

Start the VM with its console in the foreground, or start it in the background and then attach to its console. The console is just a tmux session named after the VM.

```
builder# vm start -i vm0

builder# vm start vm0
builder# vm console vm0
```

Verify the following the first time a VM boots:

- The VM's hostname is the one assigned by the DHCP server. The hostname and tty are visible on the console in the login prompt.

```
FreeBSD/amd64 (vm0) (ttyu0)
login:
```

- The VM's uart0 is the console and uart1 is for remote debugging.

```
vm0# dmesg | grep uart
[1.002244] uart0: console (115200,n,8,1)
...
[1.002252] uart1: debug port (115200,n,8,1)
```

- The work area is mounted at the expected location.

```
vm0# mount | grep nfs
vmhost:/ on /ws (nfs, read-only, nfsv4acls)
vm0# ls /ws
...
```

- The VM is reachable over ssh from the host and from the desktop (using the VM host as the jump host).

```
builder# ssh root@vm0
```

```
desktop$ ssh -J root@builder root@vm0
```

PCIe Device Driver Development in a VM

PCI passthrough allows the host to export (pass through) PCIe devices to a VM, giving it direct access to the PCIe device. This makes it possible to do device driver development for real PCIe hardware inside a VM.

The device is claimed by the ppt driver on the host and appears inside the VM as if connected to the VM's PCIe root complex. The PCIe devices on a system are identified with a BSF (or BDF) 3-tuple and it may be different inside the VM.

Use `pciconf` or `vm-bhye` to get a list of PCIe devices on the system and note the BSF tuple for the ones to pass through. Note that the `pciconf` selector ends with BSF separated by colons whereas `bhyve/vmm/ppt` use B/S/F (separated by forward slash) to identify a device. eg. the PCIe device with the selector "none193@pci0:136:0:4" is "136/0/4" in the `bhyve/ppt` notation.

```
builder# pciconf -ll
builder# vm passthru
```

Have the ppt driver claim the devices that will be passed through. This prevents the normal driver from attaching to the device.

```
builder# vim /boot/loader.conf
pptdevs="136/0/4 137/0/4"
```

Reboot so that the loader.conf changes take effect, or try to detach the device from its driver and attach it to ppt while the system is running.

```
builder# devctl detach pci0:136:0:4
builder# devctl clear driver pci0:136:0:4
builder# devctl set driver pci0:136:0:4 ppt
(repeat for 137)
```

Verify that the ppt driver attached to the devices and vm-bhyve is ready to use them.

```
builder# pciconf -ll | grep ppt
ppt0@pci0:136:0:4:      020000  00  00  1425  640d  1425  0000
ppt1@pci0:137:0:4:      020000  00  00  1425  640d  1425  0000
builder# vm passthru | awk 'NR == 1 || $3 != "No" {print}'
DEVICE      BHYVE ID    READY      DESCRIPTION
ppt0        136/0/4    Yes        T62100-CR Unified Wire Ethernet Controller
ppt1        137/0/4    Yes        T62100-CR Unified Wire Ethernet Controller
```

Reconfigure the test VM and list the devices that should be passed through to that VM.

```
builder# vm configure vm0
passthru0="136/0/4"
passthru1="137/0/4"
```

Start the test VM and verify that the PCIe devices are visible. Note that the BSFs in the VM are different from the actual hardware BSFs in the host.

```
vm0# pciconf -ll
...
none0@pci0:0:6:0:      020000  00  00  1425  640d  1425  0000
none1@pci0:0:7:0:      020000  00  00  1425  640d  1425  0000
...
```

Main Workflow Loop (edit, build, install, test, repeat)

Edit

Edit the source tree on the desktop and sent it to the builder.

```
desktop$ cd ~/work/ws/dev
desktop$ gvim sys/foo/bar.c
...
desktop$ rsync -az0 --del --no-o --no-g ~/work/ws/dev root@builder:/ws/src/
```

Build

```
builder# alias wsmake='__MAKE_CONF=${WSDIR}/src/make.conf SRC_ENV_CONF=${WSDIR}/src/src-
env.conf SRCCONF=${WSDIR}/src/src.conf make -j1C'
```

```
builder# alias wsmake='__MAKE_CONF=/ws/src/make.conf SRC_ENV_CONF=/ws/src/src-env.conf SRC-
CONF=/ws/src/src.conf make -j1C'
```

```
builder# cd ${WSDIR}/src/${WS}
builder# cd /ws/src/dev
builder# wsmake kernel-toolchain (one time)
builder# wsmake buildkernel
```

Install

1. Install kernel in the VM. INSTKERNNAME is set in make.conf so the test kernel in /boot/\${INSTKERNNAME} will not interfere with the stock kernel in /boot/kernel, which is the safe fallback if there are problems with the test kernel. It can be specified explicitly on the command line too.

```
vm0# alias wsmake='__MAKE_CONF=${WSDIR}/src/make.conf SRC_ENV_CONF=${WSDIR}/src/src-
env.conf SRCCONF=${WSDIR}/src/src.conf make -j1C'
vm0# alias wsmake='__MAKE_CONF=/ws/src/make.conf SRC_ENV_CONF=/ws/src/src-env.conf
SRCCONF=/ws/src/src.conf make -j1C'

vm0# cd ${WSDIR}/src/${WS}
vm0# cd /ws/src/dev
vm0# wsmake installkernel
```

2. Install to the builder's sysroot too if gdb on the builder will be used for source level debugging. Use the same INSTKERNNAME and KERNCONF as in the VM.

```
builder# cd /ws/src/dev
builder# wsmake installkernel DESTDIR=/ws/sysroot
```

Test

Select the test kernel for the next reboot only, or permanently.

```
vm0# nextboot -k ${WS}
vm0# nextboot -k dev
vm0# shutdown -r now

vm0# sysrc -f /boot/loader.conf kernel="${WS}"
vm0# sysrc -f /boot/loader.conf kernel="dev"
vm0# shutdown -r now
```

It is a good practice to use a debug KERNCONF (eg. the custom DEBUG0 shown earlier or the GENERIC in main) for initial testing and later switch to a release kernel (eg. the GENERIC-NODEBUG in main).

Debugging the Test Kernel

Verify that the test kernel is running currently.

```
vm0# uname -i
DEBUG0
vm0# sysctl kern.bootfile
kern.bootfile: /boot/dev/kernel
```

Backends

There are two debugger backends available and the current backend can be changed on the fly.

```
vm0# sysctl debug.kdb.available
vm0# sysctl debug.kdb.current

vm0# sysctl debug.kdb.current=ddb
vm0# sysctl debug.kdb.current=gdb
```

Breaking into the Debugger

1. Automatically, on a panic. If this sysctl is set the kernel will enter the debugger (instead of rebooting) on panic.

```
vm0# sysctl debug.debugger_on_panic
```

2. Manually, from inside the VM.

```
vm0# sysctl debug.kdb.enter=1
```

3. Manually, from the VM host. Inject an NMI into the VM if it is locked up and not responding.

```
builder# bhyvectl --vm=vm0 --inject-nmi
```

Source Level Debugging with gdb

Source level debugging requires the source code, binaries, and debug files, all of which are available on both the host and the VMs, but at different locations.

Live Remote Debugging

Make sure that the debug backend is set to gdb. If the VM has already entered the debugger with the ddb backend, switch to the gdb backend interactively.

```
vm0# sysctl debug.kdb.current=gdb

db> gdb
```

The remote gdb stub in the kernel is active when the kernel enters the debugger. Connect to the gdb stub from the host. The connection takes place over a virtual null modem cable connected to the VM's second serial port (uart1 inside the VM).


```
builder# gdb -iex 'set sysroot ${WSDIR}/sysroot' -ex 'target remote /dev/nmdm-${VM}.2B'
${WSDIR}/sysroot/boot/${INSTKERNNAME}/kernel
builder# gdb -iex 'set sysroot /ws/sysroot' -ex 'target remote /dev/nmdm-vm0.2B' /ws/sys-
root/boot/dev/kernel
```

Core Dump Analysis

Same as live debug except the target is a vmcore instead of remote.

```
builder# gdb -iex 'set sysroot ${WSDIR}/sysroot' -ex 'target vmcore ${VMCORE}' ${WSDIR}/
sysroot/boot/${INSTKERNNAME}/kernel

builder# scp root@vm0:/var/crash/vmcore.0 /ws/tmp/
builder# gdb -iex 'set sysroot /ws/sysroot' -ex 'target vmcore /ws/tmp/vmcore.0' /ws/sys-
root/boot/dev/kernel
```

NAVDEEP PARHAR has been a FreeBSD user for 20+ years and a FreeBSD developer since 2009. He is employed by Chelsio Communications to work on FreeBSD software for Chelsio Terminator family of NICs. He's the author and maintainer of the cxgbe(4) driver and his areas of interest include the networking stack, device drivers, general kernel debug and analysis.



The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website

freebsd.org/projects/newbies.html

Downloading the Software

freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at freebsd.foundation.org

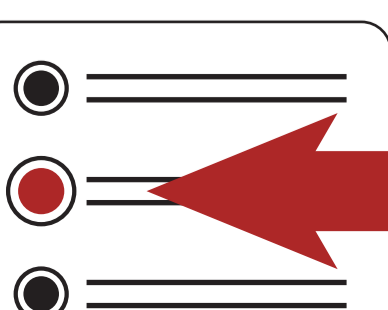
Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by



KDE CI and FreeBSD

BY BEN COOKSLEY

Continuous Integration (CI) is something KDE has worked on improving now for some years, with the first implementation of CI for KDE software starting back in August 2011. Since then, the system has evolved substantially, picking up support not only for multiple versions of Qt (the toolkit used to write most KDE software) but also support for multiple platforms.

Running all these builds reliably and consistently, across the multiple operating systems involved, is something that is only possible thanks to containers, which are increasingly ubiquitous across all platforms. To understand the challenges containers solve, as well as other challenges in building scalable CI systems though, we must go back to the beginning of KDE CI.

When the system first started life, it was a relatively simple Jenkins setup—with builds performed on the same server that hosted Jenkins. This made life quite simple, however it also had limitations. As demand for builds increased with more projects onboarding to the system, it soon became clear that more machines would be needed.

This presented a bit of a conundrum though, as KDE software tends to require other KDE libraries in order to be built, and not just any version either—usually the most recent version. This meant that it wouldn't just be a case of increasing the number of builders, we would also need to ensure that the latest version of dependencies were still available.

Due to the amount of time needed to build the full chain of dependencies for our applications, the concept of just building everything each time was quickly ruled out—meaning it would be necessary to share the binaries resulting from those builds. After a quick review of our options, rsync was quickly selected as our preferred choice, and once again all was well.

Enter FreeBSD

By 2017, the system encountered its next set of growing pains, as it became desirable to start adding support for new platforms, which is where FreeBSD enters the picture for the first time.

This initial implementation of FreeBSD support within our CI system was relatively simple, and made use of virtual machines running on our Linux CI workers. These machines were

The system has evolved substantially, picking up support not only for multiple versions of Qt but also support for multiple platforms.

individually set up with help from the KDE on FreeBSD team, and much like our Linux builds at the time, included everything needed to build KDE software.

This approach however did have its downsides. While we did ensure that all of the builders made use of the same custom FreeBSD repository that had all the necessary dependencies in it to build our software, each of the machines was still built individually. This made scaling the system non-trivial, as any changes had to be applied to each builder one by one.

It did succeed, however, in ensuring that KDE software could be reliably built on FreeBSD, and ensured dependencies were packaged in advance of KDE software starting to make use of them—improving the experience for the KDE on FreeBSD team substantially.

At the same time as we added support for FreeBSD, we also adopted something that was still fairly new at the time for our Linux builds—Docker. For the first time, we were able to produce a single master setup that could be distributed across all our builders, making it easy to roll out changes across the CI system without having to manually apply them to each machine. The golden age of container-based builds had begun its arrival. The only downside was that it was Linux only, so the question remained of how to replicate this on other platforms.

Before we could tackle that though, all of this growth in build capability had resulted in some new, and slightly unexpected problems starting to show up. From time to time builds would randomly fail, with logs indicating that files were missing or symlinks broken. Later checks would show that the files were there, and subsequent runs completed successfully. The problem? Atomicity.

Up to now, we only had a small handful of build nodes, which had certain limitations on their performance. The new setup however had much more capable hardware, and as such was completing builds faster—meaning it was increasingly likely that rsync would be mid-upload when another build went to download build artifacts. This is why we were seeing missing files and broken symlinks in some builds as the unfortunate build happened to start at the same time one of its dependencies happened to be syncing build results.

Thankfully, the answer once again was fairly simple—moving to using tarballs of build artifacts. This let us publish the full set of files from a build in one fluid atomic operation, and in addition to a change of protocol over to SFTP (to accommodate platforms without rsync) meant that the CI system was once again operating smoothly—with quite a bit more resource power, and supporting quite a few more platforms.

The issue of having to maintain machines individually by hand, however, did not go away. The migration to Gitlab and Gitlab CI made this issue more apparent than ever, as build nodes began to run out of disk space due to accumulated code checkouts and other build artifacts would quickly fill disk space. We also battled problems with leftover processes from tests chewing up CPU time (sometimes entire cores even)—all things that did not exist at all on our Docker based Linux builds.

Many options and solutions were discussed on this, including improvements to Gitlab Runner and how it handles the “shell” executor, cronjobs to perform cleanup of build arti-

We also adopted something that was still fairly new at the time for our Linux builds—Docker.

facts and code checkouts, as well as building something on top of FreeBSD Jails. None of these, however, would replicate the same experience we had on Linux with Docker.

Finding Podman

So it was one morning while looking at FreeBSD containerization options that we stumbled across Podman and its companion ocijail support. This promised us all the things we were used to enjoying with our Docker-based Linux setups, but on FreeBSD.

Significantly, it would mean that the issues we had been experiencing with stray processes and leftover build artifacts that we had to clean up manually would be solved. And it would also let us make use of a standard Open Container Initiative registry (such as Gitlab's built-in Container Registry) to distribute images of FreeBSD to all our builders—solving our issue of having to maintain the machines individually.

Getting a working image built would be our first challenge. For Linux systems, Docker and Podman are quite well established with detailed documentation on the available base images and what those base images contain. With the appropriate FreeBSD base image found though, we thought it would be a simple case of adding our normal FreeBSD package repository and installing everything we would normally need.

We would soon find our first bump on the road, as, unexpectedly, CMake indicated on our first build run in a container that it was unable to find a compiler. We thought this quite odd, as usually FreeBSD systems ship with a compiler installed. After a bit of digging, we found the first major difference between FreeBSD containers and a normal FreeBSD system—namely, that they are significantly stripped down, and therefore don't include a compiler.

After a couple of iterations, we ended up adding in a compiler and C library development headers—which allowed our very first piece of KDE software to be built in a FreeBSD container. Thinking we now had everything sorted, we pressed ahead—only for subsequent bits of KDE software to fail as additional development packages were needed. Many iterations later (including installing a bunch more development and non-development FreeBSD-* packages) we were finally presented with a completed build for a number of key KDE packages.

With this sorted, attention now turned to what Gitlab Runner calls a “helper image”, which is something it uses to perform Git operations and upload artifacts from builds to Gitlab itself among other things. While we could have made use of FreeBSD's support to run Linux binaries, that would have been an imperfect solution. So we naturally set out to build this natively for FreeBSD as well. After replicating what had been done by Gitlab themselves to build the images, but in FreeBSD, we soon had what we thought would be the final piece ready.

It was now that the fun part of the adventure began, and a deep dive into the inner workings of Gitlab Runner and Podman began. The first hurdle thrown was moments after we connected Gitlab Runner to Podman (using it's Docker compatibility option), when our first build was met with the message of “unsupported os type: freebsd”.

A quick search of the Gitlab Runner codebase revealed that for Docker it checked the operating system of the remote Docker (or in our case: Podman) host. A quick patch and re-

Getting a working image built would be our first challenge.

build of Gitlab Runner later, and we had a very similar, but not quite identical error: “unsupported OSType: freebsd”. More patching to Gitlab Runner followed only for a third much more ominous error to be returned, especially given Gitlab Runner is written in Go:

```
ERROR: Job failed (system failure): prepare environment:
Error response from daemon: runtime error: invalid memory address or nil pointer
dereference(docker.go:624:0s.
Check https://docs.gitlab.com/runner/shells/index.html#shell-profile-loading for more
information.
```

With this it became apparent that much more work would be required to get this working, however, given the promise of what containerized builds could deliver, we persisted and started looking into where this was failing. After some research through the Gitlab Runner codebase, we found code that didn't seem to be doing anything too special:

```
inspect, err := e.client.ContainerInspect(e.Context, resp.ID)
```

And so began many hours of debugging, searching for why this one line of code failed on FreeBSD yet worked absolutely fine on Linux (regardless of whether it was Podman or Docker). Eventually, however, we stumbled on the cause: the Podman daemon itself was falling over and abandoning the request. With this information in hand, the issue was soon easily reproduced by trying to run “podman inspect” against a running container, resulting in the expected crash we wanted to see. Success!

Having searched through Gitlab Runner, focus now turned to Podman itself. Before long the cause had been narrowed down to code that was specifically called for “inspect” operations, and soon after that a specific line was identified that tried to interact with Linux specific constructs no matter the platform. Yet, another patch later, we had a “podman inspect” that did not crash, and then shortly after that, our first FreeBSD build starting successfully.

Running Builds on FreeBSD

That first build may have failed (due to known issues with Git and the way Gitlab Runner interacts with containers that run as builds as users other than root), but the important thing was we had running builds on FreeBSD.

At this point, you may have thought we were home free and could begin to roll out FreeBSD based containerized builds to all KDE projects. Final testing, however, revealed one final issue: that network speeds in our FreeBSD containers appeared to be quite a bit slower than we had expected, being significantly slower than what the FreeBSD hosts were capable of.

Thankfully, this was not a new issue, was something that others had run into before, and was something we had anticipated we would encounter. This particular issue has been well written up by Tara Stella in the past, in their experiences diving into the world of Podman and FreeBSD containers, and is caused by Large Receive Offload or LRO. One quick config-

It became apparent that much more work would be required to get this working.

uration change later and we had the performance we expected—and were finally ready to go live.

Today, KDE runs FreeBSD CI builds using Podman and ocijail-based containers exclusively, with 5 FreeBSD host systems handling the build requests. These builds are performed using two different CI images—one for each of the two supported versions of Qt (being Qt 5 and Qt 6) ensuring that KDE software can be cleanly built from scratch and, optionally, has fully passing unit test results.

Since migrating over from FreeBSD dedicated virtual machines to FreeBSD containerized builds, we have gone from receiving complaints from developers due to broken builders and having to undertake maintenance on our builders several times a week (and sometimes even daily), to receiving no complaints in several weeks and only needing to undertake periodic maintenance.

The patches that we wrote (just a couple of lines for both Podman and Gitlab Runner) have been successfully upstreamed and should now be available for all to use and enjoy in building out their own CI setups.

The benefits of switching to containers—especially for Continuous Integration systems—cannot be understated and are something any team that maintains a system should consider investigating, as the returns are well worth the initial cost of migration.

BEN COOKSLEY is an accountant and also a computer scientist known for his contributions to the KDE community, particularly in system administration and infrastructure. His interest in sysadmin work stems from a curiosity about how systems operate and integrate.

Write For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)



More Modern Kernel Debugging Tools

BY TOM JONES

Panic is a truly wonderful word! It succinctly describes an incredibly complex emotional event. We can say “the soldiers panicked,” and we know how a battle went. We can use it to give weight to a small oversight, and it explains what exactly went through our mind when we stepped onto a plane and discovered doubt about the current state of the oven.

Surely I turned it off.

It might just be my favourite term to see on the cover of a book: “*Panic! Unix System Crash Dump Analysis*”—wow! I’ve gotta have that. Any author using a title like that will have written a great book, even if it is an old Sun OS/Solaris technical manual.

Great book titles are more timeless than technical treatises, and the first material to expire is vendor-level documentation for existing systems. In 2024, I find it remarkably hard to find anything written recently about debugging approaches for operating systems. Looking at the works published, you wouldn’t be blamed if you thought we’d perfected operating systems in 2004. I haven’t ever used either SunOS or Solaris, but “*Panic!*” gave me the introduction to crash analysis I have always wanted.

I will admit I’d always wondered why people wanted core dumps so badly—what more can we really get than from a stack trace I would wonder. But from “*Panic!*” I was able to take my first real steps into practically doing crash analysis, and it took me on a journey that tried bleeding edge kernel debugging tools on FreeBSD. Let me show you what I learned. Don’t worry, we won’t have to wait around for lemon-soaked, paper napkins.

Getting a Kernel Dump

I’m going to be up front, I know you will not try to do kernel dump analysis until you really need to. When you have a hung machine. On that path lies a life of printf debugging.

Getting a core to play with isn’t hard—your system needs to be set up to take crash dumps (see `dumpon(8)`) and then to enter the debugger. Normally, the system will help you by panicking, assisting you in your journey to reach the debugger. FreeBSD helpfully offers the ability to panic a kernel even when nothing has gone wrong yet. Setting `debug.kdb.panic sysctl` to 1 on a test system will drop you to a debug prompt:

```
# sysctl debug.kdb.panic=1
```

If you ran that on your desktop or a vital work machine in the cloud, you might be in a bit of trouble (and if it was your desktop, this article might have vanished). I would recommend



learning about kernel debugging on a virtual machine, or at least something that won't cause too much trouble continually crashing.

Additionally, FreeBSD virtual machine images come configured by default to run **savecore** on boot and save out your crash dump file.

Once you set the `debug.kdb.panic`, you will be dropped to a `ddb(4)` prompt. `ddb` is a full fledged live system debugger—it can be a great analysis tool, but it isn't what we want today.

From `ddb` we can dump the running kernel with the `dump` command.

```
ddb> dump
Dumping 925 out of 16047 MB:..2%..11%..21%..32%..42%..51%..61%..71%..82%..92%
```

Panicking makes the system unusable, so you need to reboot to continue.

```
ddb> reboot
```

As your VM comes back up, there will be a message from **savecore** about extracting and saving your core file.

The core will be placed in `/var/crash` along with some other files.

```
$ ls /var/crash
bounds          core.txt.0      info.0          info.last       minfree
vmcore.0       vmcore.last
```

The core file from our test is `vmcore.0`, and it comes with matching `info.0` and `core.txt.0`. The info file is a summary of the host and dump, and the `core.txt` is a summary of the dump file, any unread portions of the message buffer, and the panic string and stack trace if there is one.

```
Dump header from device: /dev/nvd0p3
Architecture: amd64
Architecture Version: 2
Dump Length: 970199040
Blocksize: 512
Compression: none
Dumptime: 2023-05-17 14:07:58 +0100
Hostname: displacementactivity
Magic: FreeBSD Kernel Dump
Version String: FreeBSD 14.0-CURRENT #2 main-n261806-d3a49f62a284: Mon Mar 27 16:15:25
UTC 2023
  tj@displacementactivity:/usr/obj/usr/src/amd64.amd64/sys/GENERIC
Panic String: Duplicate free of 0xfffff80339ef3000 from zone 0xfffffe001ec2ea00(mal-
loc-2048) slab 0xfffff80325789168(0)
Dump Parity: 3958266970
Bounds: 0
Dump Status: good
```

The bounds file lets the dumper know the next coredump will be called `vmcore.1` and right now bounds on this machine:

```
# cat /var/crash/bounds
1
```


Finally, `vmcore.last` is a link to the most recent coredump file, in case you are having an interesting week and have lost track of the most recent crash.

Symbols

The second thing we need to along with the coredump are the kernel symbols. Kernel symbols for releases are available from the `kernel-debug` package and are installed to `/usr/lib/debug/` or can be pulled out of your kernel build directory.

Looking at a Core with gdb (first)

First lets look at a core very quickly with `kgdb` to give ourselves a point of comparison for how far along `lldb` crash dump debugging is.

```
$ kgdb kernel.debug vmcore.0

Unread portion of the kernel message buffer:
panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432
cpuid = 2
time = 1706644478
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe0047d2f480
vpanic() at vpanic+0x132/frame 0xfffffe0047d2f5b0
panic() at panic+0x43/frame 0xfffffe0047d2f610
tcp_discardcb() at tcp_discardcb+0x25b/frame 0xfffffe0047d2f660
tcp_usr_detach() at tcp_usr_detach+0x51/frame 0xfffffe0047d2f680
sorele_locked() at sorele_locked+0xf7/frame 0xfffffe0047d2f6b0
tcp_close() at tcp_close+0x155/frame 0xfffffe0047d2f6e0
rack_check_data_after_close() at rack_check_data_after_close+0x8a/frame 0xfffffe0047d2f720
rack_do_fin_wait_1() at rack_do_fin_wait_1+0x141/frame 0xfffffe0047d2f7a0
rack_do_segment_nounlock() at rack_do_segment_nounlock+0x243b/frame 0xfffffe0047d2f9a0
rack_do_segment() at rack_do_segment+0xda/frame 0xfffffe0047d2fa00
tcp_input_with_port() at tcp_input_with_port+0x1157/frame 0xfffffe0047d2fb50
tcp_input() at tcp_input+0xb/frame 0xfffffe0047d2fb60
ip_input() at ip_input+0x2ab/frame 0xfffffe0047d2fbc0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fc20
ether_demux() at ether_demux+0x17a/frame 0xfffffe0047d2fc50
ether_nh_input() at ether_nh_input+0x39f/frame 0xfffffe0047d2fca0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fd00
ether_input() at ether_input+0xd9/frame 0xfffffe0047d2fd60
vtnet_rxq_eof() at vtnet_rxq_eof+0x73e/frame 0xfffffe0047d2fe20
vtnet_rx_vq_process() at vtnet_rx_vq_process+0x9c/frame 0xfffffe0047d2fe60
ithread_loop() at ithread_loop+0x266/frame 0xfffffe0047d2fef0
fork_exit() at fork_exit+0x82/frame 0xfffffe0047d2ff30
fork_trampoline() at fork_trampoline+0xe/frame 0xfffffe0047d2ff30
--- trap 0, rip = 0, rsp = 0, rbp = 0 ---
KDB: enter: panic

Reading symbols from /boot/kernel/zfs.ko...
```



```

Reading symbols from /usr/lib/debug//boot/kernel/zfs.ko.debug...
Reading symbols from /boot/kernel/tcp_rack.ko...
Reading symbols from /usr/lib/debug//boot/kernel/tcp_rack.ko.debug...
Reading symbols from /boot/kernel/tcphpts.ko...
Reading symbols from /usr/lib/debug//boot/kernel/tcphpts.ko.debug...
__curthread () at /usr/src/sys/amd64/include/pcpu_aux.h:57
57      __asm("movq %%gs:%P1,%0" : "=r" (td) : "n" (offsetof(struct pcpu,
(kgdb)

```

kgdb starts up by showing its license (removed), and then printing out the final parts of the message buffer which is a handy addition from **kgdb**. The final part of the message buffer tells us the panic message, info and a stack trace.

With the **kgdb bt** (backtrace) command, we can get a stack track, and with the frames command, we can move around the stack to see what was happening at the time of the panic.

```

(kgdb) bt
...
#10 0xffffffff80b51233 in vpanic (fmt=0xffffffff811f87ca "Assertion %s failed at %s:%d",
ap=ap@entry=0xfffffe0047d2f5f0) at /usr/src/sys/kern/kern_shutdown.c:953
#11 0xffffffff80b51013 in panic (fmt=0xffffffff81980420 <cnputs_mtx> "\371\023\025\201\377\
377\377\377") at /usr/src/sys/kern/kern_shutdown.c:889
#12 0xffffffff80d5483b in tcp_discardcb (tp=tp@entry=0xfffff80008584a80) at /usr/src/sys/
netinet/tcp_subr.c:2432
#13 0xffffffff80d60f71 in tcp_usr_detach (so=0xfffff800100b6b40) at /usr/src/sys/netinet/
tcp_usrreq.c:215
#14 0xffffffff80c01357 in sofree (so=0xfffff800100b6b40) at /usr/src/sys/kern/uipc_sock-
et.c:1209
#15 sorele_locked (so=so@entry=0xfffff800100b6b40) at /usr/src/sys/kern/uipc_socket.c:1236
#16 0xffffffff80d545b5 in tcp_close (tp=<optimized out>) at /usr/src/sys/netinet/tcp_
subr.c:2539
#17 0xffffffff82e37e0a in tcp_tv_to_usectick (sv=0xfffffe0047d2f698) at /usr/src/sys/neti-
net/tcp_hpts.h:177
#18 tcp_get_usecs (tv=0xfffffe0047d2f698) at /usr/src/sys/netinet/tcp_hpts.h:232
...
(kgdb) frame 12
#12 0xffffffff80d5483b in tcp_discardcb (tp=tp@entry=0xfffff80008584a80) at /usr/src/sys/
netinet/tcp_subr.c:2432
warning: Source file is more recent than executable.
2432
(kgdb) list
2427     #endif
2428
2429         CC_ALGO(tp) = NULL;
2430         if ((m = STAILQ_FIRST(&tp->t_inqueue)) != NULL) {
2431             struct mbuf *prev;
2432

```



```

2433         STAILQ_INIT(&tp->t_inqueue);
2434         STAILQ_FOREACH_FROM_SAFE(m, &tp->t_inqueue, m_stailqpkt, prev)
2435             m_freem(m);
2436     }

```

To review, I have listed the backtrace which led up to the panic, identified the call to panic around frame number **#11**, and asked **kgdb** to move to frame **#12** (the code which led to the panic itself), then listed the code there. Further investigation from here would help us determine whatever led to the panic in this crash dump I had lying around.

These are the basic steps in kernel debugging, looking at what was going on and interrogating the crash dump to find out what values variables held. **lldb** needs to be able to do these tasks, for it to be useful in a kernel context.

lldb

FreeBSD has been moving to the more freely licensed **llvm/clang** toolchain for the last decade. One missing piece for a while has been debugging, but in 2024 there are enough pieces in place that FreeBSD kernel debugging is possible with **lldb**.

lldb is able to import FreeBSD kernel dumps as core files and can move through stack frames.

lldb was developed by Apple. I remember when they changed the default debugger in **gdb** to **lldb** and I suffered severe culture shock. All of the debugging commands I had won from the cryptic documentation-less **gnu** world were gone, replaced with other weird commands.

lldb isn't really meant to be used as a command line interface, rather it is meant to be driven by software via an API. This shows in the verbosity of many commands. Thankfully, **lldb** has grown support for more **gdb**-like commands in its command line, meaning that more of the command interfaces match. Base commands such as printing now have compatible syntax, but many other options are different, and either better or much worse.

Poking Around with lldb

lldb doesn't need special configuration to analyze a kernel dump. Loading a crash dump in **lldb** is the same as **kgdb** just with the arguments swapped around a little:

```
$ lldb --core <corefile> path/to/kernel/symbols
```

For the examples that works out as:

```

$ lldb --core ../gdb/coredump/vmcore.0 ../gdb/coredump/kernel-debug/kernel.debug
(lldb) target create "../gdb/coredump/kernel-debug/kernel.debug" --core "../gdb/coredump/vmcore.0"
Core file '/home/tj/code/scripts/gdb/coredump/vmcore.0' (x86_64) was loaded.
(lldb)

```

That is much quieter than the start up **kgdb**, which is nice, but it is also missing out on some important context from our crash dump. What exactly led to this being dumped?

In 2024 there are enough pieces in place that FreeBSD kernel debugging is possible with **lldb**.

`kgdb` isn't able to perform any magic (if so it would have a 'fix' command to match the 'break' command). All it is doing is looking for well-known symbols in the crash dump and printing them for us on start up.

We can do that ourselves.

First, the panic message in the kernel is stored in the string `panicstr` and is set by `vpanic` (in `kern/kern_shutdown.c`). We can easily extract this from the dump from `lldb`:

```
(lldb) p panicstr
(const char *) 0xffffffff819c1a00 "Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432"
```

This might be enough for someone to start debugging. I also like stack traces which we can get with `bt` in `lldb`:

```
(lldb) bt
* thread #1, name = '(pid 1025) tcplog_dumper'
  * frame #0: 0xffffffff80b83d2a kernel.debug`sched_switch(td=0xfffff800174be740, flags=259) at sched_ule.c:2297:26
    frame #1: 0xffffffff80b5e9e3 kernel.debug`mi_switch(flags=259) at kern_synch.c:546:2
    frame #2: 0xffffffff80bb0dc4 kernel.debug`sleepq_switch(wchan=0xffffffff817e1448, pri=0) at subr_sleepqueue.c:607:2
    frame #3: 0xffffffff80bb11a6 kernel.debug`sleepq_catch_signals(wchan=0xffffffff817e1448, pri=0) at subr_sleepqueue.c:523:3
    frame #4: 0xffffffff80bb0ef9 kernel.debug`sleepq_wait_sig(wchan=<unavailable>, pri=<unavailable>) at subr_sleepqueue.c:670:11
    frame #5: 0xffffffff80b5df3c kernel.debug`_sleep(ident=0xffffffff817e1448, lock=0xffffffff817e1428, priority=256, wmesg="tcplogdev", sbt=0, pr=0, flags=256) at kern_synch.c:219:10
    frame #6: 0xffffffff8091190e kernel.debug`tcp_log_dev_read(dev=<unavailable>, uio=0xfffffe0079b4ada0, flags=0) at tcp_log_dev.c:303:9
    frame #7: 0xffffffff809d99ce kernel.debug`devfs_read_f(fp=0xfffff80012857870, uio=0xfffffe0079b4ada0, cred=<unavailable>, flags=0, td=0xfffff800174be740) at devfs_vnops.c:1413:10
    frame #8: 0xffffffff80bc9bc6 kernel.debug`dofileread [inlined] fo_read(fp=0xfffff80012857870, uio=0xfffffe0079b4ada0, active_cred=<unavailable>, flags=<unavailable>, td=0xfffff800174be740) at file.h:340:10
    frame #9: 0xffffffff80bc9bb4 kernel.debug`dofileread(td=0xfffff800174be740, fd=3, fp=0xfffff80012857870, auio=0xfffffe0079b4ada0, offset=-1, flags=0) at sys_generic.c:365:15
    frame #10: 0xffffffff80bc9712 kernel.debug`sys_read [inlined] kern_readv(td=0xfffff800174be740, fd=3, auio=0xfffffe0079b4ada0) at sys_generic.c:286:10
    frame #11: 0xffffffff80bc96dc kernel.debug`sys_read(td=0xfffff800174be740, uap=<unavailable>) at sys_generic.c:202:10
    frame #12: 0xffffffff810556a3 kernel.debug`amd64_syscall [inlined] syscallenter(td=0xfffff800174be740) at subr_syscall.c:186:11
    frame #13: 0xffffffff81055581 kernel.debug`amd64_syscall(td=0xfffff800174be740, traced=0) at trap.c:1192:2
    frame #14: 0xffffffff8102781b kernel.debug`fast_syscall_common at exception.S:578
```


We can pick an interesting frame to look at from lldb too:

```
(lldb) frame select 12
frame #12: 0xffffffff810556a3 kernel.debug`amd64_syscall [inlined] syscallenter(td=0xffff-
800174be740) at subr_syscall.c:186:11
   183             if (!sy_thr_static)
   184                 syscall_thread_exit(td, se);
   185         } else {
-> 186             error = (se->sy_call)(td, sa->args);
   187             /* Save the latest error return value. */
   188             if (__predict_false((td->td_pflags & TDP_NERRNO) != 0))
   189                 td->td_pflags &= ~TDP_NERRNO;
```

Getting the Kernel Buffer

Printing stuff and moving around the stack is most of what we need for kernel crash dump debugging. The start-up message from gdb is quite nice though, showing us the last part of the kernel message buffer as if it had come directly off the local console.

lldb doesn't yet offer a nice start-up command like that. Thankfully, "Panic!" gives us some hints as to how we might pull out this information ourselves. "Panic!" uses a macro called "msgbuf" to print the kernel message buffer from a **struct msgbuf**.

Some poking in the FreeBSD source, and we have something similar available:

```
(lldb) p *msgbufp
(msgbuf) {
    msg_ptr = 0xffff8001ffe8000 "----<<BOOT>>---\nCopyright (c) 1992-2023 The FreeBSD
Project.\nCopyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994\n\
tThe Regents of the University of California. All rights reserved.\nFreeBSD is a reg-
istered trademark of The FreeBSD Foundation.\nFreeBSD 15.0-CURRENT #0 main-272a40604:
Wed Nov 29 13:42:38 UTC 2023\n    tj@vpp:/usr/obj/usr/src/amd64.amd64/sys/GENERIC amd64\
nFreeBSD clang version 16.0.6 (https://github.com/llvm/llvm-project.git llvmorg-16.0.6-
0-g7cbf1a259152)\nWARNING: WITNESS option enabled, expect reduced performance.\nVT:
init without driver.\nCPU: 12th Gen Intel(R) Core(TM) i7-1260P (2500.00-MHz K8-class
CPU)\n  Origin="GenuineIntel"  Id=0x906a3  Family=0x6  Model=0x9a  Stepping=3\n  Fea-
tures=0x9f83fbff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE36,M-
MX,FXSR,SSE,SSE2,SS,HTT,PBE>\n  Features2=0xfeda7a17<SSE3,PCLMULQDQ,DTES64,DS_CPL,SSSE3,SD-
BG,FMA,CX16,xTPR,PCID,SSE4.1,SSE4.2,MOVBE,POPCNT,AESNI,XSAVE,OSXSAVE,AVX,F16C,RDRAND,HV>\n
AMD Features=0x2c100800<SYSCALL,"...
    msg_magic = 405602
    msg_size = 98232
    msg_wseq = 16777
    msg_rseq = 15001
    msg_cksum = 1421737
    msg_seqmod = 1571712
    msg_lastpri = -1
    msg_flags = 0
    msg_lock = {
        lock_object = {
```



```

    lo_name = 0xffffffff81230bcc "msgbuf"
    lo_flags = 196608
    lo_data = 0
    lo_witness = NULL
}
mtx_lock = 0
}
}

```

We have a struct `msgbuf` globally visible in the kernel implementing the kernel's message buffer. `lldb` shows us the start of the buffer. The fields `msg_wseq` and `msg_rseq` tell us where we have written to and where we have read from.

Reading out the unread portion of the message buffer is easy:

```

(lldb) p msgbufp->msg_ptr+msgbufp->msg_rseq
(char *) 0xffff8001ffe8a99 "panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432\ncpuid = 2\ntime = 1706644478\nKDB: stack backtrace:\ndb_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xffffe0047d2f480\nvpanic() at vpanic+0x132/frame 0xffffe0047d2f5b0\npanic() at panic+0x43/frame 0xffffe0047d2f610\ntcp_discardcb() at tcp_discardcb+0x25b/frame 0xffffe0047d2f660\ntcp_usr_detach() at tcp_usr_detach+0x51/frame 0xffffe0047d2f680\nsorele_locked() at sorele_locked+0xf7/frame 0xffffe0047d2f6b0\ntcp_close() at tcp_close+0x155/frame 0xffffe0047d2f6e0\nrack_check_data_after_close() at rack_check_data_after_close+0x8a/frame 0xffffe0047d2f720\nrack_do_fin_wait_1() at rack_do_fin_wait_1+0x141/frame 0xffffe0047d2f7a0\nrack_do_segment_nounlock() at rack_do_segment_nounlock+0x243b/frame 0xffffe0047d2f9a0\nrack_do_segment() at rack_do_segment+0xda/frame 0xffffe0047d2fa00\ntcp_input_with_port() at tcp_input_with_port+0x1157/frame 0xffffe0047d2fb50\ntcp_input() at tcp_input+0xb/frame 0xffffe0047d2fb60\nip_input() at ip_in"...

```

The output isn't formatted in a very friendly way, control characters are just printed out, but we can read out the kernel message buffer. The output is truncated before the full backtrace is available. Let's try some other commands:

```

(lldb) x/b msgbufp->msg_ptr+msgbufp->msg_rseq
0xffff8001ffe8a99: "panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432\ncpuid = 2\ntime = 1706644478\nKDB: stack backtrace:\ndb_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xffffe0047d2f480\nvpanic() at vpanic+0x132/frame 0xffffe0047d2f5b0\npanic() at panic+0x43/frame 0xffffe0047d2f610\ntcp_discardcb() at tcp_discardcb+0x25b/frame 0xffffe0047d2f660\ntcp_usr_detach() at tcp_usr_detach+0x51/frame 0xffffe0047d2f680\nsorele_locked() at sorele_locked+0xf7/frame 0xffffe0047d2f6b0\ntcp_close() at tcp_close+0x155/frame 0xffffe0047d2f6e0\nrack_check_data_after_close() at rack_check_data_after_close+0x8a/frame 0xffffe0047d2f720\nrack_do_fin_wait_1() at rack_do_fin_wait_1+0x141/frame 0xffffe0047d2f7a0\nrack_do_segment_nounlock() at rack_do_segment_nounlock+0x243b/frame 0xffffe0047d2f9a0\nrack_do_segment() at rack_do_segment+0xda/frame 0xffffe0047d2fa00\ntcp_input_with_port() at tcp_input_with_port+0x1157/frame 0xffffe0047d2fb50\ntcp_input() at tcp_input+0xb/frame 0xffffe0047d2fb60\nip_input() at ip_i"
warning: unable to find a NULL terminated string at 0xffff8001ffe8a99. Consider increasing

```


the maximum read length.

```
(lldb) x/2048b msgbufp->msg_ptr+msgbufp->msg_rseq
error: Normally, 'memory read' will not read over 1024 bytes of data.
error: Please use --force to override this restriction just once.
error: or set target.max-memory-read-size if you will often need a larger limit.
```

We hit a printing limit, and try as I might, I can't convince lldb to go further. Time for a higher tool.

Some Help From the Moon

lldb also offers a scripting interface for control, which explains why many of the commands are incredibly verbose to type out. Currently, lldb supports scripting with C++, Python and has experimental support for Lua. FreeBSD ships Lua in base, and the FreeBSD builds of lldb in 2024 include Lua support by default.

We can simply try this out with the following:

```
(lldb) script
>>> print("hello esteemed FreeBSD Journal readers!")
hello esteemed FreeBSD Journal readers!
>>> quit
```

With the >>> prompt indicating that we have moved into the Lua interpreter.

From "Panic!" we learn that adb the SunOS/Solaris debugger had a handy and easy-to-understand macro for finding and printing the message buffer:

```
msgbuf/"magic"16t"size"16t"bufx"16t"bufr"n4X
+,( *msgbuf+0t8)-*(msgbuf+0t12))&80000000$<msgbuf.wrap
.+( *msgbuf+0t12),(* (msgbuf+0t8)-*(msfbuf+0t12))/c
```

Implementing a similar mechanism with Lua should be no problem at all with that as an example.

The lldb lua interface is generated from swig bindings, this is a C++ format for describing interfaces between libraries. The Python and Lua bindings are generated the same way. For any questions you have about the API or how to use it, you can figure it out from working from the Python API documentation which is available from the lldb project. This is a very clunky way to do things, but it is possible.

I quickly get sick of running commands in the interpreter and, considering the length of some of them, they can be annoying to try. lldb can load your Lua script from a file once the interpreter has been run once. From a fresh session:

```
$ lldb --core coredump/vmcore.1 coredump/kernel-debug/kernel.debug
(lldb) target create "coredump/kernel-debug/kernel.debug" --core "coredump/vmcore.1"
Core file '/home/tj/code/scripts/gdb/coredump/vmcore.1' (x86_64) was loaded.
(lldb) script
>>> print("hello")
hello
>>> quit
(lldb) command script import ./hello.lua
hello from the script hello.lua
```


Assuming the file `hello.lua` contains:

```
print("hello from the script hello.lua")
```

The lldb Lua environment provides a `lldb` variable with members enabling access to the target, debugger, frame, process, and thread. These objects map to ones described in the Python API.

I'm not really a fan of the lldb api, it can be quite clunky to write and difficult to understand if you are having a problem with your choice of function or how variables are laid out in memory.

Once you have some experience, it gets easier to understand what it wants from you.

Let me illustrate how to use the lldb Lua bindings with an example of printing out the message buffer from a crash dump.

From the lldb Lua variable we can access files in the dump image. A big block for me when I first started doing core dump analysis was understanding how to locate things in memory. There are various kernel global variables that you can access as starting points, and most subsystems have something you can build from.

As we saw before, `msgbufp` is a global instance of the kernels message buffer. From lldb Lua we can access this with:

```
msgbuf = lldb.target:FindFirstGlobalVariable("msgbufp")
```

This gives us an instance of a SBValue representing this instance of the struct in the memory from the core dump. We can access the child members of the struct with the `GetChildMemberWithName` method and a name such as `msg_rseq`.

The `lldb.process` object gives us the ability to read out memory from our kernel dump. Sometimes it can take a bit of juggling to get together the correct references, addresses and values to perform the operations you want.

With these methods, we can assemble a point to the start of the message buffer, read it out of the core dump, and print it using Lua. I've put all of this into a script called `msgbuf.lua`:

```
msgbuf = lldb.target:FindFirstGlobalVariable("msgbufp")

msgbuf_start = msgbuf:GetChildMemberWithName("msg_rseq"):GetValue()
msgbuf_end = msgbuf:GetChildMemberWithName("msg_wseq"):GetValue()
unread_len = msgbuf_end - msgbuf_start

msgbuf_addr = msgbuf:GetChildMemberWithName("msg_ptr")
                :Dereference()
                :GetLoadAddress() + msgbuf_start
msgbuf_ptr = lldb.process:ReadMemory(msgbuf_addr, unread_len, lldb.SBError())

print("Unread portion of the kernel message buffer:")
print(msgbuf_ptr)
```

If we run this from our lldb session we get the following output:


```

(lldb) command script import ./msgbuf.lua
Unread portion of the kernel message buffer:
panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432
cpuid = 2
time = 1706644478
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe0047d2f480
vpanic() at vpanic+0x132/frame 0xfffffe0047d2f5b0
panic() at panic+0x43/frame 0xfffffe0047d2f610
tcp_discardcb() at tcp_discardcb+0x25b/frame 0xfffffe0047d2f660
tcp_usr_detach() at tcp_usr_detach+0x51/frame 0xfffffe0047d2f680
sorele_locked() at sorele_locked+0xf7/frame 0xfffffe0047d2f6b0
tcp_close() at tcp_close+0x155/frame 0xfffffe0047d2f6e0
rack_check_data_after_close() at rack_check_data_after_close+0x8a/frame 0xfffffe0047d2f720
rack_do_fin_wait_1() at rack_do_fin_wait_1+0x141/frame 0xfffffe0047d2f7a0
rack_do_segment_nounlock() at rack_do_segment_nounlock+0x243b/frame 0xfffffe0047d2f9a0
rack_do_segment() at rack_do_segment+0xda/frame 0xfffffe0047d2fa00
tcp_input_with_port() at tcp_input_with_port+0x1157/frame 0xfffffe0047d2fb50
tcp_input() at tcp_input+0xb/frame 0xfffffe0047d2fb60
ip_input() at ip_input+0x2ab/frame 0xfffffe0047d2fbc0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fc20
ether_demux() at ether_demux+0x17a/frame 0xfffffe0047d2fc50
ether_nh_input() at ether_nh_input+0x39f/frame 0xfffffe0047d2fca0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fd00
ether_input() at ether_input+0xd9/frame 0xfffffe0047d2fd60
vtnet_rxq_eof() at vtnet_rxq_eof+0x73e/frame 0xfffffe0047d2fe20
vtnet_rx_vq_process() at vtnet_rx_vq_process+0x9c/frame 0xfffffe0047d2fe60
ithread_loop() at ithread_loop+0x266/frame 0xfffffe0047d2fef0
fork_exit() at fork_exit+0x82/frame 0xfffffe0047d2ff30
fork_trampoline() at fork_trampoline+0xe/frame 0xfffffe0047d2ff30
--- trap 0, rip = 0, rsp = 0, rbp = 0 ---
KDB: enter: panic

```

Lua has kindly expanded the control characters in the buffer for us giving us nice formatted output from the message buffer.

Better Debugging Possibilities

lldb is new on the block when it comes to kernel debugging and there are still many features not available in the Lua environment, but it has enough functionality to be a useful tool. Old timer gdb users might be struggling to see the value of these examples, after all lldb adds a much more complicated syntax and it might seem like change for changes sake.

A big value that lldb and its built-in lua brings is shipping in the release FreeBSD images. lldb Lua is freely licensed and is compatible with FreeBSD, and from the start of 2024, it was enabled by default in CURRENT builds. This allows kernel developers and trouble shooters to write scripts in lldb Lua and provide them to users for analysis.

kgdb has had support for gdb script for a long time, but it isn't the most pleasant scripting language to program. Lua, on the other hand, while a little weird, is commonly used in

many environments and is part of the FreeBSD boot loader. I have written a tool to extract TCP log files from crashed kernel images--the major headaches were figuring out how to get the memory. Once I had the data, creating and writing these to files was an easy job.

Kernel dumps have everything in them and can contain sensitive information. A reasonable scripting language makes it possible for developers to provide scripts to extract further debugging information from a kernel image without the need to move around large core dumps, and without needing to handle the worries of trusting a stranger with possibly sensitive information.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.



The FreeBSD Project is looking for

- Programmers • Testers
- Researchers • Tech writers
- Anyone who wants to get involved

Find out more by

Checking out our website

freebsd.org/projects/newbies.html

Downloading the Software

freebsd.org/where.html

We're a welcoming community looking for people like you to help continue developing this robust operating system. Join us!

Already involved?

Don't forget to check out the latest grant opportunities at freebsd.foundation.org

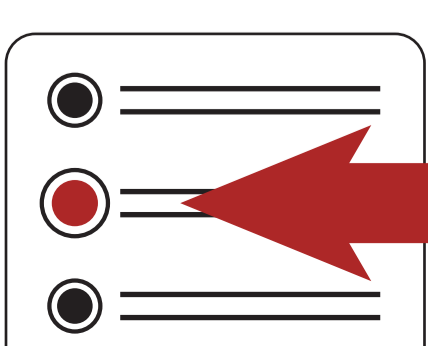
Help Create the Future. Join the FreeBSD Project!

FreeBSD is internationally recognized as an innovative leader in providing a high-performance, secure, and stable operating system.

Not only is FreeBSD easy to install, but it runs a huge number of applications, offers powerful solutions, and cutting edge features. The best part? It's FREE of charge and comes with full source code.

Did you know that working with a mature, open source project is an excellent way to gain new skills, network with other professionals, and differentiate yourself in a competitive job market? Don't miss this opportunity to work with a diverse and committed community bringing about a better world powered by FreeBSD.

The FreeBSD Community is proudly supported by



ZFS Images From Scratch, or `makefs -t zfs`

BY MARK JOHNSTON

For a long time, the FreeBSD project has made virtual machine (VM) disk images available on its download site: just go to <https://download.freebsd.org/snapshots/VM-IMAGES> to find a selection of pre-built images for download. These come in a variety of formats recognized by common hypervisors such as QEMU, VirtualBox, VMWare and bhyve. The FreeBSD project similarly distributes images for various cloud platforms, such as EC2, Azure and GCP. As a FreeBSD user, all you need to do is select the image and create an instance, and within a few seconds a fully installed FreeBSD system will be available.

For many users, the pre-built images are sufficient, but you might have special requirements that are not met by those images. In particular, until recently, all of the FreeBSD project's official images used UFS for the root filesystem. Of course, it was still possible to use ZFS in a VM by one of several strategies:

1. Keep the root filesystem on UFS but add extra disks and use them to back a ZFS pool.
2. Boot the FreeBSD installation media in a VM and use it to install FreeBSD on a virtual disk with ZFS as the root filesystem. The resulting VM image can be cloned and used as a template for other images.
3. Manually create an image by setting up a `md(4)` disk and then creating and importing a ZFS pool on top of that disk, into which FreeBSD can be installed. `poudriere image` currently works this way, for example.

While these strategies work, they all come with caveats:

- **option 1)** makes it difficult to use boot environments;
- **option 2)** requires manual effort to create and customize the template image;
- **option 3)** requires root privileges and cannot be done at all from within a jail (currently ZFS pool creation is forbidden in jails).

For a long time, I had wanted to build ZFS-based VM images locally so that I could run the FreeBSD regression test suite on both UFS and ZFS, so in 2022, I started looking at how difficult it would be to extend the tools we already use to make UFS images (i.e., `makefs(8)`) to support ZFS in some form.

`makefs(8)`

The FreeBSD project builds official VM images using `makefs(8)`, a utility originating from NetBSD. It takes one or more paths as input and generates a single file containing a filesystem image populated with the contents of those paths.

`makefs` supports several filesystems, including UFS, FAT and ISO9660. The basic idea behind its use here is to install FreeBSD to a temporary directory (e.g., with `make install-world`), and then point `makefs` at that directory. The result is a file containing a filesystem image whose root directory contains the FreeBSD installation.

For users familiar with building FreeBSD from source, the following example might provide a clearer picture:

```
# make installworld installkernel distribution DESTDIR=/tmp/foo
# makefs -t ffs fs.img /tmp/foo
# mdconfig -f fs.img
md0
# mount /dev/md0 /mnt
# ls /mnt/bin/sh
/mnt/bin/sh
```

This installs a pre-built copy of a FreeBSD distribution to `/tmp/foo` and then uses `makefs` to generate a filesystem image in `fs.img`. This image can be mounted by using `mdconfig(8)` to create a character device backed by the file. Attributes of the files in `/tmp/foo`, such as mode bits and timestamps, are preserved in the resulting image.

In contrast, a traditional “live” installation of FreeBSD might look more like this:

```
# truncate -s 50g fs.img
# mdconfig -f fs.img
md0
# newfs /dev/md0
/dev/md0: 51200.0MB (104857600 sectors) ...
# mount /dev/md0 /mnt
# cd /usr/src
# make installworld installkernel distribution DESTDIR=/mnt
# umount /mnt
```

Here, we use the `newfs(8)` utility to initialize an empty filesystem on the target device, then copy files onto it. While this works, of course, it has downsides similar to the problems with creating ZFS pools that I mentioned earlier: `newfs(8)` requires root privileges, and the resulting image is not reproducible. That is, if two images are created this way from the same pre-built FreeBSD distribution, they will not be byte-for-byte identical, for example, because the file access and modification times will be slightly different between the two images. Reproducibility is an important security property of build systems since it makes it easier to detect malicious tampering of build outputs.

I was already familiar with `makefs` from writing scripts to create VM images for my own use. As mentioned, I had wanted to be able to similarly build ZFS images, and I was not alone; a common user complaint was that all of FreeBSD’s official cloud images were UFS-based, even though ZFS is a very popular choice for the root filesystem on FreeBSD. So, I spent some time thinking about how `makefs`-generated ZFS images might look.

makefs(8) Meets ZFS

So what does `makefs` actually do? A technical answer to this question requires some knowledge of filesystem internals, but briefly: `makefs` initializes some global filesystem metadata, such as a UFS/FFS superblock, and then traverses the input directory tree(s), copying their contents into the image and adding metadata, such as directory entries, which point to file data. Whereas one traditionally starts with an empty filesystem and then asks the kernel to add data to it via utilities such as `cp(1)`, `makefs` generates a populated filesystem

tem in a single operation. So while this means that **makefs** needs to know about how a filesystem's data and metadata is arranged on disk, it can be considerably simpler than the kernel's implementation of the filesystem.

makefs never needs to look up a file by name, handle out-of-space conditions, perform buffer caching, or delete files, for example.

ZFS is large and complex, but per the observation above, a hypothetical **makefs -t zfs** can ignore a lot of the details. This was important for me: I was and am currently not a ZFS expert, and at the time, had little understanding of its on-disk format, so simplicity was the name of the game. At this point we can ask: what exactly should **makefs -t zfs** do?

My goal was to support creation of VM images with ZFS as the root filesystem. More specifically, **makefs** would need to:

1. Create a ZFS pool that has a single disk vdev. There is no need to support RAID or mirroring layouts, since for a VM image the extra redundancy is not very useful.
2. Create at least one dataset in the pool. The dataset needs to be mountable as the root filesystem. In practice, a FreeBSD-on-ZFS installation comes with a dozen or so datasets pre-created, but an initial proof-of-concept can ignore this for simplicity's sake.
3. Populate the dataset with the contents of the input directory trees. More specifically, for each input file, **makefs** needs to allocate a **dnode** and copy the file into the image somewhere. It also needs to copy attributes, such as file permissions, of the input files.

In particular, quite a few ZFS features which affect on-disk layout can simply be ignored. There is no need for **makefs** to bother with compression or snapshots, for example. So while the task still seemed somewhat daunting, by excluding all but the minimum necessary features, it seemed quite doable.

Attempt #1: **libzpool**

As a FreeBSD kernel developer I already had some experience with OpenZFS internals, but ZFS is a complex beast. The code is partitioned into quite a few different subsystems, most of which have no knowledge of how data actually gets laid out on disk, and I had no experience with the ones that do. However, it turns out that one can compile the OpenZFS kernel module into a userspace library: **libzpool.so**. This is used primarily for testing the OpenZFS code itself, but seemed like an excellent starting point for my project: **libzpool.so** knows all about the ZFS on-disk layout, so I thought I could avoid learning too much about it and instead write code that used high-level operations, similar to how commands like **zpool create** simply ask the kernel to create a pool on a set of **vdevs**.

Without going into too much detail, this approach ended up yielding a working prototype, but turned out to be a dead end. A few of the reasons:

- **libzpool.so** is really not suitable for "production" applications: it has no stable interface, and my prototype was effectively making use of undocumented kernel APIs. If I were to press on with this approach, the result would be fragile and difficult to maintain.
- The code in **libzpool.so** is mostly unmodified kernel code, and thus creates lots of

ZFS is large and complex, but a hypothetical `makefs -t zfs` can ignore a lot of the details

threads and caches file data in the ARC, all of which is unnecessary for **makefs**'s purposes. A consequence of this is that the prototype was very slow and would consume gobs of system memory, sometimes triggering the out-of-memory killer.

- The result was not reproducible. If I ran the prototype twice with identical inputs, the output would not be byte-for-byte identical.

While I had to throw away most of the prototype, writing it was a useful learning experience and helped motivate me to try a different approach.

At this point, it seemed I would have to get my hands dirty and learn about the ZFS on-disk layout. I realized that the FreeBSD boot loader would have a similar problem: in order to boot FreeBSD from a ZFS pool, the loader needs to be able to find the kernel file and load it into memory. The boot loader runs in a constrained environment and thus cannot use the kernel's ZFS code, so clearly other people had already solved similar problems.

Attempt #2: ZFS From Scratch

Fortunately, there is a [ZFS on-disk layout specification](#) floating around the Internet; it is rather incomplete and outdated, but it was much better than nothing. On top of that, I had the boot loader code to look at. In some sense it solves the inverse problem that **makefs** does: it just opens and reads data from a ZFS pool without writing anything, whereas **makefs** creates a new pool but does not need to be able to read existing pools.

The duality with the boot loader was very useful: I could write code to create a pool, and then test it by trying to use the boot loader to read a file (the kernel) from the pool. More specifically, I would first install a FreeBSD kernel to a temporary directory:

```
$ cd /usr/src
$ make buildkernel
$ make installkernel DESTDIR=/tmp/test -DNO_ROOT
```

Then I can create a ZFS image and try to load it using the legacy bhyve loader:

```
$ makefs -t zfs -o poolname=zroot zfs.img /tmp/test
$ sudo bhyveload -c stdio -d zfs.img test
```

Here, **bhyveload** is using **/boot/userboot.so**, which is a copy of the FreeBSD boot loader that is compiled to run in userspace. It has most of the functionality of the real boot loader, but rather than using, say, BIOS calls or EFI boot services to read data from disk, it uses the familiar **read(2)** system call to fetch data from the image file, **zfs.img**.

The initial goal was to get **userboot.so** to find and load the kernel located at **/boot/kernel/kernel** in **zfs.img**. This was a very convenient test harness since I could easily attach a debugger to **bhyveload** or add print statements to the loader and recompile **userboot.so**. My first milestone was to get [vdev_probe\(\)](#) to recognize the image as a valid ZFS pool.

vdev Labels and the uberblock

vdev_probe() looks at a disk to see if it belongs to a ZFS pool; that is, it determines whether the disk appears to be a **vdev**, and if so, starts loading more metadata:

```
/*
 * Ok, we are happy with the pool so far. Lets find
 * the best uberblock and then we can actually access
```



```
* the contents of the pool.
*/
vdev_uberblock_load(vdev, spa->spa_uberblock);
```

Chapter 1 of the ZFS on-disk specification describes `vdev` labels and uberblocks in a good amount of detail. The summary is that a `vdev` contains a block of metadata, the `vdev` label, which contains metadata describing the pool to which the `vdev` belongs, as well as copies of the “uberblock”, which points to the root of the `vdev`’s metadata tree. So, in order to get `userboot.so` to find my pool, I wrote [code](#) which adds `vdev` labels to the output image file.

At this point `makefs` was already making use of ZFS-specific data structures, such as `vdev_label_t` and `uberblock_t`. Rather than duplicating the definitions used by the boot loader, `makefs` shares with it a [large header](#) that contains many useful on-disk data structure definitions.

Object Sets and the MOS

Once the loader was able to probe and recognize `makefs`-generated images, the next step was to get it to mount a dataset from within the image. The loader code that handles this is mostly contained in [zfs_get_root\(\)](#).

To understand the implementation of `zfs_get_root()`, it is worth reading chapter three of the ZFS on-disk specification, which describes object sets. While the specification quickly gets into the gory details, it is worth reviewing the high-level structures that are used to represent data in ZFS.

ZFS has “block pointers”, which really just refer to the physical location of a block of data on a `vdev` (from `makefs`’s perspective, this is just an offset into the output image file). A ZFS metadata object, of which there are several dozen types, is represented by a 512-byte “dnode”. A dnode contains various bits of metadata about the object, such as its type and size, and may also contain block pointers referring to additional data. For example, a file stored in a ZFS dataset is represented by a dnode (of type `DMU_OT_PLAIN_FILE_CONTENTS`), much like an inode in a traditional Unix filesystem. Finally, an “object set” is a structure which contains an array of dnodes; a dnode is uniquely identified by the object set to which it belongs and its index in the array.

A ZAP (ZFS Attribute Processor) is a dnode which contains a set of key-value pairs. ZAPs are used to represent many higher-level ZFS metadata structures. For example, a Unix directory is represented by a ZAP whose keys are filenames and values are dnode IDs for the corresponding files.

The MOS (meta object set) is the root object set of the pool. The uberblock contains a pointer to the MOS, and from the MOS it is possible to reach all other metadata (and thus, data) in the pool. With this information, it is a bit easier to understand `zfs_get_root()`: it takes the dnode with ID 1 (which it expects to be a ZAP object), uses it to find a ZAP object containing pool properties and looks up the value of the “bootfs” property, which is used to find the dnode of the root dataset.



The next step was to get it to mount a dataset from within the image.

When creating a pool, `makefs` allocates and begins populating the MOS in `pool_init()`. Once `userboot.so` was able to process the MOS, it became possible to import a `makefs`-generated pool, at which point I started using `zdb(8)` to inspect the generated pool. `zdb`'s command-line usage is rather obscure, but simple invocations like

```
# zdb -dddd zroot 1
```

which dumps dnode 1 from the MOS, were very useful for figuring out what OpenZFS expects to see when importing a pool. For example, when dumping a ZAP object, `zdb` can print all of the key-value pairs in the ZAP. Many configuration ZAP keys have values which are dnode IDs, so `zdb` can easily be used to inspect different "layers" of the pool and dataset configuration.

Datasets and Files

ZFS datasets have names and are organized into a tree. The root dataset is named after the pool itself (e.g., "zroot"), and names of child datasets are prefixed by the parent's name. While my initial prototype of ZFS support for `makefs` automatically placed all files in the root dataset, this was not sufficient to be able to create root-on-ZFS VM images:

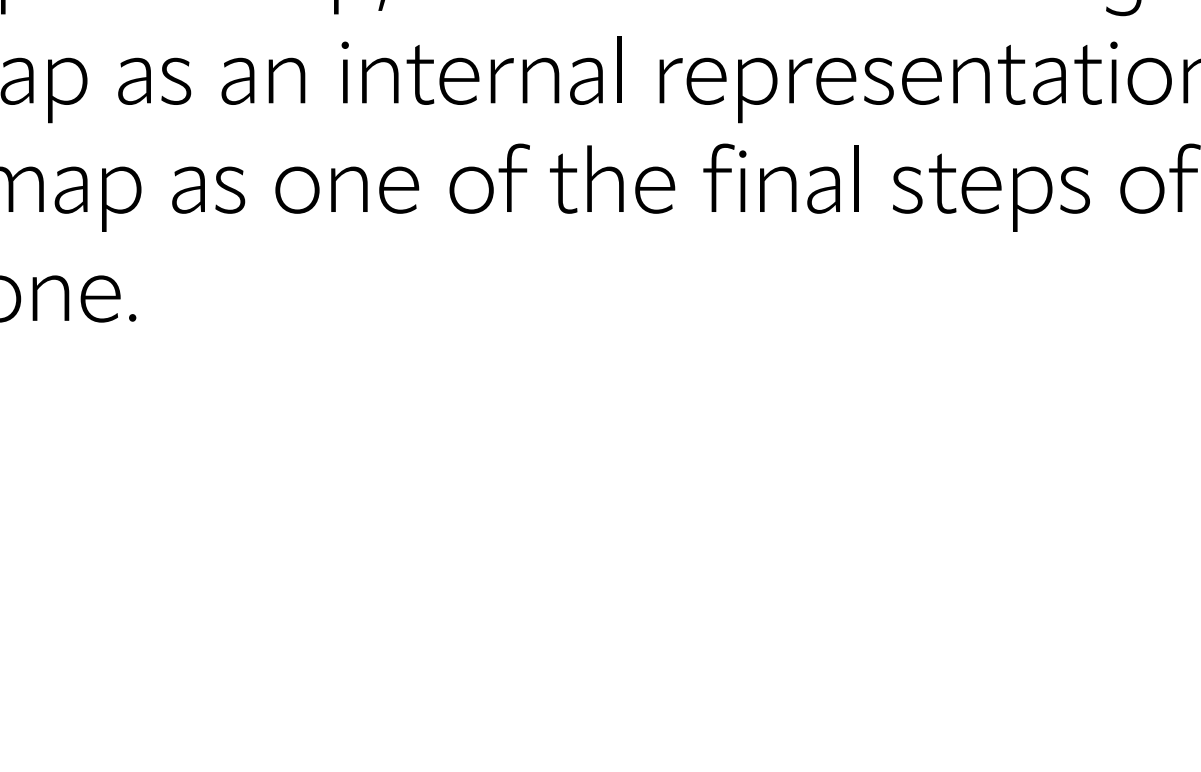
`bsdinstall` and other FreeBSD installers automatically create a number of child datasets. Some, such as `zroot/var`, are never mounted but only exist to provide settings which are inherited by child datasets, such as `zroot/var/log`. My goal was for `makefs` to be able to create a tree of datasets which matches the layout provided by `bsdinstall`.

The release image-building script [demonstrates](#) the syntax for creating multiple datasets. Each dataset is described by a `-o fs` option which contains the dataset name and a semicolon-separated list of properties. Only a small number of properties - as described in the `zfsprops(8)` manual page — are currently supported.

When `makefs -t zfs` finishes initializing various structures, it begins to [process](#) the input directory trees. Each input file is represented by a `fsnode` structure, and these structures are organized into a tree which represents the file tree. First, `makefs` determines which `fsnode` corresponds to the root of each mounted dataset. Then, it traverses the tree of `fsnodes`, allocating a dnode for each file; this happens in the context of a dataset which determines the object set from which the dnode is allocated.

To [copy a regular file](#) `makefs` allocates a dnode from the current object set and, in a loop, allocates blocks of space from the output file, and copies data from the input file into the allocations. ZFS supports power-of-2 block sizes ranging from 4KB to 128KB, so smaller files do not create excessive internal fragmentation. All allocated blocks in the image are tracked using a bitmap which is updated by the `vdev_space_alloc()` function.

Allocated space tracked by the bitmap must be recorded in the output image; ZFS uses a central data structure, the "space map," to track which regions of a vdev are currently allocated. `makefs` uses the bitmap as an internal representation of all block allocations, and uses it to generate the space map as one of the final steps of image generation, once all block allocations have been done.



The root dataset is named after the pool itself and names of child datasets are prefixed by the parent's name.

Conclusion

Adding ZFS support to `makefs` took a fair bit of effort but ultimately resulted in an implementation that I believe will be useful to many FreeBSD users, while avoiding a large maintenance burden. There is roughly 2,600 lines of ZFS-specific code in `makefs` (out of 15,000 lines in total), which is reasonably small. There is also a [regression test suite](#) which provides a good amount of coverage.

Of those 2,600 lines, over 100 are calls to `assert()` and so simply verify invariants. These assertions were very useful during development, since a lot of code was written in an incomplete manner just to get the boot loader working, and fleshed out more fully later on; they served to document the limitations of various functions and helped catch many bugs as I added more and more functionality.

Now that FreeBSD 14.0 has shipped and root-on-ZFS VM images are available, I hope that many users are taking advantage of this new feature. A number of bugs were found and fixed during the release cycle, so at least some users have been trying it out. Currently there are no enhancements planned for `makefs -t zfs` but this may change in response to feedback — please submit a bug report if you see any room for improvement.

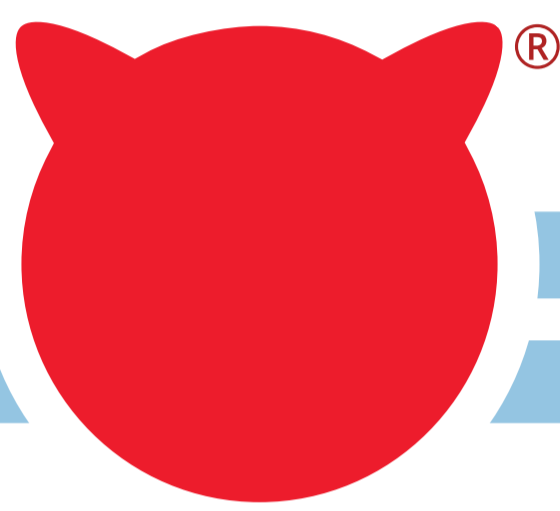
MARK JOHNSTON is a software developer and FreeBSD src developer living in Toronto, Ontario, Canada. When not sitting in front of a computer he enjoys playing in a city dodgeball league with friends.

Write For Us!

Contact Jim Maurer
with your article ideas.
(maurer.jim@gmail.com)



Support FreeBSD[®]



Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsd.foundation.org/donate



Enhance Your Git Experience

BY BENEDICT REUSCHLING

Version control has been around for a long time. Having a way to keep a certain version of a file in a state where it is retrieved is the start of our journey. After all, that's what backup software is for. Some people use version control systems (VCS) that way, but that only scratches the surface. The valuable parts of VCS is to be able to review what has changed in the file over its history. When using it with other people to keep important project files in sync with everyone else, features like blaming, diffing, branching and merging become even more relevant. These features make collaboration on anything from text, source code, configuration files, and anything worth sharing within a group possible in the first place. Most big projects that have a large set of those files under version control would simply be too tedious to maintain without sophisticated tooling around it. Version control systems have come and gone, both from commercial and open source vendors. Many of them implement a basic set of functionality that has been established by the people using it, and introducing a whole new set of terminology and complicated tooling is often seen as a hindrance to adoption.

Each code change in the long history of the FreeBSD project has had a descriptive text, called the commit message, attached to it. What may seem like a nuisance for some is actually helpful when looking for problems that are rooted in the past. Why was a change made 15 years ago? Who made that change, what other files were touched by the same commit, and what are the actual lines of code that were affected? These are all questions that not only the diffs that the VCS produces can relate, but also the commit message is a valuable help in understanding what is going on in the code today. This often bridges the past to the future since Unix systems have been around for a while and are running on hardware that people who committed the changes in the past could not possibly conceive.

With FreeBSD being so old, preserving that rich history is important. Particularly when there is a need to switch to a different version control system. Over time, FreeBSD has used CVS, then migrated to Subversion, and now uses git. (I'm fairly sure there was also RCS involved in the early days.) Each time such a migration took place, one primary goal was to preserve every single change made, along with the commit message.

Each code change
in the long history
of the FreeBSD project
has had a descriptive
text attached to it.

Switching version control systems is sometimes necessary because a vendor goes out of business or something better comes along that people find more appealing than the current solution. The best is the enemy of the good. Git has become the de-facto standard for version control these days, even if it has some rough edges and a bit of a learning curve. In my view, both as a user/consumer of the FreeBSD project as well as a committer, it took a while to get familiar again and to make git a tool that works for me and not the other way around. I may still not totally grasp all of the inner workings, but luckily, I do not have to. Developers working on the src and ports trees also use a lot more of the features like branching, merging, tagging, cherry-picking, rebasing, and others that are not required in the documentation repository. Still, sometimes a typo fix in a ports description or a man page requires people like me to also get at least a basic understanding of these concepts to do the right thing.

It's also a question of how often something is used. If I use a feature only twice per year, I may need to look it up again because the last time I did was a while ago. The basics of cloning a fresh tree, pulling updates, making changes, committing, and pushing them are all too familiar once you've done it regularly. I compare it to basic vi usage. I can open a document, make changes, save, and exit just fine. But the other features that a powerful editor like vi and friends provides may stay hidden from me forever. Same with git: it has a lot more under the hood which users do not know about or even need. More often than not though, it makes life easier—even for the basic cases—to have a bit of knowledge and configuration around these advanced features. And this holds not only for a developer, but also for a user.

FreeBSD 14 deprecated the portsnap utility which a lot of people used to get a new or updated version of the ports tree. Instruction in the handbook and other places now directs people to check out the ports tree directly from git. The same is true when a user needs a copy of the src tree, because they need to recompile the kernel module for VirtualBox. These are all good use cases, but there is a catch: both the src and ports trees take a long time to clone because of the long and rich history described above. Let's see if we can add some configuration and tools that speeds up the process. We'll also learn about other neat features that git provides, both for Joe random user as well as Jane developer.

Faster Cloning

First, install git on FreeBSD by running `pkg install git`. Since we do not have a ports tree yet and there is no more portsnap, that can turn into a chicken and egg problem on a system that does not have direct access to the FreeBSD mirrors. A poudriere machine in your local network may be the solution here, or download the file on a different machine, copy over the binary, and use `pkg install ./<packagename>` to install it.

Note that the git port itself contains a good amount of configurable options. Since we're starting out, we ignore that for now and revisit some of these when we feel more experienced. If you discover something useful, then consider blogging about it or write an article for the *FreeBSD Journal*. After all, why do I have to do all the writing work?

If I use a feature
only twice per year,
I may need to look
it up again.

Once git is available, we can return to our use case above: downloading a copy of the ports tree. *The FreeBSD Handbook* Chapter 4 tells us that we can either get the HEAD branch (the latest and greatest) or be a bit less on the bleeding edge and get the quarterly branch. Whichever we chose, we get presented with the all too familiar `git clone` command. That's all fine and good, but it takes a long time to finish downloading all those files and directories. Let's look at the quarterly branch at the time of writing this article. I'll use the `time(1)` command to measure the download time.

```
$ time git clone https://git.FreeBSD.org/ports.git -b 2024Q1 /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 6125935, done.
remote: Counting objects: 100% (960/960), done.
remote: Compressing objects: 100% (142/142), done.
Receiving objects: 100% (6125935/6125935), 1.20 GiB | 36.28 MiB/s, done.
remote: Total 6125935 (delta 925), reused 833 (delta 818), pack-reused 6124975
Resolving deltas: 100% (3700108/3700108), done.
Updating files: 100% (158490/158490), done.
git clone https://git.FreeBSD.org/ports.git /usr/ports
0.00s user 0.03s system 0% cpu 3:34.48 total
```

Bandwidth aside, this 3:34 is too long for me. We can do better than this. Since we do not need all the history and just the latest version of files, a shallow clone using `--depth=1` is much faster.

```
time git clone --depth=1 https://git.FreeBSD.org/ports.git /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 194509, done.
remote: Counting objects: 100% (194509/194509), done.
remote: Compressing objects: 100% (182218/182218), done.
remote: Total 194509 (delta 11904), reused 120301 (delta 5787), pack-reused 0
Receiving objects: 100% (194509/194509), 85.40 MiB | 10.48 MiB/s, done.
Resolving deltas: 100% (11904/11904), done.
Updating files: 100% (158490/158490), done.
git clone --depth=1 https://git.FreeBSD.org/ports.git /usr/ports
0.01s user 0.01s system 0% cpu 28.709 total
```

Much faster indeed (29s) and I get exactly what I want. What if I need the full history because I'm working on a bug? Then I can use a filter function to first get the whole commit history, but not the history. The latter may come as a separate step. I'm looking to reduce the time to download, so let's try this:

```
time git clone --filter=blob:none https://git.FreeBSD.org/ports.git /usr/ports
Cloning into '/usr/ports'...
remote: Enumerating objects: 3706789, done.
remote: Counting objects: 100% (794/794), done.
remote: Compressing objects: 100% (82/82), done.
remote: Total 3706789 (delta 771), reused 721 (delta 712), pack-reused 3705995
Receiving objects: 100% (3706789/3706789), 704.87 MiB | 48.79 MiB/s, done.
Resolving deltas: 100% (2043361/2043361), done.
remote: Enumerating objects: 152073, done.
remote: Counting objects: 100% (63494/63494), done.
remote: Compressing objects: 100% (61224/61224), done.
```

```
remote: Total 152073 (delta 7810), reused 2276 (delta 2270), pack-reused 88579
Receiving objects: 100% (152073/152073), 78.98 MiB | 10.93 MiB/s, done.
Resolving deltas: 100% (11301/11301), done.
Updating files: 100% (158490/158490), done.
git clone --filter=blob:none https://git.FreeBSD.org/ports.git /usr/ports
0.00s user 0.03s system 0% cpu 1:51.29 total
```

Git divided the work into two parts: first, all blobs (think files here) get filtered out and fetches history at first. In the second step, the blobs followed. This was faster than a full clone, but slower than the shallow copy. There was also a difference in the retrieved sizes, which contributed to the speedup. In the regular clone, we received 1.20 GB. The two-step process of the blobless clone let git receive 704.87 MB of history followed by 78.98 MB. This benefit comes with a drawback though: when I have found the bug and I want to know when this was introduced, the `git blame` operation needs to fetch those revisions from the server first. If I'm on the road without network access, I'm out of luck. The full clone could give me the information, as it has all the history already retrieved. Again, for non-developers interested in getting the files themselves, this does not matter much and the benefit is a better download time.

Scaling Up

Imagine you were working on the man pages, which reside in the src tree. Downloading the whole kernel, userland, tools, and everything in between is a lot for the initial clone. What if you only occasionally work on those man pages? Surely there are changes made by others, which we need to be aware of. Wouldn't it be nice if our system would fetch those changes for us, so that our local copy does not drift too far away from the top of the tree? The scalar tool that is part of git solves it: fast downloads of a big repository and retrieving changes from upstream in regular intervals. This puts the local clone into maintenance mode, which is a fancy word for this functionality. Here is how to use it: replace `git` with `scalar`, the rest of the command is identical.

```
time scalar clone https://git.FreeBSD.org/src.git /usr/src
Initialized empty Git repository in /usr/src/src/src/.git/
remote: Enumerating objects: 2386494, done.
remote: Counting objects: 100% (258756/258756), done.
remote: Compressing objects: 100% (16493/16493), done.
remote: Total 2386494 (delta 253705), reused 244654 (delta 242263), pack-reused 2127738
warning: fetch normally indicates which branches had a forced update,
but that check has been disabled; to re-enable, use '--show-forced-updates'
flag or run 'git config fetch.showForcedUpdates true'
warning: fetch normally indicates which branches had a forced update,
but that check has been disabled; to re-enable, use '--show-forced-updates'
flag or run 'git config fetch.showForcedUpdates true'
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 20 (delta 0), reused 0 (delta 0), pack-reused 3
Receiving objects: 100% (20/20), 196.11 KiB | 16.34 MiB/s, done.
warning: fetch normally indicates which branches had a forced update,
but that check has been disabled; to re-enable, use '--show-forced-updates'
flag or run 'git config fetch.showForcedUpdates true'
```



```
branch 'main' set up to track 'origin/main'.
Switched to a new branch 'main'
Your branch is up to date with 'origin/main'.
crontab: no crontab for root
scalar clone https://git.FreeBSD.org/src.git
0.01s user 0.00s system 0% cpu 31.971 total
```

Ignore those warnings for now, the process finishes nonetheless. There is something about `crontab(1)` here, responsible for fetching updates in regular intervals. To convert an existing repository to use `scalar`, no need to clone it again: run **`scalar register`** in the root of your repository and it will convert the local copy to use it. Neat! The `scalar` command will set up a crontab entry. If you do not have a user-specific crontab (like I have here for the root user), then run **`crontab -e`** to set it up. If all went well, git adds an entry for `scalar` to run:

```
# BEGIN GIT MAINTENANCE SCHEDULE
# The following schedule was created by Git
# Any edits made in this region might be
# replaced in the future by a Git command.

29 1-23 * * * "/usr/local/libexec/git-core/git" --exec-path="/usr/local/libexec/git-
core" for-each-repo --config=maintenance.repo maintenance run --schedule=hourly
29 0 * * 1-6 "/usr/local/libexec/git-core/git" --exec-path="/usr/local/libexec/git-core"
for-each-repo --config=maintenance.repo maintenance run --schedule=daily
29 0 * * 0 "/usr/local/libexec/git-core/git" --exec-path="/usr/local/libexec/git-core"
for-each-repo --config=maintenance.repo maintenance run --schedule=weekly
# END GIT MAINTENANCE SCHEDULE
```

Adjust those entries to your own needs or leave them as they are. If the machine has a proper mail setup, you'll receive messages containing the fetched revisions when the cron-jobs have run. Another thing that **`scalar clone`** and the associated maintenance jobs do is add an entry to your git configuration file, aptly named **`.gitconfig`**.

```
[scalar]
repo = /usr/src
[maintenance]
repo = /usr/src
```

This brings us right into git's configuration file.

Create Your (commit) History Everyday

Chances are that you are working on multiple repositories over time: one for work, another for a private project, and contributing to your favorite open source project. The configuration for those cloned repositories may be different. For example, you may use your corporate email to identify yourself in your commits, which may not be appropriate or even allowed when committing to a private project. As such, we can have a project-specific **`.git/config`** as part of the repo and a global one that applies to any and all repositories you're working on.

The global **`.gitconfig`** is in your home directory. You can either directly edit that file (if you know what you are doing) or use git to manage the contents of the file and set proper values. The latter uses this syntax:

```
git config --global NAME VALUE
```

For example, to register my name for commits, I run:

```
git config --global user.name Benedict Reuschling
```

This results in an entry like this in `.gitconfig`:

```
[user]
  name = Benedict Reuschling
```

You can see that there are categories in brackets for entries like `user` (email is under there and you better set it, too). Others are **commit**, **diff**, or **branch**.

Changing Commit Behavior

Torches and pitchforks aside, I do not like to use nano to write my commit messages. To define your own editor, execute this command:

```
git config --global core.editor nvim
```

You almost always want to set this as a global option. Having this in a project `.git/config` and committing it will cause productivity to fall sharply as your other contributors will start another holy editor war resulting in a lot of changes in the repo with nothing but trying to change it to their own personal favorite. "Well done", is what you will remember as the last words of your sarcastic boss as he closes the company door behind you forever on the same day.

There are other configuration settings that (more positively) affect your commit experience. There are too many to list here and the defaults are fine. Fiddling with some options can change your git experience somewhat and may remove some personal annoyances (see below). See `git-config(1)` for details.

How about being a bit more sophisticated and defining aliases for common but tedious to type commands? That's where the alias section comes in handy. I have these defined:

```
[alias]
  last = last -1 HEAD
  lg = log --graph --all --pretty=format:'%Cred%h -%C(yellow)%d%Creset %s %Cgreen(%ci)
%C(bold blue)<%an (%ae)>'
```

Similarly, when looking at the log, I'd like to see at least these fields: commit, the author, the date the author made the change, plus the commit and its date. This is achieved by looking at `git-log(1)` and figuring out that the option is called **fuller**:

```
git config --global format.pretty fuller
```

In my repository, I can run `git last` to run the equivalent of `git last -1 HEAD`. Getting totally fancy with colors and all, I like the command `git lg` even more when thinking about how few keystrokes it requires now. Try it out for yourself and thank me later.

When I do the actual commit, I want to see what gets committed. Git displays the diff between the head revision and my own changes below the text area for the commit message with this setting:

```
git config --global commit.verbose true
```

Your Signature, Please

Speaking of commits, why don't you sign your commits? "Well, the GPG/PGP setup is too complicated" may be an answer. There is a solution for that: use SSH instead. On services such as GitHub or your corporate (or private) GitLab instance, you have already uploaded a public key to pull repositories over SSH. Signing your commits with the same key gives your changes some extra credit. This is often honored with a "signed" icon or label next to the commit on those platforms. The setup is so easy, I wonder why this is not the default by now. Here's how:

```
git config --global gpg.format ssh
git config --global user.signingKey 'ssh-ed25519 AAAAC3(...)34rve user@host'
```

It's debatable if you want to use the same SSH key everywhere. If not, remove the `--global` option from the last line above and make that change for each repo with its own key.

Commits can now go like this:

```
git commit -S
```

To always sign, make it the default:

```
git config --global commit.gpgsign true
```

But what is this? When running `git show --show-signature` does not show our signature, but displays an error message instead. Not cool! Good for us, the message also tells us what to change: `gpg.ssh.allowedSignersFile` is the option we need to change.

Git complains because SSH does not build a web of trust that signs keys by others. Instead, we need to tell git which keys we are trusting. A separate file contains all trusted SSH signatures. Since we are orderly people, let's put this file in `~/.config/git/allowed_signers` (create the paths if they don't exist by now).

The content of `allowed_signers` is as follows:

```
email ssh-ed25519 ssh_public_key comment
```

Keen eyes will recognize it as the same format that `ssh-keygen(1)` uses. We need to at least trust our own SSH key, so put it there. Repeat this for all the other people in your circle who contribute to the repo and sign their commits, too. To teach git about this file, add yet another configuration option (the one the error message complained about earlier):

```
git config --global gpg.ssh.allowedSignersFile "~/.config/git/allowed_signers"
```

Retry the `git show --show-signature` command (and create an alias for it) to see the error message replaced by the git signature.

Fixing Minor Annoyances

To update your local copy, it's suggested that you run `git pull --ff-only`, which is the default behavior. If you keep forgetting to add the parameter, then set it as the default pull behavior like this:

```
git config --global pull.ff only
```

Of course, you could create an alias for it. This is one of those "fire-and-forget" settings you do not need to revisit in the future.

When looking at diffs, I always wondered why git uses a/ and b/ to distinguish the files from each other. I do not need those, the filenames speak for themselves. I found that disabling this behavior is possible with this option:

```
git config --global diff.noprefix true
```

Speaking of diffs, I would like to see at least 5 lines of context around my changes. That is a personal preference, but anyone can set it to their liking with this option:

```
git config --global diff.context 5
```

Working on an international project like FreeBSD has taught me that there are multiple ways to write a date. The default display that git uses is **Fri Mar 01 12:34:56 2024**. All fine with that, but I'm used to the following way: **2024-03-01 12:34:56**. This option sets it exactly how I like it:

```
git config --global log.date iso
```

Another thing that I found odd was the order in which git lists branches when running **git branch**. I would like to have the branch with the most recent commit at the top and not some other (random?) order. To change this, my **.gitconfig** contains this:

```
git config --global branch.sort -committerdate
```

Now I see exactly which branch received the most recent changes. Time to merge!

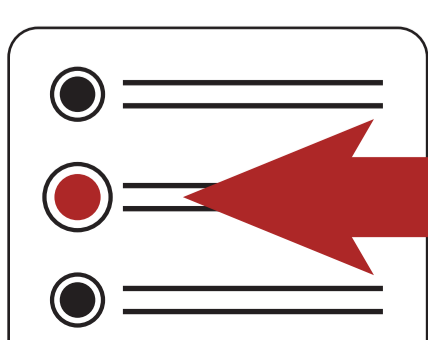
Conclusion

My configuration will probably grow over time as I discover other useful options. Git is flexible in its configuration. The defaults are fine for most people and changes are easy to make. This article should get you started writing your own config and ideally reduce some of the teeth grinding involved when working with git.

References:

<https://blog.gitbutler.com/git-tips-and-tricks/>
<https://jvns.ca/blog/2024/02/16/popular-git-config-options/>
<https://blog.dbrgn.ch/2021/11/16/git-ssh-signatures/>

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.



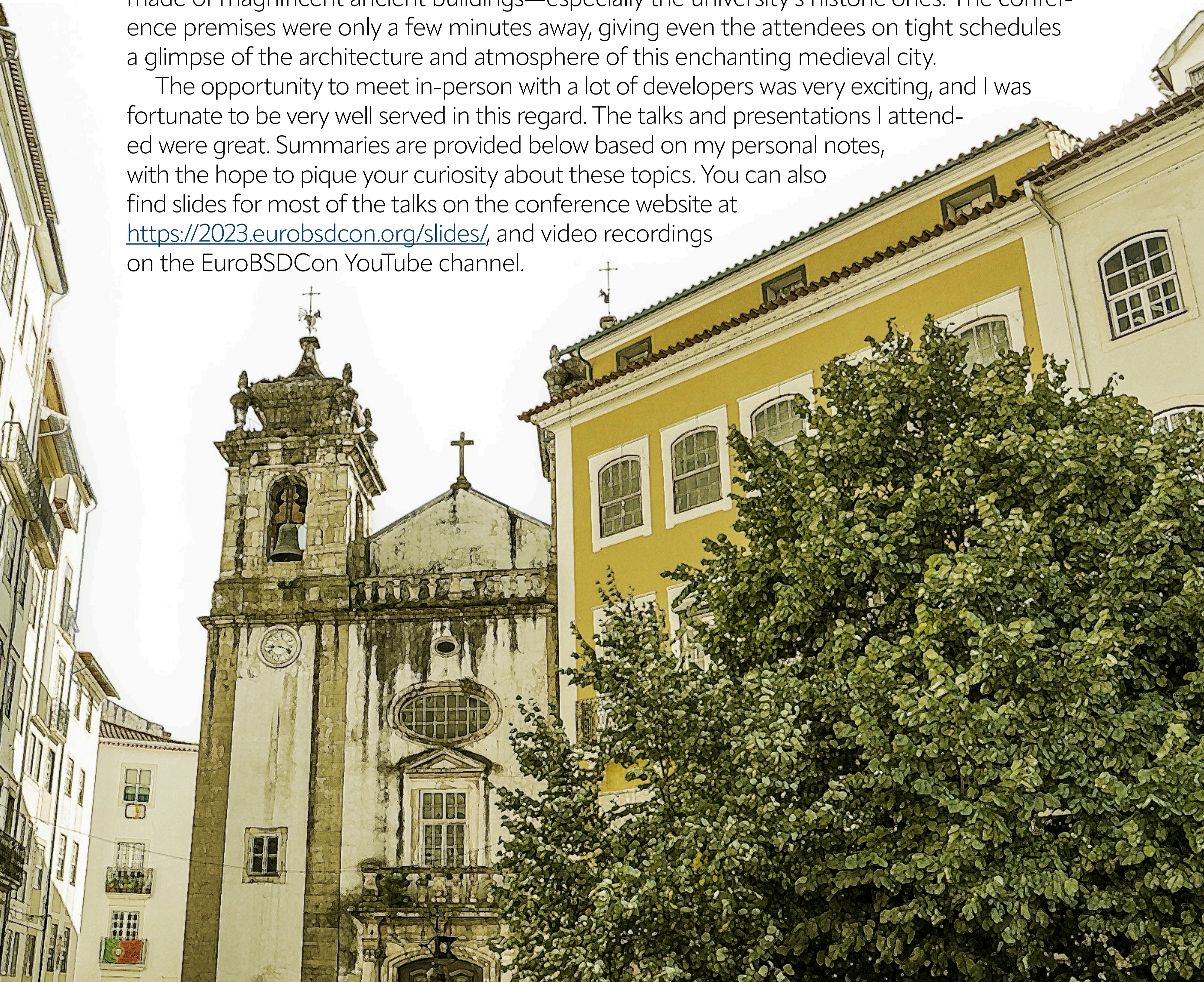
EuroBSDCon 2023

BY OLIVIER CERTNER

Last September, I had the pleasure of traveling to the city of Coimbra, Portugal, where the annual European technical conference on BSD systems, EuroBSDCon, took place.

Although I had already been to Portugal a few times, in the Douro valley near Porto, in Porto itself, or in Lisboa, I had never had the opportunity to learn about Coimbra. The city was an ancient capital of the Kingdom of Portugal, almost a thousand years ago, and, today, is a major cultural center and a vibrant city renowned for its university, one of the oldest in the world. Its region is the third most populated of Portugal, and the city has one of the highest per capita incomes in the country. Hilly, downtown Coimbra, called “Baixa”, is made of magnificent ancient buildings—especially the university’s historic ones. The conference premises were only a few minutes away, giving even the attendees on tight schedules a glimpse of the architecture and atmosphere of this enchanting medieval city.

The opportunity to meet in-person with a lot of developers was very exciting, and I was fortunate to be very well served in this regard. The talks and presentations I attended were great. Summaries are provided below based on my personal notes, with the hope to pique your curiosity about these topics. You can also find slides for most of the talks on the conference website at <https://2023.eurobsdcon.org/slides/>, and video recordings on the EuroBSDCon YouTube channel.



FreeBSD's Developer Summit

Thanks to Joseph Mingrone from the FreeBSD Foundation, as a newly contracted general developer, I was invited to attend the developer summit on Thursday and Friday. At the introductory session, approximately 30 persons presented themselves and their connection to the project. I already knew almost half of them by name and had seen a few of them on videos or conference calls but hadn't met a single one in person. Surprisingly, I don't recall having seen or met any of them at EuroBSDCon 2017 in Paris, my only other BSD conference so far. Speaking of which, let me write a belated, but big thank you to Jean-Sébastien Pédrón for taking time to introduce me to various people at that earlier conference.

A significant contingent from the FreeBSD Foundation was in attendance, as well as delegations from Netflix and Beckhoff, but also several freelance developers, faculties, and other professionals not working on FreeBSD by day, but managing to be quite active in the community, such as Guido van Rooij, one of the old-timers and a member of the EuroBSD-Con Foundation, whom I would like to thank for his warm welcome.

The first session was a report from the FreeBSD Foundation and its various on-going projects, such as improving the installer (Pierre Pronchéry), Wi-Fi support (Bjoern Zeeb, En-Wei Wu), RISC V support (Mitchell Horne), bhyve (John Baldwin, Mark Johnston), fixing pressing security issues (John Baldwin), re-implementing libc's string functions with SIMD (Robert Clausecker), implementing SU+J snapshots (Kirk McKusick), improving CI and fixing ports (Moin Rahman), making FreeBSD a Cloud Init platform (Mina Galić), importing OpenSSL 3 in base (Pierre Pronchéry), fixing `rtprio(2)` and rationalizing scheduling priorities, both for applications and internally (by yours truly; still a work in progress as of this writing). I honestly couldn't keep track of all projects, so let me apologize for anything I missed and advise the interested reader to browse the Foundation's blog (at <https://freebsd.foundation.org/blog>) and the developer summit's wiki page (<https://wiki.freebsd.org/DevSummit/202309>) for more information.

This was followed by a keynote talk by Justin Gibbs on imagining the next 30 years (since, as you must know, FreeBSD turned 30 last year). His main message, through recollections of his own involvement with the project, both technically and in establishing the Foundation, was to foster a "trying in order to be successful" attitude and to ask everyone: "What would you do if you weren't afraid?" This prompted an exchange among participants, with a lot of ideas thrown on the table, including:

- Become the OS for smartphones, IoT, any mobile computing.
- Be the best operating system to run AI workloads.
- Become a large group that can help each other grow (mentoring, teaching).
- Have people under 20 with limited budget install FreeBSD on inexpensive hardware.
- A point-and-click jail deployment system to use instead of Docker.
- Spread the culture of upstreaming, convince involved businesses that it is in their best

**"What would you do
if you weren't afraid?"
—Justin Gibbs**

interest, have them allow their experienced developers to engage more with the community.

- CI, automatic regression testing. In general, many more tests.

This synthetic list is my own, so I apologize if I've left out or misunderstood some points.

Sergio Carlavilla Delgado then presented the website for a new project he is working on and encouraged people to give feedback. A more mature front page was distributed around a month after the event through mailing lists, and it was quite exciting to see the progress made and the gap between it and our existing website.

Greg Wallace, the Foundation's Director of Partnerships & Research, organized a SWOT (Strengths, Weaknesses, Opportunities, Threats) Session, with again lots of ideas exchanged. Listing all of them here would be too long, and Greg has produced a document with all the output. It can be read at <https://wiki.freebsd.org/DevSummit/202309> (scroll to find the "community SWOT" line, referencing a PDF attachment).

We were then greeted with a surprise talk by Jordan Hubbard who, if I recall correctly, was "by chance" spending a few days near Coimba (are we supposed to really believe that?). I remember it as a great depiction of project history, but also as a focused talk on what currently matters from his perspective (he's working with NVIDIA), and what he thinks FreeBSD would be wise to embrace. I only have a few notes that probably won't do it justice, but here's the gist. AI is now "the" market, just as networking and mobile were before. Anything of significance in AI cannot run on a single GPU, nor on a single node for that matter. Consequently, speed of communication is king. Direct communication between CPUs and GPUs, or CPUs and DPUs, must be supported by the kernel. UCX (see openucx.org) is the successor to Infiniband verbs. Additionally, inference in AI will happen at the edges.

I began the first evening by chatting with Ruslan Bukin, first about his initial RISC-V port for FreeBSD, then shifting to various non-technical subjects such as special, mostly low-fat, diets, or French engineering's high quality when it comes to bicycle parts. In the process, I found myself invited to a Netflix dinner with Jonathan Looney, Warner Losh, Gleb Smirnoff, John Baldwin (and Ruslan of course), where I spent most of the time listening to the conversations while enjoying delicious seafood. I don't know how Ruslan managed it, but thanks a lot to him for dragging me in.

The next day started with a talk by Bojan Novkovic on an experimental kernel benchmarking framework called "kbench". Its goal is to facilitate kernel testing in various ways by standardizing the flow for several use cases, helping check reproducibility and catch performance regressions. It could also be a good vehicle for research projects. Its Python scripts fetch, build, and run benchmarks from a particular, pre-assembled set. Bojan's short-term goals are to expand the benchmark pool and add tracking for more metrics (`dtrace(1)`, `libxo(3)`-based utilities, `pmc(3)`). A longer-term step would be to ease collected data's post-processing. The framework can be found at <https://github.com/bnovkov/kbench>.

"AI is now 'the' market,
just as networking
and mobile were before."
— Jordan Hubbard

Loosely connected to the previous talk was Ruslan Bukin's presentation of his Hardware Trace Framework (**hwt(9)**). It relies on dedicated hardware support built into CPUs, namely Processor Trace (PT) for Intel, and the CoreSight and Statistical Profiling Extensions (SPE) technologies from ARM. The framework's kernel side weights 3,5k lines, with scheduler and **mmap()** hooks, **ioctl**-based trace context management, code to support the **/dev/hwt** devices and multiple backends. **hwt(1)** can be used to choose the mode of operation, configure address range filtering and perform process management and symbol lookup. CoreSight is the oldest technology from ARM and it offers only a single stream per CPU. The backend for it can be found in-tree under **sys/arm64/coresight** and is fully functional. Some code snippets to support Intel PT are available. ARM Ltd. is currently working on the SPE backend.

Changing subject to ports and packages, Michael Reim presented "Ravenports", the latest take on a new ports system started by John Marino in 2017. Michael is using it as he is working for a small hosting company in Germany that started on FreeBSD but is mostly Linux today. Its core features are a high concurrency build system with a high level of automation as necessary for a small maintainer team, support for sub packages and variants (corresponding somewhat to FreeBSD's flavor it seems), multi-platform support (all BSDs, Linux, Solaris, currently on hold, macOS dropped), self-contained (including toolchain, GCC-based, currently 13.2), with binary bootstrap, and with very up-to-date software versions. Some drawbacks compared to FreeBSD's ports are that it isn't currently portable to niche or obsolete ISAs (which I suspect may have to do with the non-availability of a GNAT-based Ada compiler), is only lightly tested, and has a much lower port count.

In the evening, when I contacted Joe about his plans, he responded by inviting me to a follow-up dinner to a core team and Foundation meeting, where I could chat with Deb Goodkin, Greg Wallace, Li Wen-Hsu and Ed Maste from the Foundation, and core team's Mateusz Piotrowski, all of whom were all very friendly and welcoming.

The Conference

Henning Brauer, EuroBSDCon Foudation's CTO (read: Chief Trolling Officer) opened the conference with humor, kindness, and practical information on how to get beverages and how to locate the different rooms and zones of interest. He introduced a keynote by Paula Alexandra Silva on "Facilitating Change: Embracing Gender Diversity in Computer Science".

As the first technical talk, I attended John Baldwin's NVMe over Fabrics (NVMe-oF) in the FreeBSD project. The aim of this technology is to permit the use of the NVMe interface (greater parallelism, lower latency) with devices that are not physically connected to the requesting computer, but are rather accessed over TCP, RDMA or Fibre Channel. For a reason I don't recall, I unfortunately missed the outline. Trying to make sense of my notes, I built the following short summary, to be taken with a grain of salt perhaps. FreeBSD's implementation is a 3-layer design, in the middle is the transport abstraction that handles capsules (data buffers). Thanks to that, you can just allocate a queue pair and never deal with transport specifics. The user space library, **libnvmf**, is designed for simplicity and debuggability (not thread-safe, blocking I/O on sockets). It provides a TCP implementation. There are also userspace implementations of part of the functionality of both a host (**nvmfdd**, doing I/O on a single namespace on a remote controller) and a controller (multiple namespaces supported, backed either by files, character devices or memory buffers). The kernel datapath mirrors this transport abstraction but uses asynchronous callbacks instead of blocking for

performance. **nvmf(4)** is the in-kernel Fabrics host. It creates the `/dev/nvmeX` devices (so **nvmecontrol(8)** works), and supports disk access through CAM (`/dev/ndaX`). Future work is to add an in-kernel Controller, implement support for RDMA and Fibre Channel, and TLS protection for TCP queue pairs. Current code can be seen at <https://github.com/bsdjhb/freebsd.git>, branch **nvmf2**. This work was sponsored by Chelsio.

I then went to Kristof Provost's talk on **if_ovpn(4)** and was pleased to discover that, even if Wireguard seems to be all the rage now, some people are working on improving OpenVPN's performance. **if_ovpn(4)** is a clean-room, in-kernel, OpenVPN client implementation which supports only a subset of functionalities: Only the AES-GCM and ChaCha ciphers are available (the others are old), it doesn't support layer 2 networking (to keep the kernel interface simple) and is UDP-only (much more work is required for TCP). Userspace handles the control channel, while the kernel handles the data channel. They both share a single socket, whose file descriptor is passed to the kernel during connection setup. Kernel passes up to userland unknown (i.e., control) packets. Userland drives the kernel through **ioctl(2)** and **nvlists(9)** for extensibility (where Linux uses netlink, which at time of this work had yet to be integrated into FreeBSD). Key rotation is a two-phase process: New keys are declared and later switched to, without traffic disruption. `vnet` helped a lot for testing (see `/usr/tests/sys/net/if_ovpn`). Performance of the Data Channel Offload (DCO) by **if_ovpn(4)** was tested on a Netgate 4100. DCO with QAT (Intel's Quick Assist Technology) offload reaches 1Gbit/s, DCO with AES-NI crypto ~750Mbit/s and DCO with software crypto ~210Mbit/s, to be compared with ~210Mbit/s with OpenVPN on **if_tun** but using AES-NI for crypto. Development was sponsored by Netgate.

At the lunch break, we gathered outside, in a kind of loggia because of the weather, for a family photo, a great pretext for a lot of chatting and an unexpected attraction: A unique T-shirt parodying the sleeve of a famous album by AC/DC, with the text "UNIX, Highway to Shell", worn by EuroBSDCon Foundation's Katie McMillan. Chatting and technical discussions then continued at the university's restaurant.

The afternoon session began for me with Hiroki Sato's talk on USB DbC (Debug Capability). Serial consoles make debugging firmware or early boot possible, but there are no serial ports on modern hardware, only legacy interfaces such as BMC (Baseboard Management Controller) on server machines. USB replaces all these legacy interfaces. However, it doesn't allow a direct connection between two hosts, a tiered star topology is required. Thus, USB debugging works by changing one port of the host to debug (the "target host") into a USB device managed by the debugging host. Comparable debugging technologies include IEEE 1394 (FireWire; legacy), which supports point-to-point and physical access to memory (see **dcons(4)**), but also USB 2.0 (see EHCI specification) although it requires a special repeater hardware. On the debugging host, a normal USB 3 stack is enough. On the target host, the new **udbc(4)** experimental driver manages the port turned into a device for simple serial

I was pleased to discover that, even if Wireguard seems to be all the rage now, some people are working on improving OpenVPN's performance.

communication. It doesn't need a full USB stack, as it only has to manage the TRB (Transfer Request Block) ring buffers of two USB pipes, and DbC is designed as a simple transport for more sophisticated debug protocols (such as JTAG and Intel DCI). Physically, an A-to-A USB 3.0 cross-cable is required, and, of all ports of the root hub, only one will convert to behave as a USB device to the debugging host. Code and an install image can be found at <https://people.allbsd.org/~hrs/FreeBSD/udbc/20230915>. After enough compatibility feedback, patches will be submitted. Nothing in this work is x86-specific, and porting to other BSDs should be easy. **udbc(4)** could be extended to mimic other types of USB devices (such as, possibly, a mass storage device).

Next, I attended Warner Losh's talk on booting FreeBSD with LinuxBoot, using **loader.kboot**. I'm not going into many details here since the presentation was dense and you can find them all in the public slides. LinuxBoot was started in 2017 by Google (as NERF) to provide a unified booting environment and to get rid of UEFI (mostly). With it, a low-level boot loader initializes a machine just enough to launch Linux, which then runs scripts to determine what to finally boot via **kexec()**. There is a very nice community around it. The low-level boot loader can be UEFI (Pre-EFI Interface), coreboot romstage, U-boot SPL, or the Slim bootloader. This talk only considers UEFI and its EDK2 implementation, which LinuxBoot strips down to only PEI (Pre-EFI Interface; initializes low-level details of the machine, such as memory, clocks, etc.) and the Runtime Services. By using LinuxBoot, you only have to write device drivers once—for Linux and not UEFI DXE—which brings several advantages: Faster time-to-market, often faster boot times, more security hardened (100k contributors versus only 100 to 200 for Grub/EDK2). FreeBSD needed to be ported to LinuxBoot for several reasons. On amd64 and aarch64, the kernel expects some metadata such as memory maps to boot, and some are provided by the system firmware (BIOS, UEFI) which the kernel can't necessarily access. These data are usually prepared by our **loader(8)**, so the first step was to port it as a Linux binary and add "stand" devices to access host resources from it. Also, Linux already sets the virtual address mapping, via UEFI's **SetVirtualAddressMapping()**, which can be called only once. This required changing FreeBSD's kernel to support a non-trivial virtual to physical address mapping (and there was also a problem with the GICv3 interrupt controller on aarch64). With this work, FreeBSD is the only non-Linux, non-GRUB-assisted fully booting UEFI OS under LinuxBoot on x86 and aarch64.

I finished the talks of the first day with Toshihan Bharvani's talk on Running FreeBSD on OpenPOWER, which is an exciting architecture that permits machines booting with completely open firmware. Porting efforts on POWER are shifting towards little endianness because a lot of desktop applications (such as Firefox, JS) can't run properly on a big-endian architecture. The open firmware includes OPAL (Open Power Abstraction Loader) implemented by skiboot, which launches Linux running the petitboot bootloader, but also some open BMC implementations such as LibreBMC or OpenBMC. Current developers are Alfredo Dal'Ava Junior, Leandro Lupori, Andre Fernando da Silva, among others. There is still a lot of work to do, so please volunteer! Current hardware is still expensive, with ~5k€ for an IBM AC922 Developer Machine or a Talos II entry-level developer system. "Soft" hardware, with FPGA-based Microwatt and LibreSOC, is also available. The OpenPOWER HUB initiative (<https://openpowerfoundation.org/hub>) provides access to several types of OpenPOWER hardware. The next generation hardware should come much better priced, first with entry-level PowerISA 3.1 Racks (start of 2024) to be followed by Workstations (expected price around \$1000 to \$1500), single-board computers such as PowerPi! Embedded Single Board

Computer in several versions and generations, an enablement platform for developers (maybe around \$500), microcontroller systems, and FPGA-based devices. So, please help emPOWER BSDs!

The day concluded with a beautiful social event. It took place not far from the historic center, on the other side of the Mondego River, in a restaurant with a huge room to accommodate us all. I don't remember how it happened, but I soon found myself sitting at a table between John Baldwin and Mark Johnston. When these two giants engage in technical conversations, it is always a pleasure to listen to them. We enjoyed the evening, with lots of occasions for informal chats around a great buffet.

The most spectacular moment was an unforgettable demonstration of "Fado de Coimbra" by a group of men coated with the traditional black academic suit. The emotional power their rendition conveyed is simply impossible to describe in text.

The second and last day of the conference started for me with Eirik Øverby speaking about FreeBSD at Modirum, how they use it (and other open-source software), and what makes the community so good for them. The diverse war stories involving jails, MySQL on ZFS, `SO_REUSEPORT_LB`, `epairs` and `relayd` were captivating, and Eirik made this journey very lively. His slides are available on the conference website.

I was very interested in Christos Margiolis' presentation of his work on Arbitrary Instruction Tracing with DTrace. DTrace is a powerful live debugging tool I'm still not using nearly enough, and Christos has augmented it with a new provider called `kinst` ("kernel instructions"). The base FDT provider (Function Boundary Tracing) can only trace entry and return points of a kernel function that hasn't been inlined. `kinst` provides probes to trace all instructions in a function, a specific instruction, or the entry or return points of inlined functions. The selected probe information is passed from `dtrace(1)` to `libdtrace` and then `kinst` using `/dev/dtrace/kinst`. `kinst` then disassembles the function and creates the requested probes by overwriting target instructions with a breakpoint instruction. On execution, the breakpoint handler calls `dtrace_invop()` which calls `kinst_invop()`. The instructions replaced by breakpoints nonetheless have to be executed for the execution to stay correct. Since emulation is tedious and error prone, these instructions are copied into a trampoline where execution is transferred to manually. The main difficulty with this approach is that RIP/PC-relative instructions still either have to be re-encoded (amd64) or emulated (arm64, riscv). Most of the hard work for inline function tracing is done by `libtrace` (using ELF and DWARF information). If a probe designates a function that was not inlined, `kinst` defers it to FBT to avoid code duplication. Each kernel module has to be checked for an inline function, which is currently painfully slow.

As the afternoon session was beginning, I attended the `gunion(8)` talk by Kirk McKusick. Kirk has the knack of presenting his ideas in clear and didactic fashion, as you can experience when watching videos of his presentations on YouTube. After starting with a refresher on the GEOM framework, he presented the `gunion(8)` utility, which tracks changes to a read-only disk on a writeable disk. This is similar to what `unionfs(5)` does with files and di-

The day concluded with a beautiful social event.

rectories, but at the lower level of blocks on a block device. The upper disk (the writeable one) must be at least the size of the lower disk (the read-only one). Union metadata exists only as long as the union exists (no persistency at the moment). **gunion(8)** offers **create** and **destroy** commands, as well as **revert** (discard the changes stored in the upper disk) and **commit** (write back the upper disk's changes in the lower disk). Uses of **gunion(8)** include trying to fix disks with broken filesystems without the risk of destroying them more and without the need for backing them up first: If repairing fails, just use **revert**, else you can **commit** and use the original disk. **gunion(8)** is also useful to implement poor-man's clones. **gnop(8)** was instrumental in testing error recovery, delay and out-of-order I/O handling during the implementation of **gunion(8)**.

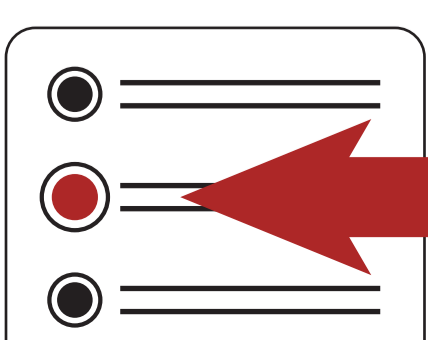
For the last talk of my day, I stayed in the "Auditorio", as I had all day, to watch Michael Dexter's presentation about "the FreeBSD Appliance". He started by stating a definition of a software appliance inspired by Wikipedia's: Just enough OS ("JeOS") to run an application on commodity hardware. The guidelines of the presentation were then twofold. First, it shows how FreeBSD is well suited to the task (integrated OpenZFS, jails, bhyve), and with some advantages over illumos or Linux. "JeOS" may become a more practical reality with packaged base, but it's already possible to rule out almost everything through build options and produce a working OS that boots in seconds, "OccamBSD" (see github.com/michael-dexter/occambsd). Then, custom images based on "OccamBSD" and working on bare metal or under hypervisors can be produced. Second, it showcases some of the new features introduced in FreeBSD 14.0, a way to stay up to date with recent developments.

I had initially hoped to attend Mateusz Piotrowski's ZFS Directory Scaling talk but changed plans for a meetup with FreeBSD Foundation's Ed Maste to discuss my current projects and see how to push the latest commits.

That evening, since I was leaving Monday morning, I was planning to go out and contacted Mateusz Piotrowski about his plans. I was directed to a tapas bar, where I met with David Cottlehuber, and a bunch of other nice guys with whom we finally had dinner, including Christos Margiolis, Luca Pizzamiglio, Mohamed, and Peter. I finished the evening in an Irish bar with Peter and some others up to a time I have not been able to remember. Fortunately, my departure train was not too early in the morning...

These were lively and fulfilling four days spent meeting a lot of people and hearing about great technical projects. That I'm now looking forward to the next BSD conferences probably won't surprise you. If you've never attended and have the opportunity, well, come to get a taste of it and I don't think you will regret it!

OLIVIER CERTNER stumbled onto FreeBSD in 2004 as the result of a search for a more serious alternative to Linux, after a cataclysmic system crash destroyed his main ext3 HDD. He has been using it ever since everywhere he can and has maintained small changes throughout the system privately for around 15 years. He has recently engaged with the community and open-source development, and is currently working on a revamp of scheduling priorities (which gave birth to a paper at AsiaBSDCon 2024), has started analyzing weird OOM behaviors, and plans to work on some deep VFS problems in the near future, including a redesign of unionfs. He has been sponsored by the FreeBSD Foundation for most of his public work.



2024 Events Calendar

BSD Events taking place through September 2024

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.



May 2024 FreeBSD Developer Summit

May 29-30, 2024

Ottawa, Canada

<https://freebsd.foundation.org/news-and-events/event-calendar/may-2024-freebsd-developer-summit/>

Join us for the May 2024 FreeBSD Developer Summit, co-located with BSDCan 2024, which will take place in Ottawa, Canada. The two-day event takes place May 29-30, 2023, and will consist of developer discussion sessions, vendor talks, and working groups.



BSDCan 2024
29 May-1 June, Ottawa, Canada

BSDCan 2024

May 29 - June 1, 2024

Ottawa, Canada

<https://www.bsdcan.org/2024/>

BSDCan is a technical conference for people working on and with BSD operating systems and related projects. It is a developers conference focusing on emerging technologies, research projects, and works in progress. It also features Userland infrastructure projects and invites contributions from both free software developers and those from commercial vendors.



EuroBSDCon 2024

September 19-22, 2024

Dublin, Ireland

<https://2024.eurobsdcon.org/>

EuroBSDCon is the International annual technical conference held in a different European country each year. It focuses on gathering users and developers working on and with 4.4BSD (Berkeley Software Distribution) based operating systems family and related projects. The FreeBSD Foundation is pleased to again be a Silver Sponsor.

